

```

// File Name: sierrachart.h

#ifndef _SIERRACHART_H_
#define _SIERRACHART_H_

// SC_DLL_VERSION gets updated only when there have been changes to the DLL
// study interface. This should be set to the Sierra Chart version number
// that includes the changes made to the interface.
#define SC_DLL_VERSION 2545

const int MINIMUM_DLL_VERSION = 2155;

#ifndef _CRT_SECURE_NO_WARNINGS
#define _CRT_SECURE_NO_WARNINGS
#endif

// Note: Sierra Chart expects a default packing of 8 bytes.
#pragma pack(push, 8)

#include "scstructures.h"
#include "ACSILDepthBars.h"
#include "IntradayRecord.h"
#include "ACSILCustomChartBars.h"

/*****

// Structure passed to custom DLL study functions.
// s_sc = structure _ sierra chart

struct s_sc
{
    /*****
    // Functions (These do not expand the size of the structure)

    s_sc(); // Only implemented internally within Sierra Chart

    s_sc(const s_sc& Source);
    s_sc& operator=(const s_sc& Source);

    void CopyDataMembers(const s_sc& Source); // Only implemented internally within Sierra Chart
    void SetFunctionPointers(); // Only implemented internally within Sierra Chart

    // Non static functions
    int CompareOpenToHighLow(float Open, float High, float Low, int ValueFormat)
    {
        float OpenToHighDiff = High - Open;
        float OpenToLowDiff = Open - Low;

        if (FormattedEvaluate(OpenToHighDiff, ValueFormat, LESS_OPERATOR, OpenToLowDiff, ValueFormat))
            return 1; // Open closer to High
        else if (FormattedEvaluate(OpenToHighDiff, ValueFormat, GREATER_OPERATOR, OpenToLowDiff, ValueFormat))
            return -1; // Open closer to Low

        return 0; // must be equal distance
    }

    void AddMessageToLog(const char* Message, int ShowLog)
    {
        p_AddMessageToLog(Message, ShowLog);
    }
    void AddMessageToLog(const SCString& Message, int ShowLog)
    {

```

```

    p_AddMessageToLog(Message.GetChars(), ShowLog);
}

void AddMessageToTradeServiceLog(const char* Message, int ShowLog = 0, int AddChartStudyDetails = 0)
{
    Internal_AddMessageToTradeServiceLog(Message, ShowLog, AddChartStudyDetails);
}
void AddMessageToTradeServiceLog(const SCString& Message, int ShowLog = 0, int AddChartStudyDetails = 0)
{
    Internal_AddMessageToTradeServiceLog(Message.GetChars(), ShowLog, AddChartStudyDetails);
}

int GetStudyIDByName(int ChartNumber, const char* Name, const int UseShortNameIfSet)
{
    return Internal_GetStudyIDByName(ChartNumber, Name, UseShortNameIfSet);
}

int GetStudyIDByName(int ChartNumber, const SCString& Name, const int UseShortNameIfSet)
{
    return Internal_GetStudyIDByName(ChartNumber, Name.GetChars(), UseShortNameIfSet);
}

/*=====
BHCS_BAR_HAS_CLOSED: Element at BarIndex has closed.
BHCS_BAR_HAS_NOT_CLOSED: Element at BarIndex has not closed.
BHCS_SET_DEFAULTS: Configuration and defaults are being set. Allow your SetDefaults code block to run.
-----*/

int GetBarHasClosedStatus()
{
    return GetBarHasClosedStatus(Index);
}

int GetBarHasClosedStatus(int BarIndex)
{
    if (SetDefaults)
        return BHCS_SET_DEFAULTS;

    if (BarIndex != ArraySize - 1)
        return BHCS_BAR_HAS_CLOSED;
    else
        return BHCS_BAR_HAS_NOT_CLOSED;
}

float GetHighest(SCFloatArrayRef In, int Index, int Length)
{
    float High = -FLT_MAX;

    // Get the high starting at Index going back by Length in the In array
    for (int SrcIndex = Index; SrcIndex > Index - Length; --SrcIndex)
    {
        if (SrcIndex < 0 || SrcIndex >= In.GetArraySize())
            continue;

        if (In[SrcIndex] > High)
            High = In[SrcIndex];
    }

    return High;
}

```

```

float GetHighest(SCFloatArrayRef In, int Length)
{
    return GetHighest( In, Index, Length);
}

float GetLowest(SCFloatArrayRef In, int Index, int Length)
{
    float Low = FLT_MAX;

    // Get the low starting at Index going back by Length in the In array
    for (int SrcIndex = Index; SrcIndex > Index - Length; --SrcIndex)
    {
        if (SrcIndex < 0 || SrcIndex >= In.GetArraySize())
            continue;

        if (In[SrcIndex] < Low)
            Low = In[SrcIndex];
    }

    return Low;
}

float GetLowest(SCFloatArrayRef In, int Length)
{
    return GetLowest( In, Index, Length);
}

int GetIndexOfHighestValue(SCFloatArrayRef In, int Index, int Length)
{
    float High = -FLT_MAX;

    int IndexAtHighest = 0;
    // Get the high starting at Index going back by Length in the In array
    for (int SrcIndex = Index; SrcIndex > Index - Length; --SrcIndex)
    {
        if (SrcIndex < 0 || SrcIndex >= In.GetArraySize())
            continue;

        if (In[SrcIndex] > High)
        {
            High = In[SrcIndex];
            IndexAtHighest = SrcIndex;
        }
    }

    return IndexAtHighest;
}

int GetIndexOfHighestValue(SCFloatArrayRef In, int Length)
{
    return GetIndexOfHighestValue( In, Index, Length);
}

int GetIndexOfLowestValue(SCFloatArrayRef In, int Index, int Length)
{
    float Low = FLT_MAX;

    int IndexAtLowest = 0;
    // Get the high starting at Index going back by Length in the In array
    for (int SrcIndex = Index; SrcIndex > Index - Length; --SrcIndex)
    {
        if (SrcIndex < 0 || SrcIndex >= In.GetArraySize())
            continue;

        if (In[SrcIndex] < Low)

```

```

    {
        Low = In[SrcIndex];
        IndexAtLowest = SrcIndex;
    }
}

return IndexAtLowest;
}

int GetIndexOfLowestValue(SCFloatArrayRef In, int Length)
{
    return GetIndexOfLowestValue( In, Index, Length);
}

int CrossOver(SCFloatArrayRef First, SCFloatArrayRef Second)
{
    return CrossOver( First, Second, Index);
}

int CrossOver(SCFloatArrayRef First, SCFloatArrayRef Second, int Index)
{
    float X1 = First[Index-1];
    float X2 = First[Index];
    float Y1 = Second[Index-1];
    float Y2 = Second[Index];

    if (X2 != Y2) // The following is not useful if X2 and Y2 are equal
    {
        // Find non-equal values for prior values
        int PriorIndex = Index - 1;
        while (X1 == Y1 && PriorIndex > 0 && PriorIndex > Index - 100)
        {
            --PriorIndex;
            X1 = First[PriorIndex];
            Y1 = Second[PriorIndex];
        }
    }

    if (X1 > Y1 && X2 < Y2)
        return CROSS_FROM_TOP;
    else if (X1 < Y1 && X2 > Y2)
        return CROSS_FROM_BOTTOM;
    else
        return NO_CROSS;
}

int ResizeArrays(int NewSize)
{
    return p_ResizeArrays(NewSize);
}

int AddElements(int NumElements)
{
    return p_AddElements(NumElements);
}

int FormattedEvaluate(float Value1, unsigned int Value1Format,
    OperatorEnum Operator,
    float Value2, unsigned int Value2Format,
    float PrevValue1 = 0.0f, float PrevValue2 = 0.0f,
    int* CrossDirection = NULL)
{
    return InternalFormattedEvaluate(Value1, Value1Format, Operator, Value2, Value2Format, PrevValue1,
    PrevValue2, CrossDirection)?1:0;
}

```

```

}

int FormattedEvaluateUsingDoubles(double Value1, unsigned int Value1Format,
    OperatorEnum Operator,
    double Value2, unsigned int Value2Format,
    double PrevValue1 = 0.0f, double PrevValue2 = 0.0f,
    int* CrossDirection = NULL)
{
    return InternalFormattedEvaluateUsingDoubles(Value1, Value1Format, Operator, Value2, Value2Format,
PrevValue1, PrevValue2, CrossDirection)?1:0;
}

int PlaySound(int AlertNumber)
{
    return InternalAlertWithMessage(AlertNumber, "", 0);
}

int PlaySound(int AlertNumber, const SCString& AlertMessage, int ShowAlertLog = 0)
{
    return InternalAlertWithMessage(AlertNumber, AlertMessage.GetChars(), ShowAlertLog);
}

int PlaySound(int AlertNumber, const char* AlertMessage, int ShowAlertLog = 0)
{
    return InternalAlertWithMessage(AlertNumber, AlertMessage, ShowAlertLog);
}

int AlertWithMessage(int AlertNumber, const SCString& AlertMessage, int ShowAlertLog = 0)
{
    return InternalAlertWithMessage(AlertNumber, AlertMessage.GetChars(), ShowAlertLog);
}

int AlertWithMessage(int AlertNumber, const char* AlertMessage, int ShowAlertLog = 0)
{
    return InternalAlertWithMessage(AlertNumber, AlertMessage, ShowAlertLog);
}

int PlaySound(const char* AlertPathAndFileName, int NumberTimesPlayAlert = 1)
{
    return InternalPlaySoundPath(AlertPathAndFileName, NumberTimesPlayAlert, "", 0);
}

int PlaySound(const SCString& AlertPathAndFileName, int NumberTimesPlayAlert = 1)
{
    return InternalPlaySoundPath(AlertPathAndFileName.GetChars(), NumberTimesPlayAlert, "", 0);
}

int PlaySound(SCString& AlertPathAndFileName, int NumberTimesPlayAlert, SCString& AlertMessage, int
ShowAlertLog = 0)
{
    return InternalPlaySoundPath(AlertPathAndFileName.GetChars(), NumberTimesPlayAlert,
AlertMessage.GetChars(), ShowAlertLog);
}

int PlaySound(const char* AlertPathAndFileName, int NumberTimesPlayAlert, const char* AlertMessage, int
ShowAlertLog = 0)
{
    return InternalPlaySoundPath(AlertPathAndFileName, NumberTimesPlayAlert, AlertMessage, ShowAlertLog);
}

void AddAlertLine(const SCString& Message, int ShowAlertLog = 0)
{
    InternalAddAlertLine(Message.GetChars(), ShowAlertLog);
}

```

```

void AddAlertLine(const char* Message, int ShowAlertLog = 0)
{
    InternalAddAlertLine(Message, ShowAlertLog);
}

SCString GetStudyName(int StudyIndex)
{
    SCString Temp;
    Temp = InternalGetStudyName(StudyIndex);
    return Temp;
}

SCFloatArrayRef CCI(SCFloatArrayRef In, SCFloatArrayRef SMAOut, SCFloatArrayRef CCIOut, int Index, int Length,
float Multiplier, unsigned int MovingAverageType = MOVAVGTYPE_SIMPLE)
{
    return InternalCCI(In, SMAOut, CCIOut, Index, Length, Multiplier, MovingAverageType);
}

SCFloatArrayRef CCI(SCFloatArrayRef In, SCFloatArrayRef SMAOut, SCFloatArrayRef CCIOut, int Length, float
Multiplier, unsigned int MovingAverageType = MOVAVGTYPE_SIMPLE)
{
    return InternalCCI(In, SMAOut, CCIOut, Index, Length, Multiplier, MovingAverageType);
}

SCFloatArrayRef CCI(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int Length, float Multiplier, unsigned int
MovingAverageType = MOVAVGTYPE_SIMPLE)
{
    return InternalCCI(In, Out.Arrays[0], Out, Index, Length, Multiplier, MovingAverageType);
}

SCFloatArrayRef CCI(SCFloatArrayRef In, SCSubgraphRef Out, int Length, float Multiplier, unsigned int
MovingAverageType = MOVAVGTYPE_SIMPLE)
{
    return InternalCCI(In, Out.Arrays[0], Out, Index, Length, Multiplier, MovingAverageType);
}

SCFloatArrayRef RSI(SCFloatArrayRef In, SCSubgraphRef RsiOut, int Index, unsigned int MovingAverageType, int
Length)
{
    SCFloatArrayRef UpSumsTemp = RsiOut.Arrays[0];
    SCFloatArrayRef DownSumsTemp = RsiOut.Arrays[1];
    SCFloatArrayRef SmoothedUpSumsTemp = RsiOut.Arrays[2];
    SCFloatArrayRef SmoothedDownSumsTemp = RsiOut.Arrays[3];

    return InternalRSI(In, RsiOut, UpSumsTemp, DownSumsTemp, SmoothedUpSumsTemp,
SmoothedDownSumsTemp,
        Index, MovingAverageType, Length);
};

SCFloatArrayRef RSI(SCFloatArrayRef In, SCSubgraphRef RsiOut, unsigned int MovingAverageType, int Length)
{
    SCFloatArrayRef UpSumsTemp = RsiOut.Arrays[0];
    SCFloatArrayRef DownSumsTemp = RsiOut.Arrays[1];
    SCFloatArrayRef SmoothedUpSumsTemp = RsiOut.Arrays[2];
    SCFloatArrayRef SmoothedDownSumsTemp = RsiOut.Arrays[3];

    return InternalRSI(In, RsiOut,
        UpSumsTemp, DownSumsTemp, SmoothedUpSumsTemp, SmoothedDownSumsTemp,
        Index, MovingAverageType, Length);
};

SCFloatArrayRef RSI(SCFloatArrayRef In, SCFloatArrayRef RsiOut,
    SCFloatArrayRef UpSumsTemp, SCFloatArrayRef DownSumsTemp,

```

```

        SCFloatArrayRef SmoothedUpSumsTemp, SCFloatArrayRef SmoothedDownSumsTemp,
        int Index, unsigned int MovingAverageType, int Length)
    {
        return InternalRSI(In, RsiOut,
            UpSumsTemp, DownSumsTemp, SmoothedUpSumsTemp, SmoothedDownSumsTemp,
            Index, MovingAverageType, Length);
    }

SCFloatArrayRef RSI(SCFloatArrayRef In, SCFloatArrayRef RsiOut,
    SCFloatArrayRef UpSumsTemp, SCFloatArrayRef DownSumsTemp,
    SCFloatArrayRef SmoothedUpSumsTemp, SCFloatArrayRef SmoothedDownSumsTemp,
    unsigned int MovingAverageType, int Length)
{
    return InternalRSI(In, RsiOut,
        UpSumsTemp, DownSumsTemp, SmoothedUpSumsTemp, SmoothedDownSumsTemp,
        Index, MovingAverageType, Length);
}

void DMI(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef PosDMIOut, SCSubgraphRef NegDMIOut, int Index, int
Length)
{
    SCFloatArrayRef InternalPosDM = PosDMIOut.Arrays[0];
    SCFloatArrayRef InternalNegDM = PosDMIOut.Arrays[1];
    SCFloatArrayRef InternalTrueRangeSummation = PosDMIOut.Arrays[2];
    SCFloatArrayRef DiffDMIOut = PosDMIOut.Arrays[3];

    InternalDMI(ChartBaseDataIn, Index, Length, 1 /* rounding disabled */,
        PosDMIOut, NegDMIOut, DiffDMIOut,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM);

    return;
}

void DMI(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef PosDMIOut, SCSubgraphRef NegDMIOut, int Length)
{
    SCFloatArrayRef InternalPosDM = PosDMIOut.Arrays[0];
    SCFloatArrayRef InternalNegDM = PosDMIOut.Arrays[1];
    SCFloatArrayRef InternalTrueRangeSummation = PosDMIOut.Arrays[2];
    SCFloatArrayRef DiffDMIOut = PosDMIOut.Arrays[3];

    InternalDMI(ChartBaseDataIn, Index, Length, 1 /* rounding disabled */,
        PosDMIOut, NegDMIOut, DiffDMIOut,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM);

    return;
}

SCSubgraphRef DMI(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int Length)
{
    SCFloatArrayRef InternalPosDM = Out.Arrays[1];
    SCFloatArrayRef InternalNegDM = Out.Arrays[2];
    SCFloatArrayRef InternalTrueRangeSummation = Out.Arrays[3];
    SCFloatArrayRef DiffDMIOut = Out.Arrays[4];

    InternalDMI(ChartBaseDataIn, Index, Length, 1 /* rounding disabled */,
        Out, Out.Arrays[0], DiffDMIOut, InternalTrueRangeSummation, InternalPosDM, InternalNegDM);

    return Out;
}

SCSubgraphRef DMI(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Length)
{
    SCFloatArrayRef InternalPosDM = Out.Arrays[1];
    SCFloatArrayRef InternalNegDM = Out.Arrays[2];
    SCFloatArrayRef InternalTrueRangeSummation = Out.Arrays[3];
    SCFloatArrayRef DiffDMIOut = Out.Arrays[4];

```

```

    InternalDMI(ChartBaseDataIn, Index, Length, 1 /* rounding disabled */,
        Out, Out.Arrays[0], DiffDMIOut, InternalTrueRangeSummation, InternalPosDM, InternalNegDM);
    return Out;
}

```

```

SCFloatArrayRef DMIDiff(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int Length)
{
    SCFloatArrayRef InternalTrueRangeSummation = Out.Arrays[0];
    SCFloatArrayRef InternalPosDM = Out.Arrays[1];
    SCFloatArrayRef InternalNegDM = Out.Arrays[2];

    return InternalDMIDiff(ChartBaseDataIn, Index, Length, Out,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM);
}

```

```

SCFloatArrayRef DMIDiff(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Length)
{
    SCFloatArrayRef InternalTrueRangeSummation = Out.Arrays[0];
    SCFloatArrayRef InternalPosDM = Out.Arrays[1];
    SCFloatArrayRef InternalNegDM = Out.Arrays[2];

    return InternalDMIDiff(ChartBaseDataIn, Index, Length, Out,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM);
}

```

```

SCFloatArrayRef ADX(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int DXLength, int
DXMovAvgLength)
{
    SCFloatArrayRef InternalTrueRangeSummation = Out.Arrays[0];
    SCFloatArrayRef InternalPosDM = Out.Arrays[1];
    SCFloatArrayRef InternalNegDM = Out.Arrays[2];
    SCFloatArrayRef InternalDX = Out.Arrays[3];

    return InternalADX(ChartBaseDataIn, Index, DXLength, DXMovAvgLength, Out,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM, InternalDX);
}

```

```

SCFloatArrayRef ADX(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int DXLength, int DXMovAvgLength)
{
    SCFloatArrayRef InternalTrueRangeSummation = Out.Arrays[0];
    SCFloatArrayRef InternalPosDM = Out.Arrays[1];
    SCFloatArrayRef InternalNegDM = Out.Arrays[2];
    SCFloatArrayRef InternalDX = Out.Arrays[3];

    return InternalADX(ChartBaseDataIn, Index, DXLength, DXMovAvgLength, Out,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM, InternalDX);
}

```

```

SCFloatArrayRef ADXR(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index,
    int DXLength, int DXMovAvgLength, int ADXRInterval)
{
    SCFloatArrayRef InternalTrueRangeSummation = Out.Arrays[0];
    SCFloatArrayRef InternalPosDM = Out.Arrays[1];
    SCFloatArrayRef InternalNegDM = Out.Arrays[2];
    SCFloatArrayRef InternalDX = Out.Arrays[3];
    SCFloatArrayRef InternalADX = Out.Arrays[4];

    return InternalADXR(ChartBaseDataIn, Index, DXLength, DXMovAvgLength, ADXRInterval, Out,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM, InternalDX, InternalADX);
}

```



```

SCFloatArrayRef ADXR(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out,
    int DXLength, int DXMovAvgLength, int ADXRInterval)
{
    SCFloatArrayRef InternalTrueRangeSummation = Out.Arrays[0];
    SCFloatArrayRef InternalPosDM = Out.Arrays[1];
    SCFloatArrayRef InternalNegDM = Out.Arrays[2];
    SCFloatArrayRef InternalDX = Out.Arrays[3];
    SCFloatArrayRef InternalADX = Out.Arrays[4];

    return InternalADX(ChartBaseDataIn, Index, DXLength, DXMovAvgLength, ADXRInterval, Out,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM, InternalDX, InternalADX);
}

SCFloatArrayRef Ergodic(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int LongEMALength, int
ShortEMALength, float Multiplier)
{
    return Internal_Ergodic( In, Out, Index, LongEMALength, ShortEMALength, Multiplier,
        Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Out.Arrays[4], Out.Arrays[5]);
}

SCFloatArrayRef Ergodic(SCFloatArrayRef In, SCSubgraphRef Out, int LongEMALength, int ShortEMALength, float
Multiplier)
{
    return Internal_Ergodic( In, Out, Index, LongEMALength, ShortEMALength, Multiplier,
        Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Out.Arrays[4], Out.Arrays[5]);
}

SCFloatArrayRef Keltner(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef In, SCSubgraphRef Out, int Index, int
KeltnerMALength, unsigned int KeltnerMAType, int TrueRangeMALength, unsigned int TrueRangeMAType, float
TopBandMultiplier, float BottomBandMultiplier)
{
    return Internal_Keltner( ChartBaseDataIn, In, Out, Out.Arrays[0], Out.Arrays[1], Index, KeltnerMALength,
        KeltnerMAType, TrueRangeMALength, TrueRangeMAType, TopBandMultiplier, BottomBandMultiplier, Out.Arrays[2],
        Out.Arrays[3]);
}

SCFloatArrayRef Keltner(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef In, SCSubgraphRef Out, int
KeltnerMALength, unsigned int KeltnerMAType, int TrueRangeMALength, unsigned int TrueRangeMAType, float
TopBandMultiplier, float BottomBandMultiplier)
{
    return Internal_Keltner(ChartBaseDataIn, In, Out, Out.Arrays[0], Out.Arrays[1], Index, KeltnerMALength,
        KeltnerMAType, TrueRangeMALength, TrueRangeMAType, TopBandMultiplier, BottomBandMultiplier, Out.Arrays[2],
        Out.Arrays[3]);
}

SCFloatArrayRef SmoothedMovingAverage(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return Internal_SmoothedMovingAverage( In, Out, Index, Length);
}

SCFloatArrayRef SmoothedMovingAverage(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return Internal_SmoothedMovingAverage(In, Out, Index, Length);
}

SCFloatArrayRef ATR(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int Length, unsigned int
MovingAverageType)
{
    return InternalATR( ChartBaseDataIn, Out.Arrays[0], Out, Index, Length, MovingAverageType);
}

SCFloatArrayRef ATR(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Length, unsigned int
MovingAverageType)
{
    return InternalATR( ChartBaseDataIn, Out.Arrays[0], Out, Index, Length, MovingAverageType);
}

```

```

}

SCFloatArrayRef ATR(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef TROut, SCFloatArrayRef ATROut, int
Index, int Length, unsigned int MovingAverageType)
{
    return InternalATR( ChartBaseDataIn, TROut, ATROut, Index, Length, MovingAverageType);
}

SCFloatArrayRef ATR(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef TROut, SCFloatArrayRef ATROut, int
Length, unsigned int MovingAverageType)
{
    return InternalATR( ChartBaseDataIn, TROut, ATROut, Index, Length, MovingAverageType);
}

void Vortex(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef VMPlusOut, SCSubgraphRef VMMinusOut, int Index,
int VortexLength)
{
    InternalVortex(ChartBaseDataIn, VMPlusOut.Arrays[0], VMPlusOut.Arrays[1], VMPlusOut.Arrays[2], VMPlusOut,
VMMinusOut, Index, VortexLength);
}

void Vortex(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef VMPlusOut, SCSubgraphRef VMMinusOut, int
VortexLength)
{
    InternalVortex(ChartBaseDataIn, VMPlusOut.Arrays[0], VMPlusOut.Arrays[1], VMPlusOut.Arrays[2], VMPlusOut,
VMMinusOut, Index, VortexLength);
}

SCFloatArrayRef Stochastic(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int FastKLength, int
FastDLength, int SlowDLength, unsigned int MovingAverageType)
{
    InternalStochastic( ChartBaseDataIn, Out, Out.Arrays[0], Out.Arrays[1], Index, FastKLength, FastDLength,
SlowDLength, MovingAverageType);
    return Out;
}

SCFloatArrayRef Stochastic(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int FastKLength, int
FastDLength, int SlowDLength, unsigned int MovingAverageType)
{
    InternalStochastic( ChartBaseDataIn, Out, Out.Arrays[0], Out.Arrays[1], Index, FastKLength, FastDLength,
SlowDLength, MovingAverageType);
    return Out;
}

//Three Separate Input Arrays
SCFloatArrayRef Stochastic(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCFloatArrayRef
InputDataLast, SCSubgraphRef Out, int Index, int FastKLength, int FastDLength, int SlowDLength, unsigned int
MovingAverageType)
{
    InternalStochastic2( InputDataHigh, InputDataLow, InputDataLast, Out, Out.Arrays[0], Out.Arrays[1], Index,
FastKLength, FastDLength, SlowDLength, MovingAverageType);
    return Out;
}

//Three Separate Input Arrays
SCFloatArrayRef Stochastic(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCFloatArrayRef
InputDataLast, SCSubgraphRef Out, int FastKLength, int FastDLength, int SlowDLength, unsigned int
MovingAverageType)
{
    InternalStochastic2( InputDataHigh, InputDataLow, InputDataLast, Out, Out.Arrays[0], Out.Arrays[1], Index,
FastKLength, FastDLength, SlowDLength, MovingAverageType);
    return Out;
}

SCFloatArrayRef MACD(SCFloatArrayRef In, SCSubgraphRef Out, int IndexParam, int FastMALength, int

```

```

SlowMALength, int MACDMALength, int MovAvgType)
{
    return InternalMACD( In, Out.Arrays[0], Out.Arrays[1], Out.Data, Out.Arrays[2], Out.Arrays[3], IndexParam,
FastMALength, SlowMALength, MACDMALength, MovAvgType);
}

SCFloatArrayRef MACD(SCFloatArrayRef In, SCSubgraphRef Out, int FastMALength, int SlowMALength, int
MACDMALength, int MovAvgType)
{
    return InternalMACD( In, Out.Arrays[0], Out.Arrays[1], Out.Data, Out.Arrays[2], Out.Arrays[3], Index, FastMALength,
SlowMALength, MACDMALength, MovAvgType);
}

SCFloatArrayRef HullMovingAverage(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int Length)
{
    return InternalHullMovingAverage( In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Index, Length);
}

SCFloatArrayRef HullMovingAverage(SCFloatArrayRef In, SCSubgraphRef Out, int Length)
{
    return InternalHullMovingAverage( In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Index, Length);
}

SCFloatArrayRef TriangularMovingAverage(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int Length)
{
    return InternalTriangularMovingAverage( In, Out, Out.Arrays[0], Index, Length);
}

SCFloatArrayRef TriangularMovingAverage(SCFloatArrayRef In, SCSubgraphRef Out, int Length)
{
    return InternalTriangularMovingAverage( In, Out, Out.Arrays[0], Index, Length);
}

SCFloatArrayRef TEMA(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int Length)
{
    return InternalTEMA( In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Index, Length);
}

SCFloatArrayRef TEMA(SCFloatArrayRef In, SCSubgraphRef Out, int Length)
{
    return InternalTEMA( In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Index, Length);
}

SCFloatArrayRef BollingerBands(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int Length, float Multiplier, int
MovingAverageType)
{
    InternalBollingerBands(In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Index, Length, Multiplier,
MovingAverageType);
    return Out;
}

SCFloatArrayRef BollingerBands(SCFloatArrayRef In, SCSubgraphRef Out, int Length, float Multiplier, int
MovingAverageType)
{
    InternalBollingerBands(In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Index, Length, Multiplier,
MovingAverageType);
    return Out;
}

SCFloatArrayRef WeightedMovingAverage(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalWeightedMovingAverage( In, Out, Index, Length);
}

```

```

SCFloatArrayRef WeightedMovingAverage(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalWeightedMovingAverage( In, Out, Index, Length);
}

SCFloatArrayRef Momentum(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalMomentum(In,Out,Index,Length);
}

SCFloatArrayRef Momentum(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalMomentum(In,Out,Index,Length);
}

SCFloatArrayRef TRIX(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int Length)
{
    return InternalTRIX(In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Index, Length);
}

SCFloatArrayRef TRIX(SCFloatArrayRef In, SCSubgraphRef Out, int Length)
{
    return InternalTRIX(In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Index, Length);
}

SCFloatArrayRef AroonIndicator(SCFloatArrayRef FloatArrayInHigh, SCFloatArrayRef FloatArrayInLow,
SCSubgraphRef Out, int Index, int Length)
{
    return Internal_AroonIndicator( FloatArrayInHigh, FloatArrayInLow, Out.Data, Out.Arrays[0], Out.Arrays[1], Index,
Length);
}

SCFloatArrayRef AroonIndicator(SCFloatArrayRef FloatArrayInHigh, SCFloatArrayRef FloatArrayInLow,
SCSubgraphRef Out, int Length)
{
    return Internal_AroonIndicator( FloatArrayInHigh, FloatArrayInLow, Out.Data, Out.Arrays[0], Out.Arrays[1], Index,
Length);
}

SCFloatArrayRef Demarker(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int Length)
{
    return Internal_Demarker( ChartBaseDataIn, Out, Out.Arrays[1], Out.Arrays[3], Out.Arrays[2], Out.Arrays[4], Index,
Length);
}

SCFloatArrayRef Demarker(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Length)
{
    return Internal_Demarker( ChartBaseDataIn, Out, Out.Arrays[1], Out.Arrays[3], Out.Arrays[2], Out.Arrays[4], Index,
Length);
}

int GetOHLCOfTimePeriod(const SCDatetime& StartDateTime, const SCDatetime& EndDateTime, float& Open,
float& High, float& Low, float& Close, float& NextOpen, int& NumberOfBars, SCDatetime& r_TotalTimeSpan)
{
    s_GetOHLCOfTimePeriod GetOHLCOfTimePeriodData(StartDateTime, EndDateTime, Open, High, Low, Close,
NextOpen, NumberOfBars, r_TotalTimeSpan);

    return InternalGetOHLCOfTimePeriod(GetOHLCOfTimePeriodData);
}

int GetOHLCOfTimePeriod(const SCDatetime& StartDateTime, const SCDatetime& EndDateTime, float& Open,
float& High, float& Low, float& Close, float& NextOpen)

```

```

{
    int NumberOfBars = 0;
    SCDatetime TotalTimeSpan;

    s_GetOHLCOfTimePeriod Data(StartDateTime, EndDateime, Open, High, Low, Close, NextOpen, NumberOfBars,
TotalTimeSpan);

    return InternalGetOHLCOfTimePeriod(Data);
}

int GetOHLCForDate(const SCDatetime& Date, float& Open, float& High, float& Low, float& Close)
{
    return Internal_GetOHLCForDate(Date, Open, High, Low, Close);
}

//Parameters reviewed for safety with different compilers
int GetOpenHighLowCloseVolumeForDate(const SCDatetime& Date, float& r_Open, float& r_High, float& r_Low,
float& r_Close, float& r_Volume)
{
    return Internal_GetOpenHighLowCloseVolumeForDate(Date, r_Open, r_High, r_Low, r_Close, r_Volume);
}

//Parameters reviewed for safety with different compilers
int GetOpenHighLowCloseVolumeForDate(const SCDatetime& Date, float& r_Open, float& r_High, float& r_Low,
float& r_Close, float& r_Volume, int IncludeFridayEveningSessionWithSundayEveningSession)
{
    return Internal_GetOpenHighLowCloseVolumeForDate2(Date, r_Open, r_High, r_Low, r_Close, r_Volume,
IncludeFridayEveningSessionWithSundayEveningSession);
}

int IsReplayRunning()
{
    if(ReplayStatus != REPLAY_STOPPED)
        return 1;

    return 0;
}

SCdatetime GetCurrentDateTime()
{
    if (IsReplayRunning())
        return CurrentDateTimeForReplay;
    else
        return CurrentSystemDateTimeMS;
}

int GetExactMatchForSCdatetime(int ChartNumber, const SCDatetime& DateTime)
{
    int Index = GetContainingIndexForSCdatetime(ChartNumber, DateTime);

    SCDatetimeArray ChartDateTimeArray;
    GetChartDateTimeArray(ChartNumber, ChartDateTimeArray);

    if (ChartDateTimeArray[Index] == DateTime)
        return Index;
    else
        return -1;
}

SCFloatArrayRef Highest(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalHighest( In, Out, Index, Length);
}

```

```

SCFloatArrayRef Highest(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalHighest( In, Out, Index, Length);
}

SCFloatArrayRef Lowest(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalLowest( In, Out, Index, Length);
}

SCFloatArrayRef Lowest(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalLowest( In, Out, Index, Length);
}

SCFloatArrayRef ChaikinMoneyFlow(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int Length)
{
    return InternalChaikinMoneyFlow(ChartBaseDataIn, Out, Out.Arrays[0], Index, Length);
}

SCFloatArrayRef ChaikinMoneyFlow(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Length)
{
    return InternalChaikinMoneyFlow(ChartBaseDataIn, Out, Out.Arrays[0], Index, Length);
}

SCFloatArrayRef EnvelopePct(SCFloatArrayRef In, SCSubgraphRef Out, float pct, int Index)
{
    return InternalEnvelopePercent(In, Out, Out.Arrays[0], pct, Index);
}

SCFloatArrayRef EnvelopePct(SCFloatArrayRef In, SCSubgraphRef Out, float pct)
{
    return InternalEnvelopePercent(In, Out, Out.Arrays[0], pct, Index);
}

SCFloatArrayRef EnvelopeFixed(SCFloatArrayRef In, SCSubgraphRef Out, float FixedValue, int Index)
{
    return InternalEnvelopeFixed(In, Out, Out.Arrays[0], FixedValue, Index);
}

SCFloatArrayRef EnvelopeFixed(SCFloatArrayRef In, SCSubgraphRef Out, float FixedValue)
{
    return InternalEnvelopeFixed(In, Out, Out.Arrays[0], FixedValue, Index);
}

SCFloatArrayRef RandomWalkIndicator(SCBaseDataRef In, SCSubgraphRef Out, int Index, int Length)
{
    return InternalRWI(In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Index, Length);
}

SCFloatArrayRef RandomWalkIndicator(SCBaseDataRef In, SCSubgraphRef Out, int Length)
{
    return InternalRWI(In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Index, Length);
}

SCFloatArrayRef UltimateOscillator(SCBaseDataRef In, SCSubgraphRef Out1, SCSubgraphRef Out2, int Index, int
Length1, int Length2, int Length3)
{
    return InternalUltimateOscillator( In,
        Out1,
        Out1.Arrays[0],
        Out1.Arrays[1],
        Out1.Arrays[2],
        Out1.Arrays[3],

```

```

        Out1.Arrays[4],
        Out1.Arrays[5],
        Out1.Arrays[6],
        Out1.Arrays[7],
        Out1.Arrays[8],
        Out2.Arrays[0],
        Out2.Arrays[1],
        Out2.Arrays[2],
        Out2.Arrays[3],
        Index, Length1, Length2, Length3);
    }

    SCFloatArrayRef UltimateOscillator(SCBaseDataRef In, SCSubgraphRef Out1, SCSubgraphRef Out2, int Length1, int
Length2, int Length3)
    {
        return InternalUltimateOscillator( In,
            Out1,
            Out1.Arrays[0],
            Out1.Arrays[1],
            Out1.Arrays[2],
            Out1.Arrays[3],
            Out1.Arrays[4],
            Out1.Arrays[5],
            Out1.Arrays[6],
            Out1.Arrays[7],
            Out1.Arrays[8],
            Out2.Arrays[0],
            Out2.Arrays[1],
            Out2.Arrays[2],
            Out2.Arrays[3],
            Index, Length1, Length2, Length3);
    }

    SCFloatArrayRef Oscillator(SCFloatArrayRef In1, SCFloatArrayRef In2, SCFloatArrayRef Out, int Index)
    {
        return InternalOscillator(In1, In2, Out, Index);
    }

    SCFloatArrayRef Oscillator(SCFloatArrayRef In1, SCFloatArrayRef In2, SCFloatArrayRef Out)
    {
        return InternalOscillator(In1, In2, Out, Index);
    }

    int UseTool(s_UseTool& ToolDetails)
    {
        return Internal_UseDrawingTool(ToolDetails);
    }

    void CalculateRegressionStatistics(SCFloatArrayRef In, double &Slope, double &Y_Intercept, int Index, int Length)
    {
        return Internal_CalculateRegressionStatistics( In, Slope, Y_Intercept, Index, Length);
    }

    void CalculateRegressionStatistics(SCFloatArrayRef In, double &Slope, double &Y_Intercept, int Length)
    {
        return Internal_CalculateRegressionStatistics( In, Slope, Y_Intercept, Index, Length);
    }

    void CalculateLogLogRegressionStatistics(SCFloatArrayRef In, double &Slope, double &Y_Intercept, int Index, int
Length)
    {
        return Internal_CalculateLogLogRegressionStatistics(In, Slope, Y_Intercept, Index, Length);
    }

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

```

```

{
    return Internal_CalculateLogLogRegressionStatistics(In, Slope, Y_Intercept, Index, Length);
}

SCFloatArrayRef LinearRegressionIndicatorAndStdErr(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef
StdErr, int Index, int Length)
{
    return InternalLinearRegressionIndicatorAndStdErr( In, Out, StdErr, Index, Length);
}

SCFloatArrayRef LinearRegressionIndicatorAndStdErr(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef
StdErr, int Length)
{
    return InternalLinearRegressionIndicatorAndStdErr( In, Out, StdErr, Index, Length);
}

SCFloatArrayRef PriceVolumeTrend(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int Index)
{
    return InternalPriceVolumeTrend( ChartBaseDataIn, Out, Index);
}

SCFloatArrayRef PriceVolumeTrend(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out)
{
    return InternalPriceVolumeTrend( ChartBaseDataIn, Out, Index);
}

int NumberOfBarsSinceLowestValue(SCFloatArrayRef in, int Index, int Length)
{
    return InternalNumberOfBarsSinceLowestValue( in, Index, Length);
}

int NumberOfBarsSinceLowestValue(SCFloatArrayRef in, int Length)
{
    return InternalNumberOfBarsSinceLowestValue( in, Index, Length);
}

int NumberOfBarsSinceHighestValue(SCFloatArrayRef in, int Index, int Length)
{
    return InternalNumberOfBarsSinceHighestValue( in, Index, Length);
}

int NumberOfBarsSinceHighestValue(SCFloatArrayRef in, int Length)
{
    return InternalNumberOfBarsSinceHighestValue( in, Index, Length);
}

float GetCorrelationCoefficient(SCFloatArrayRef In1, SCFloatArrayRef In2, int Index, int Length)
{
    float value;
    value = InternalGetCorrelationCoefficient( In1, In2, Index, Length);
    return value;
}

float GetCorrelationCoefficient(SCFloatArrayRef In1, SCFloatArrayRef In2, int Length)
{
    float value;
    value = InternalGetCorrelationCoefficient( In1, In2, Index, Length);
    return value;
}

float GetTrueRange(SCBaseDataRef ChartBaseDataIn, int Index)
{
    return InternalGetTrueRange( ChartBaseDataIn, Index);
}

float GetTrueRange(SCBaseDataRef ChartBaseDataIn)
{
    return InternalGetTrueRange( ChartBaseDataIn, Index);
}

```



```

}

float GetTrueLow(SCBaseDataRef ChartBaseDataIn, int Index)
{
    return InternalGetTrueLow( ChartBaseDataIn, Index);
}
float GetTrueLow(SCBaseDataRef ChartBaseDataIn)
{
    return InternalGetTrueLow( ChartBaseDataIn, Index);
}

float GetTrueHigh(SCBaseDataRef ChartBaseDataIn, int Index)
{
    return InternalGetTrueHigh( ChartBaseDataIn, Index);
}
float GetTrueHigh(SCBaseDataRef ChartBaseDataIn)
{
    return InternalGetTrueHigh( ChartBaseDataIn, Index);
}

int GetIslandReversal(SCBaseDataRef ChartBaseDataIn, int Index)
{
    return InternalGetIslandReversal( ChartBaseDataIn, Index);
}

int GetIslandReversal(SCBaseDataRef ChartBaseDataIn)
{
    return InternalGetIslandReversal( ChartBaseDataIn, Index);
}

SCFloatArrayRef Parabolic(SCBaseDataRef ChartBaseDataIn, SCDateTimeArrayRef BaseDateTimeIn,
SCSubgraphRef Out, float InStartAccelFactor, float InAccelIncrement, float InMaxAccelFactor, unsigned int
InAdjustForGap, int InputDataHighIndex = SC_HIGH, int InputDataLowIndex = SC_LOW)
{
    s_Parabolic ParabolicData( ChartBaseDataIn, Out, BaseDateTimeIn, Index, InStartAccelFactor, InAccelIncrement,
InMaxAccelFactor, InAdjustForGap, InputDataHighIndex, InputDataLowIndex);

    return InternalParabolic(ParabolicData);
}

SCFloatArrayRef Parabolic(SCBaseDataRef ChartBaseDataIn, SCDateTimeArrayRef BaseDateTimeIn,
SCSubgraphRef Out, int Index, float InStartAccelFactor, float InAccelIncrement, float InMaxAccelFactor, unsigned int
InAdjustForGap, int InputDataHighIndex = SC_HIGH, int InputDataLowIndex = SC_LOW)
{
    s_Parabolic ParabolicData( ChartBaseDataIn, Out, BaseDateTimeIn, Index, InStartAccelFactor, InAccelIncrement,
InMaxAccelFactor, InAdjustForGap, InputDataHighIndex, InputDataLowIndex);

    return InternalParabolic(ParabolicData);
}

SCFloatArrayRef ZigZag(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCSubgraphRef Out, int
Index, float ReversalPercent, int StartIndex = 0)
{
    return InternalResettableZigZag(InputDataHigh, InputDataLow, Out, Out.Arrays[0], Out.Arrays[1], StartIndex, Index,
ReversalPercent, 0.0f, *this);
}

SCFloatArrayRef ZigZag(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCSubgraphRef Out, float
ReversalPercent, int StartIndex = 0)
{
    return InternalResettableZigZag(InputDataHigh, InputDataLow, Out, Out.Arrays[0], Out.Arrays[1], StartIndex, Index,
ReversalPercent, 0.0f, *this);
}

```

```

SCFloatArrayRef ZigZag(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCSubgraphRef Out, int
Index, float ReversalPercent, float ReversalAmount, int StartIndex = 0)
{
    return InternalResettableZigZag(InputDataHigh, InputDataLow, Out, Out.Arrays[0], Out.Arrays[1], StartIndex, Index,
ReversalPercent, ReversalAmount, *this);
}

```

```

SCFloatArrayRef ZigZag(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCSubgraphRef Out, float
ReversalPercent, float ReversalAmount, int StartIndex = 0)
{
    return InternalResettableZigZag(InputDataHigh, InputDataLow, Out, Out.Arrays[0], Out.Arrays[1], StartIndex, Index,
ReversalPercent, ReversalAmount, *this);
}

```

```

SCFloatArrayRef ZigZag2(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCSubgraphRef Out, int
Index, int NumberOfBars, float ReversalAmount, int StartIndex = 0)
{
    return InternalResettableZigZag2(InputDataHigh, InputDataLow, Out, Out.Arrays[0], Out.Arrays[1], StartIndex, Index,
NumberOfBars, ReversalAmount, *this);
}

```

```

SCFloatArrayRef ZigZag2(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCSubgraphRef Out, int
NumberOfBars, float ReversalAmount, int StartIndex = 0)
{
    return InternalResettableZigZag2(InputDataHigh, InputDataLow, Out, Out.Arrays[0], Out.Arrays[1], StartIndex, Index,
NumberOfBars, ReversalAmount, *this);
}

```

```

SCFloatArrayRef WilliamsR(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int Index, int Length)

```

```

{
    return InternalWilliamsR( ChartBaseDataIn, Out, Index, Length);
}

```

```

SCFloatArrayRef WilliamsR(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int Length)

```

```

{
    return InternalWilliamsR( ChartBaseDataIn, Out, Index, Length);
}

```

```

//Three Separate Input Arrays

```

```

SCFloatArrayRef WilliamsR(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCFloatArrayRef
InputDataLast, SCFloatArrayRef Out, int Index, int Length)

```

```

{
    return InternalWilliamsR2( InputDataHigh, InputDataLow, InputDataLast, Out, Index, Length);
}

```

```

//Three Separate Input Arrays without Index

```

```

SCFloatArrayRef WilliamsR(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCFloatArrayRef
InputDataLast, SCFloatArrayRef Out, int Length)

```

```

{
    return InternalWilliamsR2( InputDataHigh, InputDataLow, InputDataLast, Out, Index, Length);
}

```

```

SCFloatArrayRef WilliamsAD(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int Index)

```

```

{
    return InternalWilliamsAD( ChartBaseDataIn, Out, Index);
}

```

```

SCFloatArrayRef WilliamsAD(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out)

```

```

{
    return InternalWilliamsAD( ChartBaseDataIn, Out, Index);
}

```

```

SCFloatArrayRef VHF(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)

```

```

{
    return InternalVHF( In, Out, Index, Length);
}

```

```

SCFloatArrayRef VHF(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalVHF( In, Out, Index, Length);
}

float GetDispersion(SCFloatArrayRef In, int Index, int Length)
{
    return InternalGetDispersion( In, Index, Length);
}
float GetDispersion(SCFloatArrayRef In, int Length)
{
    return InternalGetDispersion( In, Index, Length);
}

SCFloatArrayRef Dispersion(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalDispersion( In, Out, Index, Length);
}
SCFloatArrayRef Dispersion(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalDispersion( In, Out, Index, Length);
}

SCFloatArrayRef Summation(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalSummation( In, Out, Index, Length);
}
SCFloatArrayRef Summation(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalSummation( In, Out, Index, Length);
}

SCFloatArrayRef ArmsEMV(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int volinc, int Index)
{
    return InternalArmsEMV( ChartBaseDataIn, Out, volinc, Index);
}
SCFloatArrayRef ArmsEMV(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int volinc)
{
    return InternalArmsEMV( ChartBaseDataIn, Out, volinc, Index);
}

SCFloatArrayRef CumulativeSummation(SCFloatArrayRef In, SCFloatArrayRef Out, int Index)
{
    return InternalCumulativeSummation( In, Out, Index);
}
SCFloatArrayRef CumulativeSummation(SCFloatArrayRef In, SCFloatArrayRef Out)
{
    return InternalCumulativeSummation( In, Out, Index);
}

float GetSummation(SCFloatArrayRef In,int Index,int Length)
{
    return InternalGetSummation( In, Index, Length);
}
float GetSummation(SCFloatArrayRef In,int Length)
{
    return InternalGetSummation( In, Index, Length);
}

SCFloatArrayRef VolumeWeightedMovingAverage(SCFloatArrayRef InData, SCFloatArrayRef InVolume,
SCFloatArrayRef Out, int Index, int Length)
{
    return InternalVolumeWeightedMovingAverage( InData, InVolume, Out, Index, Length);
}
SCFloatArrayRef VolumeWeightedMovingAverage(SCFloatArrayRef InData, SCFloatArrayRef InVolume,

```

```

SCFloatArrayRef Out, int Length)
{
    return InternalVolumeWeightedMovingAverage( InData, InVolume, Out, Index, Length);
}

SCFloatArrayRef WellesSum(SCFloatArrayRef In ,SCFloatArrayRef Out , int Index, int Length)
{
    return InternalWellesSum( In , Out , Index, Length);
}
SCFloatArrayRef WellesSum(SCFloatArrayRef In ,SCFloatArrayRef Out , int Length)
{
    return InternalWellesSum( In , Out , Index, Length);
}

SCFloatArrayRef TrueRange(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int Index)
{
    return InternalTrueRange( ChartBaseDataIn, Out, Index);
}
SCFloatArrayRef TrueRange(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out)
{
    return InternalTrueRange( ChartBaseDataIn, Out, Index);
}

int CalculateOHLCAverages(int Index)
{
    return InternalCalculateOHLCAverages(Index);
}
int CalculateOHLCAverages()
{
    return InternalCalculateOHLCAverages(Index);
}

SCFloatArrayRef StdError(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalStdError( In, Out, Index, Length);
}
SCFloatArrayRef StdError(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalStdError( In, Out, Index, Length);
}

SCFloatArrayRef MovingAverage(SCFloatArrayRef In, SCFloatArrayRef Out, unsigned int MovingAverageType, int
Index, int Length)
{
    return InternalMovingAverage( In, Out, MovingAverageType, Index, Length);
}
SCFloatArrayRef MovingAverage(SCFloatArrayRef In, SCFloatArrayRef Out, unsigned int MovingAverageType, int
Length)
{
    return InternalMovingAverage( In, Out, MovingAverageType, Index, Length);
}

SCFloatArrayRef MovingMedian(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int Length)
{
    return InternalMovingMedian( In, Out, Out.Arrays[0], Index, Length);
}
SCFloatArrayRef MovingMedian(SCFloatArrayRef In, SCSubgraphRef Out, int Length)
{
    return InternalMovingMedian( In, Out, Out.Arrays[0], Index, Length);
}

SCDateTime GetStartOfPeriodForDateTime(const SCDateTime& DateTime, unsigned int TimePeriodType, int
TimePeriodLength, int PeriodOffset, int NewPeriodAtBothSessionStarts)
{

```

```

    return InternalGetStartOfPeriodForDateTime(DateTime, TimePeriodType, TimePeriodLength, PeriodOffset,
NewPeriodAtBothSessionStarts);
}

SCDateTime GetStartOfPeriodForDateTime(const SCDateTime& DateTime, unsigned int TimePeriodType, int
TimePeriodLength, int PeriodOffset)
{
    return InternalGetStartOfPeriodForDateTime(DateTime, TimePeriodType, TimePeriodLength, PeriodOffset, 0);
}

SCFloatArrayRef OnBalanceVolumeShortTerm(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int
Length)
{
    return InternalOnBalanceVolumeShortTerm( ChartBaseDataIn, Out, Out.Arrays[0], Index, Length);
}
SCFloatArrayRef OnBalanceVolumeShortTerm(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Length)
{
    return InternalOnBalanceVolumeShortTerm( ChartBaseDataIn, Out, Out.Arrays[0], Index, Length);
}

SCFloatArrayRef OnBalanceVolume(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int Index)
{
    return InternalOnBalanceVolume( ChartBaseDataIn, Out, Index);
}
SCFloatArrayRef OnBalanceVolume(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out)
{
    return InternalOnBalanceVolume( ChartBaseDataIn, Out, Index);
}

SCFloatArrayRef StdDeviation(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalStdDeviation( In, Out, Index, Length);
}
SCFloatArrayRef StdDeviation(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalStdDeviation( In, Out, Index, Length);
}

SCFloatArrayRef WildersMovingAverage(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalWildersMovingAverage( In, Out, Index, Length);
}
SCFloatArrayRef WildersMovingAverage(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalWildersMovingAverage( In, Out, Index, Length);
}

SCFloatArrayRef SimpleMovAvg(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalSimpleMovAvg( In, Out, Index, Length);
}
SCFloatArrayRef SimpleMovAvg(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalSimpleMovAvg( In, Out, Index, Length);
}

SCFloatArrayRef AdaptiveMovAvg(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length, float
FastSmoothConst, float SlowSmoothConst)
{
    return InternalAdaptiveMovAvg( In, Out, Index, Length, FastSmoothConst, SlowSmoothConst);
}
SCFloatArrayRef AdaptiveMovAvg(SCFloatArrayRef In, SCFloatArrayRef Out, int Length, float FastSmoothConst, float
SlowSmoothConst)
{
    return InternalAdaptiveMovAvg( In, Out, Index, Length, FastSmoothConst, SlowSmoothConst);
}

```

```

}

SCFloatArrayRef LinearRegressionIndicator(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalLinearRegressionIndicator( In, Out, Index, Length);
}
SCFloatArrayRef LinearRegressionIndicator(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalLinearRegressionIndicator( In, Out, Index, Length);
}

SCFloatArrayRef ExponentialMovAvg(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalExponentialMovAvg( In, Out, Index, Length);
}
SCFloatArrayRef ExponentialMovAvg(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalExponentialMovAvg( In, Out, Index, Length);
}

float ArrayValueAtNthOccurrence (SCFloatArrayRef TrueFalseIn, SCFloatArrayRef ValueArrayIn, int Index, int
NthOccurrence )
{
    return InternalArrayValueAtNthOccurrence (TrueFalseIn, ValueArrayIn, Index, NthOccurrence );
}

float ArrayValueAtNthOccurrence (SCFloatArrayRef TrueFalseIn, SCFloatArrayRef ValueArrayIn, int NthOccurrence )
{
    return InternalArrayValueAtNthOccurrence (TrueFalseIn, ValueArrayIn, Index, NthOccurrence );
}

double BuyEntry(s_SCNewOrder& NewOrder, int DataArrayIndex)// Manual looping
{
    return InternalBuyEntry(NewOrder, DataArrayIndex);
}

double BuyEntry(s_SCNewOrder& NewOrder)//Automatic looping
{
    return InternalBuyEntry(NewOrder, Index);
}

double BuyOrder(s_SCNewOrder& r_NewOrder, int DataArrayIndex)//Manual looping
{
    // For different Symbol or Trade Account
    if ((!r_NewOrder.Symbol.IsEmpty()
        && (r_NewOrder.Symbol.CompareNoCase(GetTradeSymbol()) != 0))
        || (!r_NewOrder.TradeAccount.IsEmpty()
            && (r_NewOrder.TradeAccount.CompareNoCase(this->SelectedTradeAccount) != 0))
        )
    {
        return InternalSubmitOrder(r_NewOrder, DataArrayIndex, BSE_BUY);
    }

    return InternalBuyEntry(r_NewOrder, DataArrayIndex);
}

double BuyOrder(s_SCNewOrder& r_NewOrder)//Automatic looping
{
    return BuyOrder(r_NewOrder, this->Index);
}

double BuyExit(s_SCNewOrder& NewOrder, int DataArrayIndex)// Manual looping
{
    return InternalBuyExit(NewOrder, DataArrayIndex);
}

```

```

}

double BuyExit(s_SCNewOrder& NewOrder)//Automatic looping
{
    return InternalBuyExit(NewOrder, Index);
}

double SellEntry(s_SCNewOrder& NewOrder, int DataArrayIndex)// Manual looping
{
    return InternalSellEntry(NewOrder, DataArrayIndex);
}

double SellEntry(s_SCNewOrder& NewOrder)//Automatic looping
{
    return InternalSellEntry(NewOrder, Index);
}

double SellOrder(s_SCNewOrder& r_NewOrder, int DataArrayIndex)//Manual looping
{
    // For different Symbol or Trade Account
    if ((!r_NewOrder.Symbol.IsEmpty()
        && (r_NewOrder.Symbol.CompareNoCase(GetTradeSymbol()) != 0))
        || (!r_NewOrder.TradeAccount.IsEmpty()
        && (r_NewOrder.TradeAccount.CompareNoCase(this->SelectedTradeAccount) != 0))
        )
    {
        return InternalSubmitOrder(r_NewOrder, DataArrayIndex, BSE_SELL);
    }

    return InternalSellEntry(r_NewOrder, DataArrayIndex);
}

double SellOrder(s_SCNewOrder& r_NewOrder)//Automatic looping
{
    return SellOrder(r_NewOrder, this->Index);
}

double SellExit(s_SCNewOrder& NewOrder, int DataArrayIndex)// Manual looping
{
    return InternalSellExit(NewOrder, DataArrayIndex);
}

double SellExit(s_SCNewOrder& NewOrder)//Automatic looping
{
    return InternalSellExit(NewOrder, Index);
}

double SubmitOCOOrder(s_SCNewOrder& NewOrder)//Automatic looping
{
    if(NewOrder.OrderType == SCT_ORDERTYPE_OCO_BUY_STOP_SELL_STOP
        || NewOrder.OrderType == SCT_ORDERTYPE_OCO_BUY_STOP_LIMIT_SELL_STOP_LIMIT
        || NewOrder.OrderType == SCT_ORDERTYPE_OCO_BUY_LIMIT_SELL_LIMIT
        )
    {
        return InternalSubmitOCOOrder(NewOrder, Index);
    }
    else
    {
        //only allow this function to be used with OCO order types
        return SCTRADING_NOT_OCO_ORDER_TYPE;
    }
}

double SubmitOCOOrder(s_SCNewOrder& NewOrder, int BarIndex)

```

```

{
    if (NewOrder.OrderType == SCT_ORDERTYPE_OCO_BUY_STOP_SELL_STOP
        || NewOrder.OrderType == SCT_ORDERTYPE_OCO_BUY_STOP_LIMIT_SELL_STOP_LIMIT
        || NewOrder.OrderType == SCT_ORDERTYPE_OCO_BUY_LIMIT_SELL_LIMIT
    )
    {
        return InternalSubmitOCOOrder(NewOrder, BarIndex);
    }
    else
    {
        //only allow this function to be used with OCO order types
        return SCTRADING_NOT_OCO_ORDER_TYPE;
    }
}

int GetTradePosition(s_SCPositionData& PositionData)
{
    return InternalGetTradePosition(PositionData);
}

int IsSwingHigh(SCFloatArrayRef In, int Index, int Length)
{
    for(int IndexOffset = 1; IndexOffset <= Length; IndexOffset++)
    {
        if (FormattedEvaluate(In[Index], BasedOnGraphValueFormat, LESS_EQUAL_OPERATOR, In[Index -
IndexOffset], BasedOnGraphValueFormat)
            || FormattedEvaluate(In[Index], BasedOnGraphValueFormat, LESS_EQUAL_OPERATOR, In[Index +
IndexOffset], BasedOnGraphValueFormat) )
        {
            return 0;
        }
    }

    return 1;
}

int IsSwingHigh(SCFloatArrayRef In, int Length)
{
    return IsSwingHigh(In, Index, Length);
}

int IsSwingLow(SCFloatArrayRef In, int Index, int Length)
{
    for(int IndexOffset = 1; IndexOffset <= Length; IndexOffset++)
    {
        if (FormattedEvaluate(In[Index], BasedOnGraphValueFormat, GREATER_EQUAL_OPERATOR, In[Index -
IndexOffset], BasedOnGraphValueFormat)
            || FormattedEvaluate(In[Index], BasedOnGraphValueFormat, GREATER_EQUAL_OPERATOR, In[Index +
IndexOffset], BasedOnGraphValueFormat) )
        {
            return 0;
        }
    }

    return 1;
}

int IsSwingLow(SCFloatArrayRef In, int Length)
{
    return IsSwingLow(In, Index, Length);
}

```



```

SCFloatArrayRef AwesomeOscillator(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int Length1, int Length2)
{
    SCFloatArrayRef TempMA1 = Out.Arrays[0];
    SCFloatArrayRef TempMA2 = Out.Arrays[1];

    return InternalAwesomeOscillator(In, Out, TempMA1, TempMA2, Index, Length1, Length2);
}

SCFloatArrayRef AwesomeOscillator(SCFloatArrayRef In, SCSubgraphRef Out, int Length1, int Length2)
{
    SCFloatArrayRef TempMA1 = Out.Arrays[0];
    SCFloatArrayRef TempMA2 = Out.Arrays[1];

    return InternalAwesomeOscillator(In, Out, TempMA1, TempMA2, Index, Length1, Length2);
}

SCFloatArrayRef Slope(SCFloatArrayRef In, SCFloatArrayRef Out)
{
    return InternalSlope(In, Out, Index);
}

SCFloatArrayRef Slope(SCFloatArrayRef In, SCFloatArrayRef Out, int Index)
{
    return InternalSlope(In, Out, Index);
}

SCFloatArrayRef CumulativeDeltaVolume(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int
ResetCumulativeCalculation)
{
    return InternalCumulativeDeltaVolume(ChartBaseDataIn, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out, Index,
ResetCumulativeCalculation);
}

SCFloatArrayRef CumulativeDeltaVolume(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int
ResetCumulativeCalculation)
{
    return InternalCumulativeDeltaVolume(ChartBaseDataIn, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out, Index,
ResetCumulativeCalculation);
}

SCFloatArrayRef CumulativeDeltaTicks(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int
ResetCumulativeCalculation)
{
    return InternalCumulativeDeltaTicks(ChartBaseDataIn, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out, Index,
ResetCumulativeCalculation);
}

SCFloatArrayRef CumulativeDeltaTicks(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int
ResetCumulativeCalculation)
{
    return InternalCumulativeDeltaTicks(ChartBaseDataIn, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out, Index,
ResetCumulativeCalculation);
}

SCFloatArrayRef CumulativeDeltaTickVolume(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int
ResetCumulativeCalculation)
{
    return InternalCumulativeDeltaTickVolume(ChartBaseDataIn, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out, Index,
ResetCumulativeCalculation);
}

SCFloatArrayRef CumulativeDeltaTickVolume(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int
ResetCumulativeCalculation)
{
    return InternalCumulativeDeltaTickVolume(ChartBaseDataIn, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out, Index,
ResetCumulativeCalculation);
}

```

```

ResetCumulativeCalculation);
}

int Round(float Number)
{
    int IntegerResult = static_cast<int>(Number);

    if ((Number > 0.0) && ((Number-IntegerResult) >= .5))
        return ++IntegerResult;
    else if ((Number < 0.0) && ((Number - IntegerResult) <= -.5))
        return --IntegerResult;

    return IntegerResult;
}

int64_t Round64(double Number)
{
    int64_t IntegerResult = static_cast<int64_t>(Number);

    if ((Number > 0.0) && ((Number - IntegerResult) >= .5))
        return ++IntegerResult;
    else if ((Number < 0.0) && ((Number - IntegerResult) <= -.5))
        return --IntegerResult;

    return IntegerResult;
}

//rounds to the nearest TickSize
float RoundToTickSize(float Value)
{
    return static_cast<float>(RoundToTickSize(static_cast<double>(Value), static_cast<double>(TickSize)));
}

double RoundToTickSize(double Value)
{
    return RoundToTickSize(Value, static_cast<double>(TickSize));
}

float RoundToTickSize(float Value, float TickSize)
{
    if (isinf(Value))
        return 0;

    return static_cast<float>(RoundToTickSize(static_cast<double>(Value), static_cast<double>(TickSize)));
}

double RoundToTickSize(double Value, float TickSize)
{
    return RoundToTickSize(Value, static_cast<double>(TickSize));
}

double RoundToIncrement(double Value, float TickSize)
{
    return RoundToTickSize(Value, static_cast<double>(TickSize));
}

double RoundToTickSize(double Value, double TickSize)
{
    if (TickSize == 0 || Value == DBL_MAX)
        return Value; // cannot round

    double ClosestMult = static_cast<int>((Value / TickSize)) * TickSize;
    double Diff = Value - ClosestMult;

```

```

double DifferenceFromIncrement = TickSize * 0.5 - Diff;

double Result;
if (Value > 0.0 && DifferenceFromIncrement <= TickSize * 0.001)
    Result = ClosestMult + TickSize;
else if (Value < 0.0 && DifferenceFromIncrement <= TickSize * 0.001)
    Result = ClosestMult - TickSize;
else
    Result = ClosestMult;

return Result;
}

int64_t PriceValueToTicks(double PriceValue)
{
    return Round64(PriceValue / TickSize);
}

double TicksToPriceValue(int64_t Ticks)
{
    return Ticks * TickSize;
}

// This uses the Volume Value Format chart setting to determine the multiplier. This is used when volumes have
fractional values and are stored as an integer.
float MultiplierFromVolumeValueFormat() const
{
    float Multiplier;
    switch (VolumeValueFormat)
    {
        default:
        case 0: Multiplier = 1.0f; break;
        case 1: Multiplier = 0.1f; break;
        case 2: Multiplier = 0.01f; break;
        case 3: Multiplier = 0.001f; break;
        case 4: Multiplier = 0.0001f; break;
        case 5: Multiplier = 0.00001f; break;
        case 6: Multiplier = 0.000001f; break;
        case 7: Multiplier = 0.0000001f; break;
        case 8: Multiplier = 0.00000001f; break;
    }
    return Multiplier;
}

void SetAlert(int AlertNumber, int ArrayIndex, const SCString& Message = "")
{
    InternalSetAlert(AlertNumber, ArrayIndex, Message);
}

void SetAlert(int AlertNumber, const SCString& Message = "")
{
    SetAlert(AlertNumber, Index, Message);
}

int GetNumberOfBaseGraphArrays() const
{
    return BaseDataIn.GetArraySize();
}

SCString &GetRealTimeSymbol()
{
    if (TradeAndCurrentQuoteSymbol.GetLength() > 0)

```

```

    return TradeAndCurrentQuoteSymbol;
else
    return Symbol;
}

/*=====
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.
-----*/
uint32_t RGBInterpolate(const uint32_t& Color1, const uint32_t& Color2, float ColorDistance)
{
    return RGB(static_cast<int>(GetRValue(Color1)*(1.0f-(ColorDistance)) + GetRValue(Color2)*(ColorDistance) +
0.5f),
        static_cast<int>(GetGValue(Color1)*(1.0f-(ColorDistance)) + GetGValue(Color2)*(ColorDistance) + 0.5f),
        static_cast<int>(GetBValue(Color1)*(1.0f-(ColorDistance)) + GetBValue(Color2)*(ColorDistance) + 0.5f));
}

SCDateTime ConvertDateTimeToChartTimeZone(const SCDateTime& DateTime, const char* TimeZonePOSIXString)
{
    return InternalConvertDateTimeToChartTimeZone(DateTime, TimeZonePOSIXString);
}

SCDateTime ConvertDateTimeToChartTimeZone(const SCDateTime& DateTime, TimeZonesEnum TimeZone)
{
    // The reason we are passing a string instead of the enumeration
    // value is because the enumeration value could change if the list of
    // time zones ever changes, but the string will not change.
    return InternalConvertDateTimeToChartTimeZone(DateTime, GetPosixStringForTimeZone(TimeZone));
}

SCDateTime ConvertDateTimeFromChartTimeZone(const SCDateTime& DateTime, const char*
TimeZonePOSIXString)
{
    return InternalConvertDateTimeFromChartTimeZone(DateTime, TimeZonePOSIXString);
}

SCDateTime ConvertDateTimeFromChartTimeZone(const SCDateTime& DateTime, TimeZonesEnum TimeZone)
{
    return InternalConvertDateTimeFromChartTimeZone(DateTime, GetPosixStringForTimeZone(TimeZone));
}

SCFloatArrayRef DoubleStochastic(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Length, int
MovAvgLength, int MovingAverageType)
{
    InternalDoubleStochastic(ChartBaseDataIn, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Index,
Length, MovAvgLength, MovingAverageType);
    return Out;
}

SCFloatArrayRef DoubleStochastic(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef Out, int Index, int Length, int
MovAvgLength, int MovingAverageType)
{
    InternalDoubleStochastic(ChartBaseDataIn, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Index,
Length, MovAvgLength, MovingAverageType);
    return Out;
}

int GetStudyIDByIndex(int ChartNumber, int StudyNumber)
{
    return Internal_GetStudyIDByIndex(ChartNumber, StudyNumber);
}

```

```

double GetStandardError(SCFloatArrayRef In, int Index, int Length)
{
    return InternalGetStandardError(In, Index, Length);
}

double GetStandardError(SCFloatArrayRef In, int Length)
{
    return InternalGetStandardError(In, Index, Length);
}

SCFloatArrayRef AccumulationDistribution ( SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int Index)
{
    return InternalAccumulationDistribution(ChartBaseDataIn, Out, Index);
}

SCFloatArrayRef AccumulationDistribution ( SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out)
{
    return InternalAccumulationDistribution(ChartBaseDataIn, Out, Index);
}

int GetValueFormat()
{
    if(ValueFormat == VALUEFORMAT_INHERITED)
    {
        return BasedOnGraphValueFormat;
    }
    else
        return ValueFormat;
}

void* AllocateMemory(int Size)
{
    char* Pointer = (char*)HeapAlloc(GetProcessHeap(), 0, (SIZE_T)Size);

    if (Pointer != NULL)
        memset(Pointer, 0, Size);

    return Pointer;
}

void FreeMemory(void* Pointer)
{
    if (Pointer != NULL)
        HeapFree(GetProcessHeap(), 0, Pointer);
}

//This function is for Intraday charts and determines if the given BarIndex is the beginning of a new trading day
according to the Session Times for the chart.
bool IsNewTradingDay(int BarIndex) const
{
    SCDateTime CurrentBarTradingDayStartDateTime =
GetTradingDayStartDateTimeOfBar(BaseDateIn[BarIndex]);
    SCDateTime PriorBarTradingDayStartDateTime = GetTradingDayStartDateTimeOfBar(BaseDateIn[BarIndex
-1]);

    return PriorBarTradingDayStartDateTime != CurrentBarTradingDayStartDateTime;

}

const char * GetTradingErrorMessage(int ErrorCode)
{
    switch (ErrorCode)
    {
        case SCTRADING_ORDER_ERROR:

```

```

        return "General order error. Refer to 'Trade >> Trade Service Log' for specific message for this trading action
error";
    case SCTRADING_NOT_OCO_ORDER_TYPE:
        return "Not an OCO order type";
    case SCTRADING_ATTACHED_ORDER_OFFSET_NOT_SUPPORTED_WITH_MARKET_PARENT:
        return "Attached order offset not supported with a parent market order. Use actual price instead";
    case SCTRADING_UNSUPPORTED_ATTACHED_ORDER:
        return "Unsupported Attached Order";
    case SCTRADING_SYMBOL_SETTINGS_NOT_FOUND:
        return "Symbol Settings not found for symbol";
    case ACSIL_GENERAL_NULL_POINTER_ERROR:
        return "General null pointer error";
    case SCT_SKIPPED_DOWNLOADING_HISTORICAL_DATA:
        return "Order entry skipped because downloading historical data";
    case SCT_SKIPPED_FULL_RECALC:
        return "Order entry skipped because full recalculation";
    case SCT_SKIPPED_ONLY_ONE_TRADE_PER_BAR:
        return "Order entry skipped because only one trade per bar is allowed";
    case SCT_SKIPPED_INVALID_INDEX_SPECIFIED:
        return "Order entry skipped because invalid bar index specified";
    case SCT_SKIPPED_TOO_MANY_NEW_BARS_DURING_UPDATE:
        return "Order entry skipped because too many new bars during chart update";
    case SCT_SKIPPED_AUTO_TRADING_DISABLED:
        return "Order entry skipped because automated trading is disabled";
    case SCTRADING_UNSUPPORTED_ORDER_TYPE:
        return "Unsupported order type";
    case SCTRADING_ERROR_SETTING_ORDER_PRICES:
        return "Error setting order prices";
    default:
        return "unknown order error";
    }
}

void GetStudyArrayFromChartUsingID(const s_ChartStudySubgraphValues& ChartStudySubgraphValues,
SCFloatArrayRef SubgraphArray)
{
    InternalGetStudyArrayFromChartUsingID(ChartStudySubgraphValues, SubgraphArray);
}

void GetStudyArrayFromChartUsingID(int ChartNumber, int StudyID , int SubgraphIndex, SCFloatArrayRef
SubgraphArray)
{
    s_ChartStudySubgraphValues ChartStudySubgraphValues;
    ChartStudySubgraphValues.ChartNumber = ChartNumber;
    ChartStudySubgraphValues.StudyID = StudyID;
    ChartStudySubgraphValues.SubgraphIndex = SubgraphIndex;

    InternalGetStudyArrayFromChartUsingID(ChartStudySubgraphValues, SubgraphArray);
}

const SCString& GetTradeSymbol() const
{
    if (!TradeAndCurrentQuoteSymbol.IsEmpty())
        return TradeAndCurrentQuoteSymbol;

    return Symbol;
}

bool IsDateTimeContainedInBarIndex(const SCDatetime& DateTime, int BarIndex) const
{
    SCDatetime BeginDateTime = BaseDateTimeIn[BarIndex];
    SCDatetime EndDateTime;

    if (BarIndex < ArraySize - 1)

```

```

        EndDateTime = BaseDateTimeIn[BarIndex + 1];
    else
        EndDateTime = DateTimeOfLastFileRecord;

    return DateTime >= BeginDateTime && DateTime < EndDateTime;
}

void HeikinAshi(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef HeikinAshiOut, int Index, int Length, int
SetCloseToCurrentPriceAtLastBar)
{
    InternalHeikinAshi(ChartBaseDataIn, Index, Length, HeikinAshiOut.Data, HeikinAshiOut.Arrays[0],
HeikinAshiOut.Arrays[1], HeikinAshiOut.Arrays[2], SetCloseToCurrentPriceAtLastBar);
}

void HeikinAshi(SCBaseDataRef ChartBaseDataIn, SCSubgraphRef HeikinAshiOut, int Length, int
SetCloseToCurrentPriceAtLastBar)
{
    InternalHeikinAshi(ChartBaseDataIn, Index, Length, HeikinAshiOut.Data, HeikinAshiOut.Arrays[0],
HeikinAshiOut.Arrays[1], HeikinAshiOut.Arrays[2], SetCloseToCurrentPriceAtLastBar);
}

//Parameters reviewed for safety with different compilers.
bool IsDateTimeContainedInBarAtIndex(const SCDatetime& DateTime, int BarIndex)
{
    return DateTime >= BaseDateTimeIn[BarIndex] && DateTime < BaseDateTimeIn[BarIndex+1];
}

void InverseFisherTransform(SCFloatArrayRef In, SCSubgraphRef Out, int HighestLowestLength, int
MovingAverageLength, int MovAvgType)
{
    Internal_InverseFisherTransform(In, Out, Out.Arrays[0], Out.Arrays[1], Index, HighestLowestLength,
MovingAverageLength, MovAvgType);
}

void InverseFisherTransform(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int HighestLowestLength, int
MovingAverageLength, int MovAvgType)
{
    Internal_InverseFisherTransform(In, Out, Out.Arrays[0], Out.Arrays[1], Index, HighestLowestLength,
MovingAverageLength, MovAvgType);
}

void InverseFisherTransformRSI(SCFloatArrayRef In, SCSubgraphRef Out, int RSILength, int
InternalRSIMovAvgType, int RSIMovingAverageLength, int MovingAverageOfRSIType)
{
    Internal_InverseFisherTransformRSI(In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3],
Out.Arrays[4], Out.Arrays[5], Out.Arrays[6], Index, RSILength, InternalRSIMovAvgType, RSIMovingAverageLength,
MovingAverageOfRSIType);
}

void InverseFisherTransformRSI(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int RSILength, int
InternalRSIMovAvgType, int RSIMovingAverageLength, int MovingAverageOfRSIType)
{
    Internal_InverseFisherTransformRSI(In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3],
Out.Arrays[4], Out.Arrays[5], Out.Arrays[6], Index, RSILength, InternalRSIMovAvgType, RSIMovingAverageLength,
MovingAverageOfRSIType);
}

void MovingAverageCumulative(SCFloatArrayRef In, SCFloatArrayRef Out)
{
    Internal_MovingAverageCumulative(In, Out, Index);
}

```

```

}

void MovingAverageCumulative(SCFloatArrayRef In, SCFloatArrayRef Out, int Index)
{
    Internal_MovingAverageCumulative(In, Out, Index);
}

SCFloatArrayRef HurstExponent(SCFloatArrayRef In, SCSubgraphRef Out, int LengthIndex)
{
    return Internal_HurstExponent(In, Out, Index, LengthIndex);
}

SCFloatArrayRef HurstExponent(SCFloatArrayRef In, SCSubgraphRef Out, int Index, int LengthIndex)
{
    return Internal_HurstExponent(In, Out, Index, LengthIndex);
}

int BarIndexToRelativeHorizontalCoordinate(int BarIndex, bool UseHighResolutionCoordinate)
{
    //This function is not always guaranteed to give an accurate result. It can be an inaccurate when a chart is first
    loaded

    const double MaxRelativeDrawingPosition = UseHighResolutionCoordinate ?
    CHART_DRAWING_MAX_HORIZONTAL_AXIS_RELATIVE_POSITION_HIGH_RES :
    CHART_DRAWING_MAX_HORIZONTAL_AXIS_RELATIVE_POSITION;

    if (BarIndex < IndexOfFirstVisibleBar)
        return 0;

    if (BarIndex > IndexOfLastVisibleBar)
        return static_cast<int>(MaxRelativeDrawingPosition);

    int NumberOfBars = IndexOfLastVisibleBar - IndexOfFirstVisibleBar + 1 + NumFillSpaceBars;

    float Percent = (BarIndex - IndexOfFirstVisibleBar) / static_cast<float>(NumberOfBars);

    return static_cast<int>(Percent * MaxRelativeDrawingPosition);
}

void FillSubgraphElementsWithLinearValuesBetweenBeginEndValues(int SubgraphIndex, int BeginIndex, int
EndIndex)
{
    if (BeginIndex >= EndIndex)
        return;

    int Distance = EndIndex - BeginIndex;

    double Difference = Subgraph[SubgraphIndex].Data[EndIndex] - Subgraph[SubgraphIndex].Data[BeginIndex];
    double Increment = Difference / Distance;

    for (int BarIndex = BeginIndex + 1; BarIndex < EndIndex; BarIndex++)
    {
        int Position = BarIndex - BeginIndex;
        Subgraph[SubgraphIndex].Data[BarIndex] = static_cast<float>(Position*Increment +
Subgraph[SubgraphIndex].Data[BeginIndex]);
    }
}

bool IsNewBar(int BarIndex)
{
    if (IsFullRecalculation || UpdateStartIndex == 0)
        return false;
}

```



```
//This function is not supported for automatic looping
if (AutoLoop)
    return false;

return BarIndex > UpdateStartIndex;
}
/*=====*/
/* This 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 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

```
*/
void GetTimeSalesArrayIndexesForBarIndex(int BarIndex, int& r_BeginIndex, int& r_EndIndex)
{
    //-1 indicates there are no valid indexes in the time and sales array for the given BarIndex.
    r_BeginIndex = -1;
    r_EndIndex = -1;

    c_SCTimeAndSalesArray TimeAndSalesArray;
    GetTimeAndSales(TimeAndSalesArray);
    int TSSize = TimeAndSalesArray.Size();
    if (TSSize == 0)
        return;

    SCDatetime BarBeginDateTime = BaseDateTimeIn[BarIndex];
    SCDatetime BarEndDateTime = GetEndingDateTimeForBarIndex(BarIndex);
    for (int TSIndex = TSSize - 1; TSIndex >= 0; --TSIndex)
    {
        // Adjust a time and sales record date-time to the time zone of the chart
        SCDatetime RecordDateTime = TimeAndSalesArray[TSIndex].DateTime;

        RecordDateTime += TimeScaleAdjustment;

        if (TSIndex == TSSize - 1 //at the end of the array
            && RecordDateTime <= BarEndDateTime && RecordDateTime >= BarBeginDateTime)
            r_EndIndex = TSIndex;

        if (TSIndex <= TSSize - 1)
        {
            SCDatetime NextRecordDateTime = TimeAndSalesArray[TSIndex + 1].DateTime;
            NextRecordDateTime += TimeScaleAdjustment;

            if (RecordDateTime <= BarEndDateTime && NextRecordDateTime > BarEndDateTime
                && RecordDateTime >= BarBeginDateTime)
                r_EndIndex = TSIndex;
        }

        if (TSIndex == 0 //at the beginning of the array
            && RecordDateTime >= BarBeginDateTime && RecordDateTime <= BarEndDateTime)
            r_BeginIndex = TSIndex;

        if (TSIndex > 0)
        {
            SCDatetime PreviousRecordDateTime = TimeAndSalesArray[TSIndex - 1].DateTime;
            PreviousRecordDateTime += TimeScaleAdjustment;

            if (RecordDateTime >= BarBeginDateTime && PreviousRecordDateTime < BarBeginDateTime
                && RecordDateTime <= BarEndDateTime)
                r_BeginIndex = TSIndex;
        }
    }
}
```

```

        if (r_BeginIndex != -1 && r_EndIndex != -1)
            break;
    }

    return;
}

/*=====*/
void T3MovingAverage(SCFloatArrayRef In, SCSubgraphRef Out, float Multiplier, int Length)
{
    Internal_T3MovingAverage(In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Out.Arrays[4],
    Out.Arrays[5], Multiplier, Index, Length);
}

/*=====*/
void T3MovingAverage(SCFloatArrayRef In, SCSubgraphRef Out, float Multiplier, int Index, int Length)
{
    Internal_T3MovingAverage(In, Out, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Out.Arrays[4],
    Out.Arrays[5], Multiplier, Index, Length);
}

/*=====*/
void CalculateAngle(SCFloatArrayRef InputArray, SCFloatArrayRef OutputArray, int Length, float ValuePerPoint)
{
    Internal_CalculateAngle(InputArray, OutputArray, Index, Length, ValuePerPoint);
}

/*=====*/
void CalculateAngle(SCFloatArrayRef InputArray, SCFloatArrayRef OutputArray, int Index, int Length, float
ValuePerPoint)
{
    Internal_CalculateAngle(InputArray, OutputArray, Index, Length, ValuePerPoint);
}

/*=====*/

int GetBarsSinceLastTradeOrderEntry()
{
    s_SCPositionData PositionData;
    GetTradePosition(PositionData);

    if (PositionData.LastEntryDateTime.IsUnset())
        return 0;

    int BarsSinceEntry = ArraySize - GetContainingIndexForSCDateTime(ChartNumber,
    PositionData.LastEntryDateTime) - 1;

    return BarsSinceEntry;
}

/*=====*/
int GetBarsSinceLastTradeOrderExit()
{
    s_SCPositionData PositionData;
    GetTradePosition(PositionData);

    if (PositionData.LastExitDateTime.IsUnset())
        return 0;

    int BarsSinceExit = ArraySize - GetContainingIndexForSCDateTime(ChartNumber, PositionData.LastExitDateTime)
- 1;

    return BarsSinceExit;
}

/*=====*/
//Parameters reviewed for safety with different compilers.
bool IsIsSufficientTimePeriodInDate(const SCDateTime& DateTime, float Percentage)

```

```

{
    SCDatetime StartDateTime = GetTradingDayStartDateTimeOfBar(DateTime);
    SCDatetime EndDateTime = StartDateTime + SCDatetime::HOURS(24) - SCDatetime::SECONDS(1);
    int FirstIndex = GetContainingIndexForSCDateTime(ChartNumber, StartDateTime);
    int LastIndex = GetContainingIndexForSCDateTime(ChartNumber, EndDateTime);

    const SCDatetime ActualTimeSpan = CalculateTimeSpanAcrossChartBars(FirstIndex, LastIndex);

    //If there is a sufficient amount of data in this time period
    return (ActualTimeSpan >= SCDatetime(Percentage * 0.01));
}

/*=====*/
SCDatetime TimeSpanOfBar( int BarIndex)
{
    //this function requires that sc.MaintainAdditionalChartDataArrays = 1 be set.
    return (BaseDataEndDateTime[BarIndex] - BaseDataTimeIn[BarIndex] ) + SCDatetime::SECONDS(1);
}

/*=====*/

double SlopeToAngleInDegrees(double Slope)
{
    return atan(Slope) * 180.0 / M_PI;
}

/*=====*/

double AngleInDegreesToSlope(double AngleInDegrees)
{
    return tan(AngleInDegrees*M_PI / 180);
}

/*=====*/

void AddAndManageSingleTextUserDrawnDrawingForStudy(SCStudyInterfaceRef &sc, int Unused, int
HorizontalPosition, int VerticalPosition, SCSubgraphRef Subgraph, int TransparentLabelBackground, SCString&
TextToDisplay, int DrawAboveMainPriceGraph, int LockDrawing, int UseBoldFont = 0)
{
    int& r_DrawingTextLineNumber = *static_cast<int*>(sc.StorageBlock);

    if (sc.LastCallToFunction)
    {
        // If the Chartbook is being closed, the Chartbook will have already been saved first by the time we enter this
        block. So this particular chart drawing if it exists, will have already been saved.
        if (r_DrawingTextLineNumber != 0
            && sc.UserDrawnChartDrawingExists(sc.ChartNumber, r_DrawingTextLineNumber) > 0)
        {
            // be sure to delete user drawn type drawing when study removed
            sc.DeleteUserDrawnACSDrawing(sc.ChartNumber, r_DrawingTextLineNumber);
        }

        return;
    }

    //Reset the line number if the drawing no longer exists.
    if (r_DrawingTextLineNumber != 0
        && sc.UserDrawnChartDrawingExists(sc.ChartNumber, r_DrawingTextLineNumber) == 0)
    {
        r_DrawingTextLineNumber = 0;
    }
}

```

```

}

if (sc.HideStudy && r_DrawingTextLineNumber != 0)
{
    sc.DeleteUserDrawnACSDrawing(sc.ChartNumber, r_DrawingTextLineNumber);
    r_DrawingTextLineNumber = 0;
}

if (sc.HideStudy)
    return;

s_UseTool Tool;
Tool.ChartNumber = sc.ChartNumber;
Tool.DrawingType = DRAWING_TEXT;
Tool.Region = sc.GraphRegion;
Tool.AddMethod = UTAM_ADD_OR_ADJUST;
Tool.AddAsUserDrawnDrawing = 1;
Tool.AllowSaveToChartbook = 1;

bool CreateNeeded = r_DrawingTextLineNumber == 0;

if (CreateNeeded)
{
    Tool.BeginDateTime = HorizontalPosition;
    Tool.BeginValue = static_cast<float>(VerticalPosition);
}

Tool.UseRelativeVerticalValues = 1;
Tool.Color = Subgraph.PrimaryColor;
Tool.FontBackColor = Subgraph.SecondaryColor;
Tool.TransparentLabelBackground = TransparentLabelBackground;

Tool.ReverseTextColor = 0;
Tool.FontBold = UseBoldFont;
Tool.FontSize = Subgraph.LineWidth;
Tool.FontFace = sc.GetChartTextFontFaceName();

Tool.Text = TextToDisplay;

Tool.DrawUnderneathMainGraph = DrawAboveMainPriceGraph ? 0 : 1;

if (r_DrawingTextLineNumber != 0)
    Tool.LineNumber = r_DrawingTextLineNumber;

if (sc.UseTool(Tool) > 0)
    r_DrawingTextLineNumber = Tool.LineNumber;
}

/*=====*/
void AddAndManageSingleTextDrawingForStudy(SCStudyInterfaceRef &sc, bool DisplayInFillSpace, int
HorizontalPosition, int VerticalPosition, SCSubgraphRef Subgraph, int TransparentLabelBackground, SCString&
TextToDisplay, int DrawAboveMainPriceGraph, int BoldFont = 1 )
{
    int& r_DrawingTextLineNumber = sc.GetPersistentInt(11110000);

    if (sc.IsFullRecalculation)
        r_DrawingTextLineNumber = 0;

    //Reset the line number if the drawing no longer exists.
    if (r_DrawingTextLineNumber != 0
        && sc.ChartDrawingExists(sc.ChartNumber, r_DrawingTextLineNumber) == 0)
    {
        r_DrawingTextLineNumber = 0;
    }
}

```

```

if (sc.HideStudy && r_DrawingTextLineNumber != 0)
{
    sc.DeleteACSCChartDrawing(sc.ChartNumber, TOOL_DELETE_CHARTDRAWING, r_DrawingTextLineNumber);
    r_DrawingTextLineNumber = 0;
}

if (sc.HideStudy)
    return;

s_UseTool Tool;
Tool.ChartNumber = sc.ChartNumber;
Tool.DrawingType = DRAWING_TEXT;
Tool.Region = sc.GraphRegion;
Tool.AddMethod = UTAM_ADD_OR_ADJUST;

if (!DisplayInFillSpace)
    Tool.BeginDateTime = HorizontalPosition;
else
    Tool.BeginDateTime = -3;

Tool.BeginValue = static_cast<float>(VerticalPosition);

Tool.UseRelativeVerticalValues = 1;
Tool.Color = Subgraph.PrimaryColor;
Tool.FontBackColor = Subgraph.SecondaryColor;
Tool.TransparentLabelBackground = TransparentLabelBackground;

Tool.ReverseTextColor = 0;
Tool.FontBold = BoldFont;
Tool.FontSize = Subgraph.LineWidth;
Tool.FontFace = sc.GetChartTextFontFaceName();

Tool.Text = TextToDisplay;

Tool.DrawUnderneathMainGraph = DrawAboveMainPriceGraph ? 0 : 1;

if (r_DrawingTextLineNumber != 0)
    Tool.LineNumber = r_DrawingTextLineNumber;

if (sc.UseTool(Tool) > 0)
    r_DrawingTextLineNumber = Tool.LineNumber;
}

/*=====*/
double GetSymbolDataValue(SymbolDataValuesEnum ValueToReturn, const SCString& SymbolForData = SCString(),
bool SubscribeToMarketData = false, bool SubscribeToMarketDepth = false)
{
    return Internal_GetSymbolDataValue(ValueToReturn, SymbolForData.GetChars(), SubscribeToMarketData ? 1 : 0,
SubscribeToMarketDepth ? 1 : 0);
}

/*=====*/
void OrderQuantityToString(const double Value, SCString& OutputString)
{
    OutputString.Format("%g", Value);
}

/*=====*/
double CreateDoublePrecisionPrice(const float PriceValue)
{
    return Round(PriceValue / TickSize)*TickSize;
}

/*=====*/
bool IsVisibleSubgraphDrawStyle(int DrawStyle)

```

```

{
    return DrawStyle != DRAWSTYLE_HIDDEN && DrawStyle != DRAWSTYLE_IGNORE;
}

SCFloatArrayRef ExampleFunction(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return InternalExampleFunction(In, Out, Index, Length);
}

SCFloatArrayRef ArnaudLegouxMovingAverage(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam, int
Length, float Sigma, float Offset)
{
    return InternalArnaudLegouxMovingAverage(In, Out, IndexParam, Length, Sigma, Offset);
}
SCFloatArrayRef ArnaudLegouxMovingAverage(SCFloatArrayRef In, SCFloatArrayRef Out, int Length, float Sigma,
float Offset)
{
    return InternalArnaudLegouxMovingAverage(In, Out, Index, Length, Sigma, Offset);
}

SCFloatArrayRef ExponentialRegressionIndicator(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam, int
Length)
{
    return InternalExponentialRegressionIndicator(In, Out, IndexParam, Length);
}
SCFloatArrayRef ExponentialRegressionIndicator(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalExponentialRegressionIndicator(In, Out, Index, Length);
}

SCFloatArrayRef InstantaneousTrendline(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam, int Length)
{
    return InternalInstantaneousTrendline(In, Out, IndexParam, Length);
}
SCFloatArrayRef InstantaneousTrendline(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalInstantaneousTrendline(In, Out, Index, Length);
}

SCFloatArrayRef CyberCycle(SCFloatArrayRef In, SCFloatArrayRef Smoothed, SCFloatArrayRef Out, int IndexParam,
int Length)
{
    return InternalCyberCycle(In, Smoothed, Out, IndexParam, Length);
}
SCFloatArrayRef CyberCycle(SCFloatArrayRef In, SCFloatArrayRef Smoothed, SCFloatArrayRef Out, int Length)
{
    return InternalCyberCycle(In, Smoothed, Out, Index, Length);
}

SCFloatArrayRef FourBarSymmetricalFIRFilter(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam)
{
    return InternalFourBarSymmetricalFIRFilter(In, Out, IndexParam);
}
SCFloatArrayRef FourBarSymmetricalFIRFilter(SCFloatArrayRef In, SCFloatArrayRef Out)
{
    return InternalFourBarSymmetricalFIRFilter(In, Out, Index);
}

SCFloatArrayRef SuperSmoother2Pole(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam, int Length)
{
    return InternalSuperSmoother2Pole(In, Out, IndexParam, Length);
}
SCFloatArrayRef SuperSmoother2Pole(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalSuperSmoother2Pole(In, Out, Index, Length);
}

```

```

}

SCFloatArrayRef SuperSmoother3Pole(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam, int Length)
{
    return InternalSuperSmoother3Pole(In, Out, IndexParam, Length);
}
SCFloatArrayRef SuperSmoother3Pole(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalSuperSmoother3Pole(In, Out, Index, Length);
}

SCFloatArrayRef ZeroLagEMA(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam, int Length)
{
    return InternalZeroLagEMA(In, Out, IndexParam, Length);
}
SCFloatArrayRef ZeroLagEMA(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalZeroLagEMA(In, Out, Index, Length);
}

SCFloatArrayRef Butterworth2Pole(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam, int Length)
{
    return InternalButterworth2Pole(In, Out, IndexParam, Length);
}
SCFloatArrayRef Butterworth2Pole(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalButterworth2Pole(In, Out, Index, Length);
}

SCFloatArrayRef Butterworth3Pole(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam, int Length)
{
    return InternalButterworth3Pole(In, Out, IndexParam, Length);
}
SCFloatArrayRef Butterworth3Pole(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalButterworth3Pole(In, Out, Index, Length);
}

SCFloatArrayRef DominantCyclePeriod(SCFloatArrayRef In, SCSubgraphRef Out, int IndexParam, int MedianLength)
{
    return InternalDominantCyclePeriod(In, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Out.Arrays[4],
    Out.Arrays[5], Out.Arrays[6], Out.Data, IndexParam, MedianLength);
}

SCFloatArrayRef DominantCyclePeriod(SCFloatArrayRef In, SCSubgraphRef Out, int MedianLength)
{
    return InternalDominantCyclePeriod(In, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Out.Arrays[4],
    Out.Arrays[5], Out.Arrays[6], Out.Data, Index, MedianLength);
}

SCFloatArrayRef LaguerreFilter(SCFloatArrayRef In, SCSubgraphRef Out, int IndexParam, float DampingFactor)
{
    return InternalLaguerreFilter(In, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Out.Data, IndexParam,
    DampingFactor);
}

SCFloatArrayRef LaguerreFilter(SCFloatArrayRef In, SCSubgraphRef Out, float DampingFactor)
{
    return InternalLaguerreFilter(In, Out.Arrays[0], Out.Arrays[1], Out.Arrays[2], Out.Arrays[3], Out.Data, Index,
    DampingFactor);
}

SCFloatArrayRef DominantCyclePhase(SCFloatArrayRef In, SCSubgraphRef Out, int IndexParam)
{
    return InternalDominantCyclePhase(In, Out, Index);
}

```

```

}

SCFloatArrayRef DominantCyclePhase(SCFloatArrayRef In, SCSubgraphRef Out)
{
    return InternalDominantCyclePhase(In, Out, Index);
}

SCFloatArrayRef LinearRegressionSlope(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam, int Length)
{
    return InternalLinearRegressionSlope(In, Out, IndexParam, Length);
}

SCFloatArrayRef LinearRegressionSlope(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalLinearRegressionSlope(In, Out, Index, Length);
}

SCFloatArrayRef LinearRegressionIntercept(SCFloatArrayRef In, SCFloatArrayRef Out, int IndexParam, int Length)
{
    return InternalLinearRegressionIntercept(In, Out, IndexParam, Length);
}

SCFloatArrayRef LinearRegressionIntercept(SCFloatArrayRef In, SCFloatArrayRef Out, int Length)
{
    return InternalLinearRegressionIntercept(In, Out, Index, Length);
}

/*=====*/
void FillSubGraphBetweenBeginEndPoints(const int SubGraphIndex, const int BeginBarIndex, const int EndBarIndex,
const double BeginYValue, const double EndYValue)
{
    if (SubGraphIndex < 0)
        return;

    if (SubGraphIndex >= MAX_STUDY_SUBGRAPHS)
        return;

    const int NumberOfBars = EndBarIndex - BeginBarIndex;

    if (NumberOfBars == 0)
        return;

    SCFloatArray& r_SubGraphDataArray = Subgraph[SubGraphIndex].Data;

    double IncrementPerBar = (EndYValue - BeginYValue) / (NumberOfBars);
    double StartYValue = BeginYValue;

    for (int BarIndex = BeginBarIndex; BarIndex <= EndBarIndex; BarIndex++)
    {
        r_SubGraphDataArray[BarIndex] = static_cast<float>(StartYValue);
        StartYValue += IncrementPerBar;
    }
}

/*=====*/
int32_t GetGraphicsSetting(const int32_t ChartNumber, const n_ACSIL::GraphicsSettingsEnum GraphicsSetting,
uint32_t& r_Color, uint32_t& r_LineWidth, SubgraphLineStyles& r_LineStyle)
{
    return Internal_GetGraphicsSetting(ChartNumber, GraphicsSetting, r_Color, r_LineWidth, r_LineStyle);
}

/*=====*/
int32_t SetGraphicsSetting(const int32_t ChartNumber, const n_ACSIL::GraphicsSettingsEnum GraphicsSetting, const
uint32_t Color = -1, const uint32_t LineWidth = -1, const SubgraphLineStyles LineStyle = LINESTYLE_UNSET)
{
    return Internal_SetGraphicsSetting(ChartNumber, GraphicsSetting, Color, LineWidth, LineStyle);
}

```



```

}

/*=====*/

/*****/
// Data Members

int ArraySize;

int DrawZeros;
int DataStartIndex;
int UpdateStartIndex;

int GraphRegion;

int FreeDLL; // Unused. Remove
float Bid;
float Ask;

uint32_t DailyVolume;
int ChartNumber;
int ChartDataType; // 1 = historical daily data, 2 = intraday data

int BidSize;
int AskSize;
int LastSize;

int (SCDLLCALL* GetNearestMatchForDateTimeIndex)(int ChartNumber, int CallingChartIndex);

int (SCDLLCALL* ChartDrawingExists)(int ChartNumber, int LineNumber);

SCString (SCDLLCALL* VersionNumber)();

void (SCDLLCALL* p_AddMessageToLog)(const char* Message, int ShowLog);

int (SCDLLCALL* DeleteACSCChartDrawing)(int ChartNumber, int Tool, int LineNumber);

void * ChartWindowHandle;
unsigned int ProcessIdentifier;
int LastCallToFunction;

void* StorageBlock = nullptr; // This is a permanent storage for any data type

float DailyHigh;
float DailyLow;

unsigned int NumFillSpaceBars;
unsigned int PreserveFillSpace; // Boolean
SCDateTime TimeScaleAdjustment;

int (SCDLLCALL* GetNearestMatchForSCDateTime)(int ChartNumber, const SCDateTime& DateTime);
int (SCDLLCALL* GetContainingIndexForDateTimeIndex)(int ChartNumber, int CallingChartIndex);
int (SCDLLCALL* GetContainingIndexForSCDateTime)(int ChartNumber, const SCDateTime& DateTime);

short GraphicalDisplacement = 0;

int SetDefaults = 0; // Boolean

int UpdateAlways = 0; // Boolean
unsigned int ActiveToolIndex = 0;
int ActiveToolYPosition = 0;

int IndexOfLastVisibleBar = 0;

```

```

unsigned int ChartBackgroundColor = 0; // Unused
SCString (SCDLLCALL* ChartTextFont)();
int StudyRegionTopCoordinate = 0;
int StudyRegionBottomCoordinate = 0;

int IsCustomChart = 0; // Boolean
int NumberOfArrays = 0;

int OutArraySize = 0;
const char* (SCDLLCALL* InternalGetStudyName)(int StudyIndex);

int (SCDLLCALL* Internal_UseDrawingTool)(s_UseTool& ToolDetails);

uint16_t ScaleType; // ScaleTypeEnum
uint16_t ScaleRangeType; // ScaleRangeTypeEnum
float ScaleRangeTop;
float ScaleRangeBottom;
float ScaleIncrement;
float ScaleConstRange;

float BaseGraphScaleIncrement;
float BaseGraphHorizontalGridLineIncrement ;

SCString GraphName;
SCString TextInput;
SCString TextInputName;

SCDateTimeArray BaseDateTimIn; // From the base graph
SCFloatArrayArray BaseDataIn; // From the base graph
SCDateTimeArray DateTimeOut;
SCInput145Array Input;

SCDateTime DateTimeOfLastFileRecord; // This is the DateTime of the last file record added to the ChartBaseDataIn
array
int SecondsPerBar;
SCString Symbol;
SCString DataFile; // The path + filename for the chart data file

void (SCDLLCALL* GetChartArray)(int ChartNumber, int DataItem, SCFloatArrayRef PriceArray);
void (SCDLLCALL* GetChartDateTimeArray)(int ChartNumber, SCDateTimeArrayRef DateTimeArray);
int (SCDLLCALL* GetStudyArray)(int StudyNumber, int StudySubgraphNumber, SCFloatArrayRef SubgraphArray);

void (SCDLLCALL* GetStudyArrayFromChart)(int ChartNumber, int StudyNumber, int StudySubgraphNumber,
SCFloatArrayRef SubgraphArray);

int ValueFormat;
SCString StudyDescription;

int StartTime1;
int EndTime1;
int StartTime2;
int EndTime2;
int UseSecondStartEndTimes;

int IndexOfFirstVisibleBar;

SCString& (SCDLLCALL* FormatString)(SCString& Out, const char* Format, ...);

const s_SCBasicSymbolData* SymbolData;

int CalculationPrecedence;

int AutoLoop;
int CurrentIndex;
int Index;

```

```

int StandardChartHeader;
int DisplayAsMainPriceGraph;

SCFloatArrayRef (SCDLLCALL* InternalExponentialMovAvg)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalLinearRegressionIndicator)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalAdaptiveMovAvg)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length, float FastSmoothConst, float SlowSmoothConst);

SCFloatArrayRef (SCDLLCALL* InternalSimpleMovAvg)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalWildersMovingAverage)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalWeightedMovingAverage)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalStdDeviation)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalCCISMA)(SCFloatArrayRef In, SCFloatArrayRef SMAOut, SCFloatArrayRef CCIOut, int Index, int Length, float Multiplier);

SCFloatArrayRef (SCDLLCALL* InternalHighest)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalLowest)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalATR)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef TROut, SCFloatArrayRef ATROut, int Index, int Length, unsigned int MovingAverageType);

SCFloatArrayRef (SCDLLCALL* InternalOnBalanceVolume)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int Index);

SCFloatArrayRef (SCDLLCALL* InternalOnBalanceVolumeShortTerm)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, SCFloatArrayRef OBVTemp, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalMovingAverage)(SCFloatArrayRef In, SCFloatArrayRef Out, unsigned int MovingAverageType, int Index, int Length);

void (SCDLLCALL* InternalStochastic)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef FastKOut, SCFloatArrayRef FastDOut, SCFloatArrayRef SlowDOut, int Index, int FastKLength, int FastDLength, int SlowDLength, unsigned int MovingAverageType);

SCString (SCDLLCALL* UserName)();

SCFloatArrayRef (SCDLLCALL* InternalStdError)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);
int (SCDLLCALL* InternalCalculateOHLCaverages)(int Index);

int (SCDLLCALL* p_ResizeArrays)(int NewSize);
int (SCDLLCALL* p_AddElements)(int NumElements);

int (SCDLLCALL* GetUserDrawnChartDrawing)(int ChartNumber, int DrawingType, s_UseTool& ChartDrawing, int DrawingIndex);

SCFloatArrayRef (SCDLLCALL* Internal_Ergodic)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int LongEMALength, int ShortEMALength, float Multiplier,
SCFloatArrayRef InternalArray1, SCFloatArrayRef InternalArray2, SCFloatArrayRef InternalArray3,
SCFloatArrayRef InternalArray4, SCFloatArrayRef InternalArray5, SCFloatArrayRef InternalArray6);

SCFloatArrayRef (SCDLLCALL* Internal_Keltner)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef In, SCFloatArrayRef KeltnerAverageOut, SCFloatArrayRef TopBandOut, SCFloatArrayRef BottomBandOut, int Index, int

```

KeltnerMALength, **unsigned int** KeltnerMAType, **int** TrueRangeMALength, **unsigned int** TrueRangeMAType, **float** TopBandMultiplier, **float** BottomBandMultiplier, SCFloatArrayRef InternalArray1, SCFloatArrayRef InternalArray2);

SCFloatArrayArray BaseData; // From the base graph

SCFloatArrayRef (SCDLLCALL* InternalTrueRange)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, **int** Index);

SCFloatArrayRef (SCDLLCALL* InternalWellesSum)(SCFloatArrayRef In ,SCFloatArrayRef Out , **int** Index, **int** Length);

void (SCDLLCALL* InternalDMI)(SCBaseDataRef ChartBaseDataIn, **int** Index, **int** Length, **int** DisableRounding, SCFloatArrayRef PosDMIOut, SCFloatArrayRef NegDMIOut, SCFloatArrayRef DiffDMIOut, SCFloatArrayRef InternalTrueRangeSummation, SCFloatArrayRef InternalPosDM, SCFloatArrayRef InternalNegDM);

SCFloatArrayRef (SCDLLCALL* InternalDMIDiff)(SCBaseDataRef ChartBaseDataIn, **int** Index, **int** Length, SCFloatArrayRef Out, SCFloatArrayRef InternalTrueRangeSummation, SCFloatArrayRef InternalPosDM, SCFloatArrayRef InternalNegDM);

SCFloatArrayRef (SCDLLCALL* InternalADX)(SCBaseDataRef ChartBaseDataIn, **int** Index, **int** DXLength, **int** DXMovAvgLength, SCFloatArrayRef Out, SCFloatArrayRef InternalTrueRangeSummation, SCFloatArrayRef InternalPosDM, SCFloatArrayRef InternalNegDM, SCFloatArrayRef InternalDX);

SCFloatArrayRef (SCDLLCALL* InternalADXR)(SCBaseDataRef ChartBaseDataIn, **int** Index, **int** DXLength, **int** DXMovAvgLength, **int** ADXRInterval, SCFloatArrayRef Out, SCFloatArrayRef InternalTrueRangeSummation, SCFloatArrayRef InternalPosDM, SCFloatArrayRef InternalNegDM, SCFloatArrayRef InternalDX, SCFloatArrayRef InternalADX);

int FileRecordIndexOfLastDataRecordInChart;

//The return value for this function must be implemented as a four byte integer, because a Boolean type may not be implemented the same way on all compilers and can cause corruptions.

int (SCDLLCALL* InternalFormattedEvaluate)(**float** Value1, **unsigned int** Value1Format, OperatorEnum Operator, **float** Value2, **unsigned int** Value2Format, **float** PrevValue1, **float** PrevValue2, **int*** CrossDirection);

int (SCDLLCALL* InternalAlertWithMessage)(**int** AlertNumber, **const char*** AlertMessage, **int** ShowAlertLog);

void (SCDLLCALL* InternalAddAlertLine)(**const char*** Message, **int** ShowAlertLog);

SCFloatArrayRef (SCDLLCALL *InternalRSI)(SCFloatArrayRef In, SCFloatArrayRef RsiOut, SCFloatArrayRef UpSumsTemp, SCFloatArrayRef DownSumsTemp, SCFloatArrayRef SmoothedUpSumsTemp, SCFloatArrayRef SmoothedDownSumsTemp, **int** Index, **unsigned int** AveragingType, **int** Length);

SCDateTime CurrentSystemDateTime;

int (SCDLLCALL* InternalPlaySoundPath)(**const char*** AlertPathAndFileName, **int** NumberTimesPlayAlert, **const char*** AlertMessage, **int** ShowLog);

SCSubgraph260Array Subgraph;

float TickSize;

SCFloatArrayRef (SCDLLCALL* InternalHullMovingAverage)(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef InternalArray1, SCFloatArrayRef InternalArray2, SCFloatArrayRef InternalArray3, **int** Index, **int** Length);

SCFloatArrayRef (SCDLLCALL* InternalTriangularMovingAverage)(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef InternalArray1, **int** Index, **int** Length);

SCFloatArrayRef (SCDLLCALL* InternalVolumeWeightedMovingAverage)(SCFloatArrayRef InData, SCFloatArrayRef InVolume, SCFloatArrayRef Out, **int** Index, **int** Length);

SCFloatArrayRef (SCDLLCALL* Internal_SmoothedMovingAverage)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index, **int** Length);

float (SCDLLCALL* InternalGetSummation)(SCFloatArrayRef In,**int** Index,**int** Length);

SCFloatArrayRef (SCDLLCALL* InternalMACD)(SCFloatArrayRef In, SCFloatArrayRef FastMAOut, SCFloatArrayRef SlowMAOut, SCFloatArrayRef MACDOut, SCFloatArrayRef MACDMAOut, SCFloatArrayRef MACDDiffOut, **int** Index, **int** FastMALength, **int** SlowMALength, **int** MACDMALength, **int** MovAvgType);

void (SCDLLCALL* GetMainGraphVisibleHighAndLow)(**float**& High, **float**& Low);

SCFloatArrayRef (SCDLLCALL* InternalTEMA)(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef InternalArray1,SCFloatArrayRef InternalArray2,SCFloatArrayRef InternalArray3, **int** Index, **int** Length);

SCFloatArrayRef (SCDLLCALL* InternalBollingerBands)(SCFloatArrayRef In, SCFloatArrayRef Avg, SCFloatArrayRef TopBand,SCFloatArrayRef BottomBand,SCFloatArrayRef StdDev, **int** Index, **int** Length,**float** Multiplier,**int** MovAvgType);

int (SCDLLCALL* Internal_GetOHLCForDate)(**const** SCDateTime& Date, **float**& Open, **float**& High, **float**& Low, **float**& Close);

int ReplayStatus;

int AdvancedFeatures;

float PreviousClose;

int BaseGraphValueFormat;

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

int UseGlobalChartColors;// Unused

unsigned int ScaleBorderColor;// Unused

int (SCDLLCALL* GetStudyArrayUsingID)(**unsigned int** StudyID, **unsigned int** StudySubgraphIndex, SCFloatArrayRef SubgraphArray);

int SelectedAlertSound;

SCFloatArrayRef (SCDLLCALL* InternalCumulativeSummation)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index);

SCFloatArrayRef (SCDLLCALL* InternalArmsEMV)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, **int** volinc, **int** Index);

SCFloatArrayRef (SCDLLCALL* InternalChaikinMoneyFlow)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, SCFloatArrayRef temp, **int** Index, **int** Length);

SCFloatArrayRef (SCDLLCALL* InternalSummation)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index, **int** Length);

SCFloatArrayRef (SCDLLCALL* InternalDispersion)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index, **int** Length);

float (SCDLLCALL* InternalGetDispersion)(SCFloatArrayRef In, **int** Index, **int** Length);

SCFloatArrayRef (SCDLLCALL* InternalEnvelopePercent)(SCFloatArrayRef In, SCFloatArrayRef Out1, SCFloatArrayRef Out2, **float** pct, **int** Index);

SCFloatArrayRef (SCDLLCALL* InternalVHF)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index, **int** Length);

SCFloatArrayRef (SCDLLCALL* InternalRWI)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out1,

```

SCFloatArrayRef Out2,
    SCFloatArrayRef TrueRangeArray, SCFloatArrayRef LookBackLowArray, SCFloatArrayRef LookBackHighArray, int
Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalUltimateOscillator)(SCBaseDataRef ChartBaseDataIn,
    SCFloatArrayRef Out,
    SCFloatArrayRef CalcE,
    SCFloatArrayRef CalcF,
    SCFloatArrayRef CalcG,
    SCFloatArrayRef CalcH,
    SCFloatArrayRef CalcI,
    SCFloatArrayRef CalcJ,
    SCFloatArrayRef CalcK,
    SCFloatArrayRef CalcL,
    SCFloatArrayRef CalcM,
    SCFloatArrayRef CalcN,
    SCFloatArrayRef CalcO,
    SCFloatArrayRef CalcP,
    SCFloatArrayRef CalcQ,
    int Index, int Length1, int Length2, int Length3);

SCFloatArrayRef (SCDLLCALL* InternalWilliamsAD)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int
Index);
SCFloatArrayRef (SCDLLCALL* InternalWilliamsR)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out, int
Index, int Length);
int (SCDLLCALL* InternalGetIslandReversal)(SCBaseDataRef ChartBaseDataIn, int Index);
SCFloatArrayRef (SCDLLCALL* InternalOscillator)(SCFloatArrayRef In1, SCFloatArrayRef In2, SCFloatArrayRef Out,
int Index);
float (SCDLLCALL* InternalGetTrueHigh)(SCBaseDataRef ChartBaseDataIn, int Index);
float (SCDLLCALL* InternalGetTrueLow)(SCBaseDataRef ChartBaseDataIn, int Index);
float (SCDLLCALL* InternalGetTrueRange)(SCBaseDataRef ChartBaseDataIn, int Index);
float (SCDLLCALL* InternalGetCorrelationCoefficient)(SCFloatArrayRef In1, SCFloatArrayRef In2, int Index, int
Length);

int (SCDLLCALL* InternalNumberOfBarsSinceHighestValue)(SCFloatArrayRef in, int Index, int Length);
int (SCDLLCALL* InternalNumberOfBarsSinceLowestValue)(SCFloatArrayRef in, int Index, int Length);

SCFloatArrayRef(SCDLLCALL* InternalPriceVolumeTrend)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out,
int Index);

SCString (SCDLLCALL* GetStudyNameUsingID)(unsigned int StudyID);
int (SCDLLCALL* GetStudyDataStartIndexUsingID)(unsigned int StudyID);

SCFloatArrayRef (SCDLLCALL* InternalLinearRegressionIndicatorAndStdErr)(SCFloatArrayRef In, SCFloatArrayRef
Out, SCFloatArrayRef StdErr, int Index, int Length);

float ValueIncrementPerBar;
SCString (SCDLLCALL* GetCountDownText)();

int GraphDrawType;

SCFloatArrayRef (SCDLLCALL* InternalMomentum)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length);

int (SCDLLCALL* InternalGetOHLCOfTimePeriod)(s_GetOHLCOfTimePeriod& Data);

int AlertOnlyOncePerBar;
int ResetAlertOnNewBar;

int AllowMultipleEntriesInSameDirection;
int SupportReversals;
int SendOrdersToTradeService;
int AllowOppositeEntryWithOpposingPositionOrOrders;
double(SCDLLCALL* InternalBuyEntry)(s_SCNewOrder& NewOrder, int Index);
double(SCDLLCALL* InternalBuyExit)(s_SCNewOrder& NewOrder, int Index);
double(SCDLLCALL* InternalSellEntry)(s_SCNewOrder& NewOrder, int Index);

```

```

double(SCDLLCALL* InternalSellExit)(s_SCNewOrder& NewOrder, int Index);
int (SCDLLCALL* CancelOrder)(uint64_t InternalOrderID);
int (SCDLLCALL* GetOrderByOrderID)(uint64_t InternalOrderID, s_SCTradeOrder& r_SCTradeOrder);
int (SCDLLCALL* GetOrderByIndex)(int OrderIndex, s_SCTradeOrder& r_SCTradeOrder);

SCFloatArrayRef (SCDLLCALL* InternalTRIX)(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef
InternalEma_1, SCFloatArrayRef InternalEma_2, SCFloatArrayRef InternalEma_3, int Index, int Length);

int DownloadingHistoricalData = 0;
SCString (SCDLLCALL* CustomAffiliateCode)();

int AllowOnlyOneTradePerBar;

void (SCDLLCALL* InternalStochastic2)(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow,
SCFloatArrayRef InputDataLast, SCFloatArrayRef FastKOut, SCFloatArrayRef FastDOut, SCFloatArrayRef SlowDOut, int
Index, int FastKLength, int FastDLength, int SlowDLength, unsigned int MovingAverageType);

SCFloatArrayRef (SCDLLCALL* InternalWilliamsR2)(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow,
SCFloatArrayRef InputDataLast, SCFloatArrayRef Out, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalParabolic)(s_Parabolic& ParabolicData);

int (SCDLLCALL* CancelAllOrders)();

int StartTimeOfDay;

float (SCDLLCALL* InternalArrayValueAtNthOccurrence)(SCFloatArrayRef TrueFalseIn, SCFloatArrayRef
ValueArrayIn, int Index, int NthOccurrence );

SCFloatArrayRef (SCDLLCALL* Internal_Demarker)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out,
SCFloatArrayRef DemMax, SCFloatArrayRef DemMin, SCFloatArrayRef SmaDemMax, SCFloatArrayRef SmaDemMin,
int Index, int Length);

int SupportAttachedOrdersForTrading;

int GlobalTradeSimulationIsOn;
int ChartTradeModeEnabled;

int CancelAllOrdersOnEntriesAndReversals;

SCFloatArrayRef (SCDLLCALL* InternalEnvelopeFixed)(SCFloatArrayRef In, SCFloatArrayRef Out1, SCFloatArrayRef
Out2, float FixedValue, int Index);

float CurrencyValuePerTick;

SCString (SCDLLCALL* GetStudyNameFromChart)(int ChartNumber, int StudyID);

double MaximumPositionAllowed;

uint16_t ChartBarSpacing;

int (SCDLLCALL* InternallsSwingHigh)(SCFloatArrayRef In, int Index, int Length);
int (SCDLLCALL* InternallsSwingLow)(SCFloatArrayRef In, int Index, int Length);

SCFloatArrayRef (SCDLLCALL* InternalZigZag2)(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow,
SCFloatArrayRef ZigZagValues, SCFloatArrayRef ZigZagPeakType, SCFloatArrayRef ZigZagPeakIndex, int Index, int
NumberOfBars, float ReversalAmount);

int (SCDLLCALL* ModifyOrder)( s_SCNewOrder& OrderModifications);

int AllowEntryWithWorkingOrders;

SCFloatArrayRef (SCDLLCALL* InternalAwesomeOscillator)(SCFloatArrayRef In, SCFloatArrayRef Out,
SCFloatArrayRef TempMA1, SCFloatArrayRef TempMA2, int Index, int Length1, int Length2);

```



```

int IsUserAllowedForSCDLLName;

int (SCDLLCALL* FlattenAndCancelAllOrders)();

int CancelAllWorkingOrdersOnExit;

int (SCDLLCALL* SessionStartTime)();
int (SCDLLCALL* IsDateTimeInSession)(const SCDateTime& DateTime);
int (SCDLLCALL* TradingDayStartsInPreviousDate)();
int (SCDLLCALL* GetTradingDayDate)(const SCDateTime& DateTime);
SCDateTime (SCDLLCALL* GetStartDateTimeForTradingDate)(const SCDateTime& TradingDate);
SCDateTime (SCDLLCALL* GetTradingDayStartDateTimeOfBar)(const SCDateTime& BarDateTime);
int (SCDLLCALL* SecondsSinceStartTime)(const SCDateTime& BarDateTime);

int HideStudy;

float ScaleValueOffset;
float AutoScalePaddingPercentage;

SCString (SCDLLCALL* GetChartName)(int ChartNumber);

int (SCDLLCALL* InternalGetTradePosition)(s_SCPositionData& PositionData);

void (SCDLLCALL* GetChartBaseData)(int ChartNumber, SCGraphData& BaseData);

void (SCDLLCALL* GetStudyArraysFromChart)(int ChartNumber, int StudyNumber, SCGraphData& GraphData);

void (SCDLLCALL* GetTimeAndSales)(c_SCTimeAndSalesArray& TSArry);

int (SCDLLCALL* InternalFormattedEvaluateUsingDoubles)(double Value1, unsigned int Value1Format,
    OperatorEnum Operator,
    double Value2, unsigned int Value2Format,
    double PrevValue1, double PrevValue2,
    int* CrossDirection);

SCString TradeWindowConfigFileName;

int StudyGraphInstanceID;

int (SCDLLCALL* GetTradingDayDateForChartNumber)(int ChartNumber, const SCDateTime& DateTime);
int (SCDLLCALL* GetFirstIndexForDate)(int ChartNumber, int Date);

int MaintainVolumeAtPriceData;

SCString (SCDLLCALL* FormatDateTime)(const SCDateTime& DateTime);

int DrawBaseGraphOverStudies;

unsigned int IntradayDataStorageTimeUnit;

int (SCDLLCALL* MakeHTTPRequest)(const SCString& URL);
SCString HTTPResponse;

void (SCDLLCALL* SetStudySubgraphDrawStyle)(int ChartNumber, int StudyID, int StudySubgraphNumber, int
DrawStyle);

void (SCDLLCALL* InternalGetStudyArrayFromChartUsingID)(const s_ChartStudySubgraphValues&
ChartStudySubgraphValues, SCFloatArrayRef SubgraphArray);

SCFloatArrayRef (SCDLLCALL* InternalSlope)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index);

void (SCDLLCALL* InternalSetAlert)(int AlertNumber, int Index, const SCString& Message);

int HideDLLAndFunctionNames;

```



```

SCFloatArray Open;
SCFloatArray High;
SCFloatArray Low;
SCFloatArray Close;
SCFloatArray Volume;
SCFloatArray OpenInterest;
SCFloatArray NumberOfTrades;
SCFloatArray OHLCAvg;
SCFloatArray HLCAvg;
SCFloatArray HLAvg;
SCFloatArray BidVolume;
SCFloatArray AskVolume;
SCFloatArray UpTickVolume;
SCFloatArray DownTickVolume;
SCFloatArray NumberOfBidTrades;
SCFloatArray NumberOfAskTrades;

int DrawStudyUnderneathMainPriceGraph = 0;

SCDateTime LatestDateTimeForLastBar;

int GlobalDisplayStudySubgraphsNameAndValue = 0;
int DisplayStudyName = 0;
int DisplayStudyInputValues = 0;

double TradeServiceAccountBalance = 0;

float ActiveToolYValue = 0;

int NewBarAtSessionStart = 0;

SCDateTime (SCDLLCALL* InternalConvertDateTimeToChartTimeZone)(const SCDateTime& DateTime, const char*
TimeZonePosixString);
float LastTradePrice;

SCDateTime ScrollToDateTime;

SCFloatArrayRef (SCDLLCALL* InternalDoubleStochastic)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef Out,
SCFloatArrayRef MovAvgIn, SCFloatArrayRef MovAvgOut, SCFloatArrayRef MovAvgIn2, SCFloatArrayRef Unused, int
Index, int Length, int EmaLength, int MovAvgType);

int SupportTradingScaleIn;
int SupportTradingScaleOut;

double TradeWindowOrderQuantity;

int IsAutoTradingEnabled;

int CurrentlySelectedDrawingTool;
int CurrentlySelectedDrawingToolState;

int ServerConnectionState; // ServerConnectionStateEnum

int MaintainAdditionalChartDataArrays;

SCString GraphShortName;

void (SCDLLCALL* GetStudyArraysFromChartUsingID)(int ChartNumber, int StudyID, SCGraphData& GraphData);
int (SCDLLCALL* Internal_GetStudyIDByIndex)(int ChartNumber, int StudyNumber);

SCDateTime (SCDLLCALL* DateTimeToStringToSCDateTime)(const SCString& DateTime);

int BaseGraphScaleRangeType;

double (SCDLLCALL* InternalGetStandardError)(SCFloatArrayRef In, int Index, int Length);

```

```

SCString DocumentationImageUrl;

int BaseGraphGraphDrawType;

int (SCDLLCALL* UserDrawnChartDrawingExists)(int ChartNumber, int LineNumber);

void (SCDLLCALL* SetCustomStudyControlBarButtonHoverText)(int ControlBarButtonNum, const char* HoverText);
void (SCDLLCALL* SetCustomStudyControlBarButtonEnable)(int ControlBarButtonNum, int Enable);

int (SCDLLCALL* AddACSCChartShortcutMenuItem)(int ChartNumber, const char* MenuText);
int (SCDLLCALL* RemoveACSCChartShortcutMenuItem)(int ChartNumber, int MenuID);

int MenuEventID = 0;
int PointerEventType = 0;

SCString TradeAndCurrentQuoteSymbol;

int GraphUsesChartColors = 0;

void (SCDLLCALL* GetBasicSymbolData)(const char* Symbol, s_SCBasicSymbolData& BasicSymbolData, int
Subscribe);

void (SCDLLCALL* SetCustomStudyControlBarButtonText)(int ControlBarButtonNum, const char* ButtonText);

SCDateTime (SCDLLCALL* AdjustDateTimeToGMT)(const SCDateTime& DateTime);

int (SCDLLCALL* DeleteUserDrawnACSDrawing)(int ChartNumber, int LineNumber);

int (SCDLLCALL* UploadChartImage)();

SCString (SCDLLCALL* ChartbookName)();

void (SCDLLCALL* RefreshTradeData)();

SCFloatArrayRef (SCDLLCALL* InternalCumulativeDeltaVolume)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef
Open, SCFloatArrayRef High, SCFloatArrayRef Low, SCFloatArrayRef Close, int Index, int ResetCumulativeCalculation);
SCFloatArrayRef (SCDLLCALL* InternalCumulativeDeltaTicks)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef
Open, SCFloatArrayRef High, SCFloatArrayRef Low, SCFloatArrayRef Close, int Index, int ResetCumulativeCalculation);

void (SCDLLCALL* GetStudyDataColorArrayFromChartUsingID)(int ChartNumber, int StudyID, int SubgraphIndex ,
SCColorArrayRef DataColorArray);

SCFloatArrayRef (SCDLLCALL* Internal_AroonIndicator)(SCFloatArrayRef FloatArrayInHigh, SCFloatArrayRef
FloatArrayInLow, SCFloatArrayRef OutUp, SCFloatArrayRef OutDown, SCFloatArrayRef OutOscillator, int Index, int
Length);

void* CreateRelayServer_NoLongerUsed;
int (SCDLLCALL* RelayNewSymbol)(const SCString& Symbol, int ValueFormat, float TickSize, const SCString&
ServiceCode);
int (SCDLLCALL* RelayTradeUpdate)(const SCString& Symbol, const SCDateTime& DateTime, float TradeValue,
unsigned int TradeVolume, int WriteRecord);
void (SCDLLCALL* RelayDataFeedAvailable)();
void (SCDLLCALL* RelayDataFeedUnavailable)();

int MaintainTradeStatisticsAndTradesData;

SCFloatArrayRef (SCDLLCALL* InternalMovingMedian)(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef
Temp, int Index, int Length);

int (SCDLLCALL* ChartIsDownloadingHistoricalData)(int ChartNumber);
int (SCDLLCALL* RelayServerConnected)();

void (SCDLLCALL* CreateProfitLossDisplayString)(double ProfitLoss, double Quantity, ProfitLossDisplayFormatEnum
ProfitLossFormat, SCString& Result);

```

```

int FlagFullRecalculate;

//Parameters reviewed for safety with different compilers
SCDateTime (SCDLLCALL* InternalGetStartOfPeriodForDateTime)(const SCDateTime& DateTime, unsigned int
TimePeriodType, int TimePeriodLength, int PeriodOffset, int NewPeriodAtBothSessionStarts);

c_VAPContainer *VolumeAtPriceForBars;

SCDateTime (SCDLLCALL* TimePeriodSpan)(unsigned int TimePeriodType, int TimePeriodLength);

int (SCDLLCALL* OpenChartOrGetChartReference)(s_ACSOpenChartParameters& Parameters);

int ProtectStudy;

float RealTimePriceMultiplier;
float HistoricalPriceMultiplier;

int (SCDLLCALL* GetTradeStatisticsForSymbol)(int Unused, int DailyStats, s_ACSTradeStatistics& TradeStats);

int ConnectToExternalServiceServer = 0;
int ReconnectToExternalServiceServer = 0;
int DisconnectFromExternalServiceServer = 0;

int (SCDLLCALL* IsDateTimeInDaySession) ( const SCDateTime &DateTime);

int AlertConditionFlags;

unsigned char (SCDLLCALL* GetTimeAndSalesForSymbol)(const SCString& Symbol, c_SCTimeAndSalesArray&
TSArry);

unsigned int StudyVersion = 0;

int AdvancedFeaturesLevel2 = 0;
void (SCDLLCALL* SetNumericInformationGraphDrawTypeConfig)(const
s_NumericInformationGraphDrawTypeConfig& NumericInformationGraphDrawTypeConfig);

double ChartTradingOrderPrice = 0;
int UseGUIAttachedOrderSetting = 0;
int BasedOnGraphValueFormat = 0;

SCFloatArrayRef (SCDLLCALL* InternalCumulativeDeltaTickVolume)(SCBaseDataRef ChartBaseDataIn,
SCFloatArrayRef Open, SCFloatArrayRef High, SCFloatArrayRef Low, SCFloatArrayRef Close, int Index, int
ResetCumulativeCalculation);

std::int64_t DLLNameUserServiceLevel = 0;

unsigned int (SCDLLCALL* CombinedForegroundColorBackgroundColorRef)(uint32_t ForegroundColor, uint32_t
BackgroundColor);

SCFloatArrayRef (SCDLLCALL* InternalAccumulationDistribution)(SCBaseDataRef ChartBaseDataIn,
SCFloatArrayRef Out, int Index);

int SaveChartImageToFile = 0;

int (SCDLLCALL* SetStudyVisibilityState)(int StudyID, int Visible);

c_VAPContainer* HistoricalHighPullbackVolumeAtPriceForBars = nullptr;

c_VAPContainer* HistoricalLowPullbackVolumeAtPriceForBars = nullptr;

int ReceivePointerEvents;

```

```

c_VAPContainer *PullbackVolumeAtPrice;

int (SCDLLCALL* GetNearestTargetOrder)(s_SCTradeOrder& OrderDetails);
int (SCDLLCALL* GetNearestStopOrder)(s_SCTradeOrder& OrderDetails);

int (SCDLLCALL* FlattenPosition)();

int (SCDLLCALL* BarIndexToXPixelCoordinate)(int Index);

int (SCDLLCALL* RegionValueToYPixelCoordinate)(float RegionValue, int ChartRegionNumber);

int (SCDLLCALL* SetACSCChartShortcutMenuItemDisplayed)(int ChartNumber, int MenuItemID, int DisplayItem);
int (SCDLLCALL* SetACSCChartShortcutMenuItemEnabled) (int ChartNumber, int MenuItemID, int Enabled);
int (SCDLLCALL* SetACSCChartShortcutMenuItemChecked)(int ChartNumber, int MenuItemID, int Checked);

fp_ACSGDIFunction p_GDIFunction;

unsigned int VolumeValueFormat;

int (SCDLLCALL* GetStudyVisibilityState)(int StudyID);

double (SCDLLCALL* InternalSubmitOCOOrder)(s_SCNewOrder& NewOrder, int Index);

float BaseGraphScaleConstRange;

int ACSVersion; // This is set to the header version the custom study was compiled with

int (SCDLLCALL* MakeHTTPBinaryRequest)(const SCString& URL);
SCConstCharArray HTTPBinaryResponse;

int (SCDLLCALL* GetTradeListEntry)(unsigned int TradeIndex, s_ACSTrade& TradeEntry);

void (SCDLLCALL* GetStudyExtraArrayFromChartUsingID)(int ChartNumber, int StudyID, int SubgraphIndex, int
ExtraArrayIndex, SCFloatArrayRef ExtraArrayRef);

int (SCDLLCALL* GetTradeListSize)();

double (SCDLLCALL* YPixelCoordinateToGraphValue)(int YPixelCoordinate);

int ContinuousFuturesContractLoading;

bool PlaceACSCChartShortcutMenuItemsAtTopOfMenu;

int (SCDLLCALL* AddACSCChartShortcutMenuSeparator)(int ChartNumber);

int UsesMarketDepthData;

float BaseGraphScaleRangeTop;
float BaseGraphScaleRangeBottom;
float BaseGraphScaleValueOffset;
float BaseGraphAutoScalePaddingPercentage;

int StudyRegionLeftCoordinate;
int StudyRegionRightCoordinate;

void (SCDLLCALL* SetNumericInformationDisplayOrderFromString)(const SCString& CommaSeparatedDisplayOrder);

SCString CustomChartTitleBarName;

SCFloatArrayRef (SCDLLCALL* InternalCCI)(SCFloatArrayRef In, SCFloatArrayRef SMAOut, SCFloatArrayRef
CCIOut, int Index, int Length, float Multiplier, unsigned int MovingAverageType);
int (SCDLLCALL* GetOrderFillArraySize)();
int (SCDLLCALL* GetOrderFillEntry)(unsigned int FillIndex, s_SCOrderFillData& FillData);

```

```

int CancelAllOrdersOnReversals;

SCString (SCDLLCALL * DateTimeToString)(const SCDatetime &DateTime,int Flags);

SCDateTime CurrentSystemDateTimeMS;

float FilterChartVolumeGreaterThanOrEqualTo;
float FilterChartVolumeLessThanOrEqualTo;

int ResetAllScales;

SCString (SCDLLCALL* FormatVolumeValue)(int64_t Volume, int VolumeValueFormat, int UseLargeNumberSuffix);

bool FilterChartVolumeTradeCompletely;

int ChartRegion1TopCoordinate;
int ChartRegion1BottomCoordinate;
int ChartRegion1LeftCoordinate;
int ChartRegion1RightCoordinate;

int BlockChartDrawingSelection;

int DaysToLoadInChart;

int EarliestUpdateSubgraphDataArrayIndex;

int& (SCDLLCALL* GetPersistentInt)(int Key);
void (SCDLLCALL* SetPersistentInt)(int Key, int Value);

float& (SCDLLCALL* GetPersistentFloat)(int Key);
void (SCDLLCALL* SetPersistentFloat)(int Key, float Value);

double& (SCDLLCALL* GetPersistentDouble)(int Key);
void (SCDLLCALL* SetPersistentDouble)(int Key, double Value);

std::int64_t& (SCDLLCALL* GetPersistentInt64)(int Key);
void (SCDLLCALL* SetPersistentInt64)(int Key, std::int64_t Value);

SCDateTime& (SCDLLCALL* GetPersistentSCDateTime)(int Key);

void*& (SCDLLCALL* GetPersistentPointer)(int Key);
void (SCDLLCALL* SetPersistentPointer)(int Key, void* Value);

int& (SCDLLCALL* GetPersistentIntFromChartStudy)(int ChartNumber, int StudyID, int Key);
void (SCDLLCALL* SetPersistentIntForChartStudy)(int ChartNumber, int StudyID, int Key, int Value);

float& (SCDLLCALL* GetPersistentFloatFromChartStudy)(int ChartNumber, int StudyID, int Key);
void (SCDLLCALL* SetPersistentFloatForChartStudy)(int ChartNumber, int StudyID, int Key, float Value);

double& (SCDLLCALL* GetPersistentDoubleFromChartStudy)(int ChartNumber, int StudyID, int Key);
void (SCDLLCALL* SetPersistentDoubleForChartStudy)(int ChartNumber, int StudyID, int Key, double Value);

std::int64_t& (SCDLLCALL* GetPersistentInt64FromChartStudy)(int ChartNumber, int StudyID, int Key);
void (SCDLLCALL* SetPersistentInt64ForChartStudy)(int ChartNumber, int StudyID, int Key, std::int64_t Value);

SCDateTime& (SCDLLCALL* GetPersistentSCDateTimeFromChartStudy)(int ChartNumber, int StudyID, int Key);

void*& (SCDLLCALL* GetPersistentPointerFromChartStudy)(int ChartNumber, int StudyID, int Key);
void (SCDLLCALL* SetPersistentPointerForChartStudy)(int ChartNumber, int StudyID, int Key, void* Value);

int (SCDLLCALL* GetLatestBarCountdownAsInteger)();

int ContinuousFuturesContractOption;

```

```

int (SCDLLCALL* IsChartDataLoadingInChartbook)();
SCString (SCDLLCALL* DataTradeServiceName)();

void (SCDLLCALL* CloseChart)(int ChartNumber);
void (SCDLLCALL* CloseChartbook)(const SCString& ChartbookFileName);

unsigned int DataFeedActivityCounter;

SCString (SCDLLCALL* GetChartTextFontFaceName)();

SCString SelectedTradeAccount;
float RoundTurnCommission;

SCString& (SCDLLCALL* GetPersistentSCString)(int Key);
SCString& (SCDLLCALL* GetPersistentSCStringFromChartStudy)(int ChartNumber, int StudyID, int Key);

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

int IsFullRecalculation;

int IntradayChartRecordingState;

int IsChartTradeModeOn;

SCdatetime DailyStatsResetTime;

double TradeServiceAvailableFundsForNewPositions;

void (SCDLLCALL* GetAttachedOrderIDsForParentOrder)(int ParentInternalOrderID, int& TargetInternalOrderID, int&
StopInternalOrderID);

int FlagToReloadChartData;

int (SCDLLCALL* GetTradePositionForSymbolAndAccount)(s_SCPositionData& r_PositionData, const SCString&
Symbol, const SCString& TradeAccount);

int ReceiveNotificationsForChangesToOrdersPositionsForAnySymbol;

int (SCDLLCALL* InternalSubmitOrder)(s_SCNewOrder& r_NewOrder, int BarIndex, BuySellEnum BuySell);

int LoadChartDataByDateRange;
int ChartDataStartDate;
int ChartDataEndDate;

int (SCDLLCALL* GetStudyDataStartIndexFromChartUsingID)(int ChartNumber, unsigned int StudyID);

void (SCDLLCALL* InternalVortex)(SCBaseDataRef ChartBaseDataIn, SCFloatArrayRef TrueRangeOut,
SCFloatArrayRef VortexMovementUpOut, SCFloatArrayRef VortexMovementDownOut, SCFloatArrayRef VMPlusOut,
SCFloatArrayRef VMMinusOut, int Index, int VortexLength);

int (SCDLLCALL* GetFirstNearestIndexForTradingDayDate)(int ChartNumber, int TradingDayDate);

int (SCDLLCALL* Internal_GetOpenHighLowCloseVolumeForDate)(const SCDatetime& Date, float& r_Open, float&
r_High, float& r_Low, float& r_Close, float& r_Volume);

const char* (SCDLLCALL* GetStudySubgraphName)(int StudyID, int SubgraphIndex);

SCdatetime (SCDLLCALL* TimeStringToSCdatetime)(const SCString& TimeString);

int (SCDLLCALL* ChangeACSCChartShortcutMenuItemText)(int ChartNumber, int MenuItemIdentifier, const char*
NewMenuText);

int ChartWindowsActive;

```

```

SCDateTime (SCDLLCALL* DateTimeDDMMYYYYToSCDateTime)(const SCString& DateTime);

float PointAndFigureXOGraphDrawTypeBoxSize;

int (SCDLLCALL* SaveChartbook)();

c_VAPContainer * VolumeAtPriceForStudy = nullptr;

SCDateTime CurrentDateTimeForReplay;


void (SCDLLCALL* InternalHeikinAshi)(SCBaseDataRef ChartBaseDataIn, int Index, int Length, SCFloatArrayRef
OpenOut, SCFloatArrayRef HighOut, SCFloatArrayRef LowOut, SCFloatArrayRef LastOut, int
SetCloseToCurrentPriceAtLastBar);

SCDateTime (SCDLLCALL* CalculateTimeSpanAcrossChartBars)(int FirstIndex, int LastIndex);

int (SCDLLCALL* GetHighestChartNumberUsedInChartBook)();

int (SCDLLCALL * GetCalculationStartIndexForStudy)();

SCDateTime (SCDLLCALL* GetEndingDateTimeForBarIndex)(int BarIndex);

HWND (SCDLLCALL* GetChartWindowHandle)(int ChartNumber);

c_ACSILDepthBars* (SCDLLCALL* GetMarketDepthBars)();
c_ACSILDepthBars* (SCDLLCALL* GetMarketDepthBarsFromChart)(int ChartNumber);
void (SCDLLCALL* SaveChartImageToFileExtended)(int ChartNumber, SCString &OutputPathAndFileName, int Width,
int Height, int IncludeOverlays);

int DrawACSDrawingsAboveOtherDrawings;

SCString(SCDLLCALL* DataFilesFolder)();

double (SCDLLCALL* GetLastPriceForTrading)();


SCString (SCDLLCALL* GetChartSymbol)(int ChartNumber);

void (SCDLLCALL* Internal_CalculateRegressionStatistics)(SCFloatArrayRef In, double &Slope, double &Y_Intercept,
int Index, int Length);

SCDateTime (SCDLLCALL* GetEndingDateTimeForBarIndexFromChart)(int ChartNumber, int BarIndex);

int (SCDLLCALL* AddLineUntilFutureIntersection)
( int StartBarIndex
, int LineIDForBar
, float LineValue
, uint32_t LineColor
, uint16_t LineWidth
, uint16_t LineStyle
, int DrawValueLabel
, int DrawNameLabel
, const SCString& NameLabel
);

int (SCDLLCALL* GetBidMarketDepthStackPullValueAtPrice)(float Price);
int (SCDLLCALL* GetAskMarketDepthStackPullValueAtPrice)(float Price);

int (SCDLLCALL* SetChartWindowState)(int ChartNumber, int WindowState);

SCDateTime (SCDLLCALL* InternalConvertDateTimeFromChartTimeZone)(const SCDateTime& DateTime, const
char* TimeZonePosixString);

```



```

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

uint64_t (SCDLLCALL* GetParentOrderIDFromAttachedOrderID)(uint64_t AttachedOrderInternalOrderID);

int (SCDLLCALL* StartDownloadHistoricalData)(const SCDatetime& StartingDateTime);

void (SCDLLCALL* ClearAllPersistentData)();

int MaintainHistoricalMarketDepthData;

void (SCDLLCALL* Internal_CalculateLogLogRegressionStatistics)(SCFloatArrayRef In, double &Slope, double
&Y_Intercept, int Index, int Length);

void (SCDLLCALL* Internal_InverseFisherTransform)(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef
CalcArray1, SCFloatArrayRef CalcArray2, int Index, int HighestLowestLength, int MovingAverageLength, int
MovAvgType);

void (SCDLLCALL* Internal_InverseFisherTransformRSI)(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef
RSIArray1, SCFloatArrayRef RSIArray2, SCFloatArrayRef RSIArray3, SCFloatArrayRef RSIArray4, SCFloatArrayRef
RSIArray5, SCFloatArrayRef CalcArray1, SCFloatArrayRef CalcArray2, int Index, int RSILength, int
InternalRSIMovAvgType, int RSIMovingAverageLength, int MovingAverageOfRSIType);

void (SCDLLCALL* Internal_MovingAverageCumulative)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index);

SCFloatArrayRef(SCDLLCALL* Internal_HurstExponent)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int
LengthIndex);

SCString (SCDLLCALL* FormatDateTimeMS)(const SCDatetime& DateTimeMS);
SCString (SCDLLCALL* DateToString)(const SCDatetime& DateTime);
SCString (SCDLLCALL* TimeToString)(const SCDatetime& DateTime);
SCString (SCDLLCALL* TimeMSToString)(const SCDatetime& DateTimeMS);

SCString (SCDLLCALL* SCDataFeedSymbol)();

unsigned int (SCDLLCALL* GetRecentBidVolumeAtPrice)(float Price);
unsigned int (SCDLLCALL* GetRecentAskVolumeAtPrice)(float Price);

SCFloatArrayRef(SCDLLCALL* InternalResettableZigZag)(SCFloatArrayRef InputDataHigh, SCFloatArrayRef
InputDataLow, SCFloatArrayRef ZigZagValues, SCFloatArrayRef ZigZagPeakType, SCFloatArrayRef ZigZagPeakIndex,
int StartIndex, int Index, float ReversalPercent, float ReversalAmount, SCStudyInterfaceRef sc);
SCFloatArrayRef(SCDLLCALL* InternalResettableZigZag2)(SCFloatArrayRef InputDataHigh, SCFloatArrayRef
InputDataLow, SCFloatArrayRef ZigZagValues, SCFloatArrayRef ZigZagPeakType, SCFloatArrayRef ZigZagPeakIndex,
int StartIndex, int Index, int NumberOfBars, float ReversalAmount, SCStudyInterfaceRef sc);

int BaseGraphConstantRangeScaleMode;

int (SCDLLCALL* GetStudyPeakValleyLine)(int ChartNumber, int StudyID, float& PeakValleyLinePrice, int&
PeakValleyType, int& StartIndex, int& PeakValleyExtensionChartColumnEndIndex, int ProfileIndex, int PeakValleyIndex);

int (SCDLLCALL* GetStudyLineUntilFutureIntersection)(int ChartNumber, int StudyID, int BarIndex, int LineIndex, int&
LineIDForBar, float& LineValue, int& ExtensionLineChartColumnEndIndex);

SCDateTimeArray BaseDataEndDateTime; // From the base graph

int (SCDLLCALL* Internal_GetOpenHighLowCloseVolumeForDate2)(const SCDatetime& Date, float& r_Open, float&
r_High, float& r_Low, float& r_Close, float& r_Volume, int IncludeFridayEveningSessionWithSundayEveningSession);

int (SCDLLCALL* GetUserDrawnDrawingByLineNumber)(int ChartNumber, int LineNumber, s_UseTool&
ChartDrawing);

int (SCDLLCALL* GetACSDrawingByLineNumber)(int ChartNumber, int LineNumber, s_UseTool& ChartDrawing);

int (SCDLLCALL* GetACSDrawingByIndex)(int ChartNumber, int Index, s_UseTool& ChartDrawing, int
ExcludeOtherStudyInstances);

```



```

int(SCDLLCALL* GetACSDrawingsCount)(int ChartNumber, int ExcludeOtherStudyInstances);

int (SCDLLCALL* GetNearestMatchForSCDateTimeExtended)(int ChartNumber, const SCDateTime& DateTime);


int ReceiveKeyboardKeyEvents;
int ReceiveCharacterEvents;
int KeyboardKeyEventCode;
int CharacterEventCode;


//From the perspective of the study developer, this is best to be set to true when they do not want the chart redrawn
after the study returns, rather than having it true by default and setting it to 0
short DoNotRedrawChartAfterStudyReturns;


uint32_t RightValuesScaleLeftCoordinate;
uint32_t RightValuesScaleRightCoordinate;


int LastFullCalculationTimeInMicroseconds;


uint16_t MaintainReferenceToOtherChartsForPersistentVariables;


int HTTPRequestID;


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


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


double (SCDLLCALL* StringToDouble)(const char* NumberString);


SCString (SCDLLCALL* ServiceCodeForSelectedDataTradingService)();


void (SCDLLCALL* GetProfitManagementStringForTradeAccount)(SCString& r_TextString);


uint16_t SupportKeyboardModifierStates;
uint16_t IsKeyPressed_Control;
uint16_t IsKeyPressed_Shift;
uint16_t IsKeyPressed_Alt;


SCFloatArrayRef (SCDLLCALL* Internal_T3MovingAverage)
( SCFloatArrayRef InputArray
, SCFloatArrayRef OutputArray
, SCFloatArrayRef CalcArray0
, SCFloatArrayRef CalcArray1
, SCFloatArrayRef CalcArray2
, SCFloatArrayRef CalcArray3
, SCFloatArrayRef CalcArray4
, SCFloatArrayRef CalcArray5
, float Multiplier
, int Index
, int Length
);


uint16_t TradingIsLocked;//read-only


uint32_t (SCDLLCALL* GetNumberOfDataFeedSymbolsTracked)();


int (SCDLLCALL* GetDOMColumnLeftCoordinate)(n_ACSIL::DOMColumnTypeEnum DOMColumn);
int (SCDLLCALL* GetDOMColumnRightCoordinate)(n_ACSIL::DOMColumnTypeEnum DOMColumn);


uint32_t (SCDLLCALL* GetCurrentTradedBidVolumeAtPrice)(float Price);
uint32_t (SCDLLCALL* GetCurrentTradedAskVolumeAtPrice)(float Price);

```

```

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

    fp_ACSCustomChartBarFunction fp_ACSCustomChartBarFunction;

    int UsesCustomChartBarFunction;

    int (SCDLLCALL* ReadIntradayFileRecordAtIndex)(int64_t Index, s_IntradayRecord& r_Record,
IntradayFileLockActionEnum IntradayFileLockAction);

    int (SCDLLCALL* ReadIntradayFileRecordForBarIndexAndSubIndex)(int BarIndex, int SubRecordIndex,
s_IntradayRecord& r_Record, IntradayFileLockActionEnum IntradayFileLockAction);

    SCFloatArrayRef(SCDLLCALL* Internal_CalculateAngle)(SCFloatArrayRef InputArray, SCFloatArrayRef OutputArray,
int Index, int Length, float ValuePerPoint);

    int NumberOfForwardColumns;

    int AllocateAndNameRenkoChartBarArrays;

    void (SCDLLCALL* OpenChartbook)(const SCString& ChartbookFileName);

    int (SCDLLCALL* IsDateTimeInEveningSession) (const SCDateTime &DateTime);

    uint16_t TransparencyLevel;

    int (SCDLLCALL* IsChartNumberExist) (int ChartNumber, const SCString& ChartbookFileName);

    void* (SCDLLCALL* GetStudyStorageBlockFromChart) (int ChartNumber, int StudyID);
    int (SCDLLCALL* WriteBarAndStudyDataToFile)(int StartingIndex, SCString &OutputPathAndFileName, int
IncludeHiddenStudies, int IncludeHiddenSubgraphs);

    int32_t ConstantRangeScaleModeTicksFromCenterOrEdge;

    void (SCDLLCALL* GetBarPeriodParameters)(n_ACSIL::s_BarPeriod& r_BarPeriod);

    uint16_t IsChartbookBeingSaved;

    double (SCDLLCALL* Internal_GetSymbolDataValue)(SymbolDataValuesEnum Value, const char* Symbol, int
SubscribeToMarketData, int SubscribeToMarketDepth);

    void (SCDLLCALL* SetBarPeriodParameters)(const n_ACSIL::s_BarPeriod& BarPeriod);

    uint16_t IncludeInStudySummary;

    uint16_t PointerVertWindowCoord;
    uint16_t PointerHorzWindowCoord;

    void (SCDLLCALL* GetPointOfControlPriceVolumeForBar)(int BarIndex, s_VolumeAtPriceV2& VolumeAtPrice);

    int (SCDLLCALL* EvaluateAlertConditionFormulaAsBoolean)(int BarIndex, int ParseAndSetFormula);

    void (SCDLLCALL* Internal_AddMessageToTradeServiceLog)(const char* Message, int ShowLog, int
AddChartStudyDetails);

    void (SCDLLCALL* SetCustomStudyControlBarButtonColor)(int ControlBarButtonNum, const uint32_t Color);

    int (SCDLLCALL* GetTradePositionByIndex)(s_SCPositionData& r_PositionData, int Index);

    int CancelAllOrdersOnEntries;

    int (SCDLLCALL* SetHorizontalGridState)(int GridIndex, int State);
    int (SCDLLCALL* SetVerticalGridState)(int State);

    int (SCDLLCALL* GetOrderForSymbolAndAccountByIndex)(const char* Symbol, const char* TradeAccount, int

```

OrderIndex, s_SCTradeOrder& r_SCTradeOrder);

```
int (SCDLLCALL* GetBidMarketDepthNumberOfLevels)();
int (SCDLLCALL* GetAskMarketDepthNumberOfLevels)();
int (SCDLLCALL* GetMaximumMarketDepthLevels)();
int (SCDLLCALL* GetBidMarketDepthEntryAtLevel)(s_MarketDepthEntry& r_DepthEntry, int LevelIndex);
int (SCDLLCALL* GetAskMarketDepthEntryAtLevel)(s_MarketDepthEntry& r_DepthEntry, int LevelIndex);
int IsAutoTradingOptionEnabledForChart;
```

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

```
uint16_t IncludeInSpreadsheet;
```

```
int (SCDLLCALL* GetVolumeAtPriceDataForStudyProfile)
( const int StudyID
, const int ProfileIndex
, const int PriceIndex//zero-based
, s_VolumeAtPriceV2& r_VolumeAtPrice
);
int (SCDLLCALL* GetNumPriceLevelsForStudyProfile)(int StudyID, int ProfileIndex);
```

```
void (SCDLLCALL* GetTradingDayStartDateTimeOfBarForChart)(const SCDatetime& BarDateTime, SCDatetime&
r_TradingDayStartDateTime, int ChartNumber);
```

```
int (SCDLLCALL* GetChartStudyInputInt)(int ChartNumber, int StudyID, int InputIndex, int& r_IntegerValue);
int (SCDLLCALL* SetChartStudyInputInt)(int ChartNumber, int StudyID, int InputIndex, int IntegerValue);
int (SCDLLCALL* GetChartStudyInputFloat)(int ChartNumber, int StudyID, int InputIndex, double& r_FloatValue);
int (SCDLLCALL* SetChartStudyInputFloat)(int ChartNumber, int StudyID, int InputIndex, double FloatValue);
```

```
int (SCDLLCALL* RecalculateChart)(int ChartNumber);
```

```
int (SCDLLCALL* Internal_GetStudyIDByName)(int ChartNumber, const char* Name, const int UseShortNameIfSet);
```

```
n_ACSIL::DTCSecurityTypeEnum (SCDLLCALL* SecurityType)();
```

```
void (SCDLLCALL* GetGraphVisibleHighAndLow)(double& High, double& Low);
```

```
uint32_t VolumeAtPriceMultiplier;
```

```
int32_t (SCDLLCALL* OpenFile)(const SCString& PathAndFileName, const int Mode, int& FileHandle);
int32_t (SCDLLCALL* ReadFile)(const int FileHandle, char* Buffer, const int BytesToRead, unsigned int*
p_BytesRead);
int32_t (SCDLLCALL* WriteFile)(const int FileHandle, const char* Buffer, const int BytesToWrite, unsigned int*
p_BytesWritten);
int32_t (SCDLLCALL* CloseFile)(const int FileHandle);
int (SCDLLCALL* GetLastFileErrorCode)(const int FileHandle);
SCString (SCDLLCALL* GetLastFileErrorMessage)(const int FileHandle);
```

```
int AlertConditionEnabled;
int (SCDLLCALL* GetSessionTimesFromChart)(const int ChartNumber, n_ACSIL::s_ChartSessionTimes&
r_ChartSessionTimes);
```

```
void* (SCDLLCALL* GetSpreadsheetSheetHandleByName)(const char* SheetCollectionName, const char*
SheetName, const int CreateSheetIfNotExist);
int (SCDLLCALL* GetSheetCellAsDouble)(void* SheetHandle, const int Column, const int Row, double& r_CellValue);
int (SCDLLCALL* SetSheetCellAsDouble)(void* SheetHandle, const int Column, const int Row, const double
CellValue);
int (SCDLLCALL* GetSheetCellAsString)(void* SheetHandle, const int Column, const int Row, SCString& r_CellString);
int (SCDLLCALL* SetSheetCellAsString)(void* SheetHandle, const int Column, const int Row, const SCString&
CellString);
```

```
SCDateTime ContractRolloverDate;
```

```
double (SCDLLCALL* GetTotalNetProfitLossForAllSymbols)(int DailyValues);
```

```

void (SCDLLCALL* SetAttachedOrders)(const s_SCNewOrder& AttachedOrdersConfiguration);

int (SCDLLCALL* GetLineNumberOfSelectedUserDrawnDrawing)();

int (SCDLLCALL* MakeHTTPPOSTRequest)(const SCString& URL, const SCString& POSTData, const
n_ACSIL::s_HTTPHeader* Headers, int NumberOfHeaders);

void (SCDLLCALL* ClearAlertSoundQueue)();

void(SCDLLCALL* SetPersistentSCDateTime)(int Key, const SCDateTime& Value);
void(SCDLLCALL* SetPersistentSCDateTimeForChartStudy)(int ChartNumber, int StudyID, int Key, const
SCDateTime& Value);
int ChartUpdateIntervalInMilliseconds;

int (SCDLLCALL* GetTradeWindowOrderType)();

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

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

int (SCDLLCALL* WriteBarAndStudyDataToFileEx)(const n_ACSIL::s_WriteBarAndStudyDataToFile&
WriteBarAndStudyDataToFileParams);

int (SCDLLCALL* RecalculateChartImmediate)(int ChartNumber);

int (SCDLLCALL* GetCustomStudyControlBarButtonEnableState)(int ControlBarButtonNum);

uint32_t (SCDLLCALL* ClearCurrentTradedBidAskVolume)();

int (SCDLLCALL* AddLineUntilFutureIntersectionEx)(const n_ACSIL::s_LineUntilFutureIntersection&
LineUntilFutureIntersection);

int (SCDLLCALL* GetBidMarketDepthNumberOfLevelsForSymbol)(const SCString& Symbol);
int (SCDLLCALL* GetAskMarketDepthNumberOfLevelsForSymbol)(const SCString& Symbol);
int (SCDLLCALL* GetBidMarketDepthEntryAtLevelForSymbol)(const SCString& Symbol, s_MarketDepthEntry&
r_DepthEntry, int LevelIndex);
int (SCDLLCALL* GetAskMarketDepthEntryAtLevelForSymbol)(const SCString& Symbol, s_MarketDepthEntry&
r_DepthEntry, int LevelIndex);

int (SCDLLCALL* GetChartStudyInputString)(int ChartNumber, int StudyID, int InputIndex, SCString& r_StringValue);
int (SCDLLCALL* SetChartStudyInputString)(int ChartNumber, int StudyID, int InputIndex, const SCString&
StringValue);

int (SCDLLCALL* GetStudySubgraphDrawStyle)(int ChartNumber, int StudyID, int StudySubgraphNumber, int&
r_DrawStyle);

void (SCDLLCALL* SetTradeWindowTextTag)(const SCString& TextTag);

int (SCDLLCALL* GetStudyLineUntilFutureIntersectionByIndex)(int ChartNumber, int StudyID, int Index, int&
r_LineIDForBar, int& r_StartIndex, float& r_LineValue, int& r_ExtensionLineChartColumnEndIndex);

int (SCDLLCALL* GetTradingKeyboardShortcutsEnableState)();
void (SCDLLCALL* SetTradingKeyboardShortcutsEnableState)(int State);

int (SCDLLCALL* GetFlatToFlatTradeListSize)();
int (SCDLLCALL* GetFlatToFlatTradeListEntry)(unsigned int TradeIndex, s_ACSTrade& TradeEntry);

int (SCDLLCALL* StartChartReplay)(int ChartNumber, float ReplaySpeed, const SCDateTime& StartDateTime);
int (SCDLLCALL* StopChartReplay)(int ChartNumber);
int (SCDLLCALL* PauseChartReplay)(int ChartNumber);
int (SCDLLCALL* ResumeChartReplay)(int ChartNumber);
int (SCDLLCALL* ChangeChartReplaySpeed)(int ChartNumber, float ReplaySpeed);

```

```

int (SCDLLCALL* GetStudyProfileInformation)(const int StudyID, const int ProfileIndex,
n_ACSIL::s_StudyProfileInformation& StudyProfileInformation);

int32_t (SCDLLCALL* GetStudySubgraphColors)(int32_t ChartNumber, int32_t StudyID, int32_t
StudySubgraphNumber, uint32_t& r_PrimaryColor, uint32_t& r_SecondaryColor, uint32_t& r_SecondaryColorUsed);

int32_t (SCDLLCALL* SetStudySubgraphColors)(int32_t ChartNumber, int32_t StudyID, int32_t
StudySubgraphNumber, uint32_t PrimaryColor, uint32_t SecondaryColor, uint32_t SecondaryColorUsed);

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

int (SCDLLCALL* IsChartZoomInStateActive)();

int (SCDLLCALL* GetTradeStatisticsForSymbolV2)(n_ACSIL::TradeStatisticsTypeEnum StatsType,
n_ACSIL::s_TradeStatistics& TradeStatistics);

int (SCDLLCALL* GetNumStudyProfiles)(int StudyID);

void (SCDLLCALL* SetPersistentSCString)(int Key, const SCString& Value);

void (SCDLLCALL* SetPersistentSCStringForChartStudy)(int ChartNumber, int StudyID, int Key, const SCString&
Value);

int32_t (SCDLLCALL* SetTradingLockState)(int32_t NewState);

int32_t (SCDLLCALL* IsChartDataLoadingCompleteForAllCharts)();

int32_t (SCDLLCALL* GetStudySubgraphNameFromChart)(int ChartNumber, int StudyID, int SubgraphIndex,
SCString& r_SubgraphName);

int32_t (SCDLLCALL* GetReplayStatusFromChart)(int ChartNumber);

int (SCDLLCALL* GetStudySummaryCellAsDouble)(const int Column, const int Row, double& r_CellValue);
int (SCDLLCALL* GetStudySummaryCellAsString)(const int Column, const int Row, SCString& r_CellString);
int (SCDLLCALL* StartChartReplayNew)(n_ACSIL::s_ChartReplayParameters& ChartReplayParameters);

int (SCDLLCALL* GetSymbolDescription)(SCString& r_Description);

int (SCDLLCALL* GetTradeWindowTextTag)(SCString& r_TextTag);

int (SCDLLCALL* StartScanOfSymbolList)(const int ChartNumber);
int (SCDLLCALL* StopScanOfSymbolList)(const int ChartNumber);

double (SCDLLCALL* GetBidMarketDepthStackPullSum)();
double (SCDLLCALL* GetAskMarketDepthStackPullSum)();

float (SCDLLCALL* GetChartReplaySpeed)(int ChartNumber);

int32_t (SCDLLCALL* 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);

int& (SCDLLCALL* GetPersistentIntFast)(int32_t Index);
float& (SCDLLCALL* GetPersistentFloatFast)(int32_t Index);
double& (SCDLLCALL* GetPersistentDoubleFast)(int32_t Index);
SCDateTime& (SCDLLCALL* GetPersistentSCDateTimeFast)(int32_t Index);

int(SCDLLCALL* SendEmailMessage)(const SCString& ToEmailAddress, const SCString& Subject, const SCString&
MessageBody);

int(SCDLLCALL* GetReplayHasFinishedStatus)();

int (SCDLLCALL* AddStudyToChart)(const n_ACSIL::s_AddStudy& AddStudy);

```

```

int32_t Unused_1 = 0;

int (SCDLLCALL* GetNumLinesUntilFutureIntersection)(int ChartNumber, int StudyID);

int (SCDLLCALL* GetPointOfControlAndValueAreaPricesForBar)(int BarIndex, double& r_PointOfControl, double&
r_ValueAreaHigh, double& r_ValueAreaLow, float ValueAreaPercentage);

void (SCDLLCALL* GetBarPeriodParametersForChart)(int ChartNumber, n_ACSIL::s_BarPeriod& r_BarPeriod);

int (SCDLLCALL* SetChartTimeZone)(const int ChartNumber, const TimeZonesEnum TimeZone);
double (SCDLLCALL* GetTradeServiceAccountBalanceForTradeAccount)(const SCString& TradeAccount);

SCDateTime (SCDLLCALL* GetStartDateTimeOfDaySessionForTradingDayDate)(const SCDateTime& TradingDate);
int (SCDLLCALL* SaveMainWindowImageToFile)(SCString& OutputPathAndFileName);

// 2210
//Derived from c_VAPContainerBase<s_VolumeLevelAtPrice>
c_VolumeLevelAtPriceContainer* p_VolumeLevelAtPriceForBars = NULL;

// 2212
int (SCDLLCALL* IsLastBarClosedBasedOnElapsedTime)(const int ChartNumber);

// 2214
int (SCDLLCALL* GetBidMarketLimitOrdersForPrice)(const int PriceInTicks, const int ArraySize,
n_ACSIL::s_MarketOrderData* p_MarketOrderDataArray);
int (SCDLLCALL* GetAskMarketLimitOrdersForPrice)(const int PriceInTicks, const int ArraySize,
n_ACSIL::s_MarketOrderData* p_MarketOrderDataArray);

// 2244
int (SCDLLCALL* GetStudyProfileInformationForChart)(const int ChartNumber, const int StudyID, const int
ProfileIndex, n_ACSIL::s_StudyProfileInformation& r_StudyProfileInformation) = nullptr;

SCFloatArrayRef (SCDLLCALL* InternalExampleFunction)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int
Length);

// 2255
int (SCDLLCALL* GetEarliestSourceIndexForDestinationDateTime)(int DestinationDateTimeIndex, const
SCDateTime& BarTimeDuration, int BarTimeMatchingMethod, int DestinationChartNumber, int SourceChartNumber, int
SourceMaximumDisplacement, int DestinationBarsForward);

int (SCDLLCALL* GetLatestSourceIndexForDestinationDateTime)(int DestinationDateTimeIndex, const SCDateTime&
BarTimeDuration, int BarTimeMatchingMethod, int DestinationChartNumber, int SourceChartNumber, int
SourceMaximumDisplacement, int DestinationBarsForward);

// 2260
int (SCDLLCALL* SetInvertUnderlyingChartData)(int InvertData);

// 2261
SCFloatArrayRef (SCDLLCALL* InternalArnaudLegouxMovingAverage)(SCFloatArrayRef In, SCFloatArrayRef Out, int
Index, int Length, float Sigma, float Offset);

int (SCDLLCALL* FlattenAndCancelAllOrdersForSymbolAndNonSimTradeAccount)(const SCString& Symbol, const
SCString& TradeAccount);

// 2273
int PriorSelectedCustomStudyControlBarButtonNumber = 0;

// 2293
SCFloatArrayRef(SCDLLCALL* InternalExponentialRegressionIndicator)(SCFloatArrayRef In, SCFloatArrayRef Out, int
Index, int Length);

SCFloatArrayRef(SCDLLCALL* InternalInstantaneousTrendline)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index,
int Length);

```


SCFloatArrayRef(SCDLLCALL* InternalCyberCycle)(SCFloatArrayRef In, SCFloatArrayRef Smoothed, SCFloatArrayRef Out, **int** Index, **int** Length);

SCFloatArrayRef(SCDLLCALL* InternalFourBarSymmetricalFIRFilter)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index);

SCFloatArrayRef(SCDLLCALL* InternalSuperSmoother2Pole)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index, **int** Length);

SCFloatArrayRef(SCDLLCALL* InternalSuperSmoother3Pole)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index, **int** Length);

//2298

SCFloatArrayRef(SCDLLCALL* InternalZeroLagEMA)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index, **int** Length);

//2301

int (SCDLLCALL* GetTargetOrderInOCOGroupNumber)(**const int** OCOGroupNumber, s_SCTradeOrder& OrderDetails);

int (SCDLLCALL* GetStopOrderInOCOGroupNumber)(**const int** OCOGroupNumber, s_SCTradeOrder& OrderDetails);

//2342

int (SCDLLCALL* GetTradeAccountData)(n_ACSIL::s_TradeAccountDataFields& r_TradeAccountDataFields, **const** SCString& TradeAccount);

// 2363

SCFloatArrayRef(SCDLLCALL* InternalButterworth2Pole)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index, **int** Length);

SCFloatArrayRef(SCDLLCALL* InternalButterworth3Pole)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index, **int** Length);

SCFloatArrayRef(SCDLLCALL* InternalDominantCyclePeriod)(SCFloatArrayRef In, SCFloatArrayRef InstPeriod, SCFloatArrayRef Q1, SCFloatArrayRef I1, SCFloatArrayRef PhaseChange, SCFloatArrayRef Temp, SCFloatArrayRef MedianPhaseChange, SCFloatArrayRef DominantCycle, SCFloatArrayRef Out, **int** Index, **int** MedianLength);

SCFloatArrayRef(SCDLLCALL* InternalLaguerreFilter)(SCFloatArrayRef In, SCFloatArrayRef L0, SCFloatArrayRef L1, SCFloatArrayRef L2, SCFloatArrayRef L3, SCFloatArrayRef Out, **int** Index, **float** DampingFactor);

int (SCDLLCALL* GetCombineTradesIntoOriginalSummaryTradeSetting());

void (SCDLLCALL* SetCombineTradesIntoOriginalSummaryTradeSetting)(**int** NewState);

// 2367

uint32_t (SCDLLCALL* ClearCurrentTradedBidAskVolumeAllSymbols());

uint32_t (SCDLLCALL* ClearRecentBidAskVolume());

uint32_t (SCDLLCALL* ClearRecentBidAskVolumeAllSymbols());

// 2380

int (SCDLLCALL* GetUserDrawnDrawingsCount)(**int** ChartNumber, **int** ExcludeOtherStudyInstances);

// 2385

void (SCDLLCALL* RemoveStudyFromChart)(**const int** ChartNumber, **const int** StudyID);

// 2386

SCString (SCDLLCALL* GetChartTimeZone)(**const int** ChartNumber);

// 2397

int (SCDLLCALL* SetChartTradeMode)(**const int** ChartNumber, **int** Enabled);

// 2414

SCFloatArrayRef (SCDLLCALL* InternalDominantCyclePhase)(SCFloatArrayRef In, SCFloatArrayRef Out, **int** Index);

int (SCDLLCALL* IsOpenGLActive());

// 2423

int (SCDLLCALL* GetHideChartDrawingsFromOtherCharts)(**const int** ChartNumber);

```

// 2440
int (SCDLLCALL* AddDateToExclude)(const int ChartNumber, const SCDatetime& DateToExclude);
int (SCDLLCALL* DatesToExcludeClear)(const int ChartNumber);

// 2451
SCDateTime (SCDLLCALL* GetDataDelayFromChart)(const int ChartNumber);

SCFloatArrayRef(SCDLLCALL* InternalLinearRegressionSlope)(SCFloatArrayRef In, SCFloatArrayRef Out, int Index,
int Length);

SCFloatArrayRef(SCDLLCALL* InternalLinearRegressionIntercept)(SCFloatArrayRef In, SCFloatArrayRef Out, int
Index, int Length);

// 2473

int32_t(SCDLLCALL* GetStudySubgraphLineStyle)(int32_t ChartNumber, int32_t StudyID, int32_t
StudySubgraphNumber, SubgraphLineStyles& r_LineStyle);

int32_t(SCDLLCALL* SetStudySubgraphLineStyle)(int32_t ChartNumber, int32_t StudyID, int32_t
StudySubgraphNumber, SubgraphLineStyles LineStyle);

int32_t(SCDLLCALL* GetStudySubgraphLineWidth)(int32_t ChartNumber, int32_t StudyID, int32_t
StudySubgraphNumber, int32_t& r_LineWidth);

int32_t(SCDLLCALL* SetStudySubgraphLineWidth)(int32_t ChartNumber, int32_t StudyID, int32_t
StudySubgraphNumber, int32_t LineWidth);

// 2483
void (SCDLLCALL* SetCustomStudyControlBarButtonShortCaption)(int ControlBarButtonNum, const SCString&
ButtonText);

// 2484
int (SCDLLCALL* GetChartStudyInputChartStudySubgraphValues)(int ChartNumber, int StudyID, int InputIndex,
s_ChartStudySubgraphValues& r_ChartStudySubgraphValues);
int (SCDLLCALL* SetChartStudyInputChartStudySubgraphValues)(int ChartNumber, int StudyID, int InputIndex,
s_ChartStudySubgraphValues ChartStudySubgraphValues);

uint32_t (SCDLLCALL* GetStudyInternalIdentifier)(int ChartNumber, int StudyID, SCString& r_StudyName);
int (SCDLLCALL* GetChartStudyInputType)(int ChartNumber, int StudyID, int InputIndex);

int (SCDLLCALL* GetBuiltInStudyName)(const int InternalStudyIdentifier, SCString& r_StudyName);

//2486
int32_t (SCDLLCALL* Internal_GetGraphicsSetting)(const int32_t ChartNumber, const
n_ACSIL::GraphicsSettingsEnum GraphicsSetting, uint32_t& r_Color, uint32_t& r_LineWidth, SubgraphLineStyles&
r_LineStyle);

int32_t (SCDLLCALL* Internal_SetGraphicsSetting)(const int32_t ChartNumber, const
n_ACSIL::GraphicsSettingsEnum GraphicsSetting, uint32_t Color, uint32_t LineWidth, SubgraphLineStyles LineStyle);

int (SCDLLCALL* SetUseGlobalGraphicsSettings)(const int ChartNumber, int State);

// 2497
SCDateTime (SCDLLCALL* ConvertDateTimeUTCToChartTimeZone)(const SCDatetime& DateTime);

// 2500
int(SCDLLCALL* IsTradingDOMMode)();

// 2513
int (SCDLLCALL* GetUseGlobalGraphicsSettings)(const int ChartNumber);
uint32_t(SCDLLCALL* ClearMarketDepthPullingStackingData)();

// 2515
int (SCDLLCALL* GetChartStudyInputCustomStringListSize)(int ChartNumber, int StudyID, int InputIndex);

```



```

int (SCDLLCALL* GetChartStudyInputCustomStringListString)(int ChartNumber, int StudyID, int InputIndex, int
StringIndex, SCString& r_StringAtIndex);

// 2523
int (SCDLLCALL* DrawGraphics_MoveTo)(const int XCoordinate, const int YCoordinate);
int (SCDLLCALL* DrawGraphics_LineTo)(const int XCoordinate, const int YCoordinate, const uint32_t Color, const
uint32_t LineWidth, const SubgraphLineStyles LineStyle);

// 2526
int32_t (SCDLLCALL* GetChartStudyVersion)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyVersion)(int32_t ChartNumber, int32_t StudyID, int32_t Version);

int32_t (SCDLLCALL* GetChartStudyValueFormat)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyValueFormat)(int32_t ChartNumber, int32_t StudyID, int32_t ValueFormat);

int32_t (SCDLLCALL* GetChartStudyShortName)(int32_t ChartNumber, int32_t StudyID, SCString& r_ShortName);
int32_t (SCDLLCALL* SetChartStudyShortName)(int32_t ChartNumber, int32_t StudyID, const SCString& ShortName);

int32_t (SCDLLCALL* GetChartStudyGraphRegion)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyGraphRegion)(int32_t ChartNumber, int32_t StudyID, int32_t GraphRegion);

float (SCDLLCALL* GetChartStudyConstantRangeScaleAmount)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyConstantRangeScaleAmount)(int32_t ChartNumber, int32_t StudyID, const float
ScaleConstRange);

float (SCDLLCALL* GetChartStudyScaleIncrement)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyScaleIncrement)(int32_t ChartNumber, int32_t StudyID, const float
ScaleIncrement);

float (SCDLLCALL* GetChartStudyUserDefinedScaleRangeTop)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyUserDefinedScaleRangeTop)(int32_t ChartNumber, int32_t StudyID, const float
ScaleRangeTop);

float (SCDLLCALL* GetChartStudyUserDefinedScaleRangeBottom)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyUserDefinedScaleRangeBottom)(int32_t ChartNumber, int32_t StudyID, const
float ScaleRangeBottom);

int32_t (SCDLLCALL* GetChartStudyScaleRangeType)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyScaleRangeType)(int32_t ChartNumber, int32_t StudyID, const
ScaleRangeTypeEnum ScaleRangeType);

int32_t (SCDLLCALL* GetChartStudyScaleType)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyScaleType)(int32_t ChartNumber, int32_t StudyID, int32_t ScaleType);

float (SCDLLCALL* GetChartStudyScaleValueOffset)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyScaleValueOffset)(int32_t ChartNumber, int32_t StudyID, const float
ScaleValueOffset);

int32_t (SCDLLCALL* GetChartStudyDisplayAsMainPriceGraph)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyDisplayAsMainPriceGraph)(int32_t ChartNumber, int32_t StudyID, const int32_t
DisplayAsMainPriceGraph);

int32_t (SCDLLCALL* GetChartStudyHideStudy)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyHideStudy)(int32_t ChartNumber, int32_t StudyID, const int32_t HideStudy);

int32_t (SCDLLCALL* GetChartStudyDrawStudyUnderneathMainPriceGraph)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyDrawStudyUnderneathMainPriceGraph)(int32_t ChartNumber, int32_t StudyID,
const int32_t DrawStudyUnderneathMainPriceGraph);

int32_t (SCDLLCALL* GetChartStudyDisplayStudyInputValues)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyDisplayStudyInputValues)(int32_t ChartNumber, int32_t StudyID, const int32_t
DisplayStudyInputValues);

int32_t (SCDLLCALL* GetChartStudyDisplayStudyName)(int32_t ChartNumber, int32_t StudyID);
int32_t (SCDLLCALL* SetChartStudyDisplayStudyName)(int32_t ChartNumber, int32_t StudyID, const int32_t

```

```
DisplayStudyName);
```

```
int32_t (SCDLLCALL* GetChartStudyTransparencyLevel)(int32_t ChartNumber, int32_t StudyID);  
int32_t (SCDLLCALL* SetChartStudyTransparencyLevel)(int32_t ChartNumber, int32_t StudyID, const int32_t  
TransparencyLevel);
```

```
int32_t (SCDLLCALL* GetChartStudyDisplaySubgraphsNameAndValue)(int32_t ChartNumber, int32_t StudyID);  
int32_t (SCDLLCALL* SetChartStudyDisplaySubgraphsNameAndValue)(int32_t ChartNumber, int32_t StudyID, const  
int32_t DisplaySubgraphsNameAndValue);
```

```
//2527  
int32_t (SCDLLCALL* GetGraphicsSettingEnableState)(const int32_t ChartNumber, const  
n_ACSIL::GraphicsSettingsEnum GraphicsSetting);
```

```
int32_t (SCDLLCALL* SetGraphicsSettingEnableState)(const int32_t ChartNumber, const  
n_ACSIL::GraphicsSettingsEnum GraphicsSetting, int32_t EnableState);
```

```
// 2538  
int32_t (SCDLLCALL* GetChartbookFilePathAndFilename)(SCString& r_FilePathAndFilename);  
int32_t (SCDLLCALL* IsDrawDOMGraphOnChartEnabled)(const int32_t ChartNumber);  
int32_t (SCDLLCALL* IsTradingChartDOMEnabled)(const int32_t ChartNumber);  
int32_t (SCDLLCALL* IsMarketDataColumnsEnabled)(const int32_t ChartNumber);
```

```
// 2539  
int32_t (SCDLLCALL* SetDrawDOMGraphOnChart)(const int32_t ChartNumber, const int32_t Enabled);  
int32_t (SCDLLCALL* SetShowMarketDataColumns)(const int32_t ChartNumber, const int32_t Enabled);
```

```
// 2543  
int32_t (SCDLLCALL* IsChartDOMRecenterLineEnabled)(const int32_t ChartNumber);
```

```
struct s_Graphics  
{
```

```
/*  
void SetBackgroundColor(const int32_t BackgroundColor);  
void SetBackgroundMode(const int BackgroundMode);
```

```
int32_t (SCDLLCALL* DrawGraphics_SetBrush)(const n_ACSIL::s_GraphicsBrush& Brush);  
//void SetClippingRegion(const c_RegionWrapper& Region);  
void SetClippingRegionFromRectangle(const c_Rectangle& Rectangle);  
//void SetMapMode(const int Mode);  
void SetMixMode(const int Mode);  
void SetPen(const c_Pen& Pen);  
void SetTextAlign(const UINT TextAlign);  
void SetTextColor(const c_Color& TextColor);  
void SetTextFont(const c_Font& TextFont);
```

```
void ResetBackgroundColor();  
void ResetBackgroundMode();  
void ResetBrush();  
void ResetClippingRegion();  
void ResetMixMode();  
void ResetPen();  
void ResetTextAlign();  
void ResetTextColor();  
void ResetTextFont();
```

```
int GetDeviceCaps(const int Index) const;  
int CalculateFontLogicalHeight(const int PointSize) const;  
int CalculateFontPointSize(const int LogicalHeight) const;  
int GetTextHeight() const;  
int GetTextHeightWithFont(const c_Font& Font) const;  
c_Size GetTextSize(const c_StringRef& Text) const;
```

```

c_Size GetTextSizeWithFont
( const c_StringRef& Text
, const c_Font& Font
) const;

void DrawRectangle(const c_Rectangle& Rectangle) const;
void DrawRectangle
( int LeftRect
, int TopRect
, int RightRect
, int BottomRect
) const;

void DrawFocusRectangle(const c_Rectangle& Rectangle) const;
void Draw3DRectangle ( const c_Rectangle& Rectangle
, const c_Color& TopLeftColor
, const c_Color& BottomRightColor
) const;
void FillRectangle(const c_Rectangle& Rectangle, const c_Brush& Brush) const;
void FillRectangleWithColor ( const c_Rectangle& Rectangle
, const c_Color& Color
) const;

template<int t_Count>
void DrawPolygon(const s_Point (&Points)[t_Count]) const;

int DrawTextW
( const c_StringRef& Text
, c_Rectangle& r_Rectangle
, const UINT Flags
) const;
void DrawTextAt
( const c_StringRef& Text
, const int TextAnchorX
, const int TextAnchorY
) const;
void DrawTextInRectangle
( const c_StringRef& Text
, const int TextAnchorX
, const int TextAnchorY
, const c_Rectangle& Rectangle
, const UINT Options
) const;
void DrawTextWithState
( const c_StringRef& Text
, const s_Point TextAnchor
, const c_Size Size
, const UINT StateFlags
) const;

void DrawArc
( const c_Rectangle& BoundingRectangle
, int StartArcX
, int StartArcY
, int EndArcX
, int EndArcY
) const;

void DrawArc
( int LeftRect
, int TopRect
, int RightRect
, int BottomRect
, int StartArcX

```

```

, int StartArcY
, int EndArcX
, int EndArcY
) const;

```

```

void DrawEllipse
( int LeftRect
, int TopRect
, int RightRect
, int BottomRect
) const;

```

```

void DrawEllipse
( int StartPointX
, int StartPointY
, int EndPointX
, int EndPointY
, int SecondRadius
) const;

```

```

void DrawEllipseOutline
( int LeftRect
, int TopRect
, int RightRect
, int BottomRect
, const c_Brush& Brush
, int Width
, int Height
) const;

```

```

void FillEllipse
( int LeftRect
, int TopRect
, int RightRect
, int BottomRect
, const c_Brush& Brush
) const;

```

```

void SetPixel ( const int XCoordinate
, const int YCoordinate
, const c_Color& Color
) const;

```

```

void MoveTo(const int XCoordinate, const int YCoordinate) const;
void MoveTo(const int XCoordinate, const int YCoordinate, s_Point *p_OldLocation) const;
void LineTo(const int XCoordinate, const int YCoordinate) const;
s_Point GetCurrentPosition() const;

```

```

void DrawEdge
(c_Rectangle& r_Rectangle
, const UINT EdgeType
, const UINT Flags
) const;

```

```

void DrawFrameControl
(c_Rectangle& r_Rectangle
, const UINT Type
, const UINT State
);

```

```

*/

```

```

int8_t Reserve[640] = { 0 };

```

```
} Graphics;
```

```
// When adding new functions that use arrays as parameters, always use  
// SCFloatArrayRef and not SCSubgraphRef. All required internal arrays  
// must be passed in as parameters. Write a wrapper function that takes  
// a SCSubgraphRef and pass into the Internal function the necessary  
// internal arrays. This ensures back compatibility if there are changes  
// to the Subgraph structure and it allows our functions to also be used  
// internally in Sierra Chart where Subgraph internal arrays are not  
// automatically supported.
```

```
// When adding pointers to functions that return a SCString object,  
// include an extra set of parentheses around the function part.  
// Otherwise g++ will give errors.  
// Example: SCString (SCDLLCALL* DataTradeServiceName)();
```

```
// When using structures/classes in functions as parameters,  
// they must always be passed by reference and not as a copy.  
// Since other compilers can generate a different structure for them.
```

```
// Always update SC_DLL_VERSION when updating the interface  
};
```

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

```
#pragma pack(pop)
```

```
#define SCStudyGraphRef SCStudyInterfaceRef
```

```
#define GetUserDrawingByLineNumber GetUserDrawnDrawingByLineNumber
```

```
#define IsWorkingOrderStatusNoChild IsWorkingOrderStatusIgnorePendingChildren
```

```
#endif
```