

Related Documentation

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

On This Page

- [Notes About Output Arrays for Functions.](#)
- [Array Based Study Functions That Do Not Use the Index Parameter.](#)
- [Return Object of Array Based Study Functions.](#)
- [Working with Intermediate Study Calculation Functions.](#)
- [Cumulative Calculations with Intermediate Study Functions.](#)
- [sc.AdaptiveMovAvg\(\).](#)
- [sc.AddACSChartShortcutMenuItem\(\).](#)
- [sc.AddACSChartShortcutMenuSeparator\(\).](#)
- [sc.AddAlertLine\(\).](#)
- [sc.AddAlertLineWithDateTime\(\).](#)
- [sc.AddAndManageSingleTextDrawingForStudy\(\).](#)
- [sc.AddAndManageSingleTextUserDrawnDrawingForStudy\(\).](#)
- [sc.AddDateToExclude\(\).](#)
- [sc.AddElements\(\).](#)
- [sc.AddLineUntilFutureIntersection\(\).](#)
- [sc.AddLineUntilFutureIntersectionEx\(\).](#)
- [sc.AddMessageToLog\(\).](#)
- [sc.AddStudyToChart\(\).](#)
- [sc.AdjustDateTimeToGMT\(\).](#)
- [sc.ADX\(\).](#)
- [sc.ADXR\(\).](#)
- [sc.AlertWithMessage\(\).](#)
- [sc.AllocateMemory\(\).](#)
- [sc.AngleInDegreesToSlope\(\).](#)
- [sc.ApplyStudyCollection\(\).](#)
- [sc.ArmsEMV\(\).](#)
- [sc.ArnaudLegouxMovingAverage\(\).](#)
- [sc.AroonIndicator\(\).](#)
- [sc.ArrayValueAtNthOccurrence\(\).](#)
- [sc.ATR\(\).](#)
- [sc.AwesomeOscillator\(\).](#)

- [sc.BarIndexToRelativeHorizontalCoordinate\(\).](#)
- [sc.BarIndexToXPixelCoordinate\(\).](#)
- [sc.BollingerBands\(\).](#)
- [sc.Butterworth2Pole\(\).](#)
- [sc.Butterworth3Pole\(\).](#)
- [sc.CalculateAngle\(\).](#)
- [sc.CalculateLogLogRegressionStatistics\(\).](#)
- [sc.CalculateOHLCAverages\(\).](#)
- [sc.CalculateRegressionStatistics\(\).](#)
- [sc.CalculateTimeSpanAcrossChartBars\(\).](#)
- [sc.CalculateTimeSpanAcrossChartBarsInChart\(\).](#)
- [sc.CancelAllOrders\(\).](#)
- [sc.CancelOrder\(\).](#)
- [sc.CCI\(\).](#)
- [sc.ChaikinMoneyFlow\(\).](#)
- [sc.ChangeACSCChartShortcutMenuItemText\(\).](#)
- [sc.ChangeChartReplaySpeed\(\).](#)
- [sc.ChartDrawingExists\(\).](#)
- [sc.ChartsDownloadingHistoricalData\(\).](#)
- [sc.ClearAlertSoundQueue\(\).](#)
- [sc.ClearAllPersistentData\(\).](#)
- [sc.ClearCurrentTradedBidAskVolume\(\).](#)
- [sc.ClearCurrentTradedBidAskVolumeAllSymbols\(\).](#)
- [sc.ClearRecentBidAskVolume\(\).](#)
- [sc.ClearRecentBidAskVolumeAllSymbols\(\).](#)
- [sc.CloseChart\(\).](#)
- [sc.CloseChartbook\(\).](#)
- [sc.CloseFile\(\).](#)
- [sc.CombinedForegroundBackgroundColorRef\(\).](#)
- [sc.ConvertCurrencyValueToCommonCurrency\(\).](#)
- [sc.ConvertDateTimeFromChartTimeZone\(\).](#)
- [sc.ConvertDateTimeToChartTimeZone\(\).](#)
- [sc.CreateDoublePrecisionPrice\(\).](#)
- [sc.CreateProfitLossDisplayString\(\).](#)
- [sc.CrossOver\(\).](#)
- [sc.CumulativeDeltaTicks\(\).](#)
- [sc.CumulativeDeltaTickVolume\(\).](#)
- [sc.CumulativeDeltaVolume\(\).](#)
- [sc.CumulativeSummation\(\).](#)
- [sc.CyberCycle\(\).](#)
- [sc.DataTradeServiceName\(\).](#)
- [sc.DatesToExcludeClear\(\).](#)
- [sc.DateStringDDMMYYYYToSCDateTime\(\).](#)
- [sc.DateStringToSCDateTime\(\).](#)
- [sc.DateTimeToString\(\).](#)
- [sc.DateToString\(\).](#)

- [sc.DeleteACSDrawing\(\)](#).
- [sc.DeleteLineUntilFutureIntersection\(\)](#).
- [sc.DeleteUserDrawnACSDrawing\(\)](#).
- [sc.Demarker\(\)](#).
- [sc.Dispersion\(\)](#).
- [sc.DMI\(\)](#).
- [sc.DMIDiff\(\)](#).
- [sc.DominantCyclePeriod\(\)](#).
- [sc.DominantCyclePhase\(\)](#).
- [sc.DoubleStochastic\(\)](#).
- [sc.EnvelopeFixed\(\)](#).
- [sc.EnvelopePct\(\)](#).
- [sc.Ergodic\(\)](#).
- [sc.EvaluateAlertConditionFormulaAsBoolean\(\)](#).
- [sc.EvaluateGivenAlertConditionFormulaAsBoolean\(\)](#).
- [sc.EvaluateGivenAlertConditionFormulaAsDouble\(\)](#).
- [sc.ExponentialMovAvg\(\)](#).
- [sc.ExponentialRegressionIndicator\(\)](#).
- [sc.FillSubgraphElementsWithLinearValuesBetweenBeginEndValues\(\)](#).
- [sc.FlattenAndCancelAllOrders\(\)](#).
- [sc.FlattenPosition\(\)](#).
- [sc.FormatDateTime\(\)](#).
- [sc.FormatDateTimeMS\(\)](#).
- [sc.FormatGraphValue\(\)](#).
- [sc.FormatString\(\)](#).
- [sc.FormattedEvaluate\(\)](#).
- [sc.FormattedEvaluateUsingDoubles\(\)](#).
- [sc.FormatVolumeValue\(\)](#).
- [sc.FourBarSymmetricalFIRFilter\(\)](#).
- [sc.FreeMemory\(\)](#).
- [sc.GetACSDrawingByIndex\(\)](#).
- [sc.GetACSDrawingByLineNumber\(\)](#).
- [sc.GetACSDrawingsCount\(\)](#).
- [sc.GetAskMarketDepthEntryAtLevel\(\)](#).
- [sc.GetAskMarketDepthEntryAtLevelForSymbol\(\)](#).
- [sc.GetAskMarketDepthNumberOfLevels\(\)](#).
- [sc.GetAskMarketDepthNumberOfLevelsForSymbol\(\)](#).
- [sc.GetAskMarketDepthStackPullSum\(\)](#).
- [sc.GetAskMarketDepthStackPullValueAtPrice\(\)](#).
- [sc.GetAskMarketLimitOrdersForPrice\(\)](#).
- [sc.GetAttachedOrderIDsForParentOrder\(\)](#).
- [sc.GetBarHasClosedStatus\(\)](#).
- [sc.GetBarPeriodParameters\(\)](#).
- [sc.GetBarsSinceLastTradeOrderEntry\(\)](#).
- [sc.GetBarsSinceLastTradeOrderExit\(\)](#).
- [sc.GetBasicSymbolData\(\)](#).

- [sc.GetBasicSymbolDataWithDepthSupport\(\).](#)
- [sc.GetBidMarketDepthEntryAtLevel\(\).](#)
- [sc.GetBidMarketDepthEntryAtLevelForSymbol\(\).](#)
- [sc.GetBidMarketDepthNumberOfLevels\(\).](#)
- [sc.GetBidMarketDepthNumberOfLevelsForSymbol\(\).](#)
- [sc.GetBidMarketDepthStackPullSum\(\).](#)
- [sc.GetBidMarketDepthStackPullValueAtPrice\(\).](#)
- [sc.GetBidMarketLimitOrdersForPrice\(\).](#)
- [sc.GetBuiltInStudyName\(\).](#)
- [sc.GetCalculationStartIndexForStudy\(\).](#)
- [sc.GetChartArray\(\).](#)
- [sc.GetChartBaseData\(\).](#)
- [sc.GetChartDateTimeArray\(\).](#)
- [sc.GetChartDrawing\(\).](#)
- [sc.GetChartFontProperties\(\).](#)
- [sc.GetChartName\(\).](#)
- [sc.GetChartReplaySpeed\(\).](#)
- [sc.GetChartStudyInputChartStudySubgraphValues\(\).](#)
- [sc.GetChartStudyInputInt\(\).](#)
- [sc.GetChartStudyInputFloat\(\).](#)
- [sc.GetChartStudyInputString\(\).](#)
- [sc.GetChartStudyInputType\(\).](#)
- [sc.GetChartSymbol\(\).](#)
- [sc.GetChartTextFontFaceName\(\).](#)
- [sc.GetChartTimeZone\(\).](#)
- [sc.GetChartWindowHandle\(\).](#)
- [sc.GetCombineTradesIntoOriginalSummaryTradeSetting\(\).](#)
- [sc.GetContainingIndexForDateTimeIndex\(\).](#)
- [sc.GetContainingIndexForSCDateTime\(\).](#)
- [sc.GetCorrelationCoefficient\(\).](#)
- [sc.GetCountDownText\(\).](#)
- [sc.GetCurrentDateTime\(\).](#)
- [sc.GetCurrentTradedAskVolumeAtPrice\(\).](#)
- [sc.GetCurrentTradedBidVolumeAtPrice\(\).](#)
- [sc.GetCustomStudyControlBarButtonEnableState\(\).](#)
- [sc.GetDataDelayFromChart\(\).](#)
- [sc.GetDispersion\(\).](#)
- [sc.GetDOMColumnLeftCoordinate\(\).](#)
- [sc.GetDOMColumnRightCoordinate\(\).](#)
- [sc.GetEndingDateTimeForBarIndex\(\).](#)
- [sc.GetEndingDateTimeForBarIndexFromChart\(\).](#)
- [sc.GetExactMatchForSCDateTime\(\).](#)
- [sc.GetFirstIndexForDate\(\).](#)
- [sc.GetFirstNearestIndexForTradingDayDate\(\).](#)
- [sc.GetFlatToFlatTradeListEntry\(\).](#)
- [sc.GetFlatToFlatTradeListSize\(\).](#)

- [sc.GetGraphicsSetting\(\).](#)
- [sc.GetGraphVisibleHighAndLow\(\).](#)
- [sc.GetHideChartDrawingsFromOtherCharts\(\).](#)
- [sc.GetHighest\(\).](#)
- [sc.GetHighestChartNumberUsedInChartBook\(\).](#)
- [sc.GetIndexOfHighestValue\(\).](#)
- [sc.GetIndexOfLowestValue\(\).](#)
- [sc.GetIslandReversal\(\).](#)
- [sc.GetLastFileErrorCode\(\).](#)
- [sc.GetLastFileErrorMessage\(\).](#)
- [sc.GetLastPriceForTrading\(\).](#)
- [sc.GetLatestBarCountdownAsInteger\(\).](#)
- [sc.GetLineNumberOfSelectedUserDrawnDrawing\(\).](#)
- [sc.GetLowest\(\).](#)
- [sc.GetMainGraphVisibleHighAndLow\(\).](#)
- [sc.GetMarketDepthBars\(\).](#)
- [sc.GetMarketDepthBarsFromChart\(\).](#)
- [sc.GetMaximumMarketDepthLevels\(\).](#)
- [sc.GetNearestMatchForDateTimeIndex\(\).](#)
- [sc.GetNearestMatchForSCDateTime\(\).](#)
- [sc.GetNearestMatchForSCDateTimeExtended\(\).](#)
- [sc.GetNearestStopOrder\(\).](#)
- [sc.GetNearestTargetOrder\(\).](#)
- [sc.GetNumberOfBaseGraphArrays\(\).](#)
- [sc.GetNumberOfDataFeedSymbolsTracked\(\).](#)
- [sc.GetNumPriceLevelsForStudyProfile\(\).](#)
- [sc.GetNumStudyProfiles\(\).](#)
- [sc.GetOHLCForDate\(\).](#)
- [sc.GetOHLCOfTimePeriod\(\).](#)
- [sc.GetOpenHighLowCloseVolumeForDate\(\).](#)
- [sc.GetOrderByIndex\(\).](#)
- [sc.GetOrderByOrderID\(\).](#)
- [sc.GetOrderFillArraySize\(\).](#)
- [sc.GetOrderFillEntry\(\).](#)
- [sc.GetOrderForSymbolAndAccountByIndex\(\).](#)
- [sc.GetParentOrderIDFromAttachedOrderID.](#)
- [sc.GetPersistentDouble\(\).](#)
- [sc.GetPersistentFloat\(\).](#)
- [sc.GetPersistentInt\(\).](#)
- [sc.GetPersistentInt64\(\).](#)
- [sc.GetPersistentPointer\(\).](#)
- [sc.GetPersistentSCDateTime\(\).](#)
- [sc.GetPersistentSCString\(\).](#)
- [Persistent Variable Functions.](#)
- [sc.GetPersistentDoubleFast\(\).](#)
- [sc.GetPersistentFloatFast\(\).](#)

- [sc.GetPersistentIntFast\(\).](#)
- [sc.GetPersistentSCDateTimeFast\(\).](#)
- [Fast Persistent Variable Functions.](#)
- [sc.GetPersistentDoubleFromChartStudy\(\).](#)
- [sc.GetPersistentFloatFromChartStudy\(\).](#)
- [sc.GetPersistentIntFromChartStudy\(\).](#)
- [sc.GetPersistentInt64FromChartStudy\(\).](#)
- [sc.GetPersistentPointerFromChartStudy\(\).](#)
- [sc.GetPersistentSCDateTimeFromChartStudy\(\).](#)
- [sc.GetPersistentSCStringFromChartStudy\(\).](#)
- [Chart Study Persistent Variable Functions.](#)
- [sc.GetPointOfControlAndValueAreaPricesForBar\(\).](#)
- [sc.GetPointOfControlPriceVolumeForBar\(\).](#)
- [sc.GetProfitManagementStringForTradeAccount\(\).](#)
- [sc.GetRealTimeSymbol\(\).](#)
- [sc.GetRecentAskVolumeAtPrice\(\).](#)
- [sc.GetRecentBidVolumeAtPrice\(\).](#)
- [sc.GetReplayHasFinishedStatus\(\).](#)
- [sc.GetReplayStatusFromChart\(\).](#)
- [sc.GetSessionTimesFromChart\(\).](#)
- [sc.GetSheetCellAsDouble\(\).](#)
- [sc.GetSheetCellAsString\(\).](#)
- [sc.GetSpreadsheetSheetHandleByName\(\).](#)
- [sc.GetStandardError\(\).](#)
- [sc.GetStartDateTimeForTradingDate\(\).](#)
- [sc.GetStartOfPeriodForDateTime\(\).](#)
- [sc.GetStudyArray\(\).](#)
- [sc.GetStudyArrayFromChart\(\).](#)
- [sc.GetStudyArrayFromChartUsingID\(\).](#)
- [sc.GetStudyArraysFromChart\(\).](#)
- [sc.GetStudyArraysFromChartUsingID\(\).](#)
- [sc.GetStudyArrayUsingID\(\).](#)
- [sc.GetStudyDataColorArrayFromChartUsingID\(\).](#)
- [sc.GetStudyDataStartIndexFromChartUsingID\(\).](#)
- [sc.GetStudyDataStartIndexUsingID\(\).](#)
- [sc.GetStudyExtraArrayFromChartUsingID\(\).](#)
- [sc.GetStudyIDByIndex\(\).](#)
- [sc.GetStudyIDByName\(\).](#)
- [sc.GetStudyInternalIdentifier\(\).](#)
- [sc.GetStudyLineUntilFutureIntersection\(\).](#)
- [sc.GetNumLinesUntilFutureIntersection\(\).](#)
- [sc.GetStudyLineUntilFutureIntersectionByIndex\(\).](#)
- [sc.GetStudyName\(\).](#)
- [sc.GetStudyNameFromChart\(\).](#)
- [sc.GetStudyNameUsingID\(\).](#)
- [sc.GetStudyPeakValleyLine\(\).](#)

- [sc.GetStudyProfileInformation\(\).](#)
- [sc.GetStudyStorageBlockFromChart\(\).](#)
- [sc.GetStudySubgraphColors\(\).](#)
- [sc.GetStudySubgraphDrawStyle\(\).](#)
- [sc.GetStudySubgraphLineStyle\(\).](#)
- [sc.GetStudySubgraphLineWidth\(\).](#)
- [sc.GetStudySubgraphName\(\).](#)
- [sc.GetStudySubgraphNameFromChart\(\).](#)
- [sc.GetStudySummaryCellAsDouble\(\).](#)
- [sc.GetStudySummaryCellAsString\(\).](#)
- [sc.GetStudyVisibilityState\(\).](#)
- [sc.GetSummation\(\).](#)
- [sc.GetSymbolDataValue\(\).](#)
- [sc.GetSymbolDescription\(\).](#)
- [sc.GetTimeAndSales\(\).](#)
- [sc.GetTimeAndSalesForSymbol\(\).](#)
- [sc.GetTimeSalesArrayIndexesForBarIndex\(\).](#)
- [sc.GetTotalNetProfitLossForAllSymbols\(\).](#)
- [sc.GetTradeAccountData\(\).](#)
- [sc.GetTradeListEntry\(\).](#)
- [sc.GetTradeListSize\(\).](#)
- [sc.GetTradePosition\(\).](#)
- [sc.GetTradePositionByIndex\(\).](#)
- [sc.GetTradePositionForSymbolAndAccount\(\).](#)
- [sc.GetTradeServiceAccountBalanceForTradeAccount\(\).](#)
- [sc.GetTradeStatisticsForSymbolV2\(\).](#)
- [sc.GetTradeSymbol\(\).](#)
- [sc.GetTradeWindowOrderType\(\).](#)
- [sc.GetTradeWindowTextTag\(\).](#)
- [sc.GetTradingDayDate\(\).](#)
- [sc.GetTradingDayDateForChartNumber\(\).](#)
- [sc.GetTradingDayStartDateTimeOfBar\(\).](#)
- [sc.GetTradingDayStartDateTimeOfBarForChart\(\).](#)
- [sc.GetTradingErrorTextMessage\(\).](#)
- [sc.GetTradingKeyboardShortcutsEnableState\(\).](#)
- [sc.GetTrueHigh\(\).](#)
- [sc.GetTrueLow\(\).](#)
- [sc.GetTrueRange\(\).](#)
- [sc.GetUserDrawingByLineNumber\(\).](#)
- [sc.GetUserDrawnChartDrawing\(\).](#)
- [sc.GetUserDrawnDrawingByLineNumber\(\).](#)
- [sc.GetUserDrawnDrawingsCount\(\).](#)
- [sc.GetValueFormat\(\).](#)
- [sc.GetVolumeAtPriceDataForStudyProfile\(\).](#)
- [sc.GetYValueForChartDrawingAtBarIndex\(\).](#)
- [sc.HeikinAshi\(\).](#)

- [sc.Highest\(\)](#).
- [sc.HullMovingAverage\(\)](#).
- [sc.HurstExponent\(\)](#).
- [sc.InstantaneousTrendline\(\)](#).
- [sc.InverseFisherTransform\(\)](#).
- [sc.InverseFisherTransformRSI\(\)](#).
- [sc.IsChartDataLoadingCompleteForAllCharts\(\)](#).
- [sc.IsChartDataLoadingInChartbook\(\)](#).
- [sc.IsChartNumberExist\(\)](#).
- [sc.IsChartZoomInStateActive\(\)](#).
- [sc.IsDateTimeContainedInBarAtIndex\(\)](#).
- [sc.IsDateTimeContainedInBarIndex\(\)](#).
- [sc.IsDateTimeInDaySession\(\)](#).
- [sc.IsDateTimeInEveningSession\(\)](#).
- [sc.IsDateTimeInSession\(\)](#).
- [sc.IsIsSufficientTimePeriodInDate\(\)](#).
- [sc.IsMarketDepthDataCurrentlyAvailable\(\)](#).
- [sc.IsNewBar\(\)](#).
- [sc.IsNewTradingDay\(\)](#).
- [sc.IsReplayRunning\(\)](#).
- [sc.IsSwingHigh\(\)](#).
- [sc.IsSwingLow\(\)](#).
- [sc.IsVisibleSubgraphDrawStyle\(\)](#).
- [IsWorkingOrderStatus\(\)](#).
- [IsWorkingOrderStatusIgnorePendingChildren\(\)](#).
- [sc.Keltner\(\)](#).
- [sc.LaguerreFilter\(\)](#).
- [sc.LinearRegressionIndicator\(\)](#).
- [sc.LinearRegressionIndicatorAndStdErr\(\)](#).
- [sc.LinearRegressionIntercept\(\)](#).
- [sc.LinearRegressionSlope\(\)](#).
- [sc.Lowest\(\)](#).
- [sc.MACD\(\)](#).
- [sc.MakeHTTPBinaryRequest\(\)](#).
- [sc.MakeHTTPPOSTRequest\(\)](#).
- [sc.MakeHTTPRequest\(\)](#).
- [sc.Momentum\(\)](#).
- [sc.MovingAverage\(\)](#).
- [sc.MovingAverageCumulative\(\)](#).
- [sc.MovingMedian\(\)](#).
- [sc.MultiplierFromVolumeValueFormat\(\)](#).
- [sc.NumberOfBarsSinceHighestValue\(\)](#).
- [sc.NumberOfBarsSinceLowestValue\(\)](#).
- [sc.OnBalanceVolume\(\)](#).
- [sc.OnBalanceVolumeShortTerm\(\)](#).
- [sc.OpenChartbook\(\)](#).

- [sc.OpenChartOrGetChartReference\(\)](#).
- [sc.OpenFile\(\)](#).
- [sc.OrderQuantityToString\(\)](#).
- [sc.Oscillator\(\)](#).
- [sc.Parabolic\(\)](#).
- [sc.PauseChartReplay\(\)](#).
- [sc.PlaySound\(\)](#).
- [sc.PriceValueToTicks\(\)](#).
- [sc.PriceVolumeTrend\(\)](#).
- [sc.RandomWalkIndicator\(\)](#).
- [sc.ReadFile\(\)](#).
- [sc.ReadIntradayFileRecordAtIndex\(\)](#).
- [sc.ReadIntradayFileRecordForBarIndexAndSubIndex\(\)](#).
- [sc.RecalculateChart\(\)](#).
- [sc.RecalculateChartImmediate\(\)](#).
- [sc.RefreshTradeData\(\)](#).
- [sc.RegionValueToYPixelCoordinate\(\)](#).
- [sc.RelayDataFeedAvailable\(\)](#).
- [sc.RelayDataFeedUnavailable\(\)](#).
- [sc.RelayNewSymbol\(\)](#).
- [sc.RelayServerConnected\(\)](#).
- [sc.RelayTradeUpdate\(\)](#).
- [sc.RemoveACSCChartShortcutMenuItem\(\)](#).
- [sc.RemoveStudyFromChart\(\)](#).
- [sc.ResizeArrays\(\)](#).
- [sc.ResumeChartReplay\(\)](#).
- [sc.RGBInterpolate\(\)](#).
- [sc.Round\(\)](#).
- [sc.RoundToTickSize\(\)](#).
- [sc.RSI\(\)](#).
- [sc.SaveChartbook\(\)](#).
- [sc.SaveChartImageToFileExtended\(\)](#).
- [sc.SecondsSinceStartTime\(\)](#).
- [sc.SecurityType\(\)](#).
- [sc.SendEmailMessage\(\)](#).
- [sc.SessionStartTime\(\)](#).
- [sc.SetACSCChartShortcutMenuItemChecked\(\)](#).
- [sc.SetACSCChartShortcutMenuItemDisplayed\(\)](#).
- [sc.SetACSCChartShortcutMenuItemEnabled\(\)](#).
- [sc.SetACSToolButtonText\(\)](#).
- [sc.SetACSToolEnable\(\)](#).
- [sc.SetACSToolToolTip\(\)](#).
- [sc.SetAlert\(\)](#).
- [sc.SetAttachedOrders\(\)](#).
- [sc.SetBarPeriodParameters\(\)](#).
- [sc.SetChartStudyInputChartStudySubgraphValues\(\)](#).

- [sc.SetChartStudyInputFloat\(\)](#).
- [sc.SetChartStudyInputInt\(\)](#).
- [sc.SetChartStudyInputString\(\)](#).
- [sc.SetChartTradeMode\(\)](#).
- [sc.SetChartWindowState\(\)](#).
- [sc.SetCombineTradesIntoOriginalSummaryTradeSetting\(\)](#).
- [sc.SetCustomStudyControlBarButtonColor\(\)](#).
- [sc.SetCustomStudyControlBarButtonEnable\(\)](#).
- [sc.SetCustomStudyControlBarButtonHoverText\(\)](#).
- [sc.SetCustomStudyControlBarButtonShortCaption\(\)](#).
- [sc.SetCustomStudyControlBarButtonText\(\)](#).
- [sc.SetGraphicsSetting\(\)](#).
- [sc.SetHorizontalGridState\(\)](#).
- [sc.SetNumericInformationDisplayOrderFromString\(\)](#).
- [sc.SetNumericInformationGraphDrawTypeConfig\(\)](#).
- [sc.SetPersistentDouble\(\)](#).
- [sc.SetPersistentDoubleForChartStudy\(\)](#).
- [sc.SetPersistentFloat\(\)](#).
- [sc.SetPersistentFloatForChartStudy\(\)](#).
- [sc.SetPersistentInt\(\)](#).
- [sc.SetPersistentIntForChartStudy\(\)](#).
- [sc.SetPersistentInt64\(\)](#).
- [sc.SetPersistentInt64ForChartStudy\(\)](#).
- [sc.SetPersistentPointer\(\)](#).
- [sc.SetPersistentPointerForChartStudy\(\)](#).
- [sc.SetPersistentSCDateTime\(\)](#).
- [sc.SetPersistentSCDateTimeForChartStudy\(\)](#).
- [sc.SetPersistentSCString\(\)](#).
- [sc.SetPersistentSCStringForChartStudy\(\)](#).
- [sc.SetSheetCellAsDouble\(\)](#).
- [sc.SetSheetCellAsString\(\)](#).
- [sc.SetStudySubgraphColors\(\)](#).
- [sc.SetStudySubgraphDrawStyle\(\)](#).
- [sc.SetStudySubgraphLineStyle\(\)](#).
- [sc.SetStudySubgraphLineWidth\(\)](#).
- [sc.SetStudyVisibilityState\(\)](#).
- [sc.SetTradeWindowTextTag\(\)](#).
- [sc.SetTradingKeyboardShortcutsEnableState\(\)](#).
- [sc.SetTradingLockState\(\)](#).
- [sc.SetUseGlobalGraphicsSettings\(\)](#).
- [sc.SetVerticalGridState\(\)](#).
- [sc.SimpleMovAvg\(\)](#).
- [sc.Slope\(\)](#).
- [sc.SlopeToAngleInDegrees\(\)](#).
- [sc.SmoothedMovingAverage\(\)](#).
- [sc.StartChartReplay\(\)](#).

- [sc.StartChartReplayNew\(\)](#).
- [sc.StartScanOfSymbolList\(\)](#).
- [sc.StartDownloadHistoricalData\(\)](#).
- [sc.StdDeviation\(\)](#).
- [sc.StdError\(\)](#).
- [sc.Stochastic\(\)](#).
- [sc.StopChartReplay\(\)](#).
- [sc.StopScanOfSymbolList\(\)](#).
- [sc.StringToDouble\(\)](#).
- [sc.SubmitOCOOrder\(\)](#).
- [sc.Summation\(\)](#).
- [sc.SuperSmoother2Pole\(\)](#).
- [sc.SuperSmoother3Pole\(\)](#).
- [sc.T3MovingAverage\(\)](#).
- [sc.TEMA\(\)](#).
- [sc.TicksToPriceValue\(\)](#).
- [sc.TimeMSToString\(\)](#).
- [sc.TimePeriodSpan\(\)](#).
- [sc.TimeSpanOfBar\(\)](#).
- [sc.TimeStringToSCDateTime\(\)](#).
- [sc.TimeToString\(\)](#).
- [sc.TradingDayStartsInPreviousDate\(\)](#).
- [sc.TriangularMovingAverage\(\)](#).
- [sc.TRIX\(\)](#).
- [sc.TrueRange\(\)](#).
- [sc.UltimateOscillator\(\)](#).
- [sc.UploadChartImage\(\)](#).
- [sc.UserDrawnChartDrawingExists\(\)](#).
- [sc.UseTool\(\)](#).
- [sc.VHF\(\)](#).
- [sc.VolumeWeightedMovingAverage\(\)](#).
- [sc.Vortex\(\)](#).
- [sc.WeightedMovingAverage\(\)](#).
- [sc.WellesSum\(\)](#).
- [sc.WildersMovingAverage\(\)](#).
- [sc.WilliamsAD\(\)](#).
- [sc.WilliamsR\(\)](#).
- [sc.WriteBarAndStudyDataToFile\(\)](#).
- [sc.WriteBarAndStudyDataToFileEx\(\)](#).
- [sc.WriteFile\(\)](#).
- [sc.YPixelCoordinateToGraphValue\(\)](#).
- [sc.ZeroLagEMA\(\)](#).
- [sc.ZigZag\(\)](#).
- [sc.ZigZag2\(\)](#).
- [min\(\)](#).
- [max\(\)](#).

- [Common Function Parameter Descriptions / Common Parameters for Intermediate Study Calculation Functions.](#)
- [Advanced Custom Study Tool and Chart Drawing Functions.](#)
- [Automated Trading Functions and Variables for Trading System Studies.](#)

Functions

Notes About Output Arrays for Functions

In the descriptions below for the functions, Intermediate Study Calculation Functions are identified by **Type:** [Intermediate Study Calculation Function](#) in the description.

Intermediate Study Calculation Functions in most cases require one or more or arrays for output of the results. These can take one of two types.

These types can be: **SCFloatArrayRef** (a reference to a Sierra Chart array of Float values) or **SCSubgraphRef** (a reference to a `sc.Subgraph[]` which contains multiple `SCFloatArray` arrays).

SCFloatArrayRef

SCFloatArrayRef: For this type you can pass a `sc.Subgraph[].Data` array using [sc.Subgraph\[\]](#) or [sc.Subgraph\[\].Data](#). In the case of `sc.Subgraph[]`, the `Data` array will automatically be passed since there is a conversion operator on that object which returns the `Data` array when a **SCFloatArrayRef** is required.

Both of these are equivalent to each other. In each case, the `sc.Subgraph[].Data[]` array of floats will get passed in.

Or, you can pass in a `sc.Subgraph[]` internal extra array using `sc.Subgraph[].Arrays[]`. If you do not need visible output on the chart for the results and you want to conserve the visible/graphable `sc.Subgraph[].Data` arrays, then use a `sc.Subgraph[].Arrays[]` internal extra array by passing a [sc.Subgraph\[\].Arrays\[\]](#) for the output array parameter.

SCSubgraphRef

SCSubgraphRef: For this type you can only pass a [sc.Subgraph\[\]](#).

The **SCSubgraphRef** type is required because internally the function will use the available internal extra arrays which are part of a `sc.Subgraph`. These arrays are either used for internal calculations or are used for additional output.

If they are used for additional output, then that is clearly explained in the documentation for the function. For example, the [sc.MACD\(\)](#) function will place output for additional MACD related lines into the `sc.Subgraph[].Arrays[]`.

There is one point of clarification. When a `sc.Subgraph` is required, you cannot use

sc.Subgraph[].Data. You must only use sc.Subgraph[]. This will pass in the entire sc.Subgraph[] object because the function requires a sc.Subgraph[] object.

When using a sc.Subgraph[] object and you do not want to have the result of the Intermediate Study Calculation Function actually drawn on the chart, then set [sc.Subgraph\[\].DrawStyle](#) = **DRAWSTYLE_IGNORE**.

Array Based Study Functions That Do Not Use the Index Parameter

The Advanced Custom Study Interface has versions of functions (function "overloads" as known in C++) that take input and output arrays as parameters, and do not require the **Index** parameter.

These functions are called Intermediate Study Calculation Functions whether they require the **Index** parameter or not.

For example, they may calculate a Moving Average from an input data array and place the results into an output data array.

The versions that do not require the **Index** parameter simplify the calling of these functions. If your study function uses **Automatic Looping** by setting **sc.AutoLoop = 1**; in the **if(sc.SetDefaults)** block, then you can use these functions.

The function versions that do not use the **Index** parameter will not function properly when you are using Manual Looping.

You will see an example below of a call to an Intermediate Study Calculation Function, that uses the **Index** parameter and another call to a second version of that same function that does not use the **Index** parameter.

If your study function is using Automatic Looping, then you can use a version of an Intermediate Study Calculation Function that takes the **Index** parameter or the one that does not.

Using the version that does not require the **Index** parameter simply makes writing your code simpler.

When an **Index** parameter is not specified, the calculation always begins at sc.Index and will refer to data at that index and prior indexes. For example, calculating a moving average with a **Length** of 10, will start the calculation at sc.Index and go back 9 prior index elements for a total of 10.

Example

```

if(sc.SetDefaults)
{
    //...
    sc.AutoLoop = 1;
    //...
}

//Calculates a 20 period moving average of Last prices.
sc.SimpleMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], sc.Index, 20);

//Calculates a 20 period moving average of Last prices.
//Index parameter not used. Same as above, but a more simple function call.
sc.SimpleMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);

```

Return Object of Array Based Study Functions

Intermediate Study Calculation Functions that take a **SCFloatArrayRef** or **SCSubgraphRef** parameter for output, will return a **SCFloatArray** object by reference. This return object is either the SCFloatArray parameter, or the sc.SCSubgraph[].Data array of the sc.SCSubgraph parameter.

Example

```

SCSubgraphRef MidBand = sc.Subgraph[1];

// Copy the middle Bollinger band value to Subgraph 10 at the current index
sc.Subgraph[10][sc.index]= sc.BollingerBands(

sc.BaseDataIn[SC_LAST],
MidBand,
Length.GetInt(),
StandardDeviations.GetFloat(),

MAType.GetMovAvgType() )[sc.index];

```

Working with Intermediate Study Calculation Functions

The code example below demonstrates using Intermediate Study Calculation Functions.

Example

```
//Below are example calls to ACSIL intermediate study calculation functions.

//In this example we are giving the study function a graphable sc.Subgraph[].Data array.
//Even though that this Subgraph result can be drawn on the chart, it does not need to be
//if it does not have a visible DrawStyle
sc.SimpleMovAvg(sc.BaseData[SC_LAST], sc.Subgraph[0].Data, 10);

//Get the value from the calculation above
float AverageAtIndex = sc.Subgraph[0].Data[sc.Index];

//In this example we are giving the study function a Subgraph internal extra
//array which is not capable of being graphed.
sc.SimpleMovAvg(sc.BaseData[SC_LAST], sc.Subgraph[0].Arrays[0], 10);

AverageAtIndex = sc.Subgraph[0].Arrays[0][sc.Index];
```

The actual source code for intermediate study calculations functions is located in the **SCStudyFunctions.cpp** file in the **/ACS_Source** folder in the folder Sierra Chart is installed to on your computer system.

Cumulative Calculations with Intermediate Study Functions

The Output array parameter of a intermediate study calculation function can be used as the Input array parameter for another intermediate study calculation function.

The **scsf_AverageOfAverage** function in the **/ACS_Source/studies3.cpp** file in the Sierra Chart installation folder is an example of a function that shows how to use 2 intermediate study calculation functions (sc.LinearRegressionIndicator, sc.ExponentialMovAvg) together.

It calculates the Exponential Moving Average of a Linear Regression Indicator by passing the Output array from the Linear Regression Indicator to the Input array parameter of Exponential Moving Average.

sc.AdaptiveMovAvg()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **AdaptiveMovAvg**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**, float **FastSmoothConstant**, float **SlowSmoothConstant**);

SCFloatArrayRef **AdaptiveMovAvg**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**, float **FastSmoothConstant**, float **SlowSmoothConstant**); [Auto-looping only](#).

The **sc.AdaptiveMovAvg()** function calculates the standard Adaptive Moving Average study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).

- [Index](#).
- [Length](#).
- **FastSmoothConstant**: Fast smoothing constant.
- **SlowSmoothConstant**: Slow smoothing constant.

Example

```
sc.AdaptiveMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20, 2.0f, 30.0f);

float MAValue = sc.Subgraph[0][sc.Index];
```

sc.AddACSChartShortcutMenuItem()

Refer to the [sc.AddACSChartShortcutMenuItem\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

sc.AddACSChartShortcutMenuSeparator()

Refer to the [sc.AddACSChartShortcutMenuSeparator\(\)](#) section for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

sc.AddAlertLine()

Type: Function

```
AddAlertLine(SCString Message, int ShowAlertLog = 0);
```

```
AddAlertLine(char* Message, int ShowAlertLog = 0);
```

sc.AddAlertLine() is a function for adding an Alert Message to the Sierra Chart **Alerts Log**.

To open the Alerts Log, select **Window >> Alert Manager >> Alert Log**.

This function adds a type of Alert Message to the **Alerts Log** which allows the **Go to Chart** commands on the **Alerts Log** to be used.

The **Message** text can be any text that you want to display in the **Alerts Log**. **Message** can be either a **SCString** type or a plain C++ string ("This is an Example").

ShowAlertLog needs to be set to **1** to cause the **Alerts Log** to open, if it is not already, when a message is added. Otherwise, **ShowAlertLog** needs to be **0** or it can be optionally left out to not open the **Alerts Log** when a message is added.

Refer to the **scsf_LogAndAlertExample()** function in the **/ACS_Source/studies.cpp** file in the Sierra Chart installation folder for example code to work with this function.

To make an Alert Message that contains formatted variables, refer to the [Working With Text](#)

[Strings](#) section.

Example

```
// Add an alert line to the Alerts Log
sc.AddAlertLine("Condition is TRUE");

sc.AddAlertLine("Condition is TRUE. The Alerts Log will open if it is not already.",1);

SCString MyString= "This is my string.";

sc.AddAlertLine(MyString,1);
```

sc.AddAlertLineWithDateTime()

Type: Function

AddAlertLineWithDateTime(const char* **Message**, int **ShowAlertLog**, SCDatetime **AlertDateTime**);

The **sc.AddAlertLineWithDateTime** function is identical to [sc.AddAlertLine](#) except that it has an **AlertDateTime** parameter of type [SCDateTime](#), which can be set to a Date-Time value which will be included in the Alerts Log message.

When using the **Go to Chart** commands on the Alerts Log, the chart will be scrolled to this particular Date-Time.

sc.AddAndManageSingleTextDrawingForStudy()

Type: Function

void AddAndManageSingleTextDrawingForStudy(SCStudyInterfaceRef &**sc**, bool **DisplayInFillSpace**, int **HorizontalPosition**, int **VerticalPosition**, SCSubgraphRef **Subgraph**, int **TransparentLabelBackground**, SCString& **TextToDisplay**, int **DrawAboveMainPriceGraph**, int **BoldFont**);

The **sc.AddAndManageSingleTextDrawingForStudy** function adds a text drawing to the chart based upon the given parameters and implements the low-level management of that drawing.

This drawing is a nonuser drawn type of drawing. It cannot be interacted with by the user on the chart.

Only a single text drawing can be added with this function per study instance. Multiple text drawings are not supported. To do multiple text drawings you need to use the [Text Drawing Tool](#) from ACSIL.

For an example to use this function, refer to the **scsf_LargeTextDisplayForStudy** study function in the /ACS_Source/studies3.cpp file in the Sierra Chart installation folder.

Parameters

- **sc**: A reference to the SCStudyInterface structure given to the study function.
- **DisplayInFillSpace**: Set to 1 to indicate to display the drawing in the fill space on the right side of the chart.
- **HorizontalPosition**: Sets the horizontal position relative to the left edge of the chart window. For more details, refer to [s UseTool::BeginDateTime](#). This is not a Date-Time value.
- **VerticalPosition**: This is a relative vertical value relative to the bottom of the Chart Region the study is located in. This is not a price value. For more details, refer to [s UseTool::BeginValue](#).
- **Subgraph**: This is a reference to the study Subgraph which controls the text color and text size. The Subgraph needs to have the DrawStyle set to DRAWSTYLE_CUSTOM_TEXT.
- **TransparentLabelBackground**: Set to 1 to use a transparent background for the text.
- **TextToDisplay**: This is the actual text string to display.
- **DrawAboveMainPriceGraph**: Set to 1 to draw above the main price graph, which means it will be displayed above the chart bars.
- **BoldFont**: Set to 1 to use a bold font. Set this to 0 to not use a bold font.

sc.AddAndManageSingleTextUserDrawnDrawingForStudy()

Type: Function

```
void AddAndManageSingleTextUserDrawnDrawingForStudy(SCStudyInterfaceRef &sc, int Unused, int HorizontalPosition, int VerticalPosition, SCSubgraphRef Subgraph, int TransparentLabelBackground, SCString& TextToDisplay, int DrawAboveMainPriceGraph, int LockDrawing, int UseBoldFont);
```

Parameters

- **sc**: A reference to the SCStudyInterface structure given to the study function.
- **Unused**:
- **HorizontalPosition**: Sets the horizontal position relative to the left edge of the chart window. For more details, refer to [s UseTool::BeginDateTime](#). This is not a Date-Time value.
- **VerticalPosition**: This is a relative vertical value relative to the bottom of the Chart Region the study is located in. This is not a price value. For more details, refer to [s UseTool::BeginValue](#).
- **Subgraph**: This is a reference to the study Subgraph which controls the text color and text size. The Subgraph needs to have the DrawStyle set to DRAWSTYLE_CUSTOM_TEXT.
- **TransparentLabelBackground**: Set to 1 to use a transparent background for the text.
- **TextToDisplay**: This is the actual text string to display.

- **DrawAboveMainPriceGraph:** Set to 1 to draw above the main price graph, which means it will be displayed above the chart bars.
- **LockDrawing:**
- **BoldFont:** Set to 1 to use a bold font. Set this to 0 to not use a bold font.

sc.AddDateToExclude()

Type: Function

```
int AddDateToExclude(const int ChartNumber, const SCDatetime& DateToExclude);
```

The **sc.AddDateToExclude()** function .

Parameters

- **ChartNumber:**
- **DateToExclude:**

Example

sc.AddElements()

Type: Function

```
int AddElements(int NumElements);
```

sc.AddElements() adds the number of elements specified with the **NumElements** parameter, to the [sc.Subgraph\[\].Data\[\]](#) arrays.

The arrays up to the last actually used **sc.Subgraph[].Data** array, will actually have elements added. Unused **sc.Subgraph[].Data** arrays will be left un-allocated until they are needed.

This function must only used when you have set [sc.IsCustomChart](#) to 1 (TRUE).

The function returns 0 if it fails to add the requested number of elements to all the arrays.

This function also adds elements to the [sc.DateTimeOut\[\]](#) , [sc.Subgraph\[\].DataColor\[\]](#), and [sc.Subgraph\[\].Arrays\[\]\[\]](#) arrays if they are used.

Example

```
sc.AddElements(5); // Add five elements to the arrays
```

sc.AddLineUntilFutureIntersection()

Type: Function

AddLineUntilFutureIntersection(int **StartBarIndex** , int **LineIDForBar** , float **LineValue** , COLORREF **LineColor** , unsigned short **LineWidth** , unsigned short **LineStyle** , int **DrawValueLabel** , int **DrawNameLabel** , const SCString& **NameLabel**);

The **sc.AddLineUntilFutureIntersection()** function draws a line from the chart bar specified by the **StartBarIndex** parameter which specifies its array index, and at the value specified by the **LineValue** parameter. The line extends until it is intersected by a future price bar. The other supported parameters are described below.

All lines added by this function are automatically deleted any time the study they were added by, is removed from the chart or any time [sc.IsFullRecalculation](#) is TRUE.

Therefore, there is no need to delete them by calling [sc.DeleteLineUntilFutureIntersection](#). However, this function can be used in other cases.

The type of line drawn by the **sc.AddLineUntilFutureIntersection()** function is not related to [Chart Drawing Tools](#) or to the [sc.UseTool](#) function.

Parameters

- **StartBarIndex**: This is the array index of the bar that the line begins at.
- **LineIDForBar**: This is the identifier of the extension line for a chart bar. If there is only one line for a chart bar, this can be set to 0 and that will be the identifier. If there are multiple lines for a chart bar, each line needs to have a unique integer identifier. When you want to update the line, specify the same **LineIDForBar** as was previously specified.
- **LineValue**: This the vertical axis level at which the line is drawn at.
- **LineColor**: This is the color of the line.
- **LineWidth**: This is the width of the line in pixels.
- **LineStyle**: This is the style of the line. For the available integer constants which can be used, refer to [sc.Subgraph\[\].LineStyle](#).
- **DrawValueLabel**: Set this to 1/TRUE to draw value label with the line. This will be the **LineValue** displayed as text.
- **DrawNameLabel**: Set this to 1/TRUE to draw a name label with the line.
- **NameLabel**: When **DrawNameLabel** is set to 1/TRUE, then the specifies the text to display.

Example

For an example to use the function, refer to the **scsf_ExtendClosesUntilFutureIntersection** function in the **/ACS_Source/studies5.cpp** in the Sierra Chart installation folder.

sc.AddLineUntilFutureIntersectionEx()

Type: Function

```
AddLineUntilFutureIntersectionEx(const n_ACSIL::s_LineUntilFutureIntersection  
LineUntilFutureIntersection);
```

The **sc.AddLineUntilFutureIntersectionEx()** function draws a line from the chart bar specified by the **n_ACSIL::s_LineUntilFutureIntersection::StartBarIndex** parameter which specifies its array index, and at the value specified by the **n_ACSIL::s_LineUntilFutureIntersection::LineValue** parameter.

The line extends until it is intersected by a future price bar. The other supported parameters are described below.

All lines added by this function are automatically deleted any time the study they were added by, is removed from the chart or any time [sc.IsFullRecalculation](#) is TRUE.

Therefore, there is no need to delete them by calling [sc.DeleteLineUntilFutureIntersection](#). However, this function can be used in other cases.

The type of line drawn by the **sc.AddLineUntilFutureIntersection()** function is not related to [Chart Drawing Tools](#) or to the [sc.UseTool](#) function.

s_LineUntilFutureIntersection struct

- **StartBarIndex (int)**: This variable can be set to the chart bar index where the future intersection line needs to stop. Once this is set to a nonzero value, the line will no longer automatically stop when it intersects a price bar. When **EndBarIndex** is set, the ending chart bar index is controlled by the study directly.
- **EndBarIndex (int)**: When this is set to a nonzero value, it specifies at what chart bar index the future intersection line will end. Therefore, it no longer extends until intersecting with a particular price bar.

When this is set to a nonzero value, it is the study itself which controls the ending chart bar index of the line.

- **LineIDForBar (int)**: This is the identifier of the extension line for a chart bar. If there is only one line for a chart bar, this can be set to 0 and that will be the identifier. If there are multiple lines for a chart bar, each line needs to have a unique integer identifier. When you want to update the line, specify the same **LineIDForBar** as was previously specified.
- **LineValue (float)**: This the vertical axis level at which the line is drawn at.
- **LineValue2ForRange (float)**: Refer to **UseLineValue2**.
- **UseLineValue2 (int)**: When **UseLineValue2** is set to a nonzero value, then **LineValue2ForRange** specifies the vertical axis value for the other side of the range that is filled with the **LineColor**. **LineValue** specifies one side of the range and **LineValue2ForRange** specifies the other side of the range and it is filled with the **LineColor**.

- **LineColor** (uint32_t): This is the color of the line or the filled range.
- **LineWidth** (unsigned short): This is the width of the line in pixels. Only applicable when **UseLineValue2** is sent to 0.
- **LineStyle** (unsigned short): This is the style of the line. For the available integer constants which can be used, refer to [sc.Subgraph\[\].LineStyle](#). Only applicable when **UseLineValue2** is sent to 0.
- **DrawValueLabel** (int): Set this to 1/TRUE to draw value label with the line. This will be the **LineValue** displayed as text.
- **DrawNameLabel** (int): Set this to 1/TRUE to draw a name label with the line.
- **NameLabel** (SCString): When **DrawNameLabel** is set to 1/TRUE, then the specifies the text to display.
- **AlwaysExtendToEndOfChart** (int): When this is set to a nonzero value, then the line always extends to the end of the chart.
- **TransparencyLevel** (int): This specifies the transparency level as a percentage where 0 = 0% and 100 = 100%. 100 means that the extension line would not be visible.
- **PerformCloseCrossoverComparison** (int): When this is set to a nonzero value the future intersection is determined by the future intersection line crossing the closing prices of the bar.

Example

For an example to use the function, refer to the **scsf_VolumeAtPriceThresholdAlertV2** function in the **/ACS_Source/studies8.cpp** in the Sierra Chart installation folder.

sc.AddMessageToLog()

Type: Function

AddMessageToLog(SCString& **Message**, int **Showlog**);

sc.AddMessageToLog() is used to add a message to the log. See the **scsf_LogAndAlertExample()** function in the studies.cpp file inside the ACS_Source folder inside of the Sierra Chart installation folder for example code on how to work with this function.

Example

```
sc.AddMessageToLog("This line of text will be added to the Message Log, but the message log will not pop
sc.AddMessageToLog("This line of text will be added to the Message Log, and this call will show the Messe
```

If you want to make a message line that contains formatted variables to add to the Message Log, refer to the [Working With Text Strings and Setting Names](#) section.

sc.AddStudyToChart()

Type: Function

```
int AddStudyToChart(const n_ACSIL::s_AddStudy& AddStudy);
```

The **sc.AddStudyToChart** is used to add a new study to a chart. This function takes a reference to the **n_ACSIL::s_AddStudy** structure which is documented below which specifies all of the parameters.

The various inputs available with a study can be set with the [sc.SetChartStudyInput*](#) functions.

n_ACSIL::s_AddStudy structure

- **ChartNumber (int)**: The chart number to add the study to. This needs to be within the same Chartbook as the study function which is adding the study.
- **StudyID (int)**: The configured study identifier to add if it is a Sierra Chart built-in study. To see the Study Identifiers you need to be running a current version of Sierra Chart. The Study Identifiers are listed in the [Studies to Graph](#) list in the **Studies Window**. They are prefixed with: **S_ID:**.

When adding an advanced custom study this needs to be zero.

- **CustomStudyFileAndFunctionName (SCString)**: In the case of when adding an advanced custom study, this string specifies the DLL filename without extension, followed by a dot (.) character, and the function name to be added. When both StudyID and CustomStudyFileAndFunctionName have not been set and are at default values, then **sc.AddStudy** returns 0.

This method can also be used to add built-in studies. In this case the DLL file name and function name can be determined through the

DLLName.FunctionName setting in the [Study Settings](#) window for the study.

Example: **SierraChartStudies_64.scsf MovingAverageBlock**.

- **ShortName (SCString)**: This will be set as the short name for the added study. This can be used to get the ID for the study later using **sc.GetStudyIDByName**.

Example

```

// Do data processing
int& r_MenuID = sc.GetPersistentInt(1);

if (sc.LastCallToFunction)
{
    // Remove menu items when study is removed
    sc.RemoveACSCChartShortcutMenuItem(sc.ChartNumber, r_MenuID);

    return;
}

if (sc.UpdateStartIndex == 0 && r_MenuID <= 0)
{
    r_MenuID = sc.AddACSCChartShortcutMenuItem(sc.ChartNumber, "Add Study Example");
}

if (sc.MenuEventID == r_MenuID)
{
    n_ACSIL::s_AddStudy AddStudy;
    AddStudy.ChartNumber = sc.ChartNumber;
    AddStudy.StudyID = 2;
    AddStudy.ShortName = "New Moving Average";

    sc.AddStudyToChart(AddStudy);
}

```

sc.AdjustDateTimeToGMT()

Type: Function

AdjustDateTimeToGMT(const SCDatetime& **DateTime**);

The **sc.AdjustDateTimeToGMT()** function converts the given **DateTime** from the time zone Sierra Chart is set to, to the GMT time zone.

Example

```

SCDateTime DateTimeInGMT;
sc.AdjustDateTimeToGMT(sc.BaseDateTimeIn[sc.Index]);

```

sc.ADX()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **ADX** (SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **DXLength**, int **DXMovAvgLength**);

SCFloatArrayRef **ADX** (SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **DXLength**, int **DXMovAvgLength**); [Auto-looping only](#).

The **sc.ADX()** function calculates the Average Directional Index (ADX) study.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#): For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [DXLength](#).
- [DXMovAvgLength](#).

Example

```
sc.ADX(sc.BaseDataIn, sc.Subgraph[0], 14, 14);

float ADXValue = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.ADXR()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **ADXR** (SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **DXLength**, int **DXMovAvgLength**, int **ADXRIInterval**);

SCFloatArrayRef **ADXR** (SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **DXLength**, int **DXMovAvgLength**, int **ADXRIInterval**); [Auto-looping only](#).

The **sc.ADXR()** function calculates the Average Directional Movement Rating.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#): For this function, sc.Subgraph[].Arrays[0-4] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [DXLength](#).
- [DXMovAvgLength](#).
- [ADXRIInterval](#).

Example

```
sc.ADXR(sc.BaseDataIn, sc.Subgraph[0], 14, 14, 14);

float ADXRValue = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.AllocateMemory()

Type: Function

```
void* AllocateMemory (int Size);
```

The **sc.AllocateMemory** function allocates memory for the number of bytes specified by the **Size** parameter. A pointer to the beginning of the memory block is returned or a null pointer is returned if the memory could not be allocated by the operating system.

It is necessary to release the memory with the [sc.FreeMemory](#) function when finished with the memory, or when [sc.LastCallToFunction](#) is true.

It is also necessary to release the memory when the DLL module containing the custom study is released. For more information, refer to [Modifying Advanced Custom Study Code](#).

Parameters

- **Size**: The number of bytes to allocate.

Example

sc.AngleInDegreesToSlope()

Type: Function

```
double AngleInDegreesToSlope(double AngleInDegrees);
```

Parameters

- **AngleInDegrees**:

Example

sc.ApplyStudyCollection()

Type: Function

```
int ApplyStudyCollection(int ChartNumber, const SCString& StudyCollectionName, const int ClearExistingStudiesFromChart);
```

The **sc.ApplyStudyCollection** function applies the specified [Study Collection](#) to the specified chart.

Returns 1 if there is no error. Returns 0 if **ChartNumber** is invalid. 1 will still be returned even if

StudyCollectionName is not found.

Parameters

- **ChartNumber**: The chart number to apply the Study Collection to. The chart numbers are displayed at the top line of the chart after the #. The chart must be in the same Chartbook containing the chart which contains the study instance which calls this function.
- **StudyCollectionName**: The Study Collection name. This must not include the path or the file extension (.StdyCollct).
- **ClearExistingStudiesFromChart**: This can be set to 1 to clear the existing studies from the chart or 0 to not clear the existing studies from the chart. In either case, there is no prompt when the Study Collection is applied to the chart.

Example

sc.ArmsEMV()

Type: Intermediate Study Calculation Function

This study function calculates the Arms Ease of Movement study.

```
SCFloatArrayRef ArmsEMV(SCBaseDataRef BaseDataIn, SCFloatArrayRef FloatArrayOut, int  
VolumeDivisor, int Index);
```

```
SCFloatArrayRef ArmsEMV(SCBaseDataRef BaseDataIn, SCFloatArrayRef FloatArrayOut, int  
VolumeDivisor); Auto-looping only.
```

The **sc.ArmsEMV()** function calculates the Arms Ease of Movement Value.

Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- **VolumeDivisor**: The Volume Divisor as an Integer.
- [Index](#).

Example

```
sc.ArmsEMV(sc.BaseDataIn, sc.Subgraph[0], 10);

float ArmsEMV = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.ArnaudLegouxMovingAverage()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **ArnaudLegouxMovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**, float **Sigma**, float **Offset**);

SCFloatArrayRef **ArnaudLegouxMovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**, float **Sigma**, float **Offset**); [Auto-looping only](#).

The **sc.ArnaudLegouxMovingAverage()** function calculates the Arnaud Legoux Moving Average study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).
- **Sigma:** This parameter partially controls the width of the Gaussian distribution of the weights. It does **not** play the role of a standard deviation. The standard deviation of the distribution is determined by Length/Sigma.
- **Offset:** This parameter partially controls the center of the Gaussian distribution of the weights. The center is determined by floor(Offset*(Length - 1)).

Example

```
sc.ArnaudLegouxMovingAverage(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 9, 6.0f, 0.5f);

float ArnaudLegouxMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.AroonIndicator()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **AroonIndicator**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **AroonIndicator**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.AroonIndicator()** function calculates the Aroon Indicator.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, 1 `sc.Subgraph[]`.Arrays[] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

Example

```
sc.AroonIndicator(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
float AroonIndicatorUp = sc.Subgraph[0][sc.Index]; //Access the Aroon Indicator Up study value at the current  
float AroonIndicatorDown = sc.Subgraph[0].Arrays[0][sc.Index]; //Access the Aroon Indicator Down study value at the current
```

sc.ArrayValueAtNthOccurrence()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **ArrayValueAtNthOccurrence**(SCFloatArrayRef **FloatArrayIn1**,
SCFloatArrayRef **FloatArrayIn2**, int **Index**, int **NthOccurrence**);

SCFloatArrayRef **ArrayValueAtNthOccurrence**(SCFloatArrayRef **FloatArrayIn1**,
SCFloatArrayRef **FloatArrayIn2**, int **NthOccurrence**); [Auto-looping only](#).

The **sc.ArrayValueAtNthOccurrence()** function iterates the **FloatArrayIn1** array for non-zero values starting at **Index** and going back. When the number of non-zero values found in **FloatArrayIn1** equals the number specified by **NthOccurrence**, then the value of **FloatArrayIn2** is returned at the **Index** where the **NthOccurrence** of non-zero values was found in **FloatArrayIn1**.

For an example, refer to the **scaf_ArrayValueAtNthOccurrenceSample** function in the `/ACS_Source/studies6.cpp` file.

Parameters

- [FloatArrayIn1](#).
- [FloatArrayIn2](#).
- [Index](#).
- **NthOccurrence**

Example

```
SCSubgraphRef ValueAtOccurrence = sc.Subgraph[1];
ValueAtOccurrence[sc.Index] = sc.ArrayValueAtNthOccurrence(StochasticData.Arrays[1],
StochasticData, NumberOfOccurrences.GetInt());
```

sc.ATR()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **ATR**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**, unsigned int **MovingAverageType**);

SCFloatArrayRef **ATR**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**, unsigned int **MovingAverageType**); [Auto-looping only](#).

SCFloatArrayRef **ATR**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut_1**, SCFloatArrayRef **FloatArrayOut_2**, int **Index**, int **Length**, unsigned int **MovingAverageType**);

SCFloatArrayRef **ATR**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut_1**, SCFloatArrayRef **FloatArrayOut_2**, int **Length**, unsigned int **MovingAverageType**);

The **sc.ATR()** function calculates the Average TRUE Range.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, 1 sc.Subgraph[].Arrays[] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).
- [MovingAverageType](#).
- [FloatArrayOut_1](#). This is the True Range output array. This is for the implementation of sc.ATR which does not require a SCSubgraphRef.
- [FloatArrayOut_2](#). This is the Average True Range output array. This is for the implementation of sc.ATR which does not require a SCSubgraphRef.

Example

```
sc.ATR(sc.BaseDataIn, sc.Subgraph[0], 20, MOVAVGTYPE_SIMPLE);

float ATR = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.AwesomeOscillator()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **AwesomeOscillator**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef

```
SubgraphOut, int Index, int Length1, int Length2);
```

```
SCFloatArrayRef AwesomeOscillator(SCFloatArrayRef FloatArrayIn, SCSubgraphRef  
SubgraphOut, int Length1, int Length2); Auto-looping only.
```

The **sc.AwesomeOscillator()** function calculates the Awesome Oscillator study.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, `sc.Subgraph[]`.Arrays[0-1] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length1](#).
- [Length2](#).

Example

```
SCSubgraphRef MovAvgDiff = sc.Subgraph[0];  
sc.AwesomeOscillator(sc.BaseDataIn[InputData.GetInputDataIndex()], MovAvgDiff, MA1Length.GetInt(), M
```

sc.BarIndexToRelativeHorizontalCoordinate()

Type: Function

```
int BarIndexToRelativeHorizontalCoordinate(int BarIndex);
```

Example

```
int X = sc.BarIndexToRelativeHorizontalCoordinate (sc.Index);
```

sc.BarIndexToXPixelCoordinate()

Type: Function

```
int BarIndexToXPixelCoordinate(int Index);
```

The **sc.BarIndexToXPixelCoordinate** function will calculate the corresponding X-axis pixel coordinate from the given chart bar **Index** .

The returned x-coordinate is in relation to the chart window itself.

Example

```
int X = sc.BarIndexToXPixelCoordinate (sc.Index);
```

sc.BollingerBands()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **BollingerBands** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**, float **Multiplier**, int **MovingAverageType**);

SCFloatArrayRef **BollingerBands** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Length**, float **Multiplier**, int **MovingAverageType**); [Auto-looping only](#).

The **sc.BollingerBands()** function calculates the Bollinger or Standard Deviation bands.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-1] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).
- [Multiplier](#).
- [MovingAverageType](#).

Example

```
sc.BollingerBands(sc.BaseData[SC_LAST], sc.Subgraph[0], 10, 1.8, MOVAVGTYPE_SIMPLE);

//Access the individual lines
float Average = sc.Subgraph[0][sc.Index]; //Access the study value at the current index

float UpperBand = sc.Subgraph[0].Arrays[0][sc.Index];

float LowerBand = sc.Subgraph[0].Arrays[1][sc.Index];

//Copy to Visible Subgraphs
sc.Subgraph[1][sc.Index] = UpperBand;
sc.Subgraph[2][sc.Index] = LowerBand;
```

sc.Butterworth2Pole()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Butterworth2Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **Butterworth2Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef

FloatArrayOut, int **Length**); [Auto-looping only](#).

The **sc.Butterworth2Pole()** function calculates a smoothing of data and is frequently used in the studies written by John Ehlers.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.Butterworth2Pole(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
  
float Butterworth2Pole = sc.Subgraph[0][sc.Index]; //Access the function value at the current index
```

sc.Butterworth3Pole()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Butterworth3Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **Butterworth3Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.Butterworth3Pole()** function calculates a smoothing of data and is frequently used in the studies written by John Ehlers.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.Butterworth3Pole(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
  
float Butterworth3Pole = sc.Subgraph[0][sc.Index]; //Access the function value at the current index
```

sc.CalculateAngle()

Type: Function

```
void CalculateAngle(SCFloatArrayRef InputArray, SCFloatArrayRef OutputArray, int Length, float ValuePerPoint);
```

```
void CalculateAngle(SCFloatArrayRef InputArray, SCFloatArrayRef OutputArray, int Index, int Length, float ValuePerPoint);
```

sc.CalculateAngle() calculates the angle from the horizontal of a given line defined by the starting position of InputArray[Index - Length], the ending position of InputArray[Index] and the ValuePerPoint. The results are contained within OutputArray at each Index point.

Parameters

- **InputArray**: The array of input values.
- **OutputArray**: The array of output values.
- **Index**: The Index of the ending point within the InputArray.
- **Length**: The size of the number of indices to go back to determine the starting point within the InputArray.
- **ValuePerPoint**: The slope of the line as Value per Index as defined in the InputArray.

sc.CalculateLogLogRegressionStatistics()

Type: Function

```
void CalculateLogLogRegressionStatistics (SCFloatArrayRef In, double& Slope, double& Y_Intercept, int Index, int Length);
```

```
void CalculateLogLogRegressionStatistics (SCFloatArrayRef In, double& Slope, double& Y_Intercept, int Length); Auto-looping only
```

Parameters

- [In](#).
- **Slope**: .
- **Y_Intercept**: .
- [Index](#).
- [Length](#).

Example

sc.CalculateOHLCAverages()

Type: Function

```
int CalculateOHLCAverages(int Index);
```

```
int CalculateOHLCAverages(); Auto-looping only.
```

sc.CalculateOHLCAverages() calculates the averages from the [sc.Subgraph\[\].Data\[\]](#) arrays **SC_OPEN** (0), **SC_HIGH** (1), **SC_LOW** (2), **SC_LAST** (3), and fills in the [sc.Subgraph\[\].Data\[\]](#) arrays for subgraphs **SC_OHLC** (4), **SC_HLC** (5), and **SC_HL** (6) for the elements at **Index**. You will want to use this if your study acts as a Main Price Graph. This would be the case when you set [sc.UsePriceGraphStyle](#) and [sc.DisplayAsMainPriceGraph](#) to 1 (TRUE).

Example

```
// Fill in the averages arrays  
sc.CalculateOHLCAverages(sc.Index);
```

sc.CalculateRegressionStatistics()

Type: Function

```
void CalculateRegressionStatistics (SCFloatArrayRef In, double& Slope, double&  
Y_Intercept, int Index, int Length);
```

```
void CalculateRegressionStatistics (SCFloatArrayRef In, double& Slope, double&  
Y_Intercept, int Length); Auto-looping only.
```

Parameters

- [In](#).
- **Slope**: .
- **Y_Intercept**: .
- [Index](#).
- [Length](#).

Example

sc.CalculateTimeSpanAcrossChartBars()

Type: Function

```
double CalculateTimeSpanAcrossChartBars(int FirstIndex, int LastIndex);
```

The **sc.CalculateTimeSpanAcrossChartBars** function calculates the time span across the chart

bars specified by the bar index parameters **FirstIndex** and **LastIndex**. The time length of the bar specified by **LastIndex** is also included in the time span.

If **FirstIndex** and **LastIndex** are the same, then the time length of the single specified bar will be what is returned.

The time span of a chart bar is determined to be the difference between the next bar's time and the bar's starting time. For the last bar in the chart, the time span is the difference between the last Date-Time contained within the chart bar and the bar's starting time.

However, the [Session Times](#) can affect the calculation of the time span for a bar. The very last bar just before the **Session Times >> End Time** will cause the time span to be limited to this ending time. The **Session Times >> Start Time** is not relevant or used in the calculations.

The return type is a double which is directly equivalent to a [SCDateTime](#) type and can be assigned to a SCDateTime type.

One useful purpose of this function is to determine if the particular span of time is sufficient enough for other calculations. This can be useful for detecting holidays.

sc.CalculateTimeSpanAcrossChartBarsInChart()

Type:: Function

```
void CalculateTimeSpanAcrossChartBarsInChart(int ChartNumber, int FirstIndex, int LastIndex, SCDateTime& TimeSpan);
```

The **sc.CalculateTimeSpanAcrossChartBarsInChart** function is nearly identical to the [sc.CalculateTimeSpanAcrossChartBars](#) function except that it allows a specific Chart to be referenced in the variable **ChartNumber** and returns the result in the referenced variable **TimeSpan**.

The **sc.CalculateTimespanAcrossChartBarsInChart** function calculates the time span across the chart bars specified by the bar index parameters **FirstIndex** and **LastIndex**. The time length of the bar specified by **LastIndex** is also included in the time span.

If **FirstIndex** and **LastIndex** are the same, then the time length of the single specified bar will be what is returned.

The time span of a chart bar is determined to be the difference between the next bar's time and the bar's starting time. For the last bar in the chart, the time span is the difference between the last Date-Time contained within the chart bar and the bar's starting time.

However, the [Session Times](#) can affect the calculation of the time span for a bar. The very last bar just before the **Session Times >> End Time** will cause the time span to be limited to this ending time. The **Session Times >> Start Time** is not relevant or used in the calculations.

The information is returned in the **TimeSpan** variable, which is an SCDateTime variable.

sc.CancelAllOrders()

Refer to the [sc.CancelAllOrders\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.CancelOrder()

Refer to the [sc.CancelOrder\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.CCI()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **CCI**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**, float **Multiplier**);

SCFloatArrayRef **CCI**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Length**, float **Multiplier**); [Auto-looping only](#).

The **sc.CCI()** function calculates the Commodity Channel Index.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[0].Arrays[0] (Extra Array) is used for internal calculations and additional results output.
- [Index](#).
- [Length](#).
- [Multiplier](#).

Example

```
// Subgraph 0 will contain the CCI output
sc.CCI(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20, 0.015);

float CCI = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.ChaikinMoneyFlow()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **ChaikinMoneyFlow**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **ChaikinMoneyFlow**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.ChaikinMoneyFlow()** function calculates the Chaikin Money Flow.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, `sc.Subgraph[]`.Arrays[0] (Extra Array) is used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

Example

```
sc.ChaikinMoneyFlow(sc.BaseDataIn, sc.Subgraph[0], 10);  
  
float ChaikinMoneyFlow = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.ChangeACSChartShortcutMenuItemText()

Refer to the [sc.ChangeACSChartShortcutMenuItemText\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

sc.ChangeChartReplaySpeed()

Type: Function

```
int ChangeChartReplaySpeed(int ChartNumber, float ReplaySpeed);
```

The **sc.ChangeChartReplaySpeed** function changes the replay speed for the replaying chart specified by the **ChartNumber** parameter.

It is only possible to specify Chart Numbers that are within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay speed is changed after the study function returns.

Parameters

- **ChartNumber**: The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To specify the same chart the study instance is on, use **sc.ChartNumber** for this parameter.
- **ReplaySpeed**: The replay speed. A speed of 1 is the same as real time.

Example

```
int Result = sc.ChangeChartReplaySpeed(sc.ChartNumber, 10);
```

sc.ChartDrawingExists()

Refer to the [sc.ChartDrawingExists\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

sc.ChartsDownloadingHistoricalData()

Type: Function

```
int ChartsDownloadingHistoricalData(int ChartNumber);
```

The **sc.ChartsDownloadingHistoricalData()** function returns TRUE (1) if the chart the study instance is on, is downloading historical chart data. This function returns FALSE(0) if historical chart data is not being downloaded even when the chart data is being read from the local data file.

This result can be useful when calling functions that play alert sounds. For example, if historical chart data is being downloaded, then avoid calling the `sc.PlaySound()` function because many alerts may be generated by your study function from historical data being downloaded.

It is normal that when a chart is first initially opened and historical chart data is not immediately downloaded, for this function to initially return FALSE when the study is initially calculated. Thereafter, the historical data will be downloaded if necessary.

This function should not be called frequently. Normally it is recommended to use it with [manual looping](#) and only call it at most once per study function call. Or in the case of [automatic looping](#), it should be called only in relation to the most recent chart bar or once during a full recalculation.

Parameters

- **ChartNumber:** The number of the chart to determine the historical data download status of. Normally this will be set to **sc.ChartNumber** to determine the historical data download status of the chart the study instance is applied to.

Example

```
int IsDownloading = sc.ChartsDownloadingHistoricalData(sc.ChartNumber);
SCString MessageText;
MessageText.Format("Downloading state: %d", IsDownloading);

sc.AddMessageToLog(MessageText.GetChars(), 0);

//Only play alert sound and add message if chart is not downloading historical data
if (IsDownloading == 0)
    sc.PlaySound(1,"My Alert Message");
```

sc.ClearAlertSoundQueue()

Type: Function

```
void ClearAlertSoundQueue();
```

sc.ClearAllPersistentData()

Type: Function

```
void ClearAllPersistentData();
```

The **sc.ClearAllPersistentData** function clears all persistent data variables which have been set through the [sc.GetPersistent*\(\)](#) or [sc.SetPersistent*\(\)](#) functions.

The persistent data is held in STL maps and these maps are fully cleared.

sc.ClearCurrentTradedBidAskVolume()

Type: Function

```
uint32_t ClearCurrentTradedBidAskVolume();
```

The **sc.ClearCurrentTradedBidAskVolume** function clears the current traded Bid and Ask volume for the symbol of the chart.

For further information, refer to [Current Traded Bid Volume](#).

sc.ClearCurrentTradedBidAskVolumeAllSymbols()

Type: Function

```
uint32_t ClearCurrentTradedBidAskVolumeAllSymbols();
```

The **sc.ClearCurrentTradedBidAskVolumeAllSymbols** function .

sc.ClearRecentBidAskVolume()

Type: Function

```
uint32_t ClearRecentBidAskVolume();
```

The **sc.ClearRecentBidAskVolume** function .

sc.ClearRecentBidAskVolumeAllSymbols()

Type: Function

```
uint32_t ClearRecentBidAskVolumeAllSymbols();
```

The **sc.ClearRecentBidAskVolumeAllSymbols** function .

sc.CloseChart()

Type: Function


```
void CloseChart(int ChartNumber);
```

The **sc.CloseChart()** function closes the chart specified by the **ChartNumber** parameter. The chart must exist within the same Chartbook that the custom study is also contained in.

For an example, refer to the **scsf_CloseChart** function in the **/ACS_Source/studies5.cpp** file in the Sierra Chart installation folder.

sc.CloseChartbook()

Type: Function

```
void CloseChartbook(const SCString& ChartbookFileName);
```

The **sc.CloseChartbook()** function closes the Chartbook specified by the **ChartbookFileName** filename parameter. The filename should not contain the path, only the filename.

For an example, refer to the **scsf_CloseChartbook** function in the **/ACS_Source/studies5.cpp** file in the Sierra Chart installation folder.

sc.CloseFile()

Type: Function

```
int CloseFile(const int FileHandle);
```

The **sc.CloseFile()** function closes the file with the File Handle, **FileHandle**, for the file which was opened with [sc.OpenFile\(\)](#).

The function returns **False** if there is an error closing the file, otherwise it returns **True**.

Also refer to [sc.OpenFile](#), [sc.CloseFile](#), [sc.ReadFile](#), [sc.WriteFile](#), [sc.GetLastFileErrorCode](#).

sc.CombinedForegroundColorBackgroundColorRef()

Type: Function

```
unsigned int CombinedForegroundColorBackgroundColorRef(COLORREF ForegroundColor,  
COLORREF BackgroundColor);
```

The **sc.CombinedForegroundColorBackgroundColorRef** function combines foreground and background color values into a single 32-bit value. This function is meant to be used to set the foreground and background colors through the [sc.Subgraph\[\].DataColor\[\]](#) array when displaying a table of values on a chart.

Since two 24-bit color values are combined into a 32 bit value, it reduces the color detail and the exact colors are not necessarily going to be represented.

Refer to the code example below.

```
sc.Subgraph[0].DataColor[sc.Index] = sc.CombinedForegroundBackgroundColorRef(COLOR_BLACK, COLOR_
```

sc.ConvertCurrencyValueToCommonCurrency()

Type: Function

```
double ConvertCurrencyValueToCommonCurrency(double CurrencyValue, const SCString&  
SourceSymbol, SCString &OutputCurrency);
```

The **sc.ConvertCurrencyValueToCommonCurrency** function takes the parameters of **CurrencyValue** and **SourceSymbol**, determines the underlying currency for the given **SourceSymbol** based upon the **Currency** field of the Symbol Settings for that symbol and then returns the equivalent value in the [Common Profit/Loss Currency](#) setting.

Typically the **SourceSymbol** would be set to `sc.Symbol`.

The **Common Profit/Loss Currency** setting must be set to a value other than **None** for this function to convert the price.

The **Common Profit/Loss Currency** symbol is also returned in the reference SCString variable **OutputCurrency**.

For additional information, refer to [Common Profit/Loss Currency](#).

sc.ConvertDateTimeFromChartTimeZone()

Type: Function

```
SCDateTime ConvertDateTimeFromChartTimeZone(const SCDateTime& DateTime, const  
char* TimeZonePOSIXString);
```

```
SCDateTime ConvertDateTimeFromChartTimeZone(const SCDateTime& DateTime,  
TimeZonesEnum TimeZone);
```

The **sc.ConvertDateTimeFromChartTimeZone()** function converts the **DateTime** parameter which is a [SCDateTime](#) variable which should be in the time zone of the chart the study instance is applied to, to the time zone specified by the **TimeZone** parameter.

The destination time zone can also be specified as a POSIX text string specified by the **TimeZonePOSIXString** parameter. For a list of these strings refer to the `TIME_ZONE_POSIX_STRINGS` array in the `/ACS_Source/SCConstants.h` file in the folder where Sierra Chart is installed to.

List of **TimeZone** constants that can be used:

- `TIMEZONE_HONOLULU`
- `TIMEZONE_ANCHORAGE`
- `TIMEZONE_LOS_ANGELES`

- TIMEZONE_PHOENIX_ARIZONA
- TIMEZONE_DENVER
- TIMEZONE_CHICAGO
- TIMEZONE_NEW_YORK
- TIMEZONE_HALIFAX
- TIMEZONE_UTC
- TIMEZONE_LONDON
- TIMEZONE_BRUSSELS
- TIMEZONE_CAPE_TOWN
- TIMEZONE_ATHENS
- TIMEZONE_BAGHDAD
- TIMEZONE_MOSCOW
- TIMEZONE_DUBAI
- TIMEZONE_ISLAMABAD
- TIMEZONE_NEW_DELHI
- TIMEZONE_DHAKA
- TIMEZONE_JAKARTA
- TIMEZONE_HONG_KONG
- TIMEZONE_TOKYO
- TIMEZONE_BRISBANE
- TIMEZONE_SYDNEY
- TIMEZONE_UTC_PLUS_12
- TIMEZONE_AUCKLAND

sc.ConvertDateTimeToChartTimeZone()

Type: Function

```
SCDateTime ConvertDateTimeToChartTimeZone(const SCDateTime& DateTime, const char*  
TimeZonePOSIXString);
```

```
SCDateTime ConvertDateTimeToChartTimeZone(const SCDateTime& DateTime,  
TimeZonesEnum TimeZone);
```

The **sc.ConvertDateTimeToChartTimeZone()** function converts the **DateTime** SCDateTime variable from the specified **TimeZone** to the Time Zone used by the chart the study instance is applied to. This can either be the global Time Zone or a chart specific Time Zone. For additional information, refer to [Time Zone](#).

The source time zone can also be specified as a POSIX text string. For a list of these strings refer to the TIME_ZONE_POSIX_STRINGS array in the /ACS_Source/SCConstants.h file in the folder where Sierra Chart is installed to.

List of **TimeZone** constants that can be used:

- TIMEZONE_HONOLULU
- TIMEZONE_ANCHORAGE
- TIMEZONE_LOS_ANGELES

- TIMEZONE_PHOENIX_ARIZONA
- TIMEZONE_DENVER
- TIMEZONE_CHICAGO
- TIMEZONE_NEW_YORK
- TIMEZONE_HALIFAX
- TIMEZONE_UTC
- TIMEZONE_LONDON
- TIMEZONE_BRUSSELS
- TIMEZONE_CAPE_TOWN
- TIMEZONE_ATHENS
- TIMEZONE_BAGHDAD
- TIMEZONE_MOSCOW
- TIMEZONE_DUBAI
- TIMEZONE_ISLAMABAD
- TIMEZONE_NEW_DELHI
- TIMEZONE_DHAKA
- TIMEZONE_JAKARTA
- TIMEZONE_HONG_KONG
- TIMEZONE_TOKYO
- TIMEZONE_BRISBANE
- TIMEZONE_SYDNEY
- TIMEZONE_UTC_PLUS_12
- TIMEZONE_AUCKLAND

Example

```
// Convert the DateTime from New York time to the time zone used for charts
DateTime = sc.ConvertDateTimeToChartTimeZone(DateTime, TIMEZONE_NEW_YORK);
DateTime.GetDateTimeYMDHMS(Year, Month, Day, Hour, Minute, Second);

// Write to the message log
SCString MessageString.Format("Converted from New York time: %d-%02d-%02d %02d:%02d:%02d", Year,
sc.AddMessageToLog(MessageString, 1);
```

sc.CreateDoublePrecisionPrice()

Type: Function

double **CreateDoublePrecisionPrice**(const float **PriceValue**);

Parameters

- **PriceValue**: .

Example

sc.CreateProfitLossDisplayString()

Type: Function

```
void CreateProfitLossDisplayString(double ProfitLoss, int Quantity,  
PositionProfitLossDisplayEnum ProfitLossFormat, SCString& Result);
```

The **sc.CreateProfitLossDisplayString()** function creates a Profit/Loss text string based on the provided parameters.

The **Result** parameter is passed by reference and receives the formatted Profit/Loss text string.

Parameters

- **ProfitLoss**: The profit or loss value as a floating-point number.
- **Quantity**: The Position or Trade quantity.
- **ProfitLossFormat**: The profit or loss format. The possibilities are:
PPLD_DO_NOT_DISPLAY, **PPLD_CURRENCY_VALUE**, **PPLD_POINTS**,
PPLD_POINTS_IGNORE_QUANTITY, **PPLD_TICKS**,
PPLD_TICKS_IGNORE_QUANTITY, **PPLD_CV_AND_POINTS**,
PPLD_CV_AND_POINTS_IGNORE_QUANTITY, **PPLD_CV_AND_TICKS**,
PPLD_CV_AND_TICKS_IGNORE_QUANTITY. For descriptions of these, refer to [Profit/Loss Format](#).
- **Result**: A [SCString](#) variable to receive the formatted profit/loss string.

Example

```
if (ShowOpenPL.GetYesNo())  
{  
    SCString PLString;  
    double Commission = RTCommissionRate * PositionData.PositionQuantity / 2.0;  
    sc.CreateProfitLossDisplayString(PositionData.OpenProfitLoss-Commission, PositionData.PositionQuan  
    Text.Format("Open PL: %s", PLString.GetChars());  
}
```

sc.CrossOver()

Type: Function

```
int CrossOver (SCFloatArrayRef First, SCFloatArrayRef Second, int Index);
```

```
int CrossOver (SCFloatArrayRef First, SCFloatArrayRef Second); Auto-looping only.
```

sc.CrossOver returns a value indicating if the **First** study Subgraph has crossed the **Second** subgraph at the array index specified by **Index**.

For an example, refer to the **scsf_StochasticCrossover** function in the /ACS_Source/Systems.cpp file in the Sierra Chart installation folder.

Function return values:

- **CROSS_FROM_TOP** - This constant value is returned when the **First** subgraph crosses the **Second** subgraph from the top.
- **CROSS_FROM_BOTTOM** - This constant value is returned when the **First** subgraph crosses the **Second** subgraph from the bottom.
- **NO_CROSS** - This constant value is returned when the **First** subgraph does not cross the **Second** subgraph.

Example

```
if (sc.Crossover(sc.Subgraph[3], sc.Subgraph[4]) == CROSS_FROM_BOTTOM)
{
    //Code
}

// This is an example of looking for a crossover between the values in
// a subgraph and a fixed value. We first have to put the fixed value
// into an array. We are using one of the Subgraph internal arrays.
sc.Subgraph[2].Arrays[8][sc.Index] = 100.0;

if (sc.Crossover(sc.Subgraph[2], sc.Subgraph[2].Arrays[8]) == CROSS_FROM_BOTTOM)
{
    //Code
}
```

Checking for a Crossover between an Array of Values and a Fixed Value

When you need to check for a crossover between a Subgraph array and a fixed value, you need to put the fixed value in an array first and pass this array to the sc.Crossover function. It is recommended to use one of the [sc.Subgraph. Arrays\[\]](#) for this purpose.

```
//Check for crossover of Subgraph 0 and a fixed value set into an extra array on the Subgraph
SCFloatArrayRef FixedValueArray = sc.Subgraph[0].Arrays[4];
FixedValueArray[sc.Index] = 100;

//Check if a crossover occurred
if(sc.Crossover(sc.Subgraph[0], FixedValueArray) != NO_CROSS)
{
}
}
```

sc.CumulativeDeltaTicks()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef CumulativeDeltaTicks(SCBaseDataRef BaseDataIn, SCSubgraphRef SubgraphOut, int Index, int ResetCumulativeCalculation);
```

```
SCFloatArrayRef CumulativeDeltaTicks(SCBaseDataRef BaseDataIn, SCSubgraphRef SubgraphOut, int ResetCumulativeCalculation); Auto-looping only.
```

The **sc.CumulativeDeltaTicks** function calculates the Cumulative Delta Ticks study and provides the Open, High, Low and Last values for each bar. This function can only be used on Intraday charts.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- **ResetCumulativeCalculation**: When this is set to TRUE, the cumulative calculations are reset back to 0. You may want to set this to TRUE at the **Index** which is at the start of a new trading day.

For an example of how to use a function of this type, refer to the **scsf_CumulativeDeltaBarsTicks** function in the **/ACS_Source/studies8.cpp** file in the Sierra Chart installation folder.

sc.CumulativeDeltaTickVolume()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef CumulativeDeltaTickVolume(SCBaseDataRef BaseDataIn, SCSubgraphRef SubgraphOut, int Index, int ResetCumulativeCalculation);
```

```
SCFloatArrayRef CumulativeDeltaTickVolume(SCBaseDataRef BaseDataIn, SCSubgraphRef SubgraphOut, int ResetCumulativeCalculation); Auto-looping only.
```

The **sc.CumulativeDeltaTickVolume** function calculates the Cumulative Delta Up/Down Tick Volume study and provides the Open, High, Low and Last values for each bar. This function can only be used on Intraday charts.

For an example of how to use a function of this type, refer to the **scsf_CumulativeDeltaBarsTicks** function in the **/ACS_Source/studies8.cpp** file in the Sierra Chart installation folder.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- **ResetCumulativeCalculation**: When this is set to TRUE, the cumulative calculations are reset back to 0. You may want to set this to TRUE at the **Index** which is at the start of a new trading day.

sc.CumulativeDeltaVolume()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **CumulativeDeltaVolume**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **ResetCumulativeCalculation**);

SCFloatArrayRef **CumulativeDeltaVolume**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **ResetCumulativeCalculation**); [Auto-looping only](#).

The **sc.CumulativeDeltaVolume** function calculates the Cumulative Delta Volume study and provides the Open, High, Low and Last values for each bar. This function can only be used on Intraday charts.

For an example of how to use a function of this type, refer to the **scsf_CumulativeDeltaBarsTicks** function in the **/ACS_Source/studies8.cpp** file in the Sierra Chart installation folder.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- **ResetCumulativeCalculation**: When this is set to TRUE, the cumulative calculations are reset back to 0. You may want to set this to TRUE at the **Index** which is at the start of a new trading day.

sc.CumulativeSummation()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **CumulativeSummation** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**);

SCFloatArrayRef **CumulativeSummation** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**); [Auto-looping only](#).

The **sc.CumulativeSummation** function calculates the summation of all of the elements in **FloatArrayIn** up to the current **Index**. Therefore, this function is cumulative in that the summation

is across all of the elements in **FloatArrayIn** from the beginning.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

Example

```
sc.CumulativeSummation(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0]);  
  
float CumulativeSummation = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.CyberCycle()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **CyberCycle**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArraySmoothed**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **CyberCycle**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArraySmoothed**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.CyberCycle()** function calculates Ehlers' Cyber Cycle study.

Parameters

- [FloatArrayIn](#).
- **FloatArraySmoothed**: This array holds the smoothed data that is used in the study calculation.
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.CyberCycle(sc.BaseDataIn[SC_LAST], Array_SmoothedData, sc.Subgraph[0], 28);  
  
float CyberCycle = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.DataTradeServiceName()

Type: Function

```
SCString DataTradeServiceName();
```

The **sc.DataTradeServiceName()** function returns the text name of the **Service** currently set in **Global Settings >> Data/trade Service Settings**. The return type is a SCString.

sc.DatesToExcludeClear()

Type: Function

```
int sc.DatesToExcludeClear(const int ChartNumber);
```

The **sc.DatesToExcludeClear()** function

Parameters

- **ChartNumber:** .

Example

sc.DateStringDDMMYYYYToSCDateTime()

Type: Function

```
double sc.DateStringDDMMYYYYToSCDateTime(const SCString& DateString);
```

The **sc.DateStringDDMMYYYYToSCDateTime()** function converts the **DateString** text string parameter to an [SCDateTime](#).

DateString is a [SCString](#) type.

The return type is a double data type directly compatible with a SCDateTime and can be assigned to a SCDateTime.

Parameters

- **DateString:** The date string in the format DDMMYYYY (Day, Month, Year).

Example

```
SCDateTime Date = sc.DateStringDDMMYYYYToSCDateTime("05012017");
```

sc.DateStringToSCDateTime()

Type: Function

```
SCDateTime DateStringToSCDateTime(const SCString& DateString);
```

The **sc.DateStringToSCDateTime()** function converts the **DateString** text string to an **SCDateTime** variable. This function only works with dates.

Example

```
SCString DateString ("2011-12-1");  
SCDateTime DateValue;  
DateValue = sc.DateStringToSCDateTime(DateString);
```

sc.DateTimeToString()

Type: Function

```
SCString DateTimeToString(const double& DateTime, int Flags);
```

The **sc.DateTimeToString** function will convert the given **DateTime** variable to a text string. The format is specified by the **Flags** parameter.

The flags can be any of the following:

- FLAG_DT_YEAR
- FLAG_DT_MONTH
- FLAG_DT_DAY
- FLAG_DT_HOUR
- FLAG_DT_MINUTE
- FLAG_DT_SECOND
- FLAG_DT_PLUS_WITH_TIME
- FLAG_DT_NO_ZERO_PADDING_FOR_DATE
- FLAG_DT_HIDE_SECONDS_IF_ZERO
- FLAG_DT_NO_HOUR_PADDING
- FLAG_DT_MILLISECOND
- FLAG_DT_COMPACT_DATE
- FLAG_DT_COMPLETE_DATE
- FLAG_DT_COMPLETE_TIME
- FLAG_DT_COMPLETE_DATETIME
- FLAG_DT_COMPLETE_DATETIME_MS

Example

```

if(sc.Index == sc.ArraySize - 1)
{
    // Log the current time
    SCString DateTimeString = sc.DateTimeToString(sc.CurrentSystemDateTime,FLAG_DT_COMPLETE_D

    sc.AddMessageToLog(DateTimeString, 0);
}

```

sc.DateTimeToString()

Type: Function

SCString DateTimeToString(const SCDatetime& DateTime);

The **sc.DateTimeToString** function returns a text string for the date within the given **DateTime** parameter. Any time component in the given **DateTime** parameter will be ignored.

Example

```

if (sc.Index == sc.ArraySize - 1)
{
    // Log the current date.
    SCString DateTimeString = sc.DateTimeToString(sc.CurrentSystemDateTime);

    sc.AddMessageToLog(DateTimeString, 0);
}

```

sc.DeleteACSChartDrawing()

Refer to the [sc.DeleteACSChartDrawing\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

sc.DeleteLineUntilFutureIntersection()

Type: Function

int DeleteLineUntilFutureIntersection(int **StartBarIndex**, int **LineIDForBar**);

The **sc.DeleteLineUntilFutureIntersection** function deletes a line added by the [sc.AddLineUntilFutureIntersection](#) function.

Parameters

- **StartBarIndex:** This is the array index of the chart bar which was previously specified to the **sc.AddLineUntilFutureIntersection()** function, for the line to delete.

- **LineIDForBar**: This is the identifier of the extension line for the chart bar which was previously specified by the **sc.AddLineUntilFutureIntersection()** function, for the line to delete.

sc.DeleteUserDrawnACSDrawing()

Refer to the [sc.DeleteUserDrawnACSDrawing\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

sc.Demarker()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Demarker**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **Demarker**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.Demarker()** function calculates the Demarker study.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[.Arrays[0-4] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

Example

```
sc.Demarker(sc.BaseDataIn, sc.Subgraph[0], 10);
float Demarker = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.Dispersion()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Dispersion**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **Dispersion**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.Dispersion()** function calculates the Dispersion study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.Dispersion(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0], 10);  
  
float Dispersion = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.DMI()

Type: Intermediate Study Calculation Function

DMI(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

DMI(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.DMI()** function calculates the Directional Movement Index study.

Parameters

- [BaseDataIn](#): sc.BaseDataIn input arrays.
- [SubgraphOut](#). For this function, sc.Subgraph[0].Arrays[0-4] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

Example

```
sc.DMI(sc.BaseDataIn, sc.Subgraph[0], 10);  
  
//Access the individual study values  
float DMIPositive = sc.Subgraph[0][sc.Index];  
  
float DMINegative = sc.Subgraph[0].Arrays[0][sc.Index];  
  
//Copy DMINegative to a visible Subgraph  
sc.Subgraph[1][sc.Index] = DMINegative;
```

sc.DMIDiff()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **DMIDiff** (SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **DMIDiff** (SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.DMIDiff()** function calculates the Directional Movement Index Difference study.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

Example

```
sc.DMIDiff(sc.BaseDataIn, sc.Subgraph[0], 10);  
  
float DMIDiff = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.DominantCyclePeriod()

Type: Intermediate Study Calculation Function.

SCFloatArrayRef **sc.DominantCyclePeriod** (SCFloatArrayRef In, SCSubgraphRef Out, int IndexParam, int MedianLength);

The **sc.DominantCyclePeriod()** function calculates the Dominant Cycle Period and is frequently used in the studies written by John Ehlers.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-6] (Extra Arrays) are used for internal calculations.
- [IndexParam](#).
- [MedianLength](#).

Example

```
sc.DominantCyclePeriod(sc.BaseData[SC_LAST], sc.Subgraph[0], 5);
```

sc.DominantCyclePhase()

Type: Intermediate Study Calculation Function.

SCFloatArrayRef **sc.DominantCyclePhase** (SCFloatArrayRef In, SCSubgraphRef Out, int Index);

The **sc.DominantCyclePhase()** function calculates the Dominant Cycle Phase and is frequently used in the studies written by John Ehlers.

Parameters

- [FloatArrayIn](#): .
- [SubgraphOut](#): For this function, sc.Subgraph[].Arrays[0-6] (Extra Arrays) are used for internal calculations.
- [Index](#): .

Example

```
sc.DominantCyclePhase(sc.BaseData[SC_LAST], sc.Subgraph[0]);
```

sc.DoubleStochastic()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **DoubleStochastic** (SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**, int **MovAvgLength**, int **MovingAverageType**);

SCFloatArrayRef **DoubleStochastic** (SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**, int **MovAvgLength**, int **MovingAverageType**); [Auto-looping only](#).

The **sc.DoubleStochastic()** function calculates the Double Stochastic study.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).
- [MovAvgLength](#).
- [MovingAverageType](#).

Example


```
sc.DoubleStochastic(sc.BaseData, sc.Subgraph[0], Length.GetInt(), MovAvgLength.GetInt(), MovAvgType.
//Access the study value at the current index
float DoubleStochastic = sc.Subgraph[0][sc.Index];
```

sc.EnvelopeFixed()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **EnvelopeFixed**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, float **FixedValue**, int **Index**);

SCFloatArrayRef **EnvelopeFixed**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, float **FixedValue**); [Auto-looping only](#).

The **sc.EnvelopeFixed()** function calculates the Fixed Envelope study.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0] (Extra Array) is used for internal calculations and additional results output.
- **FixedValue**: This is the amount added and subtracted to **FloatArrayIn** at the current **Index**.
- [Index](#).

Example

```
sc.EnvelopeFixed(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 3.5);
//Access the individual study values at the current index
float EnvelopeTop = sc.Subgraph[0][sc.Index];
float EnvelopeBottom = sc.Subgraph[0].Arrays[0][sc.Index];
```

sc.EnvelopePct()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **EnvelopePct**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, float **Percent**, int **Index**);

SCFloatArrayRef **EnvelopePct**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, float **Percent**); [Auto-looping only](#).

The **sc.EnvelopePct()** function calculates the Percentage Envelope study.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, `sc.Subgraph[]`.Arrays[0] (Extra Array) is used for internal calculations and additional results output.
- [Percent](#).
- [Index](#).

Example

```
sc.EnvelopePct(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0], 0.05);

//Access the individual study values at the current index
float Top = sc.Subgraph[0][sc.Index];

float Bottom = sc.Subgraph[0].Arrays[0][sc.Index];

//Copy to Visible Subgraphs
sc.Subgraph[1][sc.Index] = Bottom;
```

sc.Ergodic()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Ergodic**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **LongEMALength**, int **ShortEMALength**, float **Multiplier**);

SCFloatArrayRef **Ergodic**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **LongEMALength**, int **ShortEMALength**, float **Multiplier**); [Auto-looping only](#).

The **sc.Ergodic()** function calculates the True Strength Index.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, `sc.Subgraph[]`.Arrays[0-5] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [LongEMALength](#).
- [ShortEMALength](#).
- [Multiplier](#).

Example

```

sc.Ergodic(
    sc.BaseDataIn[SC_LAST],
    sc.Subgraph[0],
    15, // Long EMA length
    3, // Short EMA length
    1.0f, // Multiplier
);

// You can calculate the signal line and oscillator with the following code:

// Calculate the signal line with an exponential moving average with a length of 10
sc.ExponentialMovAvg(sc.Subgraph[0], sc.Subgraph[1], 10);

// Calculate the oscillator
sc.Subgraph[2][sc.Index] = sc.Subgraph[0][sc.Index] - sc.Subgraph[1][sc.Index];

```

sc.EvaluateAlertConditionFormulaAsBoolean()

Type: Function

```
int EvaluateAlertConditionFormulaAsBoolean(int BarIndex, int ParseAndSetFormula);
```

The **sc.EvaluateAlertConditionFormulaAsBoolean()** function evaluates the Alert formula entered on the study that calls this function, and returns a 1 (true) or 0 (false) depending upon whether the formula is true or false at the specified **BarIndex**.

When the **ParseAndSetFormula** parameter is set to a nonzero value, then the Alert condition is internally formatted and stored prior to testing the condition at the BarIndex. It is necessary to set **ParseAndSetFormula** to a nonzero value the first time running this function or whenever the alert formula changes. It should not be set to a nonzero value every time because it is not efficient doing that.

Parameters

- **BarIndex:** Integer index value of the bar to evaluate the alert condition formula on.
- **ParseAndSetFormula:** Integer variable that when set to any value other than 0, forces an internal format and storage of the alert condition formula prior to evaluation.

sc.EvaluateGivenAlertConditionFormulaAsBoolean()

Type: Function

```
int EvaluateGivenAlertConditionFormulaAsBoolean(int BarIndex, int ParseAndSetFormula, const SCString& Formula);
```

The **sc.EvaluateGivenAlertConditionFormulaAsBoolean()** function evaluates the given **Formula** parameter, and returns a 1 (true) or 0 (false) depending upon whether the given Formula at the specified **BarIndex** is true or false.

When the **ParseAndSetFormula** parameter is set to a nonzero value, then the **Formula** is internally formatted and stored prior to testing the condition at the **BarIndex**. It is necessary to set **ParseAndSetFormula** to a nonzero value the first time running this function or whenever the Formula changes. It should not be set to a nonzero value every time because it is not efficient doing that.

Parameters

- **BarIndex**: Integer index value of the bar to evaluate the Formula on.
- **ParseAndSetFormula**: Integer variable that when set to any value other than 0, forces an internal format and storage of the **Formula** parameter prior to evaluation.
- **Formula**: The formula to evaluate as a text string. This only needs to be provided when **ParseAndSetFormula** is a nonzero value. Otherwise, it can be an empty string just by passing "". The formula needs to follow the format documented in the [Alert Condition Formula Format](#) section.

sc.EvaluateGivenAlertConditionFormulaAsDouble()

Type: Function

```
double EvaluateGivenAlertConditionFormulaAsDouble(int BarIndex, int ParseAndSetFormula, const SCString& Formula);
```

The **sc.EvaluateGivenAlertConditionFormulaAsDouble()** function evaluates the given **Formula** parameter, and returns the calculated value for the formula at the specified **BarIndex**.

When the **ParseAndSetFormula** parameter is set to a nonzero value, then the **Formula** is internally formatted and stored prior to calculating the formula at the **BarIndex**. It is necessary to set **ParseAndSetFormula** to a nonzero value the first time running this function or whenever the Formula changes. It should not be set to a nonzero value every time because it is not efficient doing that.

Parameters

- **BarIndex**: Integer index value of the bar to evaluate the Formula on.
- **ParseAndSetFormula**: Integer variable that when set to any value other than 0, forces an internal format and storage of the **Formula** parameter prior to evaluation.
- **Formula**: The formula to evaluate as a text string. This only needs to be provided when **ParseAndSetFormula** is a nonzero value. Otherwise, it can be an empty string just by passing "". For information about the formula format, refer to [Spreadsheet Formula](#).

sc.ExponentialMovAvg()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef ExponentialMovAvg(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef ExponentialMovAvg(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.ExponentialMovAvg()** function calculates the exponential moving average.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.ExponentialMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);  
  
float ExponentialMovAvg = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.ExponentialRegressionIndicator()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef ExponentialRegressionIndicator(SCFloatArrayRef FloatArrayIn,  
SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef ExponentialRegressionIndicator(SCFloatArrayRef FloatArrayIn,  
SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.ExponentialRegressionIndicator()** function calculates the Exponential Regression Indicator study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.ExponentialRegressionIndicator(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
float ExponentialRegressionIndicator = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.FillSubgraphElementsWithLinearValuesBetweenBeginEndValues

Type: Function

```
void FillSubgraphElementsWithLinearValuesBetweenBeginEndValues(int SubgraphIndex,  
int BeginIndex, int EndIndex);
```

The **sc.FillSubgraphElementsWithLinearValuesBetweenBeginEndValues()** function fills the study Subgraph array specified by the **SubgraphIndex** parameter from the element after **BeginIndex** to the element just before the **EndIndex** parameters, with the linear values in between the values specified by those index parameters. **SubgraphIndex** is 0 based, therefore 0 is the first Subgraph.

For example, if the value at BeginIndex is 2.0 and the index is 10, and the value at EndIndex is 2.4 and the index is 14, then at indices 11, 12, and 13 the set values will be 2.1, 2.2, and 2.3 respectively.

sc.FlattenAndCancelAllOrders()

Refer to the [sc.FlattenAndCancelAllOrders\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.FlattenPosition()

Refer to the [sc.FlattenPosition\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.FormatDateTime()

Type: Function

```
SCString FormatDateTime(const SCDateTime& DateTime);
```

The **sc.FormatDateTime()** function returns a formatted text string for the given **DateTime** parameter. The returned text string will contain both the date and the time.

Example

```
SCString Message;

Message.Format("Current Bar Date-Time: %s", sc.FormatDateTime(sc.BaseDateTimeIn[sc.Index]).GetCha
sc.AddMessageToLog(Message, 0);
```

sc.FormatDateTimeMS()

Type: Function

SCString **FormatDateTimeMS**(const SCDatetime& **DateTimeMS**);

The **sc.FormatDateTimeMS()** function returns a formatted text string for the given **DateTimeMS** parameter. The returned text string will contain both the date and the time, including milliseconds.

Example

```
SCString Message;

Message.Format("Current System Date-Time: %s", sc.FormatDateTimeMS(sc.CurrentSystemDateTimeMS
sc.AddMessageToLog(Message, 0);
```

sc.FormatGraphValue()

Type: Function

SCString **FormatGraphValue**(double **Value**, int **ValueFormat**);

sc.FormatGraphValue() formats a numeric value as text based on the specified value format. The text is returned as an SCString.

Parameters

- **Value:** The Integer or floating-point value to format.
- **ValueFormat:** The formatting code. Can be **sc.BaseGraphValueFormat**. This number sets the number of decimal places to display. If ValueFormat is greater than 100, then it will be a fractional format where the denominator is specified as **Denominator + 100** . For example, 132 will format the value in a fractional format using a denominator of 1/32.

Example

```
SCString CurHigh = sc.FormatGraphValue(sc.BaseData[SC_HIGH][CurrentVisibleIndex], sc.BaseGraphVa
s_UseTool Tool;
Tool.Text = CurHigh;
```

sc.FormatString()

Type: Function

SCString& **FormatString**(SCString& **Out**, const char* **Format**, [Variable list of parameters]);

The **sc.FormatString()** function creates a text string using the specified **Format**. For more information, refer to the [Setting A Name To A Formatted String](#) section. **Out** is the SCString you need to provide where the text output is copied to.

Example

```
SCString Output;
sc.FormatString(Output, "The result is: %f", sc.Subgraph[0].Data[sc.Index]);
```

sc.FormattedEvaluate()

Type: Function

int **FormattedEvaluate**(float **Value1**, int **Value1Format**, OperatorEnum **Operator**, float **Value2**, int **Value2Format**, float **PrevValue1** = 0.0f, float **PrevValue2** = 0.0f);

sc.FormattedEvaluate() evaluates the relationship between 2 floating-point numbers using the specified **Operator** and Value Formats (**Value1Format**, **Value2Format**). The evaluation is performed as follows: **Value1 Operator Value2**.

The more precise of the two Value Formats is used for the comparison. Once this is determined, half of the more precise Value Format is calculated. If the Value Format is 2 which is equivalent to .01, then the tolerance is .005. In this particular example, two values would be considered equal if the difference between them is less than .005.

Another example: Assuming the most precise Value Format is .01, Value1 will be considered less than Value2 if the difference between the two of them is less than -.005.

The function returns 1 if the evaluation is TRUE, and 0 if it is FALSE.

The reason why this function needs to be used when performing floating-point number comparisons is due to what is known as floating point error. Refer to **Accuracy Problems** in the [Floating Point Wikipedia article](#). For example, the number .01 may internally be represented in a single precision floating-point variable as .0099998, therefore making comparisons using the built-in operators of the C++ language not accurate.

Parameters

- **Value1**: Any number to use on the left side of the **Operator**.
- **Value1Format**: The formatting code. Can be set to **sc.BaseGraphValueFormat** or **sc.GetValueFormat()**. For more information, refer to [sc.ValueFormat](#).
- **Operator**:
 - NOT_EQUAL_OPERATOR
 - LESS_EQUAL_OPERATOR
 - GREATER_EQUAL_OPERATOR
 - CROSS_EQUAL_OPERATOR
 - CROSS_OPERATOR
 - EQUAL_OPERATOR
 - LESS_OPERATOR
 - GREATER_OPERATOR
- **Value2**: Any number to use on the right side of the **Operator**.
- **Value2Format**: The formatting code. Can be set to **sc.BaseGraphValueFormat** or **sc.GetValueFormat()**. For more information, refer to [sc.ValueFormat](#).
- **PrevValue1**: This only needs to be specified when using the **CROSS_OPERATOR** operator. In this case, provide the prior value (usually from a line) to use on the left side of the operator in the comparison.
- **PrevValue2**: This only needs to be specified when using the **CROSS_OPERATOR** operator. In this case, provide the prior value (usually from a line) to use on the right side of the operator in the comparison.

Example

```
int Return = sc.FormattedEvaluate(CurrentClose, sc.BaseGraphValueFormat, LESS_OPERATOR, PriorLow
```

sc.FormattedEvaluateUsingDoubles()

Type: Function

```
int FormattedEvaluateUsingDoubles(double Value1, int Value1Format, OperatorEnum  
Operator, double Value2, int Value2Format, double PrevValue1 = 0.0f, double PrevValue2 =  
0.0f);
```

sc.FormattedEvaluateUsingDoubles() evaluates the relationship between 2 double precision floating-point numbers using the specified **Operator** and Value Formats (**Value1Format**, **Value2Format**).

This function is identical to [sc.FormattedEvaluate\(\)](#) except that it uses double precision floating-point numbers instead of single precision floating-point numbers.

For additional details, refer to [sc.FormattedEvaluate\(\)](#).

sc.FormatVolumeValue()

Type: Function

```
SCString sc.FormatVolumeValue(int64_t Volume, int VolumeValueFormat, int
UseLargeNumberSuffix);
```

The **sc.FormatVolumeValue()** function formats an integer volume value into a text string. The result is returned as a [SCString](#) type.

Parameters

- **Volume:** The volume value as an integer.
- **VolumeValueFormat:** The Volume Value Format. Normally pass the `sc.VolumeValueFormat` variable as the parameter.
- **UseLargeNumberSuffix:** 1 to use a large number suffix or 0 not to. For example, in the case of volume values of 1000000 or higher, the M suffix will be used when this is set to 1.

sc.FourBarSymmetricalFIRFilter()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef FourBarSymmetricalFIRFilter(SCFloatArrayRef FloatArrayIn,
SCFloatArrayRef FloatArrayOut, int Index);
```

```
SCFloatArrayRef FourBarSymmetricalFIRFilter(SCFloatArrayRef FloatArrayIn,
SCFloatArrayRef FloatArrayOut); Auto-looping only.
```

The **sc.FourBarSymmetricalFIRFilter()** function calculates a four-bar smoothing of data and is frequently used in the studies written by John Ehlers.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

Example

```
sc.FourBarSymmetricalFIRFilter(sc.BaseDataIn[SC_LAST], sc.Subgraph[0]);

float FourBarSymmetricalFIRFilter = sc.Subgraph[0][sc.Index]; //Access the function value at the current inc
```

sc.FreeMemory()

Type: Function

```
void FreeMemory (void* Pointer);
```

The **sc.FreeMemory** function releases the memory allocated with the [sc.AllocateMemory](#) function.

Parameters

- **Pointer:** The pointer to the memory block to release.

sc.GetACSDrawingByIndex()

Type: Function

For complete documentation for the **sc.GetACSDrawingByIndex** function, refer to [sc.GetACSDrawingByIndex\(\)](#).

sc.GetACSDrawingByLineNumber()

Refer to the [sc.GetACSDrawingByLineNumber\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

sc.GetACSDrawingsCount()

Refer to the [sc.GetACSDrawingsCount\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

sc.GetAskMarketDepthEntryAtLevel

Type: Function

```
int GetAskMarketDepthEntryAtLevel(s_MarketDepthEntry& DepthEntry, int LevelIndex);
```

sc.GetAskMarketDepthEntryAtLevel returns in the **DepthEntry** structure of type **s_MarketDepthEntry**, the ask side market depth data for the level specified by the **LevelIndex** variable for the symbol of the chart.

The function returns 1 if the function call was successful, otherwise 0.

To receive Market Depth data it is also necessary to set [sc.UsesMarketDepthData](#) to 1 in the study function. This normally should be placed in the **sc.SetDefaults** code block.

For an example to access market depth data, refer to the **scsf_DepthOfMarketData()** function in the **/ACS_Source/studies.cpp** file in the folder where Sierra Chart is installed to.

The **s_MarketDepthEntry** structure contains a **Price** member and an **AdjustedPrice** member. Normally these are the same price. However, when a Real-time Price Multiplier is set to a value

other than 1.0 in a chart, then Price will contain the original unadjusted price and AdjustedPrice will contain the Price multiplied by the Real-time Price Multiplier.

s_MarketDepthEntry

```
struct s_MarketDepthEntry
{
    float Price;
    t_MarketDataQuantity Quantity;
    uint32_t NumOrders;
    float AdjustedPrice;
}
```

sc.GetAskMarketDepthEntryAtLevelForSymbol

Type: Function

```
int GetAskMarketDepthEntryAtLevelForSymbol(const SCString& Symbol,
s_MarketDepthEntry& r_DepthEntry, int LevelIndex);
```

sc.GetAskMarketDepthEntryAtLevelForSymbol returns in the **DepthEntry** structure of type **s_MarketDepthEntry**, the ask side market depth data for the level specified by the **LevelIndex** variable for the specified **Symbol**.

The function returns 1 if the function call was successful, otherwise 0.

To receive Market Depth data it is also necessary to set [sc.UsesMarketDepthData](#) to 1 in the study function. This normally should be placed in the **sc.SetDefaults** code block.

For an example to access market depth data, refer to the **scsf_DepthOfMarketData()** function in the **/ACS_Source/studies.cpp** file in the folder where Sierra Chart is installed to.

s_MarketDepthEntry

```
struct s_MarketDepthEntry
{
    float Price;
    t_MarketDataQuantity Quantity;
    uint32_t NumOrders;
    float AdjustedPrice;
}
```

sc.GetAskMarketDepthNumberOfLevels

Type: Function

```
int GetAskMarketDepthNumberOfLevels();
```

The **sc.GetAskMarketDepthNumberOfLevels** function returns the number of available market depth levels on the Ask side for the symbol of the chart.

There are settings which affect the number of market depth levels. Refer to [Not Receiving the Full Number of Market Depth Levels](#).

sc.GetAskMarketDepthNumberOfLevelsForSymbol

Type: Function

```
int GetAskMarketDepthNumberOfLevelsForSymbol(const SCString& Symbol);
```

The **sc.GetAskMarketDepthNumberOfLevelsForSymbol** function returns the number of available market depth levels on the Ask side for the specified symbol.

There are settings which affect the number of market depth levels. Refer to [Not Receiving the Full Number of Market Depth Levels](#).

Parameters

- **Symbol:** The symbol to get the number of market depth levels for.

Example

```
int NumberOfLevels = GetBidMarketDepthNumberOfLevelsForSymbol(sc.Symbol);
```

sc.GetAskMarketDepthStackPullSum()

sc.GetBidMarketDepthStackPullSum()

Type: Function

```
double GetAskMarketDepthStackPullSum();
```

```
double GetBidMarketDepthStackPullSum();
```

The **sc.GetAskMarketDepthStackPullSum** and **sc.GetBidMarketDepthStackPullSum** functions return the current total sum of the [market depth data stacking or pulling values](#) for either the Ask or Bid side respectively.

For these functions to return a value, it is necessary that

Trade >> Trading Chart DOM On / Show Market Data Columns is enabled for one of the charts for the Symbol. And one of the **Pulling/Stacking** columns needs to be added. Refer to [Customize Trade/Chart DOM Columns and Descriptions](#) for instructions.

Example

```
// Get the current stacking and pulling sum on the Ask side
int MarketDepthStackPullSum = sc.GetAskMarketDepthStackPullSum();
```

sc.GetAskMarketDepthStackPullValueAtPrice()

sc.GetBidMarketDepthStackPullValueAtPrice()

Type: Function

```
int GetAskMarketDepthStackPullValueAtPrice(float Price);
```

```
int GetBidMarketDepthStackPullValueAtPrice(float Price);
```

The **sc.GetAskMarketDepthStackPullValueAtPrice** and **sc.GetBidMarketDepthStackPullValueAtPrice** functions return the current [market depth data stacking or pulling value](#) for either the Ask or Bid side respectively, for the given **Price** parameter.

For these functions to return a value, it is necessary that

Trade >> Trading Chart DOM On / Show Market Data Columns is enabled for one of the charts for the Symbol. And one of the **Pulling/Stacking** columns needs to be added. Refer to [Customize Trade/Chart DOM Columns and Descriptions](#) for instructions.

Also refer to [sc.GetAskMarketDepthStackPullSum\(\)](#) and [sc.GetBidMarketDepthStackPullSum\(\)](#).

Example

```
// Get the current stacking and pulling value on the bid side for the current best bid price.
int StackPullValue = sc.GetBidMarketDepthStackPullValueAtPrice(sc.Bid);
```

sc.GetAskMarketLimitOrdersForPrice()

Type: Function

```
int GetAskMarketLimitOrdersForPrice(const int PricInTicks, const int ArraySize,
n_ACSIL::s_MarketOrderData* p_MarketOrderDataArray);
```

The **sc.GetAskMarketLimitOrdersForPrice** function is used to get the order ID and order quantity for working limit orders from the market data feed at the specified price level. This data is called [Market by Order](#) data. This includes all the working limit orders as provided by the market data feed.

Parameters

- **PricInTicks:** This is the price to get the working limit orders for. This is an integer value. You need to take the floating-point actual price and divide by

sc.TickSize and round to the nearest integer.

- **ArraySize**: This is the size of the `p_MarketOrderDataArray` array as a number of elements, that you provide the function.
- **p_MarketOrderDataArray**: This is a pointer to the array of type `n_ACSIL::s_MarketOrderData` of the size specified by **ArraySize** that is filled in with the working limit orders data upon return of the function.

Example

For an example to use this function, refer to the **scsf_MarketLimitOrdersForPriceExample** study function in the `/ACS_Source/Studies2.cpp` file in the Sierra Chart installation folder.

sc.GetAttachedOrderIDsForParentOrder()

Type: Function

```
void GetAttachedOrderIDsForParentOrder(int ParentInternalOrderID, int&
TargetInternalOrderID, int& StopInternalOrderID);
```

The **sc.GetAttachedOrderIDsForParentOrder** function is used to get the Internal Order IDs for a Target and Stop order for a given Parent Internal Order ID.

The **ParentInternalOrderID** parameter specifies the Parent Internal Order ID. The **TargetInternalOrderID**, **StopInternalOrderID** parameters receive the Target and Stop Internal Order IDs respectively.

For information about Internal Order IDs, refer to the [ACSIL Trading](#) page. These Internal Order IDs can be obtained when submitting an order.

Example

```
int TargetInternalOrderID = -1;
int StopInternalOrderID = -1;

//This needs to be set to the parent internal order ID search for. Since we do not know what that is in this co
int ParentInternalOrderID = 0;

sc.GetAttachedOrderIDsForParentOrder(ParentInternalOrderID, TargetInternalOrderID, StopInternalOrderID)
```

sc.GetBarHasClosedStatus()

Type: Function

```
int GetBarHasClosedStatus(int BarIndex);
```

```
int GetBarHasClosedStatus(); Auto-looping only.
```

The **sc.GetBarHasClosedStatus** function is used to determine if the data for a particular bar in

the **sc.BaseData[][]** arrays at the specified **BarIndex** parameter (required for manual looping), has completed and will no longer be updated during real-time or replay updating.

This is useful when you only want to perform calculations on a fully completed bar. It is equivalent to "Signal Only on Bar Close" with the Spreadsheet studies.

This function has several return values described below.

Refer to the **scsf_GetBarHasClosedStatusExample()** function in the **/ACS_Source/studies.cpp** file in the Sierra Chart installation folder for example code to work with this function.

This function can be used with [manual or automatic looping](#). It can be given an index of any bar. **sc.BaseData[][]** contains the underlying bar data for the chart. In this function call: **sc.GetBarHasClosedStatus(sc.Index)**, **sc.Index** is passed through the **BarIndex** parameter and this will refer to the bar data at **sc.BaseData[][sc.Index]**.

The very last bar in the chart is never considered a closed bar until there is a new bar added to the chart. It is not possible to know otherwise because of the following reasons: The chart bars are based upon a variable timeframe like **Number of Trades** or **Volume** and the ending can never be known until there is a new bar, or because there is not a trade at the very final second of a fixed time bar.

In the case of chart bars which are based upon a fixed amount of time, the only way to know when the end of that chart bar has occurred, is to consider the time length of the bar through [sc.SecondsPerBar](#). Use [sc.UpdateAlways](#), to have the study function periodically and continuously called. And use the current Date-Time through [sc.GetCurrentDateTime\(\)](#) to know when the bar is considered closed.

Return Values:

- **BHCS_BAR_HAS_CLOSED**: Element at BarIndex has closed. This will always be returned for any bar in the chart other than the last bar in the chart.
- **BHCS_BAR_HAS_NOT_CLOSED**: Element at BarIndex has not closed. This will always be returned for the last bar in the chart.
- **BHCS_SET_DEFAULTS**: Configuration and defaults are being set. Allow the **sc.SetDefaults** code block to run. Bar has closed status is not available.

Example

```
if(sc.GetBarHasClosedStatus(Index)==BHCS_BAR_HAS_NOT_CLOSED)
{
    return;//do not do any processing if the bar at the current index has not closed
}
```

sc.GetBarPeriodParameters()

Type: Function

```
void GetBarPeriodParameters(n_ACSIL::s_BarPeriod& r_BarPeriod)
```

The **sc.GetBarPeriodParameters()** function copies the chart bar period parameters into the structure of type **n_ACSIL::s_BarPeriod** which is passed to the function as **r_BarPeriod**. Refer to the example below.

When using [sc.SetBarPeriodParameter\(\)](#) to change the period for a chart bar, and then making a call to **sc.GetBarPeriodParameters()** immediately after, but during the processing in the study function, it will then return these new set parameters.

The **n_ACSIL::s_BarPeriod** structure members are the following:

- **s_BarPeriod::ChartDataTypes**: Can be one of the following:
 - DAILY_DATA = 1
 - INTRADAY_DATA = 2.
- **s_BarPeriod::HistoricalChartBarPeriodType**: Can be one of the following:
 - HISTORICAL_CHART_PERIOD_DAYS = 1
 - HISTORICAL_CHART_PERIOD_WEEKLY = 2
 - HISTORICAL_CHART_PERIOD_MONTHLY = 3
 - HISTORICAL_CHART_PERIOD_QUARTERLY = 4
 - HISTORICAL_CHART_PERIOD_YEARLY = 5
- **s_BarPeriod::HistoricalChartDaysPerBar**: When **s_BarPeriod::HistoricalChartBarPeriodType** is set to HISTORICAL_CHART_PERIOD_DAYS, then this specifies the number of days per historical chart bar.
- **s_BarPeriod::IntradayChartBarPeriodType**: The type of bar period that is being used in the case of an Intraday chart. To determine the chart data type, use **s_BarPeriod::ChartDataTypes**. For example, this would be set to the enumeration value IBPT_DAYS_MINS_SECS if the Intraday Chart **Bar Period Type** is set to **Days-Minutes-Seconds**. Can be any of the following constants:
 - IBPT_DAYS_MINS_SECS = 0:
 - s_BarPeriod::IntradayChartBarPeriodParameter1** is the number of seconds in one bar in an Intraday chart. This is set by the **Days-Mins-Secs** setting in the **Chart >> Chart Settings** window for the chart. For example, for a 1 Minute chart this will be set to 60. For a 30 Minute chart this will be set to 1800.
 - IBPT_VOLUME_PER_BAR = 1
 - IBPT_NUM_TRADES_PER_BAR = 2
 - IBPT_RANGE_IN_TICKS_STANDARD = 3
 - IBPT_RANGE_IN_TICKS_NEWBAR_ON_RANGEMET = 4
 - IBPT_RANGE_IN_TICKS_TRUE = 5
 - IBPT_RANGE_IN_TICKS_FILL_GAPS = 6
 - IBPT_REVERSAL_IN_TICKS = 7
 - IBPT_RENKO_IN_TICKS = 8
 - IBPT_DELTA_VOLUME_PER_BAR = 9

- IBPT_FLEX_RENKO_IN_TICKS = 10
 - IBPT_RANGE_IN_TICKS_OPEN_EQUAL_CLOSE = 11
 - IBPT_PRICE_CHANGES_PER_BAR = 12
 - IBPT_MONTHS_PER_BAR = 13
 - IBPT_POINT_AND_FIGURE = 14
 - IBPT_FLEX_RENKO_IN_TICKS_INVERSE_SETTINGS = 15
 - IBPT_ALIGNED_RENKO = 16
 - IBPT_RANGE_IN_TICKS_NEW_BAR_ON_RANGE_MET_OPEN_EQUALS_PRIOR = 17
 - IBPT_ACSIL_CUSTOM = 18: This is used when the chart contains an advanced custom study that creates custom chart bars. The study name is contained within `s_BarPeriod::ACSILCustomChartStudyName`. For complete documentation for building custom chart bars, refer to [ACSIL Interface - Custom Chart Bars](#).
- **s_BarPeriod::IntradayChartBarPeriodParameter1**: The first parameter for the bar period that is being used. In the case of `IntradayChartBarPeriodType` being set to `IBPT_DAYS_MINS_SECS`, this will be set to the number of seconds. In the case of `IntradayChartBarPeriodType` being set to `IBPT_FLEX_RENKO_IN_TICKS`, this would be the **Bar Size** value in ticks.
 - **s_BarPeriod::IntradayChartBarPeriodParameter2**: The second parameter for the bar period that is being used. For example, this would be the **Trend Offset** value when using `IBPT_FLEX_RENKO_IN_TICKS` for the Intraday Chart Bar Period Type. If this parameter is unused for the Intraday Chart Bar Period Type, then the value is set to **0**.
 - **s_BarPeriod::IntradayChartBarPeriodParameter3**: The third parameter for the bar period that is being used. For example, this would be the **Reversal Offset** value when using `IBPT_FLEX_RENKO_IN_TICKS` for the Intraday Chart Bar Period Type. If this parameter is unused for the Intraday Chart Bar Period Type, then the value is set to **0**.
 - **s_BarPeriod::IntradayChartBarPeriodParameter4**: The fourth parameter for the bar period that is being used. For example, this would be the **Renko New Bar Mode** setting when using any of the **Renko** Intraday Chart Bar Period Types. If this parameter is unused for the Intraday Chart Bar Period Type, then the value is set to **0**. The values for **Renko New Bar Mode** are listed below.
 - NEW_TREND_BAR_WHEN_EXCEEDED_NEW_REVERSAL_BAR_WHEN_REVERSED = 0
 - NEW_TREND_BAR_WHEN_EXCEEDED_NEW_BAR_WHEN_OPEN_CROSSED = 1
 - NEW_TREND_BAR_WHEN_EXCEEDED_NEW_REVERSAL_BAR_WHEN_REVERSED = 2
 - NEW_TREND_BAR_WHEN_RANGE_MET_NEW_REVERSAL_BAR_WHEN_REVERSED = 3
 - NEW_TREND_BAR_WHEN_RANGE_MET_NEW_BAR_WHEN_OPEN_CROSSED = 4
 - NEW_TREND_BAR_WHEN_RANGE_MET_NEW_REVERSAL_BAR_WHEN_REVERSED = 5

- = 5
 - NEW_TREND_BAR_AFTER_RANGE_MET_NEW_REVERSAL_BAR_WHEN_REVE
= 6
 - NEW_TREND_BAR_AFTER_RANGE_MET_NEW_BAR_WHEN_OPEN_CROSSED
= 7
 - NEW_TREND_BAR_AFTER_RANGE_MET_NEW_REVERSAL_BAR_WHEN_REVE
= 8
 - NEW_TREND_BAR_AFTER_RANGE_MET_ALLOW_CHANGE_OF_DIRECTION_O
= 9
 - NEW_TREND_BAR_AFTER_RANGE_MET_NEW_REVERSAL_BAR_WHEN_REVE
= 10
 - NEW_TREND_BAR_WHEN_EXCEEDED_NEW_REVERSAL_BAR_WHEN_REVER!
= 11
- **s_BarPeriod::ACSILCustomChartStudyName:** When the chart contains an advanced custom study that creates custom chart bars, this contains the name as a texturing of the custom study which creates those custom chart bars. The name is set through [sc.GraphName](#).

For complete documentation for building custom chart bars, refer to [ACSIL Interface - Custom Chart Bars](#).

Example

```
n_ACSIL::s_BarPeriod BarPeriod;
sc.GetBarPeriodParameters(BarPeriod);
if (BarPeriod.ChartDataType == INTRADAY_DATA && BarPeriod.IntradayChartBarPeriodType == IBPT_D
{
    int SecondsPerBar = BarPeriod.IntradayChartBarPeriodParameter1;
}
```

sc.GetBarsSinceLastTradeOrderEntry()

Type: Function

int GetBarsSinceLastTradeOrderEntry()

This function returns the number of chart bars counting from the end of the chart since the Date-Time of the last entry trade order for the Symbol and Trade Account of the chart the study is applied to, whether the order was from ACSIL or not.

If there has been no entry order or the entry was on the last bar of the chart, then this function returns 0.

sc.GetBarsSinceLastTradeOrderExit()

Type: Function

int **GetBarsSinceLastTradeOrderExit()**

This function returns the number of chart bars counting from the end of the chart since the Date-Time of the last exit trade order for the Symbol and Trade Account of the chart the study is applied to, whether the order was from ACSIL or not.

If there has been no exit order or the exit was on the last bar of the chart, then this function returns 0.

sc.**GetBasicSymbolData()**

Type: Function

```
void GetBasicSymbolData(const char* Symbol, s_SCBasicSymbolData& BasicSymbolData,  
bool Subscribe)
```

The **sc.GetBasicSymbolData** will fill out the **BasicSymbolData** data structure with all of the Current Quote data for the specified **Symbol**. This data is only considered valid when connected to the data feed. The Current Quote data is the data that is found in

Window >> Current Quote Window. Market Depth data is also included in the data structure.

Refer to declaration of s_SCBasicSymbolData in the /ACS_Source/scsymboldata.h file in the folder Sierra Chart is installed to for all of the structure members. The data that is returned when calling this function is the most current available data.

The **Subscribe** parameter will subscribe to market data for the symbol when connected to the data feed. Subscribing to a symbol, does not mean that the study function will be called for all updates on the symbol. The study function will only be called based upon the updates of the symbol of the chart the study instance is on.

As an example, you could have a list of 10 symbols to request data for by calling **sc.GetBasicSymbolData()** for each symbol. After each call, examine the returned data and do any required calculations. So in this scenario, you can form an index by adding all of the last price values together and dividing by 10, and placing the results in a **sc.Subgraph[].Data[]** array.

Parameters

- **Symbol**: The symbol to get the Symbol Data for. For information about working with strings, refer to [Working with Strings](#).
- **BasicSymbolData**: A reference to a data structure of type s_SCBasicSymbolData
- **SubscribeToData**: Set this to 1 to subscribe to the standard real-time market data. Otherwise, set this to zero to not subscribe.
- **SubscribeToMarketDepth**: Set this to 1 to subscribe to the real-time market depth data. This parameter is only supported by the [sc.GetBasicSymbolDataWithDepthSupport](#) function. Otherwise, set this to zero to not subscribe.

Example

```
// Get the daily high price for another symbol. The first time this
// function is called after connecting to the data feed, the symbol data
// will not be available initially. Once the data becomes available, when
// this function is called after, the symbol data will be available. If
// this study is used on a chart for a symbol which does not frequently
// update, then it will be a good idea to set sc.UpdateAlways = 1.
const char * Symbol = "ABC";
s_SCBasicSymbolData BasicSymbolData;
sc.GetBasicSymbolData(Symbol,BasicSymbolData, true);
float DailyHigh = BasicSymbolData.High;

float LatestAsk = BasicSymbolData.Ask;// Also refer to the latest ask price
```

sc.GetBasicSymbolDataWithDepthSupport

Type: Function

```
void GetBasicSymbolDataWithDepthSupport(const char* Symbol, s_SCBasicSymbolData&
BasicSymbolData, int SubscribeToData, int SubscribeToMarketDepth);
```

The **sc.GetBasicSymbolDataWithDepthSupport()** function is identical to the [sc.GetBasicSymbolData](#) function.

The difference is that it supports subscribing to market depth data.

sc.GetBidMarketDepthEntryAtLevel

Type: Function

```
int GetBidMarketDepthEntryAtLevel(s_MarketDepthEntry& DepthEntry, int LevelIndex);
```

sc.GetBidMarketDepthEntryAtLevel returns in the **DepthEntry** structure of type **s_MarketDepthEntry**, the bid side market depth data for the level specified by the **LevelIndex** variable for the symbol of the chart.

The function returns 1 if the function call was successful, otherwise 0.

To receive Market Depth data it is also necessary to set [sc.UsesMarketDepthData](#) to 1 in the study function. This normally should be placed in the **sc.SetDefaults** code block.

For an example to access market depth data, refer to the **scsf_DepthOfMarketData()** function in the **/ACS_Source/studies.cpp** file in the folder where Sierra Chart is installed to.

The **s_MarketDepthEntry** structure contains a **Price** member and an **AdjustedPrice** member. Normally these are the same price. However, when a Real-time Price Multiplier is set to a value other than 1.0 in a chart, then **Price** will contain the original unadjusted price and **AdjustedPrice** will contain the **Price** multiplied by the Real-time Price Multiplier.

s_MarketDepthEntry

```

struct s_MarketDepthEntry
{
    float Price;
    t_MarketDataQuantity Quantity;
    uint32_t NumOrders;
    float AdjustedPrice;
}

```

sc.GetBidMarketDepthEntryAtLevelForSymbol

Type: Function

```

int GetBidMarketDepthEntryAtLevelForSymbol(const SCString& Symbol,
s_MarketDepthEntry& r_DepthEntry, int LevelIndex);

```

sc.GetBidMarketDepthEntryAtLevelForSymbol returns in the **DepthEntry** structure of type **s_MarketDepthEntry**, the bid side market depth data for the level specified by the **LevelIndex** variable for the specified **Symbol**.

The function returns 1 if the function call was successful, otherwise 0.

To receive Market Depth data it is also necessary to set [sc.UsesMarketDepthData](#) to 1 in the study function. This normally should be placed in the **sc.SetDefaults** code block.

For an example to access market depth data, refer to the **scsf_DepthOfMarketData()** function in the **/ACS_Source/studies.cpp** file in the folder where Sierra Chart is installed to.

s_MarketDepthEntry

```

struct s_MarketDepthEntry
{
    float Price;
    t_MarketDataQuantity Quantity;
    uint32_t NumOrders;
    float AdjustedPrice;
}

```

sc.GetBidMarketDepthNumberOfLevels

Type: Function

```

int GetBidMarketDepthNumberOfLevels();

```

The **sc.GetBidMarketDepthNumberOfLevels** function returns the number of available market depth levels on the Bid side for the symbol of the chart.

There are settings which affect the number of market depth levels. Refer to [Not Receiving the Full Number of Market Depth Levels](#).

sc.GetBidMarketDepthNumberOfLevelsForSymbol

Type: Function

```
int GetBidMarketDepthNumberOfLevelsForSymbol(const SCString& Symbol);
```

The **sc.GetBidMarketDepthNumberOfLevelsForSymbol** function returns the number of available market depth levels on the Bid side for the specified symbol.

There are settings which affect the number of market depth levels. Refer to [Not Receiving the Full Number of Market Depth Levels](#).

Parameters

- **Symbol:** The symbol to get the number of market depth levels for.

Example

```
int NumberOfLevels = GetBidMarketDepthNumberOfLevelsForSymbol(sc.Symbol);
```

sc.GetBidMarketLimitOrdersForPrice

Type: Function

```
int GetBidMarketLimitOrdersForPrice(const int PriceInTicks, const int ArraySize,  
n_ACSIL::s_MarketOrderData* p_MarketOrderDataArray);
```

The **sc.GetBidMarketLimitOrdersForPrice** function is used to get the order ID and order quantity for working limit orders from the market data feed at the specified price level. This data is called [Market by Order](#) data. This includes all the working limit orders as provided by the market data feed.

Parameters

- **PriceInTicks:** This is the price to get the working limit orders for. This is an integer value. You need to take the floating-point actual price and divide by **sc.TickSize** and round to the nearest integer.
- **ArraySize:** This is the size of the **p_MarketOrderDataArray** array as a number of elements, that you provide the function.
- **p_MarketOrderDataArray:** This is a pointer to the array of type **n_ACSIL::s_MarketOrderData** of the size specified by **ArraySize** that is filled in with the working limit orders data upon return of the function.

Example

For an example to use this function, refer to the **scsf_MarketLimitOrdersForPriceExample** study function in the /ACS_Source/Studies2.cpp file in the Sierra Chart installation folder.

sc.GetBuiltInStudyName

Type: Function

```
int GetBuiltInStudyName(const int InternalStudyIdentifier, SCString& r_StudyName);
```

The **sc.GetBuiltInStudyName** function .

Parameters

- **InternalStudyIdentifier:** .
- **r_StudyName:** .

Example

sc.GetCalculationStartIndexForStudy()

Type: Function

```
int GetCalculationStartIndexForStudy();
```

The **sc.GetCalculationStartIndexForStudy()** function returns the starting index where a study function needs to begin its calculations at when it has a dependency on a study which has started a calculation at an earlier start index than normal. This function is for specialized purposes. It is not normally used.

For example, the Zig Zag study during a chart update may change the Zig Zag line at a chart bar index which is earlier than the chart bar being updated and any new bars added to the chart.

Any study function which uses this function, must use [Manual Looping](#).

For a related variable, also refer to [sc.EarliestUpdateSubgraphDataArrayIndex](#).

Example

The following code example is from the **Study Subgraph Add** study.


```

int CalculationStartIndex = sc.GetCalculationStartIndexForStudy();

for (int Index = CalculationStartIndex; Index < sc.ArraySize; Index++)
{
    float BasedOnStudySubgraphValue = sc.BaseData[InputData.GetInputDataIndex()][Index];

    if (AddToZeroValuesInBasedOnStudy.GetYesNo() == 0 && BasedOnStudySubgraphValue == 0.0)
    {
        Result[Index] = 0.0;
    }
    else
        Result[Index] = BasedOnStudySubgraphValue + AmountToAdd.GetFloat();
}

sc.EarliestUpdateSubgraphDataArrayIndex = CalculationStartIndex;

```

sc.GetChartArray()

Type: Function

GetChartArray(int **ChartNumber**, int **InputData**, SCFloatArrayRef **PriceArray**);

sc.GetChartArray() is for accessing the main/primary base graph data in other loaded charts in the same chartbook containing the chart that your study function is applied to. This is an older function and it is highly recommended that you use **sc.GetChartBaseData** instead. See the **scsf_GetChartArrayExample()** function in the studies.cpp file in the ACS_Source folder in the Sierra Chart installation folder for example code to work with this function.

Parameters

- **ChartNumber**: The number of the chart you want to get data from. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **InputData**: This can be one of the following:
 - **SC_OPEN or 0**: The array of opening prices for each bar.
 - **SC_HIGH or 1**: The array of high prices for each bar.
 - **SC_LOW or 2**: The array of low prices for each bar.
 - **SC_LAST or 3**: The array of closing/last prices for each bar.
 - **SC_VOLUME or 4**: The array of trade volumes for each bar.
 - **SC_NUM_TRADES / SC_OPEN_INTEREST or 5**: The array of the number of trades for each bar for Intraday charts. Or the open interest for each bar for daily charts.
 - **SC_OHLC or 6**: The array of the average prices of the open, high, low, and close prices for each bar.
 - **SC_HLC or 7**: The array of the average prices of the high, low, and close prices for each bar.

- **SC_HL or 8:** The array of the average prices of the high and low prices for each bar.
- **SC_BIDVOL or 9:** The array of Bid Volumes for each bar. This represents the volumes of the trades that occurred at the bid.
- **SC_ASKVOL or 10:** The array of Ask Volumes for each bar. This represents the volumes of the trades that occurred at the ask.
- **PriceArray:** A SCFloatArray object which will be set to the data array that you requested. If the data array that you requested could not be retrieved, then the size of this array, **PriceArray.GetArraySize()**, will be 0.

Example

```
SCFloatArray PriceArray;

// Get the close/last array from chart #1

sc.GetCharArray(1, SC_LAST, PriceArray);

// The PriceArray may not exist or is empty. Either way we can not do anything with it.
if (PriceArray.GetArraySize() == 0)

    return;
```

sc.GetChartData()

Type: Function

GetChartData(int ChartNumber, SCGraphData & BaseData);

The **sc.GetChartData()** function is for accessing all of the main/primary Base Data arrays in another loaded chart in the same Chartbook as the one containing the chart that your custom study function is applied to which calls this function.

The main Base Data arrays refer to the arrays of the main price graph in a chart. If the main price graph has been replaced by using a custom price chart study such as the Point and Figure chart study, then this function will get this new main price graph from the specified ChartNumber.

Refer to the example below. For a complete working example, refer to [Referencing Other Time Frames and Symbols When Using the ACSIL](#).

For information about getting the corresponding index in the arrays returned, refer to [Accessing Correct Array Indexes in Other Chart Arrays](#).

When you are getting data from another chart with this function, this other chart during a chart replay may not have the data which is up to date to the time of the chart containing the study calling the **sc.GetChartData** function.

Parameters

- **ChartNumber:** The number of the chart you want to get data from. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **BaseData:** A SCGraphData object which will be set to all of the main/primary base graph arrays from the specified chart.

Example

```
// The following code is for getting the High array
// and corresponding index from another chart.

// Define a graph data object to get all of the base graph data
SCGraphData BaseGraphData;

// Get the base graph data from the specified chart
sc.GetChartData(BaseGraphData, ChartNumber.GetInt(), BaseGraphData);

// Define a reference to the High array
SCFloatArrayRef HighArray = BaseGraphData[SC_HIGH];

// Array is empty. Nothing to do.
if(HighArray.GetArraySize() == 0)

return;

// Get the index in the specified chart that is
// nearest to current index.
int RefChartIndex =
sc.GetNearestMatchForDateTimeIndex(ChartNumber.GetInt(), sc.Index);

float NearestRefChartHigh = HighArray[RefChartIndex];
```

sc.GetChartDataTimeArray()

Type: Function

GetChartDataTimeArray(int ChartNumber, SCDateTimeArray& DateTimeArray);

sc.GetChartDataTimeArray() is used to access the [SCDateTime](#) array ([sc.BaseDateTimeIn\[\]](#)) in other loaded charts in the same chartbook containing the chart that your study is applied to. See the **sclf_GetChartArrayExample()** function in the studies.cpp file inside the ACS_Source folder inside of the Sierra Chart installation folder for example code to work with this function.

Parameters

- **ChartNumber:** The number of the chart you want to get the data from. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied

to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass - 5.

- **DateTimeArray**: A SCDatetimeArray object which will be set to the SCDatetime array that you requested. If the SCDatetime array that you requested could not be retrieved, then the size of this array (**DateTimeArray.GetArraySize()**) will be 0.

Example

```
SCDateTimeArray DateTimeArray;  
  
// Get the DateTime array from chart #1  
sc.GetChartDateTimeArray(1, DateTimeArray);  
  
// The array may not exist or is empty. Either way we can not do anything with it.  
if (DateTimeArray.GetArraySize() == 0)  
    return;
```

sc.GetChartDrawing()

Type: Function

For more information, refer to the [sc.GetUserDrawnChartDrawing\(\)](#) section on the **Using Tools From an Advanced Custom Study** page.

sc.GetChartFontProperties()

Type: Function

```
int GetChartFontProperties(SCString& r_FontName, int32_t& r_FontSize, int32_t&  
r_FontBold, int32_t& r_FontUnderline, int32_t& r_FontItalic);
```

The **sc.GetChartFontProperties** function gets the information for the Chart Text Font that is in use for the chart associated with the study from which this function is called.

The function returns a value of **1** if it is able to successfully get the Chart Text Font information, otherwise it returns a value of **0**.

Parameters

- **r_FontName**: A SCString that contains the name of the font.
- **r_FontSize**: An integer that contains the size of the font.
- **r_FontBold**: An integer that specifies if the font is bold or not. A value of **0** indicates it is not bold. A value of **1** indicates the font is bold.
- **r_FontUnderline**: An integer that specifies if the font is underlined or not. A value of **0** indicates it is not underlined. A value of **1** indicates the font is underlined.

- **r_Font_Italic**: An integer that specifies if the font is italic or not. A value of 0 indicates it is not italic. A value of 1 indicates the font is italic.

sc.GetChartName()

Type: Function

```
SCString GetChartName(int ChartNumber);
```

The **sc.GetChartName** function, returns the name of the chart specified by the ChartNumber parameter.

It is only possible to access charts which are in the same Chartbook as the chart containing the study function which is calling this function.

The name contains the symbol of the chart as well as the timeframe per bar and the chart number.

For an example of how to use this function, refer to the **scsf_Spread3Chart** function in the **/ACS_Source/studies7.cpp** file in the folder where Sierra Chart is installed to.

Example

```
SCString Chart1Name = sc.GetChartName(sc.ChartNumber);
```

sc.GetChartReplaySpeed()

Type: Function

```
float GetChartReplaySpeed(int ChartNumber);
```

Parameters

- **ChartNumber**: The number of the chart to get the replay speed for. Refer to [Chart Number](#).

sc.GetChartStudyInputChartStudySubgraphValues()

Type: Function

```
int GetChartStudyInputChartStudySubgraphValues(int ChartNumber, int StudyID, int InputIndex, s_ChartStudySubgraphValues& r_ChartStudySubgraphValues);
```

The **sc.GetChartStudyInputChartStudySubgraphValues** function .

Parameters

- **ChartNumber:** The chart number containing the study to get the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to get the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **r_ChartStudySubgraphValues:** .

Example

sc.GetChartStudyInputFloat()

Type: Function

```
int GetChartStudyInputFloat(int ChartNumber, int StudyID, int InputIndex, double& r_FloatValue);
```

The **sc.GetChartStudyInputFloat** function gets the study Input value as a double from the specified chart number and study ID.

This function works with all Input types and returns the value as a double. The necessary conversions are automatically made.

The function returns 1 if successful. Otherwise, 0.

Parameters

- **ChartNumber:** The chart number containing the study to get the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to get the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **FloatValue:** This parameter receives the double value. It is a reference.

Example

```
double FloatInput = 0;
sc.GetChartStudyInputFloat(sc.ChartNumber, 1, 1, FloatInput);
```

sc.SetChartStudyInputFloat()

Type: Function

```
int SetChartStudyInputFloat(int ChartNumber, int StudyID, int InputIndex, double FloatValue);
```

The **sc.SetChartStudyInputFloat** function sets the study Input value as a double data type in the specified chart number and study ID.

This function works with all Input types and sets the value as a double. The necessary conversions are automatically made. When working with Date and Time inputs which use [SCDateTime](#), use this function for setting them.

After changing a study Input, in most cases you will need to recalculate the chart the study is on in order for the new Input to be used. But this depends upon how the study is using it. If it is a study that you have developed and you know that a recalculation is not necessary, then this is not necessary. To recalculate the chart, use the [sc.RecalculateChart\(\)](#) function.

The function returns 1 if successful. Otherwise, 0.

Parameters

- **ChartNumber:** The chart number containing the study to set the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to set the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **FloatValue:** This parameter is the double value to set.

Example

```
int Result = sc.SetChartStudyInputFloat(sc.ChartNumber, 1, 1, 1.5);
```

sc.GetChartStudyInputInt()

Type: Function

```
int GetChartStudyInputInt(int ChartNumber, int StudyID, int InputIndex, int& r_IntegerValue);
```

The **sc.GetChartStudyInputInt** function gets the study Input value as an integer from the specified chart number and study ID.

This function works with all Input types and returns the value as an integer. The necessary conversions are automatically made.

The function returns 1 if successful. Otherwise, 0.

Parameters

- **ChartNumber:** The chart number containing the study to get the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to get the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **IntegerValue:** This parameter receives the integer value. It is a reference.

Example

```
int IntegerInput = 0;  
sc.GetChartStudyInputInt(sc.ChartNumber, 1, 1, IntegerInput);
```

sc.SetChartStudyInputInt()

Type: Function

```
int SetChartStudyInputInt(int ChartNumber, int StudyID, int InputIndex, int IntegerValue);
```

The **sc.SetChartStudyInputInt** function sets the study Input value as an integer in the specified chart number and study ID.

This function works with all Input types and sets the value using an integer. The necessary conversions are automatically made.

After changing a study Input, in most cases you will need to recalculate the chart the study is on in order for the new Input to be used. But this depends upon how the study is using it. If it is a study that you have developed and you know that a recalculation is not necessary, then this is not necessary. To recalculate the chart, use the [sc.RecalculateChart\(\)](#) function.

Another reason why you would not have to do a recalculate with **sc.RecalculateChart()** is if after changing a study Input you are then starting a replay for the chart using the [sc.StartChartReplayNew\(\)](#) function.

The function returns 1 if successful. Otherwise, 0.

Parameters

- **ChartNumber:** The chart number containing the study to set the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to

[UniqueStudyInstanceIdentifiers](#) for more information.

- **InputIndex:** The zero-based index of the Input to set the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **IntegerValue:** This parameter is the integer value to set.

Example

```
int Result = sc.SetChartStudyInputInt(sc.ChartNumber, 1, 1, 20);
```

sc.GetChartStudyInputString()

Type: Function

```
int GetChartStudyInputString(int ChartNumber, int StudyID, int InputIndex, SCString& r_StringValue);
```

The **sc.GetChartStudyInputString** function gets the study Input value as a text string from the specified chart number and study ID.

This function works with only string Input types and returns the value as a string.

The function returns 1 if successful. Otherwise, 0.

Parameters

- **ChartNumber:** The chart number containing the study to get the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to get the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **r_StringValue:** This [SCString](#) parameter receives the string. It is a reference.

Example

```
SCString StringInput;  
sc.GetChartStudyInputString(sc.ChartNumber, 1, 1, StringInput);
```

sc.GetChartStudyInputType()

Type: Function

```
int GetChartStudyInputType(int ChartNumber, int StudyID, int InputIndex);
```

The **sc.GetChartStudyInputType** function

Parameters

- **ChartNumber**: The chart number containing the study to get the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID**: The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex**: The zero-based index of the Input to get the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).

Example



sc.SetChartStudyInputString()

Type: Function

```
int SetChartStudyInputString(int ChartNumber, int StudyID, int InputIndex, const SCString& StringValue);
```

The **sc.SetChartStudyInputString** function sets the study Input with the specified text string in the specified chart number and study ID.

This function works only with string Input types. If the Input is not already a string, then the function will return 0 and does nothing.

After changing a study Input, in most cases you will need to recalculate the chart the study is on in order for the new Input to be used. But this depends upon how the study is using it. If it is a study that you have developed and you know that a recalculation is not necessary, then this is not necessary. To recalculate the chart, use the [sc.RecalculateChart\(\)](#) function.

The function returns 1 if successful. Otherwise, 0.

Parameters

- **ChartNumber**: The chart number containing the study to set the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID**: The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex**: The zero-based index of the Input to set the value for. The Input

index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).

- **StringValue:** This parameter is the string to set.

Example

```
int Result = sc.SetChartStudyInputString(sc.ChartNumber, 1, 1, "My Text String");
```

sc.SetChartTradeMode()

Type: Function

```
int SetChartTradeMode(int ChartNumber, int Enabled);
```

The **sc.SetChartTradeMode** function .

Parameters

- **ChartNumber:** The chart number containing the study to set the Input value for. For more information, refer to [sc.ChartNumber](#).
- **Enabled:** .

Example

sc.GetChartSymbol()

Type: Function

```
SCString GetChartSymbol(int ChartNumber);
```

The **sc.GetChartSymbol()** function returns as a text string of type [SCString](#), the symbol of the chart specified by the **ChartNumber** parameter. To get the symbol of the chart the study instance is applied to, specify [sc.ChartNumber](#) for the **ChartNumber** parameter.

If the returned text string is blank, this indicates the chart does not exist.

sc.GetChartTextFontFaceName()

Type: Function

```
SCString sc.GetChartTextFontFaceName();
```

The **sc.GetChartTextFontFaceName()** function returns the font name as a text string of the font

that the chart the study instance is applied to, is set to use.

Example

```
Tool.FontFace = sc.GetChartTextFontFaceName();
```

sc.GetChartTimeZone()

Type: Function

```
SCString sc.GetChartTimeZone(const int ChartNumber);
```

The **sc.GetChartTimeZone()** function .

Parameters

- **ChartNumber:** The number of the chart to get the window handle for. Each chart has a number and the Chart Number is displayed on its title bar.

Example

sc.GetChartWindowHandle()

Type: Function

```
HWND sc.GetChartWindowHandle(int ChartNumber);
```

The **sc.GetChartWindowHandle()** function returns the window handle as an integer for the specified **ChartNumber**. Use version 1616 or higher to get the proper handle when using this function.

Parameters

- **ChartNumber:** The number of the chart to get the window handle for. Each chart has a number and the Chart Number is displayed on its title bar.

Example

```
HWND WindowHandle = sc.GetChartWindowHandle(sc.ChartNumber)
```

sc.GetCombineTradesIntoOriginalSummaryTradeSetting()

Type: Function

```
int sc.GetCombineTradesIntoOriginalSummaryTradeSetting();
```

The **sc.GetCombineTradesIntoOriginalSummaryTradeSetting()** function

Parameters

- This function has no parameters.

Example

sc.GetContainingIndexForDateTimeIndex()

Type: Function

```
int GetContainingIndexForDateTimeIndex(int ChartNumber, int DateTimeIndex);
```

sc.GetContainingIndexForDateTimeIndex() returns the index into the Base Data arrays of the chart specified by **ChartNumber** that contains the Date-Time at the index, on the chart your study function is applied to, specified by **DateTimeIndex**. If the Date-Time at **DateTimeIndex** is before any Date-Time in the chart specified with **ChartNumber**, then the index of the first element is given which will be 0. If the Date-Time at **DateTimeIndex** is after any Date-Time in the chart specified with **ChartNumber**, then the index of the last element is given which will be **sc.ArraySize - 1**.

Containing means that the chart bar starting Date-Time and ending Date-Time in **ChartNumber** contains the Date-Time specified by the **DateTimeIndex** parameter.

This function can be used on the **sc.BaseData[]** arrays for the same chart that your study function is applied to when searching for the Date-Time. To do this, pass **sc.ChartNumber** for the **ChartNumber** parameter.

If **ChartNumber** is given as a negative number, the bar period and other Chart Settings are synchronized between the two charts. If it is positive, this does not occur. For example, if you want to get the index from chart #5, and you want to synchronize the charts, then use -5 for the **ChartNumber** parameter.

Error Return Value

This function will return -1 if the chart it is referencing does not exist or its data is not fully loaded.

When the chart it is referencing is opened and fully loaded, then the study instance that made this call will be called again and will be fully recalculated. What actually happens in this case, is that the chart the reference was made from, will have all of its studies recalculated.

Example

```
// Get the index in the Base Data arrays for chart #2 that contains
// the Date-Time at the current Index of the chart that this study
// is applied to.

int Chart2Index = sc.GetContainingIndexForDateTimeIndex(2, sc.Index);
```

sc.GetContainingIndexForSCDateTime()

Type: Function

```
int GetContainingIndexForSCDateTime(int ChartNumber, SCDateTime DateTime);
```

sc.GetContainingIndexForSCDateTime() returns the index into the Base Data arrays of the chart specified by **ChartNumber** that contains **DateTime** . If **DateTime** is before any Date-Time in the chart specified with **ChartNumber**, then the index of the first element is given which is 0. If **DateTime** is after any Date-Time in the chart specified with **ChartNumber** , then the index of the last element is given which is `sc.ArraySize - 1`.

Containing means that the chart bar starting Date-Time and ending Date-Time in **ChartNumber** contains the Date-Time specified by the **DateTime** parameter.

This function can be used on the **sc.BaseData[]** arrays for the same chart that your study function is applied to when searching for the Date-Time. To do this, pass **sc.ChartNumber** for the **ChartNumber** parameter.

If **ChartNumber** is given as a negative number, the bar period and other Chart Settings are synchronized between the two charts. If it is positive, this does not occur. For example, if you want to get the index from chart #5, and you want to synchronize the charts, then use -5 for the **ChartNumber** parameter.

Error Return Value

This function will return -1 if the chart it is referencing does not exist or its data is not fully loaded.

When the chart it is referencing is opened and fully loaded, then the study instance that made this call will be called again and will be fully recalculated. What actually happens in this case, is that the chart the reference was made from, will have all of its studies recalculated.

Example

```
// Get the index in the Base Data arrays for chart #2 that contains the specified Date-Time.

SCDateTime DateTime = sc.BaseDateTimeIn[sc.Index];

int Chart2Index = sc.GetContainingIndexForSCDateTime(2, DateTime);
```

sc.GetCorrelationCoefficient()

Type: Intermediate Study Calculation Function

```
float GetCorrelationCoefficient(SCFloatArrayRef FloatArrayIn1, SCFloatArrayRef
FloatArrayIn2, int Index, int Length);
```

```
float GetCorrelationCoefficient(SCFloatArrayRef FloatArrayIn1, SCFloatArrayRef
FloatArrayIn2, int Length); Auto-looping only.
```

The **sc.GetCorrelationCoefficient()** function calculates the [Pearson product-moment correlation coefficient](#) from the data in the FloatArrayIn1 and FloatArrayIn2 arrays. The result is returned as a single float value.

Parameters

- [FloatArrayIn1](#).
- [FloatArrayIn2](#).
- [Index](#).
- [Length](#).

Example

```
float Coefficient = sc.GetCorrelationCoefficient(sc.Subgraph[0], sc.Subgraph[1], 10);
```

sc.GetCountDownText()

Type: Function

```
SCString GetCountDownText();
```

The **sc.GetCountDownText()** function gets the current countdown timer text for the chart.

Example

```
s_UseTool Tool;
Tool.Text.Format("%s",sc.GetCountDownText().GetChars());
```

sc.GetCurrentDateTime()

Type: Function

```
SCDateTime GetCurrentDateTime();
```

The **sc.GetCurrentDateTime** function returns an [SCDateTime](#) variable which indicates the current Date-Time in the time zone of the chart the study is applied to.

This is obtained from the local system time. So it can be inaccurate if your system time is not set accurately.

During a chart replay, this time is calculated by Sierra Chart based upon the starting Date-Time in the chart where the replay began, the actual elapsed time, and the replay speed. In this particular case you need to be careful when using the returned value. If your computer and Sierra Chart are not able to keep up with the amount of data that needs to be processed during a fast replay, then this Date-Time can be significantly ahead of the Date-Time of the most recent added bar during a chart replay.

During a chart replay, this function returns the same value as [sc.CurrentDateTimeForReplay](#).

In general, this Date-Time should never be considered an accurate reference in relation to the last bar start or end Date-Time. And it should not be used during fast Back Testing because it would be inherently inaccurate and unstable because it is a calculated time based on time. The only built-in study which uses this is the **Countdown Timer** study and that is meant to be used for real-time updating and slow speed replays only and only provides indicative visual information.

To access the bar times themselves which is recommended, use the array [sc.BaseDateTimeln\[\]](#) for the bar beginning times, and the array [sc.BaseDataEndDateTime\[\]](#) for the bar ending times.

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

sc.GetCurrentTradedAskVolumeAtPrice()

Type: Function

```
uint32_t GetCurrentTradedAskVolumeAtPrice(float Price);
```

The **sc.GetCurrentTradedAskVolumeAtPrice** returns the Ask Volume traded at the given **Price** for the symbol of the chart of which the study instance that calls this function is applied to. For more detailed information, refer to [Chart/Trade DOM Column Descriptions](#).

For this data to be up-to-date and available, there must be a connection to the [data feed](#), or the chart needs to be [replaying](#).

sc.GetCurrentTradedBidVolumeAtPrice()

Type: Function

```
uint32_t GetCurrentTradedBidVolumeAtPrice(float Price);
```


The **sc.GetCurrentTradedBidVolumeAtPrice** returns the Bid Volume traded at the given **Price** for the symbol of the chart of which the study instance that calls this function is applied to. For more detailed information, refer to [Chart/Trade DOM Column Descriptions](#).

For this data to be up-to-date and available, there must be a connection to the [data feed](#), or the chart needs to be [replaying](#).

sc.GetCustomStudyControlBarButtonEnableState()

Type: Function

```
int sc.GetCustomStudyControlBarButtonEnableState()(int ControlBarButtonNum);
```

The **GetCustomStudyControlBarButtonEnableState()** function gets the enable state of the specified Advanced Custom Study Control Bar button.

Returns 1 if the button is enabled. Returns 0 if the button is disabled.

For further details about Advanced Custom study Control Bar buttons, refer to [Advanced Custom Study Buttons and Pointer Events](#).

Parameters

- **ControlBarButtonNum:** The integer number of the Advanced Custom study Control Bar button. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).

sc.GetDataDelayFromChart()

Type: Function

```
SCDateTime GetDataDelayFromChart(const int ChartNumber);
```

The **GetDataDelayFromChart()** function .

Parameters

- **ChartNumber:** .

Example

sc.GetDispersion()

Type: Intermediate Study Calculation Function

```
float GetDispersion(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetDispersion(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetDispersion()** function calculates the dispersion. The result is returned as a single float value.

Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

Example

```
float Dispersion = sc.GetDispersion(sc.BaseDataIn[SC_HIGH], 10);
```

sc.GetDOMColumnLeftCoordinate()

sc.GetDOMColumnRightCoordinate()

Type: Function

```
int GetDOMColumnLeftCoordinate(n_ACSIL::DOMColumnTypeEnum DOMColumn);
```

```
int GetDOMColumnRightCoordinate(n_ACSIL::DOMColumnTypeEnum DOMColumn);
```

The **sc.GetDOMColumnLeftCoordinate** and **sc.GetDOMColumnRightCoordinate** functions return the X-axis pixel coordinates for the left and right sides of the given **DOMColumn** on the current chart. If the requested **DOMColumn** does not exist on the chart, the functions will return a value of 0.

Valid values for the **DOMColumn** parameter can be found in the **DOMColumnTypeEnum** located in the **n_ACSIL** namespace.>/p>

- DOM_COLUMN_PRICE
- DOM_COLUMN_BUY_ORDER
- DOM_COLUMN_SELL_ORDER
- DOM_COLUMN_BID_SIZE
- DOM_COLUMN_ASK_SIZE
- DOM_COLUMN_COMBINED_BID_ASK_SIZE
- DOM_COLUMN_BID_SIZE_BUY
- DOM_COLUMN_ASK_SIZE_SELL
- DOM_COLUMN_LAST_SIZE
- DOM_COLUMN_CUMULATIVE_LAST_SIZE
- DOM_COLUMN_RECENT_BID_VOLUME

- DOM_COLUMN_RECENT_ASK_VOLUME
- DOM_COLUMN_CURRENT_TRADED_BID_VOLUME
- DOM_COLUMN_CURRENT_TRADED_ASK_VOLUME
- DOM_COLUMN_CURRENT_TRADED_TOTAL_VOLUME
- DOM_COLUMN_BID_MARKET_DEPTH_PULLING_STACKING
- DOM_COLUMN_ASK_MARKET_DEPTH_PULLING_STACKING
- DOM_COLUMN_COMBINED_BID_ASK_MARKET_DEPTH_PULLING_STACKING
- DOM_COLUMN_PROFIT_AND_LOSS
- DOM_COLUMN_SUBGRAPH_LABELS
- DOM_COLUMN_GENERAL_PURPOSE_1
- DOM_COLUMN_GENERAL_PURPOSE_2

One use of these functions is to draw into the general purpose Chart/Trade DOM columns by using the [Custom Free Form Drawing](#) operating system API.

Example

```
int GeneralPurposeColumnLeft = sc.GetDOMColumnLeftCoordinate(n_ACSIL::DOM_COLUMN_GENERAL
int GeneralPurposeColumnRight = sc.GetDOMColumnRightCoordinate(n_ACSIL::DOM_COLUMN_GENE
```

sc.GetEndingDateTimeForBarIndex()

Type: Function

```
double GetEndingDateTimeForBarIndex(int BarIndex);
```

The **sc.GetEndingDateTimeForBarIndex** function returns the ending Date-Time of a chart bar specified by its bar index. The chart bar index is specified with the **BarIndex** parameter.

The time returned has microsecond precision and can be assigned to a [SCDateTime](#) variable.

This Date-Time is calculated for bars other than at the last bar in the chart. For the last bar in the chart, the ending date time is known precisely if the

Global Settings >> Data/Trade Service Settings >> Intraday Data Storage Time Unit is **1 Tick** or **1 Second Per Bar** .

The chart [Session Times](#) are also used in the calculation where necessary. For example, if a chart bar is cut short because it has encountered the end of a specified Intraday session, the end of the session is used as the ending Date-Time for the bar.

sc.GetEndingDateTimeForBarIndexFromChart()

Type: Function

```
double GetEndingDateTimeForBarIndexFromChart(int ChartNumber, int BarIndex);
```

The **sc.GetEndingDateTimeForBarIndexFromChart** function returns the ending Date-Time of a chart bar specified by its bar index. This function can reference any chart in the same Chartbook containing the chart the study function is called from.

The return type is a double which can be directly assigned to a [SCDateTime](#) variable.

The chart is specified with the **ChartNumber** parameter. The chart bar index is specified with the **BarIndex** parameter.

The returned Date-Time is calculated for bars prior to the last bar in the chart. The calculation is simply the Date-Time of the following bar minus 1 second. However, if the particular time is outside of the Session Times, then it is adjusted to be within the Session Times as explained below.

For the last bar in the chart, the ending Date-Time is known precisely if the

Global Settings >> Data/Trade Service Settings >> Intraday Data Storage Time Unit is **1 Tick** or **1 Second Per Bar** and is the Date-Time of the last trade.

The chart [Session Times](#) are also used in the calculation where necessary. For example, if a chart bar is cut short because it has encountered the end of a specified Intraday session, the end of the session is used at the ending Date-Time for the bar.

sc.GetExactMatchForSCDateTime()

Type: Function

GetExactMatchForSCDateTime(int **ChartNumber**, SCDateTime **DateTime**);

sc.GetExactMatchForSCDateTime() returns the index into the Base Data arrays of the chart specified by **ChartNumber** that exactly matches the **DateTime**. If there is no exact match, this function returns **-1**.

This function can be used on the **sc.BaseData[]** arrays for the same chart that your study function is applied to when searching for the Date-Time. To do this, pass **sc.ChartNumber** for the **ChartNumber** parameter.

If **ChartNumber** is given as a negative number, the bar period and other Chart Settings are synchronized between the two charts. If it is positive, this does not occur. For example, if you want to get the index from chart #5, and you want to synchronize the charts, then use -5 for the **ChartNumber** parameter.

For complete information about the **SCDateTime** parameter type, refer to [SCDateTime](#).

Example

```
// Get the index into the Base Data of chart #2
// that exactly matches the DateTime given through the Input 0

// Get the DateTime from Input 0
SCDateTime DateTime = sc.Input[0].GetDateTime();

int Chart2Index = sc.GetExactMatchForSCDateTime(2, DateTime);

if(Chart2Index != -1)
{
    //Your Code
}
```

sc.GetFirstIndexForDate()

Type: Function

```
int GetFirstIndexForDate(int ChartNumber, int TradingDayDate);
```

The **sc.GetFirstIndexForDate()** function returns the first array index into the **sc.BaseDateTimeln[][]** array for the specified **ChartNumber** where the given **TradingDayDate** first occurs.

For an understanding of Trading day dates, refer to [Understanding Trading Day Dates Based on Session Times](#).

If there are no bars in the array matching the requested TradingDayDate, then the index of the first array index after the requested TradingDayDate is returned.

The **TradingDayDate** parameter is a [SCDateTime](#) type that contains the date only. If it contains a time, the time part will be ignored.

Example

```
FirstIndexOfReferenceDay = sc.GetFirstIndexForDate(sc.ChartNumber, ReferenceDay);

if (sc.GetTradingDayDate(sc.BaseDateTimeln[FirstIndexOfReferenceDay]) == ReferenceDay)
    --InNumberOfDaysBack;
```

sc.GetFirstNearestIndexForTradingDayDate()

Type: Function

```
int sc.GetFirstNearestIndexForTradingDayDate(int ChartNumber, int TradingDayDate);
```

The **sc.GetFirstNearestIndexForTradingDayDate()** function returns the first array index into the **sc.BaseDateTimeln[][]** array for the specified **ChartNumber** where the given **TradingDayDate** first occurs.

For an understanding of Trading day dates, refer to [Understanding Trading Day Dates Based on Session Times](#).

If there are no bars in the array matching the requested TradingDayDate, then the index with the date-time that is nearest to the given TradingDayDate is returned.

The **TradingDayDate** parameter is a [SCDateTime](#) type that contains the date only. If it contains a time, the time part will be ignored.

sc.GetFlatToFlatTradeListEntry()

Refer to the [sc.GetFlatToFlatTradeListEntry\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetFlatToFlatTradeListSize()

Refer to the [sc.GetFlatToFlatTradeListSize\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetGraphicsSetting()

Type: Function

```
int32_t GetGraphicsSetting(const int32_t ChartNumber, const  
n_ACSIL::GraphicsSettingsEnum GraphicsSetting, uint32_t& r_Color, uint32_t& r_LineWidth,  
SubgraphLineStyles& r_LineStyle);
```

The **sc.GetGraphicsSetting** function .

Parameters

- **ChartNumber:** .
- **GraphicsSetting:** .
- **r_Color:** .
- **r_LineWidth:** .
- **r_LineStyle:** .

Example



sc.GetGraphVisibleHighAndLow()

Type: Function

```
void GetGraphVisibleHighAndLow(double& High, double& Low);
```

The **sc.GetGraphVisibleHighAndLow** function determines the highest and lowest price values for the scale of the study instance from which this function is called at the time of the function call and puts that information into the referenced variables.

Parameters

- **High**: The highest price of the Scale for the study.
- **Low**: The lowest price of the Scale for the study.

sc.GetHideChartDrawingsFromOtherCharts()

Type: Function

```
int GetHideChartDrawingsFromOtherCharts(const int ChartNumber);
```

The **sc.GetHideChartDrawingsFromOtherCharts** function .

Parameters

- **ChartNumber**: .

Example

sc.GetHighest()

Type: Intermediate Study Calculation Function

```
float GetHighest(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetHighest(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetHighest()** function returns the highest value over the specified Length in the FloatArrayIn array.

Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

Example

```
//Get the highest high from the base graph over the last 20 bars
float Highest = sc.GetHighest(sc.BaseDataIn[SC_HIGH], 20);

//Get the highest value from sc.Subgraph[0] over the last 20 bars
Highest = sc.GetHighest(sc.Subgraph[0], 20);
```

sc.GetHighestChartNumberUsedInChartBook()

Type: Function

```
int GetHighestChartNumberUsedInChartBook();
```

The **sc.GetHighestChartNumberUsedInChartBook** function returns the highest Chart Number used in the Chartbook the chart the study function is applied to, belongs to.

Each chart has a number which is displayed on its title bar. This is its Chart Number for identification purposes.

sc.GetIndexOfHighestValue()

Type: Function

```
float GetIndexOfHighestValue(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetIndexOfHighestValue(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetIndexOfHighestValue()** function returns the [bar index](#) of the highest value over the specified Length in the FloatArrayIn array.

Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

sc.GetIndexOfLowestValue()

(SCFloatArrayRef In, int Length)

Type: Function

```
float GetIndexOfLowestValue(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetIndexOfLowestValue(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetIndexOfLowestValue()** function returns the [bar index](#) of the lowest value over the specified Length in the FloatArrayIn array.

Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

sc.GetIslandReversal()

Type: Intermediate Study Calculation Function

```
int GetIslandReversal(SCBaseDataRef BaseDataIn, int Index);
```

```
int GetIslandReversal(SCBaseDataRef BaseDataIn); Auto-looping only.
```

The **sc.GetIslandReversal()** function determines an Island Reversal. This function returns one of the following values:

0 = No Gap.

1 = Gap Up.

-1 = Gap Down.

Parameters

- [BaseDataIn](#).
- [Index](#).

Example

```
int IslandReversal = sc.GetIslandReversal(sc.BaseDataIn);
```

sc.GetLastFileErrorCode()

Type: Function

```
int GetLastFileErrorCode(const int FileHandle);
```

The **sc.GetLastFileErrorCode** function returns the last error code associated with the **FileHandle** parameter which is obtained by [sc.OpenFile\(\)](#).

The returned code is the standard operating system error code.

Additionally the following error codes are defined:

- ERROR_MISSING_PATH_AND_FILE_NAME = 0x20000001
- ERROR_NO_BUFFER_GIVEN_INPUT = 0x20000002
- ERROR_NO_BUFFER_GIVEN_OUTPUT = 0x20000003
- ERROR_NOT_ENOUGH_DATA_FOR_VALUE = 0x20000004
- ERROR_AT_END_OF_FILE = 0x20000005
- ERROR_END_OF_LINE_NOT_FOUND = 0x20000006
- ERROR_FILE_NOT_OPEN = 0x20000007
- ERROR_ALLOCATING_MEMORY = 0x20000008

sc.GetLastFileErrorMessage()

Type: Function

```
SCString GetLastFileErrorMessage(const int FileHandle);
```

The **sc.GetLastFileErrorMessage()** function returns the last error message, if there is one, associated with the File Handle defined by **FileHandle**.

If there is no error message, an empty string is returned.

sc.GetLastPriceForTrading()

Type: Function

```
double GetLastPriceForTrading();
```

GetLastPriceForTrading returns the most current last trade price for the Symbol of the chart either from the connected data feed or from the last bar in the chart if the data feed price is 0 meaning it is not available.

This function also works during a chart replay.

Example

```
double LastPrice = sc.GetLastPriceForTrading();
```

sc.GetLatestBarCountdownAsInteger()

Type: Function

```
int scGetLatestBarCountdownAsInteger();
```

The **sc.GetLatestBarCountdownAsInteger()** function returns the remaining time/value for the most recent chart bar until it is considered finished, as an integer value.

Example

```
int CountdownValue = sc.GetLatestBarCountdownAsInteger();  
RemainingAmountForSubgraphValue = (float)CountdownValue;  
  
Tool.Text.Format("%d",CountdownValue);
```

sc.GetLineNumberOfSelectedUserDrawnDrawing

Refer to the [sc.GetLineNumberOfSelectedUserDrawnDrawing\(\)](#) section on the [Using Drawing Tools From an Advanced Custom Study](#) page for information on this function.

sc.GetLowest()

Type: Intermediate Study Calculation Function

```
float GetLowest(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetLowest(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetLowest()** function returns the lowest value over the specified Length in the FloatArrayIn array.

Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

Example

```
float Lowest = sc.GetLowest(sc.BaseDataIn[SC_LOW], 20);  
Lowest = sc.GetLowest(sc.Subgraph[0], 20);
```

sc.GetMainGraphVisibleHighAndLow()

Type: Function

```
GetMainGraphVisibleHighAndLow (float& High, float& Low);
```

sc.GetMainGraphVisibleHighAndLow will get the highest High and lowest Low for the visible bars of the main price graph in the chart.

Example

```
float High, Low;  
sc.GetMainGraphVisibleHighAndLow(High,Low);
```

sc.GetMarketDepthBars()

Type: Function

```
c_ACSILDepthBars* GetMarketDepthBars();
```

sc.GetMarketDepthBars returns access to the historical market depth bars that are on the same chart that your custom study is applied to. The return value is a pointer to a c_ACSILDepthBars object, which has various functions for retrieving data from the historical market depth bars. See the [c_ACSILDepthBars class](#) for details.

sc.GetMarketDepthBars should never return a null pointer, but it is still good practice to check

to make sure the returned pointer is not null before using it.

sc.GetMarketDepthBarsFromChart()

Type: Function

```
c_ACSILDepthBars* GetMarketDepthBarsFromChart(int ChartNumber);
```

sc.GetDepthBarFromChart returns access to the historical market depth bar from the chart that matches the given **ChartNumber** parameter. The return value is a pointer to a `c_ACSILDepthBars` object, which has various functions for retrieving data from the historical market depth bars. See the [c_ACSILDepthBars class](#) for details.

sc.GetDepthBarFromChart should never return a null pointer, but it is still good practice to check to make sure the returned pointer is not null before using it.

Parameters

- **ChartNumber:** The number of the chart you want to get data from. This is the number that is shown on the top line in the Chart window, after the #. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart the study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.

sc.GetMaximumMarketDepthLevels

Type: Function

```
int GetMaximumMarketDepthLevels();
```

The **sc.GetMaximumMarketDepthLevels** function returns the maximum number of available market depth levels for the symbol of the chart. This will be the maximum for both the bid and ask sides, whichever is greater.

sc.GetNearestMatchForDateTimeIndex()

Type: Function

```
int GetNearestMatchForDateTimeIndex(int ChartNumber, int DateTimeIndex);
```

sc.GetNearestMatchForDateTimeIndex() returns the index into the Base Data arrays of the chart specified by **ChartNumber** with the Date-Time closest to the Date-Time at the index specified by **DateTimeIndex** in the chart your study is applied to. If the Date-Time at **DateTimeIndex** is before any Date-Time in the specified chart, then the index of the first element is given (0). If the Date-Time at **DateTimeIndex** is after any Date-Time in the specified chart, then the index of the last element is given (`sc.ArraySize - 1`).

This function can be used on the **sc.BaseData[]** arrays for the same chart that your study function is applied to when searching for the Date-Time. To do this, pass **sc.ChartNumber** for the **ChartNumber** parameter.

If **ChartNumber** is given as a negative number, the bar period and other Chart Settings are synchronized between the two charts. If it is positive, this does not occur. For example, if you want to get the index from chart #5, and you want to synchronize the charts, then use -5 for the **ChartNumber** parameter.

Error Return Value

This function will return -1 if the chart it is referencing does not exist or its data is not fully loaded.

When the chart it is referencing is opened and fully loaded, then the study instance that made this call will be called again and will be fully recalculated. What actually happens in this case, is that the chart the reference was made from, will have all of its studies recalculated.

Matching Rules for a Nearest Date-Time Match

Refer to the [Matching Rules for a Nearest Date-Time Match](#) section.

Example

```
// Get the index in the Base Data arrays for chart #2 that
// has the nearest matching Date-Time to the Date-Time
// at the current index being calculated for the chart that
// this study is on.

int Chart2Index = sc.GetNearestMatchForDateTimeIndex(2, sc.Index);

SCGraphData ReferenceArrays;
sc.GetChartBaseData(2, ReferenceArrays);

if (ReferenceArrays[SC_HIGH].GetArraySize() < 1)//Array is empty, nothing to do.

    return;

//Get the corresponding High
float High = ReferenceArrays[SC_HIGH][Chart2Index];
```

sc.GetNearestMatchForSCDateTime()

Type: Function

```
int GetNearestMatchForSCDateTime(int ChartNumber, SCDateTime DateTime);
```

sc.GetNearestMatchForSCDateTime() returns the index into the Base Data arrays of the chart specified by the **ChartNumber** parameter with the Date-Time closest to the **DateTime** parameter.

If the specified **DateTime** is before any Date-Time in the specified chart, then the index of the first element is given (0). If **DateTime** is after any Date-Time in the specified chart, then the index of the last element is given (**sc.ArraySize** - 1).

The extended array elements of the chart in the forward projection area of the chart are not included in the search.

This function can be used on the **sc.BaseData[]** arrays for the same chart that your study function is applied to when searching for the Date-Time. To do this, pass **sc.ChartNumber** for the **ChartNumber** parameter.

If **ChartNumber** is given as a negative number, the bar period and other Chart Settings are synchronized between the two charts. If it is positive, this does not occur. For example, if you want to get the index from chart #5, and you want to synchronize the charts, then use -5 for the **ChartNumber** parameter.

For complete information about the **SCDateTime** parameter type, refer to [SCDateTime](#).

Error Return Value

This function will return -1 if the chart it is referencing does not exist or its data is not fully loaded.

When the chart it is referencing is opened and fully loaded, then the study instance that made this call will be called again and will be fully recalculated. What actually happens in this case, is that the chart the reference was made from, will have all of its studies recalculated.

Matching Rules for a Nearest Date-Time Match

1. If an exact match can be done, then the index with the exact match will be returned. If there are repeating timestamps, then the first index in the repeating times is given.
2. If there is not an exact match, then the index of the nearest matching date-time is given.
3. If the given date-time is equidistant between two date-times, then the index for the higher date-time is given.

Example

```

// Get the index in the Base Data arrays for chart #2 that
// has the nearest matching Date-Time to the given DateTime.

SCDateTime DateTime = sc.BaseDateTimeIn[sc.Index];

int Chart2Index = sc.GetNearestMatchForSCDateTime(2, DateTime);

SCGraphData ReferenceArrays;
sc.GetChartData(2, ReferenceArrays);

if (ReferenceArrays[SC_HIGH].GetArraySize() < 1)//Array is empty, nothing to do.

    return;

//Get the corresponding High
float High = ReferenceArrays[SC_HIGH][Chart2Index];

```

sc.GetNearestMatchForSCDateTimeExtended()

Type: Function

```
int GetNearestMatchForSCDateTimeExtended(int ChartNumber, const SCDateTime&
DateTime);
```

The **sc.GetNearestMatchforSCDateTimeExtended()** function is identical to the [sc.GetNearestMatchForSCDateTime\(\)](#) function, except that it also searches the extended array elements of the chart in the forward projection area.

For complete documentation for this function, refer to [sc.GetNearestMatchForSCDateTime\(\)](#). The only other difference has been explained here.

For a complete example to use this function, refer to the **scsf_VerticalDateTimeLine** function in the **/ACS_Source/studies2.cpp** file in the folder Sierra Chart is installed to.

sc.GetNearestStopOrder()

Refer to the [sc.GetNearestStopOrder\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetNearestTargetOrder()

Refer to the [sc.GetNearestTargetOrder\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetNumberOfBaseGraphArrays()

Type: Function

```
int GetNumberOfBaseGraphArrays();
```

The **sc.GetNumberOfBaseGraphArrays()** function gets the number of arrays allocated in **sc.BaseData[]**.

Example

```
int NumberOfBaseDataArrays = sc.GetNumberOfBaseGraphArrays();
```

sc.GetNumberOfDataFeedSymbolsTracked()

Type: Function

```
int GetNumberOfDataFeedSymbolsTracked();
```

The **sc.GetNumberOfDataFeedSymbolsTracked()** function returns the number of symbols Sierra Chart is currently tracking through the connected data feed. This function has no parameters.

For additional information, refer to [Status Bar](#).

sc.GetNumPriceLevelsForStudyProfile()

Type: Function

```
int GetNumPriceLevelsForStudyProfile(int StudyID, int ProfileIndex);
```

The **sc.GetNumPriceLevelsForStudyProfile** function is to be used with the [sc.GetVolumeAtPriceDataForStudyProfile](#) function. It is used to return the number of price levels contained within a Volume Profile.

These Volume Profiles are from a [TPO Profile Chart](#) or [Volume by Price](#) study on the chart.

The function returns the number of price levels within the specified Volume Profile.

Parameters

- **StudyID:** The unique study identifier for the **Volume by Price** or **TPO Profile Chart** study. Refer to [UniqueStudyInstanceIdentifiers](#).
- **ProfileIndex:** The zero-based index of the volume profile relative to the end of the chart. 0 equals the latest profile in the chart at the end or rightmost side. This needs to always be set to a positive number.

sc.GetNumStudyProfiles()

Type: Function

```
int GetNumStudyProfiles(int StudyID);
```

sc.GetNumStudyProfiles will get the number of TPO Profiles or Volume Profiles for the [TPO Profile Chart](#) and the [Volume by Price](#) study respectively in the instance of the study specified by the **StudyID** parameter.

Parameters

- **StudyID**: The unique ID for the study to get the number of study profiles for.
For more information, refer to [Unique Study Instance Identifiers](#).

Example

```
int NumProfiles = sc.GetNumStudyProfiles(1);
```

sc.GetOHLCForDate()

Type: Function

GetOHLCForDate(double **Date**, float& **Open**, float& **High**, float& **Low**, float& **Close**);

The **sc.GetOHLCForDate()** function returns the Open, High, Low and Close of the period of time in an Intraday chart that is for the date specified by the **Date** parameter.

The Period of time is 1 Day. The starting and ending times are controlled by the [Session Times](#) settings in **Chart >> Chart Settings** for the Chart.

The end of the day is the time set by the **End Time**. The day includes the 24 hours prior to that time.

For efficiency, this function should only be called once a day while iterating through the chart bars in the study function. The values returned should be saved for the next iteration until a new day is encountered according to the Session Times.

Example

```
float Open;  
float High;  
  
float Low;  
float Close;  
  
sc.GetOHLCForDate(sc.BaseDateTimeIn[sc.ArraySize-1], Open, High, Low, Close);  
  
SCString Message;  
Message.Format("O: %f, H: %f, L: %f , C: %f",Open,High,Low,Close);  
  
sc.AddMessageToLog(Message,1);
```

sc.GetOHLCOfTimePeriod()

Type: Function

int GetOHLCOfTimePeriod(SCDateTime StartDateTime, SCDateTime EndDateTime, float& Open, float& High, float& Low, float& Close, float& NextOpen);

sc.GetOHLCOfTimePeriod() returns the Open, High, Low, Close and NextOpen of the period of time in an Intraday chart specified by the **StartDateTime** and **EndDateTime** parameters.

Example

```
// In this example these are not set to anything. You will need to
// set them to the appropriate starting DateTime and ending DateTime
SCDateTime dt_StartTime, dt_EndTime;

float Open;
float High;
float Low;
float Close;

float NextOpen;

sc.GetOHLCOfTimePeriod( dt_StartTime, dt_EndTime, Open, High, Low, Close, NextOpen );
```

sc.GetOpenHighLowCloseVolumeForDate()

```
int GetOpenHighLowCloseVolumeForDate(double Date, float& r_Open, float& r_High, float&
r_Low, float& r_Close, float& r_Volume);
```

The `sc.GetOpenHighLowCloseVolumeForDate` function returns the Open, High, Low, Close and Volume values for the specified trading day **Date**.

This function is affected by the chart [Session Times](#) when determining the specific date-time range for these values.

This function returns 1 if it is successful or 0 if an error was encountered or the date is not found.

sc.GetOrderByIndex()

Refer to the [sc.GetOrderByIndex\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetOrderByOrderID()

Refer to the [sc.GetOrderByOrderID\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetOrderFillArraySize()

Refer to the [sc.GetOrderFillArraySize\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetOrderFillEntry()

Refer to the [sc.GetOrderFillEntry\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetOrderForSymbolAndAccountByIndex()

Refer to the [sc.GetOrderForSymbolAndAccountByIndex\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetParentOrderIDFromAttachedOrderID()

Type: Function

```
void GetParentOrderIDFromAttachedOrderID(int AttachedOrderInternalOrderID);
```

The **sc.GetParentOrderIDFromAttachedOrderID** function is to return the parent Internal Order ID for the given Attached Order Internal Order ID.

For information about Internal Order IDs, refer to the [ACSIL Trading](#) page. These Internal Order IDs can be obtained when submitting an order.

sc.GetPersistentDouble()

sc.SetPersistentDouble()

```
double& GetPersistentDouble(int Key);
```

```
void SetPersistentDouble(int Key, double Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

sc.GetPersistentDoubleFromChartStudy()

sc.SetPersistentDoubleForChartStudy()

```
double& GetPersistentDoubleFromChartStudy(int ChartNumber, int StudyID, int Key);
```

```
void SetPersistentDoubleForChartStudy(int ChartNumber, int StudyID, int Key, double Value);
```

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

sc.GetPersistentFloat()

sc.SetPersistentFloat()

```
float& GetPersistentFloat(int Key);
```

```
void SetPersistentFloat(int Key, float Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

sc.GetPersistentFloatFromChartStudy()

sc.SetPersistentFloatForChartStudy()

```
float& GetPersistentFloatFromChartStudy(int ChartNumber, int StudyID, int Key);
```

```
void SetPersistentFloatForChartStudy(int ChartNumber, int StudyID, int Key, float Value);
```

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

sc.GetPersistentInt()

sc.SetPersistentInt()

```
int& GetPersistentInt(int Key);
```

```
void SetPersistentInt(int Key, int Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

sc.GetPersistentIntFromChartStudy()

sc.SetPersistentIntForChartStudy()

```
int& GetPersistentIntFromChartStudy(int ChartNumber, int StudyID, int Key);
```

```
void SetPersistentIntForChartStudy(int ChartNumber, int StudyID, int Key, int Value);
```

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

sc.GetPersistentInt64()

sc.SetPersistentInt64()

```
__int64& GetPersistentInt64(int Key);
```

```
void SetPersistentInt64(int Key, __int64 Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

sc.GetPersistentInt64FromChartStudy()

sc.SetPersistentInt64ForChartStudy()

```
__int64& GetPersistentInt64FromChartStudy(int ChartNumber, int StudyID, int Key);
```

```
void SetPersistentInt64ForChartStudy(int ChartNumber, int StudyID, int Key, __int64 Value);
```

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

sc.GetPersistentPointer()

sc.SetPersistentPointer()

```
void*& GetPersistentPointer(int Key);
```

```
void SetPersistentPointer(int Key, void* Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

sc.GetPersistentPointerFromChartStudy()

sc.SetPersistentPointerForChartStudy()

```
void*& GetPersistentPointerFromChartStudy(int ChartNumber, int StudyID, int Key);
```

```
void SetPersistentPointerForChartStudy(int ChartNumber, int StudyID, int Key, void* Value);
```

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

sc.GetPersistentSCDateTime()

sc.SetPersistentSCDateTime()

```
SCDateTime& GetPersistentSCDateTime(int Key);
```

```
void SetPersistentSCDateTime(int Key, SCDateTime Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

sc.GetPersistentSCDateTimeFromChartStudy()

sc.SetPersistentSCDateTimeForChartStudy()

```
SCDateTime& GetPersistentSCDateTimeFromChartStudy(int ChartNumber, int StudyID, int Key);
```

```
void SetPersistentSCDateTimeForChartStudy(int ChartNumber, int StudyID, int Key, SCDateTime Value);
```

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

sc.GetPersistentSCString()

sc.SetPersistentSCString()

```
SCString& GetPersistentSCString(int Key);
```

```
void SetPersistentSCString(int Key, SCString Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

Persistent Variable Functions

The **GetPersistent*** functions are for getting a reference to a persistent variable identified by the **Key** parameter. The **Key** parameter is an integer and can be any integer value in the range 0 to INT_MAX. It can also be a negative number in the range of 0 to INT_MIN.

The **SetPersistent*** functions are for setting a value into a persistent variable identified by the **Key** parameter. The **Key** parameter is an integer and can be any integer value in the range 0 to INT_MAX. It can also be a negative number in the range of 0 to INT_MIN.

Since the **GetPersistent*** functions return a reference to a persistent variable, usually this reference is what is used to set a value into the persistent variable and normally the **SetPersistent*** functions will not be used.

The persistent variables are specific to each individual study instance. Each study has its own persistent variable storage.

The following example code explains how to define a variable which holds a reference to the reference returned.

```
int& Variable = sc.GetPersistentInt(1);
```

All of the basic data types that Sierra Chart works with are supported for persistent variables.

The initial value for all persistent variables when they have not been previously set, is 0.

You may want to have the value in a variable remain persistent between calls to your study function. For example, when using [Automatic-Looping](#) (the default and recommended method of looping) your study function is called for every bar in the chart. If you want to have a variable retain its value between calls into your study function, then you need to use persistent variables.

For each data type, up to 50000 persistent variables are supported. You can use any **Key** value to identify these 50000 values. When the 50000 values have been exceeded, getting or setting a persistent variable with a new key which exceeds the limit will only set the internal dummy value which is not persistent. Although using this number of persistent variables will have a performance impact and it is recommended to avoid if possible having this large number of persistent variables.

The persistent variables remain persistent between calls to your study function. However, they are not saved when a Chartbook is saved and closed, or when Sierra Chart is closed. They are unique to each and every instance of your study.

Persistence of Persistent Variables: Persistent variables remain persistent until the study is removed from the chart or when a chart is closed either individually or when a Chartbook is closed that the chart is part of.

Therefore, when a study is added to a chart either through the Chart Studies window or through Study Collections, persistent variables are initially set to 0 for that study. When a chart that is saved as part of a chartbook is opened, all of the persistent variables for all of the studies on that chart, are set to 0.

You can get and set persistent variables in the [sc.SetDefaults](#) code block at the top of the study function.

If you need to have permanent storage that is saved when a Chartbook is saved, then you will need to use the [sc.StorageBlock](#) instead of persistent variables.

Also refer to [Chart Study Persistent Variable Functions](#) for functions to get/set persistent data from other studies on the chart or different charts.

If these functions which get references to persistent data do not meet your requirements, then you will need to allocate your own memory. Refer to [Dynamic Memory Allocations](#).

References to Persistent Variables

You can use references to give names to the persistent variables by defining new variables that refer to the persistent variables. This is helpful because it makes your code easier to understand and work with. This really is the usual way in which persistent variables should be worked with since the `sc.GetPersistent*` functions always return a reference. Refer to the code example below for how to do this.

```
// Use persistent variables to remember attached order IDs so they can be modified or canceled.
int& Target1OrderID = sc.GetPersistentInt(1);
int& Stop1OrderID = sc.GetPersistentInt(2);

sc.SetPersistentFloat(1, 25.4f);
```

Resetting Persistent Variables Example

Usually you will want to reset persistent variables back to default values when a study is recalculated. This does not happen automatically and must be specifically programmed when there is an indication a study is being fully recalculated. This can be done by using the following code below.

```
if (sc.IsFullRecalculation && sc.Index == 0)//This indicates a study is being recalculated.
{
    // When there is a full recalculation of the study,
    // reset the persistent variables we are using
    sc.GetPersistentInt(0) = 0;
    sc.GetPersistentInt(1) = 0;
    sc.GetPersistentInt(2) = 0;
    sc.GetPersistentInt(3) = 0;
}
```

References to SCDatetime Variables

[SCDateTime](#) and [SCDateTimeMS](#) variables internally are of the same type.

`sc.GetPersistentSCDateTime` can be used to obtain a persistent variable for either of these types.

sc.GetPersistentDoubleFast()

```
double& GetPersistentDoubleFast(int32_t Index);
```

Refer to the common [Fast Persistent Variable Functions](#) documentation.

sc.GetPersistentFloatFast()

```
float& GetPersistentFloatFast(int32_t Index);
```

Refer to the common [Fast Persistent Variable Functions](#) documentation.

sc.GetPersistentIntFast()

```
int& GetPersistentIntFast(int32_t Index);
```

Refer to the common [Fast Persistent Variable Functions](#) documentation.

sc.GetPersistentSCDateTimeFast()

```
SCDateTime& GetPersistentSCDateTimeFast(int32_t Index);
```

Refer to the common [Fast Persistent Variable Functions](#) documentation.

Fast Persistent Variable Functions

The **sc.GetPersistent*Fast** functions perform in the same way as the **sc.GetPersistent*** functions except that the retrieval of the persistent variable is faster. This makes a difference when there are hundreds of persistent variables being used of a particular type.

The other difference is they use an **Index** parameter as the key in the range of 1 to 10,000. Start at zero and increment up to 9999 according to the number of variables needed.

Minimize the number of calls into the persistent variable functions as much as possible whether the fast ones are used or not. The best performance is achieved when using [manual looping](#). Although if there are less than 50 persistent variables used within a function, the lookup of these variables is very fast and next to no processing time whether the fast persistent variable functions are used or not.

For complete documentation to use the persistent variable functions, refer to [Persistent Variable Functions](#)

```
int& IntValue = sc.GetPersistentIntFast(0);  
IntValue = sc.Index;
```

Chart Study Persistent Variable Functions

The **sc.GetPersistent*FromChartStudy** and **sc.SetPersistent*ForChartStudy** functions are identical to the [GetPersistent*](#) and the [SetPersistent*](#) functions except that they will get and set a persistent variable from/for the study specified by the **ChartNumber** and **StudyID** parameters.

If the **ChartNumber** parameter is not valid, then an internal dummy variable is referenced with a value of zero and will not have the intended effect.

If setting or getting a persistent variable in another chart (Source chart), then when that Source chart is updated for any reason, the chart that made the reference to it (Destination chart), will have its studies calculated. In this way your study on the Destination chart will be aware of the changes to the persistent variable in a Source chart. When running a multiple chart Back Test, charts will be calculated in the proper order as well.

Refer to the code example below for usage.

```
SCSFExport scsf_GetStudyPersistentVariableFromChartExample(SCStudyInterfaceRef sc)
{
    SCInputRef ChartStudyReference = sc.Input[0];

    if (sc.SetDefaults)
    {
        // Set the configuration and defaults

        sc.GraphName = "Get Study Persistent Variable from Chart Example";
        sc.AutoLoop = 1;

        ChartStudyReference.Name = "Chart Study Reference";
        ChartStudyReference.SetChartStudyValues(1, 0);

        return;
    }

    //Get a reference to a persistent variable with key value 100 in the chart and study specified by the ChartStudyF
    float & FloatFromChartStudy = sc.GetPersistentFloatFromChartStudy(ChartStudyReference.GetChartNumber(
}
```

sc.GetPointOfControlAndValueAreaPricesForBar()

Type: Function

```
int GetPointOfControlAndValueAreaPricesForBar(int BarIndex, double& r_PointOfControl,
double& r_ValueAreaHigh, double& r_ValueAreaLow, float ValueAreaPercentage);
```

The **sc.GetPointOfControlAndValueAreaPricesForBar()** function gets the price values for the Point of Control, Value Area High, and Value Area Low into their respective variables, for the given BarIndex and ValueAreaPercentage.

Parameters

- **BarIndex**: The Index of the chart bar for which the Point of Control and Value Areas High and Low are requested.
- **r_PointOfControl**: The returned price level of the Point of Control for the given BarIndex.
- **r_ValueAreaHigh**: The returned price level of the Value Area High for the **ValueAreaPercentage**.
- **r_ValueAreaLow**: The returned price level of the Value Area Low for the **ValueAreaPercentage**.
- **ValueAreaPercentage**: The Value Area Percentage to be used for the Value

Area Calculations. This number is to be entered as a percentage value (example 80.5).

sc.GetPointOfControlPriceVolumeForBar()

Type: Function

```
void GetPointOfControlPriceVolumeForBar(int BarIndex, s_VolumeAtPriceV2&  
VolumeAtPrice);
```

The **sc.GetPointOfControlPriceVolumeForBar** function fills out a **s_VolumeAtPriceV2** structure object passed to the function for the Point of Control price level based on volume for the chart bar at the index specified by **BarIndex**.

It is necessary to set **sc.MaintainVolumeAtPriceData = 1;** in the **sc.SetDefaults** code block at the top of the function in order to have the volume at price data maintained so that this function returns valid data.

Parameters

- **BarIndex:** The Index of the chart bar for which the Point of Control volume data is requested.
- **VolumeAtPrice:** A **s_VolumeAtPriceV2** structure that is filled in with the volume data at the price level of the Point Of Control of the chart bar.

Example

```
s_VolumeAtPriceV2 VolumeAtPrice;  
sc.GetPointOfControlPriceVolumeForBar(sc.Index, VolumeAtPrice);
```

sc.GetProfitManagementStringForTradeAccount()

Type: Function

```
void GetProfitManagementStringForTradeAccount(SCString& r_TextString);
```

The **sc.GetProfitManagementStringForTradeAccount** function returns the Profit/Loss Status text string for the Trade Account the chart is set to, from the [Global Profit/Loss Management](#), into the variable **r_TextString**.

For an example of how this function is used, refer to the study function **scsf_TradingProfitManagementStatus** in the **/ACS_Source/studies5.cpp** file in the Sierra_Chart installation folder.

sc.GetRealTimeSymbol()

Type: Function

```
SCString& GetRealTimeSymbol();
```

Example

**sc.GetRecentAskVolumeAtPrice() /
sc.GetRecentBidVolumeAtPrice()**

Type: Function

```
unsigned int GetRecentAskVolumeAtPrice(float Price);
```

```
unsigned int GetRecentBidVolumeAtPrice(float Price);
```

The **sc.GetRecentAskVolumeAtPrice** and **sc.GetRecentBidVolumeAtPrice** functions return the recent Ask or Bid volume respectively, for the given **Price** parameter. This is one of the market data columns on the Chart/Trade DOM. This data is maintained from the real-time market data and also during a chart replay.

For a description of this particular market data column, refer to [Recent Bid Volume/Recent Ask Volume](#).

For these functions to return a volume value, it is necessary

Trade >> Trading Chart DOM On is enabled for one of the charts for the Symbol. And the **Recent Bid/Ask Volume** columns need to be added. Refer to [Customize Trade/Chart DOM Columns and Descriptions](#) for instructions.

Example

```
unsigned int RecentAskVolume = sc.GetRecentAskVolumeAtPrice(sc.LastTradePrice);
```

sc.GetReplayHasFinishedStatus()

Type: Function

```
int GetReplayHasFinishedStatus();
```

When a [Chart Replay](#) has finished in a chart, at that time the study function will be called one more time and you can check that it has finished, by calling the **sc.GetReplayHasFinishedStatus** function.

The return value of this function will be 1, if the replay has just finished. If it is 0, then the replay has not finished or the finished state has been cleared. A return value of 1 is only going to be provided for one chart calculation.

sc.GetReplayStatusFromChart()

Type: Function

```
int32_t GetReplayStatusFromChart(int ChartNumber);
```

The **sc.GetReplayStatusFromChart** function returns one of the following values indicating the replay status for the specified Chart Number.

- **REPLAY_STOPPED** = 0
- **REPLAY_RUNNING** = 1
- **REPLAY_PAUSED** = 2

Parameters

- **ChartNumber:** The number of the chart to get the replay status for.
Refer to [Chart Number](#).

sc.GetSessionTimesFromChart()

Type: Function

```
int GetSessionTimesFromChart(const int ChartNumber,  
n_ACSIL::s_ChartSessionTimes& r_ChartSessionTimes);
```

The **sc.GetSessionTimesFromChart()** function sets the chart session times in **r_ChartSessionTimes** for the chart defined by **ChartNumber**.

Parameters

- **ChartNumber:** The number of the chart for the session times.
- **r_ChartSessionTimes:** A **s_ChartSessionTimes** structure that contains the session times for the given chart. The **s_ChartSessionTimes** members are the following:
 - **SCDateTime StartTime**
 - **SCDateTime EndTime**
 - **SCDateTime EveningStartTime**
 - **SCDateTime EveningEndTime**
 - **int UseEveningSessionTimes**
 - **int NewBarAtSessionStart**
 - **int LoadWeekendDataSetting**

sc.GetSheetCellAsDouble()

Type: Function

```
int GetSheetCellAsDouble(void* SheetHandle, const int Column, const int Row, double&  
r_CellValue);
```

The **sc.GetSheetCellAsDouble()** function places the value of the requested Spreadsheet Sheet Cell in the double variable **r_CellValue**.

If the cell does not contain a value, then this function returns a value of **0**. Otherwise, it returns a value of **1**.

Parameters

- **SheetHandle**: The handle of the Spreadsheet Sheet as returned by the [sc.GetSpreadsheetSheetHandleByName\(\)](#) function.
- **Column**: The column number for the Sheet Cell to get the value from. This is a zero-based array, so column B in the Sheet would be a value of 1.
- **Row**: The row number for the Sheet Cell to get the value from. This is a zero-based array, so row 2 in the Sheet would be a value of 1.
- **r_CellValue**: A reference to a double variable that receives the value of the Sheet Cell.

Also refer to the following functions: [sc.SetSheetCellAsDouble\(\)](#), [sc.GetSheetCellAsString\(\)](#), [sc.SetSheetCellAsString\(\)](#), [sc.GetSpreadsheetSheetHandleByName\(\)](#).

Example

```
const char* SheetCollectionName = "ACSILInteractionExample";

const char* SheetName = "Sheet1";

void* SheetHandle = sc.GetSpreadsheetSheetHandleByName(SheetCollectionName, SheetName,

// Get the result from cell B4. Column and row indexes are zero-based.
double CellValue = 0.0;
sc.GetSheetCellAsDouble(SheetHandle, 1, 3, CellValue);
```

sc.GetSheetCellAsString()

Type: Function

```
int GetSheetCellAsString(void* SheetHandle, const int Column, const int Row, SCString&
r_CellString);
```

The **sc.GetSheetCellAsString()** function places the value of the requested Spreadsheet Sheet Cell in the SCString variable **r_CellValue**.

If the cell does not contain a value, then this function returns a value of **0**. Otherwise, it returns a value of **1**.

Parameters

- **SheetHandle**: The handle of the Spreadsheet Sheet as returned by the [sc.GetSpreadsheetSheetHandleByName\(\)](#) function.
- **Column**: The column number for the Sheet Cell to get the value from. This is a zero-based array, so column B in the Sheet would be a value of 1.
- **Row**: The row number for the Sheet Cell to get the value from. This is a zero-based array, so row 2 in the Sheet would be a value of 1.
- **r_CellString**: A reference to a SCString variable that receives the value of the Sheet Cell.

Also refer to the following functions: [sc.GetSheetCellAsDouble\(\)](#), [sc.SetSheetCellAsDouble\(\)](#), [sc.SetSheetCellAsString\(\)](#), [sc.GetSpreadsheetSheetHandleByName\(\)](#).

Example

```
const char* SheetCollectionName = "ACSILInteractionExample";

const char* SheetName = "Sheet1";

void* SheetHandle = sc.GetSpreadsheetSheetHandleByName(SheetCollectionName, SheetName, fa

// Get the text from cell B6, if it exists, and add it to the message log.
SCString CellString;
if (sc.GetSheetCellAsString(SheetHandle, 1, 5, CellString))
    sc.AddMessageToLog(CellString, 0);
```

sc.GetSpreadsheetSheetHandleByName()

Type: Function

```
void* GetSpreadsheetSheetHandleByName(const char* SheetCollectionName, const
char* SheetName, const int CreateSheetIfNotExist);
```

The **GetSpreadsheetSheetHandleByName()** function returns a pointer to the Spreadsheet Handle for the given **SheetCollectionName** and **SheetName**.

Parameters

- **SheetCollectionName**: Either the complete path and file extension for the Spreadsheet, or just the name of the spreadsheet file itself without the extension if the Spreadsheet is located in the Sierra Chart [Data Files Folder](#).
- **SheetName**: The Sheet name as found within the Spreadsheet.
- **CreateSheetIfNotExist**: If this value is non-zero (true), then the

specified SheetName will be created if it does not already exist within the SheetCollectionName.

Also refer to the following functions: [sc.GetSheetCellAsDouble\(\)](#), [sc.SetSheetCellAsDouble\(\)](#), [sc.GetSheetCellAsString\(\)](#), [sc.SetSheetCellAsString\(\)](#).

Example

```
void* SheetHandle = sc.GetSpreadsheetSheetHandleByName(SheetCollectionName, SheetName, fa
```

sc.GetStandardError()

Type: Intermediate Study Calculation Function

```
double GetStandardError(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
double GetStandardError(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetStandardError()** function calculates the Standard Error. The result is returned as a double precision float value.

Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

Example

```
double StandardErrorValue = sc.GetStandardError(sc.BaseDataIn[SC_LAST], Length.GetInt());
```

sc.GetStartDateTimeForTradingDate()

Type: Function

```
double GetStartDateTimeForTradingDate(int TradingDate);
```

The **sc.GetStartDateTimeForTradingDate()** function returns a Date-Time as a double which can be assigned to a SCDateTime variable, which is the starting Date and Time of the trading session that the given **TradingDate** is within. **TradingDate** is considered the date of the trading day independent of the actual date of trading. For example, trading during the evening session the day before next days Day session will have a TradingDate 1 day ahead of the actual date of trading. Therefore, if a trading session spans over midnight according to the Session Times of the chart the study is applied to, then the Date of the returned Date-

Time will always be one less than the given **TradingDate**.

Example

```
SCDateTime TradingDate;  
SCDateTime SessionStartDateTime = sc.GetStartDateTimeForTradingDate(TradingDate);
```

sc.GetStartOfPeriodForDateTime()

Type: Function

SCDateTime **GetStartOfPeriodForDateTime**(SCDateTime **DateTime**, unsigned int **TimePeriodType**, int **TimePeriodLength**, int **PeriodOffset**);

The **sc.GetStartOfPeriodForDateTime()** function gets the beginning of a time period of a length specified by the **TimePeriodType** and **TimePeriodLength** parameters, for a given Date-Time.

The start of a time period will always be aligned to the session Start Time in the case of Intraday charts. For periods based on more than one day, the starting reference point will be 1950-1-1. For periods based on weeks, the start will be either Sunday or Monday depending upon the setting of **Global Settings >> General Settings >> General >> Data >> Use Monday as Start of Week instead of Sunday**. For months, the start will be the first day of the month. For years, the start will be January 1.

The specified time length, will organize time into blocks of the specified length. For example, if the specified time length is 30 minutes, time will be organized this way, assuming the Session Times in **Chart >> Chart Setting** use a Start Time that begins at the beginning of an hour: |9:30-----|10:00-----|10:30----- . Given a **DateTime** with a time of 10:07, this function will return 10:00 and the Date of **DateTime**, since it falls within that section. The return type is an [SCDateTime](#).

Parameters

- **DateTime**: The Date-Time value used to return the starting Date-Time of the time period that it is contained within.
- **TimePeriodType**: The type of time period. This can be any of:
 - TIME_PERIOD_LENGTH_UNIT_MINUTES
 - TIME_PERIOD_LENGTH_UNIT_DAYS
 - TIME_PERIOD_LENGTH_UNIT_WEEKS
 - TIME_PERIOD_LENGTH_UNIT_MONTHS
 - TIME_PERIOD_LENGTH_UNIT_YEARS
- **TimePeriodLength**: The number of units specified with **TimePeriodType**. For example if you want 1 Day, then you will set this to 1 and **TimePeriodType** to TIME_PERIOD_LENGTH_UNIT_DAYS.

- **PeriodOffset**: This normally should be set to 0. When it is set to -1, the function will return the prior block of time which is before the block of time that **DateTime** is within. When it is set to +1, the function will return next block of time which is after the block of time that **DateTime** is within. Any positive or negative nonzero number can be used to shift the period forward or backward. For example, +2 will get 2 periods forward from the block of time that **DateTime** is within.
- **NewPeriodAtBothSessionStarts** : When this is set to 1 and you have defined and enabled **Evening Session** times in **Chart >> Chart Settings** for an Intraday Chart, **TimePeriodType** is set to **TIME_PERIOD_LENGTH_UNIT_DAYS**, **TimePeriodLength** is set to 1, a new period will also begin at the **Session Times >> Start Time** in addition to the **Session Times >> Evening Start Time**. Otherwise, a new period only begins at the **Session Times >> Evening Start Time**.

Example

For an example, refer to the **scsf_PivotPointsVariablePeriod** function in the /ACS_Source folder in the folder that Sierra Chart is installed to.

sc.GetStudyArray()

Type: Function

```
GetStudyArray(int StudyNumber, int StudySubgraphNumber, SCFloatArrayRef SubgraphArray);
```

It is recommended that you use the [sc.GetStudyArrayUsingID](#) function. It is a newer function which replaces this function.

sc.GetStudyArray() works similar to the [sc.GetChartArray\(\)](#) function. It gets a **sc.Subgraph[]**.Data array from another study on a chart. This way you can use the data from other studies on the same chart as your Advanced Custom Study function is on. The **StudyNumber** is 1-based, where 1 refers to the first study on the chart. If you specify 0 for **StudyNumber**, then this function will refer to the main price graph of the chart your study instance is applied to. The **StudySubgraphNumber** is 1-based and refers to a Subgraph of the study, where 1 is the first subgraph.

Refer to the **scsf_GetStudyArrayExample()** function in the **studies.cpp** file inside the ACS_Source folder inside of the Sierra Chart installation folder for example code to work with this function. When calling this function, it fills in the **SubgraphArray** parameter. If the function fails to get the requested study array for any reason, **SubgraphArray** will be empty and indicated by **SubgraphArray.GetArraySize()** equaling 0.

A reason you would want to access studies on a chart rather than internally calculating them and putting the results into **sc.Subgraph[]**.Data[] or Extra Arrays, is so that you can see the studies on the chart and allow the inputs to be easily adjusted by the user. This saves memory and calculations if the studies that you are working with internally in your study are

needed to be viewed on the chart anyway.

You need to set the [Calculation Precedence](#) to **LOW_PREC_LEVEL**, in the [sc.SetDefaults](#) code block when using this function. Otherwise, you may get an array that has a size of zero and is empty. By setting the calculation precedence to LOW_PREC_LEVEL, you ensure that your custom study will get calculated after other studies on the chart. This will ensure you get an array filled with up to date values.

Example

```
if (sc.SetDefaults)
{
    sc.CalculationPrecedence = LOW_PREC_LEVEL;
    return;
}

SCFloatArray SubgraphArray;

// Get the third (3) Subgraph data array from the second (2) study
sc.GetStudyArray(2, 3, SubgraphArray);

if (SubgraphArray.GetArraySize() == 0)
{
    return; // The SubgraphArray may not exist or is empty. Either way we can not do anything with it.
}
```

sc.GetStudyArrayFromChart()

Type: Function

GetStudyArrayFromChart(int **ChartNumber**, int **StudyNumber**, int **StudySubgraphNumber**, SCFloatArrayRef **SubgraphArray**);

sc.GetStudyArrayFromChart() works identically to the [sc.GetStudyArray\(\)](#) function, except that it will also access studies on another chart.

It is recommended to use the [sc.GetStudyArrayFromChartUsingID](#) study instead as it references a study by its unique ID, rather than the **sc.GetStudyArrayFromChart()** function.

Parameters

- **ChartNumber:** The number of the chart you want to get data from. This is the number that is shown on the top line in the Chart window, after the #. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.

- **StudyNumber**: The 1-based number for the study on the chart, where 1 refers to the first study on the specified chart. If you specify 0 for **StudyNumber**, then this function will refer to the main price graph of the chart.
- **StudySubgraphNumber**: The 1-based number for the subgraph in the study, where 1 is the first Subgraph.
- **SubgraphArray**: This will be filled in with the found data.

If the function is unable to get the subgraph array, the array's size will be zero (For example, **SubgraphArray.GetArraySize() == 0**).

One reason you may want to access a study from another chart is to use the results of a study which is calculated over a different time period per bar.

Example

```
SCFloatArray SubgraphArray;

// Get the subgraph number 3 data array from the study number 2 on chart number 1
sc.GetStudyArrayFromChart(1, 2, 3, SubgraphArray);

if (SubgraphArray.GetArraySize() == 0)
    return; // The SubgraphArray may not exist or is empty. Either way we can not do anything with it.
```

sc.GetStudyArrayFromChartUsingID()

Type: Function

```
void GetStudyArrayFromChartUsingID(const s_ChartStudySubgraphValues&
ChartStudySubgraphValues, SCFloatArrayRef SubgraphArray);
```

The **sc.GetStudyArrayFromChartUsingID()** function gets a **sc.Subgraph[].Data[]** array from another study on another chart specified with the **ChartStudySubgraphValues** parameter. The array is set into the **SubgraphArray** parameter. This function works with the **sc.Input().GetChartStudySubgraphValues()** input function.

Note: if ChartStudySubgraphValues.ChartNumber is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.

Example

```

SCInputRef StudySubgraphReference = sc.Input[0];

if (sc.SetDefaults)
{
    StudySubgraphReference.Name = "Study And Subgraph To Display";

    StudySubgraphReference.SetChartStudySubgraphValues(1,1, 0);
    return;
}

SCFloatArray StudyReference;
sc.GetStudyArrayFromChartUsingID(StudySubgraphReference.GetChartStudySubgraphValues(), Stu

```

sc.GetStudyArraysFromChart()

Type: Function

GetStudyArraysFromChart(int **ChartNumber**, int **StudyNumber**, SCGraphData& **GraphData**);

sc.GetStudyArraysFromChart() is for getting all of the Subgraph arrays from a study on another chart or the same chart. A more up-to-date version of this function is the [sc.GetStudyArraysFromChartUsingID\(\)](#) function.

One reason you may want to access a study from another chart is to use the results of a study which is calculated over a different time frame per bar.

Parameters

- **ChartNumber:** The number of the chart you want to get data from. This is the number that is shown on the top line in the Chart window, after the #. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **StudyNumber:** This is 1-based, where 1 refers to the first study on the specified chart number. If you specify 0 for **StudyNumber**, then this function will refer to the main price graph of the chart.
- **GraphData:** A SCGraphData object.

Example

```

int ChartNumber = 1;

// Get the study arrays from the first study in the specified chart
SCGraphData StudyData;
sc.GetStudyArraysFromChart(ChartNumber, 1, StudyData);

// Check if we got the data and then get the first subgraph array from the study data
if(StudyData.GetArraySize() == 0)
    return;

SCFloatArrayRef StudyArray = StudyData[0];

if(StudyArray.GetArraySize() == 0)
    return;

```

sc.GetStudyArraysFromChartUsingID()

Type: Function

```
void GetStudyArraysFromChartUsingID(int ChartNumber, int StudyID, SCGraphData& GraphData);
```

The **sc.GetStudyArraysFromChartUsingID()** function gets all of the Subgraph arrays from a study specified by the **ChartNumber** and the unique **StudyID** parameters.

The [StudyID](#) is the unique study identifier.

Parameters

- **ChartNumber:** The number of the chart you want to get data from. This is the number that is shown on the top line in the Chart window, after the #. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **StudyID:** The unique identifier for the study to get data from. For more information, refer to [Unique Study Instance Identifiers](#).
- **GraphData:** A SCGraphData type object.

Example

```

// Define a graph data object to get all of the study data
SCGraphData StudyData;

// Get the study data from the specified chart
sc.GetStudyArraysFromChartUsingID(ChartStudyInput.GetChartNumber(), ChartStudyInput.GetStudyID(), StudyData);

//Check if the study has been found. If it has, GetArraySize() will return the number of Subgraphs in the study
if(StudyData.GetArraySize() == 0)
return;

// Define a reference to the first subgraph array
SCFloatArrayRef Array1 = StudyData[0];

// Check if array is not empty.
if(Array1.GetArraySize() != 0)
{
    // Get last value in array
    float LastValue = Array1[Array1.GetArraySize() - 1];
}

```

sc.GetStudyDataColorArrayFromChartUsingID()

Type: Function

```

void GetStudyDataColorArrayFromChartUsingID(int ChartNumber, int StudyID, int SubgraphIndex , SCColorArrayRef DataColorArray);

```

The **sc.GetStudyDataColorArrayFromChartUsingID** function is used to get the [sc.Subgraph\[\].DataColor](#) array at the Subgraph index specified by **SubgraphIndex** for the study specified by the **StudyID** parameter, in the chart specified by the **ChartNumber** parameter.

The [sc.Subgraph\[\].DataColor](#) array being returned, must be used by the study Subgraph for it to contain data. Otherwise, it will be empty.

Refer to the example below.

```

//Example for sc.GetStudyDataColorArrayFromChartUsingID
//SCInputRef ChartStudySubgraphInput = sc.Input[4];
//ChartStudySubgraphInput.SetChartStudySubgraphValues(1, 1, 0);

SCColorArray DataColorArray;
sc.GetStudyDataColorArrayFromChartUsingID(ChartStudySubgraphInput.GetChartNumber(), ChartStudySubgraphInput.GetStudyID(), ChartStudySubgraphInput.GetSubgraphIndex(), DataColorArray);

if (DataColorArray.GetArraySize() > 0)//size is nonzero meaning that we have gotten a valid array
{
    //Use the array for something
}

```

sc.GetStudyArrayUsingID()

Type: Function

```
int GetStudyArrayUsingID(unsigned int StudyID, unsigned int StudySubgraphIndex,  
SCFloatArrayRef SubgraphArray);
```

sc.GetStudyArrayUsingID() gets the specified `sc.Subgraph[]`.Data array from a study using the study's unique identifier. These are studies that are on the same chart that your custom study is applied to.

If the study you are referencing is moved up or down in the list of **Studies to Graph** in the **Chart Studies** window for the chart, the reference still stays valid.

This function works with the [sc.Input](#) functions **sc.Input[]**.SetStudyID() or **sc.Input[]**.SetStudySubgraphValues and the related functions for getting the study ID and the study Subgraph values.

StudySubgraphIndex parameter: This is a 0 based index indicating which Subgraph to get from the study specified by **StudyID**. These indexes directly correspond to the Subgraphs listed in the [Subgraphs tab](#) of the Study Settings window for the study. Although this is zero-based and the Subgraphs are 1-based as they are listed. For example, Subgraph 1 (**SG1**) in the Study Settings window will have a **StudySubgraphIndex** of 0.

Set the [Calculation Precedence](#) to **LOW_PREC_LEVEL**, in the [sc.SetDefaults](#) code block when using this function.

Otherwise, you may get an array that has a size of zero and is empty. By setting the calculation precedence to **LOW_PREC_LEVEL**, you ensure that the custom study will get calculated after other studies on the chart. This will ensure the study instance receives an array filled with up to date values.

When using this function or any of the ACSIL functions for getting a study array, it is possible to get the **sc.Subgraph.Data[]** arrays of a **Spreadsheet** study on the chart. This will allow you to get any of the formula columns data (K-Z and more depending upon the setting of the **Number of Formula Columns** input). Additionally, by using the appropriate formulas in the Spreadsheet formula columns, you are able to reference other data on the Spreadsheet by having those formula column formulas reference the particular cells that you need.

Return Value: Returns 1 on success or 0 if the StudyID is not found or the StudySubgraphIndex parameter is outside the valid range of subgraph indexes.

For an example, refer to the **scsf_ReferenceStudyData** function in the `/ACS_Source/studies8.cpp` file or to the **scsf_StudySubgraphsDifference** function in the `/ACS_Source/studies5.cpp` file which are located in the Sierra Chart installation folder.

Example

```

if (sc.SetDefaults)
{
    sc.CalculationPrecedence = LOW_PREC_LEVEL;

    sc.Input[1].Name = "Study Reference";
    sc.Input[1].SetStudyID(1);
}

SCFloatArray StudyReference;

//Get the first (0) subgraph from the study the user has selected.

if (sc.GetStudyArrayUsingID(sc.Input[1].GetStudyID(), 0, StudyReference) > 0
    && StudyReference.GetArraySize() > 0)
{
    //Copy the study data that we retrieved using GetStudyArrayUsingID, into a subgraph data output a
    sc.Subgraph[0][sc.Index] = StudyReference[sc.Index];
}

```

sc.GetStudyDataStartIndexFromChartUsingID()

Type: Function

```
int sc.GetStudyDataStartIndexFromChartUsingID(int ChartNumber, unsigned int StudyID);
```

The **sc.GetStudyDataStartIndexFromChartUsingID()** function returns the [sc.DataStartIndex](#) from the study specified by the **StudyID** parameter and the chart specified by the **ChartNumber** parameter.

Parameters

- **ChartNumber**: The Chart Number of the chart containing the study to get the **sc.DataStartIndex** for. This can be obtained through the [sc.Input\[\].SetChartStudyValues](#) Input and the related **sc.Input[].Get*** functions.
- **StudyID**: The unique ID for the study. For more information, refer to [Unique Study Instance Identifiers](#).

sc.GetStudyDataStartIndexUsingID()

Type: Function

```
int GetStudyDataStartIndexUsingID(unsigned int StudyID);
```

The **sc.GetStudyDataStartIndexUsingID()** function gets the **sc.DataStartIndex** value from another study on the same chart that your custom study is applied to, using the study's unique ID. This is for specialised purposes.

Example

```
SCInputRef StudyReference = sc.Input[0];  
sc.DataStartIndex = sc.GetStudyDataStartIndexUsingID(StudyReference.GetStudyID());
```

sc.GetStudyExtraArrayFromChartUsingID()

Type: Function

```
int GetStudyExtraArrayFromChartUsingID(int ChartNumber, int StudyID, int  
SubgraphIndex, int ExtraArrayIndex, SCFloatArrayRef ExtraArrayRef);
```

The **sc.GetStudyExtraArrayFromChartUsingID()** function is used for getting one of the [sc.Subgraph\[\].Arrays\[\]\[\]](#) on a **sc.Subgraph** from another study. This study can also be on another chart.

Example

```
SCFloatArray ExtraArrayFromChart ;  
sc.GetStudyExtraArrayFromChartUsingID(1, 1, 0, 0, ExtraArrayFromChart);  
if (ExtraArrayFromChart.GetArraySize()>0)  
{  
  
}
```

sc.GetStudyIDByIndex()

Type: Function

```
int GetStudyIDByIndex(int ChartNumber, int StudyNumber);
```

The **sc.GetStudyIDByIndex()** function returns the unique **sc.StudyGraphInstanceID** for the study specified by the **ChartNumber** and the one based **StudyNumber** parameters. Chart identifying numbers are shown on the top line of the chart after the #. The **StudyNumber** is the one based index of the study in the **Studies to Graph** list in the [Chart Studies](#) window.

Setting **StudyNumber** to 0 will refer to the underlying main price graph of the chart.

sc.GetStudyIDByName()

Type: Function

```
int sc.GetStudyIDByName(int ChartNumber, const char* Name, const int  
UseShortNameIfSet);
```

The **sc.GetStudyIDByName()** function returns the study identifier of the study identified by

the **Name** parameter.

Parameters

- **ChartNumber:**
- **Name:.**
- **UseShortNameIfSet:.**

sc.GetStudyInternalIdentifier()

Type: Function

```
uint32_t sc.GetStudyInternalIdentifier(int ChartNumber, int StudyID, SCString& r_StudyName);
```

The **sc.GetStudyInternalIdentifier()** function .

Parameters

- **ChartNumber:**
- **StudyID:.**
- **r_StudyName:.**

sc.GetStudyLineUntilFutureIntersection()

Type: Function

```
int GetStudyLineUntilFutureIntersection(int ChartNumber, int StudyID, int BarIndex, int LineIndex, int& LineIDForBar, float &LineValue, int &ExtensionLineChartColumnEndIndex));
```

The **sc.GetStudyLineUntilFutureIntersection** function is used to get the details of a line until future intersection from a study which has added one of these lines using the function [sc.AddLineUntilFutureIntersection\(\)](#).

Return Value: If the line is found, 1 is returned. If the line is not found, 0 is returned.

Parameters

- **ChartNumber:** The chart number containing the study to get the future intersection line from. Normally this will be set to [sc.ChartNumber](#).
- **StudyID:** The unique ID for the study to get the future intersection line from. For more information, refer to [Unique Study Instance Identifiers](#).
- **BarIndex:** The bar index where the future intersection line begins.
- **LineIndex:** The line index for the future intersection line. This is a zero-based index. The first future intersection line at a chart bar will have an index of 0.

- **LineIDForBar**: This parameter is a reference. This is the identifier of the extension line for a chart bar.
- **LineValue**: This parameter is a reference and is set to the vertical axis level at which the line is drawn at.
- **ExtensionLineChartColumnEndIndex**: This parameter is a reference. This is set to the bar index the line ends at. A value of zero means that the line has not yet intersected a future price bar.

sc.scGetNumLinesUntilFutureIntersection()

Type: Function

```
int GetStudyLineUntilFutureIntersection(int ChartNumber, int StudyID, int Index);
```

The **sc.GetNumLinesUntilFutureIntersection** function is used to get the number of lines which have been added to a particular study through the [sc.AddLineUntilFutureIntersection](#) function.

Return Value: The number lines. 0 if there are no lines.

Parameters

- **ChartNumber**: The chart number containing the study to get the number of lines from. Normally this will be set to [sc.ChartNumber](#).
- **StudyID**: The unique ID for the study to get the number of lines from. For more information, refer to [Unique Study Instance Identifiers](#).

sc.GetStudyLineUntilFutureIntersectionByIndex()

Type: Function

```
int GetStudyLineUntilFutureIntersectionByIndex(int ChartNumber, int StudyID, int Index, int& r_LineIDForBar, int& r_StartIndex, float& r_LineValue, int& r_ExtensionLineChartColumnEndIndex);
```

The **sc.GetStudyLineUntilFutureIntersectionByIndex** function is used to get the details of a line until future intersection from a study which has added one of these lines using the function [sc.AddLineUntilFutureIntersection](#).

This function specifies the future intersection line by its zero-based index (**Index**) in the container where the lines are stored for the study. The upper bound of this parameter will be the value returned by [sc.GetNumLinesUntilFutureIntersection](#) -1.

Return Value: If the line is found, 1 is returned. If the line is not found, 0 is returned.

Parameters

- **ChartNumber**: The chart number containing the study to get the future

intersection line from. Normally this will be set to [sc.ChartNumber](#).

- **StudyID**: The unique ID for the study to get the future intersection line from. For more information, refer to [Unique Study Instance Identifiers](#).
- **Index**: This is the zero-based index for the future intersection line details to return. If this index is not valid, the function returns 0.
- **r_LineIDForBar**: This parameter is a reference and is an output value. Upon return it is set to the identifier of the extension line for a chart bar.
- **r_StartIndex**: This parameter is a reference and is an output value. Upon return it is set to the starting bar index of the extension line.
- **r_LineValue**: This parameter is a reference and is an output value. Upon return it is set to the vertical axis value at which the line is drawn at.
- **r_ExtensionLineChartColumnEndIndex**: This parameter is a reference and is an output value. Upon return it is set to the chart bar index the line ends at. A value of zero means that the line has not yet intersected a future price bar.

sc.GetStudyName()

Type: Function

```
SCString GetStudyName(int StudyIndex);
```

sc.GetStudyName() returns the name of the study at the index **StudyIndex**. **StudyIndex** is not the unique study identifier. It is the one based index based on the order of the studies in the **Studies to Graph** list in the Chart Studies window for the chart.

To get the name of the main price/base graph for the chart use a **StudyIndex** of 0.

sc.GetStudyNameFromChart()

Type: Function

```
SCString GetStudyNameFromChart(int ChartNumber, int StudyID);
```

The **sc.GetStudyNameFromChart()** function gets the name of the study identified by **StudyID** from the chart specified by **ChartNumber**. **StudyID** can be 0 to get the name of the underlying main graph of the chart.

Example

```
SCInputRef ChartStudySubgraphReference = sc.Input[4];

int ChartNumber = ChartStudySubgraphReference.GetChartNumber();
int StudyID = ChartStudySubgraphReference.GetStudyID();

SCString StudyName = sc.GetStudyNameFromChart(ChartNumber, StudyID);
```

sc.GetStudyNameUsingID()

Type: Function

```
SCString GetStudyNameUsingID(unsigned int StudyID);
```

The **sc.GetStudyNameUsingID()** function gets the name of a study on the same chart that your custom study is applied to, using the study's unique ID.

Example

```
SCInputRef StudyReference = sc.Input[3];  
  
SCString StudyName = sc.GetStudyNameUsingID(StudyReference.GetStudyID());  
  
sc.GraphName.Format("Avg of %s", StudyName);
```

sc.GetStudyPeakValleyLine()

Type: Function

```
int GetStudyPeakValleyLine(int ChartNumber, int StudyID, float& PeakValleyLinePrice,  
int& PeakValleyType, int& StartIndex, int& PeakValleyExtensionChartColumnEndIndex,  
int ProfileIndex , int PeakValleyIndex);
```

The **sc.GetStudyPeakValleyLine()** function is used for obtaining the details about a Peak or Valley line from a study if the study supports Peak and Valley lines.

Studies that support these Peak and Valley lines are the **TPO Profile Chart** and the **Volume by Price** studies. Peak and Valley lines cannot be obtained for a **Volume by Price** study which uses **Visible Bars** for the **Volume Graph Period Type** Input.

Refer to the parameter list below to understand how to use this function.

This function returns 1 if a Peak or Valley line was found. It returns 0 if one was not found.

This function is only properly supported in version 1844 and higher.

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

Parameters

- **ChartNumber:** The number of the chart containing the study to get the Peak or Valley line details from. Normally this will be set to **sc.ChartNumber**.
- **StudyID:** The unique ID for the study to get the Peak or Valley line from. For more information, refer to [Unique Study Instance Identifiers](#).
- **PeakValleyLinePrice:** This parameter is a reference. On return, this is

the price of the Peak or Valley line that has been found according to the **ProfileIndex** and **PeakValleyIndex** parameters. It will be 0 if the line was not found.

- **PeakValleyType**: This parameter is a reference. On return, this is the type of the Peak or Valley line that has been found according to the **ProfileIndex** and **PeakValleyIndex** parameters. It will be 0 if the line was not found. It will be set to one of the following:
PEAKVALLEYTYPE_NONE = 0, PEAKVALLEYTYPE_PEAK = 1,
PEAKVALLEYTYPE_VALLEY = 2.
- **StartIndex**: This parameter is a reference. On return, this is the start chart bar array index of the Peak or Valley line that has been found according to the **ProfileIndex** and **PeakValleyIndex** parameters. It will be 0 if the line was not found.
- **PeakValleyExtensionChartColumnEndIndex**: This parameter is a reference. On return, this is the ending chart bar array index of the Peak or Valley line that has been found according to the **ProfileIndex** and **PeakValleyIndex** parameters. It will be 0 if the line was not found. If this line extends to the very end of the chart, this variable will be set to 0. This will also be 0 in the case where the Peak or Valley line is not set to extend until future intersection.
- **ProfileIndex** : With the **Volume by Price** and **TPO Profile Chart** studies, they consist of multiple profiles. **ProfileIndex** specifies the zero-based index of the profile to obtain the Peak or Valley line from. Effective with version 1843, this can be a negative number. In this case -1 will reference the last profile in the chart. -2 references the second to last profile in the chart and so on.
- **PeakValleyIndex**: Each profile in the study as specified by the **ProfileIndex** parameter may contain Peak or Valley lines. **PeakValleyIndex** is the zero-based index of the Peak or Valley line with the profile to return. Start setting this to 0 and increment it until there are no longer any Peak or Valley lines obtained and then you know you have obtained them all from a particular profile in the study.

sc.GetStudyProfileInformation

Type:Function

```
int GetStudyProfileInformation(const int StudyID, const int ProfileIndex,  
n_ACSIL::s_StudyProfileInformation& StudyProfileInformation);
```

This function when called fills out a **StudyProfileInformation** structure of type **n_ACSIL::s_StudyProfileInformation** with all of the numerical details of a Volume Profile or TPO profile when using the [Volume by Price](#) study or the [TPO Profile Chart](#) study, respectively.

The members of the **n_ACSIL::s_StudyProfileInformation** structure are listed further below.

Parameters

- **StudyID**: The unique study identifier for the **Volume by Price** or **TPO Profile Chart** study. Refer to [UniqueStudyInstanceIdentifiers](#).
- **ProfileIndex**: The zero-based index of the Volume or TPO profile relative to the end of the chart. A profile is a group of volume bars (in the case of the Volume by Price study) or letters/blocks (in the case of a TPO Profile Chart) covering the time period that the study specifies per profile (for example 1 Day). 0 equals the latest profile in the chart at the end or rightmost side. 1 equals the second to last profile in the chart. This needs to always be set to a positive number.
- **StudyProfileInformation**: This is a reference to a structure of type **n_ACSIL::s_StudyProfileInformation** which receives the information for the volume or TPO profile specified by **StudyID** and **ProfileIndex**.

n_ACSIL::s_StudyProfileInformation Members

- m_StartDateTime
- m_NumberOfTrades
- m_Volume
- m_BidVolume
- m_AskVolume
- m_TotalTPOCount
- m_OpenPrice
- m_HighestPrice
- m_LowestPrice
- m_LastPrice
- m_TPOMidpointPrice
- m_TPOMean
- m_TPOStdDev
- m_TPOErrorOfMean
- m_TPOPOCPrice
- m_TPOValueAreaHigh
- m_TPOValueAreaLow
- m_TPOCountAbovePOC
- m_TPOCountBelowPOC
- m_VolumeMidpointPrice
- m_VolumePOCPrice
- m_VolumeValueAreaHigh
- m_VolumeValueAreaLow
- m_VolumeAbovePOC
- m_VolumeBelowPOC
- m_POCAboveBelowVolumeImbalancePercent
- m_VolumeAboveLastPrice
- m_VolumeBelowLastPrice

- m_BidVolumeAbovePOC
- m_BidVolumeBelowPOC
- m_AskVolumeAbovePOC
- m_AskVolumeBelowPOC
- m_VolumeTimesPriceInTicks
- m_TradesTimesPriceInTicks
- m_TradesTimesPriceSquaredInTicks
- m_IBRHighPrice
- m_IBRLowPrice
- m_OpeningRangeHighPrice
- m_OpeningRangeLowPrice
- m_VolumeWeightedAveragePrice
- m_MaxTPOBlocksCount
- m_TPOCountMaxDigits
- m_DisplayIndependentColumns
- m_EveningSession
- m_AverageSubPeriodRange
- m_RotationFactor
- m_VolumeAboveTPOPOC
- m_VolumeBelowTPOPOC
- m_EndDateTime
- m_BeginIndex
- m_EndIndex

sc.GetStudyStorageBlockFromChart

Type:Function

```
void* GetStudyStorageBlockFromChart(int ChartNumber, int StudyID);
```

This function returns a pointer to the [Storage Block](#) for the specified ChartNumber and StudyID.

For information about Study IDs, refer to [Unique Study Instance Identifiers](#).

sc.GetStudySubgraphColors()

Type: Function

```
int32_t GetStudySubgraphColors(int32_t ChartNumber, int32_t StudyID, int32_t StudySubgraphNumber, uint32_t& r_PrimaryColor, uint32_t& r_SecondaryColor, uint32_t& r_SecondaryColorUsed);
```

The **sc.GetStudySubgraphColors()** function gets the primary and secondary colors of a Subgraph in another study in the Chartbook. The study can be in a different chart.

The **ChartNumber**, **StudyID**, and **StudySubgraphNumber** parameters should be obtained using the [sc.Input\[\].GetChartStudySubgraphValues](#) function.

The colors from the other study are set into the **r_PrimaryColor** and **r_SecondaryColor** parameters which are references.

The function returns 1 if the study was found. Otherwise, 0 is returned.

Example

```
uint32_t PrimaryColor = 0;
uint32_t SecondaryColor = 0;
uint32_t SecondaryColorUsed = 0;
sc.GetStudySubgraphColors(1, 1, 0, PrimaryColor, SecondaryColor, SecondaryColorUsed);
```

sc.GetStudySubgraphDrawStyle()

Type: Function

```
int GetStudySubgraphDrawStyle(int ChartNumber, int StudyID, int
StudySubgraphNumber, int& r_DrawStyle);
```

The **sc.GetStudySubgraphDrawStyle()** function gets the Draw Style of a Subgraph in another study in the Chartbook. The study can be in a different chart.

The **ChartNumber**, **StudyID**, and **StudySubgraphNumber** parameters should be obtained using the [sc.Input\(\).GetChartStudySubgraphValues](#) function.

The Draw Style from the other study is set into the **r_DrawStyle** parameter which is a reference.

The function returns 1 if the study was found, otherwise 0 is returned.

Example

```
int DrawStyle = 0;
sc.GetStudySubgraphDrawStyle(1, 1, 0, DrawStyle);
```

sc.GetStudySubgraphLineStyle()

Type: Function

```
int32_t GetStudySubgraphLineStyle(int32_t ChartNumber, int32_t StudyID, int32_t
StudySubgraphNumber, SubgraphLineStyles& r_LineStyle);
```

The **sc.GetStudySubgraphLineStyle()** function .

Parameters

- **ChartNumber:** .

- **StudyID:** .
- **StudySubgraphNumber:** .
- **r_LineStyle:** .

Example



sc.GetStudySubgraphLineWidth()

Type: Function

```
int32_t GetStudySubgraphLineWidth(int32_t ChartNumber, int32_t StudyID, int32_t StudySubgraphNumber, int32_t& r_LineWidth);
```

The **sc.GetStudySubgraphLineWidth()** function .

Parameters

- **ChartNumber:** .
- **StudyID:** .
- **StudySubgraphNumber:** .
- **r_LineWidth:** .

Example



sc.GetStudySubgraphName

Type: Function

```
SCString GetStudySubgraphName(int StudyID, int SubgraphIndex);
```

The **sc.GetStudySubgraphName()** function gets the name of a Subgraph for a study on the same chart that the study instance is applied to. The **StudyID** parameter specifies the study ID which is the studies unique identifier. The **SubgraphIndex** parameter specifies the zero-based Subgraph index for the Subgraph.

The general method to easily select and determine a StudyID and SubgraphIndex is through the related study Input functions. Refer to [sc.Input\[\].SetStudySubgraphValues\(\)](#).

sc.GetStudySubgraphNameFromChart()

Type: Function

```
int32_t GetStudySubgraphNameFromChart(int ChartNumber, int StudyID, int SubgraphIndex, SCString& r_SubgraphName);
```

The **sc.GetStudySubgraphNameFromChart()** function gets the name of a Subgraph for a study on the same or a different chart as the study instance calling this function is applied to.

The **ChartNumber** parameter specifies the chart number. The **StudyID** parameter specifies the study ID which is the studies unique identifier. The **SubgraphIndex** parameter specifies the zero-based Subgraph index for the Subgraph.

There are various Inputs available to support obtaining the **ChartNumber**, **StudyID**, **SubgraphIndex** parameters through a Study Input in the user interface. Refer to [sc.Input\[\]](#).

sc.GetStudySummaryCellAsDouble()

Type: Function

```
int GetStudySummaryCellAsDouble(const int Column, const int Row, double& r_CellValue);
```

The **sc.GetStudySummaryCellAsDouble** function is used to obtain the data from a cell of the [Study Summary](#) window as a numeric double type. The cell is specified by the row and column.

Parameters

- **Column:** This is a zero-based integer specifying the column index of the cell to get the numeric double from.
- **Row:** This is a zero-based integer specifying the row index of the cell to get the numeric double from.
- **r_CellValue:** This is a reference to a double variable that receives the value from the Study Summary window.

Example

```
double StudySummaryCellValue;  
sc.GetStudySummaryCellAsDouble(5, 3, StudySummaryCellValue);
```

sc.GetStudySummaryCellAsString()

Type: Function

```
int GetStudySummaryCellAsString(const int Column, const int Row, SCString& r_CellString);
```

The **sc.GetStudySummaryCellAsString** function is used to obtain the data from a cell of the [Study Summary](#) window as a text string. The cell is specified by the row and column.

Parameters

- **Column**: This is a zero-based integer specifying the column index of the cell to get the text string from.
- **Row**: This is a zero-based integer specifying the row index of the cell to get the text string from.
- **r_CellString**: This is a reference to a SCString that receives the text string from the Study Summary window.

Example

```
SCString StudySummaryCellText;  
sc.GetStudySummaryCellAsString(5, 3, StudySummaryCellText);
```

sc.GetStudyVisibilityState()

Type: Function

```
int GetStudyVisibilityState(int StudyID);
```

The **sc.GetStudyVisibilityState()** function returns 1 if the study specified by the **StudyID** parameter is visible, or 0 if it is set to be hidden.

Parameters

- **StudyID**: The unique ID of the study to get the visibility state of. For more information, refer to [Unique Study Instance Identifiers](#).

sc.GetSummation()

Type: Intermediate Study Calculation Function

```
float GetSummation(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetSummation(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetSummation()** function calculates the sum of the values in **FloatArrayIn** over the specified **Length**. The result is returned as a single float value.

Parameters

- [FloatArrayIn](#).

- [Length](#).
- [Index](#).

Example

```
float Summation = sc.GetSummation(sc.BaseDataIn[SC_LAST],sc.Index,10);
```

sc.GetSymbolDataValue()

Type: Function

```
GetSymbolDataValue(SymbolDataValuesEnum ValueToReturn, const SCString&  
SymbolForData = SCString(), int SubscribeToMarketData = false, int  
SubscribeToMarketDepth = false);
```

The **sc.GetSymbolDataValue** function is used to obtain a particular data value for the specified symbol and also subscribe to market data. The data value could be for example the last trade price or the bid or ask prices. Refer to the list below.

This function also can be used during a [chart replay](#) but during a chart replay, when **SymbolForData** is an empty string, not all fields of data will have valid values. When **SymbolForData** is set to a particular symbol, the fields of data always come from the connected streaming data feed and the current values are provided. In this last case they do not come from a replaying chart.

Parameters

- **ValueToReturn:** Refer to the SymbolDataValuesEnum list below.
- **SymbolForData:** This is an optional symbol to return the data value for. If it is not set, it will be the symbol of the chart the study is applied to.
- **SubscribeToMarketData:** Set this to 1 to subscribe to streaming market data for the symbol of the chart or the specified symbol.
- **SubscribeToMarketDepth:** Set this to 1 to subscribe to streaming market depth data for the symbol of the chart or the specified symbol.

The following are the possible constant values to use for **ValueToReturn**. They are also listed in **/ACS_Source/scconstants.h** file. They are Of enumeration type SymbolDataValuesEnum.

- SYMBOL_DATA_UNSET = 0
- SYMBOL_DATA_DAILY_OPEN = 1
- SYMBOL_DATA_DAILY_HIGH = 2
- SYMBOL_DATA_DAILY_LOW = 3
- SYMBOL_DATA_DAILY_NUMBER_OF_TRADES = 4
- SYMBOL_DATA_LAST_TRADE_PRICE = 5

- SYMBOL_DATA_LAST_TRADE_VOLUME = 6
- SYMBOL_DATA_ASK_QUANTITY = 7
- SYMBOL_DATA_BID_QUANTITY = 8
- SYMBOL_DATA_BID_PRICE = 9
- SYMBOL_DATA_ASK_PRICE = 10
- SYMBOL_DATA_CURRENCY_VALUE_PER_TICK = 11
- SYMBOL_DATA_SETTLEMENT_PRICE = 12
- SYMBOL_DATA_OPEN_INTEREST = 13
- SYMBOL_DATA_DAILY_VOLUME = 14
- SYMBOL_DATA_SHARES_OUTSTANDING = 15
- SYMBOL_DATA_EARNINGS_PER_SHARE = 16
- SYMBOL_DATA_TICK_DIRECTION = 17
- SYMBOL_DATA_LAST_TRADE_AT_SAME_PRICE = 18
- SYMBOL_DATA_STRIKE_PRICE = 19
- SYMBOL_DATA_SELL_ROLLOVER_INTEREST = 20
- SYMBOL_DATA_PRICE_FORMAT = 21
- SYMBOL_DATA_BUY_ROLLOVER_INTEREST = 22
- SYMBOL_DATA_TRADE_INDICATOR = 23
- SYMBOL_DATA_LAST_TRADE_AT_BID_ASK = 24
- SYMBOL_DATA_VOLUME_VALUE_FORMAT = 25
- SYMBOL_DATA_TICK_SIZE = 26
- SYMBOL_DATA_LAST_TRADE_DATE_TIME = 27
- SYMBOL_DATA_ACCUMULATED_LAST_TRADE_VOLUME = 28
- SYMBOL_DATA_LAST_TRADING_DATE_FOR_FUTURES = 29
- SYMBOL_DATA_TRADING_DAY_DATE = 30
- SYMBOL_DATA_LAST_MARKET_DEPTH_UPDATE_DATE_TIME = 31
- SYMBOL_DATA_DISPLAY_PRICE_MULTIPLIER = 32
- SYMBOL_DATA_SETTLEMENT_PRICE_DATE = 33
- SYMBOL_DATA_DAILY_OPEN_PRICE_DATE = 34
- SYMBOL_DATA_DAILY_HIGH_PRICE_DATE = 35
- SYMBOL_DATA_DAILY_LOW_PRICE_DATE = 36
- SYMBOL_DATA_DAILY_VOLUME_DATE = 37
- SYMBOL_DATA_NUMBER_OF_TRADES_AT_CURRENT_PRICE = 38

sc.GetSymbolDescription()

Type: Function

```
int GetSymbolDescription(SCString& r_Description);
```

The **sc.GetSymbolDescription** function sets the r_Description with the full text description, if available, for the symbol of the chart the study is on.

The function returns 1 if successful. Otherwise, 0 is returned.

Parameters

- **r_Description:** This is a SCString parameter which receives the full text description if available.

sc.GetTimeAndSales()

Type: Function

GetTimeAndSales(c_SCTimeAndSalesArray& TSArray);

sc.GetTimeAndSales() gets the Time and Sales data for the symbol of the chart that the study is on. This function returns an array of **s_TimeAndSales** structure.

This data also includes Bid and Ask price and size/quantity updates as well.

Refer to any of the following functions in the folder Sierra Chart is installed to for an example to work with the **sc.GetTimeAndSales** function:

- **TimeAndSales()** in the **/ACS_Source/studies.cpp** file.
- [scsf_TimeAndSalesPrice\(\)](#) in the **/ACS_Source/studies.cpp** file.
- [scsf_TimeAndSalesVolume\(\)](#) in the **/ACS_Source/studies.cpp** file.
- [scsf_TimeAndSalesTime\(\)](#) in the **/ACS_Source/studies.cpp** file.
- **scsf_TimeAndSalesIterationExample** in the **/ACS_Source/studies5.cpp** file.

For the possible values that the **Type** member of the **s_TimeAndSales** structure, refer to the list below:

1. **SC_TS_MARKER:** This indicates a gap in the time and sales data.
2. **SC_TS_BID:** This is a trade and it is considered to have occurred at Bid price or lower.
3. **SC_TS_ASK:** This is a trade and it is considered to have occurred at Ask price or higher.
4. **SC_TS_BIDASKVALUES:** This is a Bid and Ask quote data update only.

For the possible values that the **UnbundledTradeIndicator** member of the **s_TimeAndSales** structure, refer to the list below. These values only apply when using the **Sierra Chart Exchange Data Feed** and only for CME symbols.

1. **UNBUNDLED_TRADE_NONE:** The trade is not part of a bundled trade.
2. **FIRST_SUB_TRADE_OF_UNBUNDLED_TRADE:** This is the first trade in a larger trade consisting of multiple sub trades.
3. **LAST_SUB_TRADE_OF_UNBUNDLED_TRADE:** This is the last trade in a larger trade consisting of multiple sub trades.

For the complete definition of the **s_TimeAndSales** structure members, refer to the **/ACS_Source/sierrachart.h** file in the Sierra Chart installation folder.

The **s_TimeAndSales::Price**, **s_TimeAndSales::Bid**, **s_TimeAndSales::Ask** price related

members in some cases need to be multiplied by the [sc.RealTimePriceMultiplier](#) multiplier variable. As a general rule, it is best practice to always apply the multiplier in case it is needed.

This function also gets all Bid, Ask, Bid Size, and Ask Size data received from the data feed.

When iterating through the Time and Sales data, to only process those records which are trades, you need to compare the `s_TimeAndSales::Type` member to `SC_TS_BID` and `SC_TS_ASK` and only process records with these types.

For Time and Sales related settings, select

Global Settings >> Data/Trade Service Settings on the menu. Refer to [Number of Stored Time and Sales Records](#). This setting controls the maximum Time and Sales records you will receive when calling **sc.GetTimeAndSales**.

After changing this setting, you need to reconnect to the data feed using the Disconnect and Connect commands on the **File** menu. You will only be able to get Time and Sales data when there is active trading activity for a symbol. It is only maintained during a Sierra Chart session. When you restart Sierra Chart, it is lost.

The [Combine Records](#) options in the Time and Sales window settings for the chart, has no effect on the Time and Sales records obtained with the **sc.GetTimeAndSales** function.

Every Time and Sales record has a unique sequence number (`s_TimeAndSales::Sequence` member) internally calculated by a Sierra Chart instance. This number is not reset when reconnecting to the data feed. The only time it is reset back to 1, is when Sierra Chart is restarted. In that case, Time and Sales data is lost. This unique sequence number is used to determine what Time and Sales records the study function has already processed and what records it needs to process.

In the case of a [Chart Replay](#) the record sequence number is reset back to 1 when the replay is started and any other time the chart is reloaded which can occur if the chart is scrolled back in time and the replay is started again.

If the last record sequence number available during the prior call to your function was 100, then only records after that number will your study function need to process. You can store the last sequence number processed by using the [sc.GetPersistentInt\(\)](#) function, in order to know it on the next call into your function.

For a particular symbol, the Time and Sales sequence numbers are not necessarily consecutive. There can be skipped sequence numbers. However, the sequence number will always ascend.

If you require your study function to be aware of every new tick/trade as it occurs, then you will want to use the **sc.GetTimeAndSales** function to get all of the trades that occurred between calls to your study function. Your study function is not called at every trade, rather instead at the **Chart Update Interval**. The **Chart Update Interval** is set through **Global Settings >> General Settings**. You might want to reduce that setting in this particular case.

The `DateTime` member of the **s_TimeAndSales** data structure is in UTC time. The below code example shows how to adjust it to the time zone in Sierra Chart.

When the real-time market data subscribed to for a symbol from the data feed, the time and sales data is stored from that point in time using data from the streaming data feed. There is no historical time and sales data. However, the existing time and sales data is remembered between data feed connections. It is only cleared on a restart of Sierra Chart. There is a limit to the number of records stored. Refer to [Number of Stored Time and Sales Records](#).

Example

```
// Get the Time and Sales
c_SCTimeAndSalesArray TimeSales;
sc.GetTimeAndSales(TimeSales);

if (TimeSales.Size() == 0)
    return; // No Time and Sales data available for the symbol

// Loop through the Time and Sales
int OutputArrayIndex = sc.ArraySize;
for (int TSIndex = TimeSales.Size() - 1; TSIndex >= 0; --TSIndex)
{
    //Adjust timestamps to Sierra Chart TimeZone
    SCDateTime AdjustedDateTime = TimeSales[TSIndex].DateTime;
    AdjustedDateTime += sc.TimeScaleAdjustment;
}
```

Alternative Way For Obtaining Time and Sales Data

Open a chart that is set to 1 **Number of Trades Per Bar** for the symbol you want Time and Sales for.

Select **Global Settings >> Data/Trade Service Settings** and make sure the **Intraday Data Storage Time Unit** is set to 1 Tick.

Use the [sc.GetChartArray](#) and the [sc.GetChartDateTimeArray](#) functions to access the price, volume and `DateTime` arrays. Every element in these arrays is 1 trade. This may actually be a preferred way of accessing trade by trade data since there will be an abundant amount of history available.

Total Bid and Ask Depth Data

The **s_TimeAndSales** data structure also contains the **s_TimeAndSales::TotalBidDepth** and the **s_TimeAndSales::TotalAskDepth** members.

These members are only set when **s_TimeAndSales::Type** is set to the constant **SC_TS_BIDASKVALUES** which means that the Time and Sales record contains a Bid and Ask quote update.

The **s_TimeAndSales::TotalBidDepth** and **s_TimeAndSales::TotalAskDepth** members are set to the total of the Bid Sizes/Quantities and Ask Sizes/Quantities, respectively for all of the market depth levels available for the symbol.

If there are no market depth features being used in Sierra Chart or the ACSIL function has not set **sc.UsesMarketDepthData** to a nonzero value, then market depth data usually is not being received and maintained for the symbol. Therefore, these members will equal the best Bid Size and Ask Size, respectively at each record with a **s_TimeAndSales::Type** set to **SC_TS_BIDASKVALUES**.

sc.GetTimeAndSalesForSymbol()

Type: Function

```
int GetTimeAndSalesForSymbol(const SCString& Symbol, c_SCTimeAndSalesArray& TSArry);
```

The **sc.GetTimeAndSalesForSymbol** function is identical to [sc.GetTimeAndSales](#) except that it has a **Symbol** parameter allowing the study to get Time and Sales data for a different symbol.

Refer to the documentation for [sc.GetTimeAndSales](#) for the documentation for this function.

The **sc.GetTimeAndSalesForSymbol** function is only able to get the current Time and Sales data received from that real-time data feed, and not Time and Sales data for a different symbol during a chart replay. This function will also not return the Time and Sales data based on a chart replay for the same symbol as the chart which contains the custom study this function is called from.

sc.GetTimeSalesArrayIndexesForBarIndex()

Type: Function

```
void GetTimeSalesArrayIndexesForBarIndex(int BarIndex, int& r_BeginIndex, int& r_EndIndex);
```

The **sc.GetTimeSalesArrayIndexesForBarIndex** function receives a **BarIndex** parameter specifying a particular chart bar index. It gets the Date-Time of that chart bar and returns the beginning and ending indexes into the **c_SCTimeAndSalesArray** array set by the [sc.GetTimeAndSales\(\)](#) function. The beginning and ending time and sales array indexes are set through the **r_BeginIndex** and **r_EndIndex** integers passed by reference to the **sc.GetTimeSalesArrayIndexesForBarIndex** function.

r_BeginIndex and **r_EndIndex** will be set to -1 if there is no element in the time and sales array contained within the bar specified by **BarIndex**.

```
int BeginIndex = 0, EndIndex = 0;
sc.GetTimeSalesArrayIndexesForBarIndex(sc.UpdateStartIndex, BeginIndex, EndIndex);
SCString DebugString;
DebugString.Format("Time and Sales BeginIndex = %d, EndIndex = %d", BeginIndex, EndIndex);
sc.AddMessageToLog(DebugString, 0);
```

sc.GetTotalNetProfitLossForAllSymbols()

Type: Function

double **GetTotalNetProfitLossForAllSymbols**(int **DailyValues**);

The **sc.GetTotalNetProfitLossForAllSymbols** function returns the total Closed Trades Profit/Loss and Open Trades Profit/Loss (net profit/loss) for the unique symbols for open charts which are maintaining a Trades list and match the Trade Account of the chart that the study function is called from.

The Profit/loss value can be controlled to give the results for just the current trading day, or for all days for which there is order fill data. This is controlled through the **DailyValues** variable.

The amount is a Currency Value and it is also converted to the [Common Profit/Loss Currency](#) if that is set in the Global Trade Settings.

Parameters

- **DailyValues:** When this variable is set to a nonzero value, then the Daily Net Profit/Loss is returned. Otherwise, it returns the Net Profit/Loss for all days for which there is order fill data.

sc.GetTradeAccountData()

Type: Function

int **GetTotalNetProfitLossForAllSymbols**(n_ACSIL::s_TradeAccountDataFields& **r_TradeAccountDataFields**, const SCString& **TradeAccount**);

The **sc.GetTradeAccountData()** function gets all of the Trade Account Data fields for the specified Trade Account.

Parameters

- **r_TradeAccountDataFields:** This is a reference to a variable of structure type n_ACSIL::s_TradeAccountDataFields which receives the Trade Account Data fields. Refer to the example below.
- **TradeAccount:** This is the trade account identifier as a string. To specify the same trade account the chart is set to that the study

instance is applied to, use **sc.SelectedTradeAccount** for this parameter.

Example

```
n_ACSIL::s_TradeAccountDataFields TradeAccountDataFields;  
if (sc.GetTradeAccountData(TradeAccountDataFields, sc.SelectedTradeAccount))  
{  
    double AccountValue = TradeAccountDataFields.m_AccountValue;  
}
```

sc.GetTradeListEntry()

Refer to the [sc.GetTradeListEntry\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetTradeListSize()

Refer to the [sc.GetTradeListSize\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetTradePosition()

Refer to the [sc.GetTradePosition\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetTradePositionByIndex()

Refer to the [sc.GetTradePositionByIndex\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetTradePositionForSymbolAndAccount()

Refer to the [sc.GetTradePositionForSymbolAndAccount\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.GetTradeServiceAccountBalanceForTradeAccount()

Type: Function

```
double GetTradeServiceAccountBalanceForTradeAccount(const SCString&  
TradeAccount);
```

The **sc.GetTradeServiceAccountBalanceForTradeAccount()** function, returns the account balance from the external trading service for the specified **TradeAccount**.

sc.GetTradeStatisticsForSymbolV2()

Type: Function

```
int GetTradeStatisticsForSymbolV2(n_ACSIL::TradeStatisticsTypeEnum StatsType,  
n_ACSIL::s_TradeStatistics& TradeStatistics);
```

The **sc.GetTradeStatisticsForSymbolV2** function obtains the trade statistics for the Symbol and Trade Account of the chart that the study is applied to. These are the same Trade Statistics as the data on the [Trade Statistics](#) tab of the Trade Activity Log.

The **sc.GetTradeStatisticsForSymbolV2** function provides a lot more trade statistics and Flat to Flat trade statistics as compared to the **sc.GetTradeStatisticsForSymbol** function. Refer to that section for field descriptions.

The trade statistics are calculated in the internal Trades list in the chart which loads the available order fills for the Symbol and Trade Account of the chart.

To control the starting Date-Time of the fills, refer to [Understanding and Setting the Start Date-Time for a Trades List](#).

In order for this function to work properly, there needs to be a Trades list being maintained in the chart the study instance is applied to. Therefore, it is necessary to set **sc.MaintainTradeStatisticsAndTradesData = true** in the **sc.SetDefaults** section of the study function when using **sc.GetTradeStatisticsForSymbolV2**.

When the **Trade >> Trade Simulation Mode On** setting is enabled, then the trade statistics will be for simulated trading. When the **Trade >> Trade Simulation Mode On** setting is disabled, then the trade statistics will be for non-simulated trading.

In the case of daily trade statistics it is important to understand the [Daily Reset Time](#).

Parameters

- **StatsType:** Can be one of the following constants:
STATS_TYPE_ALL_TRADES, **STATS_TYPE_LONG_TRADES**,
STATS_TYPE_SHORT_TRADES,
STATS_TYPE_DAILY_ALL_TRADES.

These constants, will retrieve the Trade Statistics for all trades, long trades, short trades, or all trades for the most recent trading day loaded in the Trades list, respectively.

- **TradeStatistics:** The passed n_ACSIL::s_TradeStatistics structure is filled with the requested Trade Statistics data. The Trade Statistics values can then be retrieved from any of the member variables of that structure. For the complete definition of that structure, refer to the [/ACS_Source/scstructures.h](#) file in the Sierra Chart installation folder.

Example

```

n_ACSIL::s_TradeStatistics TradeStatistics;
if (sc.GetTradeStatisticsForSymbolV2(n_ACSIL::STATS_TYPE_DAILY_ALL_TRADES, TradeStatistic
{
    double ClosedTradesProfitLoss = TradeStatistics.ClosedTradesProfitLoss;
}

```

sc.GetTradeSymbol()

Type: Function

```
const SCString& GetTradeSymbol () const;
```

Example



sc.GetTradeWindowOrderType()

Type: Function

The **sc.GetTradeWindowOrderType** function returns the [Order Type](#) which is currently set on the Trade Window of the chart the study instance is applied to.

For the possible return values, refer to [Order Type Constants](#).

sc.GetTradeWindowTextTag()

Type: Function

```
int GetTradeWindowTextTag(SCString& r_TextTag);
```

The **sc.GetTradeWindowTextTag** function gets the [Text Tag](#) setting from the Trade Window for the chart the study instance is applied to. Pass an SCString for the r_TextTag parameter to receive this text.

The function returns 1 if successful, otherwise 0 is returned.

Example

```

SCString TextTag;
sc.GetTradeWindowTextTag(TextTag);

```

sc.GetTradingDayDate()

Type: Function

```
int GetTradingDayDate(const SCDatetime& DateTime);
```

The **sc.GetTradingDayDateForChartNumber()** function returns a [Date Value](#) which is the trading day date for the given **DateTime** parameter.

The trading day date is the date of the trading day that the **DateTime** belongs to based upon the **Session Times** set in **Chart >> Chart Settings** for the chart the study function is applied to.

The returned date will be the same date as the given **DateTime** parameter when the Session Times do not span over midnight. However, in the case where the **Session Start Time** is greater than the **Session End Time** and spans over midnight, then the trading day date will always be the date of the day beginning with midnight during the trading session.

Example

```
SCDateTime TradingDayDate = sc.GetTradingDayDate(sc.BaseDateTimeIn[sc.Index]);
```

sc.GetTradingDayDateForChartNumber()

Type: Function

```
int GetTradingDayDateForChartNumber(int ChartNumber, const SCDatetime& DateTime);
```

The **sc.GetTradingDayDateForChartNumber()** function returns a [Date Value](#) which is the trading day date for the given **ChartNumber** and **DateTime** parameters.

The trading day date is the date of the trading day that the **DateTime** belongs to based upon the **Session Times** set in **Chart >> Chart Settings** for the chart the study function is applied to.

The returned date will be the same date as the given **DateTime** parameter when the Session Times do not span over midnight. However, in the case where the **Session Start Time** is greater than the **Session End Time** and spans over midnight, then the trading day date will always be the date of the day beginning with midnight during the trading session.

Parameters

- **ChartNumber:** The number of the chart to use. This is the number that is shown on the top line in the Chart window, after the #. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want

to get the base data from chart #5 and you want to synchronize the charts, then pass -5.

- **DateTime**: The date and time to use to find the corresponding trading day date.

Example

```
SCDateTime TradingDayDate = sc.GetTradingDayDateForChartNumber(sc.ChartNumber, sc.BaseDate);
```

sc.GetTradingDayStartTimeOfBar()

Type: Function

```
SCDateTime GetTradingDayStartTimeOfBar(SCDateTime& BarDateTime);
```

The **sc.GetTradingDayStartTimeOfBar()** function will return the starting Date-Time of the trading day given a SCDateTime variable. Typically this will be the Date-Time of a bar. The starting Date-Time is based upon the Intraday Session Times settings for the chart.

For an example, refer to the function **scsf_CumulativeSumOfStudy** in the **/ACS_Source/studies8.cpp** file in the folder Sierra Chart is installed to.

Parameters

- **BarDateTime**: An SCDateTime variable.

Example

```
SCDateTime CurrentBarTradingDayStartTime = GetTradingDayStartTimeOfBar(BaseDateT  
SCDateTime CurrentBarTradingDayEndTime = CurrentBarTradingDayStartTime + 24 * HO
```

sc.GetTradingDayStartTimeOfBarForChart()

Type: Function

```
void GetTradingDayStartTimeOfBarForChart(SCDateTime& BarDateTime,  
SCDateTime& r_TradingDayStartTime, int ChartNumber);
```

The **sc.GetTradingDayStartTimeOfBarForChart()** function is identical to the [sc.GetTradingDayStartTimeOfBar](#) function except that it also takes a **ChartNumber** parameter to specify the chart number which it will apply to, which normally will be a different chart than the study that this function is called from, is on.

The resulting trading day start date-time is returned in the **r_TradingDayStartDateTime** SCDatetime parameter which is passed by reference.

Parameters

- **BarDateTime**: An SCDatetime variable of the bar Date-Time.
- **r_TradingDayStartDateTime**: An reference to a SCDatetime variable which receives trading day start date-Time.
- **ChartNumber**: The chart number the function will operate on.

sc.GetTradingErrorMessage()

Type: Function

```
const char * GetTradingErrorMessage(int ErrorCode);
```

Parameters

- **ErrorCode**:.

sc.GetTradingKeyboardShortcutsEnableState()

Type: Function

```
int GetTradingKeyboardShortcutsEnableState();
```

The **sc.GetTradingKeyboardShortcutsEnableState** function gets the state of **Trade >> Trading Keyboard Shortcuts Enabled**.

Refer to [Trading Keyboard Shortcuts Enabled](#).

A return of 0 indicates disabled keyboard shortcuts. A return of 1 indicates enabled keyboard shortcuts.

sc.GetTrueHigh()

Type: Intermediate Study Calculation Function

```
float GetTrueHigh(SCBaseDataRef BaseDataIn, int Index);
```

```
float GetTrueHigh(SCBaseDataRef BaseDataIn); Auto-looping only.
```

The **sc.GetTrueHigh()** function calculates the True High of a bar. The result is returned as a single float value. The True High is either the high of the bar at **Index** or the close of the prior bar, whichever is higher.

Parameters

- [BaseDataIn](#).
- [Index](#).

Example

```
float TrueHigh = sc.GetTrueHigh(sc.BaseDataIn);
```

sc.GetTrueLow()

Type: Intermediate Study Calculation Function

```
float GetTrueLow(SCBaseDataRef BaseDataIn, int Index);
```

```
float GetTrueLow(SCBaseDataRef BaseDataIn); Auto-looping only.
```

The **sc.GetTrueLow()** function calculates the True Low of a bar. The result is returned as a single float value. The True Low is either the low of the bar at **Index** or the close of the prior bar. Whichever is lower.

Parameters

- [BaseDataIn](#).
- [Index](#).

Example

```
float TrueLow = sc.GetTrueLow(sc.BaseDataIn);
```

sc.GetTrueRange()

Type: Intermediate Study Calculation Function

```
float GetTrueRange(SCBaseDataRef BaseDataIn, int Index);
```

```
float GetTrueRange(SCBaseDataRef BaseDataIn); Auto-looping only.
```

The **sc.GetTrueRange()** function calculates the True Range of a bar at **Index**. The result is returned as a single float value.

Parameters

- [BaseDataIn](#).
- [Index](#).

Example

```
float TrueRange = sc.GetTrueRange(sc.BaseDataIn);
```

sc.GetUserDrawingByLineNumber()

Refer to the [sc.GetUserDrawingByLineNumber\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

sc.GetUserDrawnChartDrawing()

Refer to the [sc.GetUserDrawnChartDrawing\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

sc.GetUserDrawnDrawingByLineNumber()

Refer to the [sc.GetUserDrawnDrawingByLineNumber\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

sc.GetUserDrawnDrawingsCount()

Type: Function

```
int GetUserDrawnDrawingsCount(int ChartNumber, int ExcludeOtherStudyInstances);
```

The **sc.GetUserDrawnDrawingsCount** function

Parameters

- **ChartNumber:**
- **ExcludeOtherStudyInstances:**

sc.GetValueFormat()

Type: Function

```
int GetValueFormat();
```

Example

sc.GetVolumeAtPriceDataForStudyProfile()

Type: Function

```
int GetVolumeAtPriceDataForStudyProfile(const int StudyID, const int ProfileIndex,
```

```
const int PricelIndex, s_VolumeAtPriceV2& VolumeAtPrice);
```

The **sc.GetVolumeAtPriceDataForStudyProfile** function fills out a **s_VolumeAtPriceV2** structure passed to the function. When filled in, the structure contains the volume data for a price level for the specified Volume Profile or TPO profile.

The Volume Profile can be obtained from a [TPO Profile Chart](#) or [Volume by Price](#) study on the chart.

TPO profiles also contain volume data at each price level which can be obtained. In the case of TPO Profiles, the **s_VolumeAtPriceV2::NumberOfTrades** member contains the number of TPOs at the price level.

The function returns 1 if successful. Otherwise, 0 is returned.

For an example to use this function, refer to the `scsf_GetVolumeAtPriceDataForStudyProfileExample` function in `/ACS_Source/Studies2.cpp` file in the Sierra Chart installation folder.

Parameters

- **StudyID**: The unique study identifier for the **Volume by Price** or **TPO Profile Chart** study. Refer to [UniqueStudyInstanceIdentifiers](#).
- **ProfileIndex**: The zero-based index of the volume profile relative to the end of the chart. 0 equals the latest profile in the chart at the end or rightmost side. This needs to always be set to a positive number.
- **PricelIndex**: The zero-based price index. Zero is the lowest price and increasing numbers are for increasing prices. The last valid **PricelIndex** can be determined by calling [sc.GetNumPriceLevelsForStudyProfile](#) and subtracting 1.
- **VolumeAtPrice**: A reference to the **s_VolumeAtPriceV2** data structure. This structure will be filled in with the volume data for **PricelIndex** when the function returns. For the data members of this structure, refer to the `/ACS_Source/VAPContainer.h` file. To access the data members of this structure, just simply directly access the member variables. There are no functions used with it.

sc.GetYValueForChartDrawingAtBarIndex()

Type: Function

```
int32_t GetYValueForChartDrawingAtBarIndex(const int32_t ChartNumber, const int32_t IsUserDrawn, const int32_t LineNumber, const int32_t LineIndex, const int32_t BarIndex1, int32_t BarIndex2, float& YValue1, float& YValue2);
```

Parameters

- [ChartNumber.](#)
- **IsUserDrawn**
- **LineNumber**
- **LineIndex**
- [BarIndex1.](#)
- [BarIndex2.](#)
- **YValue1**
- **YValue2**

Example



sc.HeikinAshi()

Type: Function

```
void HeikinAshi(SCBaseDataRef BaseDataIn, SCSubgraphRef HeikinAshiOut, int Index,  
int Length, int SetCloseToCurrentPriceAtLastBar);
```

```
void HeikinAshi(SCBaseDataRef BaseDataIn, SCSubgraphRef HeikinAshiOut, int  
Length, int SetCloseToCurrentPriceAtLastBar); Auto-looping only.
```

Parameters

- [BaseDataIn.](#)
- [HeikinAshiOut.](#) For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for additional results output. HeikinAshiOut.Data is the Open price array. HeikinAshiOut.Arrays[0] is the High price array. HeikinAshiOut.Arrays[1] is the Low price array. HeikinAshiOut.Arrays[2] is the Last/Close price array.
- [Index.](#)
- [Length.](#)
- **SetCloseToCurrentPriceAtLastBar:** When this is set to a nonzero value, then the last/close price of the last Heikin-Ashi bar is set to the current price of the underlying data.

sc.Highest()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef Highest(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef  
FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef Highest(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef  
FloatArrayOut, int Length); Auto-looping only.
```

The **sc.Highest()** function calculates the highest value of the data in **FloatArrayIn** over the specified **Length** beginning at **Index**. The result is put into the **FloatArrayOut** array.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.Highest(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0], 20);  
  
float Highest = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.HullMovingAverage()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **HullMovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **HullMovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.HullMovingAverage()** function calculates the Hull Moving Average study.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

Example

```
sc.HullMovingAverage(sc.BaseData[SC_LAST], sc.Subgraph[0], 10);  
  
float HullMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.HurstExponent()

Type: Function

SCFloatArrayRef **HurstExponent** (SCFloatArrayRef **In**, SCSubgraphRef **Out**, int **Index**, int **LengthIndex**);

SCFloatArrayRef **HurstExponent** (SCFloatArrayRef **In**, SCSubgraphRef **Out**, int **LengthIndex**); [Auto-looping only](#).

Parameters

- [In](#).
- [Out](#).
- [Index](#).
- [LengthIndex](#).

Example

sc.InstantaneousTrendline()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **InstantaneousTrendline**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **InstantaneousTrendline**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.InstantaneousTrendline()** function calculates Ehlers' Instantaneous Trendline study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.InstantaneousTrendline(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
  
float InstantaneousTrendline = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.InverseFisherTransform()

Type: Function

```
void InverseFisherTransform (SCFloatArrayRef In, SCSubgraphRef Out, int Index, int HighestLowestLength, int MovingAverageLength, int MovAvgType);
```

```
void InverseFisherTransform (SCFloatArrayRef In, SCSubgraphRef Out, int HighestLowestLength, int MovingAverageLength, int MovAvgType); Auto-looping only.
```

Parameters

- [In](#).
- [Out](#).
- [Index](#).
- [HighestLowestLength](#).
- [MovingAverageLength](#).
- [MovAvgType](#).

Example

sc.InverseFisherTransformRSI()

Type: Function

```
void InverseFisherTransformRSI (SCFloatArrayRef In, SCSubgraphRef Out, int Index, int RSILength, int InternalRSIMovAvgType, int RSIMovingAverageLength, int MovingAverageOfRSIType);
```

```
void InverseFisherTransformRSI (SCFloatArrayRef In, SCSubgraphRef Out, int RSILength, int InternalRSIMovAvgType, int RSIMovingAverageLength, int MovingAverageOfRSIType); Auto-looping only.
```

Parameters

- [In](#).
- [Out](#).
- [Index](#).
- [RSILength](#).
- [InternalRSIMovAvgType](#).
- [RSIMovingAverageLength](#).
- [MovingAverageOfRSIType](#).

Example

Example

sc.IsChartDataLoadingCompleteForAllCharts()

Type: Function

```
int IsChartDataLoadingCompleteForAllCharts();
```

The **sc.IsChartDataLoadingCompleteForAllCharts** function returns a nonzero value when the loading of chart data from the local chart data files is complete for all charts within the Chartbook containing the chart that the study instance calling this function is applied to.

Otherwise, this function returns 0 while one or more charts is loading data from the local chart data files.

Example

```
bool LoadingIsComplete = sc.IsChartDataLoadingCompleteForAllCharts();
```

sc.IsChartDataLoadingInChartbook()

Type: Function

```
int IsChartDataLoadingInChartbook();
```

The **sc.IsChartDataLoadingInChartbook()** function returns 1 when it is called and an Intraday chart is loading data in the same Chartbook that the chart belongs to that the study instance is applied to. So this can be during the time the Chartbook is in the process of being opened or at any time after. An example of an intraday chart loading data would be when certain chart settings are changed through **Chart >> Chart Settings**.

If none of the Intraday charts are loading data within the Chartbook, then the return value is 0.

sc.IsChartNumberExist()

Type: Function

```
int IsChartNumberExist(int ChartNumber, const SCString& ChartbookFileName);
```

The **sc.IsChartNumberExist** function returns a value of 1 if the specified **ChartNumber** exists within the given **ChartbookFileName**, otherwise it returns a value of 0.

The parameter **ChartbookFileName** may be an empty string (""), in which case the Chartbook used to check for the **ChartNumber** is the Chartbook that contains the study from which this function is called. The **ChartbookFileName** must also contain the **.cht** extension,

but it does not need to contain the complete path. Only the file name.

sc.IsChartZoomInStateActive()

Type: Function

```
int IsChartZoomInStateActive();
```

The **sc.IsChartZoomInStateActive** function returns 1 if **Tools >> Zoom In** mode is active for the chart the study instance is applied to. Otherwise, 0 is returned .

sc.IsDateTimeContainedInBarIndex()

Type: Function

```
int IsDateTimeContainedInBarIndex(const SCDatetime& DateTime, int BarIndex) const;
```

The **sc.IsDateTimeContainedInBarIndex()** function returns TRUE if the specified **DateTime** is within the time range of the specified **BarIndex**. The function returns FALSE otherwise.

Falling within the time range means that the DateTime is greater than or equal to the time represented by the bar and is less than the time represented by the next bar.

sc.IsDateTimeContainedInBarAtIndex()

Type: Function

```
bool IsDateTimeContainedInBarAtIndex(const SCDatetime& DateTime, int BarIndex);
```

Parameters

- [DateTime](#).
- [Index](#).

Example

sc.IsDateTimeInDaySession()

Type: Function

```
int IsDateTimeInDaySession(const SCDatetime& DateTime);
```

The **sc.IsDateTimeInDaySession()** function returns TRUE (non-zero value) or FALSE indicating whether the given **DateTime** is within the Day Session Times in **Chart >> Chart Settings**. **DateTime** is a [SCDateTime](#) type.

Example

```
int IsInDaySession = sc.IsDateTimeInDaySession(sc.BaseDateTimeIn[sc.Index]+ 1*HOURS);
```

sc.IsDateTimeInEveningSession()

Type: Function

```
int IsDateTimeInEveningSession(const SCDatetime &DateTime);
```

The **sc.IsDateTimeInEveningSession** returns a value of **1** if the specified time in **DateTime** is within the defined hours for the Evening Session as set by the [Evening Start](#) and [Evening End](#) times and a **0** otherwise. If the option for [Use Evening Session](#) is not set, then this function returns a value of **0**.

sc.IsDateTimeInSession()

Type: Function

```
int IsDateTimeInSession(const SCDatetime& DateTime);
```

The **sc.IsDateTimeInSession()** function returns TRUE (non-zero value) or FALSE indicating whether the given **DateTime** is within the Session Times in **Chart >> Chart Settings**. **DateTime** is an [SCDateTime](#) type.

Example

```
int IsInSession = sc.IsDateTimeInSession(sc.BaseDateTimeIn[sc.Index]+ 1*HOURS);
```

sc.IsIsSufficientTimePeriodInDate()

Type: Function

```
bool IsIsSufficientTimePeriodInDate(const SCDatetime& DateTime, float Percentage);
```

Parameters

- [DateTime](#).
- **Percentage**.

Example

sc.IsMarketDepthDataCurrentlyAvailable()

Type: Function

```
int IsMarketDepthDataCurrentlyAvailable()
```

The **sc.IsMarketDepthDataCurrentlyAvailable** function returns 1 when the **sc.SymbolData->BidDOM** and **sc.SymbolData->AskDOM** contain market depth data at levels 1 and 2. This is an indication that market depth data is currently available in those arrays.

For additional information, refer to [sc.SymbolData](#).

sc.IsNewBar()

Type: Function

```
bool IsNewBar(int BarIndex);
```

Parameters

- [BarIndex](#).

sc.IsNewTradingDay()

Type: Function

```
bool IsNewTradingDay(int BarIndex);
```

The **sc.IsNewTradingDay()** function returns 1 if the chart bar index specified by the **BarIndex** parameter is the start of a new trading day according to the [Session Times](#) set for the chart the study instance is applied to. Otherwise, zero is returned.

BarIndex is the same index used with the second array operator with **sc.BaseData[][]**. For additional information, refer to [Working with ACSIL Arrays and Understanding Looping](#).

Even if the prior bar has the same date as the date of the **BarIndex** bar, this can still be the start of a new trading day in the case when reversed session times are used or when the **Use Evening Session** option is enabled. Refer to [Session Times](#).

sc.IsReplayRunning()

Type: Function

```
int IsReplayRunning();
```

sc.IsReplayRunning() returns TRUE (1) if a replay is running on the chart the study instance is applied to or the replay is paused. Otherwise, it returns FALSE (0).

Internally this function simply checks the state of **sc.ReplayStatus** and checks if the replay

is not stopped.

Example

```
int ReplayRunning = sc.IsReplayRunning();
```

sc.IsSwingHigh()

Type: Intermediate Study Calculation Function

```
int IsSwingHigh (SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
int IsSwingHigh (SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.IsSwingHigh()** function returns TRUE (1) if there is a Swing High. Otherwise, it returns FALSE (0).

Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

Example

```
int SwingHigh = sc.IsSwingHigh(sc.BaseData[SC_HIGH],2);
```

sc.IsSwingLow()

Type: Intermediate Study Calculation Function

```
int IsSwingLow(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
int IsSwingLow(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.IsSwingLow()** function returns TRUE (1) if there is a Swing Low. Otherwise, it returns FALSE (0).

Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

Example

```
int SwingLow = sc.IsSwingLow(sc.BaseData[SC_LOW],2);
```

sc.IsVisibleSubgraphDrawStyle()

Type: Function

```
bool IsVisibleSubgraphDrawStyle(int DrawStyle);
```

Parameters

- **DrawStyle:** .

Example

IsWorkingOrderStatus()

Refer to the [IsWorkingOrderStatus\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

IsWorkingOrderStatusIgnorePendingChildren()

Refer to the [IsWorkingOrderStatusIgnorePendingChildren\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.Keltner()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef Keltner(SCBaseDataRef BaseDataIn, SCFloatArrayRef FloatArrayIn,  
SCSubgrapRef SubgraphOut, int Index, int KeltnerMovAvgLength, unsigned int  
KeltnerMovAvgType, int TrueRangeMovAvgLength, unsigned int  
TrueRangeMovAvgType, float TopBandMultiplier, float BottomBandMultiplier);
```

```
SCFloatArrayRef Keltner(SCBaseDataRef BaseDataIn, SCFloatArrayRef FloatArrayIn,  
SCSubgrapRef SubgraphOut, int KeltnerMovAvgLength, unsigned int  
KeltnerMovAvgType, int TrueRangeMovAvgLength, unsigned int  
TrueRangeMovAvgType, float TopBandMultiplier, float BottomBandMultiplier); Auto-  
looping only.
```

The **sc.Keltner()** function calculates the Keltner study.

Parameters

- [BaseDataIn](#).
- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [KeltnerMovAvgLength](#).
- [KeltnerMovAvgType](#).
- [TrueRangeMovAvgLength](#).
- [TrueRangeMovAvgType](#).
- [TopBandMultiplier](#).
- [BottomBandMultiplier](#).

Example

```
sc.Keltner(
    sc.BaseDataIn,

    sc.BaseDataIn[SC_LAST],
    sc.Subgraph[0],
    10,

    MOVAVGTYPE_SIMPLE,
    10,
    MOVAVGTYPE_WILDERS,
    1.8f,
    1.8f,
);

//Access the individual Keltner lines
float KeltnerAverageOut = sc.Subgraph[0][sc.Index];

float TopBandOut = sc.Subgraph[0].Arrays[0][sc.Index];

float BottomBandOut = sc.Subgraph[0].Arrays[1][sc.Index];

//Copy to Visible Subgraphs
sc.Subgraph[1][sc.Index] = TopBandOut;
sc.Subgraph[2][sc.Index] = BottomBandOut;
```

sc.LaguerreFilter()

Type: [Intermediate Study Calculation Function](#)

SCFloatArrayRef **LaguerreFilter**(SCFloatArrayRef In, SCSubgraphRef Out, int IndexParam, float DampingFactor);

SCFloatArrayRef **LaguerreFilter**(SCFloatArrayRef In, SCSubgraphRef Out, float DampingFactor); [Auto-looping only](#).

The **sc.LaguerreFilter()** function calculates the Laguerre Filter study.

Parameters

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [IndexParam](#).
- **DampingFactor**: A factor to determine the weight given to both current and previous values of the Input Data.

Example

```
sc.LaguerreFilter(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 0.8);  
  
float LaguerreFilter = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.LinearRegressionIndicatorAndStdErr()

Type: [Intermediate Study Calculation Function](#)

SCFloatArrayRef **LinearRegressionIndicatorAndStdErr**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, SCFloatArrayRef **StdErr**, int **Length**);

SCFloatArrayRef **LinearRegressionIndicatorAndStdErr**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, SCFloatArrayRef **StdErr**, int **Index**, int **Length**); [Auto-looping only](#).

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- **StdErr**: .
- [Index](#).
- [Length](#).

Example

sc.LinearRegressionIntercept()

Type: Function

SCFloatArrayRef **LinearRegressionIntercept**(SCFloatArrayRef **In**, SCFloatArrayRef **Out**, int **Index**, int **Length**);

The **LinearRegressionIntercept()** function .

Parameters

- In: .
- Out: .
- Index: .
- Length: .

Example

sc.LinearRegressionSlope()

Type: Function

```
SCFloatArrayRef LinearRegressionSlope(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);
```

The **LinearRegressionSlope()** function .

Parameters

- In: .
- Out: .
- Index: .
- Length: .

Example

sc.LinearRegressionIndicator()

Type: [Intermediate Study Calculation Function](#)

```
SCFloatArrayRef LinearRegressionIndicator(SCFloatArrayRef FloatArrayIn,  
SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef LinearRegressionIndicator(SCFloatArrayRef FloatArrayIn,  
SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.LinearRegressionIndicator()** function calculates the [Moving Average - Linear Regression](#) study. The result is placed into **FloatArrayOut** at the array index specified by **Index**.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.LinearRegressionIndicator(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);  
  
float LinearRegression = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.Lowest()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Lowest**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **Lowest**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.Lowest()** function calculates the lowest value of the data in **FloatArrayIn** over the specified **Length** beginning at **Index**. The result is put into the **FloatArrayOut** array.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.Lowest(sc.BaseDataIn[SC_LOW], sc.Subgraph[0], 20);  
  
float Lowest = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.MACD()

Type: Intermediate Study Calculation Function.

SCFloatArrayRef **sc.MACD** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **FastMALength**, int **SlowMALength**, int **MACDMALength**, int

MovAvgType);

SCFloatArrayRef **sc.MACD** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **FastMALength**, int **SlowMALength**, int **MACDMALength**, int **MovAvgType**); [Auto-looping only](#).

The **sc.MACD()** function calculates the standard Moving Average Convergence Divergence study.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [FastMALength](#).
- [SlowMALength](#).
- [MACDMALength](#).
- [MovingAverageType](#).

Example

```
sc.MACD(sc.BaseData[SC_LAST], sc.Subgraph[0], 5, 10, 10, MOVAVGTYPE_SIMPLE);

//Access the individual lines
float MACD = sc.Subgraph[0][sc.Index]; //Access the study value at the current index

float MACDMovingAverage = sc.Subgraph[0].Arrays[2][sc.Index];

float MACDDifference = sc.Subgraph[0].Arrays[3][sc.Index];

//Copy to Visible Subgraphs
sc.Subgraph[1][sc.Index] = MACDMovingAverage;
sc.Subgraph[2][sc.Index] = MACDDifference;
```

sc.MakeHTTPBinaryRequest()

Type: Function

int MakeHTTPBinaryRequest(const SCString& **URL**);

The **sc.MakeHTTPBinaryRequest()** function is identical to [sc.MakeHTTPRequest\(\)](#), except that it is designed to be used in cases where the data returned by the Web server is non-text data.

When the request completes, the returned data from the Web server is placed into the **sc.HTTPBinaryResponse** variable and the study function will be called again.

This response is a SCConstCharArray type and holds an array of bytes/characters. Refer to [/ACS_Source/SCStructures.h](#) for additional information on this class type.

The return value is zero if there is an error. If there is no error, the return value is the request identifier which is later returned in the **sc.HTTPRequestID** member when the study function is called when the data is received after the response completes.

Parameters

- **URL**: The full URL of the website resource. Example:
<http://www.sierrachart.com/ACSiLHTTPTest.txt>

sc.MakeHTTPPOSTRequest()

Type: Function

```
int MakeHTTPPOSTRequest(const SCString& URL, const SCString& POSTData, const  
n_ACSIL::s_HTTPHeader* Headers, int NumberOfHeaders);
```

Parameters

- **URL**..
- **POSTData**..
- **Headers**..
- **NumberOfHeaders**..

Example

sc.MakeHTTPRequest()

Type: Function

```
int MakeHTTPRequest(const SCString& URL);
```

The **sc.MakeHTTPRequest()** function makes a website Hypertext Transfer Protocol (HTTP/HTTPS) request over the network. The website address and the file to request are contained in the **URL** parameter.

The **URL** parameter needs to contain the website address and the file to retrieve. Standard HTTP GET parameters can also be added.

This function is nonblocking and immediately returns.

When the request is complete, the response will be placed into the **sc.HTTPResponse** SCString member. At the time the request is complete, the study instance which made the request will be called and at that time the **sc.HTTPResponse** member can be checked.

If there is an error with making the request to the remote server or the server returns an

error, **sc.HTTPResponse** will be set to "ERROR".

Returns 1 if successful. Returns 0 on error.

If **sc.MakeHTTPRequest** returns 0, then **sc.HTTPResponse** will be set to "HTTP_REQUEST_ERROR". Otherwise, **sc.HTTPResponse** will be set to an empty string after calling the **sc.MakeHTTPRequest** function and **sc.HTTPResponse** will be set to the response from the server when the server later responds after the study function returns.

It is only supported to make one request at a time. The current request must finish before another one can be made.

The only persistent memory used for a HTTP/HTTPS request will be the maximum size of the data received among the HTTP/HTTPS requests by a study. There will be some temporary memory use for the network socket connection, but that will get released when the request is complete.

Also refer to [sc.MakeHTTPBinaryRequest](#), [sc.MakeHTTPPOSTRequest\(\)](#).

Each chart has its own HTTP object. Therefore, when the chart is closed, all of the outstanding HTTP requests are deleted and that time and will never complete.

Example

```

enum {REQUEST_NOT_SENT = 0, REQUEST_SENT, REQUEST_RECEIVED};
int& RequestState = sc.GetPersistentInt(1);

if (sc.Index == 0)
{
    if (RequestState == REQUEST_NOT_SENT)
    {
        // Make a request for a text file on the server. When the request is complete and all of the data
        // has been downloaded, this study function will be called with the file placed into the sc.HTTPRes

        if (!sc.MakeHTTPRequest("https://www.sierrachart.com/ACSILHTTPTest.txt"))
        {
            sc.AddMessageToLog("Error making HTTP request.", 1);

            // Indicate that the request was not sent.
            // Therefore, it can be made again when sc.Index equals 0.
            RequestState = REQUEST_NOT_SENT;
        }
        else
        {
            RequestState = REQUEST_SENT;
        }
    }
}

if (RequestState == REQUEST_SENT && sc.HTTPResponse != "")
{
    RequestState = REQUEST_RECEIVED;

    // Display the response from the Web server in the Message Log

    sc.AddMessageToLog(sc.HTTPResponse, 1);
}
else if (RequestState == REQUEST_SENT && sc.HTTPResponse == "")
{
    //The request has not completed, therefore there is nothing to do so we will return
    return;
}

```

sc.Momentum()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Momentum**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **Momentum**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.Momentum()** function calculates the momentum.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.Momentum(sc.BaseDataIn[SC_LAST], sc.Subgraph[0].Arrays[0], 20);  
  
float Momentum = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.MovingAverage()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **MovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, unsigned int **MovingAverageType**, int **Index**, int **Length**);

SCFloatArrayRef **MovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, unsigned int **MovingAverageType**, int **Length**); [Auto-looping only](#).

The **sc.MovingAverage()** function calculates a Moving Average of the specified Type and Length.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).
- [MovingAverageType](#).

Example

```
sc.MovingAverage(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], MOVAVGTYPE_SIMPLE, 20);  
  
float MovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.MovingAverage()

Type: Intermediate Study Calculation Function

void **MovingAverageCumulative**(SCFloatArrayRef **In**, SCFloatArrayRef **Out**, int **Index**);

void **MovingAverageCumulative**(SCFloatArrayRef **In**, SCFloatArrayRef **Out**); [Auto-looping only](#).

Parameters

- [In](#).

- [Out.](#)
- [Index.](#)

Example

sc.MovingMedian()

Type: Function

SCFloatArrayRef **MovingMedian**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **MovingMedian**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Length**);

The **sc.MovingMedian()** function calculates the Moving Median of the specified Length.

Parameters

- [FloatArrayIn.](#)
- [SubgraphOut.](#)
- [Index.](#)
- [Length.](#)

Example

```
SCSubgraphRef Median = sc.Subgraph[0];

SCInputRef InputData = sc.Input[0];
SCInputRef Length = sc.Input[1];
SCFloatArrayRef In = sc.BaseDataIn[InputData.GetInputDataIndex()];

sc.MovingMedian(In, Median, Length.GetInt());
```

sc.MultiplierFromVolumeValueFormat()

Type: Function

float **MultiplierFromVolumeValueFormat**() const;

The **sc.MultiplierFromVolumeValueFormat()** function uses the **Chart >> Chart Settings >> Symbol >> Volume Value Format** setting to determine the multiplier to use to multiply a volume value by to return the actual true volume value.

This is used when volumes have fractional values that are stored as an integer.

Example

```
float ActualVolume = sc.Volume[sc.Index] * sc.MultiplierFromVolumeValueFormat();
```

sc.NumberOfBarsSinceHighestValue()

Type: Intermediate Study Calculation Function

```
int NumberOfBarsSinceHighestValue (SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
int NumberOfBarsSinceHighestValue (SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.NumberOfBarsSinceHighestValue()** function calculates the number of bars between the highest value in the **FloatArrayIn** array, which is determined over the specified **Length**, and the **Index** element of the **FloatArrayIn** array. The highest value is calculated over the range from **Index** to **Index-Length+1**.

Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

Example

```
int NumBars = sc.NumberOfBarsSinceHighestValue(sc.BaseDataIn[SC_LAST], 10);
```

sc.NumberOfBarsSinceLowestValue()

Type: Intermediate Study Calculation Function

```
int NumberOfBarsSinceLowestValue (SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
int NumberOfBarsSinceLowestValue (SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.NumberOfBarsSinceLowestValue()** function calculates the number of bars between the lowest value in the **FloatArrayIn** array, which is determined over the specified **Length**, and the **Index** element of the **FloatArrayIn** array. The lowest value is calculated over the range from **Index** to **Index-Length+1**.

Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

Example

```
int NumBars = sc.NumberOfBarsSinceLowestValue(sc.BaseDataIn[SC_LAST], 10);
```

sc.OnBalanceVolume()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **OnBalanceVolume** (SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**);

SCFloatArrayRef **OnBalanceVolume** (SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**); [Auto-looping only](#).

The **sc.OnBalanceVolume()** function calculates the On Balance Volume study.

Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- [Index](#).

Example

```
sc.OnBalanceVolume(sc.BaseDataIn, sc.Subgraph[0]);  
  
float OnBalanceVolume = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.OnBalanceVolumeShortTerm()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **OnBalanceVolumeShortTerm**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **OnBalanceVolumeShortTerm**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.OnBalanceVolumeShortTerm()** function calculates the On Balance Volume Short

Term study.

Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.OnBalanceVolumeShortTerm(sc.BaseDataIn, sc.Subgraph[0], 20);  
  
float ShortTermOnBalanceVolume = sc.Subgraph[0][sc.Index]; //Access the study value at the current
```

sc.OpenChartbook()

Type: Function

```
void OpenChartbook(const SCString& ChartbookFileName);
```

The **sc.OpenChartbook** function opens the Chartbook specified by the **ChartbookFileName** parameter.

The **ChartbookFileName** must include the file extension (.cht). The **ChartbookFileName** can include the full path to the file, but is not required.

If the full path is not specified, the default path to the data files, as defined by the [Data Files Folder](#) setting is used.

sc.OpenChartOrGetChartReference()

Type: Function

```
int OpenChartOrGetChartReference(s_ACSOpenChartParameters &  
ACSOpenChartParameters);
```

The **sc.OpenChartOrGetChartReference** function returns the Chart Number of a chart matching a given set of parameters if there is already an open chart within the Chartbook that contains the study calling the **sc.OpenChartOrGetChartReference** function, that matches all of the given set of parameters. In this case the Chart Number of that chart will be returned.

Otherwise, a new chart will be automatically opened with the given set of parameters, and the Chart Number of the new chart will be returned.

If there is an error encountered within the function, the function return value will be **0**.

For an example to use this function, refer to the **scsf_OpenChartOrGetChartReferenceExample** function in the **/ACS_Source/studies.cpp** file located in the Sierra Chart installation folder on your system.

If the chart that has been automatically opened with this function, does not need to be viewed, then its window can be minimized to prevent any graphics output which eliminates the CPU load related to graphics output.

After you obtain the Chart Number with this function, then you can access the data from this chart with the [sc.GetChartData\(\)](#) function.

For efficiency, only call the **sc.OpenChartOrGetChartReference** function when **sc.IsFullRecalculation** is nonzero.

This function cannot be called from within the [sc.SetDefaults](#) code block. It will result in an incorrect Chart Number being used, when a Chartbook is being opened and one of the studies in the Chartbook makes a call to the **sc.OpenChartOrGetChartReference()** function. Therefore, it must be called below the **sc.SetDefaults** code block.

Parameters

The parameters of the requested chart are specified using the **s_ACSOpenChartParameters** structure. This structure contains the following parameters:

- **PriorChartNumber**: This is the first parameter that gets checked. If the **sc.OpenChartOrGetChartReference** function has been called at least once before, remember the Chart Number that gets returned, and set it into this parameter. This allows for a more efficient lookup. Set this parameter to **0** if the Chart Number is unknown.
- **ChartDataTypes**: Set this to **DAILY_DATA** for a Historical Daily data chart, or **INTRADAY_DATA** for an Intraday data chart.
- **Symbol**: Set this to the symbol for the requested chart. The symbol must be in the format required by the selected **Service** in **Global Settings >> Data/Trade Service Settings**.

These will be the same symbols that you will see listed in **File >> Find Symbol**.

When you need to use the Symbol of the chart the study function is on, the easiest way to get the correct symbol is to use the **sc.Symbol** ACSIL structure member.

- **IntradayBarPeriodType**: This is used for an Intraday chart and sets the period/type of the chart bar. Must be one of the below values. It is also necessary to set the actual associated value through the **IntradayBarPeriodLength** member.
 - **IBPT_DAYS_MINS_SECS** : Specify the time length in

seconds through **IntradayBarPeriodLength**.

- **IBPT_VOLUME_PER_BAR**
- **IBPT_NUM_TRADES_PER_BAR** : The number of trades per bar.
- **IBPT_RANGE_IN_TICKS_STANDARD**
- **IBPT_RANGE_IN_TICKS_NEWBAR_ON_RANGEMET**
- **IBPT_RANGE_IN_TICKS_TRUE**
- **IBPT_RANGE_IN_TICKS_FILL_GAPS**
- **IBPT_REVERSAL_IN_TICKS**
- **IBPT_RENKO_IN_TICKS**
- **IBPT_DELTA_VOLUME_PER_BAR**
- **IBPT_FLEX_RENKO_IN_TICKS**
- **IBPT_RANGE_IN_TICKS_OPEN_EQUAL_CLOSE**
- **IBPT_PRICE_CHANGES_PER_BAR**
- **IBPT_MONTHS_PER_BAR**

To determine the **Bar Period Type** that the chart currently uses, call functions like [sc.AreTimeSpecificBars\(\)](#).

- **IntradayBarPeriodLength**: This is used for Intraday charts, in conjunction with the **IntradayBarPeriodType** parameter.
- **DaysToLoad**: This sets the number of days of data that will be loaded into the chart whether it is a Historical Daily chart or an Intraday chart. If this value is left at 0, then the

Chart >> Chart Settings >> Use Number of Days to Load >> Days to Load setting of the calling chart is used.

When getting a reference to an existing chart, if the existing chart has a higher Days to Load setting, then that chart will be returned and that Days to Load setting for the existing chart will remain.

- **SessionStartTime, SessionEndTime**: These specify the primary Session Start and End times. These session times are optional and only apply to Intraday charts. If the session times are not specified, then the Intraday Session Times from [Global Symbol Settings](#) are used.
- **EveningSessionStartTime, EveningSessionEndTime**: The evening session times are optional, and only apply to Intraday charts. If the evening session times are not specified, then the Intraday Session Times from [Global Symbol Settings](#) are used.
- **IntradayBarPeriodParm2**: This is used when **IntradayBarPeriodType** is set to **IBPT_FLEX_RENKO_IN_TICKS**. This is the Trend Offset.
- **IntradayBarPeriodParm3**: This is used when **IntradayBarPeriodType** is set to **IBPT_FLEX_RENKO_IN_TICKS**. This is the Reversal Offset.
- **IntradayBarPeriodParm4**: This is used when **IntradayBarPeriodType** is set to **IBPT_FLEX_RENKO_IN_TICKS**. This is the **New Bar When Exceeded** option. This can be set to 1 or 0.
- **HistoricalChartBarPeriod**: In the case of when **ChartData Type** is set to **DAILY_CHART**, then this specifies the time period for each chart bar. It can be one of the following constants:

- **HISTORICAL_CHART_PERIOD_DAYS**
 - **HISTORICAL_CHART_PERIOD_WEEKLY**
 - **HISTORICAL_CHART_PERIOD_MONTHLY**
 - **HISTORICAL_CHART_PERIOD_QUARTERLY**
 - **HISTORICAL_CHART_PERIOD_YEARLY**
- **HistoricalChartBarPeriodLengthInDays**: When **HistoricalChartBarPeriod** is set to **HISTORICAL_CHART_PERIOD_DAYS**, then this variable can be set to the number of days per bar in a Historical Chart. The default is 1. This variable only applies to Historical charts and not Intraday charts.
- **int ChartLinkNumber**: This can be optionally set to a nonzero [Chart Link Number](#). The default is 0.
- **ContinuousFuturesContractOption**: This can be set to one of the following constants:
 - **CFCO_NONE**
 - **CFCO_DATE_RULE_ROLLOVER**
 - **CFCO_VOLUME_BASED_ROLLOVER**
 - **CFCO_DATE_RULE_ROLLOVER_BACK_ADJUSTED**
 - **CFCO_VOLUME_BASED_ROLLOVER_BACK_ADJUSTED**
- **LoadWeekendData**: Set this to 1 to load weekend (Saturday and Sunday) data, the default, or to zero to not load weekend data. This only applies to Intraday charts.
- **UseEveningSession**: Set this to 1 to use the Session Times >> Evening Start and Evening End times set for the chart.
- **HideNewChart**: Set this to 1 to cause the chart to be hidden and not visible. It can be displayed through the **CW** menu. To minimize system resources, it is recommended to enable **Global Settings >> General Settings >> GUI >> Application GUI >> Destroy Chart Windows When Hidden**.
- **AlwaysOpenNewChart**: Set this to 1 to always open a new chart when calling the **sc.OpenChartOrGetChartReferencefunction** function under all conditions.
- **IsNewChart**: This variable is set to 1, if a new chart is opened by the **sc.OpenChartOrGetChartReferencefunction** function. So this variable is not set by your study function but it is set by **sc.OpenChartOrGetChartReferencefunction** upon returning.
- **IncludeColumnsWithNoData**: Set this to 1, to enable the [Include Columns With No Data](#) setting for the chart.
- **UpdatePriorChartNumberParametersToMatch**: This variable only applies when **PriorChartNumber** has been set and the chart number specified actually exists within the Chartbook which contains the chart, which contains the study instance calling the **sc.OpenChartOrGetChartReferencefunction** function.

When this is set to 1 and the chart parameters do not match the existing found chart, that chart will have its parameters updated to match.

- **ChartTimeZone:** This is an optional parameter and specifies the Time Zone to use for the chart as a string. If is not specified, the global time zone will be used. The available time zones are listed in the /ACS_Source/SCConstants.h file. You can get the time zone for the chart, that your study instance is applied to, through the **sc.GetChartTimeZone()** function.

sc.OpenFile()

Type: Function

```
int OpenFile(const SCString& PathAndFileName, const int Mode, int& FileHandle);
```

The **sc.OpenFile** function opens the file specified by **PathAndFileName** per the specified **Mode** and puts the File Handle into **FileHandle**.

The function returns **True** if the file is able to be opened in the specified Mode. Otherwise it returns **False**.

Parameters

- **PathAndFileName:** An [SCString](#) variable with the path and filename to the file.
- **Mode:** An enumeration that sets the mode in which the file will be opened. The following are available:
 - FILE_MODE_CREATE_AND_OPEN_FOR_READ_WRITE
 - FILE_MODE_OPEN_EXISTING_FOR_SEQUENTIAL_READING
 - FILE_MODE_OPEN_TO_APPEND
 - FILE_MODE_OPEN_TO_REWRITE_FROM_START
- **FileHandle:** A integer variable that contains the File Handle of the opened file. This handle is used with the [sc.ReadFile](#) and [sc.WriteFile](#) functions.

Also refer to [sc.OpenFile](#), [sc.CloseFile](#), [sc.ReadFile](#), [sc.WriteFile](#), [sc.GetLastFileErrorCode](#).

sc.OrderQuantityToString()

Type: Function

```
void OrderQuantityToString(const t_OrderQuantity32_64 Value, SCString& OutputString);
```

Parameters

- **Value:** .
- **OutputString:** .

Example

sc.Oscillator()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Oscillator**(SCFloatArrayRef **FloatArrayIn1**, SCFloatArrayRef **FloatArrayIn2**, SCFloatArrayRef **FloatArrayOut**, int **Index**);

SCFloatArrayRef **Oscillator**(SCFloatArrayRef **FloatArrayIn1**, SCFloatArrayRef **FloatArrayIn2**, SCFloatArrayRef **FloatArrayOut**); [Auto-looping only](#).

The **sc.Oscillator()** function calculates the difference between FloatArrayIn1 and FloatArrayIn2.

Parameters

- [FloatArrayIn1](#).
- [FloatArrayIn2](#).
- [FloatArrayOut](#).
- [Index](#).

Example

```
//Calculate the Oscillator between the first two extra arrays in sc.Subgraph[0]
// and output the results to sc.Subgraph[0].Data.
sc.Oscillator(sc.Subgraph[0].Arrays[0], sc.Subgraph[0].Arrays[1], sc.Subgraph[0])

//Access the study value at the current index
float Oscillator = sc.Subgraph[0][sc.Index];
```

sc.Parabolic()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Parabolic**(SCBaseDataRef **BaseDataIn**, SCDateTimeArray **BaseDateTimeln**, SCSubgraphRef **SubGraphOut**, int **Index**, float **InStartAccelFactor**, float **InAccelIncrement**, float **InMaxAccelFactor**, unsigned int **InAdjustForGap**);

SCFloatArrayRef **Parabolic**(SCBaseDataRef **BaseDataIn**, SCDateTimeArray **BaseDateTimeln**, SCSubgraphRef **SubGraphOut**, float **InStartAccelFactor**, float **InAccelIncrement**, float **InMaxAccelFactor**, unsigned int **InAdjustForGap**); [Auto-looping only](#).

The **sc.Parabolic()** function calculates the standard parabolic study.

Parameters

- [BaseDataIn](#).
- [BaseDateTimeIn](#).
- [SubgraphOut](#). For this function, `sc.Subgraph[]`.Arrays[0-4] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- **InStartAccelFactor**: The starting acceleration factor. For reference, refer to the default value with the Parabolic study in Sierra Chart.
- **InAccelIncrement**: The acceleration increment. For reference, refer to the default value with the Parabolic study in Sierra Chart.
- **InMaxAccelFactor**: The maximum acceleration factor. For reference, refer to the default value with the Parabolic study in Sierra Chart.
- **InAdjustForGap**: Set this to 1 to adjust the parabolic for price gaps. Otherwise, set it to 0.

Example

```
SCSubgraphRef Parabolic = sc.Subgraph[0];

sc.Parabolic(
    sc.BaseDataIn,
    sc.BaseDateTimeIn,

    Parabolic,
    sc.Index,
    StartAccelerationFactor.GetFloat(),
    AccelerationIncrement.GetFloat(),

    MaxAccelerationFactor.GetFloat(),
    AdjustForGap.GetYesNo(),
    InputDataHigh.GetInputDataIndex(),

    InputDataLow.GetInputDataIndex()
);

float SAR = Parabolic[sc.Index]; //Access the study value at the current index
```

sc.PauseChartReplay()

Type: Function

```
int PauseChartReplay(int ChartNumber);
```

The **sc.PauseChartReplay** function pauses a chart replay for the chart specified by the **ChartNumber** parameter. This function can only pause the replay for the single specified chart.

It is only possible to specify Chart Numbers that are within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay is paused after the study function returns.

Parameters

- **ChartNumber**: The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To pause a chart replay for the same chart the study instance is on, use **sc.ChartNumber** for this parameter.

Example

```
int Result = sc.PauseChartReplay(sc.ChartNumber);
```

sc.PlaySound()

sc.AlertWithMessage()

Type: Function

```
int PlaySound(int AlertNumber);
```

```
int PlaySound(int AlertNumber, const SCString &AlertMessage, int ShowAlertLog = 0)
```

```
int PlaySound(SCString &AlertPathAndFileName, int NumberTimesPlayAlert = 1);
```

```
int PlaySound(const char *AlertPathAndFileName, int NumberTimesPlayAlert = 1);
```

```
int PlaySound(SCString &AlertPathAndFileName, int NumberTimesPlayAlert, const  
SCString &AlertMessage, int ShowAlertLog = 0)
```

```
int AlertWithMessage(int AlertNumber, const SCString& AlertMessage, int  
ShowAlertLog = 0)
```

```
int AlertWithMessage(int AlertNumber, const char* AlertMessage, int ShowAlertLog = 0)
```

To play an alert sound when a condition is TRUE, it is recommended to use the [sc.SetAlert](#) function, instead of the **sc.PlaySound/sc.AlertWithMessage** function, since it provides a more controlled logic for providing alerts.

The **sc.PlaySound/sc.AlertWithMessage** function is used to play the alert sound associated with the **AlertNumber** parameter or the file specified by the **AlertPathAndFileName** parameter.

A sound will be played every time this function is called. There is no restriction logic used as is the case with the [sc.SetAlert\(\)](#) function.

The alerts sounds are queued up and played asynchronously. This function returns 1 on success, and 0 on failure.

Refer to the **scsf_LogAndAlertExample()** function in the **/ACS_Source/studies.cpp** file in the Sierra Chart installation folder for example code to work with this function.

An Alert Message is added to the **Alerts Log** when this function is called. To open the Alerts Log, select **Window >> Alert Manager >> Alerts Log**.

Parameters

- **AlertNumber**: The **AlertNumber** parameter specifies a particular alert sound to play which is associated with this number. These Alert Numbers are configured through the **Global Settings >> General Settings** window. For complete documentation, refer to [Alert Sound Settings](#). Specify one of the numbers which is supported in the Alert Sound Settings.
- **AlertMessage**: The alert message to add to the **Alerts Log**. **AlertMessage** can either be a **SCString** type or a plain C++ string. For information about **AlertMessage** and how to set this parameter, refer to the [sc.AddAlertLine\(\)](#) function.
- **ShowAlertLog**: **ShowAlertLog** needs to be set to **1** to cause the **Alerts Log** to open up when a message is added. Otherwise, **ShowAlertLog** needs to be **0** or it can be left out.
- **AlertPathAndFileName**: The complete path and filename text string for the **wav** sound file to play.
- **NumberTimesPlayAlert**: The number of times to play the specified Alert Number or specified sound file.

Example

```
sc.PlaySound(45); // Plays Alert Sound Number 45

sc.PlaySound(1,"My Alert Message");

sc.PlaySound(1,"My Alert Message that opens the Alerts Log",1);

sc.PlaySound("C:\\WavFiles\\MyAlert.wav",1,"My Alert Message");
```

sc.PriceValueToTicks()

Type: Function

```
unsigned int PriceValueToTicks(float PriceValue);
```

sc.PriceVolumeTrend()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef PriceVolumeTrend(SCBaseDataRef BaseDataIn, SCFloatArrayRef
FloatArrayOut, int Index);
```

```
SCFloatArrayRef PriceVolumeTrend(SCBaseDataRef BaseDataIn, SCFloatArrayRef
```

FloatArrayOut); [Auto-looping only](#).

The **sc.PriceVolumeTrend()** function calculates the Price Volume Trend study.

Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- [Index](#).

Example

```
sc.PriceVolumeTrend(sc.BaseDataIn, sc.Subgraph[0]);  
  
float PriceVolumeTrends = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.RandomWalkIndicator()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **RandomWalkIndicator**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **RandomWalkIndicator**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.RandomWalkIndicator()** function calculates the Random Walk Indicator study.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

Example

```

sc.RandomWalkIndicator(sc.BaseDataIn, sc.Subgraph[0], 10);

//Access the individual lines
float High = sc.Subgraph[0][sc.Index];

float Low = sc.Subgraph[0].Arrays[0][sc.Index];

//Copy Low to a visible Subgraph
sc.Subgraph[1][sc.Index] = Low;

```

sc.ReadFile()

Type: Function

```

int ReadFile(const int FileHandle, char* Buffer, const int BytesToRead, unsigned int*
p_BytesRead);

```

The **sc.ReadFile()** function reads a file opened with [sc.OpenFile\(\)](#) using the **FileHandle**. It reads the number of **BytesToRead** and stores them in **Buffer**. The actual number of bytes read is then stored in **p_BytesRead**, which could be less than the number of bytes requested, if for instance, it reaches the end of the file before reading in the requested number of bytes.

The function returns **0** if there is an error reading the bytes from the file (this does not include hitting the End of File). Otherwise, the function returns **1**.

Parameters

- **FileHandle**: The File Handle to read the data from. This file handle is returned by the [sc.OpenFile\(\)](#) function call.
- **Buffer**: The pointer to the buffer where the data read will be placed into. This buffer needs to be allocated ahead of calling this function. It can be allocated either on the stack or heap.
- **BytesToRead**: The size of the **Buffer** in bytes.
- **p_BytesRead**: A pointer to the variable that stores the actual number of bytes read from the file. This will not exceed **BytesToRead**.

Also refer to [sc.OpenFile](#), [sc.CloseFile](#), [sc.ReadFile](#), [sc.WriteFile](#), [sc.GetLastFileErrorCode](#).

sc.ReadIntradayFileRecordAtIndex()

Type: Function

```

int ReadIntradayFileRecordAtIndex(int64_t Index, s_IntradayRecord& r_Record,
IntradayFileLockActionEnum IntradayFileLockAction);

```

The **sc.ReadIntradayFileRecordAtIndex()** function is for reading the actual Intraday chart data file records in the Intraday chart data file for the chart the study instance is applied to which is calling this function.

For an example to use this function, refer to the function

scsf_ReadFromUnderlyingIntradayFileExample in the **/ACS_Source/Studies2.cpp** file in the folder Sierra Chart is installed to.

When reading records from the Intraday chart data file a lock is necessary. This is specified through the **IntradayFileLockAction** function parameter. The use of the lock must be completely efficient and proper during the reading of Intraday file records within a call into the study function.

The lock will need to be obtained when reading the first record and released after reading the last record during the call to the study function. If only a single record is read, the lock will need to be obtained before reading that record and released after.

If reading multiple records, the first record needs to be read using this lock enumeration: **IFLA_LOCK_READ_HOLD**. When reading additional records but not the last record use: **IFLA_NO_CHANGE**. When reading the last record, use: **IFLA_RELEASE_AFTER_READ**. If there is only a need to read one record, then use: **IFLA_LOCK_READ_RELEASE**.

The following are the values for the **IntradayFileLockAction** parameter:

- **IFLA_LOCK_READ_HOLD = 1**
- **IFLA_NO_CHANGE = 2**
- **IFLA_RELEASE_AFTER_READ = 3**
- **IFLA_LOCK_READ_RELEASE = 4**

sc.ReadIntradayFileRecordForBarIndexAndSubIndex()

Type: Function

```
int ReadIntradayFileRecordForBarIndexAndSubIndex(int BarIndex, int
SubRecordIndex, s_IntradayRecord& r_Record, IntradayFileLockActionEnum
IntradayFileLockAction);
```

The **sc.ReadIntradayFileRecordForBarIndexAndSubIndex()** function is for reading the Intraday chart data file records from the chart data file (**.scid** file extension) for the chart at a particular chart bar index. This will allow you to access all of the individual Intraday chart data records contained within each chart bar.

This function should not be used to read the records in every chart bar at a time since that will cause the user interface of Sierra Chart to freeze for an extended time if there is a lot of tick by tick data loaded into the chart. It should be used for example only on the current day as needed.

For an example to use this function, refer to the function

scsf_ReadChartBarRecordsFromUnderlyingIntradayFileExample in the **/ACS_Source/Studies2.cpp** file.

In the case of tick by tick data, the **s_IntradayRecord& r_Record** record parameter will have the Ask price in the High field and the Bid price in the Low field. The open will be 0.

When reading records from the Intraday chart data file a lock is necessary. This is specified through the **IntradayFileLockAction** function parameter. The use of the lock must be completely efficient and proper during the reading of Intraday file records within a call into the study function. The lock will need to be obtained when reading the first record and released after reading the last record during the call to the study function. If only a single record is read, the lock will need to be locked and released during the reading of that record.

If reading multiple records, the first record needs to be read using this lock enumeration: IFLA_LOCK_READ_HOLD. When reading additional records but not the last record use: IFLA_NO_CHANGE. When reading the last record, use: IFLA_RELEASE_AFTER_READ. If there is only a need to read one record, then use: IFLA_LOCK_READ_RELEASE.

The **sc.ReadIntradayFileRecordForBarIndexAndSubIndex()** function is in no way affected by the [Session Times](#) for the chart containing the ACSIL function which calls this function. If you want to stop reading data outside of the Session Times, that needs to be implemented by you.

sc.RecalculateChart()

Type: Function

```
int RecalculateChart(int ChartNumber);
```

The **sc.RecalculateChart()** function recalculates the chart specified by the **ChartNumber** parameter. The recalculation happens on a delay. It happens the next time the specified chart performs an update.

For more information about the ChartNumber parameter, refer to [sc.ChartNumber](#).

This function is normally used after changing a study Input with functions like [sc.SetChartStudyInputInt\(\)](#).

When using **sc.RecalculateChart**, the study arrays will not be cleared. The study function will be called with sc.Index, assuming auto looping, initially being set to 0. Your study needs to perform calculations that it normally does starting at that bar index. In the case of when opening the Chart Studies window and then pressing OK, in that particular case the study arrays are completely cleared and reallocated and will have a default value of 0 at each element. This clearing of arrays does not happen when sc.RecalculateChart is used.

sc.RecalculateChartImmediate()

Type: Function

```
int RecalculateChart(int ChartNumber);
```

The **sc.RecalculateChartImmediate()** function recalculates the chart specified by the **ChartNumber** parameter. The recalculation happens immediately before the study function returns.

It is not possible to recalculate the chart that the study function instance that is calling this

function is applied to. In this case 0 is returned and nothing happens.

For more information about the ChartNumber parameter, refer to [sc.ChartNumber](#).

This function is normally used after changing a study Input with functions like [sc.SetChartStudyInputInt\(\)](#).

When using **sc.RecalculateChartImmediate**, the study arrays will not be cleared. The study functions will be called with sc.Index, assuming auto looping, initially being set to 0. Your study needs to perform calculations that it normally does starting at that bar index. In the case of when opening the Chart Studies window and then pressing OK, in that particular case the study arrays are completely cleared and reallocated and will have a default value of 0 at each element. This clearing of arrays does not happen when sc.RecalculateChartImmediate is used.

sc.RefreshTradeData()

Type: Function

```
int RefreshTradeData();
```

The **sc.RefreshTradeData()** function performs the very same action as selecting **Trade >> Refresh Trade Data from Service**. Refer to the [Refresh Trade Data From Service](#) documentation for further information.

Normally there is never a reason to use this function. It is generally never recommended to make a call to this function. It can cause a problem where the trading server ignores one or more requests related to the requests made in response to this function call if there are too many of them over a short period of time, potentially could cause an inconsistent state with the orders table in Sierra Chart, or Sierra Chart ignores the request because there is already an outstanding request in progress.

sc.RegionValueToYPixelCoordinate()

Type: Function

```
int RegionValueToYPixelCoordinate (float RegionValue, int ChartRegionNumber);
```

The **sc.RegionValueToYPixelCoordinate** function calculates the Y-axis pixel coordinate for the given **RegionValue** and the zero-based **ChartRegionNumber** which specifies the Chart Region.

RegionValue will be a value within the Scale Range of the specified Chart Region.
ChartRegionNumber is zero-based.

The returned y-coordinate is in relation to the chart window itself.

Example

```
int YPixelCoordinate = sc.RegionValueToYPixelCoordinate(sc.ActiveToolYValue, sc.GraphRegion);
```

sc.RelayDataFeedAvailable()

Type: Function

```
void RelayDataFeedAvailable();
```

sc.RelayDataFeedUnavailable()

Type: Function

```
void RelayDataFeedUnavailable();
```

sc.RelayNewSymbol()

Type: Function

```
int RelayNewSymbol(const SCString& Symbol, int ValueFormat, float TickSize, const  
SCString& ServiceCode);
```

The **sc.RelayNewSymbol** function is used along with the [sc.RelayTradeUpdate](#) function to allow an Advanced Custom Study to be able to generate its own custom trading data which can be independently charted as a normal symbol through an Intraday chart.

The first step is to make a call to the **sc.RelayNewSymbol** function after Sierra_ Chart is connected to the data feed. Use the [sc.ServerConnectionState](#) member variable to determine when connected to the data feed.

The following are the parameters for this function:

- **Symbol:** The custom symbol for the trading data.
- **ValueFormat:** The [ValueFormat](#) for the symbol.
- **TickSize:** The tick size for the symbol. This is a floating-point value.
- **ServiceCode:** The symbol can have a custom identifier associated with it when it is sent through the Sierra Chart DTC Protocol server. This is set into the **Exchange** field of messages. The **ServiceCode** is this identifier.

sc.RelayServerConnected()

Type: Function

```
int RelayServerConnected();
```

sc.RelayTradeUpdate()

Type: Function

```
int RelayTradeUpdate(SCString& Symbol, SCDatetime& DateTime, float TradeValue,  
unsigned int TradeVolume, int WriteRecord);
```

The **sc.RelayTradeUpdate** function is used along with the [sc.RelayNewSymbol](#) function to allow an Advanced Custom Study to be able to generate its own custom trading data which can be independently charted as a normal symbol through an Intraday chart.

Call the **sc.RelayTradeUpdate** when the custom study wants to generate a trade for its own custom calculations and have that recorded in the Intraday data file for its custom symbol.

The following are the parameters for this function:

- **Symbol**: The custom symbol for the trading data.
- **DateTime**: The [SCDateTime](#) value for the trade in UTC.
- **TradeValue**: The value of the trade. This is a floating-point value.
- **TradeVolume**: The volume of the trade. This is an integer.
- **WriteRecord**: A flag indicating whether data should be written to the Intraday data file or not. Use 1 to cause the data to be written. Use 0 to cause the data not to be written.

sc.RemoveACSChartShortcutMenuItem()

Refer to the [sc.RemoveACSChartShortcutMenuItem\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

sc.RemoveStudyFromChart()

Type: Function

```
void RemoveStudyFromChart(const int ChartNumber, const int StudyID);
```

The **sc.RemoveStudyFromChart** removes a study from the chart. The study is identified by the [ChartNumber](#) parameter and the StudyID parameter.

Parameters

- **ChartNumber**: The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To resume a chart replay for the same chart the study instance is on, use **sc.ChartNumber** for this parameter.
- **StudyID**: .

The study is removed after a delay. Not immediately.

sc.ResizeArrays()

Type: Function

```
int ResizeArrays(int NewSize);
```

sc.ResizeArrays() resizes all of the [sc.Subgraph\[\].Data\[\]](#) arrays to the size specified with the **NewSize** parameter. This function is only useful when you have set [sc.IsCustomChart](#) to 1 (TRUE). The function returns 0 if it fails to resize all the arrays. This function also affects the [sc.DateTimeOut\[\]](#) and [sc.Subgraph\[\].DataColor\[\]](#) arrays.

Example

```
sc.ResizeArrays(100); // Resize the arrays to 100 elements
```

sc.ResumeChartReplay()

Type: Function

int ResumeChartReplay(int ChartNumber);

The **sc.ResumeChartReplay** function resumes a chart replay for the chart specified by the **ChartNumber** parameter.

It is only possible to specify Chart Numbers that are within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay is resumed after the study function returns.

Parameters

- **ChartNumber:** The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To resume a chart replay for the same chart the study instance is on, use **sc.ChartNumber** for this parameter.

Example

```
int Result = sc.ResumeChartReplay(sc.ChartNumber);
```

sc.RGBInterpolate()

Type: Function

COLORREF RGBInterpolate(const COLORREF& Color1, const COLORREF& Color2, float ColorDistance);

The **sc.RGBInterpolate()** function returns the color at the RGB distance between **Color1** and **Color2**, where **ColorDistance** is a value between 0 and 1. If **ColorDistance** is 0, then **Color1** is returned. If **ColorDistance** is 1, then **Color2** is returned. If **ColorDistance** is 0.5f, then the color half way between **Color1** and **Color2** is returned.

Example

```
COLORREF NewColor = sc.RGBInterpolate(RGB(255,0,0), RGB(0,255,0), 0.5);
```

sc.Round()

Type: Function

int **Round**(float **Number**)

The **sc.Round()** function rounds the given floating-point **Number** to the nearest integer.

Example

```
int Rounded = sc.Round(1.2);  
  
//1.2 rounded will be equal to 1
```

sc.RoundToTickSize() / sc.RoundToIncrement()

Type: Function

float **RoundToTickSize**(float **Number**, float **Increment**)

The **sc.RoundToTickSize()** and **sc.RoundToIncrement()** functions round the given floating-point **Number** to the nearest **Increment**. These functions are the same, they just have different names.

Example

```
int Rounded = sc.RoundToTickSize(1.2,0.25);  
  
//1.2 rounded to the increment of .25 will equal 1.25
```

sc.RSI()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **RSI**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, unsigned int **MovingAverageType**, int **Length**)

SCFloatArrayRef **RSI**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, unsigned int **MovingAverageType**, int **Length**) [Auto-looping only](#).

The **sc.RSI()** function calculates the Wilders Relative Strength Index study.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, `sc.Subgraph[]`.Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).
- [MovingAverageType](#).

Example

```
sc.RSI(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], MOVAVGTYPE_SIMPLE, 20);  
float RSI = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.SaveChartbook()

Type: Function

```
int SaveChartbook();
```

The **sc.SaveChartbook()** function will save the Chartbook containing the chart the study function instance is applied to.

This saving occurs immediately when this function is called and will be complete when it returns. So it occurs synchronously.

sc.SaveChartImageToFileExtended()

Type: Function

```
void SaveChartImageToFileExtended(int ChartNumber, SCString&  
OutputPathAndFileName, int Width, int Height, int IncludeOverlays);
```

The **sc.SaveChartImageToFileExtended()** function saves the chart specified by the **ChartNumber** parameter to the file specified by the **OutputPathAndFileName** parameter. The saving of the chart occurs on a delay and happens after the study function returns.

If this function is called multiple times during a call into a study function, only the most recent call will be processed. The other calls to this function will be disregarded.

A chart image can be taken with this function, when it is hidden (**Window >> Hide Window**). Although the **IncludeOverlays** parameter must be set to 0 in this case. A chart image cannot be taken when it is destroyed. Refer to the setting [DestroyChartWindowsWhenHidden](#).

Parameters

- **ChartNumber:** The Chart Number of the chart to save. The chart must be within the same Chartbook as the chart containing study instance this function is called from.
- **OutputPathAndFileName:** The path and filename to save the chart image to. The format of the image is PNG so the file extension should be PNG.
- **Width:** The width of the chart image in pixels. The chart will be resized if this is set to a nonzero value. Set this to 0 to not use this parameter.
- **Height:** The height of the chart image in pixels. The chart will be resized if this is set to a nonzero value. Set this to 0 to not use this parameter.
- **IncludeOverlays:** When this is set to 1 or a nonzero value, then any other windows which overlay the chart will also be included in the chart image.

Also refer to [sc.SaveChartImageToFile](#) and [sc.UploadChartImage\(\)](#).

Example

```
sc.SaveChartImageToFileExtended( sc.ChartNumber, "ImageFilename.PNG", 0, 0, 0);
```

sc.SecondsSinceStartTime()

Type: Function

```
int SecondsSinceStartTime(const SCDatetime& BarDateTime);
```

The **sc.SecondsSinceStartTime()** function returns the number of seconds from the beginning of the trading session as defined by the [Start Time](#) until the given **BarDateTime** parameter.

sc.SecurityType

Type: Function.

The **sc.SecurityType()** function returns the security type for the symbol of the chart the study is applied to. The return type is `n_ACSIL::DTCSecurityTypeEnum`.

It returns one of the following values. Currently it is only set for some Data/Trading services.

- `n_ACSIL::SECURITY_TYPE_UNSET = 0`
- `n_ACSIL::SECURITY_TYPE_FUTURES = 1`
- `n_ACSIL::SECURITY_TYPE_STOCK = 2`
- `n_ACSIL::SECURITY_TYPE_FOREX = 3`
- `n_ACSIL::SECURITY_TYPE_INDEX = 4`
- `n_ACSIL::SECURITY_TYPE_FUTURES_STRATEGY = 5`
- `n_ACSIL::SECURITY_TYPE_FUTURES_OPTION = 7`

- n_ACSIL::SECURITY_TYPE_STOCK_OPTION = 6
- n_ACSIL::SECURITY_TYPE_INDEX_OPTION = 8
- n_ACSIL::SECURITY_TYPE_BOND = 9
- n_ACSIL::SECURITY_TYPE_MUTUAL_FUND = 10

sc.SendEmailMessage

Type: Function

```
int SendEmailMessage(const SCString& ToEmailAddress, const SCString& Subject,  
const SCString& MessageBody);
```

The **sc.SendEmailMessage** function will send an email message to the **ToEmailAddress** with the given **Subject** and **MessageBody**.

There are sending limits which will be applied over a period of time. So you cannot always expect an email message to be sent when using this function.

Parameters

- **ToEmailAddress:** The email address to which the email is to be sent.
- **Subject:** The subject of the email to be sent.
- **MessageBody:** The body of the email message to be sent.

sc.SessionStartTime()

Type: Function

```
int SessionStartTime();
```

The **sc.SessionStartTime()** function gets the start of the trading day in Intraday charts which is based on the Session Times set in **Chart >> Chart Settings**. Normally this is the **Session Times >>Start Time**.

If **Use Evening Session** is enabled in the Chart Settings, then this function returns the Start Time of the evening session.

The value returned is a [Time Value](#).

Example

```
int StartTime = sc.SessionStartTime();
```

sc.SetACSChartShortcutMenuItemChecked()

Refer to the [sc.SetACSChartShortcutMenuItemChecked\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar](#)

[Buttons, Pointer Events](#) documentation.

sc.SetACSChartShortcutMenuItemDisplayed()

Refer to the [sc.SetACSChartShortcutMenuItemDisplayed\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

sc.SetACSChartShortcutMenuItemEnabled()

Refer to the [sc.SetACSChartShortcutMenuItemEnabled\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

sc.SetACSToolButtonText()

Refer to the [sc.SetACSToolButtonText\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

sc.SetACSToolEnable()

Refer to the [sc.SetACSToolEnable\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

sc.SetACSToolToolTip()

Refer to the [sc.SetACSToolToolTip\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

sc.SetAlert()

Type: Function

```
int SetAlert(int AlertNumber, int Index, SCString Message);
```

```
int SetAlert(int AlertNumber, int Index);
```

```
int SetAlert(int AlertNumber, SCString Message); Auto-looping only.
```

```
int SetAlert(int AlertNumber); Auto-looping only.
```

The **sc.SetAlert()** function will play an alert sound and add a message to the **Window >> Alert Manager >> Log**.

This can be used when you have a condition that you want to provide a signal for at the specified **sc.Subgraph[].Data** array **Index**. **Index** is a parameter to the function specifying the array index the alert is associated with.

The versions of the functions that do not use this parameter, internally set it to **sc.Index** when using Automatic Looping.

To open the Alerts Log, select **Window >> Alert Manager >> Log** on the menu.

Since the **sc.Subgraph[].Data[]** arrays directly correspond to the **sc.BaseData[][]** arrays, **Index** corresponds to the **sc.BaseData[][]** arrays indexes. If you are using [Automatic Looping](#), then the **Index** parameter is automatically set for you and does not need to be specified.

The **AlertNumber** parameter sets an Alert Sound Number to play. This can be anywhere from 1 to 150. To configure the specific sound file to play for a particular Alert Number, select **Global Settings >> General Settings >> Alerts** on the Sierra Chart menu. For more information, refer to the [Alert Sound Settings](#).

Using an **AlertNumber** of 0 means there will be no Alert Sound played.

The optional **Message** parameter specifies the text message to be added to the **Alerts Log**. If it is not specified, a standard message will be added to the log.

If you set an Alert on an **Index** that is not at the end of a **sc.Subgraph[].Data** array, it will be ignored. However, if there are multiple chart bars added during an update, the alerts will be processed for those new bars as well as the prior bar. Therefore, if you are providing an alert on a bar which has just closed, it will be processed.

If historical data is being downloaded in the chart, then calls to **sc.SetAlert()** are ignored. This is to prevent Alert sounds and Messages from occurring when historical data is being downloaded.

During a call into a study function, if there are multiple calls to **sc.SetAlert()** at the same or different bar indexes, only one will actually generate an alert. The rest will be ignored. However, if [sc.ResetAlertOnNewBar](#) is set to a nonzero value, then calling **sc.SetAlert()** at different bar indexes will generate separate alerts.

If at every call into the study function you make a call to **sc.SetAlert()**, only the first time **sc.SetAlert()** is called will that generate an alert. There must be at least one call into the study function where **sc.SetAlert()** is not called, to reset the internal alert state for the study to allow for another call to **sc.SetAlert()** to generate an alert. However, if [sc.ResetAlertOnNewBar](#) is set to a nonzero value, then calling **sc.SetAlert()** at a new bar index will generate a new alert.

The [sc.AlertOnlyOncePerBar](#) and the [sc.ResetAlertOnNewBar](#) variables work with **sc.SetAlert()**. They control the alert processing. These variables only need to be set once in the **sc.SetDefaults** code block.

Even if [sc.AlertOnlyOncePerBar](#) is set to 0, then multiple calls to **sc.SetAlert()** during a call into a study function at the same or different bar indexes, will still only generate one alert, as explained above.

For an example to work with the **sc.SetAlert()** function, see **sclf_SimpMovAvg()** in

studies.cpp inside the ACS_Source folder in the Sierra Chart installation folder.

When the alert by the **sc.SetAlert()** function is processed, and a message is added to the Alerts Log, the Alerts Log can be automatically opened by enabling the **Global Settings >> General Settings >> Alerts >> Additional Settings >> Show Alerts Log on Study Alert** option.

Example

```
sc.SetAlert(5, "Condition #1 is TRUE.");
```

The [Study Summary Window](#) will use the **Study Summary: Alert True Background** color set through the **Graphics Settings** window for the displayed study Subgraphs for a study included in the Study Summary window, when the Alert Condition (alert state) is true for a study.

An ACSIL study can set the alert state for itself to true for the last bar by using the following code. This assumes [Automatic Looping](#) is being used.

Example

```
if (sc.Index == sc.ArraySize - 1)
    sc.SetAlert(1);
```

sc.SetAttachedOrders()

Type: Function

```
void SetAttachedOrders(const s_SCNewOrder& AttachedOrdersConfiguration);
```

Refer to the [sc.SetAttachedOrders\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.SetBarPeriodParameters()

Type: Function

```
void SetBarPeriodParameters(const n_ACSIL::s_BarPeriod& BarPeriod);
```

The **sc.SetBarPeriodParameters()** function sets the chart bar period parameters according to the structure of type **n_ACSIL::s_BarPeriod** which is passed to the function as **BarPeriod**.

After calling the **sc.SetBarPeriodParameters()** function, the changes go into effect after returning from your ACSIL study function. The necessary chart reload will occur before the next call into the study function.

The **n_ACSIL::s_BarPeriod** structure members are the following:

- **s_BarPeriod::ChartDataTypes**: Can be one of the following:

- DAILY_DATA = 1
- INTRADAY_DATA = 2

If this is not set, the chart will retain the current setting.

- **s_BarPeriod::HistoricalChartBarPeriodType**: Can be one of the following:

- HISTORICAL_CHART_PERIOD_DAYS = 1
- HISTORICAL_CHART_PERIOD_WEEKLY = 2
- HISTORICAL_CHART_PERIOD_MONTHLY = 3
- HISTORICAL_CHART_PERIOD_QUARTERLY = 4
- HISTORICAL_CHART_PERIOD_YEARLY = 5

- **s_BarPeriod::HistoricalChartDaysPerBar**: When **s_BarPeriod::HistoricalChartBarPeriodType** is set to HISTORICAL_CHART_PERIOD_DAYS, then this specifies the number of days per historical chart bar.

- **s_BarPeriod::IntradayChartBarPeriodType**: The type of bar period to be used in the case of an Intraday chart. To determine the chart data type, use **s_BarPeriod::ChartDataTypes**. For example, set to the enumeration value IBPT_DAYS_MINS_SECS for an Intraday Chart **Bar Period Type** of **Days-Minutes-Seconds**. Can be any of the following constants:

- IBPT_DAYS_MINS_SECS = 0:

s_BarPeriod::IntradayChartBarPeriodParameter1 is the number of seconds in one bar in an Intraday chart. This is set by the **Days-Mins-Secs** setting in the **Chart >> Chart Settings** window for the chart. For example, for a 1 Minute chart this will be set to 60. For a 30 Minute chart this will be set to 1800.

- IBPT_VOLUME_PER_BAR = 1
- IBPT_NUM_TRADES_PER_BAR = 2
- IBPT_RANGE_IN_TICKS_STANDARD = 3
- IBPT_RANGE_IN_TICKS_NEWBAR_ON_RANGEMET = 4
- IBPT_RANGE_IN_TICKS_TRUE = 5
- IBPT_RANGE_IN_TICKS_FILL_GAPS = 6
- IBPT_REVERSAL_IN_TICKS = 7
- IBPT_RENKO_IN_TICKS = 8
- IBPT_DELTA_VOLUME_PER_BAR = 9
- IBPT_FLEX_RENKO_IN_TICKS = 10
- IBPT_RANGE_IN_TICKS_OPEN_EQUAL_CLOSE = 11
- IBPT_PRICE_CHANGES_PER_BAR = 12
- IBPT_MONTHS_PER_BAR = 13
- IBPT_POINT_AND_FIGURE = 14
- IBPT_FLEX_RENKO_INV = 15
- IBPT_ALIGNED_RENKO = 16
- IBPT_RANGE_IN_TICKS_NEW_BAR_ON_RANGE_MET_OPEN_EQUALS_PR = 17

- **IBPT_ACSIL_CUSTOM = 18**: This is used when the chart contains an advanced custom study that creates custom chart bars. The study name is contained within `s_BarPeriod::ACSILCustomChartStudyName`.
- **s_BarPeriod::IntradayChartBarPeriodParameter1**: The first parameter for the bar period to be used. In the case of `IntradayChartBarPeriodType` being set to `IBPT_DAYS_MINS_SECS`, this will be set to the number of seconds. In the case of `IntradayChartBarPeriodType` being set to `IBPT_FLEX_RENKO_IN_TICKS`, this would be the **Bar Size** value in ticks.
- **s_BarPeriod::IntradayChartBarPeriodParameter2**: The second parameter for the bar period that to be used. For example, this would be the **Trend Offset** value when using `IBPT_FLEX_RENKO_IN_TICKS` for the Intraday Chart Bar Period Type. If this parameter is unused for the Intraday Chart Bar Period Type, then either do not set this value or set it to **0**.
- **s_BarPeriod::IntradayChartBarPeriodParameter3**: The third parameter for the bar period that to be used. For example, this would be the **Reversal Offset** value when using `IBPT_FLEX_RENKO_IN_TICKS` for the Intraday Chart Bar Period Type. If this parameter is unused for the Intraday Chart Bar Period Type, then either do not set this value or set it to **0**.
- **s_BarPeriod::IntradayChartBarPeriodParameter4**: The fourth parameter for the bar period that is being used. For example, this would be the **Renko New Bar Mode** setting when using any of the **Renko** Intraday Chart Bar Period Types. If this parameter is unused for the Intraday Chart Bar Period Type, then either do not set this value or set it to **0**.
- **s_BarPeriod::ACSILCustomChartStudyName**: Not relevant when setting bar period parameters.

Example

```
n_ACSIL::s_BarPeriod NewBarPeriod;
NewBarPeriod.ChartDataType = INTRADAY_DATA;
NewBarPeriod.IntradayChartBarPeriodType = IBPT_DAYS_MINS_SECS;
NewBarPeriod.IntradayChartBarPeriodParameter1 = 300; // 300 seconds per bar which is 5 minutes

//Set the bar period parameters. This will go into effect after the study function returns.
sc.SetBarPeriodParameters(NewBarPeriod);
```

sc.SetChartStudyInputChartStudySubgraphValues()

Type: Function

```
int SetChartStudyInputChartStudySubgraphValues(int ChartNumber, int StudyID, int
InputIndex, s_ChartStudySubgraphValues ChartStudySubgraphValues);
```

The `sc.SetChartStudyInputChartStudySubgraphValues()` function .

Parameters

- **ChartNumber:** .
- **StudyID:** .
- **InputIndex:** .
- **ChartStudySubgraphValues:** .

Example

sc.SetChartWindowState

Type: Function

```
int SetChartWindowState(int ChartNumber, int WindowState);
```

The **sc.SetChartWindowState** function is used to minimize, restore or maximize the chart specified by the **ChartNumber** parameter in the same Chartbook containing the chart the study instance is applied to.

The following are the possible values for **WindowState**:

- CWS_MINIMIZE = 1
- CWS_RESTORE = 2
- CWS_MAXIMIZE = 3

The function returns 1 on success and 0 if the Chart Number is not found.

sc.SetCombineTradesIntoOriginalSummaryTradeSetting

Type: Function

```
void sc.SetCombineTradesIntoOriginalSummaryTradeSetting(int NewState);
```

The **sc.SetCombineTradesIntoOriginalSummaryTradeSetting**

Parameters

- **NewState:**

sc.SetCustomStudyControlBarButtonColor

Type: Function

```
void SetCustomStudyControlBarButtonColor(int ControlBarButtonNum, const COLORREF Color);
```

The **sc.SetCustomStudyControlBarButtonColor** function sets the background color of the specified Advanced Custom study Control Bar button. For further details about Advanced Custom study Control Bar buttons, refer to [Advanced Custom Study Buttons and Pointer Events](#).

The change of the color goes into effect immediately.

Parameters

- **ControlBarButtonNum**: The integer number of the Advanced Custom study Control Bar button that is to be modified. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).
- **Color**: The color in [RGB Format](#) for the Control Bar button background.

sc.SetCustomStudyControlBarButtonEnable

Type: Function

```
void SetCustomStudyControlBarButtonEnable(int ControlBarButtonNum, int Enabled);
```

Parameters

- **ControlBarButtonNum**: The integer number of the Advanced Custom study Control Bar button that is to be modified. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).
- **Enabled**: .

sc.SetCustomStudyControlBarButtonHoverText

Type: Function

```
void SetCustomStudyControlBarButtonHoverText(int ControlBarButtonNum, const char* HoverText);
```

Parameters

- **ControlBarButtonNum**: The integer number of the Advanced Custom study Control Bar button that is to be modified. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).
- **HoverText**: .

sc.SetCustomStudyControlBarButtonShortCaption

Type: Function

```
int SetCustomStudyControlBarButtonShortCaption(int ControlBarButtonNum, const
```

```
SCString& ButtonText);
```

The **sc.SetCustomStudyControlBarButtonShortCaption** function .

Parameters

- **ControlBarButtonNum**: The integer number of the Advanced Custom study Control Bar button that is to be modified. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).
- **ButtonText**: .

sc.SetCustomStudyControlBarButtonText

Type: Function

```
void SetCustomStudyControlBarButtonText(int ControlBarButtonNum, const char*  
ButtonText);
```

Parameters

- **ControlBarButtonNum**: The integer number of the Advanced Custom study Control Bar button that is to be modified. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).
- **ButtonText**: .

sc.SetGraphicsSetting

Type: Function

```
int32_t SetGraphicsSetting(const int32_t ChartNumber, const  
n_ACSIL::GraphicsSettingsEnum GraphicsSetting, uint32_t Color, uint32_t LineWidth,  
SubgraphLineStyles LineStyle);
```

The **sc.SetGraphicsSetting** function .

Parameters

- **ChartNumber**: .
- **GraphicsSetting**: .
- **Color**: .
- **LineWidth**: .
- **LineStyle**: .

sc.SetHorizontalGridState

Type: Function

```
int SetHorizontalGridState(int GridIndex, int State);
```

The **sc.SetHorizontalGridState** function enables or disables the Horizontal Grid for a given [Chart Region](#) or for all Chart Regions. The function returns the state of the Horizontal Grid for the given Chart Region prior to the changes.

Parameters

- **GridIndex**: The one based Chart Region number for the Horizontal Grid. If a value of **0** is passed, then all Chart Regions will be affected.
- **State**: A value of **0** disables the Horizontal Grid for the specified Chart Region. A nonzero value will enable the Horizontal Grid for the specified Chart Region.

sc.SetNumericInformationDisplayOrderFromString

Type: Function

```
void SetNumericInformationDisplayOrderFromString(const SCString&  
CommaSeparatedDisplayOrder);
```

For complete documentation for this function, refer to [Numeric Information Table Graph Draw Type](#).

sc.SetNumericInformationGraphDrawTypeConfig()

Type: Function

```
void SetNumericInformationGraphDrawTypeConfig(const  
s_NumericInformationGraphDrawTypeConfig&  
NumericInformationGraphDrawTypeConfig);
```

For complete documentation for this function, refer to [Numeric Information Table Graph Draw Type](#).

sc.SetSheetCellAsDouble()

Type: Function

```
int SetSheetCellAsDouble(void* SheetHandle, const int Column, const int Row, const  
double CellValue);
```

The **sc.SetSheetCellAsDouble()** function places the value of the double variable **CellValue** into the specified Spreadsheet Sheet Cell.

If the function is unable to determine the Spreadsheet Sheet Cell then it returns a value of **0**. Otherwise it returns a value of **1**.

Parameters

- **SheetHandle**: The handle of the Spreadsheet Sheet as returned by the [sc.GetSpreadsheetSheetHandleByName\(\)](#) function
- **Column**: The column number for the Sheet Cell to set the value in. This is a zero-based array, so column B in the Sheet would be a value of 1.
- **Row**: The row number for the Sheet Cell to set the value in. This is a zero-based array, so row 2 in the Sheet would be a value of 1.
- **CellValue**: The double value that is entered into the Sheet Cell.

Also refer to the following functions: [sc.GetSheetCellAsDouble\(\)](#), [sc.GetSheetCellAsString\(\)](#), [sc.SetSheetCellAsString\(\)](#), [sc.GetSpreadsheetSheetHandleByName\(\)](#).

Example

```
const char* SheetCollectionName = "ACSILInteractionExample";  
  
const char* SheetName = "Sheet1";  
  
void* SheetHandle = sc.GetSpreadsheetSheetHandleByName(SheetCollectionName, SheetName, fa  
  
// Set values in column B, row 2.  
sc.SetSheetCellAsDouble(SheetHandle, 1, 1, sc.BaseData[SC_HIGH][sc.Index]);
```

sc.SetSheetCellAsString()

Type: Function

```
int SetSheetCellAsString(void* SheetHandle, const int Column, const int Row, const  
SCString& CellString);
```

The **sc.SetSheetCellAsString()** function places the value of the SCString variable **CellString** into the specified Spreadsheet Sheet Cell.

If the function is unable to determine the Spreadsheet Sheet Cell, then it returns a value of 0. Otherwise it returns a value of 1.

Parameters

- **SheetHandle**: The handle of the Spreadsheet Sheet as returned by the [sc.GetSpreadsheetSheetHandleByName\(\)](#) function
- **Column**: The column number for the Sheet Cell to set the value in. This is a zero-based array, so column B in the Sheet would be a value of 1.
- **Row**: The row number for the Sheet Cell to set the value in. This is a zero-based array, so row 2 in the Sheet would be a value of 1.
- **CellString**: An SCString value to be entered into the Sheet Cell.

Also refer to the following functions: [sc.GetSheetCellAsDouble\(\)](#), [sc.SetSheetCellAsDouble\(\)](#), [sc.GetSheetCellAsString\(\)](#), [sc.GetSpreadsheetSheetHandleByName\(\)](#).

Example

```
const char* SheetCollectionName = "ACSILInteractionExample";

const char* SheetName = "Sheet1";

void* SheetHandle = sc.GetSpreadsheetSheetHandleByName(SheetCollectionName, SheetName, fa

// Set labels in column A.
sc.SetSheetCellAsString(SheetHandle, 0, 1, "High");
sc.SetSheetCellAsString(SheetHandle, 0, 2, "Low");
sc.SetSheetCellAsString(SheetHandle, 0, 3, "Enter Formula");
sc.SetSheetCellAsString(SheetHandle, 0, 5, "Log Message");
```

sc.SetStudySubgraphColors()

Type: Function

```
int32_t SetStudySubgraphColors(int32_t ChartNumber, int32_t StudyID, int32_t
StudySubgraphNumber, uint32_t PrimaryColor, uint32_t SecondaryColor, uint32_t
SecondaryColorUsed);
```

The **sc.SetStudySubgraphColors()** function sets the primary and secondary colors of a Subgraph in another study in the Chartbook. The study can be in a different chart.

The **ChartNumber**, **StudyID**, and **StudySubgraphNumber** parameters should be obtained using the [sc.Input\[\].GetChartStudySubgraphValues](#) function.

The colors from the other study are set through the **PrimaryColor** and **SecondaryColor** parameters which are references.

The function returns 1 if the study was found. Otherwise, 0 is returned.

Example

```
uint32_t PrimaryColor = RGB(128, 0, 0);
uint32_t SecondaryColor = RGB(0, 255, 0);
uint32_t SecondaryColorUsed = 1;
sc.SetStudySubgraphColors(1, 1, 0, PrimaryColor, SecondaryColor, SecondaryColorUsed);
```

sc.SetStudySubgraphDrawStyle()

Type: Function

```
void SetStudySubgraphDrawStyle(int ChartNumber, int StudyID, int
```

```
StudySubgraphNumber, int DrawStyle);
```

The **sc.SetStudySubgraphDrawStyle()** function sets the Draw Style of a Subgraph in another study in the Chartbook. The study can be in a different chart. The **ChartNumber**, **StudyID**, and **StudySubgraphNumber** parameters should be obtained using the [sc.Input\(\).GetChartStudySubgraphValues](#) function.

Example

```
sc.SetStudySubgraphDrawStyle(1, 1, 0, DRAWSTYLE_LINE);
```

sc.SetStudySubgraphLineStyle()

Type: Function

```
int32_t SetStudySubgraphLineStyle(int32_t ChartNumber, int32_t StudyID, int32_t  
StudySubgraphNumber, SubgraphLineStyles LineStyle);
```

The **sc.SetStudySubgraphLineStyle()** function .

Parameters

- **ChartNumber:** .
- **StudyID:** .
- **StudySubgraphNumber:** .
- **LineStyle:** .

Example

sc.SetStudySubgraphLineWidth()

Type: Function

```
int32_t SetStudySubgraphLineWidth(int32_t ChartNumber, int32_t StudyID, int32_t  
StudySubgraphNumber, int32_t LineWidth);
```

The **sc.SetStudySubgraphLineWidth()** function .

Parameters

- **ChartNumber:** .
- **StudyID:** .

- **StudySubgraphNumber:** .
- **LineWidth:** .

sc.SetStudyVisibilityState

Type: Function

```
int SetStudyVisibilityState(int StudyID, int Visible);
```

The **sc.SetStudyVisibilityState()** function sets the visibility state of a study by either making it visible or hiding it.

Parameters

- **StudyID:** The unique ID of the study to set the visibility state of. For more information, refer to [Unique Study Instance Identifiers](#).
- **Visible:** This needs to be 1 to make the study visible or 0 to make the study hidden.

sc.SetTradeWindowTextTag()

Type: Function

```
void SetTradeWindowTextTag(const SCString& TextTag);
```

The **sc.SetTradeWindowTextTag** function sets the specified **TextTag** to the Trade Window of the Chart that the study instance is applied to. For more information, refer to [Text Tag](#).

sc.SetTradingKeyboardShortcutsEnableState()

Type: Function

```
void SetTradingKeyboardShortcutsEnableState(int State);
```

The **sc.SetTradingKeyboardShortcutsEnableState** function enables or disables the state of **Trade >> Trading Keyboard Shortcuts Enabled**. Refer to [Trading Keyboard Shortcuts Enabled \(Trade menu\)](#).

Setting **State** to 0 will disable. Setting **State** to 1 will enable.

sc.SetTradingLockState()

Type: Function

```
int32_t SetTradingLockState(int32_t NewState);
```

The **sc.SetTradingLockState()** function enables or disables the state of **Trade >> Trading Locked**.

Setting **NewState** to 0 disables the lock. Setting **NewState** to 1 enables the lock.

sc.SetUseGlobalGraphicsSettings

Type: Function

```
int SetUseGlobalGraphicsSettings(const int ChartNumber, int State);
```

The **sc.SetUseGlobalGraphicsSettings()** function .

Parameters

- **ChartNumber:** .
- **State:** A value of **0** turns the Horizontal Grid **Off** for the defined Region. Any other value will turn the Horizontal Grid **On** for the defined Region.

sc.SetVerticalGridState

Type: Function

```
int SetVerticalGridState(int State);
```

The **sc.SetVerticalGridState()** function turns on or off the Vertical Grid. The function returns the state of the Horizontal Grid for the given region prior to the changes.

Parameters

- **GridIndex:** The Region number for the Horizontal Grid. If a value of **0** is passed, then all regions will be affected.
- **State:** A value of **0** turns the Horizontal Grid **Off** for the defined Region. Any other value will turn the Horizontal Grid **On** for the defined Region.

sc.SimpleMovAvg()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef SimpleMovAvg(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef  
FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef SimpleMovAvg(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef  
FloatArrayOut, int Length); Auto-looping only.
```

The **sc.SimpleMovAvg()** function calculates the simple moving average study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

- [Length](#).

Example

```
sc.SimpleMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);

float SimpleMovAvg = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.Slope()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Slope**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**);

SCFloatArrayRef **Slope**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**); [Auto-looping only](#).

The **sc.Slope()** function calculates a simple slope.

This is the difference between the **FloatArrayIn** at the **Index** currently being calculated and the prior **FloatArrayIn** value.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

Example

```
sc.Slope(sc.BaseData[SC_LAST], sc.Subgraph[0]);

//Access the study value at the current index
float SlopeValue = sc.Subgraph[0][sc.Index];
```

sc.SlopeToAngleInDegrees()

Type: Function

double **SlopeToAngleInDegrees**(double **Slope**);

Parameters

- **Slope:**

Example

sc.SmoothedMovingAverage()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **SmoothedMovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **SmoothedMovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.SmoothedMovingAverage()** function calculates the Smoothed Moving Average study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.SmoothedMovingAverage(sc.BaseData[SC_LAST], sc.Subgraph[0], 10);  
float SmoothedMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.StartChartReplay()

Type: Function

int **StartChartReplay**(int **ChartNumber**, float **ReplaySpeed**, const SCDatetime& **StartDateTime**);

The **sc.StartChartReplay** function starts a chart replay for the chart specified by the **ChartNumber** parameter. It is only possible to specify Chart Numbers that are within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay is started after the study function returns.

.

It is not possible to start a replay on a chart which is still in the process of loading data from the chart data file. There will be no error returned but the replay will not start. It is possible to know when all charts are loaded in a Chartbook by using the [sc.IsChartDataLoadingCompleteForAllCharts](#) function.

Parameters

- **ChartNumber**: The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To start a chart replay for the same chart the study instance is on, use **sc.ChartNumber** for this parameter.
- **ReplaySpeed**: The replay speed. A speed of 1 is the same as real time.
- **StartDateTime**: A [SCDateTime](#) variable set to the starting Date-Time to start the replay at.

Example



sc.StartChartReplayNew()

Type: Function

```
int StartChartReplayNew(n_ACSIL::s_ChartReplayParameters& ChartReplayParameters);
```

The **sc.StartChartReplayNew** function starts a chart replay for a chart. The parameters of the replay including the particular Chart Number to start the replay on are specified by the **ChartReplayParameters** data structure parameter.

It is only possible to start a replay for a chart which is within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay is started after the study function returns.

It is not possible to start a replay on a chart which is still in the process of loading data from the chart data file. There will be no error returned but the replay will not start. It is possible to know when all charts are loaded in a Chartbook by using the [sc.IsChartDataLoadingCompleteForAllCharts](#) function.

Parameters

The following are the parameters of the **n_ACSIL::s_ChartReplayParameters** data structure.

- **ChartNumber**: The number of the chart to replay. Refer to [ChartNumber Parameter](#).
- **ReplaySpeed**: The replay speed. A speed of 1 is the same as real time. It is possible to use a fractional value for this.
- **StartDateTime**: A [SCDateTime](#) variable set to the starting Date-Time to start the replay at. This is in the chart's time zone.
- **SkipEmptyPeriods**: When set to 1 this enables the [Skip Empty Periods](#) option during a replay.
- **ReplayMode**: Sets the replay mode for the replay. Can be one of the following

values. The default for this is: `REPLAY_MODE_UNSET`.

- `REPLAY_MODE_UNSET` = 0
- `REPLAY_MODE_STANDARD` = 1
- `REPLAY_MODE_ACCURATE_TRADING_SYSTEM_BACK_TEST` = 2
- `REPLAY_MODE_CALCULATE_AT EVERY TICK` = 3
- `REPLAY_MODE_CALCULATE_SAME_AS_REAL_TIME` = 4
- **ClearExistingTradeSimulationDataForSymbolAndTradeAccount**: Set this to 1, to clear Trade Activity, Orders and Position for the symbol and Trade Account to be replayed. Set this to 0 to not clear this data when starting the replay.

Example

sc.StartScanOfSymbolList()

Type: Function

```
int StartScanOfSymbolList(const int ChartNumber);
```

Parameters

- **ChartNumber**: The number of the chart to start the scan for. Refer to [Chart Number](#).

Example

sc.StartDownloadHistoricalData()

Type: Function

```
int StartDownloadHistoricalData(double StartingDateTime);
```

The **sc.StartDownloadHistoricalData()** function starts a historical data download in the chart. As of version 1887 this function can be used with Historical Daily data charts and Intraday charts.

The **StartingDateTime** is an [SCDateTime](#) type and specifies the starting date-time of the historical data download in UTC. Set this to 0.0 to download from the last date in the chart data file or all available data in the case of an empty chart data file.

When **StartingDateTime** specifies a Date-Time which is earlier than the last Date-Time in the chart, then the historical data request starts at that time and the downloaded data replaces the data already in the chart.

This function only downloads historical data for the symbol of the chart and not prior contracts, in the case of a Continuous Futures Contract chart.

sc.StdDeviation()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef StdDeviation(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef StdDeviation(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.StdDeviation()** function calculates the standard deviation of the data.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.StdDeviation(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);  
  
float StdDeviation = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.StdError()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef StdError(SCFloatArrayRef FloarArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef StdError(SCFloatArrayRef FloarArrayIn, SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.StdError()** function calculates the standard error of the data.

Parameters

- [FloatArrayIn](#).

- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.StdError(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);

float StdError = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.Stochastic()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **Stochastic**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **FastKLength**, int **FastDLength**, int **SlowDLength**, unsigned int **MovingAverageType**);

SCFloatArrayRef **Stochastic**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **FastKLength**, int **FastDLength**, int **SlowDLength**, unsigned int **MovingAverageType**); [Auto-looping only](#).

The **sc.Stochastic()** function calculates the Fast %K, Fast %D, and Slow %D Stochastic lines.

Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-1] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [FastKLength](#).
- [FastDLength](#).
- [SlowDLength](#).
- [MovingAverageType](#).

Example

```

sc.Stochastic(sc.BaseDataIn, sc.Subgraph[0], 10, 3, 3, MOVAVGTYPE_SIMPLE);

//Access the individual study values at the current index
float FastK = sc.Subgraph[0][sc.Index];

float FastD = sc.Subgraph[0].Arrays[0][sc.Index];

float SlowD = sc.Subgraph[0].Arrays[1][sc.Index];

//Copy to Visible Subgraphs
sc.Subgraph[1][sc.Index] = FastD;
sc.Subgraph[2][sc.Index] = SlowD;

```

sc.StopChartReplay()

Type: Function

```
int StopChartReplay(int ChartNumber);
```

The **sc.StopChartReplay** function stops a chart replay for the chart specified by the **ChartNumber** parameter.

It is only possible to specify Chart Numbers that are within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay is stopped after the study function returns.

Parameters

- **ChartNumber:** The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To stop a chart replay for the same chart the study instance is on, use **sc.ChartNumber** for this parameter.

Example

```
int Result = sc.StopChartReplay(sc.ChartNumber);
```

sc.StopScanOfSymbolList()

Type: Function

```
int StopScanOfSymbolList(const int ChartNumber);
```

Parameters

- **ChartNumber:** The number of the chart to stop the scan for. Refer to [Chart Number](#).

Example

sc.StringToDouble()

Type: Function

```
double StringToDouble (const char* NumberString);
```

The **sc.StringToDouble** function returns the number value represented by the character string **NumberString** as a double variable.

sc.SubmitOCOOrder()

Type: Function

```
int sc.SubmitOCOOrder(s_SCNewOrder& NewOrder); Note: For use with Auto-Looping only.
```

Refer to the [sc.SubmitOCOOrder\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

sc.Summation()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef Summation (SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut,  
int Index, int Length);
```

```
SCFloatArrayRef Summation (SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut,  
int Length); Auto-looping only.
```

The **sc.Summation** function calculates the summation over the specified Length.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
Summation(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0], 10);  
  
float Summation = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.SuperSmoother2Pole()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef SuperSmoother2Pole(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef SuperSmoother2Pole(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.SuperSmoother2Pole()** function calculates a smoothing of data and is frequently used in the studies written by John Ehlers.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.SuperSmoother2Pole(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
  
float SuperSmoother2Pole = sc.Subgraph[0][sc.Index]; //Access the function value at the current index
```

sc.SuperSmoother3Pole()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef SuperSmoother3Pole(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef SuperSmoother3Pole(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.SuperSmoother3Pole()** function calculates a smoothing of data and is frequently used in the studies written by John Ehlers.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.SuperSmoother3Pole(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
  
float SuperSmoother3Pole = sc.Subgraph[0][sc.Index]; //Access the function value at the current index
```

sc.T3MovingAverage()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **T3MovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, float **Multiplier**, int **Index**, int **Length**);

SCFloatArrayRef **T3MovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, float **Multiplier**, int **Length**); [Auto-looping only](#).

The **sc.T3MovingAverage()** function calculates the T3 Moving Average study.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-5] (Extra Arrays) are used for internal calculations and additional results output.
- **Multiplier**: The T3 study multiplier value.
- [Index](#).
- [Length](#).

Example

```
sc.T3MovingAverage(sc.BaseDataIn[InputData.GetInputDataIndex()], T3, Multiplier.GetFloat(), Length.GetInt()
```

sc.TEMA()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **TEMA** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **TEMA** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.TEMA()** function calculates the Triple Exponential Moving Average study.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, `sc.Subgraph[]`.Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

Example

```
sc.TEMA(sc.BaseData[SC_LAST], sc.Subgraph[0], 10);

float TEMA = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.TicksToPriceValue()

Type: Function

float TicksToPriceValue(unsigned int **Ticks**);

The **sc.TicksToPriceValue()** function converts the price value from the `s_VolumeAtPriceV2::PriceInTicks` structure member and returns the actual floating-point price value.

Example

```
s_VolumeAtPriceV2 VolumeAtPrice;
sc.GetPointOfControlPriceVolumeForBar(BarIndex, VolumeAtPrice);

if (VolumeAtPrice.PriceInTicks != 0)
    Subgraph_VPOC.Data[BarIndex] = sc.TicksToPriceValue(VolumeAtPrice.PriceInTicks);
```

sc.TimeMSToString()

Type: Function

SCString TimeMSToString(const SCDateTime& **DateTimeMS**);

The **sc.TimeMSToString** function returns a text string for the time within the given **DateTimeMS** parameter. This includes the milliseconds part of the time component. Any date component in the given **DateTime** parameter will be ignored.

Example


```

if (sc.Index == sc.ArraySize - 1)
{
    // Log the current time.
    SCString TimeString = sc.TimeMSToString(sc.CurrentSystemDateTimeMS);

    sc.AddMessageToLog(TimeString, 0);
}

```

sc.TimePeriodSpan()

Type: Function

double **TimePeriodSpan**(unsigned int **TimePeriodType**, int **TimePeriodLength**);

The **sc.TimePeriodSpan()** function calculates the span of time based upon a time period length unit enumeration constant and a length parameter specifying the number of units.

Parameters

- **TimePeriodType**: The type of time period. This can be any of:
 - TIME_PERIOD_LENGTH_UNIT_MINUTES
 - TIME_PERIOD_LENGTH_UNIT_DAYS
 - TIME_PERIOD_LENGTH_UNIT_WEEKS
 - TIME_PERIOD_LENGTH_UNIT_MONTHS
 - TIME_PERIOD_LENGTH_UNIT_YEARS
- **TimePeriodLength**: The number of units specified with **TimePeriodType**. For example if you want 1 Day, then you will set TimePeriodLength to 1 and **TimePeriodType** to TIME_PERIOD_LENGTH_UNIT_DAYS.

Example

```
SCDateTime TimeIncrement = sc.TimePeriodSpan(TimePeriodType.GetTimePeriodType(), TimePeriodLength);
```

sc.TimeSpanOfBar()

Type: Function

SCDateTime **TimeSpanOfBar**(int **BarIndex**);

This function returns the time span of the bar specified by the **BarIndex** parameter.

Parameters

- [BarIndex](#).

Example

sc.TimeStringToSCDateTime()

Type: Function

SCDateTime **TimeStringToSCDateTime**(const SCString& **TimeString**)

The **sc.TimeStringToSCDateTime()** function converts the given **TimeString** parameter to a [SCDateTime](#) type.

The supported time format is: **HH:MM:SS.MS**. Milliseconds are optional. The hours can have a negative sign in front of it.

For information about the SCString type, refer to [Working With Text Strings and Setting Names](#).

sc.TimeToString()

Type: Function

SCString **TimeToString**(const SCDateTime& **DateTime**);

The **sc.TimeToString** function returns a text string for the time within the given **DateTime** parameter. Any date component in the given **DateTime** parameter will be ignored.

Example

```
if (sc.Index == sc.ArraySize - 1)
{
    // Log the current time.
    SCString TimeString = sc.TimeToString(sc.CurrentSystemDateTime);

    sc.AddMessageToLog(TimeString, 0);
}
```

sc.TradingDayStartsInPreviousDate()

Type: Function

int **TradingDayStartsInPreviousDate**();

The **sc.TradingDayStartsInPreviousDate()** function

sc.TriangularMovingAverage()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **TriangularMovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **TriangularMovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.TriangularMovingAverage()** function calculates the Triangular Moving Average study.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0] (Extra Array) is used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

Example

```
sc.TriangularMovingAverage(sc.BaseData[SC_LAST], sc.Subgraph[0], 10);  
float TriangularMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.TRIX()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **TRIX**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **TRIX**(SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.TRIX()** function calculates the TRIX study.

Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

Example

```
sc.TRIX(sc.BaseDataIn[SC_LAST], sc.Subgraph[0].Arrays[0], 20);  
float TRIX = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.TrueRange()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **TrueRange**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**);

SCFloatArrayRef **TrueRange**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**);
[Auto-looping only](#).

The **sc.TrueRange()** function calculates the True Range study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

Example

```
sc.TrueRange(sc.BaseDataIn, sc.Subgraph[0]);  
float TrueRange = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.UltimateOscillator()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **UltimateOscillator**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut1**, SCSubgraphRef **SubgraphOut2**, int **Index**, int **Length1**, int **Length2**, int **Length3**);

SCFloatArrayRef **UltimateOscillator**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut1**, SCSubgraphRef **SubgraphOut2**, int **Length1**, int **Length2**, int **Length3**); [Auto-looping only](#).

The **sc.UltimateOscillator()** function calculates the Ultimate Oscillator.

Parameters

- [BaseDataIn](#).

- [SubgraphOut1](#). For this function, sc.Subgraph[].Arrays[0-8] (Extra Arrays) are used for internal calculations and additional results output.
- [SubgraphOut2](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length1](#).
- [Length2](#).
- [Length3](#).

Example

```
sc.UltimateOscillator(sc.BaseDataIn, sc.Subgraph[0], sc.Subgraph[1], 7, 14, 28);

//Access the study value at the current index
float UltimateOscillator = sc.Subgraph[0][sc.Index];
```

sc.UploadChartImage()

Type: Function

The **sc.UploadChartImage** function when called will upload an image of the chart the study instance is applied to, to the Sierra Chart server. The full URL to the image is saved in a text file ([Sierra Chart installation folder]\Images\ImageLog.txt).

This functionality is documented on the [Image Upload Service](#) page.

This function takes no parameters.

Also refer to [sc.SaveChartImageToFile](#) and [sc.SaveChartImageToFileExtended\(\)](#).

Example

```
sc.UploadChartImage();
```

sc.UserDrawnChartDrawingExists()

Refer to the [sc.UserDrawnChartDrawingExists\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

sc.UseTool()

Type: Function

For more information, refer to the [Using Tools with sc.UseTool\(\)](#) section on the **Using Tools From an Advanced Custom Study** page.

sc.VHF()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **VHF**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **VHF**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.VHF()** function calculates the Vertical Horizontal Filter study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.VHF(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0], 10);  
  
float VHF = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.VolumeWeightedMovingAverage()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **VolumeWeightedMovingAverage** (SCFloatArrayRef **FloatArrayDataIn**, SCFloatArrayRef **FloatArrayVolumIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **VolumeWeightedMovingAverage** (SCFloatArrayRef **InData**, SCFloatArrayRef **InVolume**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.VolumeWeightedMovingAverage()** function calculates the Volume Weighted Moving Average study.

Parameters

- [FloatArrayDataIn](#).
- [FloatArrayVolumIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.VolumeWeightedMovingAverage(sc.BaseData[SC_LAST], sc.BaseData[SC_VOLUME], sc.Subgraph[0], 100);  
float VolumeWeightedMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.Vortex()

Type: Intermediate Study Calculation Function

```
void Vortex (SCBaseDataRef ChartBaseDataIn, SCSubgraphRef VMPlusOut, SCSubgraphRef VMMinusOut, int Index, int VortexLength);
```

```
void Vortex (SCBaseDataRef ChartBaseDataIn, SCSubgraphRef VMPlusOut, SCSubgraphRef VMMinusOut, int VortexLength); Auto-looping only.
```

Parameters

- [ChartBaseDataIn](#).
- [VMPlusOut](#).
- [VMMinusOut](#).
- [Index](#).
- [VortexLength](#).

Example

sc.WeightedMovingAverage()

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef WeightedMovingAverage(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef WeightedMovingAverage(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.WeightedMovingAverage()** function calculates the Weighted Moving Average study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

- [Length](#).

Example

```
sc.WeightedMovingAverage(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);

float WeightedMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.WellesSum()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **WellesSum** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **WellesSum** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.WellesSum()** function calculates the Welles Summation of the **FloatArrayIn** data.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.WellesSum(sc.Subgraph[1], sc.Subgraph[0], 10);

float WellesSum = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.WildersMovingAverage()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **WildersMovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **WildersMovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.WildersMovingAverage()** function calculates the Wilders Moving Average study.

Parameters

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.WildersMovingAverage(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);  
  
float WildersMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.WilliamsAD()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **WilliamsAD**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**);

SCFloatArrayRef **WilliamsAD**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**);
[Auto-looping only](#).

The **sc.WilliamsAD()** function calculates the Williams Accumulation/Distribution study.

Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- [Index](#).

Example

```
sc.WilliamsAD(sc.BaseDataIn, sc.Subgraph[0]);  
  
float WilliamsAD = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.WilliamsR()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **WilliamsR**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **WilliamsR**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.WilliamsR()** function calculates the Williams %R study.

Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.WilliamsR(sc.BaseDataIn, sc.Subgraph[0], 10);  
  
float WilliamsR = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.WriteBarAndStudyDataToFile()

Type: Function

```
int sc.WriteBarAndStudyDataToFile(int StartingIndex, SCString &OutputPathAndFileName, int IncludeHiddenStudies);
```

The **sc.WriteBarAndStudyDataToFile()** function writes the chart bar data and all of the Subgraphs data for all of the studies on the chart, to the specified file for the chart the study function instance is applied to.

Parameters

- **StartingIndex**: When this parameter is set to 0, then a new file is created, and data for all chart columns/bars are written to the file except for the last chart column/bar. A header line is also written to the file when this parameter is 0. When this is set to an index greater than zero, then the chart bar and study data starting at this bar index are appended to the existing specified file.
- **OutputPathAndFileName**: This is the complete path and filename for the file to output the chart bar and study data to.
- **IncludeHiddenStudies**: When this is set to a nonzero number, then hidden studies are also written to the file. Otherwise, they are not.

Example

For an example to use this function, refer to the **scsf_WriteBarAndStudyDataToFile** function in the **/ACS_Source/studies6.cpp** file in the folder Sierra Chart is installed to.

sc.WriteBarAndStudyDataToFileEx()

Type: Function

```
int WriteBarAndStudyDataToFileEx(const n_ACSIL::s_WriteBarAndStudyDataToFile&
WriteBarAndStudyDataToFileParams);
```

The **sc.WriteBarAndStudyDataToFileEx** function writes the chart bar data and all of the Subgraphs data for all of the studies on the chart, to the specified file for the chart the study function instance is applied to.

It supports additional parameters for more control as compared to the [sc.WriteBarAndStudyDataToFile](#) function.

Parameters

The members of the s_WriteBarAndStudyDataToFile structure for the **WriteBarAndStudyDataToFileParams** parameter are as follows:

- **int StartingIndex:** When this parameter is set to 0, then a new file is created, and data for all chart columns/bars are written to the file except for the last chart column/bar. A header line is also written to the file when this parameter is 0. When this is set to an index greater than zero, then the chart bar and study data starting at this bar index are appended to the existing specified file.
- **SCString &OutputPathAndFileName:** This is the complete path and filename for the file to output the chart bar and study data to.
- **int IncludeHiddenStudies:** When this is set to a nonzero number, then hidden studies are also written to the file. Otherwise, they are not.
- **int IncludeHiddenSubgraphs:** When this is set to a nonzero number, then study Subgraphs which have the Draw Style set to be either Hidden or Ignore are also written to the file. Otherwise, they are not.
- **int AppendOnlyToFile:** When this is set to a nonzero number, then the rows of data beginning at **StartingIndex** are always appended to the file rather than the file being rewritten. And append will also occur when StartingIndex is > 0.
- **int IncludeLastBar:** When this is set to a nonzero number, then the data for the last bar in the chart will be written to the file as well. You have to be careful when setting this to a nonzero number because during study updating, you will not want to use the same **StartingIndex** unless that is 0. Otherwise, the last bar and study data will be outputted more than once to the file.

sc.WriteFile()

Type: Function

```
int WriteFile(const int FileHandle, const char* Buffer, const int BytesToWrite, unsigned int*
p_BytesWritten);
```

The **sc.WriteFile()** function writes to a file opened with [sc.OpenFile\(\)](#) using the provided **FileHandle** from the sc.OpenFile function.

It writes the number of **BytesToWrite** that are stored in **Buffer**. The actual number of bytes written is then stored in **p_BytesWritten**.

The function returns **0** if there is an error writing the bytes to the file. Otherwise, the function returns **1**.

Parameters

- **FileHandle**: The File Handle for the requested file as determined from the `sc.OpenFile()` function call.
- **Buffer**: Holds the data to write to the file.
- **BytesToWrite**: The number of bytes to write to the file.
- **p_BytesWritten**: A pointer to the variable that stores the actual number of bytes written to the file.

Example

```
if (sc.Index == sc.ArraySize - 1)
{
    int FileHandle;
    sc.OpenFile("Testing.txt", n_ACSIL::FILE_MODE_OPEN_TO_APPEND, FileHandle);

    unsigned int BytesWritten = 0;

    sc.WriteFile(FileHandle, "Test Line\r\n", 11, &BytesWritten);

    sc.CloseFile(FileHandle);
}
```

Also refer to [sc.OpenFile](#), [sc.CloseFile](#), [sc.ReadFile](#), [sc.WriteFile](#), [sc.GetLastFileErrorCode](#).

sc.YPixelCoordinateToGraphValue()

Type: Function

```
double YPixelCoordinateToGraphValue(int YPixelCoordinate);
```

The **sc.YPixelCoordinateToGraphValue()** function calculates the study or main price graph value from the given Y-axis pixel coordinate. Each chart is divided into Chart Regions and each Chart Region can contain one or more graphs. This function will determine the value used by a graph based upon the scaling of the Chart Region for the given Y-axis coordinate.

This function can only function properly after the chart is actually drawn. Otherwise, 0 will be returned. It will return a value based upon what was last drawn and not what the study function is currently doing. This is something to keep in mind because the function is going to be returning data based upon the last time the chart was drawn. The chart is always drawn after the study is calculated. So a study function may be receiving a value from this function which is not current relative to changes with the data in the study instance.

Example

```
float GraphValue= sc.YPixelCoordinateToGraphValue(sc.ActiveToolYPosition);
```

sc.ZeroLagEMA()

Type: Intermediate Study Calculation Function

SCFloatArrayRef **ZeroLagEMA**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **ZeroLagEMA**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.ZeroLagEMA()** function calculates Ehlers' Zero Lag Exponential Moving Average study.

Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

Example

```
sc.ZeroLagEMA(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
  
float ZeroLagEMA = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

sc.ZigZag()

Type: Function

SCFloatArrayRef **ZigZag** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSubgraphRef **Out**, int **Index**, float **ReversalPercent**, int **StartIndex**);

SCFloatArrayRef **ZigZag** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSubgraphRef **Out**, float **ReversalPercent**, int **StartIndex**); [Auto-looping only](#).

SCFloatArrayRef **ZigZag** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSubgraphRef **Out**, int **Index**, float **ReversalPercent**, float **ReversalAmount**, int **StartIndex**);

SCFloatArrayRef **ZigZag** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSubgraphRef **Out**, float **ReversalPercent**, float **ReversalAmount**, int **StartIndex**); [Auto-looping only](#).

Parameters

- [InputDataHigh](#).
- [InputDataLow](#).
- [Out](#).
- [Index](#).
- **ReversalPercent**: .
- **ReversalAmount**: .
- [StartIndex](#).

Example

sc.ZigZag2()

Type: Function

SCFloatArrayRef **ZigZag2** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSubgraphRef **Out**, int **Index**, int **NumberOfBars**, float **ReversalAmount**, int **StartIndex**);

SCFloatArrayRef **ZigZag2** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSubgraphRef **Out**, int **NumberOfBars**, float **ReversalAmount**, int **StartIndex**); [Auto-looping only](#).

Parameters

- [InputDataHigh](#).
- [InputDataLow](#).
- [Out](#).
- [Index](#).
- **NumberOfBars**: .
- **ReversalAmount**: .
- [StartIndex](#).

Example

min()

Type: Function

min(a, b)

min() takes two parameters (**a** and **b**) and returns the lesser of the two.

Example

```
int MinValue = min(-5, 3);  
  
// MinValue will equal -5
```

max()

Type: Function

max(a, b)

max() takes two parameters (**a** and **b**) and returns the greater of the two.

Example

```
int MaxValue = max(-5, 3);  
  
// MaxValue will equal 3
```

Common Function Parameter Descriptions

This is a list of all of the parameter descriptions for the common parameters for ACSIL functions in general.

This also includes Intermediate Study Calculation functions. For example, [sc.SimpleMovAvg\(\)](#) is an intermediate study calculation function.

BarIndex

int **BarIndex**: This specifies the index of the chart bar the function needs to perform a calculation on or perform some other function on.

ChartNumber

int **ChartNumber**: This specifies the specific chart identified by a number, within the Chartbook the study instance is contained within that the function call is related to.

Each chart has a number and can be seen on the title bar of the chart window after the #. Depending upon the configuration of the [Chart Header](#) it may also be displayed along the top header line of the chart.

To specify the same chart number of the chart the study instance is on, use **sc.ChartNumber** for this

parameter.

DateTime

SCDateTime **DateTime**: This specifies a Date-Time as a [SCDateTime](#) type.

DateTimeMS

SCDateTime **DateTimeMS**: This specifies a Date-Time as a [SCDateTime](#) type which is identical to a SCDateTime type.

BaseDateTimeln

SCDateTimeArray **BaseDateTimeln**: The type is a reference to a **SCDateTimeArray** . This is the main price/chart graph date and time array. Can only be **sc.BaseDateTimeln**.

FileHandle

int **FileHandle**: This is a file handle obtained with the function [sc.OpenFile](#).

BaseDataIn

SCBaseDataRef **BaseDataIn**: The BaseDataIn parameter is defined as a reference to a **SCFloatArrayArray**. This is the main chart graph arrays. The only object that you can use is **sc.BaseDataIn** for this parameter type.

FloatArrayIn

SCFloatArrayRef **FloatArrayIn**: The FloatArrayIn parameter is defined as a reference to a **SCFloatArray**. This is a data array of float values which is used as input to the intermediate study calculation function. This parameter type can be one of the following types: **sc.Subgraph[]**, **sc.Subgraph[].Data**, **sc.Subgraph[].Arrays[]**, **sc.BaseDataIn[ArrayIndex]**, or **sc.BaseData[ArrayIndex]**.

FloatArrayOut

SCFloatArrayRef **FloatArrayOut**: The type is a reference to a **SCFloatArray**. This is an output data array of float values. You need to pass a **sc.Subgraph[]**, **sc.Subgraph[].Data** or **sc.Subgraph[].Arrays** array.

Index

int **Index**: This parameter is the Index to the element in an output array (**SubgraphOut**,

FloatArrayOut) where the calculation result is outputted to. And the Index to the element or elements in the input array or arrays (**FloatArrayIn**, **BaseDataIn**, **BaseDateTimeln**) that are used in the internal calculations of the intermediate study calculation function, and where the calculation begins when back referencing the data in the input arrays. This parameter is optional when using Automatic Looping. It can be left out in all of the intermediate study calculation functions and will internally be set to **sc.Index** .

Length

int **Length**: The number of data array elements used in the calculations. This definition applies to all parameters with **Length** in their name.

Price

float **Price**: This is a price value as a float.

Multiplier

float **Multiplier**: A multiplier parameter is simply a value which is used to multiply another value used in the calculation. Such as multiplying the data from the input data array or a sub calculation from that input data array.

MovingAverageType

int **MovingAverageType**: MovingAverageType needs to be set to the type of moving average you want to use. This description also applies to parameters that have **MovAvgType**, **MovingAverageType**, or **MAType** in their name. The following are the Moving Average type constants that you can use:

- **MOVAVGTYPE_SIMPLE**: Simple moving average.
- **MOVAVGTYPE_EXPONENTIAL**: Exponential moving average.
- **MOVAVGTYPE_LINEARREGRESSION**: Linear regression or least-squares moving average.
- **MOVAVGTYPE_WEIGHTED** : Weighted moving average.
- **MOVAVGTYPE_WILDERS**: Wilder's moving average.
- **MOVAVGTYPE_SIMPLE_SKIP_ZEROS**: Simple Moving Average that skips zero values.
- **MOVAVGTYPE_SMOOTHED**: Smoothed moving average.

StudyID

int **StudyID**: This is the study ID which is obtained using the following functions:

- [sc.Input\[\].GetStudyID](#)

- [sc.Input\[\].SetStudyID](#)
- [sc.Input\[\].GetChartStudySubgraphValues](#)
- [sc.Input\[\].SetChartStudySubgraphValues](#)
- [sc.Input\[\].SetChartStudyValues](#)
- [sc.Input\[\].SetStudySubgraphValues](#)

Subgraph

SCSubgraphRef **Subgraph**: The type is a reference to a **sc.Subgraph[]**. You need to pass a **sc.Subgraph[]** object.

This type is either used for data input or output. In the case of data output, the results are written to the **sc.Subgraph[].Data** and **sc.Subgraph[].Arrays** arrays. The **sc.Subgraph[].Arrays** arrays are used for intermediate calculations and additional output.

SubgraphOut

SCSubgraphRef **SubgraphOut**: The type is a reference to a **sc.Subgraph[]**. You need to pass a **sc.Subgraph[]** object. The results are written to the **sc.Subgraph[].Data** and **sc.Subgraph[].Arrays** arrays. The **sc.Subgraph[].Arrays** arrays are used for intermediate calculations and additional output.

Symbol

SCString **Symbol**: This specifies a [symbol](#) where a function requires that a symbol is specified.

To use the symbol of the chart which contains the study instance which is calling the function which requires this parameter, use [sc.Symbol](#).

*Last modified Wednesday, 23rd August, 2023.