

```
// File Name: scstructures.h
```

```
// Notice: Changes to existing data structures and types in this file need to consider that existing custom studies can be affected by this. There must not be changes to existing members.
```

```
#ifndef _SCSTRUCTURES_H_
#define _SCSTRUCTURES_H_
```

```
#ifndef NOMINMAX
```

```
// If NOMINMAX is not already defined, make a note if it so that it can be
// disabled at a later point.
```

```
#define SC_UNDEF_NOMINMAX
#endif
```

```
#include <windows.h> //Main Windows SDK header file
#include <stdio.h> //This header is for input/output including file input/output
#include <stdarg.h>
#include <float.h>
#include <io.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
// #include "ASCILDrawGraphics.h"
#include <algorithm>
```

```
#include <cstdlib>
#ifdef __GNUC__
// cstdlib on MinGW/GCC appears to always define NOMINMAX, which is not
// usually desired on remote builds for custom study DLLs.
```

```
//
// Note: we do not want to undefine NOMINMAX if it was intentionally defined
// earlier than this include.
```

```
#ifdef SC_UNDEF_NOMINMAX
#undef NOMINMAX
```

```
#endif
#endif
#include <signal>
#include <setjmp>
#include <cstdarg>
#include <typeinfo>
#include <bitset>
#include <functional>
#include <utility>
#include <ctime>
#include <cstddef>
#include <new>
#include <memory>
#include <climits>
#include <cfloat>
#include <limits>
#include <exception>
#include <stdexcept>
#include <cassert>
#include <cerrno>
#include <cctype>
#include <cwctype>
#include <cstring>
#include <wchar>
#include <string>
#include <vector>
#include <deque>
```

```

#include <list>
#include <set>
#include <map>
#include <stack>
#include <queue>
#include <algorithm>
#include <iterator>
#include <cmath>
#include <complex>
#include <valarray>
#include <numeric>
#include <iosfwd>
#include <ios>
#include <istream>
#include <ostream>
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <streambuf>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <chrono>

#ifndef NOMINMAX
#ifndef max
#define max(a,b)      ((a) > (b)) ? (a) : (b)
#endif

#ifndef min
#define min(a,b)      (((a) < (b)) ? (a) : (b))
#endif
#endif

#define RGB_COLOR(r,g,b)      (((uint32_t)((uint8_t)(r)|((uint16_t)((uint8_t)(g))<<8))|(((uint32_t)(uint8_t)(b))<<16)))

/*****

#include "scconstants.h"
#include "scdatetime.h"
#include "SCSymbolData.h"
#include "sccolors.h"
#include "VAPContainer.h"
#include "SCString.h"

struct s_sc; // forward declaration

typedef s_sc& SCStudyInterfaceRef;

const int CURRENT_SC_NEW_ORDER_VERSION = 18;
const int CURRENT_SC_ORDER_FILL_DATA_VERSION = 4;

#define SCDLLEXPORT extern "C" __declspec(dllexport)
#define SCDLLCALL __cdecl

#define SCSFExport SCDLLEXPORT void SCDLLCALL // Use SCSFExport when defining a study function

#define SCDLLName(DLLName) \
    SCDLLEXPORT int SCDLLCALL scdll_DLLVersion() { return SC_DLL_VERSION; } \
    SCDLLEXPORT const char* SCDLLCALL scdll_DLLName() { return DLLName; }

typedef void (SCDLLCALL* fp_ACSGDIFunction)(HWND WindowHandle, HDC DeviceContext, SCStudyInterfaceRef);

```

```
/******
```

```
struct s_TimeAndSales
{
    SCDatetimeMS DateTime;// UTC
    float Price = 0;
    uint32_t Volume = 0;
    float Bid = 0;
    float Ask = 0;
    uint32_t BidSize = 0;
    uint32_t AskSize = 0;

    //This will always be a value >= 1. It is unlikely to wrap around, but it could. It will never be 0.
    uint32_t Sequence = 0;

    int16_t UnbundledTradeIndicator = 0;
    int16_t Type = 0;
    uint32_t TotalBidDepth = 0;
    uint32_t TotalAskDepth = 0;
    int8_t TradeIndicator = 0;

    s_TimeAndSales()
    {

    }

    explicit s_TimeAndSales(int Value)
    {

    }

    s_TimeAndSales& operator *= (float Multiplier)
    {
        Price *= Multiplier;
        Bid *= Multiplier;
        Ask *= Multiplier;

        return *this;
    }

    void Clear()
    {
        *this = s_TimeAndSales();
    }

    bool operator < (const s_TimeAndSales& Right) const
    {
        return Sequence < Right.Sequence;
    }
};

class c_SCTimeAndSalesArray
{
private:

protected:
    const int m_Version=1;
    s_TimeAndSales* m_p_Data = nullptr;
    int m_MaxSize = 0;
    int m_NextIndex = 0;
    int m_StartAtNext = 0;

public:
    /*=====*/
    c_SCTimeAndSalesArray()
```

```

{
}

/*=====*/
~c_SCTimeAndSalesArray()
{
}

/*=====*/
//Thread safe as long as an instance of this object is only used by one thread at a time.
int IsCreated() const
{
    return (m_p_Data != NULL) ? 1 : 0;
}

/*=====*/
int MaxSize() const
{
    return m_MaxSize;
}

/*=====*/
int Size() const
{
    if (m_p_Data == NULL)
        return 0;

    if (m_StartAtNext)
        return m_MaxSize; // Using the full buffer
    else
        return m_NextIndex; // Using only up to m_NextIndex
}

/*=====
This function is for backwards compatibility.
-----*/
int GetArraySize() const
{
    return Size();
}

/*=====
The oldest element is at index 0, and the newest element is at
index Size()-1.
-----*/
const s_TimeAndSales& operator [] (int Index) const
{
    static s_TimeAndSales DefaultElement;

    if (m_p_Data == NULL)
        return DefaultElement;

    if (Index < 0 || Index >= Size())
        return DefaultElement;

    // Translate the index
    if (m_StartAtNext)
    {
        Index += m_NextIndex;

        if (Index >= m_MaxSize)
            Index -= m_MaxSize;
    }

    return m_p_Data[Index];
}

```

```

}

/*=====
This does not copy the data, it just copies the pointer from the
given Time and Sales array.
-----*/
c_SCTimeAndSalesArray& operator = (const c_SCTimeAndSalesArray& Rhs)
{
    if (&Rhs == this)
        return *this;

    m_p_Data = Rhs.m_p_Data;
    m_MaxSize = Rhs.m_MaxSize;
    m_NextIndex = Rhs.m_NextIndex;
    m_StartAtNext = Rhs.m_StartAtNext;

    return *this;
}

bool IsRecordIndexTrade (int Index)
{
    return (operator[](Index).Type == SC_TS_ASK || operator[](Index).Type == SC_TS_BID );
}

SCDateTimeMS GetDateTimeOfLastTradeRecord()
{
    int ArraySize = Size();
    if (ArraySize == 0)
        return 0.0;

    for (int RecordIndex = ArraySize - 1; RecordIndex >= 0; RecordIndex--)
    {
        if (IsRecordIndexTrade(RecordIndex))
            return operator[](RecordIndex).DateTime;
    }

    return 0.0;
}

const uint32_t GetIndexOfLessEqualRecordAtDateTime(const SCDateTimeMS& SearchDateTime) const
{
    if (Size() == 0)
        return 0;

    //Perform binary search using operator[] to locate search date-time
    //Elements are in time-ascending order

    int IndexUpper = 0;
    int IndexLower = IndexUpper;
    const size_t EndIndex = Size();
    int Count = Size();

    while (Count > 0)
    {
        const int Step = Count / 2;
        const int TestIndex = IndexUpper + Step;

        if (operator[](TestIndex).DateTime <= SearchDateTime)
        {
            IndexLower = TestIndex;
            IndexUpper = TestIndex + 1;
            Count -= Step + 1;
        }
        else
        {

```

```

        Count = Step;
    }
}

return IndexLower;
}

const s_TimeAndSales& GetLessEqualRecordAtDateTime(const SCDatetimeMS& SearchDateTime) const
{
    static s_TimeAndSales DefaultElement;

    if (Size() == 0)
        return DefaultElement;

    uint32_t FoundIndex = GetIndexOfLessEqualRecordAtDateTime(SearchDateTime);

    return operator[] (FoundIndex);
}

```

```

uint32_t GetRecordIndexAtGreaterThanSequenceNumber(uint32_t SequenceNumber) const
{
    /* This function needs to perform a fast binary search of the c_SCTimeAndSalesArray
    container and return the index of the record with a sequence number greater than
    the given SequenceNumber.
    Keep in mind that the member variables m_NextIndex, m_StartAtNext indicate what
    the first record is in the array.
    std::upper_bound cannot be used unless we implement forward iterators on this class.
    Return -1 if the array is empty.
    If the first array element has a sequence number greater than SequenceNumber, then return 0.
    If the last array element has a sequence number less than SequenceNumber, then return the size of the array -

```

1.

```

    */

    if (Size() == 0)
        return -1;

    if (SequenceNumber == 0)
        return 0;

    int IndexUpper = 0;
    int Count = Size();

    while (Count > 0)
    {
        const int Step = Count >> 1;
        const int TestIndex = IndexUpper + Step;

        if (operator[] (TestIndex).Sequence > SequenceNumber)
        {
            Count = Step;
        }
        else
        {
            IndexUpper = TestIndex + 1;
            Count -= Step + 1;
        }
    }

    if (IndexUpper >= Size())
        IndexUpper = Size() - 1;

    return IndexUpper;
}

```

```

uint32_t GetSequenceNumberOfLastElement() const
{
    uint32_t ArraySize = Size();

    if (ArraySize == 0)
        return 0;

    return operator[](ArraySize - 1).Sequence;
}

void ValidateAndCorrectPriorSequenceNumber(uint32_t& r_PriorSequenceNumber) const
{
    //There has either been a sequence number wrap or reset back to 0 so we need to consider that. Otherwise, we
    create a condition, where we do not process new time and sales data.

    if (r_PriorSequenceNumber > GetSequenceNumberOfLastElement())
    {
        //CommonAddMessageToLog("Prior Time Sales sequence number reset", false);

        r_PriorSequenceNumber = 0;
    }
}
};

```

```

/*****/

class s_UseTool
{
public:
    static const int DEFAULT_COLOR_VALUE = 0xFF000000;

private:
    unsigned int Size = sizeof(s_UseTool);
public:
    int ChartNumber = 0;
    int Unused_1 = 0;
    int LineNumber = -1;
    int ExtendLeft = -1;
    int ExtendRight = -1;
    int TransparencyLevel = -1;
    int FourthIndex = -1;
    float BeginValue = FLT_MIN;
    float EndValue = FLT_MIN;
    int Region = -1;
    uint32_t Color = DEFAULT_COLOR_VALUE;
    uint64_t Unused_2 = 0;
    int FontSize = -1;
    char ShowBeginMark = -1;
    char ShowEndMark = -1;
    int16_t IsSelected = 0;
    float RetracementLevels[ACSIL_DRAWING_MAX_LEVELS] = {};
    SCDateTimeMS BeginDateTime;
    SCDateTimeMS EndDateTime;
    SCString Text;

    DrawingTypeEnum DrawingType = DRAWING_UNKNOWN;
    uint16_t LineWidth = 0xffff;
    SubgraphLineStyles LineStyle = LINESSTYLE_UNSET;

    int AddMethod = UTAM_ADD_OR_ADJUST;

private:
    unsigned int Version = SC_DLL_VERSION;

```

```

public:
    int ReverseTextColor = -1;

    unsigned int FontBackColor = DEFAULT_COLOR_VALUE;
    SCString FontFace;
    int FontBold = -1;
    int FontItalic = -1;
    int TextAlignment = -1;
    int UseRelativeVerticalValues = -1;

    uint32_t SecondaryColor = DEFAULT_COLOR_VALUE;
    int SquareEdge = -1;

    int DisplayHorizontalLineValue = -1;

    SCDateTimeMS ThirdDateTime;
    float ThirdValue = FLT_MIN;

    int BeginIndex = -1;
    int EndIndex = -1;
    int ThirdIndex = -1;

    uint32_t LevelColor[ACSIL_DRAWING_MAX_LEVELS] = {};
    int LevelStyle[ACSIL_DRAWING_MAX_LEVELS] = {}; // SubgraphLineStyles
    int LevelWidth[ACSIL_DRAWING_MAX_LEVELS] = {};

    int ShowTickDifference = -1;
    int ShowCurrencyValue = -1;
    int ShowPriceDifference = -1;
    int ShowPercentChange = -1;
    int ShowTimeDifference = -1;
    int ShowNumberOfBars = -1;
    int ShowAngle = -1;
    int ShowEndPointPrice = -1;
    int MultiLineLabel = -1;

    int ShowEndPointDateTime = -1;
    int ShowEndPointDate = -1;
    int ShowEndPointTime = -1;

    int ShowPercent = -1;
    int ShowPrice = -1;
    int RoundToTickSize = -1;

    int FixedSizeArrowHead = -1;
    float ArrowHeadSize = -FLT_MAX;

    int MarkerType = -1;
    int MarkerSize = -1;
    int UseBarSpacingForMarkerSize = -1;

    int TimeExpVerticals = -1;
    int TimeExpTopLabel1 = -1;
    int TimeExpTopLabel2 = -1;
    int TimeExpBottomLabel1 = -1;
    int TimeExpBottomLabel2 = -1;
    int TimeExpBasedOnTime = -1;

    int TransparentLabelBackground = -1;
    int AddAsUserDrawnDrawing = 0;

    int ShowVolume = -1;
    int DrawFromEnd = -1;
    int DrawUnderneathMainGraph = -1;

```



```

int ShowLabels = -1;
int ShowLabelsAtEnd = -1;
int AllLevelsSameColorStyle = -1;

int UseToolConfigNum = -1;

int ShowAskBidDiffVolume = -1;

int FlatEdge = -1;

int DrawWithinRegion = -1;

int VerticalText = -1;

uint32_t TextColor = DEFAULT_COLOR_VALUE;

int CenterText = -1;
int CenterPriceLabels = -1;

int NoVerticalOutline = -1;

int AllowSaveToChartbook = -1;

int AssociatedStudyID = 0;

int16_t HideDrawing = -1;

int16_t ClearExistingText = 0;

int16_t LockDrawing = -1;

int16_t DrawOutlineOnly = -1;

int16_t NumCycles = -1;

float ExtendMultiplier = -1;

int16_t DrawMidline = -1;

int16_t AllowCopyToOtherCharts = -1;

//2250
SCString Symbol;

// 2271
int FontUnderline = -1;

// 2293
int AdvancedCustomStudyAdded = 0;

// 2545
uint8_t UseHighResolutionForWindowRelativeCoordinates = 0;

// Note: When adding new members, remember to update CopyAndUpdateUseToolStruct function.

s_UseTool()
{
    for (int Index = 0; Index < ACSIL_DRAWING_MAX_LEVELS; ++Index)
    {
        RetracementLevels[Index] = -FLT_MAX;
        LevelColor[Index] = DEFAULT_COLOR_VALUE;
        LevelStyle[Index] = LINESTYLE_UNSET;
        LevelWidth[Index] = -1;
    }
}

```

```
}
```

```
void Clear()
```

```
{
```

```
    const s_UseTool DefaultUseTool;
```

```
    ChartNumber = DefaultUseTool.ChartNumber;
```

```
    LineNumber = DefaultUseTool.LineNumber;
```

```
    ExtendLeft = DefaultUseTool.ExtendLeft;
```

```
    ExtendRight = DefaultUseTool.ExtendRight;
```

```
    TransparencyLevel = DefaultUseTool.TransparencyLevel;
```

```
    FourthIndex = DefaultUseTool.FourthIndex;
```

```
    BeginValue = DefaultUseTool.BeginValue;
```

```
    EndValue = DefaultUseTool.EndValue;
```

```
    Region = DefaultUseTool.Region;
```

```
    Color = DefaultUseTool.Color;
```

```
    FontSize = DefaultUseTool.FontSize;
```

```
    ShowBeginMark = DefaultUseTool.ShowBeginMark;
```

```
    ShowEndMark = DefaultUseTool.ShowEndMark;
```

```
    IsSelected = DefaultUseTool.IsSelected;
```

```
    for (int LevelIndex = 0; LevelIndex < ACSIL_DRAWING_MAX_LEVELS; LevelIndex++)
```

```
    {
```

```
        RetracementLevels[LevelIndex] = DefaultUseTool.RetracementLevels[LevelIndex];
```

```
        LevelColor[LevelIndex] = DefaultUseTool.LevelColor[LevelIndex];
```

```
        LevelStyle[LevelIndex] = DefaultUseTool.LevelStyle[LevelIndex];
```

```
        LevelWidth[LevelIndex] = DefaultUseTool.LevelWidth[LevelIndex];
```

```
    }
```

```
    BeginDateTime = DefaultUseTool.BeginDateTime;
```

```
    EndDateTime = DefaultUseTool.EndDateTime;
```

```
    DrawingType = DefaultUseTool.DrawingType;
```

```
    LineWidth = DefaultUseTool.LineWidth;
```

```
    LineStyle = DefaultUseTool.LineStyle;
```

```
    AddMethod = DefaultUseTool.AddMethod;
```

```
    ReverseTextColor = DefaultUseTool.ReverseTextColor;
```

```
    FontBackColor = DefaultUseTool.FontBackColor;
```

```
    FontFace = DefaultUseTool.FontFace;
```

```
    FontBold = DefaultUseTool.FontBold;
```

```
    FontItalic = DefaultUseTool.FontItalic;
```

```
    TextAlignment = DefaultUseTool.TextAlignment;
```

```
    UseRelativeVerticalValues = DefaultUseTool.UseRelativeVerticalValues;
```

```
    SecondaryColor = DefaultUseTool.SecondaryColor;
```

```
    SquareEdge = DefaultUseTool.SquareEdge;
```

```
    DisplayHorizontalLineValue = DefaultUseTool.DisplayHorizontalLineValue;
```

```
    ThirdDateTime = DefaultUseTool.ThirdDateTime;
```

```
    ThirdValue = DefaultUseTool.ThirdValue;
```

```
    BeginIndex = DefaultUseTool.BeginIndex;
```

```
    EndIndex = DefaultUseTool.EndIndex;
```

```
    ThirdIndex = DefaultUseTool.ThirdIndex;
```

```
    ShowTickDifference = DefaultUseTool.ShowTickDifference;
```

```
    ShowCurrencyValue = DefaultUseTool.ShowCurrencyValue;
```

```
    ShowPriceDifference = DefaultUseTool.ShowPriceDifference;
```

```
    ShowPercentChange = DefaultUseTool.ShowPercentChange;
```

```
    ShowTimeDifference = DefaultUseTool.ShowTimeDifference;
```

```
    ShowNumberOfBars = DefaultUseTool.ShowNumberOfBars;
```

```
    ShowAngle = DefaultUseTool.ShowAngle;
```

```
    ShowEndPointPrice = DefaultUseTool.ShowEndPointPrice;
```

```
    MultiLineLabel = DefaultUseTool.MultiLineLabel;
```

```
    ShowEndPointDateTime = DefaultUseTool.ShowEndPointDateTime;
```

```
    ShowEndPointDate = DefaultUseTool.ShowEndPointDate;
```

```
    ShowEndPointTime = DefaultUseTool.ShowEndPointTime;
```

```
    ShowPercent = DefaultUseTool.ShowPercent;
```

```
    ShowPrice = DefaultUseTool.ShowPrice;
```

```

RoundToTickSize = DefaultUseTool.RoundToTickSize;
FixedSizeArrowHead = DefaultUseTool.FixedSizeArrowHead;
ArrowHeadSize = DefaultUseTool.ArrowHeadSize;
MarkerType = DefaultUseTool.MarkerType;
MarkerSize = DefaultUseTool.MarkerSize;
UseBarSpacingForMarkerSize = DefaultUseTool.UseBarSpacingForMarkerSize;
TimeExpVerticals = DefaultUseTool.TimeExpVerticals;
TimeExpTopLabel1 = DefaultUseTool.TimeExpTopLabel1;
TimeExpTopLabel2 = DefaultUseTool.TimeExpTopLabel2;
TimeExpBottomLabel1 = DefaultUseTool.TimeExpBottomLabel1;
TimeExpBottomLabel2 = DefaultUseTool.TimeExpBottomLabel2;
TimeExpBasedOnTime = DefaultUseTool.TimeExpBasedOnTime;
TransparentLabelBackground = DefaultUseTool.TransparentLabelBackground;
AddAsUserDrawnDrawing = DefaultUseTool.AddAsUserDrawnDrawing;
ShowVolume = DefaultUseTool.ShowVolume;
DrawFromEnd = DefaultUseTool.DrawFromEnd;
DrawUnderneathMainGraph = DefaultUseTool.DrawUnderneathMainGraph;
ShowLabels = DefaultUseTool.ShowLabels;
ShowLabelsAtEnd = DefaultUseTool.ShowLabelsAtEnd;
AllLevelsSameColorStyle = DefaultUseTool.AllLevelsSameColorStyle;
UseToolConfigNum = DefaultUseTool.UseToolConfigNum;
ShowAskBidDiffVolume = DefaultUseTool.ShowAskBidDiffVolume;
FlatEdge = DefaultUseTool.FlatEdge;
DrawWithinRegion = DefaultUseTool.DrawWithinRegion;
VerticalText = DefaultUseTool.VerticalText;
TextColor = DefaultUseTool.TextColor;
CenterText = DefaultUseTool.CenterText;
CenterPriceLabels = DefaultUseTool.CenterPriceLabels;
NoVerticalOutline = DefaultUseTool.NoVerticalOutline;
AllowSaveToChartbook = DefaultUseTool.AllowSaveToChartbook;
AssociatedStudyID = DefaultUseTool.AssociatedStudyID;
HideDrawing = DefaultUseTool.HideDrawing;
ClearExistingText = DefaultUseTool.ClearExistingText;
LockDrawing = DefaultUseTool.LockDrawing;
DrawOutlineOnly = DefaultUseTool.DrawOutlineOnly;
NumCycles = DefaultUseTool.NumCycles;
ExtendMultiplier = DefaultUseTool.ExtendMultiplier;
DrawMidline = DefaultUseTool.DrawMidline;
AllowCopyToOtherCharts = DefaultUseTool.AllowCopyToOtherCharts;

Symbol.Clear();

FontUnderline = DefaultUseTool.FontUnderline;
AdvancedCustomStudyAdded = DefaultUseTool.AdvancedCustomStudyAdded;

UseHighResolutionForWindowRelativeCoordinates =
DefaultUseTool.UseHighResolutionForWindowRelativeCoordinates;
}

unsigned int GetSize() const
{
    return Size;
}

unsigned int GetVersion() const
{
    return Version;
}
};

/*****
struct s_ACSTradeStatistics
{
    static const int CURRENT_TRADESTATS_VERSION = 3;

```

```

double ClosedProfit = 0;
double ClosedLoss = 0;

double TotalCommission = 0;

double MaximumRunup = 0;
double MaximumDrawdown = 0;

double MaximumTradeRunup = 0;
double MaximumTradeDrawdown = 0;
double MaximumOpenPositionProfit = 0;
double MaximumOpenPositionLoss = 0;
int TotalWinningTrades = 0;
int TotalLosingTrades = 0;
int TotalLongTrades = 0;
int TotalShortTrades = 0;

double TotalWinningQuantity = 0;
double TotalLosingQuantity = 0;
double TotalFilledQuantity = 0;
double LargestTradeQuantity = 0;

double LargestWinningTrade = 0;
double LargestLosingTrade = 0;
int TimeInWinningTrades = 0;
int TimeInLosingTrades = 0;

int MaxConsecutiveWinners = 0;
int MaxConsecutiveLosers = 0;

int Version = CURRENT_TRADESTATS_VERSION;
double LastTradeProfitLoss = 0;
double LastTradeQuantity = 0;
SCDateTime LastFillDateTime = 0;
SCDateTime LastEntryDateTime = 0;
SCDateTime LastExitDateTime = 0;

double TotalBuyQuantity = 0;
double TotalSellQuantity = 0;

double ClosedProfitLoss() const
{
    return ClosedProfit + ClosedLoss;
}

double ProfitFactor() const
{
    return ClosedLoss == 0 ? 0 : fabs(ClosedProfit / ClosedLoss);
}

int TotalTrades() const
{
    return TotalWinningTrades + TotalLosingTrades;
}

double TotalPercentProfitable() const
{
    return TotalTrades() == 0 ? 0 : static_cast<double>(TotalWinningTrades) / static_cast<double>(TotalTrades());
}

double TotalClosedQuantity() const
{
    return TotalWinningQuantity + TotalLosingQuantity;
}

```

```

double AvgQuantityPerTrade() const
{
    return TotalTrades() == 0 ? 0 : static_cast<double>(TotalClosedQuantity()) / static_cast<double>(TotalTrades());
}

double AvgQuantityPerWinningTrade() const
{
    return TotalWinningTrades == 0 ? 0 : static_cast<double>(TotalWinningQuantity) / static_cast<double>(TotalWinningTrades);
}

double AvgQuantityPerLosingTrade() const
{
    return TotalLosingTrades == 0 ? 0 : static_cast<double>(TotalLosingQuantity) / static_cast<double>(TotalLosingTrades);
}

double AvgProfitLoss() const
{
    return TotalTrades() == 0 ? 0 : ClosedProfitLoss() / TotalTrades();
}

double AvgProfit() const
{
    return TotalWinningTrades == 0 ? 0 : ClosedProfit / TotalWinningTrades;
}

double AvgLoss() const
{
    return TotalLosingTrades == 0 ? 0 : ClosedLoss / TotalLosingTrades;
}

double AvgProfitFactor() const
{
    return AvgLoss() == 0 ? 0 : fabs(AvgProfit() / AvgLoss());
}

double LargestWinnerPercentOfProfit() const
{
    return ClosedProfit == 0 ? 0 : LargestWinningTrade / ClosedProfit;
}

double LargestLoserPercentOfLoss() const
{
    return ClosedLoss == 0 ? 0 : fabs(LargestLosingTrade / ClosedLoss);
}

int TimeInTrades() const
{
    return TimeInWinningTrades + TimeInLosingTrades;
}

int AvgTimeInTrade() const
{
    return TotalTrades() == 0 ? 0 : TimeInTrades() / TotalTrades();
}

int AvgTimeInWinningTrade() const
{
    return TotalWinningTrades == 0 ? 0 : TimeInWinningTrades / TotalWinningTrades;
}

int AvgTimeInLosingTrade() const
{

```

```

    return TotalLosingTrades == 0 ? 0 : TimeInLosingTrades / TotalLosingTrades;
}

void Clear()
{
    *this = s_ACSTradeStatistics();
}
};

```

```

/*****

```

```

struct s_ACSTrade
{
    SCDateTime OpenDateTime;
    SCDateTime CloseDateTime;
    int TradeType = 0; // +1=long, -1=short
    double TradeQuantity = 0;
    double MaxClosedQuantity = 0; // Max closed quantity.
    double MaxOpenQuantity = 0;
    double EntryPrice = 0;
    double ExitPrice = 0;
    double Unused = 0;
    double TradeProfitLoss = 0;
    double MaximumOpenPositionLoss = 0;
    double MaximumOpenPositionProfit = 0;
    double FlatToFlatMaximumOpenPositionProfit = 0;
    double FlatToFlatMaximumOpenPositionLoss = 0;
    double Commission = 0;
    int IsTradeClosed = 0;
    SCString Note;

    s_ACSTrade()
    {

    }

    void Clear()
    {
        *this = s_ACSTrade();
    }
};

```

```

/*****

```

```

struct s_SCOrderFillData
{
    int Version = CURRENT_SC_ORDER_FILL_DATA_VERSION;

    SCString Symbol;
    SCString TradeAccount;
    uint32_t InternalOrderID = 0;
    SCDateTimeMS FillDateTime = 0.0; // Adjusted to chart time zone
    BuySellEnum BuySell = BSE_UNDEFINED;
    double Quantity = 0;
    double FillPrice = 0.0;

    SCString FillExecutionServiceID;

    double TradePositionQuantity = 0;

    int IsSimulated = 0;

    SCString OrderActionSource;
    SCString Note;

```

```

void Clear()
{
    *this = s_SCOrderFillData();
}

bool operator < (const s_SCOrderFillData& RightValue) const
{
    return FillDateTime < RightValue.FillDateTime;
}
};

/*****
struct s_SCNewOrder
{
    int Version = CURRENT_SC_NEW_ORDER_VERSION;

    int OrderType = SCT_ORDERTYPE_MARKET;

    double OrderQuantity = 0;

    double Price1 = DBL_MAX;
    double Price2 = DBL_MAX;

    uint32_t InternalOrderID = 0;

    uint32_t InternalOrderID2 = 0;

    SCString TextTag;
    SCTimeInForceEnum TimeInForce = SCT_TIF_UNSET;

    double Target1Offset = 0.0;
    uint32_t Target1InternalOrderID = 0;
    double Stop1Offset = 0.0;
    uint32_t Stop1InternalOrderID = 0;
    double OCOGroup1Quantity = 0;

    //These depend upon the parent order price or the current price in the case of a market order
    double Target1Price = 0.0;
    double Stop1Price = 0.0;

    double Target2Offset = 0.0;
    uint32_t Target2InternalOrderID = 0;
    double Stop2Offset = 0.0;
    uint32_t Stop2InternalOrderID = 0;
    double OCOGroup2Quantity = 0;
    double Target2Price = 0.0;
    double Stop2Price = 0.0;

    double Target3Offset = 0.0;
    uint32_t Target3InternalOrderID = 0;
    double Stop3Offset = 0;
    uint32_t Stop3InternalOrderID = 0;
    double OCOGroup3Quantity = 0;
    double Target3Price = 0.0;
    double Stop3Price = 0.0;

    double Target4Offset = 0.0;
    uint32_t Target4InternalOrderID = 0;
    double Stop4Offset = 0.0;
    uint32_t Stop4InternalOrderID = 0;
    double OCOGroup4Quantity = 0;
    double Target4Price = 0.0;
    double Stop4Price = 0.0;

```

```

double Target5Offset = 0.0;
uint32_t Target5InternalOrderID = 0;
double Stop5Offset = 0.0;
uint32_t Stop5InternalOrderID = 0;
double OCOGroup5Quantity = 0;
double Target5Price = 0.0;
double Stop5Price = 0.0;

double StopAllOffset = 0.0;
uint32_t StopAllInternalOrderID = 0;
double StopAllPrice = 0.0;

double MaximumChaseAsPrice = 0.0;

char AttachedOrderTarget1Type = SCT_ORDERTYPE_LIMIT;
char AttachedOrderTarget2Type = SCT_ORDERTYPE_LIMIT;
char AttachedOrderTarget3Type = SCT_ORDERTYPE_LIMIT;
char AttachedOrderTarget4Type = SCT_ORDERTYPE_LIMIT;
char AttachedOrderTarget5Type = SCT_ORDERTYPE_LIMIT;

char AttachedOrderStop1Type = SCT_ORDERTYPE_STOP;
char AttachedOrderStop2Type = SCT_ORDERTYPE_STOP;
char AttachedOrderStop3Type = SCT_ORDERTYPE_STOP;
char AttachedOrderStop4Type = SCT_ORDERTYPE_STOP;
char AttachedOrderStop5Type = SCT_ORDERTYPE_STOP;
char AttachedOrderStopAllType = SCT_ORDERTYPE_STOP;

double AttachedOrderMaximumChase = 0.0;
double TrailStopStepPriceAmount = 0.0;
double TriggeredTrailStopTriggerPriceOffset = 0.0;
double TriggeredTrailStopTrailPriceOffset = 0.0;

struct s_MoveToBreakEven
{
    int Type = MOVETO_BE_ACTION_TYPE_NONE;
    int BreakEvenLevelOffsetInTicks = 0;
    int TriggerOffsetInTicks = 0;
    int TriggerOCOGroup = 0;
};

s_MoveToBreakEven MoveToBreakEven;

double StopLimitOrderLimitOffset = DBL_MAX;

uint32_t Target1InternalOrderID_2 = 0;
uint32_t Stop1InternalOrderID_2 = 0;
uint32_t Target2InternalOrderID_2 = 0;
uint32_t Stop2InternalOrderID_2 = 0;
uint32_t Target3InternalOrderID_2 = 0;
uint32_t Stop3InternalOrderID_2 = 0;
uint32_t Target4InternalOrderID_2 = 0;
uint32_t Stop4InternalOrderID_2 = 0;
uint32_t Target5InternalOrderID_2 = 0;
uint32_t Stop5InternalOrderID_2 = 0;
uint32_t StopAllInternalOrderID_2 = 0;

double Target1Offset_2 = 0.0;
double Target1Price_2 = 0.0;
double Stop1Offset_2 = 0.0;
double Stop1Price_2 = 0.0;

double Target2Offset_2 = 0.0;
double Target2Price_2 = 0.0;
double Stop2Offset_2 = 0.0;

```



```

double Stop2Price_2 = 0.0;

double Target3Offset_2 = 0.0;
double Target3Price_2 = 0.0;
double Stop3Offset_2 = 0.0;
double Stop3Price_2 = 0.0;

double Target4Offset_2 = 0.0;
double Target4Price_2 = 0.0;
double Stop4Offset_2 = 0.0;
double Stop4Price_2 = 0.0;

double Target5Offset_2 = 0.0;
double Target5Price_2 = 0.0;
double Stop5Offset_2 = 0.0;
double Stop5Price_2 = 0.0;

double StopAllOffset_2 = 0.0;
double StopAllPrice_2 = 0.0;

SCString Symbol;
SCString TradeAccount;

double StopLimitOrderLimitOffsetForAttachedOrders = DBL_MAX;

int32_t SubmitAsHeldOrder = 0;

double QuantityTriggeredAttachedStop_QuantityForTrigger = 0.0;

double AttachedOrderStop1_TriggeredTrailStopTriggerPriceOffset = 0.0;
double AttachedOrderStop1_TriggeredTrailStopTrailPriceOffset = 0.0;
double AttachedOrderStop2_TriggeredTrailStopTriggerPriceOffset = 0.0;
double AttachedOrderStop2_TriggeredTrailStopTrailPriceOffset = 0.0;
double AttachedOrderStop3_TriggeredTrailStopTriggerPriceOffset = 0.0;
double AttachedOrderStop3_TriggeredTrailStopTrailPriceOffset = 0.0;
double AttachedOrderStop4_TriggeredTrailStopTriggerPriceOffset = 0.0;
double AttachedOrderStop4_TriggeredTrailStopTrailPriceOffset = 0.0;
double AttachedOrderStop5_TriggeredTrailStopTriggerPriceOffset = 0.0;
double AttachedOrderStop5_TriggeredTrailStopTrailPriceOffset = 0.0;

double TriggeredLimitOrStopAttachedOrderTriggerOffset = 0.0;//As a price offset

s_MoveToBreakEven MoveToBreakEven_2;

s_SCNewOrder()
{
}

void Reset()
{
    *this = s_SCNewOrder();
}

bool IsPrice1Set() const
{
    return (Price1 != DBL_MAX);
}

bool IsPrice2Set() const
{
    return (Price2 != DBL_MAX);
}

bool AreAttachedOrderVariablesSet() const
{

```

```

    return Target1Offset != 0.0
        || Stop1Offset != 0.0
        || Target1Price != 0.0
        || Stop1Price != 0.0
        || StopAllOffset != 0.0;
}
};

/*****

struct s_SCTradeOrder
{
    int Version = 19; // No longer used

    uint32_t InternalOrderID = 0;
    SCString Symbol;
    SCString OrderType;
    double OrderQuantity = 0;
    SCString TextTag;

    double Price1 = 0.0;
    double Price2 = 0.0;
    double AvgFillPrice = 0.0;
    int SourceChartNumber = 0;

    SCOrderStatusCodeEnum OrderStatusCode = SCT_OSC_UNSPECIFIED;

    double FilledQuantity = 0;

    uint32_t ParentInternalOrderID = 0;
    uint32_t LinkID = 0;

    SCDateTime LastActivityTime;

    OpenCloseEnum OpenClose = OCE_UNDEFINED;
    BuySellEnum BuySell = BSE_UNDEFINED;

    SCDateTime EntryDateTime;

    int OrderTypeAsInt = 0;

    double LastModifyQuantity = 0;
    double LastModifyPrice1 = 0.0;

    double LastFillPrice = 0.0;
    double LastFillQuantity = 0;

    SCString SourceChartbookFileName;

    int IsSimulated = 0;

    uint32_t TargetChildInternalOrderID = 0;
    uint32_t StopChildInternalOrderID = 0;
    uint32_t OCOSiblingInternalOrderID = 0;

    int32_t EstimatedPositionInQueue = 0;

    int32_t TriggeredTrailingStopTriggerHit = 0;

    SCString LastOrderActionSource;

    // Structure version 19
    SCString TradeAccount;

    // version 2302

```

```

uint64_t TrueExchangeOrderID = 0;

// version 2347
SCTimeInForceEnum TimeInForce = SCT_TIF_UNSET;

s_SCTradeOrder()
{
}

bool IsWorking()
{
    return IsWorkingOrderStatus(OrderStatusCode);
}

bool IsLimitOrder()
{
    return (OrderTypeAsInt == SCT_ORDERTYPE_LIMIT
        || OrderTypeAsInt == SCT_ORDERTYPE_MARKET_IF_TOUCHED
        || OrderTypeAsInt == SCT_ORDERTYPE_LIMIT_CHASE
        || OrderTypeAsInt == SCT_ORDERTYPE_LIMIT_TOUCH_CHASE);
}

bool IsAttachedOrder()
{
    return ParentInternalOrderID != 0;
}
};

/*****/
struct s_SCPositionData
{
    static const int CURRENT_POSITION_DATA_VERSION = 12;

    int Version = CURRENT_POSITION_DATA_VERSION;

    SCString Symbol;
    double PositionQuantity = 0;
    double AveragePrice = 0;
    double OpenProfitLoss = 0;
    double CumulativeProfitLoss = 0;
    int WinTrades = 0;
    int LoseTrades = 0;
    int TotalTrades = 0;
    double MaximumOpenPositionLoss = 0;
    double MaximumOpenPositionProfit = 0;
    double LastTradeProfitLoss = 0;

    double PositionQuantityWithAllWorkingOrders = 0;
    double PositionQuantityWithExitWorkingOrders = 0;
    int WorkingOrdersExist = 0;

    double DailyProfitLoss = 0;

    SCDateTime LastFillDateTime;
    SCDateTime LastEntryDateTime;
    SCDateTime LastExitDateTime;

    double PriceHighDuringPosition = 0;
    double DailyTotalQuantityFilled = 0;
    double PriceLowDuringPosition = 0;
    double PriorPositionQuantity = 0;
    double PositionQuantityWithExitMarketOrders = 0;

    double TotalQuantityFilled = 0;

```

```

double LastTradeQuantity = 0;

double TradeStatsDailyProfitLoss = 0;
double TradeStatsDailyClosedQuantity = 0;
double TradeStatsOpenProfitLoss = 0;
double TradeStatsOpenQuantity = 0;

int NonAttachedWorkingOrdersExist = 0;

SCString TradeAccount;

double PositionQuantityWithMarketWorkingOrders = 0;
double PositionQuantityWithAllWorkingOrdersExceptNonMarketExits = 0;
double PositionQuantityWithAllWorkingOrdersExceptAllExits = 0;
double AllWorkingBuyOrdersQuantity = 0;
double AllWorkingSellOrdersQuantity = 0;

s_SCPositionData()
{
}

};

/*****/

struct s_GetOHLCOfTimePeriod
{
    s_GetOHLCOfTimePeriod(SCDateTime StartDateTime, SCDateTime EndDateTime, float& Open, float& High, float& Low, float& Close, float& NextOpen, int& NumberOfBars, SCDateTime& TotalTimeSpan)
    : m_Open(Open),
      m_High(High),
      m_Low(Low),
      m_Close(Close),
      m_NextOpen(NextOpen),
      m_NumberOfBars(NumberOfBars),
      m_TotalTimeSpan(TotalTimeSpan)
    {
        m_StartDateTime = StartDateTime;
        m_EndDateTime = EndDateTime;
    }

    SCDateTime m_StartDateTime;
    SCDateTime m_EndDateTime;
    float& m_Open;
    float& m_High;
    float& m_Low;
    float& m_Close;
    float& m_NextOpen;
    int& m_NumberOfBars;
    SCDateTime& m_TotalTimeSpan;
};

/*****/

typedef void (SCDLLCALL* fp_SCDLLOnArrayUsed)(int);

template <typename T> class c_ArrayWrapper;
template <typename T> class c_ConstArrayWrapper;

template <typename T>
class c_ArrayWrapper
{
private:
    T* m_Data;
    int m_NumElements;

```

```

fp_SCDLLOnArrayUsed m_fp_OnArrayUsed;
int m_OnArrayUsedParam;

int m_NumExtendedElements;
int m_TotalNumElements;

T m_DefaultElement;

public:
    c_ArrayWrapper()
    {
        ResetMembers();
    }
    explicit c_ArrayWrapper(int i)
    {
        ResetMembers();
    }
    c_ArrayWrapper(const c_ArrayWrapper& Src)
        : m_Data(Src.m_Data)
        , m_NumElements(Src.m_NumElements)
        , m_NumExtendedElements(Src.m_NumExtendedElements)
        , m_TotalNumElements(Src.m_TotalNumElements)
        , m_fp_OnArrayUsed(Src.m_fp_OnArrayUsed)
        , m_OnArrayUsedParam(Src.m_OnArrayUsedParam)
        , m_DefaultElement(0)
    {
    }

    const c_ArrayWrapper& operator = (const c_ArrayWrapper& Src)
    {
        m_Data = Src.m_Data;
        m_NumElements = Src.m_NumElements;
        m_NumExtendedElements = Src.m_NumExtendedElements;
        m_TotalNumElements = Src.m_TotalNumElements;
        m_fp_OnArrayUsed = Src.m_fp_OnArrayUsed;
        m_OnArrayUsedParam = Src.m_OnArrayUsedParam;
        m_DefaultElement = Src.m_DefaultElement;

        return *this;
    }

    void ResetMembers()
    {
        m_Data = NULL;
        m_NumElements = 0;
        m_NumExtendedElements = 0;
        m_TotalNumElements = 0;
        m_fp_OnArrayUsed = NULL;
        m_OnArrayUsedParam = 0;
        m_DefaultElement = T(0);
    }

    void AllocateArray()
    {
        if (m_Data == NULL)
        {
            if (m_fp_OnArrayUsed != NULL)
            {
                // This will allocate and set up the array if it can
                m_fp_OnArrayUsed(m_OnArrayUsedParam);
            }
        }
    }

```

```

}

T& operator [] (int Index)
{
    return ElementAt(Index);
}

const T& operator [] (int Index) const
{
    return ElementAt(Index);
}

T& ElementAt(int Index)
{
    if (m_Data == NULL)
    {
        if (m_fp_OnArrayUsed != NULL)
        {
            // This will allocate and set up the array if it can
            m_fp_OnArrayUsed(m_OnArrayUsedParam);
        }

        if (m_Data == NULL)
            return m_DefaultElement;
    }

    if (m_TotalNumElements == 0)
        return m_DefaultElement;

    if (Index < 0)
        Index = 0;

    if (Index >= m_TotalNumElements)
        Index = m_TotalNumElements - 1;

    return m_Data[Index];
}

const T& ElementAt(int Index) const
{
    if (m_Data == NULL)
    {
        if (m_fp_OnArrayUsed != NULL)
        {
            // This will allocate and set up the array if it can
            m_fp_OnArrayUsed(m_OnArrayUsedParam);
        }

        if (m_Data == NULL)
            return m_DefaultElement;
    }

    if (m_TotalNumElements == 0)
        return m_DefaultElement;

    if (Index < 0)
        Index = 0;

    if (Index >= m_TotalNumElements)
        Index = m_TotalNumElements - 1;

    return m_Data[Index];
}

T& GetAt(int Index)

```

```

{
    return m_Data[Index];
}

const T& GetAt(int Index) const
{
    return m_Data[Index];
}

// For Sierra Chart internal use only.
void InternalSetArray(T* DataPointer, int Size, int NumExtendedElements = 0)
{
    m_NumElements = Size;
    m_NumExtendedElements = NumExtendedElements;

    if (m_NumElements == 0)
        m_TotalNumElements = 0;
    else
        m_TotalNumElements = m_NumElements + m_NumExtendedElements;

    m_Data = DataPointer;
}

// For Sierra Chart internal use only
void InternalSetOnUsed(fp_SCDLLOnArrayUsed OnArrayUsed, int Param)
{
    m_fp_OnArrayUsed = OnArrayUsed;
    m_OnArrayUsedParam = Param;
}

int GetArraySize() const
{
    return m_NumElements;
}

int GetExtendedArraySize() const
{
    return m_TotalNumElements; // m_NumElements + m_NumExtendedElements;
}

// This should only be used when absolutely necessary and you know what
// you are doing with it.
T* GetPointer()
{
    //Returns non-constant data pointer which can be used for rapidly setting blocks of data
    return m_Data;
}

const T* GetPointer() const
{
    return m_Data;
}

friend class c_ConstArrayWrapper<T>;
};

template <typename T>
class c_ConstArrayWrapper
{
private:
    const T* m_Data;
    int m_NumElements;

    fp_SCDLLOnArrayUsed m_OnArrayUsed;
    int m_OnArrayUsedParam;

```

```

int m_NumExtendedElements;
int m_TotalNumElements;

T m_DefaultElement;

public:
    c_ConstArrayWrapper()
        : m_Data(NULL)
        , m_NumElements(0)
        , m_NumExtendedElements(0)
        , m_TotalNumElements(0)
        , m_OnArrayUsed(NULL)
        , m_OnArrayUsedParam(0)
        , m_DefaultElement(0)
    {
    }

    explicit c_ConstArrayWrapper(int i)
        : m_Data(NULL)
        , m_NumElements(0)
        , m_NumExtendedElements(0)
        , m_TotalNumElements(0)
        , m_OnArrayUsed(NULL)
        , m_OnArrayUsedParam(0)
        , m_DefaultElement(0)
    {
    }

    c_ConstArrayWrapper(const c_ConstArrayWrapper& Src)
        : m_Data(Src.m_Data)
        , m_NumElements(Src.m_NumElements)
        , m_NumExtendedElements(Src.m_NumExtendedElements)
        , m_TotalNumElements(Src.m_TotalNumElements)
        , m_OnArrayUsed(Src.m_OnArrayUsed)
        , m_OnArrayUsedParam(Src.m_OnArrayUsedParam)
        , m_DefaultElement(0)
    {
    }

    c_ConstArrayWrapper(const c_ArrayWrapper<T>& Src)
        : m_Data(Src.m_Data)
        , m_NumElements(Src.m_NumElements)
        , m_NumExtendedElements(Src.m_NumExtendedElements)
        , m_TotalNumElements(Src.m_TotalNumElements)
        , m_OnArrayUsed(Src.m_OnArrayUsed)
        , m_OnArrayUsedParam(Src.m_OnArrayUsedParam)
        , m_DefaultElement(0)
    {
    }

    ~c_ConstArrayWrapper()
    {
    }

    const c_ConstArrayWrapper& operator = (const c_ConstArrayWrapper& Src)
    {
        m_Data = Src.m_Data;
        m_NumElements = Src.m_NumElements;
        m_NumExtendedElements = Src.m_NumExtendedElements;
        m_TotalNumElements = Src.m_TotalNumElements;
        m_OnArrayUsed = Src.m_OnArrayUsed;
        m_OnArrayUsedParam = Src.m_OnArrayUsedParam;
        m_DefaultElement = Src.m_DefaultElement;
    }

```



```

    return *this;
}
const c_ConstArrayWrapper& operator = (const c_ArrayWrapper<T>& Src)
{
    m_Data = Src.m_Data;
    m_NumElements = Src.m_NumElements;
    m_NumExtendedElements = Src.m_NumExtendedElements;
    m_TotalNumElements = Src.m_TotalNumElements;
    m_OnArrayUsed = Src.m_OnArrayUsed;
    m_OnArrayUsedParam = Src.m_OnArrayUsedParam;
    m_DefaultElement = Src.m_DefaultElement;

    return *this;
}

const T& operator [] (int Index) const
{
    return ElementAt(Index);
}

const T& ElementAt(int Index) const
{
    if (m_Data == NULL)
    {
        if (m_OnArrayUsed != NULL)
            m_OnArrayUsed(m_OnArrayUsedParam); // This will allocate and set up the array if it can

        if (m_Data == NULL)
            return m_DefaultElement;
    }

    if (m_TotalNumElements == 0)
        return m_DefaultElement;

    if (Index < 0)
        Index = 0;

    if (Index >= m_TotalNumElements)
        Index = m_TotalNumElements - 1;

    return m_Data[Index];
}

// For Sierra Chart internal use only
void InternalSetArray(const T* DataPointer, int Size, int NumExtendedElements = 0)
{
    m_NumElements = Size;
    m_NumExtendedElements = NumExtendedElements;

    if (m_NumElements == 0)
        m_TotalNumElements = 0;
    else
        m_TotalNumElements = m_NumElements + m_NumExtendedElements;

    m_Data = DataPointer;
}

// For Sierra Chart internal use only
void InternalSetOnUsed(fp_SCDLLOnArrayUsed OnArrayUsed, int Param)
{
    m_OnArrayUsed = OnArrayUsed;
    m_OnArrayUsedParam = Param;
}

```

```

int GetArraySize() const
{
    return m_NumElements;
}

int GetExtendedArraySize() const
{
    return m_TotalNumElements; //m_NumElements + m_NumExtendedElements;
}

// This should only be used when absolutely necessary and you know what
// you are doing with it.
const T* GetPointer() const
{
    return m_Data;
}

friend class c_ArrayWrapper<T>;
};

/*****

class SCDatetimeArray
: public c_ArrayWrapper<SCDateTime>
{
public:
    /*=====
    Days since 1899-12-30.
    -----*/
    int DateAt(int Index) const
    {
        return ElementAt(Index).GetDate();
    }
    /*=====
    Seconds since midnight.
    -----*/
    int TimeAt(int Index) const
    {
        return ElementAt(Index).GetTimeInSeconds();
    }
    /*=====
    Returns the date that was set.
    -----*/
    int SetDateAt(int Index, int Date)
    {
        ElementAt(Index) = SCDatetime(Date, TimeAt(Index));
        return Date;
    }
    /*=====
    Returns the time that was set.
    -----*/
    int SetTimeAt(int Index, int Time)
    {
        ElementAt(Index) = SCDatetime(DateAt(Index), Time);
        return Time;
    }
};

typedef SCDatetimeArray &SCDatetimeArrayRef;
typedef c_ArrayWrapper<uint16_t> SCUShortArray;
typedef c_ArrayWrapper<unsigned int> SCUIntArray;
typedef c_ArrayWrapper<unsigned int>& SCUIntArrayRef;
typedef c_ArrayWrapper<SCUIntArray> SCUIntArrayArray;
typedef c_ArrayWrapper<float> SCFloatArray;

```

```

typedef c_ArrayWrapper<uint32_t> SCColorArray;
typedef c_ArrayWrapper<uint32_t>& SCColorArrayRef;
typedef c_ConstArrayWrapper<uint16_t> SCConstUShortArray;
typedef c_ConstArrayWrapper<char> SCConstCharArray;

typedef c_ArrayWrapper<float> SCConstFloatArray;

typedef c_ArrayWrapper<SCFloatArray>& SCBaseDataRef;
typedef c_ArrayWrapper<float>& SCFloatArrayRef;
typedef c_ArrayWrapper<float>& SCConstFloatArrayRef;
typedef const c_ArrayWrapper<float>& SCFloatArrayInRef;

typedef c_ArrayWrapper<SCConstFloatArray> SCConstFloatArrayArray;
typedef c_ArrayWrapper<SCFloatArray> SCFloatArrayArray;

typedef c_ArrayWrapper<SCString> SCStringArray;

class SCGraphData
: public SCFloatArrayArray
{
private:
    SCFloatArray BaseData[SC_SUBGRAPHS_AVAILABLE];

public:
    SCGraphData()
    {
        InternalSetArray(BaseData, SC_SUBGRAPHS_AVAILABLE);
    }

    SCFloatArrayRef InternalAccessBaseDataArray(int Index)
    {
        if (Index < 0)
            return BaseData[0];

        if (Index >= SC_SUBGRAPHS_AVAILABLE)
            return BaseData[SC_SUBGRAPHS_AVAILABLE - 1];

        return BaseData[Index];
    }
};

/*****
struct s_SCSubgraph_260
{
    SCString Name;

    unsigned int PrimaryColor = 0;
    unsigned int SecondaryColor = 0;
    unsigned int SecondaryColorUsed = 0; // boolean

    uint16_t DrawStyle = 0;
    SubgraphLineStyles LineStyle = LINESTYLE_SOLID;
    uint16_t LineWidth = 0;
    unsigned char UseLabelsColor = 0; // Line label Name/Value
    unsigned char DisplayNameValueInDataLine = 0;

    SCFloatArray Data; // Array of data values
    SCColorArray DataColor; // Array of colors for each of the data elements

    SCFloatArrayArray Arrays; // Array of extra arrays

    unsigned char IncludeInStudySummary = 0;
    unsigned char UseStudySummaryCellBackgroundColor = 0;
    unsigned char AutoColoring = 0; // boolean

```

```

int GraphicalDisplacement = 0;

int DrawZeros = 0; // boolean

// This controls whether the Subgraph name and value are displayed in the Values windows and on the Region Data
Line.
uint16_t DisplayNameValueInWindowsFlags = 0;

unsigned int LineLabel = 0; // Line label Name/Value
int ExtendedArrayElementsToGraph = 0;

SCString TextDrawStyleText;

float GradientAngleUnit = 0;
float GradientAngleMax = 0;

SCString ShortName;
uint32_t StudySummaryCellBackgroundColor = 0;
SCString StudySummaryCellText;
uint32_t LabelsColor = 0;
unsigned char IncludeInSpreadsheet = 0;
unsigned char UseTransparentLabelBackground = 0;
char Reserve[40] = {};

s_SCSubgraph_260()
{
}

// This constructor is so this struct can work as an c_ArrayWrapper element
explicit s_SCSubgraph_260(int Integer)
{
}

float& operator [] (int Index)
{
    return Data[Index];
}

operator SCFloatArray& ()
{
    return Data;
}
};

typedef c_ArrayWrapper<s_SCSubgraph_260> SCSubgraph260Array;
typedef s_SCSubgraph_260& SCSubgraph260Ref;
typedef s_SCSubgraph_260& SCSubgraphRef;

/*****
struct s_ChartStudySubgraphValues
{
    int ChartNumber = 0;
    int StudyID = 0;
    int SubgraphIndex = 0;
};

struct s_SCInput_145
{
    SCString Name;

    unsigned char ValueType = 0;
    unsigned char Unused1 = 0;
    uint16_t DisplayOrder = 0;
#pragma pack(push, 4) // This is necessary otherwise data items larger than 4 bytes will throw off the alignment and

```

size of the union

```
union
{
    unsigned int IndexValue;
    float FloatValue;
    unsigned int BooleanValue;
    struct
    {
        double DateTimeValue; // SCDateTime
        unsigned int TimeZoneValue;
    };

    int IntValue;
    unsigned int ColorValue;
    s_ChartStudySubgraphValues ChartStudySubgraphValues;
    double DoubleValue;
    struct
    {
        bool StringModified;
        const char* StringValue;
    };

    unsigned char ValueBytes[16] = {};
};

union
{
    float FloatValueMin;
    double DateTimeValueMin; // SCDateTime
    int IntValueMin;
    double DoubleValueMin;

    unsigned char ValueBytesMin[16] = {};
};

union
{
    float FloatValueMax;
    double DateTimeValueMax; // SCDateTime
    int IntValueMax;
    double DoubleValueMax;

    unsigned char ValueBytesMax[16] = {};
};
#pragma pack(pop)

// Do not use or modify these. For internal use only
int InputIndex = 0;
void (SCDLLCALL* InternalSetCustomStrings)(int InputIndex, const char* CustomStrings) = NULL;
const char* SelectedCustomInputString = NULL;

void (SCDLLCALL* InternalSetDescription)(int InputIndex, const char* Description) = NULL;

char Reserve[48] = {};

s_SCInput_145()
{
}

explicit s_SCInput_145(int Integer)
{
}

void Clear()
{
}
```

```

    *this = s_SCInput_145();
}

bool UsesStringValue() const
{
    return ValueType == STRING_VALUE
        || ValueType == PATH_AND_FILE_NAME_VALUE
        || ValueType == FIND_SYMBOL_VALUE;
}

unsigned int GetIndex() const
{
    switch (ValueType)
    {
        case OHLC_VALUE: // IndexValue
        case STUDYINDEX_VALUE: // IndexValue
        case SUBGRAPHINDEX_VALUE: // IndexValue
        case MOVAVGTYPE_VALUE: // IndexValue
        case TIME_PERIOD_LENGTH_UNIT_VALUE:
        case STUDYID_VALUE: // IndexValue
        case CANDLESTICK_PATTERNS_VALUE:
        case CUSTOM_STRING_VALUE:
        case TIMEZONE_VALUE: // IndexValue
        case ALERT_SOUND_NUMBER_VALUE:
            return IndexValue;

        case FLOAT_VALUE: // FloatValue
            return static_cast<unsigned int>((FloatValue < 0.0f) ? (FloatValue - 0.5f) : (FloatValue + 0.5f));

        case YESNO_VALUE: // BooleanValue
            return (BooleanValue != 0) ? 1 : 0;

        case DATE_VALUE: // DateTimeValue
        case TIME_VALUE: // DateTimeValue
        case DATETIME_VALUE: // DateTimeValue
            return 0;

        case INT_VALUE: // IntValue
        case CHART_NUMBER:
            return static_cast<unsigned int>(IntValue);

        case COLOR_VALUE: // ColorValue
            return static_cast<unsigned int>(ColorValue);

        case CHART_STUDY_SUBGRAPH_VALUES:
        case STUDY_SUBGRAPH_VALUES:
        case CHART_STUDY_VALUES:
            return 0;

        case DOUBLE_VALUE: // DoubleValue
            return static_cast<unsigned int>((DoubleValue < 0.0) ? (DoubleValue - 0.5f) : (DoubleValue + 0.5f));

        case TIME_WITH_TIMEZONE_VALUE:
            return TimeZoneValue;

        case STRING_VALUE:
        case PATH_AND_FILE_NAME_VALUE:
        case FIND_SYMBOL_VALUE:
            return 0;

        default:
        case NO_VALUE:
            return 0;
    }
}

```

```

float GetFloat() const
{
    switch (ValueType)
    {
        case OHLC_VALUE: // IndexValue
        case STUDYINDEX_VALUE: // IndexValue
        case SUBGRAPHINDEX_VALUE: // IndexValue
        case MOVAVGTYPE_VALUE: // IndexValue
        case TIME_PERIOD_LENGTH_UNIT_VALUE:
        case STUDYID_VALUE: // IndexValue
        case CANDLESTICK_PATTERNS_VALUE:
        case CUSTOM_STRING_VALUE:
        case TIMEZONE_VALUE: // IndexValue
        case ALERT_SOUND_NUMBER_VALUE:
        return static_cast<float>(IndexValue);

        case FLOAT_VALUE: // FloatValue
        return FloatValue;

        case YESNO_VALUE: // BooleanValue
        return (BooleanValue != 0) ? 1.0f : 0.0f;

        case DATE_VALUE: // DateTimeValue
        case TIME_VALUE: // DateTimeValue
        case DATETIME_VALUE: // DateTimeValue
        return 0.0f;

        case INT_VALUE: // IntValue
        case CHART_NUMBER:
        return static_cast<float>(IntValue);

        case COLOR_VALUE: // ColorValue
        return 0.0f;

        case CHART_STUDY_SUBGRAPH_VALUES:
        case STUDY_SUBGRAPH_VALUES:
        case CHART_STUDY_VALUES:
        return 0.0f;

        case DOUBLE_VALUE: // DoubleValue
        return static_cast<float>(DoubleValue);

        case TIME_WITH_TIMEZONE_VALUE:
        return 0.0f;

        case STRING_VALUE:
        case PATH_AND_FILE_NAME_VALUE:
        case FIND_SYMBOL_VALUE:
        return 0.0f;

        default:
        case NO_VALUE:
        return 0.0f;
    }
}

```

```

unsigned int GetBoolean() const
{
    switch (ValueType)
    {
        case OHLC_VALUE: // IndexValue
        case STUDYINDEX_VALUE: // IndexValue
        case SUBGRAPHINDEX_VALUE: // IndexValue
        case MOVAVGTYPE_VALUE: // IndexValue

```

```

    case TIME_PERIOD_LENGTH_UNIT_VALUE:
    case STUDYID_VALUE: // IndexValue
    case CANDLESTICK_PATTERNS_VALUE:
    case CUSTOM_STRING_VALUE:
    case TIMEZONE_VALUE: // IndexValue
    case ALERT_SOUND_NUMBER_VALUE:
    return (IndexValue != 0) ? 1 : 0;

    case FLOAT_VALUE: // FloatValue
    return (FloatValue != 0.0f) ? 1 : 0;

    case YESNO_VALUE: // BooleanValue
    return (BooleanValue != 0) ? 1 : 0;

    case DATE_VALUE: // DateTimeValue
    case TIME_VALUE: // DateTimeValue
    case DATETIME_VALUE: // DateTimeValue
    return (DateTimeValue != 0) ? 1 : 0;

    case INT_VALUE: // IntValue
    case CHART_NUMBER:
    return (IntValue != 0) ? 1 : 0;

    case COLOR_VALUE: // ColorValue
    return (ColorValue != 0) ? 1 : 0;

    case CHART_STUDY_SUBGRAPH_VALUES:
    case STUDY_SUBGRAPH_VALUES:
    case CHART_STUDY_VALUES:
    return 0;

    case DOUBLE_VALUE: // DoubleValue
    return (DoubleValue != 0.0) ? 1 : 0;

    case TIME_WITH_TIMEZONE_VALUE:
    return (DateTimeValue != 0) ? 1 : 0;

    case STRING_VALUE:
    case PATH_AND_FILE_NAME_VALUE:
    case FIND_SYMBOL_VALUE:
    return 0;

    default:
    case NO_VALUE:
    return 0;
}
}

SCDateTime GetDateTime() const
{
    switch (ValueType)
    {
        case OHLC_VALUE: // IndexValue
        case STUDYINDEX_VALUE: // IndexValue
        case SUBGRAPHINDEX_VALUE: // IndexValue
        case MOVAVGTYPE_VALUE: // IndexValue
        case TIME_PERIOD_LENGTH_UNIT_VALUE:
        case STUDYID_VALUE: // IndexValue
        case CANDLESTICK_PATTERNS_VALUE:
        case CUSTOM_STRING_VALUE:
        case TIMEZONE_VALUE: // IndexValue
        case ALERT_SOUND_NUMBER_VALUE:
        return 0.0;

        case FLOAT_VALUE: // FloatValue

```



```

return 0.0;

case YESNO_VALUE: // BooleanValue
return 0.0;

case DATE_VALUE: // DateTimeValue
case TIME_VALUE: // DateTimeValue
case DATETIME_VALUE: // DateTimeValue
return DateTimeValue;

case INT_VALUE: // IntValue
case CHART_NUMBER:
return 0.0;

case COLOR_VALUE: // ColorValue
return 0.0;

case CHART_STUDY_SUBGRAPH_VALUES:
case STUDY_SUBGRAPH_VALUES:
case CHART_STUDY_VALUES:
return 0.0;

case DOUBLE_VALUE: // DoubleValue
return 0.0;

case TIME_WITH_TIMEZONE_VALUE:
return DateTimeValue;

case STRING_VALUE:
case PATH_AND_FILE_NAME_VALUE:
case FIND_SYMBOL_VALUE:
return 0.0;

default:
case NO_VALUE:
return 0.0;
}
}

int GetInt() const
{
switch (ValueType)
{
case OHLC_VALUE: // IndexValue
case STUDYINDEX_VALUE: // IndexValue
case SUBGRAPHINDEX_VALUE: // IndexValue
case MOVAVGTYPE_VALUE: // IndexValue
case TIME_PERIOD_LENGTH_UNIT_VALUE:
case STUDYID_VALUE: // IndexValue
case CANDLESTICK_PATTERNS_VALUE:
case CUSTOM_STRING_VALUE:
case TIMEZONE_VALUE: // IndexValue
case ALERT_SOUND_NUMBER_VALUE:
return static_cast<int>(IndexValue);

case FLOAT_VALUE: // FloatValue
return static_cast<int>((FloatValue < 0.0f) ? (FloatValue - 0.5f) : (FloatValue + 0.5f));

case YESNO_VALUE: // BooleanValue
return (BooleanValue != 0) ? 1 : 0;

case DATE_VALUE: // DateTimeValue
return SCDateTime(DateTimeValue).GetDate();

case TIME_VALUE: // DateTimeValue

```

```

return SCDateTime(DateTimeValue).GetTimeInSeconds();

case DATETIME_VALUE: // DateTimeValue
return 0;

case INT_VALUE: // IntValue
case CHART_NUMBER:
return IntValue;

case COLOR_VALUE: // ColorValue
return static_cast<int>(ColorValue);

case CHART_STUDY_SUBGRAPH_VALUES:
case STUDY_SUBGRAPH_VALUES:
case CHART_STUDY_VALUES:
return 0;

case DOUBLE_VALUE: // DoubleValue
return static_cast<int>((DoubleValue < 0.0) ? (DoubleValue - 0.5) : (DoubleValue + 0.5));

case TIME_WITH_TIMEZONE_VALUE:
return SCDateTime(DateTimeValue).GetTimeInSeconds();

case STRING_VALUE:
case PATH_AND_FILE_NAME_VALUE:
case FIND_SYMBOL_VALUE:
return 0;

default:
case NO_VALUE:
return 0;
}
}

unsigned int GetColor() const
{
switch (ValueType)
{
case OHLC_VALUE: // IndexValue
case STUDYINDEX_VALUE: // IndexValue
case SUBGRAPHINDEX_VALUE: // IndexValue
case MOVAVGTYPE_VALUE: // IndexValue
case TIME_PERIOD_LENGTH_UNIT_VALUE:
case STUDYID_VALUE: // IndexValue
case CANDLESTICK_PATTERNS_VALUE:
case CUSTOM_STRING_VALUE:
case TIMEZONE_VALUE: // IndexValue
case ALERT_SOUND_NUMBER_VALUE:
return static_cast<unsigned int>(IndexValue);

case FLOAT_VALUE: // FloatValue
return 0;

case YESNO_VALUE: // BooleanValue
return (BooleanValue != 0) ? RGB(255,255,255) : 0;

case DATE_VALUE: // DateTimeValue
case TIME_VALUE: // DateTimeValue
case DATETIME_VALUE: // DateTimeValue
return 0;

case INT_VALUE: // IntValue
case CHART_NUMBER:
return static_cast<unsigned int>(IntValue);

```

```

    case COLOR_VALUE: // ColorValue
    return ColorValue;

    case CHART_STUDY_SUBGRAPH_VALUES:
    case STUDY_SUBGRAPH_VALUES:
    case CHART_STUDY_VALUES:
    return 0;

    case DOUBLE_VALUE: // DoubleValue
    return 0;

    case TIME_WITH_TIMEZONE_VALUE:
    return 0;

    case STRING_VALUE:
    case PATH_AND_FILE_NAME_VALUE:
    case FIND_SYMBOL_VALUE:
    return 0;

    default:
    case NO_VALUE:
    return 0;
}
}

double GetDouble() const
{
    switch (ValueType)
    {
        case OHLC_VALUE: // IndexValue
        case STUDYINDEX_VALUE: // IndexValue
        case SUBGRAPHINDEX_VALUE: // IndexValue
        case MOVAVGTYPE_VALUE: // IndexValue
        case TIME_PERIOD_LENGTH_UNIT_VALUE:
        case STUDYID_VALUE: // IndexValue
        case CANDLESTICK_PATTERNS_VALUE:
        case CUSTOM_STRING_VALUE:
        case TIMEZONE_VALUE: // IndexValue
        case ALERT_SOUND_NUMBER_VALUE:
        return static_cast<double>(IndexValue);

        case FLOAT_VALUE: // FloatValue
        return static_cast<double>(FloatValue);

        case YESNO_VALUE: // BooleanValue
        return (BooleanValue != 0) ? 1.0 : 0.0;

        case DATE_VALUE: // DateTimeValue
        case TIME_VALUE: // DateTimeValue
        case DATETIME_VALUE: // DateTimeValue
        return DateTimeValue;

        case INT_VALUE: // IntValue
        case CHART_NUMBER:
        return static_cast<double>(IntValue);

        case COLOR_VALUE: // ColorValue
        return 0.0;

        case CHART_STUDY_SUBGRAPH_VALUES:
        case STUDY_SUBGRAPH_VALUES:
        case CHART_STUDY_VALUES:
        return 0.0;

        case DOUBLE_VALUE: // DoubleValue

```

```

    return DoubleValue;

    case TIME_WITH_TIMEZONE_VALUE:
    return DateTimeValue;

    case STRING_VALUE:
    case PATH_AND_FILE_NAME_VALUE:
    case FIND_SYMBOL_VALUE:
    return 0.0;

    default:
    case NO_VALUE:
    return 0.0;
}
}

unsigned int GetInputDataIndex() const
{
    unsigned int InputDataIndex = GetIndex();

    if (InputDataIndex > SC_SUBGRAPHS_AVAILABLE - 1)
        InputDataIndex = SC_SUBGRAPHS_AVAILABLE - 1;

    return InputDataIndex;
}

unsigned int GetStudyIndex() const
{
    return GetIndex();
}

unsigned int GetSubgraphIndex() const
{
    unsigned int SubgraphIndex;
    if (ValueType == CHART_STUDY_SUBGRAPH_VALUES
        || ValueType == STUDY_SUBGRAPH_VALUES)
    {
        SubgraphIndex = static_cast<unsigned int>(ChartStudySubgraphValues.SubgraphIndex);
    }
    else
        SubgraphIndex = GetIndex();

    if (SubgraphIndex > SC_SUBGRAPHS_AVAILABLE - 1)
        SubgraphIndex = SC_SUBGRAPHS_AVAILABLE - 1;

    return SubgraphIndex;
}

unsigned int GetMovAvgType() const
{
    unsigned int MovAvgTypeValue = GetIndex();
    if (MovAvgTypeValue >= MOVAVGTYPE_NUMBER_OF_AVERAGES)
        MovAvgTypeValue = 0;
    return MovAvgTypeValue;
}

unsigned int GetTimePeriodType() const
{
    return GetIndex();
}

unsigned int GetAlertSoundNumber() const
{
    return GetIndex();
}

```

```

unsigned int GetCandleStickPatternIndex() const
{
    return GetIndex();
}

unsigned int GetStudyID() const
{
    if (ValueType == CHART_STUDY_SUBGRAPH_VALUES
        || ValueType == STUDY_SUBGRAPH_VALUES
        || ValueType == CHART_STUDY_VALUES)
    {
        return static_cast<unsigned int>(ChartStudySubgraphValues.StudyID);
    }
    else
        return GetIndex();
}

unsigned int GetYesNo() const
{
    return GetBoolean();
}

int GetDate() const
{
    return GetDateTime().GetDate();
}

int GetTime() const
{
    return GetDateTime().GetTimeInSeconds();
}

int GetTimeInMilliseconds() const
{
    return GetDateTime().GetTimeInMilliseconds();
}

void GetChartStudySubgraphValues(int& ChartNumber, int& StudyID, int& SubgraphIndex) const
{
    ChartNumber = ChartStudySubgraphValues.ChartNumber;
    StudyID = ChartStudySubgraphValues.StudyID;
    SubgraphIndex = ChartStudySubgraphValues.SubgraphIndex;
}

s_ChartStudySubgraphValues GetChartStudySubgraphValues() const
{
    return ChartStudySubgraphValues;
}

int GetChartNumber()
{
    if (ValueType == CHART_STUDY_SUBGRAPH_VALUES
        || ValueType == CHART_STUDY_VALUES)
    {
        return ChartStudySubgraphValues.ChartNumber;
    }
    else if (ValueType == CHART_NUMBER)
        return IntValue;
    else
        return GetInt();
}

const SCString GetSelectedCustomString()
{

```

```

    SCString TempString;
    if(SelectedCustomInputString != NULL)
        TempString = SelectedCustomInputString;

    return TempString;
}

TimeZonesEnum GetTimeZone()
{
    unsigned int TimeZone = TIMEZONE_NOT_SPECIFIED;

    if (ValueType == TIMEZONE_VALUE)
        TimeZone = GetIndex();
    else if (ValueType == TIME_WITH_TIMEZONE_VALUE)
        TimeZone = TimeZoneValue;

    if (TimeZone >= NUM_TIME_ZONES)
        TimeZone = TIMEZONE_NOT_SPECIFIED;

    return static_cast<TimeZonesEnum>(TimeZone);
}

const char* GetString() const
{
    if (UsesStringValue() && StringValue != NULL)
        return StringValue;

    return "Unset";
}

const char* GetPathAndFileName()
{
    if (UsesStringValue() && StringValue != NULL)
        return StringValue;

    return "";
}

const char* GetSymbol()
{
    if (UsesStringValue() && StringValue != NULL)
        return StringValue;

    return "";
}

s_SCInput_145& SetInputDataIndex(unsigned int Value)
{
    ValueType = OHLC_VALUE;
    IndexValue = Value;
    return *this;
}

s_SCInput_145& SetStudyIndex(unsigned int Value)
{
    ValueType = STUDYINDEX_VALUE;
    IndexValue = Value;
    return *this;
}

s_SCInput_145& SetSubgraphIndex(unsigned int Value)
{
    ValueType = SUBGRAPHINDEX_VALUE;
    IndexValue = Value;
    return *this;
}

```

```

}

s_SCInput_145& SetMovAvgType(unsigned int Value)
{
    ValueType = MOVAVGTYPE_VALUE;
    IndexValue = Value;
    return *this;
}

s_SCInput_145& SetTimePeriodType(unsigned int Value)
{
    ValueType = TIME_PERIOD_LENGTH_UNIT_VALUE;
    IndexValue = Value;
    return *this;
}

s_SCInput_145& SetFloat(float Value)
{
    ValueType = FLOAT_VALUE;
    FloatValue = Value;
    return *this;
}

s_SCInput_145& SetYesNo(unsigned int Value)
{
    ValueType = YESNO_VALUE;
    BooleanValue = (Value != 0)?1:0;
    return *this;
}

s_SCInput_145& SetDate(int Value)
{
    ValueType = DATE_VALUE;
    SCDateTime DateValue(Value, 0);
    DateTimeValue = DateValue.GetAsDouble();
    return *this;
}

s_SCInput_145& SetTime(int Value)
{
    ValueType = TIME_VALUE;
    SCDateTime TimeValue(0, Value);
    DateTimeValue = TimeValue.GetAsDouble();

    return *this;
}

s_SCInput_145& SetTimeAsSCDateTime(const SCDateTime& Time)
{
    ValueType = TIME_VALUE;
    DateTimeValue = Time.GetTimeAsDouble();
    return *this;
}

s_SCInput_145& SetDateTime(const SCDateTime& Value)
{
    ValueType = DATETIME_VALUE;
    DateTimeValue = Value.GetAsDouble();
    return *this;
}

s_SCInput_145& SetInt(int Value)
{
    ValueType = INT_VALUE;
    IntValue = Value;
}

```

```

    return *this;
}

s_SCInput_145& SetIntWithoutTypeChange(int Value)
{
    switch (ValueType)
    {
        case OHLC_VALUE: // IndexValue
        case STUDYINDEX_VALUE: // IndexValue
        case SUBGRAPHINDEX_VALUE: // IndexValue
        case MOVAVGTYPE_VALUE: // IndexValue
        case TIME_PERIOD_LENGTH_UNIT_VALUE:
        case STUDYID_VALUE: // IndexValue
        case CANDLESTICK_PATTERNS_VALUE:
        case CUSTOM_STRING_VALUE:
        case TIMEZONE_VALUE: // IndexValue
        case ALERT_SOUND_NUMBER_VALUE:
            IndexValue = Value;
            break;

        case FLOAT_VALUE: // FloatValue
            FloatValue = static_cast<float>(Value);
            break;

        case YESNO_VALUE: // BooleanValue
            BooleanValue = ((Value != 0.0) ? 1 : 0);
            break;

        case DATE_VALUE: // DateTimeValue
        case TIME_VALUE: // DateTimeValue
        case DATETIME_VALUE: // DateTimeValue
            DateTimeValue = static_cast<double>(Value);
            break;

        case INT_VALUE: // IntValue
        case CHART_NUMBER:
            IntValue = Value;
            break;

        case COLOR_VALUE: // ColorValue
            ColorValue = static_cast<unsigned int>(Value);
            break;

        case CHART_STUDY_SUBGRAPH_VALUES:
        case STUDY_SUBGRAPH_VALUES:
        case CHART_STUDY_VALUES:
            break;

        case DOUBLE_VALUE: // DoubleValue
            DoubleValue = static_cast<double>(Value);
            break;

        case TIME_WITH_TIMEZONE_VALUE:
            DateTimeValue = static_cast<double>(Value);
            break;

        case STRING_VALUE:
        case PATH_AND_FILE_NAME_VALUE:
        case FIND_SYMBOL_VALUE:
            break;

        default:
        case NO_VALUE:
            break;
    }
}

```



```

    return *this;
}

s_SCInput_145& SetStudyID(unsigned int Value)
{
    ValueType = STUDYID_VALUE;
    IndexValue = Value;
    return *this;
}

s_SCInput_145& SetColor(unsigned int Color)
{
    ValueType = COLOR_VALUE;
    ColorValue = Color;
    return *this;
}

s_SCInput_145& SetColor(unsigned char Red, unsigned char Green, unsigned char Blue)
{
    return SetColor(RGB(Red, Green, Blue));
}

s_SCInput_145& SetAlertSoundNumber(unsigned int Value)
{
    ValueType = ALERT_SOUND_NUMBER_VALUE;
    IndexValue = Value;
    return *this;
}

s_SCInput_145& SetCandleStickPatternIndex(unsigned int Value)
{
    ValueType = CANDLESTICK_PATTERNS_VALUE;
    IndexValue = Value;
    return *this;
}

s_SCInput_145& SetChartStudySubgraphValues(int ChartNumber, int StudyID, int SubgraphIndex)
{
    ValueType = CHART_STUDY_SUBGRAPH_VALUES;
    ChartStudySubgraphValues.ChartNumber = ChartNumber;
    ChartStudySubgraphValues.StudyID = StudyID;
    ChartStudySubgraphValues.SubgraphIndex = SubgraphIndex;
    return *this;
}

s_SCInput_145& SetChartNumber(int ChartNumber)
{
    ValueType = CHART_NUMBER;
    IntValue = ChartNumber;
    return *this;
}

s_SCInput_145& SetStudySubgraphValues(int StudyID, int SubgraphIndex)
{
    ValueType = STUDY_SUBGRAPH_VALUES;
    ChartStudySubgraphValues.StudyID = StudyID;
    ChartStudySubgraphValues.SubgraphIndex = SubgraphIndex;
    return *this;
}

s_SCInput_145& SetChartStudyValues(int ChartNumber, int StudyID)
{
    ValueType = CHART_STUDY_VALUES;
    ChartStudySubgraphValues.ChartNumber = ChartNumber;

```

```

    ChartStudySubgraphValues.StudyID = StudyID;
    return *this;
}

s_SCInput_145& SetCustomInputIndex(unsigned int Value)
{
    ValueType = CUSTOM_STRING_VALUE;
    IndexValue = Value;
    return *this;
}

s_SCInput_145& SetDouble(double Value)
{
    ValueType = DOUBLE_VALUE;
    DoubleValue = Value;
    return *this;
}

s_SCInput_145& SetDoubleWithoutTypeChange(double Value)
{
    switch (ValueType)
    {
        case OHLC_VALUE: // IndexValue
        case STUDYINDEX_VALUE: // IndexValue
        case SUBGRAPHINDEX_VALUE: // IndexValue
        case MOVAVGTYPE_VALUE: // IndexValue
        case TIME_PERIOD_LENGTH_UNIT_VALUE:
        case STUDYID_VALUE: // IndexValue
        case CANDLESTICK_PATTERNS_VALUE:
        case CUSTOM_STRING_VALUE:
        case TIMEZONE_VALUE: // IndexValue
        case ALERT_SOUND_NUMBER_VALUE:
            IndexValue = static_cast<int>(Value);
            break;

        case FLOAT_VALUE: // FloatValue
            FloatValue = static_cast<float>(Value);
            break;

        case YESNO_VALUE: // BooleanValue
            BooleanValue = ((Value != 0.0) ? 1 : 0);
            break;

        case DATE_VALUE: // DateTimeValue
        case TIME_VALUE: // DateTimeValue
        case DATETIME_VALUE: // DateTimeValue
            DateTimeValue = Value;
            break;

        case INT_VALUE: // IntValue
        case CHART_NUMBER:
            IntValue = static_cast<int>(Value);
            break;

        case COLOR_VALUE: // ColorValue
            ColorValue = static_cast<unsigned int>(Value);
            break;

        case CHART_STUDY_SUBGRAPH_VALUES:
        case STUDY_SUBGRAPH_VALUES:
        case CHART_STUDY_VALUES:
            break;

        case DOUBLE_VALUE: // DoubleValue
            DoubleValue = Value;
    }
}

```

```

        break;

    case TIME_WITH_TIMEZONE_VALUE:
        DateTimeValue = Value;
        break;

    case STRING_VALUE:
    case PATH_AND_FILE_NAME_VALUE:
    case FIND_SYMBOL_VALUE:
        break;

    default:
    case NO_VALUE:
        break;
    }

    return *this;
}

s_SCInput_145& SetTimeZone(TimeZonesEnum Value)
{
    ValueType = TIMEZONE_VALUE;
    IndexValue = Value;
    return *this;
}

s_SCInput_145& SetTimeWithTimeZone(int ValueTime, TimeZonesEnum ValueTimeZone)
{
    ValueType = TIME_WITH_TIMEZONE_VALUE;
    DateTimeValue = SCDateTime(0, ValueTime).GetTimeAsDouble();
    TimeZoneValue = ValueTimeZone;
    return *this;
}

s_SCInput_145& SetString(const char* Value)
{
    ClearString();

    size_t StringLength = 0;
    if (Value != NULL)
        StringLength = strlen(Value);

    if (StringLength != 0)
    {
        size_t NumBytes = StringLength + 1;

        char* NewString = (char*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, NumBytes);

        if (NewString != NULL)
        {
            #if __STDC_WANT_SECURE_LIB__
                strcpy_s(NewString, NumBytes, Value);
            #else
                strcpy(NewString, Value);
            #endif
            StringValue = NewString;
        }
    }

    ValueType = STRING_VALUE;

    //if (StringLength > 0 && StringValue != NULL)
        StringModified = true;

    return *this;
}

```

```

}

void ClearString()
{
    if (UsesStringValue() && StringModified && StringValue != NULL)
        HeapFree(GetProcessHeap(), 0, (char*)StringValue);

    StringModified = false;

    StringValue = NULL;
}

s_SCInput_145& SetPathAndFileName(const char* PathAndFileName)
{
    SetString(PathAndFileName);
    ValueType = PATH_AND_FILE_NAME_VALUE;
    return *this;
}

s_SCInput_145& SetSymbol(const char* Symbol)
{
    SetString(Symbol);
    ValueType = FIND_SYMBOL_VALUE;
    return *this;
}

void SetFloatLimits(float Min, float Max)
{
    FloatValueMin = Min;
    FloatValueMax = Max;
}

void SetDateTimeLimits(SCDateTime Min, SCDateTime Max)
{
    DateTimeValueMin = Min.GetAsDouble();
    DateTimeValueMax = Max.GetAsDouble();
}

void SetIntLimits(int Min, int Max)
{
    IntValueMin = Min;
    IntValueMax = Max;
}

void SetDoubleLimits(double Min, double Max)
{
    DoubleValueMin = Min;
    DoubleValueMax = Max;
}

void SetCustomInputStrings(const char* p_CustomStrings)
{
    if (InternalSetCustomStrings != NULL)
    {
        InternalSetCustomStrings(InputIndex, p_CustomStrings);
        ValueType = CUSTOM_STRING_VALUE;
    }
}

void SetDescription(const char* Description)
{
    if (InternalSetDescription != NULL && Description != NULL)
        InternalSetDescription(InputIndex, Description);
}
};

```

```
typedef c_ArrayWrapper<s_SCInput_145> SCInput145Array;
typedef s_SCInput_145& SCInputRef;
```

```
/* **** */
```

```
struct s_Parabolic
```

```
{
    s_Parabolic(SCBaseDataRef BaseDataIn, SCSubgraphRef Out, SCDateTimeArrayRef BaseDateTimeIn, int Index,
float InStartAccelFactor, float InAccelIncrement, float InMaxAccelFactor, unsigned int InAdjustForGap, int
InputDataHighIndex = SC_HIGH, int InputDataLowIndex = SC_LOW)
    : m_BaseDataIn(BaseDataIn),
      m_Out(Out),
      m_BaseDateTimeIn(BaseDateTimeIn),
      m_Index(Index),
      m_InStartAccelFactor(InStartAccelFactor),
      m_InAccelIncrement(InAccelIncrement),
      m_InMaxAccelFactor(InMaxAccelFactor),
      m_InAdjustForGap(InAdjustForGap),
      m_InputDataHighIndex(InputDataHighIndex),
      m_InputDataLowIndex(InputDataLowIndex)
    {
    }

```

```
SCBaseDataRef m_BaseDataIn;
SCSubgraphRef m_Out;
SCDateTimeArrayRef m_BaseDateTimeIn;
int m_Index;
float m_InStartAccelFactor;
float m_InAccelIncrement;
float m_InMaxAccelFactor;
unsigned int m_InAdjustForGap;
int m_InputDataHighIndex;
int m_InputDataLowIndex;
```

```
inline float BaseDataHigh(int Index)
{
    return m_BaseDataIn[m_InputDataHighIndex][Index];
}

```

```
inline float BaseDataLow(int Index)
{
    return m_BaseDataIn[m_InputDataLowIndex][Index];
}

```

```
};
```

```
/* **** */
```

```
struct s_NumericInformationGraphDrawTypeConfig
```

```
{
    static const int NUMBER_OF_THRESHOLDS = 3;
    uint32_t TextBackgroundColor = COLOR_BLACK;
    bool TransparentTextBackground = true;
    bool LabelsOnRight = false;
    bool AllowLabelOverlap = false;
    bool DrawGridlines = true;
    int GridLineStyleSubgraphIndex = -1;
    int FontSize = 0;
    bool ShowPullback = false;
    bool Unused = false;
    bool HideLabels = false;
    char Reserved = 0;
    int ValueFormat[SC_SUBGRAPHS_AVAILABLE] = {};
    int SubgraphOrder[SC_SUBGRAPHS_AVAILABLE] = {};

```

```

bool ColorBackgroundBasedOnValuePercent = false;
int DetermineMaxMinForBackgroundColoringFrom = 0;
float HighestValue[SC_SUBGRAPHS_AVAILABLE] = {};
float LowestValue[SC_SUBGRAPHS_AVAILABLE] = {};
float PercentCompareThresholds[NUMBER_OF_THRESHOLDS] = {};
uint32_t Range3UpColor = COLOR_BLACK;
uint32_t Range2UpColor = COLOR_BLACK;
uint32_t Range1UpColor = COLOR_BLACK;
uint32_t Range0UpColor = COLOR_BLACK;
uint32_t Range0DownColor = COLOR_BLACK;
uint32_t Range1DownColor = COLOR_BLACK;
uint32_t Range2DownColor = COLOR_BLACK;
uint32_t Range3DownColor = COLOR_BLACK;
bool DifferentPullbackAndLabelsColor = false;
uint32_t PullbackAndLabelsColor = COLOR_WHITE;
bool ColorPullbackBackgroundBasedOnPositiveNegative = false;
bool UseDefaultNumberFormattingForAllSubgraphs = false;
std::vector<float> DailyHighestValue[SC_SUBGRAPHS_AVAILABLE]; //cannot safely be used by ACSIL
std::vector<float> DailyLowestValue[SC_SUBGRAPHS_AVAILABLE]; //cannot safely be used by ACSIL
SCDateTimeMS DateTimeOfHigh;
SCDateTimeMS DateTimeOfLow;
int VolumeDisplayMultiplierIndex = 0;

s_NumericInformationGraphDrawTypeConfig()
{
    for (int Index = 0; Index < SC_SUBGRAPHS_AVAILABLE; Index++)
    {
        SubgraphOrder[Index] = Index;
        ValueFormat[Index] = VALUEFORMAT_INHERITED;
        HighestValue[Index] = -FLT_MAX;
        LowestValue[Index] = FLT_MAX;
    }
}

};

/*****

struct s_ACSSOpenChartParameters
{
    int Version = 5;
    int PriorChartNumber = 0; // This gets checked first, 0 is unknown

    ChartDataTypeEnum ChartDataType = NO_DATA_TYPE;
    SCString Symbol;

    IntradayBarPeriodTypeEnum IntradayBarPeriodType = IBPT_DAYS_MINS_SECS;
    int IntradayBarPeriodLength = 0;

    SCDateTime SessionStartTime;
    SCDateTime SessionEndTime;
    SCDateTime EveningSessionStartTime;
    SCDateTime EveningSessionEndTime;

    int DaysToLoad = 0;

    int IntradayBarPeriodParm2 = 0;
    int IntradayBarPeriodParm3 = 0;

    int IntradayBarPeriodParm4 = 0;

    HistoricalChartBarPeriodEnum HistoricalChartBarPeriod
        = HISTORICAL_CHART_PERIOD_DAYS;

    int ChartLinkNumber = 0;

```

```

int ContinuousFuturesContractOption = CFCO_NONE; // ContinuousFuturesContractOptionsEnum

int LoadWeekendData = 1;

int HistoricalChartBarPeriodLengthInDays = 1;

int UseEveningSession = 1;
int HideNewChart = 0;

int16_t AlwaysOpenNewChart = 0;
int16_t IsNewChart = 0;
int16_t IncludeColumnsWithNoData = 0;
int16_t UpdatePriorChartNumberParametersToMatch = 0;

SCString ChartTimeZone;
int UseExactMatchForDaysToLoad = 0;

};

namespace n_ACSIL
{
    struct s_BarPeriod
    {
        ChartDataTypeEnum ChartDataType = NO_DATA_TYPE;
        IntradayBarPeriodTypeEnum IntradayChartBarPeriodType = IBPT_DAYS_MINS_SECS;
        int IntradayChartBarPeriodParameter1 = 0;
        int IntradayChartBarPeriodParameter2 = 0;
        int IntradayChartBarPeriodParameter3 = 0;
        int IntradayChartBarPeriodParameter4 = 0;
        HistoricalChartBarPeriodEnum HistoricalChartBarPeriodType = HISTORICAL_CHART_PERIOD_DAYS;
        int HistoricalChartDaysPerBar = 0;
        SCString ACSILCustomChartStudyName;

        s_BarPeriod()
        {
        }

        int AreRangeBars()
        {
            return
                (IntradayChartBarPeriodType == IBPT_RANGE_IN_TICKS_STANDARD
                 || IntradayChartBarPeriodType == IBPT_RANGE_IN_TICKS_NEWBAR_ON_RANGEMET
                 || IntradayChartBarPeriodType == IBPT_RANGE_IN_TICKS_TRUE
                 || IntradayChartBarPeriodType == IBPT_RANGE_IN_TICKS_FILL_GAPS
                 || IntradayChartBarPeriodType == IBPT_RANGE_IN_TICKS_OPEN_EQUAL_CLOSE
                 || IntradayChartBarPeriodType ==
IBPT_RANGE_IN_TICKS_NEW_BAR_ON_RANGE_MET_OPEN_EQUALS_PRIOR_CLOSE
                )
                ? 1 : 0;
        }

        int AreRenkoBars()
        {
            return
                (IntradayChartBarPeriodType == IBPT_RENKO_IN_TICKS
                 || IntradayChartBarPeriodType == IBPT_FLEX_RENKO_IN_TICKS
                 || IntradayChartBarPeriodType == IBPT_FLEX_RENKO_IN_TICKS_INVERSE_SETTINGS
                 || IntradayChartBarPeriodType == IBPT_ALIGNED_RENKO
                )
                ? 1 : 0;
        }
    };
};

```

```

struct s_ChartSessionTimes
{
    SCDateTime StartTime;
    SCDateTime EndTime;
    SCDateTime EveningStartTime;
    SCDateTime EveningEndTime;

    int UseEveningSessionTimes = 0;
    int NewBarAtSessionStart = 0;
    int LoadWeekendDataSetting = 0;
};

struct s_HTTPHeader
{
    SCString Name;
    SCString Value;
};

struct s_WriteBarAndStudyDataToFile
{
    int StartingIndex = 0;
    SCString OutputPathAndFileName;
    int IncludeHiddenStudies = 0;
    int IncludeHiddenSubgraphs = 0;
    int AppendOnlyToFile = 0;
    int IncludeLastBar = 0;
    int UseUTCTime = 0;
    int WriteStudyData = 1;
    int UseTabDelimiter = 0;//0 means use commas
};

```

```

struct s_LineUntilFutureIntersection
{
    int StartBarIndex = 0;

```

//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 stops at that index and does not extend. When EndBarIndex is set, the ending chart bar index is controlled by the study directly.

```

    int EndBarIndex = 0;

    int LineIDForBar = 0;
    float LineValue = 0;
    float LineValue2ForRange = 0;
    int UseLineValue2 = 0;
    uint32_t LineColor = 0;
    uint16_t LineWidth = 0;
    uint16_t LineStyle = 0;
    int DrawValueLabel = 0;
    int DrawNameLabel = 0;
    SCString NameLabel;
    int AlwaysExtendToEndOfChart = 0;
    int TransparencyLevel = 50;
    int PerformCloseCrossoverComparison = 0;
};

```

```

struct s_StudyProfileInformation
{
    SCDateTime m_StartDateTime;
    int64_t m_NumberOfTrades = 0;
    int64_t m_Volume = 0;
    int64_t m_BidVolume = 0;
    int64_t m_AskVolume = 0;

```



```

int m_TotalTPOCount = 0;

float m_OpenPrice = 0;
float m_HighestPrice = 0;
float m_LowestPrice = 0;
float m_LastPrice = 0;

float m_TPOMidpointPrice = 0;
float m_TPOMean = 0;
float m_TPOStdDev = 0;
float m_TPOErrorOfMean = 0;
float m_TPOPOCPrice = 0;
float m_TPOValueAreaHigh = 0;
float m_TPOValueAreaLow = 0;
int64_t m_TPOCountAbovePOC = 0;
int64_t m_TPOCountBelowPOC = 0;

float m_VolumeMidpointPrice = 0;
float m_VolumePOCPrice = 0;
float m_VolumeValueAreaHigh = 0;
float m_VolumeValueAreaLow = 0;
int64_t m_VolumeAbovePOC = 0;
int64_t m_VolumeBelowPOC = 0;
double m_POCAboveBelowVolumeImbalancePercent = 0;
int64_t m_VolumeAboveLastPrice = 0;
int64_t m_VolumeBelowLastPrice = 0;
int64_t m_BidVolumeAbovePOC = 0;
int64_t m_BidVolumeBelowPOC = 0;
int64_t m_AskVolumeAbovePOC = 0;
int64_t m_AskVolumeBelowPOC = 0;
int64_t m_VolumeTimesPriceInTicks = 0;
int64_t m_TradesTimesPriceInTicks = 0;
int64_t m_TradesTimesPriceSquaredInTicks = 0;

float m_IBRHighPrice = 0;
float m_IBRLowPrice = 0;

float m_OpeningRangeHighPrice = 0;
float m_OpeningRangeLowPrice = 0;

float m_VolumeWeightedAveragePrice = 0;
int m_MaxTPOBlocksCount = 0;

int m_TPOCountMaxDigits = 0;

uint8_t m_ColumnsDisplayStyle = 0;

bool m_EveningSession = false;

float m_AverageSubPeriodRange = 0;
int32_t m_RotationFactor = 0;

int64_t m_VolumeAboveTPOPOC = 0;
int64_t m_VolumeBelowTPOPOC = 0;
SCDateTime m_EndDateTime;

uint32_t m_BeginIndex = 0;
uint32_t m_EndIndex = 0;
};

struct s_TradeStatistics
{
    double ClosedTradesProfitLoss = 0;
    double ClosedTradesTotalProfit = 0;
    double ClosedTradesTotalLoss = 0;

```

```

double ProfitFactor = 0;
double EquityPeak = 0;
double EquityValley = 0;
double MaximumRunup = 0;
double MaximumDrawdown = 0;
double MaximumFlatToFlatTradeOpenProfit = 0;
double MaximumFlatToFlatTradeOpenLoss = 0;
double AverageTradeOpenProfit = 0;
double AverageTradeOpenLoss = 0;
double AverageWinningTradeOpenProfit = 0;
double AverageWinningTradeOpenLoss = 0;
double AverageLosingTradeOpenProfit = 0;
double AverageLosingTradeOpenLoss = 0;
double MaximumTradeOpenProfit = 0;
double MaximumTradeOpenLoss = 0;
double HighestPriceDuringPositions = 0;
double LowestPriceDuringPositions = 0;
double TotalCommissions = 0;
int32_t TotalTrades = 0;
int32_t TotalFlatToFlatTrades = 0;
int32_t TotalFilledQuantity = 0;
double PercentProfitable = 0;
double FlatToFlatPercentProfitable = 0;
int32_t WinningTrades = 0;
int32_t WinningFlatToFlatTrades = 0;
int32_t LosingTrades = 0;
int32_t LosingFlatToFlatTrades = 0;
int32_t LongTrades = 0;
int32_t LongFlatToFlatTrades = 0;
int32_t ShortTrades = 0;
int32_t ShortFlatToFlatTrades = 0;
double AverageTradeProfitLoss = 0;
double AverageFlatToFlatTradeProfitLoss = 0;
double AverageWinningTrade = 0;
double AverageFlatToFlatWinningTrade = 0;
double AverageLosingTrade = 0;
double AverageFlatToFlatLosingTrade = 0;
double AverageProfitFactor = 0;
double AverageFlatToFlatProfitFactor = 0;
double LargestWinningTrade = 0;
double LargestFlatToFlatWinningTrade = 0;
double LargestLosingTrade = 0;
double LargestFlatToFlatLosingTrade = 0;
double LargestWinnerPercentOfProfit = 0;
double LargestFlatToFlatWinnerPercentOfProfit = 0;
double LargestLoserPercentOfLoss = 0;
double LargestFlatToFlatLoserPercentOfLoss = 0;
int32_t MaxConsecutiveWinners = 0;
int32_t MaxConsecutiveLosers = 0;
int32_t AverageTimeInTrades = 0; // In seconds
int32_t AverageTimeInWinningTrades = 0; // In seconds
int32_t AverageTimeInLosingTrades = 0; // In seconds
int32_t LongestHeldWinningTrade = 0; // In seconds
int32_t LongestHeldLosingTrade = 0; // In seconds
int32_t TotalQuantity = 0;
int32_t WinningQuantity = 0;
int32_t LosingQuantity = 0;
float AverageQuantityPerTrade = 0;
float AverageQuantityPerFlatToFlatTrade = 0;
float AverageQuantityPerWinningTrade = 0;
float AverageQuantityPerFlatToFlatWinningTrade = 0;
float AverageQuantityPerLosingTrade = 0;
float AverageQuantityPerFlatToFlatLosingTrade = 0;
int32_t LargestTradeQuantity = 0;
int32_t LargestFlatToFlatTradeQuantity = 0;

```

```

int32_t MaximumOpenPositionQuantity = 0;
double LastTradeProfitLoss = 0;
int32_t LastTradeQuantity = 0;
int32_t NumberOfOpenTrades = 0;
int32_t OpenTradesOpenQuantity = 0;
double OpenTradesAverageEntryPrice = 0;
SCDateTime LastFillDateTime;
SCDateTime LastEntryDateTime;
SCDateTime LastExitDateTime;
int32_t TotalBuyQuantity = 0;
int32_t TotalSellQuantity = 0;

double ClosedFlatToFlatTradesProfitLoss = 0;
};

struct s_ChartReplayParameters
{
    int ChartNumber = 0;
    float ReplaySpeed = 1;
    SCDateTimeMS StartDateTime;
    int SkipEmptyPeriods = 0;
    n_ACSIL::ChartReplayModeEnum ReplayMode = REPLAY_MODE_UNSET;
    int ClearExistingTradeSimulationDataForSymbolAndTradeAccount = 1;
    n_ACSIL::ChartsToReplayEnum ChartsToReplay = CHARTS_TO_REPLAY_UNSET;
};

struct s_AddStudy
{
    // 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.
    int ChartNumber = 0;

    // The configured study identifier to add if it is a built-in study. The
    // study identifiers are specified in c_Graph::Configure. When adding an
    // advanced custom study this needs to be zero. The study identifiers will
    // be visible in an upcoming release through the Study Settings window.
    int StudyID = 0;

    // 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
    SCString CustomStudyFileAndFunctionName;

    // 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.
    SCString ShortName;
};

struct s_MarketOrderData
{
    t_MarketDataQuantity OrderQuantity = 0;
    uint64_t OrderID = 0;
};

struct s_TradeAccountDataFields
{
    SCString m_TradeAccount;
    uint8_t m_IsSimulated = 0;
};

```

```

SCString m_CurrencyCode;
double m_CurrentCashBalance = 0;
uint64_t m_TransactionIdentifierForCashBalanceAdjustment = 0;
double m_AvailableFundsForNewPositions = 0;
double m_MarginRequirement = 0;
double m_MarginRequirementFull = 0;
double m_MarginRequirementFullPositionsOnly = 0;
double m_PeakMarginRequirement = 0;
double m_AccountValue = 0;
double m_OpenPositionsProfitLoss = 0;
double m_DailyProfitLoss = 0;
double m_CalculatedDailyNetLossLimitInAccountCurrency = 0;
double m_TrailingAccountValueAtWhichToNotAllowNewPositions = 0;
uint8_t m_DailyNetLossLimitHasBeenReached = 0;
uint8_t m_IsUnderRequiredMargin = 0;
uint8_t m_IsUnderRequiredAccountValue = 0;
uint8_t m_TradingIsDisabled = 0;
uint8_t m_ClosePositionsAtEndOfDay = 0;
SCString m_Description;
};

```

```

struct s_GraphicsColor
{
    union u_Color
    {
        uint32_t RawColor = RGB(0, 0, 0);
        struct
        {
            uint8_t Red;
            uint8_t Green;
            uint8_t Blue;
            uint8_t Alpha;
        }s_RgbColor;
    };
};

```

```

struct s_GraphicsPen
{
    enum class e_PenStyle
    {
        PEN_STYLE_SOLID
        , PEN_STYLE_DASH
        , PEN_STYLE_DOT
        , PEN_STYLE_DASHDOT
        , PEN_STYLE_DASHDOTDOT
        , PEN_STYLE_NULL
    }
    PenStyle = e_PenStyle::PEN_STYLE_SOLID;

    s_GraphicsColor PenColor;
};

```

```

struct s_GraphicsBrush
{
    enum e_BrushStyle
    {
        BRUSH_STYLE_NULL,
        BRUSH_STYLE_SOLID,
        BRUSH_STYLE_PATTERN
    }
    BrushType = BRUSH_STYLE_NULL;
    s_GraphicsColor BrushColor;
    std::vector<unsigned char> BrushBitPattern;
    int Width = 0;
};

```

```
    int Height = 0;
};

struct s_GraphicsFont
{
    SCString FontFaceName;
    int FontHeight = 0;
    int FontWeight = 0;
    bool FontIsItalic = false;
    bool FontIsUnderline = false;
    int FontAngle = 0;
};

struct s_GraphicsRectangle
{
    int Left = 0;
    int Top = 0;
    int Right = 0;
    int Bottom = 0;
};

struct s_GraphicsPoint
{
    int X = 0;
    int Y = 0;
};

} // namespace n_ACSIL

#endif
```