

```
#include "sierrachart.h"
#include "scstudyfunctions.h"
```

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

User note: The functions in this file are intermediate level functions you can copy and call from your primary scsf\_ functions. These functions are not complete study functions. They are used by primary study functions to perform calculations. Where you see \_S at the end, this means that the function is a single step function, and only fills in the array element at the index specified by the Index parameter.

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

```
/*=====*/
```

```
SCFloatArrayRef CCI_S(SCFloatArrayRef In, SCFloatArrayRef MAOut, SCFloatArrayRef CCIOut, int Index, int Length, float Multiplier, unsigned int MovingAverageType)
```

```
{
    if (Length < 1)
        return CCIOut;

    switch (MovingAverageType)
    {
    case MOVAVGTYPE_EXPONENTIAL:
        ExponentialMovingAverage_S(In, MAOut, Index, Length);
        break;

    case MOVAVGTYPE_LINEARREGRESSION:
        LinearRegressionIndicator_S(In, MAOut, Index, Length);
        break;

    default: // Unknown moving average type
    case MOVAVGTYPE_SIMPLE:
        SimpleMovAvg_S(In, MAOut, Index, Length);
        break;

    case MOVAVGTYPE_WEIGHTED:
        WeightedMovingAverage_S(In, MAOut, Index, Length);
        break;

    case MOVAVGTYPE_WILDERS:
        WildersMovingAverage_S(In, MAOut, Index, Length);
        break;

    case MOVAVGTYPE_SIMPLE_SKIP_ZEROS:
        SimpleMovAvgSkipZeros_S(In, MAOut, Index, Length);
        break;

    case MOVAVGTYPE_SMOOTHED:
        SmoothedMovingAverage_S(In, MAOut, Index, Length);
        break;
    }

    float Num0 = 0;
    for (int j = Index; j > Index - Length && j >= 0; j--)
        Num0 += fabs(MAOut[Index] - In[j]);

    Num0 /= Length;

    CCIOut[Index] = (In[Index] - MAOut[Index]) / (Num0 * Multiplier);

    return CCIOut;
}
```

```
/*=====*/
```

```

SCFloatArrayRef CCISMA_S(SCFloatArrayRef In, SCFloatArrayRef SMAOut, SCFloatArrayRef CCIOut, int Index, int
Length, float Multiplier)
{
    return CCI_S(In, SMAOut, CCIOut, Index, Length, Multiplier, MOVAVGTYPE_SIMPLE);
}

/*=====*/
float GetHighest(SCFloatArrayRef In, int StartIndex, int Length)
{
    float High = -FLT_MAX;

    // Get the high from the last Length indexes in the In array
    for (int SrcIndex = StartIndex; SrcIndex > StartIndex - Length; --SrcIndex)
    {
        if (SrcIndex < 0 || SrcIndex >= In.GetArraySize())
            continue;

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

    return High;
}

/*=====*/
SCFloatArrayRef Highest_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    // Get the high from the last Length indexes in the In array
    float High = GetHighest(In, Index, Length);

    // Put the high in the Out array
    if (High == -FLT_MAX)
        Out[Index] = 0.0f;
    else
        Out[Index] = High;

    return Out;
}

/*=====*/
float GetLowest(SCFloatArrayRef In, int StartIndex, int Length)
{
    float Low = FLT_MAX;

    // Get the low from the last Length indexes in the In array
    for (int SrcIndex = StartIndex; SrcIndex > StartIndex - Length; --SrcIndex)
    {
        if (SrcIndex < 0 || SrcIndex >= In.GetArraySize())
            continue;

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

    return Low;
}

/*=====*/
SCFloatArrayRef Lowest_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    // Get the low from the last Length indexes in the In array
    float Low = GetLowest(In, Index, Length);

    // Put the low in the Out array
    if (Low == FLT_MAX)

```

```

        Out[Index] = 0.0f;
    else
        Out[Index] = Low;

    return Out;
}

/*=====*/
float TrueRange(SCBaseDataRef BaseDataIn, int Index)
{
    if (Index == 0)
        return BaseDataIn[SC_HIGH][0] - BaseDataIn[SC_LOW][0];

    float HighLowRange = BaseDataIn[SC_HIGH][Index] - BaseDataIn[SC_LOW][Index];
    float HighToPreviousCloseDifference = fabs(BaseDataIn[SC_HIGH][Index] - BaseDataIn[SC_LAST][Index - 1]);
    float LowToPreviousCloseDifference = fabs(BaseDataIn[SC_LOW][Index] - BaseDataIn[SC_LAST][Index - 1]);

    return max(HighLowRange, max(HighToPreviousCloseDifference, LowToPreviousCloseDifference));
}

/*=====*/
SCFloatArrayRef TrueRange_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, int Index)
{
    Out[Index] = TrueRange(BaseDataIn, Index);

    return Out;
}

/*=====*/
SCFloatArrayRef AverageTrueRange_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef TROut, SCFloatArrayRef
ATROut, int Index, int Length, unsigned int MovingAverageType)
{
    // True Range
    TrueRange_S(BaseDataIn, TROut, Index);

    // Average
    MovingAverage_S(TROut, ATROut, MovingAverageType, Index, Length);

    return ATROut;
}

/*=====*/
SCFloatArrayRef OnBalanceVolume_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, int Index)
{
    // Min index of 1
    if (Index < 1)
        return Out;

    if (BaseDataIn[SC_LAST][Index] > BaseDataIn[SC_LAST][Index - 1])
        Out[Index] = Out[Index - 1] + BaseDataIn[SC_VOLUME][Index];
    else if (BaseDataIn[SC_LAST][Index] < BaseDataIn[SC_LAST][Index - 1])
        Out[Index] = Out[Index - 1] - BaseDataIn[SC_VOLUME][Index];
    else // Equal
        Out[Index] = Out[Index - 1];

    return Out;
}

/*=====*/
SCFloatArrayRef OnBalanceVolumeShortTerm_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, SCFloatArrayRef
OBVTemp, int Index, int Length)
{
    // Need at least two bars in chart
    if (Index < 1)

```

```

    return Out;

// OBV
if (BaseDataIn[SC_LAST][Index - 1] > BaseDataIn[SC_LAST][Index])
{
    // Since current close is less than previous, subtract
    OBVTemp[Index] = -BaseDataIn[SC_VOLUME][Index];
}
else if (BaseDataIn[SC_LAST][Index - 1] < BaseDataIn[SC_LAST][Index])
{
    // Add
    OBVTemp[Index] = BaseDataIn[SC_VOLUME][Index];
}
else // Equal
{
    // Do not change
    OBVTemp[Index] = 0;
}

// Prevent looking back into a negative array index
if (Index < Length)
{
    // Regular OBV for first N terms
    Out[Index] = Out[Index - 1] + OBVTemp[Index];
}
else
{
    // Adjust current OBV by subtracting what we did to get the
    // previous OBV. Example, if at position 11, data was added to OBV, then
    // at position 12, that data would be subtracted to make it a
    // 'zero' start.
    Out[Index] = Out[Index - 1] + OBVTemp[Index] - OBVTemp[Index - Length];
}

return Out;
}

/*=====*/
SCFloatArrayRef MovingAverage_S(SCFloatArrayRef In, SCFloatArrayRef Out, unsigned int MovingAverageType, int
Index, int Length)
{
    switch (MovingAverageType)
    {
        case MOVAVGTYPE_EXPONENTIAL:
            return ExponentialMovingAverage_S(In, Out, Index, Length);

        case MOVAVGTYPE_LINEARREGRESSION:
            return LinearRegressionIndicator_S(In, Out, Index, Length);

        default: // Unknown moving average type
            case MOVAVGTYPE_SIMPLE:
                return SimpleMovAvg_S(In, Out, Index, Length);

            case MOVAVGTYPE_WEIGHTED:
                return WeightedMovingAverage_S(In, Out, Index, Length);

            case MOVAVGTYPE_WILDERS:
                return WildersMovingAverage_S(In, Out, Index, Length);

            case MOVAVGTYPE_SIMPLE_SKIP_ZEROS:
                return SimpleMovAvgSkipZeros_S(In, Out, Index, Length);

            case MOVAVGTYPE_SMOOTHED:
                return SmoothedMovingAverage_S(In, Out, Index, Length);
    }
}

```

```

    }
}

/*=====*/
void Stochastic_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef FastKOut, SCFloatArrayRef FastDOut,
SCFloatArrayRef SlowDOut, int Index, int FastKLength, int FastDLength, int SlowDLength, unsigned int
MovingAverageType)
{
    Stochastic2_S(BaseDataIn[SC_HIGH], BaseDataIn[SC_LOW], BaseDataIn[SC_LAST], FastKOut, FastDOut,
SlowDOut, Index, FastKLength, FastDLength, SlowDLength, MovingAverageType);
}

/*=====*/
void Stochastic2_S(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCFloatArrayRef InputDataLast,
SCFloatArrayRef FastKOut, SCFloatArrayRef FastDOut, SCFloatArrayRef SlowDOut, int Index, int FastKLength, int
FastDLength, int SlowDLength, unsigned int MovingAverageType)
{
    float High = GetHighest(InputDataHigh, Index, FastKLength);
    float Low = GetLowest(InputDataLow, Index, FastKLength);

    float Range = High - Low;
    if (Range == 0)
        FastKOut[Index] = 100.0f;
    else
        FastKOut[Index] = 100.0f * (InputDataLast[Index] - Low) / Range;

    MovingAverage_S(FastKOut, FastDOut, MovingAverageType, Index, FastDLength);
    MovingAverage_S(FastDOut, SlowDOut, MovingAverageType, Index, SlowDLength);
}

/*=====
This function calculates an exponential moving average of the In array
and puts the results in the Out array.
-----*/
SCFloatArrayRef ExponentialMovingAverage_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Index >= In.GetArraySize())
        return Out;

    if (Index < 1 || Length < 1)
        return Out;

    if (Index < Length - 1)
        Length = Index + 1;

    double Multiplier1 = 2.0f / (Length + 1);
    double Multiplier2 = 1.0f - Multiplier1;
    double PreviousMovingAverageValue = Out[Index - 1];

    //Check for a previous moving average value of 0 so we can properly initialize the previous value, and also check for
out of range values.
    if (PreviousMovingAverageValue == 0.0f
        || !((PreviousMovingAverageValue > -FLT_MAX) && (PreviousMovingAverageValue < FLT_MAX)))
    {
        Out[Index - 1] = In[Index - 1];
    }

    float Average = static_cast<float>((Multiplier1 * In[Index]) + (Multiplier2 * Out[Index - 1]));
    Out[Index] = Average;

    return Out;
}

```

```

/*=====*/
void CalculateRegressionStatistics(SCFloatArrayRef In, double &Slope, double &Y_Intercept, int Index, int Length)
{
    double sum_y=0, sum_x=0, sum_x2=0, sum_x_2=0, sum_xy=0;

    if (Index < ( Length - 1 ) )
    {
        Index = Length - 1;
    }

    sum_x = static_cast<float>((Length * (Length+1)) / 2.0);
    sum_x_2 = sum_x * sum_x;
    sum_y = GetSummation(In, Index, Length);
    for(int Offset=0; Offset < Length; Offset++)
    {
        sum_xy += In[Index-Offset] * (Length-Offset);
    }

    sum_x2=(Length+1)*Length*(2*Length+1)/6.0f;

    double b_numerator    = (Length * sum_xy - sum_x * sum_y);
    double b_denominator  = Length * sum_x2 - sum_x_2;
    Slope  = b_numerator / b_denominator;
    Y_Intercept = (sum_y - Slope * sum_x) / Length;
}

```

```

/*=====*/
void CalculateLogLogRegressionStatistics(SCFloatArrayRef In, double &Slope, double &Y_Intercept, int Index, int Length)
{
    if (Index <= (Length - 1)) //This prevents processing if there is insufficient data and prevents processing of index 0
    which is not possible in order to avoid taking the logarithm of 0.
        return;

    double sum_y = 0, sum_x = 0, sum_x2 = 0, sum_x_2 = 0, sum_xy = 0;

    for (int Offset = 0; Offset < Length; Offset++)
    {
        float ValueAtIndex = In[Index - Offset];
        int XCoordinate = Index - Offset;

        sum_x += log(XCoordinate);

        sum_x2 += log(XCoordinate) * log(XCoordinate);

        if (ValueAtIndex != 0)
        {
            sum_y += log(ValueAtIndex);

            sum_xy += log(XCoordinate) * log(ValueAtIndex);
        }
    }

    sum_x_2 = sum_x * sum_x;

    double b_numerator = (Length * sum_xy - sum_x * sum_y);
    double b_denominator = Length * sum_x2 - sum_x_2;
    Slope = b_numerator / b_denominator;
    Y_Intercept = (sum_y - Slope * sum_x) / Length;
}

```

```

/*=====*/
SCFloatArrayRef LinearRegressionIndicator_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Index >= In.GetArraySize())
        return Out;

    double Slope = 0;
    double Y_Intercept = 0;
    CalculateRegressionStatistics(In, Slope, Y_Intercept, Index, Length);

    //compute the end point of the linear regression trendline == linear regression indicator.
    Out[Index] = static_cast<float>(Y_Intercept + Slope * Length);

    return Out;
}

/*=====*/
// Calculates the Linear Regression Indicator. Also computes standard error which is the last value of a linear regression
trend line.

SCFloatArrayRef LinearRegressionIndicatorAndStdErr_S(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef
StdErr, int Index, int Length)
{
    if (Index >= In.GetArraySize())
        return Out;

    double Slope = 0;
    double Y_Intercept = 0;
    double sum_y = 0, sum_x = 0, sum_x2 = 0, sum_y2 = 0, sum_x_2 = 0, sum_y_2 = 0, sum_xy = 0;

    if (Index < (Length - 1))
    {
        Index = Length - 1;
    }

    sum_x = static_cast<float>((Length * (Length + 1)) / 2.0);
    sum_x_2 = sum_x * sum_x;
    sum_y = GetSummation(In, Index, Length);
    sum_y_2 = sum_y * sum_y;
    for (int nn = 0; nn < Length; nn++)
    {
        sum_y2 += In[Index - nn] * In[Index - nn];

        sum_xy += In[Index - nn] * (Length - nn);
    }

    sum_x2 = (Length + 1)*Length*(2 * Length + 1) / 6.0f;

    double Slope_numerator = (Length * sum_xy - sum_x * sum_y);
    double Slope_denominator = Length * sum_x2 - sum_x_2;
    Slope = Slope_numerator / Slope_denominator;
    Y_Intercept = (sum_y - Slope * sum_x) / Length;

    //compute the end point of the linear regression trendline == linear regression indicator.
    Out[Index] = static_cast<float>(Y_Intercept + Slope * Length);

    // compute the standard error
    double temp = (1.0 / (Length * (Length-2.0))) * ( Length * sum_y2 - sum_y_2 - Slope * Slope_numerator);

    // Check to ensure the data is non-negative before performing the square root operation.
    if(temp > 0)
    {

```

```

        StdErr[Index] = static_cast<float>(sqrt(temp));
    }
    else
    {
        StdErr[Index] = 0;
    }

    return Out;
}

/*=====
This function calculates an adaptive moving average of the In array and
puts the results in the Out array.
-----*/
SCFloatArrayRef AdaptiveMovAvg_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length, float
FastSmoothConst, float SlowSmoothConst)
{
    if (Index >= In.GetArraySize())
        return Out;

    if (Length < 1)
        return Out;

    FastSmoothConst = 2 / (FastSmoothConst + 1);
    SlowSmoothConst = 2 / (SlowSmoothConst + 1);

    int StartIndex = max(Length, Index);

    float Direction = (In[StartIndex] - In[StartIndex - Length]);
    float InputArrayDiffSum = 0.0f;

    for (int SumIndex = StartIndex - (Length - 1); SumIndex <= StartIndex; SumIndex++)
    {
        InputArrayDiffSum += fabs(In[SumIndex] - In[SumIndex - 1]);
    }

    float Volatility = InputArrayDiffSum;

    if (Volatility == 0.0f)
        Volatility = .000001f;

    float DirectionVolatilityRatio = fabs(Direction / Volatility);

    float Multiplier = DirectionVolatilityRatio * (FastSmoothConst - SlowSmoothConst) + SlowSmoothConst;

    Multiplier = Multiplier * Multiplier;

    if (Out[StartIndex - 1] == 0.0f)
        Out[StartIndex] = In[StartIndex - 1] + Multiplier * (In[StartIndex] - In[StartIndex - 1]);
    else
        Out[StartIndex] = Out[StartIndex - 1] + Multiplier * (In[StartIndex] - Out[StartIndex - 1]);

    return Out;
}

/*=====
This function calculates a simple moving average of the In array and puts
the results in the Out array.
-----*/
SCFloatArrayRef SimpleMovAvg_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Length < 1)
        return Out;

    float Sum = 0;

```



```

    if (Index < Length - 1)
        Length = Index + 1;

    if (Index >= In.GetExtendedArraySize() || Index < 0)
        return Out;

    for(int InputIndex = Index - Length + 1; InputIndex <= Index; InputIndex++)
        Sum = Sum + In.GetAt(InputIndex);

    Out[Index] = Sum / Length;

    return Out;
}

/*=====*/
SCFloatArrayRef MovingMedian_S(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef Temp, int Index, int
Length)
{
    //Adjust the Length

    if (Index < Length - 1)
        Length = Index + 1;

    if (Index >= In.GetExtendedArraySize() || Index < 0)
        return Out;

    const int BaseIndex = Index - Length + 1;

    if (BaseIndex < 0)//Unnecessary but for safety.
        return Out;

    Temp.AllocateArray();

    // Make a copy of the elements to be sorted
    for (int BarIndex = BaseIndex; BarIndex <= Index; ++BarIndex)
        Temp.GetAt(BarIndex) = In.GetAt(BarIndex);

    // This section of code sorts the temporary array just enough to figure
    // out what the middle element is, which is what we need to know to get
    // the median.
    {
        int MiddleIndex = BaseIndex + Length / 2;

        int Bottom = BaseIndex;
        int Top = Index;
        while (Bottom < Top)
        {
            float MiddleValue = Temp[MiddleIndex];

            int NextBottom = Bottom;
            int NextTop = Top;
            do
            {
                while (Temp[NextBottom] < MiddleValue)
                    ++NextBottom;

                while (Temp[NextTop] > MiddleValue)
                    --NextTop;

                if (NextBottom <= NextTop)
                {
                    float SwapTemp = Temp[NextBottom];
                    Temp[NextBottom] = Temp[NextTop];
                    Temp[NextTop] = SwapTemp;
                }
            }
        }
    }
}

```

```

        ++NextBottom;
        --NextTop;
    }
} while (NextBottom <= NextTop);

if (NextBottom > MiddleIndex)
    Top = NextTop;

if (NextTop < MiddleIndex)
    Bottom = NextBottom;
}
}

if (Length % 2 != 0) //Has a remainder
{
    Out[Index] = Temp[BaseIndex + Length / 2];
}
else
{
    Out[Index] = (Temp[BaseIndex + Length / 2 - 1] + Temp[BaseIndex + Length / 2]) / 2;
}

return Out;
}

/*=====*/
SCFloatArrayRef SimpleMovAvgSkipZeros_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    int count = 0;
    float Sum = 0.0f;

    if (Length > Index + 1)
        Length = Index + 1;

    for(int ArrayIndex= Index - Length + 1; ArrayIndex <= Index; ArrayIndex++)
    {
        float Value = In[ArrayIndex];

        if (Value != 0.0f)
        {
            Sum += Value;
            count++;
        }
    }

    if (count > 0)
    {
        Out[Index] = Sum / count;
    }
    else
    {
        Out[Index] = 0.0f;
    }

    return Out;
}

/*=====*/
SCFloatArrayRef WildersMovingAverage_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Index < 1)
        return Out;
}

```

```

    if (Out[Index - 1] == 0.0)
    {
        SimpleMovAvgSkipZeros_S(In, Out, Index - 1, Length);
    }

    if (Out[Index - 1] != 0.0)
    {
        Out[Index] = Out[Index - 1]
            + (
                (1.0f / static_cast<float>(Length))
                *(In[Index] - Out[Index - 1])
            );
    }

    return Out;
}

/*=====*/
SCFloatArrayRef WeightedMovingAverage_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Length < 1)
        return Out;

    // Adjust Length based on Index in the case where the Length is greater the available number of array elements.
    if (Index < Length - 1)
        Length = Index + 1;

    if (Index >= In.GetExtendedArraySize() || Index < 0)
        return Out;

    double Sum = 0;
    double Divider = (Length * (Length + 1.0) / 2.0);

    int CurrentWeight = Length;
    const int EndIndex = Index - Length;
    for (int PriorIndex = Index; PriorIndex > EndIndex; PriorIndex--)
    {
        Sum += In.GetAt(PriorIndex) * static_cast<float>(CurrentWeight);
        CurrentWeight--;
    }

    Out[Index] = static_cast<float>(Sum / Divider);

    return Out;
}

/*=====*/
SCFloatArrayRef HullMovingAverage_S(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef InternalArray1,
SCFloatArrayRef InternalArray2, SCFloatArrayRef InternalArray3, int Index, int Length)
{
    int HalfLength = Length/2; // integer of Length/2

    WeightedMovingAverage_S(In, InternalArray1, Index, HalfLength);
    WeightedMovingAverage_S(In, InternalArray2, Index, Length);

    float WeightedAverage1AtIndex = InternalArray1[Index];
    float WeightedAverage2AtIndex = InternalArray2[Index];
    InternalArray3[Index] = 2*WeightedAverage1AtIndex - WeightedAverage2AtIndex;

    int RoundedSquareRootOfLength = int(sqrt(static_cast<double>(Length)) + 0.5); // rounded square root of Length
    WeightedMovingAverage_S(InternalArray3, Out, Index, RoundedSquareRootOfLength); // HMA

    return Out;
}

```

```

}

/*=====*/
SCFloatArrayRef TriangularMovingAverage_S(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef
InternalArray1, int Index, int Length)
{

    int Length1, Length2;

    if(Length % 2)
    {
        Length1 = Length2 = Length/2 + 1;
    }
    else
    {
        Length1 = Length/2;
        Length2 = Length1 + 1;
    }

    SimpleMovAvg_S(In, InternalArray1, Index, Length1);
    SimpleMovAvg_S( InternalArray1, Out, Index, Length2);

    return Out;
}

/*=====*/
SCFloatArrayRef VolumeWeightedMovingAverage_S(SCFloatArrayRef InPrice, SCFloatArrayRef InVolume,
SCFloatArrayRef Out, int Index, int Length)
{

    float summPV = 0;
    float summV = 0;

    for(int i = max(0, Index - Length + 1); i<=Index; i++)
    {
        summPV += InPrice[i]*InVolume[i];
        summV += InVolume[i];
    }

    if (summV!=0.0f)
    {
        Out[Index] = summPV/summV;
    }
    else
    {
        Out[Index] = 0;
    }

    return Out;
}

/*=====*/
void GetStandardDeviation(SCFloatArrayRef In, float& Out, int StartIndex, int Length)
{
    double Variance = 0;

    Variance = GetVariance(In, StartIndex, Length);

    if(Variance < 0)
    {
        Out = 0;
    }
    else
    {

```

```

    Out = static_cast<float>(sqrt(Variance));
}
}

/*=====*/
SCFloatArrayRef StandardDeviation_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    GetStandardDeviation(In, Out[Index], Index, Length);

    return Out;
}

/*=====*/
SCFloatArrayRef Ergodic_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int LongEMALength, int
ShortEMALength, float Multiplier,
    SCFloatArrayRef InternalArray1, SCFloatArrayRef InternalArray2, SCFloatArrayRef InternalArray3, SCFloatArrayRef
InternalArray4, SCFloatArrayRef InternalArray5, SCFloatArrayRef InternalArray6)
{
    // Formula:
    // Numerator = EMA( EMA(Price - LastPrice, LongEMALength), ShortEMALength)
    // Denominator = EMA( EMA( Abs(Price - LastPrice), LongEMALength), ShortEMALength)
    // TSI = Multiplier * Numerator / Denominator

    if (Index < 1)
        return Out; // Not enough elements

    // Internal array names
    SCFloatArrayRef PriceChangeArray = InternalArray1;
    //InternalArray2
    SCFloatArrayRef NumeratorArray = InternalArray3;
    SCFloatArrayRef AbsPriceChangeArray = InternalArray4;
    //InternalArray5
    SCFloatArrayRef DenominatorArray = InternalArray6;

    float PriceChange = In[Index] - In[Index - 1]; // Price - LastPrice

    // Numerator
    PriceChangeArray[Index] = PriceChange;
    ExponentialMovingAverage_S(PriceChangeArray, InternalArray2, Index, LongEMALength);
    ExponentialMovingAverage_S(InternalArray2, NumeratorArray, Index, ShortEMALength);
    float Numerator = NumeratorArray[Index];

    // Denominator
    AbsPriceChangeArray[Index] = abs(PriceChange);
    ExponentialMovingAverage_S(AbsPriceChangeArray, InternalArray5, Index, LongEMALength);
    ExponentialMovingAverage_S(InternalArray5, DenominatorArray, Index, ShortEMALength);
    float Denominator = DenominatorArray[Index];

    Out[Index] = Multiplier * Numerator / Denominator;

    return Out;
}

/*=====*/
SCFloatArrayRef Keltner_S(SCBaseDataRef BaseDataIn, 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)
{
    MovingAverage_S(In, KeltnerAverageOut, KeltnerMAType, Index, KeltnerMALength);

    AverageTrueRange_S(BaseDataIn, InternalArray1, InternalArray2, Index, TrueRangeMALength, TrueRangeMAType);

    TopBandOut[Index] = KeltnerAverageOut[Index] + InternalArray2[Index] * TopBandMultiplier;

```

```

BottomBandOut[Index] = KeltnerAverageOut[Index] - InternalArray2[Index] * BottomBandMultiplier;

return KeltnerAverageOut;
}

/*=====*/
float WellesSum(float In, int Index, int Length, SCFloatArrayRef Out)
{
    if (Index == 0)
    {
        Out[0] = In;
    }
    else
    {
        if (Index < Length)
            Out[Index] = Out[Index - 1] + In;
        else
            Out[Index] = Out[Index - 1] - (Out[Index - 1] / Length) + In;
    }

    return Out[Index];
}

/*=====*/
SCFloatArrayRef WellesSum_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    WellesSum(In[Index], Index, Length, Out);

    return Out;
}

/*=====
Local function
This function is used by DMI, ADX, and ADXR.
=====*/
void DirectionalMovementTrueRangeSummation(SCBaseDataRef BaseDataIn, int Index, int Length,
    SCFloatArrayRef InternalTrueRangeSummation, SCFloatArrayRef InternalPosDM, SCFloatArrayRef InternalNegDM)
{
    if (Index < 1)
        return;

    float HighChange = BaseDataIn[SC_HIGH][Index] - BaseDataIn[SC_HIGH][Index - 1];
    float LowChange = BaseDataIn[SC_LOW][Index - 1] - BaseDataIn[SC_LOW][Index];

    float PercentDifference = fabs(1 - (HighChange / LowChange)); // .001 = .1%

    {
        float Positive = 0.0f;
        if (HighChange > LowChange && HighChange > 0 && PercentDifference >= 0.001)
            Positive = HighChange;

        WellesSum(Positive, Index, Length, InternalPosDM);
    }

    {
        float Negative = 0.0f;
        if (LowChange > HighChange && LowChange > 0 && PercentDifference >= 0.001)
            Negative = LowChange;

        WellesSum(Negative, Index, Length, InternalNegDM);
    }

    float TrueRangeValue = TrueRange(BaseDataIn, Index);
    WellesSum(TrueRangeValue, Index, Length, InternalTrueRangeSummation);
}

```

```

/*=====*/
void DMI_S(SCBaseDataRef BaseDataIn, int Index, int Length, int DisableRounding,
    SCFloatArrayRef PosDMIOut, SCFloatArrayRef NegDMIOut, SCFloatArrayRef DiffDMIOut, SCFloatArrayRef
InternalTrueRangeSummation, SCFloatArrayRef InternalPosDM, SCFloatArrayRef InternalNegDM)
{
    DirectionalMovementTrueRangeSummation(BaseDataIn, Index, Length,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM);

    if (InternalTrueRangeSummation[Index] == 0.0f) // Simple prevention of dividing by zero
    {
        PosDMIOut[Index] = PosDMIOut[Index - 1];
        NegDMIOut[Index] = NegDMIOut[Index - 1];
    }
    else
    {
        PosDMIOut[Index] = 100.0f * InternalPosDM[Index] / InternalTrueRangeSummation[Index];
        NegDMIOut[Index] = 100.0f * InternalNegDM[Index] / InternalTrueRangeSummation[Index];
    }

    DiffDMIOut[Index] = fabs(PosDMIOut[Index] - NegDMIOut[Index]);
}

/*=====*/
SCFloatArrayRef DMIDiff_S(SCBaseDataRef BaseDataIn,
    int Index,
    int Length,
    SCFloatArrayRef Out,
    SCFloatArrayRef InternalTrueRangeSummation, SCFloatArrayRef InternalPosDM, SCFloatArrayRef InternalNegDM)
{
    if (Index <= Length - 1)
    {
        Out[Index] = 0.0f;
        return Out;
    }

    DirectionalMovementTrueRangeSummation(BaseDataIn, Index, Length,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM);

    float TrueRangeSum = InternalTrueRangeSummation[Index];

    if (TrueRangeSum == 0.0f)
    {
        Out[Index] = Out[Index-1];
    }
    else
    {
        float PosDMI = 100.0f * InternalPosDM[Index] / TrueRangeSum;
        float NegDMI = 100.0f * InternalNegDM[Index] / TrueRangeSum;
        Out[Index] = PosDMI - NegDMI;
    }

    return Out;
}

/*=====*/
SCFloatArrayRef ADX_S(SCBaseDataRef BaseDataIn,
    int Index,
    int DXLength, int DXMovAvgLength,

```

```

SCFloatArrayRef Out,
SCFloatArrayRef InternalTrueRangeSummation,
SCFloatArrayRef InternalPosDM,
SCFloatArrayRef InternalNegDM,
SCFloatArrayRef InternalDX)
{
    DirectionalMovementTrueRangeSummation(BaseDataIn, Index, DXLength,
        InternalTrueRangeSummation, InternalPosDM, InternalNegDM);

    // Compute +DMI and -DMI identically to how it is done in DMI without rounding
    float PosDMI = 100.0f, NegDMI = 100.0f;
    float TrueRangeSum = InternalTrueRangeSummation[Index];

    if (TrueRangeSum != 0.0f)
    {
        PosDMI *= InternalPosDM[Index] / TrueRangeSum;
        NegDMI *= InternalNegDM[Index] / TrueRangeSum;
    }

    // if (PosDMI - NegDMI == 0.0f) // Quick zero (also preventing potential divide by zero)
    //     InternalDX[Index] = 0.0f;
    // else if (PosDMI + NegDMI == 0.0f) // Simple prevention of dividing by zero
    //     InternalDX[Index] = 100.0f;
    // else
    //     InternalDX[Index] = 100.0f * fabs(PosDMI - NegDMI) / (PosDMI + NegDMI);

    if (PosDMI + NegDMI == 0.0f) // Simple prevention of dividing by zero
        InternalDX[Index] = InternalDX[Index - 1];
    else
        InternalDX[Index] = 100.0f * fabs(PosDMI - NegDMI) / (PosDMI + NegDMI);

    if (Index < DXLength - 1)
        return Out;

    if (Index == DXLength + DXMovAvgLength - 1)
    {
        Out[Index] = 0.0f;
        for (int IDXIndex = 0; IDXIndex < DXMovAvgLength; ++IDXIndex)
            Out[Index] += InternalDX[Index - IDXIndex];

        Out[Index] = Out[Index] / (DXMovAvgLength); //DXLength +
    }
    else if (Index > DXLength + DXMovAvgLength - 1)
    {
        // Take the average of DX to get ADX
        WildersMovingAverage_S(InternalDX, Out, Index, DXMovAvgLength);
    }

    return Out;
}

```

```

/*=====
    Note: The InternalADX array does not contain the same values as the
    output of ADX.
    -----*/

```

```

SCFloatArrayRef ADXR_S(SCBaseDataRef BaseDataIn,
    int Index,
    int DXLength, int DXMovAvgLength, int ADXRInterval,
    SCFloatArrayRef Out,
    SCFloatArrayRef InternalTrueRangeSummation, SCFloatArrayRef InternalPosDM, SCFloatArrayRef InternalNegDM,
    SCFloatArrayRef InternalDX, SCFloatArrayRef InternalADX)
{
    if (Index < 1)
        return Out;
}

```



```

DirectionalMovementTrueRangeSummation(BaseDataIn, Index, DXLength,
    InternalTrueRangeSummation, InternalPosDM, InternalNegDM);

// Compute +DMI and -DMI identically to how it is done in DMI without rounding
float PosDMI = 100.0f, NegDMI = 100.0f;
float TrueRangeSum = InternalTrueRangeSummation[Index];
if (TrueRangeSum != 0.0f)
{
    PosDMI *= InternalPosDM[Index] / TrueRangeSum;
    NegDMI *= InternalNegDM[Index] / TrueRangeSum;
}

if (PosDMI + NegDMI == 0.0f) // Simple prevention of dividing by zero
    InternalDX[Index] = InternalDX[Index - 1];
else
    InternalDX[Index] = 100.0f * fabs(PosDMI - NegDMI) / (PosDMI + NegDMI);

if (Index < DXLength - 1)
    return Out;

if (Index == DXLength + DXMovAvgLength - 1)
{
    InternalADX[Index] = 0.0f;

    for (int IDXIndex = 0; IDXIndex < DXMovAvgLength; ++IDXIndex)
        InternalADX[Index] += InternalDX[Index - IDXIndex];

    InternalADX[Index] = InternalADX[Index] / DXMovAvgLength;
}
else if (Index > DXLength + DXMovAvgLength - 1)
{
    // Take the average of DX to get ADX
    WildersMovingAverage_S(InternalDX, InternalADX, Index, DXMovAvgLength);
}

if (Index < DXLength + DXMovAvgLength + ADXRInterval - 2)
    return Out;

Out[Index] = (InternalADX[Index] + InternalADX[Index - ADXRInterval + 1]) * 0.5f;

return Out;
}

/*=====*/
SCFloatArrayRef RSI_S(SCFloatArrayRef In,
    SCFloatArrayRef RSIOut,
    SCFloatArrayRef UpSumsTemp,
    SCFloatArrayRef DownSumsTemp,
    SCFloatArrayRef SmoothedUpSumsTemp,
    SCFloatArrayRef SmoothedDownSumsTemp,
    int Index, unsigned int AveragingType, int Length)
{
    if (Length < 1 || Index < 1)
        return RSIOut;

    // calculate Up/Down sums
    float PreviousValue = In[Index - 1];
    float CurrentValue = In[Index];

    if (CurrentValue > PreviousValue)
    {
        // upward change
        UpSumsTemp[Index] = CurrentValue - PreviousValue;
        DownSumsTemp[Index] = 0;
    }

```

```

    }
    else
    {
        UpSumsTemp[Index] = 0;
        DownSumsTemp[Index] = PreviousValue - CurrentValue;
    }

    // smooth the up/down sums
    MovingAverage_S(UpSumsTemp, SmoothedUpSumsTemp, AveragingType, Index, Length);
    MovingAverage_S(DownSumsTemp, SmoothedDownSumsTemp, AveragingType, Index, Length);

    // compute RSI
    if (SmoothedDownSumsTemp[Index] != 0.0f) // avoid division by zero
    {
        RSIOut[Index] = 100.0f - 100.0f / (1 + SmoothedUpSumsTemp[Index] / (SmoothedDownSumsTemp[Index]));
    }
    else
    {
        RSIOut[Index] = 100; // RSIOut[Index - 1];
    }

    return RSIOut;
}

/*=====*/
SCFloatArrayRef SmoothedMovingAverage_S(SCFloatArrayRef In, SCFloatArrayRef SmoothedAverageOut, int Index, int
Length)
{
    if (Index < Length - 1)
        SmoothedAverageOut[Index] = 0;

    else if (Index == Length - 1)
    {
        SimpleMovAvg_S(In, SmoothedAverageOut, Length - 1, Length);
    }

    else
    {
        SmoothedAverageOut[Index] = (Length * SmoothedAverageOut[Index - 1] - SmoothedAverageOut[Index - 1] +
In[Index]) / Length;
    }

    return SmoothedAverageOut;
}

/*=====*/
SCFloatArrayRef MACD_S(SCFloatArrayRef In, SCFloatArrayRef FastMAOut, SCFloatArrayRef SlowMAOut,
SCFloatArrayRef MACDOut, SCFloatArrayRef MACDMAOut, SCFloatArrayRef MACDDiffOut, int Index, int
FastMALength, int SlowMALength, int MACDMALength, int MovAvgType)
{
    MovingAverage_S(In, FastMAOut, MovAvgType, Index, FastMALength);
    MovingAverage_S(In, SlowMAOut, MovAvgType, Index, SlowMALength);

    MACDOut[Index] = FastMAOut[Index] - SlowMAOut[Index];

    if (Index < max(SlowMALength, FastMALength) + MACDMALength)
        return MACDOut;

    MovingAverage_S(MACDOut, MACDMAOut, MovAvgType, Index, MACDMALength);

    MACDDiffOut[Index] = MACDOut[Index] - MACDMAOut[Index];

```

```

    return MACDOut;
}

/*=====*/
SCFloatArrayRef TEMA_S(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef InternalArray1, SCFloatArrayRef
InternalArray2, SCFloatArrayRef InternalArray3, int Index, int Length)
{
    ExponentialMovingAverage_S(In, InternalArray1, Index, Length);
    ExponentialMovingAverage_S(InternalArray1, InternalArray2, Index, Length);
    ExponentialMovingAverage_S(InternalArray2, InternalArray3, Index, Length);

    Out[Index] = 3.0f*InternalArray1[Index] - 3.0f*InternalArray2[Index] + InternalArray3[Index];
    return Out;
}

/*=====*/
SCFloatArrayRef BollingerBands_S(SCFloatArrayRef In, SCFloatArrayRef Avg, SCFloatArrayRef TopBand,
SCFloatArrayRef BottomBand, SCFloatArrayRef StdDev, int Index, int Length, float Multiplier, int MovAvgType)
{
    if(Length < 1)
    {
        return Avg;
    }

    StandardDeviation_S( In, StdDev, Index, Length);

    MovingAverage_S( In, Avg, MovAvgType, Index, Length);

    TopBand[Index] = static_cast<float>(static_cast<double>(Avg[Index]) + (static_cast<double>(StdDev[Index]) *
static_cast<double>(Multiplier)));
    BottomBand[Index] = static_cast<float>(static_cast<double>(Avg[Index]) - (static_cast<double>(StdDev[Index]) *
static_cast<double>(Multiplier)));
    return Avg;
}

/*=====*/
//This calculates the Bollinger Bands using a different method. The difference is the standard deviation is calculated on
the moving average rather than the underlying chart data.
SCFloatArrayRef BollingerBands_StandardDeviationOfAverage_S(SCFloatArrayRef In, SCFloatArrayRef Avg,
SCFloatArrayRef TopBand, SCFloatArrayRef BottomBand, SCFloatArrayRef StdDev, int Index, int Length, float
Multiplier, int MovAvgType)
{
    if(Length < 1)
    {
        return Avg;
    }

    MovingAverage_S( In, Avg, MovAvgType, Index, Length);

    StandardDeviation_S(Avg, StdDev, Index, Length);

    TopBand[Index] = Avg[Index]+ (StdDev[Index] * Multiplier);
    BottomBand[Index] = Avg[Index]- (StdDev[Index] * Multiplier);
    return Avg;
}

/*=====*/
void Summation(SCFloatArrayRef in, float& out, int start_indx, int length)
{

```

```

float sum=0;
out = 0;
if (start_indx < ( length - 1 ) )
{
    start_indx = length - 1;
}

if (start_indx >= in.GetArraySize())
    return;

for(int nn=0;nn<length;nn++)
{
    sum += in[start_indx-nn];
}
out = sum;
}

/*=====*/
float GetSummation(SCFloatArrayRef In,int Index,int Length)
{

float Sum = 0 ;

for (int i = 0; i < Length; i++)
    Sum = Sum + In[Index - i];

return Sum;
}

/*=====*/
SCFloatArrayRef AccumulationDistribution_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, int Index)
{

const float BIAS_DIVISION_FLOAT = static_cast<float>(10e-10);
SCFloatArrayRef phigh = BaseDataIn[SC_HIGH];
SCFloatArrayRef plow = BaseDataIn[SC_LOW];
SCFloatArrayRef pvolume = BaseDataIn[SC_VOLUME];
SCFloatArrayRef pclose = BaseDataIn[SC_LAST];

if (Index == 0)
{
    Out[Index] = ( (pclose[Index] - plow[Index]) - (phigh[Index] - pclose[Index]) ) / (phigh[Index] - plow[Index] +
BIAS_DIVISION_FLOAT) * pvolume[Index];
}
else
{
    Out[Index] = ( (pclose[Index] - plow[Index]) - (phigh[Index] - pclose[Index]) ) / (phigh[Index] - plow[Index] +
BIAS_DIVISION_FLOAT) * pvolume[Index] + Out[Index-1];
}

return Out;
}

/*=====*/

double GetMovingAverage(SCFloatArrayRef In, int start_indx, int length)
{
    double sum= 0;
    double out = 0;

    if (start_indx < length - 1 )
    {

```

```

        start_indx = length - 1;
    }
    if (start_indx >= In.GetArraySize())
        return 0;

    for(int nn=0;nn < length;nn++)
    {
        sum += static_cast<double>(In[start_indx-nn]);
    }
    out = sum / static_cast<double>(length);

    return out;
}

/*****
* Method          - Variance
* Description     - Computes the Variance
* shortcut to computing the variance  $V(x) = E[x^2] - E[x]^2$ 

```

```

*****/
double GetVariance(SCFloatArrayRef InData, int StartIndex, int Length)
{
    double mean1 = 0; //  $E[X^2]$ 
    double mean2 = 0; //  $E[X]^2$ 

    for(int Offset = 0; Offset < Length; Offset++)
    {
        // compute  $X^2$ 
        if (StartIndex-Offset < 0)
            break;

        double value = static_cast<double>(InData[StartIndex-Offset]);
        mean1 += value * value;
    }

    // compute  $E[X^2]$ 
    mean1 = mean1 / Length;

    // compute  $E[X]^2$ 
    mean2 = GetMovingAverage(InData,StartIndex, Length);
    mean2 *= mean2;

    //Compute  $V(x)$ 
    return mean1 - mean2;
}

```

```

/*=====*/
SCFloatArrayRef CumulativeSummation_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index)
{
    if(Index==0)
        Out[Index] = In[Index];
    else
        Out[Index] = Out[Index-1] + In[Index];

    return Out;
}

```

```

/*=====*/
SCFloatArrayRef ArmsEaseOfMovement_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, int VolumeDivisor, int Index)
{
    float PriceRange = 0;
    float BoxRatio = 0;
    float MidpointMove = 0;

```

```

float Volume = 0;

if(Index == 0)
    Out[Index] = 0;
else if (BaseDataIn[SC_VOLUME][Index] <= 0 || BaseDataIn[SC_HIGH][Index] == BaseDataIn[SC_LOW][Index])
    Out[Index] = 0;
else
{
    PriceRange = BaseDataIn[SC_HIGH][Index] - BaseDataIn[SC_LOW][Index];
    Volume = BaseDataIn[SC_VOLUME][Index];
    Volume /= VolumeDivisor;
    BoxRatio = Volume / PriceRange;
    MidpointMove = ((BaseDataIn[SC_HIGH][Index] + BaseDataIn[SC_LOW][Index]) / 2) -
        ((BaseDataIn[SC_HIGH][Index - 1] + BaseDataIn[SC_LOW][Index - 1]) / 2);
    Out[Index] = MidpointMove / BoxRatio;
}

return Out;
}

/*=====*/
SCFloatArrayRef ChaikinMoneyFlow_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, SCFloatArrayRef
InternalArray, int Index, int Length)
{
    SCFloatArrayRef MoneyFlowVolume = InternalArray;
    SCFloatArrayRef CMF = Out;

    float MoneyFlowMultiplier = 0;

    if (BaseDataIn[SC_HIGH][Index] == BaseDataIn[SC_LOW][Index])
    {
        if (Index <= 0 || BaseDataIn[SC_LAST][Index] >= BaseDataIn[SC_LAST][Index - 1])
            MoneyFlowMultiplier = 1.0f;
        else
            MoneyFlowMultiplier = -1.0f;
    }
    else
    {
        MoneyFlowMultiplier =
            ( (BaseDataIn[SC_LAST][Index] - BaseDataIn[SC_LOW][Index]) - (BaseDataIn[SC_HIGH][Index] -
BaseDataIn[SC_LAST][Index]) )
            / (BaseDataIn[SC_HIGH][Index] - BaseDataIn[SC_LOW][Index]);
    }

    MoneyFlowVolume[Index] = MoneyFlowMultiplier * BaseDataIn[SC_VOLUME][Index];

    if (Index < Length-1)
    {
        CMF[Index] = 0;
        return CMF;
    }

    float SumMoneyFlowVolume = 0;
    float SumVolume = 0;

    for(int SumIndex = Index - Length + 1; SumIndex <= Index; SumIndex++)
    {
        SumVolume += BaseDataIn[SC_VOLUME][SumIndex];
        SumMoneyFlowVolume += MoneyFlowVolume[SumIndex];
    }

    if (SumVolume == 0)
        CMF[Index] = 0;
    else
        CMF[Index] = SumMoneyFlowVolume / SumVolume;
}

```

```

    return CMF;
}

/*=====*/
SCFloatArrayRef Summation_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    float sum = 0;

    if (Index < Length - 1 )
    {
        Out[Index] = 0;
    }
    else
    {
        for(int pos = Index-Length+1; pos <= Index; pos++)
        {
            sum += In[pos];
        }
    }
    Out[Index] = sum;

    return Out;
}

/*=====*/
SCFloatArrayRef Dispersion_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    Out[Index] = GetDispersion(In, Index, Length);
    return Out;
}

/*=====*/
float GetDispersion(SCFloatArrayRef In, int Index, int Length)
{
    if(Index < Length - 1)
        return 0;

    float accum = 0;
    float mean = 0;
    for (int pos = Index - Length + 1; pos <= Index; pos ++ )
    {
        accum += In[pos];
    }
    mean = accum / Length;

    accum = 0;
    float curr = 0;
    for (int pos = Index - Length + 1; pos <= Index; pos ++ )
    {
        curr = ((mean - In[pos]) * (mean - In[pos])) / Length;
        accum += curr;
    }

    return accum;
}

/*=====*/
SCFloatArrayRef EnvelopePercent_S(SCFloatArrayRef In, SCFloatArrayRef Out1, SCFloatArrayRef Out2, float Percent,
int Index)
{
    Out1[Index] = In[Index] + In[Index] * Percent;
    Out2[Index] = In[Index] - In[Index] * Percent;
}

```

```

    return Out1;
}

/*=====*/
SCFloatArrayRef EnvelopeFixed_S(SCFloatArrayRef In, SCFloatArrayRef Out1, SCFloatArrayRef Out2, float
FixedValue, int Index)
{
    Out1[Index] = In[Index] + FixedValue;
    Out2[Index] = In[Index] - FixedValue;

    return Out1;
}

/*=====*/
SCFloatArrayRef VerticalHorizontalFilter_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    float Sum = 0;
    float HighestValue;
    float LowestValue;

    for (int pos = Index - Length + 1; pos <= Index; pos++)
    {
        Sum += fabs((In[pos] - In[pos - 1]));
    }

    LowestValue = GetLowest(In, Index, Length);
    HighestValue = GetHighest(In, Index, Length);

    Out[Index] = (HighestValue - LowestValue) / Sum;

    return Out;
}

/*=====*/
SCFloatArrayRef RWI_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef HighRWI, SCFloatArrayRef LowRWI,
SCFloatArrayRef TrueRangeArray, SCFloatArrayRef LookBackLowArray, SCFloatArrayRef LookBackHighArray, int
Index, int Length)
{
    TrueRange_S(BaseDataIn, TrueRangeArray, Index);

    HighRWI[Index] = 0;
    LowRWI[Index] = 0;

    if (Index >= Length)
    {
        float ATRSum = 0;

        for (int Iteration = 1; Iteration <= Length; Iteration++)
        {
            ATRSum += TrueRangeArray[Index-Iteration];

            float ATR = ATRSum / Iteration;

            float HighRWIValue = 0;
            float LowRWIValue = 0;

            float Denom = ATR * sqrt(static_cast<float>(Iteration));
            if (Denom != 0)
            {
                HighRWIValue = (BaseDataIn[SC_HIGH][Index] - BaseDataIn[SC_LOW][Index-Iteration]) / Denom;
                LowRWIValue = (BaseDataIn[SC_HIGH][Index-Iteration] - BaseDataIn[SC_LOW][Index]) / Denom;
            }
        }
    }
}

```



```

        if (HighRWIValue > HighRWI[Index])
            HighRWI[Index] = HighRWIValue;

        if (LowRWIValue > LowRWI[Index])
            LowRWI[Index] = LowRWIValue;
    }
}

return HighRWI;
}

```

```

/* ===== */

```

```

SCFloatArrayRef UltimateOscillator_S(SCBaseDataRef BaseDataIn,

```

```

    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, const int Length1, const int Length2, const int Length3)

```

```

{
/*

```

-Column E calculates the True High, which is the greater of Today's High and Yesterday's Close.

- Column F calculates the True Low, which is the lesser of Today's Low and Yesterday's Close.

- Column G subtracts the True Low (column F) from the True High (column E) to get the Range (also called Total Activity).

- Column H is a 7-day summation of the Range (column G).

- Column I subtracts the True Low (column F) from the Close, to get the Buying Units (also called Buying Pressure).

- Column J is a 7-day summation of the Buying Units (column I).

- Column K divides Buying Units (column J) by Range (column H).

- Column L is a 14-day summation of the Range (column G).

- Column M is a 14-day summation of the Buying Units (column I).

- Column N divides Buying Units (column M) by Range (column L).

- Column O is a 28-day summation of the Range (column G).

- Column P is a 28-day summation of the Buying Units (column I).

- Column Q divides Buying Units (column P) by Range (column O)

Column R is the Ultimate Oscillator, which is four times the 7-day division (column K) + 2 times the 14-day division (column N) plus the 28-day division (column Q), divided by 7 and multiplied by 100. Formula:  $=((4*K40)+(2*N40)+Q40)/7*100$  \*/

```

    CalcE[Index] = GetTrueHigh(BaseDataIn, Index);

```

```

    CalcF[Index] = GetTrueLow(BaseDataIn, Index);

```

```

    CalcG[Index] = CalcE[Index] - CalcF[Index];

```

```

    Summation_S(CalcG, CalcH, Index, Length1);

```

```

    CalcI[Index] = BaseDataIn[SC_LAST][Index] - CalcF[Index];

```

```

    Summation_S(CalcI, CalcJ, Index, Length1);

```

```

    if (CalcH[Index] != 0)
        CalcK[Index] = CalcJ[Index] / CalcH[Index];
    else

```

```

    CalcK[Index] = 0;

    Summation_S(CalcG, CalcL, Index, Length2);

    Summation_S(CalcI, CalcM, Index, Length2);

    if (CalcL[Index] != 0)
        CalcN[Index] = CalcM[Index] / CalcL[Index];
    else
        CalcN[Index] = 0;

    Summation_S(CalcG, CalcO, Index, Length3);

    Summation_S(CalcI, CalcP, Index, Length3);

    if (CalcO[Index] != 0)
        CalcQ[Index] = CalcP[Index] / CalcO[Index];
    else
        CalcQ[Index] = 0;

    Out[Index] = ((4 * CalcK[Index]) + (2 * CalcN[Index]) + CalcQ[Index]) / 7 * 100;

    return Out;
}

/*=====*/
SCFloatArrayRef WilliamsAD_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, int Index)
{
    if(Index > 0)
    {
        if (BaseDataIn[SC_LAST][Index] > BaseDataIn[SC_LAST][Index - 1])
            Out[Index] = Out[Index - 1] + (BaseDataIn[SC_LAST][Index] - min(BaseDataIn[SC_LOW][Index],
BaseDataIn[SC_LAST][Index - 1]));

        else if (BaseDataIn[SC_LAST][Index] == BaseDataIn[SC_LAST][Index - 1])
            Out[Index] = Out[Index - 1];

        else if (BaseDataIn[SC_LAST][Index] < BaseDataIn[SC_LAST][Index - 1])
            Out[Index] = Out[Index - 1] +
            ( BaseDataIn[SC_LAST][Index] - max(BaseDataIn[SC_HIGH][Index], BaseDataIn[SC_LAST][Index - 1]) );
    }
    else
        Out[Index] = 0;

    return Out;
}

/*=====*/
SCFloatArrayRef WilliamsR_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, int Index, int Length)
{
    return WilliamsR2_S(BaseDataIn[SC_HIGH], BaseDataIn[SC_LOW], BaseDataIn[SC_LAST], Out, Index, Length);
}

/*=====*/
SCFloatArrayRef WilliamsR2_S(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow, SCFloatArrayRef
InputDataLast, SCFloatArrayRef Out, int Index, int Length)
{
    if(Index < Length)
        Out[Index]=0;
    else
    {
        float High = GetHighest(InputDataHigh,Index, Length);

```

```

        float Low = GetLowest(InputDataLow,Index,Length);
        Out[Index] = 100 * ( High-InputDataLast[Index]) / (High - Low);
    }

    return Out;
}

/*=====*/
float GetArrayValueAtNthOccurrence(SCFloatArrayRef TrueFalseIn, SCFloatArrayRef ValueArrayIn, int Index, int
NthOccurrence )
{
    int Occurrence = 0;
    int CurrentIndex = Index;

    while(Occurrence < NthOccurrence)
    {
        if(TrueFalseIn[CurrentIndex] != 0)
            Occurrence++;

        CurrentIndex--;

        if(CurrentIndex < 0)
            break;
    }

    if((CurrentIndex < 0) && (Occurrence < NthOccurrence))
        return 0;

    return ValueArrayIn[CurrentIndex+1];
}

/*=====*/
SCFloatArrayRef Parabolic_S(s_Parabolic& ParabolicData)
{
    SCBaseDataRef BaseDataIn = ParabolicData.m_BaseDataIn;
    int BaseDataIndex = ParabolicData.m_Index;
    SCSubgraphRef Out = ParabolicData.m_Out;
    SCDateTimeArrayRef BaseDateTimeln = ParabolicData.m_BaseDateTimeln;
    float InStartAccelFactor = ParabolicData.m_InStartAccelFactor;
    float InAccelIncrement = ParabolicData.m_InAccelIncrement;
    float InMaxAccelFactor = ParabolicData.m_InMaxAccelFactor;
    unsigned int InAdjustForGap = ParabolicData.m_InAdjustForGap;

    SCFloatArrayRef AccelerationFactor = Out.Arrays[0];
    SCFloatArrayRef ExtremeHighOrLowDuringTrend = Out.Arrays[1];
    SCFloatArrayRef InitialCalculationsLow = Out.Arrays[2];
    SCFloatArrayRef InitialCalculationsHigh = Out.Arrays[3];
    SCFloatArrayRef CurrentParabolicDirection = Out.Arrays[4];

    const int PARABOLIC_LONG = 2;
    const int PARABOLIC_SHORT = 1;

    if (BaseDataIndex < 1)//Reset
    {
        CurrentParabolicDirection[0] = 0;
        AccelerationFactor[0] = 0;
        ExtremeHighOrLowDuringTrend[0] = 0;

        InitialCalculationsLow[0] = ParabolicData.BaseDataLow(0);
        InitialCalculationsHigh[0] = ParabolicData.BaseDataHigh(0);

        return Out;
    }

    //Carry forward persistent values

```

```

ExtremeHighOrLowDuringTrend[BaseDataIndex] = ExtremeHighOrLowDuringTrend[BaseDataIndex - 1];
AccelerationFactor[BaseDataIndex] = AccelerationFactor[BaseDataIndex - 1];

InitialCalculationsLow[BaseDataIndex] = InitialCalculationsLow[BaseDataIndex - 1];
InitialCalculationsHigh[BaseDataIndex] = InitialCalculationsHigh[BaseDataIndex - 1];

if (CurrentParabolicDirection[BaseDataIndex - 1] == 0) // The CurrentParabolicDirection is unknown.
{
    AccelerationFactor[BaseDataIndex] = InStartAccelFactor;

    if (InitialCalculationsLow[BaseDataIndex] > ParabolicData.BaseDataLow(BaseDataIndex))
        InitialCalculationsLow[BaseDataIndex] = ParabolicData.BaseDataLow(BaseDataIndex);

    if (InitialCalculationsHigh[BaseDataIndex] < ParabolicData.BaseDataHigh(BaseDataIndex))
        InitialCalculationsHigh[BaseDataIndex] = ParabolicData.BaseDataHigh(BaseDataIndex);

    if (ParabolicData.BaseDataHigh(BaseDataIndex) > ParabolicData.BaseDataHigh(BaseDataIndex-1) &&
        ParabolicData.BaseDataLow(BaseDataIndex) > ParabolicData.BaseDataLow(BaseDataIndex-1))
    {
        ExtremeHighOrLowDuringTrend[BaseDataIndex] = ParabolicData.BaseDataHigh(BaseDataIndex);
        Out[BaseDataIndex] = InitialCalculationsLow[BaseDataIndex];
        CurrentParabolicDirection[BaseDataIndex] = PARABOLIC_LONG;
        //LastPriceAtReversalIndex[BaseDataIndex] = BaseDataIn[SC_LAST][BaseDataIndex];

        return Out;
    }

    if (ParabolicData.BaseDataHigh(BaseDataIndex) < ParabolicData.BaseDataHigh(BaseDataIndex-1) &&
        ParabolicData.BaseDataLow(BaseDataIndex) < ParabolicData.BaseDataLow(BaseDataIndex-1))
    {
        ExtremeHighOrLowDuringTrend[BaseDataIndex] = ParabolicData.BaseDataLow(BaseDataIndex);
        Out[BaseDataIndex] = InitialCalculationsHigh[BaseDataIndex];
        //LastPriceAtReversalIndex[BaseDataIndex] = BaseDataIn[SC_LAST][BaseDataIndex];
        CurrentParabolicDirection[BaseDataIndex] = PARABOLIC_SHORT;

        return Out;
    }
}
else
{
    if (CurrentParabolicDirection[BaseDataIndex - 1] == PARABOLIC_LONG)
    {
        if (ExtremeHighOrLowDuringTrend[BaseDataIndex - 1] < ParabolicData.BaseDataHigh(BaseDataIndex-1))
        {
            ExtremeHighOrLowDuringTrend[BaseDataIndex] = ParabolicData.BaseDataHigh(BaseDataIndex-1);

            AccelerationFactor[BaseDataIndex] = AccelerationFactor[BaseDataIndex - 1] + InAccelIncrement;
        }

        CurrentParabolicDirection[BaseDataIndex] = CurrentParabolicDirection[BaseDataIndex - 1];
    }
    else if (CurrentParabolicDirection[BaseDataIndex - 1] == PARABOLIC_SHORT)
    {
        if (ExtremeHighOrLowDuringTrend[BaseDataIndex - 1] > ParabolicData.BaseDataLow(BaseDataIndex-1))
        {
            ExtremeHighOrLowDuringTrend[BaseDataIndex] = ParabolicData.BaseDataLow(BaseDataIndex-1);

            AccelerationFactor[BaseDataIndex] = AccelerationFactor[BaseDataIndex] + InAccelIncrement;
        }

        CurrentParabolicDirection[BaseDataIndex] = CurrentParabolicDirection[BaseDataIndex - 1];
    }
}

```

```

    if (AccelerationFactor[BaseDataIndex] > InMaxAccelFactor)
        AccelerationFactor[BaseDataIndex] = InMaxAccelFactor;

    float CalcResult= AccelerationFactor[BaseDataIndex] * (ExtremeHighOrLowDuringTrend[BaseDataIndex] -
    Out[BaseDataIndex - 1]);

    Out[BaseDataIndex] = Out[BaseDataIndex - 1] + CalcResult;

    float Gap = 0.0;
    float PreviousTrueHigh = 0;
    if ((BaseDataIndex < 2) || (ParabolicData.BaseDataHigh(BaseDataIndex-1) >=
    ParabolicData.BaseDataLow(BaseDataIndex-2)))
        PreviousTrueHigh = ParabolicData.BaseDataHigh(BaseDataIndex-1);
    else
        PreviousTrueHigh = ParabolicData.BaseDataLow(BaseDataIndex-2);

    if (InAdjustForGap
        && BaseDataIn[SC_OPEN][BaseDataIndex] > PreviousTrueHigh
        && BaseDateTimeln.DateAt(BaseDataIndex) != BaseDateTimeln.DateAt(BaseDataIndex - 1)
    )
    {
        Gap = (BaseDataIn[SC_OPEN][BaseDataIndex] - BaseDataIn[SC_LAST][BaseDataIndex - 1]);
        Out[BaseDataIndex] = Out[BaseDataIndex - 1] + Gap;
        ExtremeHighOrLowDuringTrend[BaseDataIndex] = ExtremeHighOrLowDuringTrend[BaseDataIndex - 1] + Gap;
    }

    float PreviousTrueLow = 0;
    if ((BaseDataIndex < 2) || (ParabolicData.BaseDataLow(BaseDataIndex-1) <=
    ParabolicData.BaseDataHigh(BaseDataIndex-2)))
        PreviousTrueLow = ParabolicData.BaseDataLow(BaseDataIndex-1);
    else
        PreviousTrueLow = ParabolicData.BaseDataHigh(BaseDataIndex-2);

    if (InAdjustForGap
        && BaseDataIn[SC_OPEN][BaseDataIndex] < PreviousTrueLow
        && BaseDateTimeln.DateAt(BaseDataIndex) != BaseDateTimeln.DateAt(BaseDataIndex - 1)
    )
    {
        Gap = BaseDataIn[SC_LAST][BaseDataIndex - 1] - BaseDataIn[SC_OPEN][BaseDataIndex];
        Out[BaseDataIndex] = Out[BaseDataIndex - 1] - Gap;

        ExtremeHighOrLowDuringTrend[BaseDataIndex] = ExtremeHighOrLowDuringTrend[BaseDataIndex - 1] - Gap;
    }

    if (Gap == 0.0f && CurrentParabolicDirection[BaseDataIndex] == PARABOLIC_LONG
        && (Out[BaseDataIndex] > ParabolicData.BaseDataLow(BaseDataIndex-1) || Out[BaseDataIndex] >
    ParabolicData.BaseDataLow(BaseDataIndex-2))
    )
        Out[BaseDataIndex] = min(ParabolicData.BaseDataLow(BaseDataIndex-1),
    ParabolicData.BaseDataLow(BaseDataIndex-2));

    else if (Gap == 0.0f && CurrentParabolicDirection[BaseDataIndex] == PARABOLIC_SHORT
        && (Out[BaseDataIndex] < ParabolicData.BaseDataHigh(BaseDataIndex-1) || Out[BaseDataIndex] <
    ParabolicData.BaseDataHigh(BaseDataIndex-2))
    )
        Out[BaseDataIndex] = max(ParabolicData.BaseDataHigh(BaseDataIndex-1),
    ParabolicData.BaseDataHigh(BaseDataIndex-2));

    //Check to see if we need to reverse

    if (CurrentParabolicDirection[BaseDataIndex] == PARABOLIC_LONG &&
        Out[BaseDataIndex] >= ParabolicData.BaseDataLow(BaseDataIndex))
    {

```

```

CurrentParabolicDirection[BaseDataIndex] = PARABOLIC_SHORT;

Out[BaseDataIndex] = ExtremeHighOrLowDuringTrend[BaseDataIndex];

AccelerationFactor[BaseDataIndex] = InStartAccelFactor;
ExtremeHighOrLowDuringTrend[BaseDataIndex] = ParabolicData.BaseDataLow(BaseDataIndex);

return Out;
}
else if (CurrentParabolicDirection[BaseDataIndex] == PARABOLIC_SHORT &&
Out[BaseDataIndex] <= ParabolicData.BaseDataHigh(BaseDataIndex))
{
CurrentParabolicDirection[BaseDataIndex] = PARABOLIC_LONG;

Out[BaseDataIndex] = ExtremeHighOrLowDuringTrend[BaseDataIndex];

AccelerationFactor[BaseDataIndex] = InStartAccelFactor;
ExtremeHighOrLowDuringTrend[BaseDataIndex] = ParabolicData.BaseDataHigh(BaseDataIndex);

return Out;
}
}

return Out;
}
/*=====*/
int GetIslandReversal_S(SCBaseDataRef BaseDataIn, int Index)
{
if (Index == 0)
return 0;

float Range = (BaseDataIn[SC_HIGH][Index] - BaseDataIn[SC_LOW][Index]) ;

if (
BaseDataIn[SC_HIGH][Index] < BaseDataIn[SC_LOW][Index - 1]
&&
(BaseDataIn[SC_LAST][Index] > (BaseDataIn[SC_LOW][Index] +
(Range * .70f)))
)
return 1;

if (
BaseDataIn[SC_LOW][Index] > BaseDataIn[SC_HIGH][Index - 1]
&&
(BaseDataIn[SC_LAST][Index] < (BaseDataIn[SC_HIGH][Index] -
(Range * .70f)))
)
return -1;

return 0;
}

/*=====*/
SCFloatArrayRef Oscillator_S(SCFloatArrayRef In1, SCFloatArrayRef In2, SCFloatArrayRef Out, int Index)
{
Out[Index] = In1[Index] - In2[Index];

return Out;
}

/*=====*/
float GetTrueHigh(SCBaseDataRef BaseDataIn, int Index)

```

```

{
    if (Index == 0 || BaseDataIn[SC_HIGH][Index] > BaseDataIn[SC_LAST][Index - 1])
        return (BaseDataIn[SC_HIGH][Index]);
    else
        return (BaseDataIn[SC_LAST][Index - 1]);
}

/*=====*/
float GetTrueLow(SCBaseDataRef BaseDataIn, int Index)
{
    if (Index == 0 || BaseDataIn[SC_LOW][Index] < BaseDataIn[SC_LAST][Index - 1])
        return (BaseDataIn[SC_LOW][Index]);
    else
        return (BaseDataIn[SC_LAST][Index - 1]);
}

/*=====*/
float GetTrueRange(SCBaseDataRef BaseDataIn, int Index)
{
    return (GetTrueHigh(BaseDataIn, Index) - GetTrueLow(BaseDataIn, Index));
}

/*=====*/
float GetRange(SCBaseDataRef BaseDataIn, int Index)
{
    return BaseDataIn[SC_HIGH][Index] - BaseDataIn[SC_LOW][Index];
}

/*=====*/

float GetCorrelationCoefficient(SCFloatArrayRef In1, SCFloatArrayRef In2, int Index, int Length)
{
    float result;
    int i;
    float xmean;
    float ymean;
    float s;
    float xv;
    float yv;
    float t1;
    float t2;

    xv = 0;
    yv = 0;
    if (Length <= 1)
    {
        result = 0;
        return result;
    }

    //
    // Mean
    //
    xmean = 0;
    ymean = 0;
    for(i = 0; i < Length; i++)
    {
        xmean += In1[Index-i];
        ymean += In2[Index-i];
    }
    xmean = xmean/Length;
    ymean = ymean/Length;

    //

```

```

// numerator and denominator
//
s = 0;
xv = 0;
yv = 0;

for(i = 0; i <= Length-1; i++)
{
    t1 = ln1[Index-i]-xmean;
    t2 = ln2[Index-i]-ymean;
    xv = xv+(t1*t1);
    yv = yv+(t2*t2);
    s = s+t1*t2;
}
if (xv==0||yv==0 )
{
    result = 0;
}
else
{
    result = s/(sqrt(xv)*sqrt(yv));
}

return result;
}

/*=====*/
int NumberOfBarsSinceHighestValue(SCFloatArrayRef ln, int Index, int Length)
{
    float max = ln[Index];
    int maxIndex = Index;
    for(int i = Index; i > Index - Length && i >= 0; i--)
    {
        if(ln[i] > max)
        {
            max = ln[i];
            maxIndex = i;
        }
    }
    return Index - maxIndex;
}

/*=====*/
int NumberOfBarsSinceLowestValue(SCFloatArrayRef ln, int Index, int Length)
{
    float min = ln[Index];
    int minIndex = Index;

    for(int i = Index; i > Index - Length && i >= 0; i--)
    {
        if(ln[i] < min)
        {
            min = ln[i];
            minIndex = i;
        }
    }
    return Index - minIndex;
}

/*=====*/
SCFloatArrayRef PriceVolumeTrend_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, int Index)
{
    if(Index==0)
    {
        Out[Index] = 0;
    }
}

```



```

    return Out;
}

Out[Index] = ((BaseDataIn[SC_LAST][Index] - BaseDataIn[SC_LAST][Index - 1])
    / BaseDataIn[SC_LAST][Index - 1])
    * BaseDataIn[SC_VOLUME][Index] + Out[Index-1];

return Out;
}

/*=====*/
SCFloatArrayRef Momentum_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    Out[Index] = (In[Index]/In[Index-Length])*100;
    return Out;
}

/*=====*/
SCFloatArrayRef TRIX_S(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef InternalEma_1, SCFloatArrayRef
InternalEma_2, SCFloatArrayRef InternalEma_3, int Index, int Length)
{
    const double BIAS_DIVISION_DOUBLE = static_cast<double>(10e-20);

    ExponentialMovingAverage_S(In, InternalEma_1, Index, Length);
    ExponentialMovingAverage_S(InternalEma_1, InternalEma_2, Index, Length);
    ExponentialMovingAverage_S(InternalEma_2, InternalEma_3, Index, Length);

    Out[Index] = static_cast<float>(100 * (InternalEma_3[Index] - InternalEma_3[Index-1]) / (InternalEma_3[Index-1] +
BIAS_DIVISION_DOUBLE));

    return Out;
}

/*=====*/
SCFloatArrayRef Demarker_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, SCFloatArrayRef DemMax,
SCFloatArrayRef DemMin, SCFloatArrayRef SmaDemMax, SCFloatArrayRef SmaDemMin, int Index, int Length)
{
    if(Index > 0)
    {
        float high = BaseDataIn[SC_HIGH][Index];
        float highOld = BaseDataIn[SC_HIGH][Index-1];

        if (high > highOld)
        {
            DemMax[Index] = high - highOld;
        }
        else
        {
            DemMax[Index] = 0.0f;
        }

        float low = BaseDataIn[SC_LOW][Index];
        float lowOld = BaseDataIn[SC_LOW][Index-1];

        if (low < lowOld)
        {
            DemMin[Index] = lowOld - low;
        }
        else
        {
            DemMin[Index] = 0.0f;
        }
    }
    else
    {

```

```

    DemMax[Index] = 0.0f;
    DemMin[Index] = 0.0f;
}

SimpleMovAvg_S(DemMax, SmaDemMax, Index, Length);
SimpleMovAvg_S(DemMin, SmaDemMin, Index, Length);

float summ = SmaDemMax[Index] + SmaDemMin[Index];
if (summ != 0.0f)
{
    Out[Index] = SmaDemMax[Index] / summ;
}
else
{
    Out[Index] = Out[Index-1];
}

return Out;
}

/*=====*/
SCFloatArrayRef AroonIndicator_S(SCFloatArrayRef FloatArrayInHigh, SCFloatArrayRef FloatArrayInLow,
SCFloatArrayRef OutUp, SCFloatArrayRef OutDown, SCFloatArrayRef OutOscillator, int Index, int Length)
{
    int BarsSinceHighest = 0;
    float Highest = FloatArrayInHigh[Index];

    int BarsSinceLowest = 0;
    float Lowest = FloatArrayInLow[Index];

    for (int i = 1; i <= Length ; i++)
    {
        if (FloatArrayInHigh[Index - i] > Highest)
        {
            Highest = FloatArrayInHigh[Index - i];
            BarsSinceHighest = i;
        }

        if (FloatArrayInLow[Index - i] < Lowest)
        {
            Lowest = FloatArrayInLow[Index - i];
            BarsSinceLowest = i;
        }
    }

    OutUp[Index] = ((Length - static_cast<float>(BarsSinceHighest)) / Length) * 100.0f;
    OutDown[Index] = ((Length - static_cast<float>(BarsSinceLowest)) / Length) * 100.0f;

    OutOscillator [Index] =OutUp[Index] -OutDown[Index];

    return OutUp;
}

/*=====*/
int IsSwingHighAllowEqual_S(SCStudyInterfaceRef sc, int AllowEqual, int Index, int Length)
{
    for(int i = 1; i <= Length; i++)
    {
        if (AllowEqual)
        {
            if (sc.FormattedEvaluate(sc.BaseData [SC_HIGH][Index] , sc.BasedOnGraphValueFormat, LESS_OPERATOR,
sc.BaseData [SC_HIGH][Index-i], sc.BasedOnGraphValueFormat)
                ||
                sc.FormattedEvaluate(sc.BaseData [SC_HIGH][Index] , sc.BasedOnGraphValueFormat, LESS_OPERATOR,

```

```

sc.BaseData [SC_HIGH][Index+i], sc.BasedOnGraphValueFormat)
    )
    return 0;

```

```

    }
    else
    {
        if (sc.FormattedEvaluate(sc.BaseData [SC_HIGH][Index] , sc.BasedOnGraphValueFormat,
LESS_EQUAL_OPERATOR, sc.BaseData [SC_HIGH][Index-i], sc.BasedOnGraphValueFormat)
            ||
            sc.FormattedEvaluate(sc.BaseData [SC_HIGH][Index] , sc.BasedOnGraphValueFormat,
LESS_EQUAL_OPERATOR, sc.BaseData [SC_HIGH][Index+i], sc.BasedOnGraphValueFormat)
        )
            return 0;
    }
}

```

```

    return 1;
}

```

```

/*=====*/

```

```

int IsSwingLowAllowEqual_S(SCStudyInterfaceRef sc, int AllowEqual, int Index, int Length)
{

```

```

    for(int i = 1; i <= Length; i++)
    {

```

```

        if (AllowEqual)
        {
            if (sc.FormattedEvaluate(sc.BaseData [SC_LOW][Index] , sc.BasedOnGraphValueFormat,
GREATER_OPERATOR, sc.BaseData [SC_LOW][Index-i], sc.BasedOnGraphValueFormat)
                ||
                sc.FormattedEvaluate(sc.BaseData [SC_LOW][Index] , sc.BasedOnGraphValueFormat,
GREATER_OPERATOR, sc.BaseData [SC_LOW][Index+i], sc.BasedOnGraphValueFormat)
            )
                return 0;

```

```

        }
        else
        {
            if (sc.FormattedEvaluate(sc.BaseData [SC_LOW][Index] , sc.BasedOnGraphValueFormat,
GREATER_EQUAL_OPERATOR, sc.BaseData [SC_LOW][Index-i], sc.BasedOnGraphValueFormat)
                ||
                sc.FormattedEvaluate(sc.BaseData [SC_LOW][Index] , sc.BasedOnGraphValueFormat,
GREATER_EQUAL_OPERATOR, sc.BaseData [SC_LOW][Index+i], sc.BasedOnGraphValueFormat)
            )
                return 0;

```

```

        }
    }
    return 1;
}

```

```

/*=====*/

```

```

SCFloatArrayRef AwesomeOscillator_S(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef TempMA1,
SCFloatArrayRef TempMA2, int Index, int Length1, int Length2)
{

```

```

    SimpleMovAvg_S(In, TempMA1, Index, Length1);
    SimpleMovAvg_S(In, TempMA2, Index, Length2);

```

```

    Out[Index] = TempMA2[Index] - TempMA1[Index];

    return Out;
}

/*=====*/
int CalculatePivotPoints
( float PriorOpen
, float PriorHigh
, float PriorLow
, float PriorClose
, float CurrentOpen
, float& PivotPoint
, float& PivotPointHigh
, float& PivotPointLow
, float& R_5
, float& R1, float& R1_5
, float& R2, float& R2_5
, float& R3
, float& S_5
, float& S1, float& S1_5
, float& S2, float& S2_5
, float& S3
, float& R3_5
, float& S3_5
, float& R4
, float& R4_5
, float& S4
, float& S4_5
, float& R5
, float& S5
, float& R6
, float& S6
, float& R7
, float& S7
, float& R8
, float& S8
, float& R9
, float& S9
, float& R10
, float& S10
, int FormulaType
)
{
    if (FormulaType == 0)
    {
        PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;
        R1 = (PivotPoint * 2) - PriorLow;
        R_5 = (PivotPoint + R1) / 2;
        R2 = PivotPoint + (PriorHigh - PriorLow);
        R1_5 = (R1 + R2) / 2;
        S1 = (2 * PivotPoint) - PriorHigh;
        S_5 = (PivotPoint + S1) / 2;
        S2 = PivotPoint - (PriorHigh - PriorLow);
        S1_5 = (S1 + S2) / 2;
        R3 = 2 * PivotPoint + (PriorHigh - 2*PriorLow);
        R2_5 = (R2 + R3) / 2;
        S3 = 2 * PivotPoint - (2 * PriorHigh - PriorLow);
        S2_5 = (S2 + S3) / 2;
        R4 = 3 * PivotPoint + (PriorHigh - 3 * PriorLow);
        S4 = 3 * PivotPoint - (3 * PriorHigh - PriorLow);
        R3_5 = (R3 + R4) / 2;
        S3_5 = (S3 + S4) / 2;
        R5 = 4 * PivotPoint + (PriorHigh - 4 * PriorLow);
        S5 = 4 * PivotPoint - (4 * PriorHigh - PriorLow);
    }
}

```

```

R6 = 5 * PivotPoint + (PriorHigh - 5 * PriorLow);
S6 = 5 * PivotPoint - (5 * PriorHigh - PriorLow);
R7 = 6 * PivotPoint + (PriorHigh - 6 * PriorLow);
S7 = 6 * PivotPoint - (6 * PriorHigh - PriorLow);
R8 = 7 * PivotPoint + (PriorHigh - 7 * PriorLow);
S8 = 7 * PivotPoint - (7 * PriorHigh - PriorLow);
R9 = 8 * PivotPoint + (PriorHigh - 8 * PriorLow);
S9 = 8 * PivotPoint - (8 * PriorHigh - PriorLow);
R10 = 9 * PivotPoint + (PriorHigh - 9 * PriorLow);
S10 = 9 * PivotPoint - (9 * PriorHigh - PriorLow);

/*

R3.5 = (R3 + R4)/2
S3.5 = (S3 + S4)/2
R5 = 4 * PivotPoint + (Yesterday's High - (4 * Yesterday's Low))
S5 = (4 * PivotPoint) - ((4 * Yesterday's High) - Yesterday's Low)
R4.5 = (R4 + R5)/2
S4.5 = (S4 + S5)/2
*/
}

else if (FormulaType == 1)
{
    float YesterdaysRange;

    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;
    R1 = (PivotPoint * 2) - PriorLow;
    R_5 = (PivotPoint + R1) / 2;
    R2 = PivotPoint + (PriorHigh - PriorLow);
    R1_5 = (R1 + R2) / 2;
    S1 = (2 * PivotPoint) - PriorHigh;
    S_5 = (PivotPoint + S1) / 2;
    S2 = PivotPoint - (PriorHigh - PriorLow);
    S1_5 = (S1 + S2) / 2;
    YesterdaysRange = (PriorHigh - PriorLow);

    // This is R3 Pattern Trapper
    R3 = PriorHigh + YesterdaysRange; // 2 * High - Low
    R2_5 = (R2 + R3)/2;

    // This is S3 Pattern Trapper
    S3 = 2*PivotPoint - PriorHigh - YesterdaysRange; // 2 * PivotPoint - (2 * High + Low)
    S2_5 = (S2 + S3) / 2;

    R4 = PivotPoint + 3 * (PriorHigh - PriorLow);
    S4 = PivotPoint - 3 * (PriorHigh - PriorLow);
}

else if (FormulaType == 2)
{
    PivotPoint = (CurrentOpen + PriorHigh + PriorLow + PriorClose) / 4;
    R1 = (PivotPoint * 2) - PriorLow;
    R_5 = (PivotPoint + R1) / 2;
    R2 = PivotPoint + (PriorHigh - PriorLow);
    R1_5 = (R1 + R2) / 2;
    S1 = (2 * PivotPoint) - PriorHigh;
    S_5 = (PivotPoint + S1) / 2;
    S2 = PivotPoint - (PriorHigh - PriorLow);
    S1_5 = (S1 + S2) / 2;
    R3 = 2*PivotPoint + (PriorHigh - 2*PriorLow); // R3 std
    R2_5 = (R2 + R3) / 2;
    S3 = 2*PivotPoint - (2*PriorHigh - PriorLow); // S3 std
    S2_5 = (S2 + S3) / 2;
    R4 = 3 * PivotPoint + (PriorHigh - 3 * PriorLow);
    S4 = 3 * PivotPoint - (3 * PriorHigh - PriorLow);
}

```

```

}
else if (FormulaType == 3)
{
    PivotPoint = (PriorHigh + PriorLow + CurrentOpen+CurrentOpen) / 4;
    R1 = (PivotPoint * 2) - PriorLow;
    R_5 = (PivotPoint + R1) / 2;
    R2 = PivotPoint + (PriorHigh - PriorLow);
    R1_5 = (R1 + R2) / 2;
    S1 = (2 * PivotPoint) - PriorHigh;
    S_5 = (PivotPoint + S1) / 2;
    S2 = PivotPoint - (PriorHigh - PriorLow);
    S1_5 = (S1 + S2) / 2;
    R3 = 2 * PivotPoint + (PriorHigh - 2*PriorLow); // R3 std
    R2_5 = (R2 + R3) / 2;
    S3 = 2 * PivotPoint - (2 * PriorHigh - PriorLow); // S3 std
    S2_5 = (S2 + S3) / 2;
    R4 = 3 * PivotPoint + (PriorHigh - 3 * PriorLow);
    S4 = 3 * PivotPoint - (3 * PriorHigh - PriorLow);
}
else if (FormulaType == 4)
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;
    R1 = (PivotPoint * 2) - PriorLow;
    R_5 = (PivotPoint + R1) / 2;
    R2 = PivotPoint + (PriorHigh - PriorLow);
    R1_5 = (R1 + R2) / 2;
    S1 = (2 * PivotPoint) - PriorHigh;
    S_5 = (PivotPoint + S1) / 2;
    S2 = PivotPoint - (PriorHigh - PriorLow);
    S1_5 = (S1 + S2) / 2;
    R3 = PivotPoint + 2 * (PriorHigh - PriorLow);
    R2_5 = (R2 + R3) / 2;
    S3 = PivotPoint - 2 * (PriorHigh - PriorLow);
    S2_5 = (S2 + S3) / 2;
    R4 = PivotPoint + 3 * (PriorHigh - PriorLow);
    S4 = PivotPoint - 3 * (PriorHigh - PriorLow);
}
else if (FormulaType == 5) // Camarilla Pivot Points
{
    float Range = PriorHigh - PriorLow;
    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;

    R_5 = PriorClose + Range * 1.1f/18;
    R1 = PriorClose + Range * 1.1f/12;
    R1_5 = PriorClose + Range * 1.1f/9;
    R2 = PriorClose + Range * 1.1f/6;
    R2_5 = PriorClose + Range * 1.1f/5;
    R3 = PriorClose + Range * 1.1f/4;
    R3_5 = PriorClose + Range * 1.1f/3;
    R4 = PriorClose + Range * 1.1f/2;
    R4_5 = PriorClose + Range * 1.1f / 1.33f;

    S_5 = PriorClose - Range * 1.1f/18;
    S1 = PriorClose - Range * 1.1f/12;
    S1_5 = PriorClose - Range * 1.1f/9;
    S2 = PriorClose - Range * 1.1f/6;
    S2_5 = PriorClose - Range * 1.1f/5;
    S3 = PriorClose - Range * 1.1f/4;
    S3_5 = PriorClose - Range * 1.1f/3;
    S4 = PriorClose - Range * 1.1f/2;
    S4_5 = PriorClose - Range * 1.1f / 1.33f;
    R5 = (PriorHigh/PriorLow) *PriorClose;
    S5 = PriorClose -(R5-PriorClose);
}
else if (FormulaType == 6) // Tom DeMark's Pivot Points

```

```

{
    float X = 0;
    if (PriorClose < PriorOpen)
        X = PriorHigh + PriorLow + PriorLow + PriorClose;
    else if (PriorClose > PriorOpen)
        X = PriorHigh + PriorHigh + PriorLow + PriorClose;
    else if (PriorClose == PriorOpen)
        X = PriorHigh + PriorLow + PriorClose + PriorClose;

    PivotPoint = X / 4;
    R1 = X / 2 - PriorLow;
    S1 = X / 2 - PriorHigh;
    R_5 = R2 = R1_5 = S_5 = S2 = S1_5 = R3 =
        R2_5 = S3 = S2_5 = R4 = S4 = 0;
}
else if (FormulaType == 7) // Frank Dileria Pivots
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;
    R1 = PivotPoint + (PriorHigh - PriorLow)/2;
    R2 = PivotPoint + (PriorHigh - PriorLow) * 0.618f;
    R3 = PivotPoint + (PriorHigh - PriorLow);
    S1 = PivotPoint - (PriorHigh - PriorLow)/2;
    S2 = PivotPoint - (PriorHigh - PriorLow) * 0.618f;
    S3 = PivotPoint - (PriorHigh - PriorLow);

    S2_5 = (S3 + S2)/2;
    S1_5 = (S2 + S1)/2;
    S_5 = (S1 + PivotPoint)/2;
    R_5 = (R1 + PivotPoint)/2;
    R1_5 = (R2 + R1)/2;
    R2_5 = (R3 + R2)/2;
}
else if (FormulaType == 8) // Shadow Trader. www.shadowtrader.net
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;
    R1 = (2 * PivotPoint) - PriorLow;
    R3 = 2*(PivotPoint - PriorLow) + PriorHigh;
    R4 = PriorHigh + 3*(PivotPoint - PriorLow);

    S1 = (2 * PivotPoint) - PriorHigh;
    S2 = PivotPoint - (R1 - S1);
    S3 = PriorLow - 2 * (PriorHigh - PivotPoint);
    S4 = PriorLow - 3 * (PriorHigh - PivotPoint);

    R2 = PivotPoint + (R1 - S1);

    S3_5 = (S3 + S4)/2;
    S2_5 = (S3 + S2)/2;
    S1_5 = (S2 + S1)/2;
    S_5 = (S1 + PivotPoint)/2;
    R_5 = (R1 + PivotPoint)/2;
    R1_5 = (R2 + R1)/2;
    R2_5 = (R3 + R2)/2;
    R3_5 = (R3 + R4)/2;
}
else if (FormulaType == 9)
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose)/3; //PP = H + L + C /3
    float PPSquareRoot = sqrt(sqrt(PivotPoint));
    R1 = PivotPoint + PPSquareRoot; //R1 = PP + SQRT(SQRT(PP))
    R2 = R1 + PPSquareRoot; //R2 = R1 + SQRT(SQRT(PP))
    R3 = R2 + PPSquareRoot; //R3 = R2 + SQRT(SQRT(PP))
    S1 = PivotPoint - PPSquareRoot; //S1 = PP - SQRT(SQRT(PP))
}

```

```

S2 = S1 - PPSquareRoot; //S2 = S1 - SQRT(SQRT(PP))
S3 = S2 - PPSquareRoot; //S3 = S2 - SQRT(SQRT(PP))

S2_5 = (S3 + S2)/2;
S1_5 = (S2 + S1)/2;
S_5 = (S1 + PivotPoint)/2;
R_5 = (R1 + PivotPoint)/2;
R1_5 = (R2 + R1)/2;
R2_5 = (R3 + R2)/2;

}
else if (FormulaType == 10)
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose)/3;
    R1 = 2 * PivotPoint - PriorLow;
    R2 = PivotPoint + PriorHigh - PriorLow;
    R3 = R1 + PriorHigh - PriorLow;
    R4 = R3 + (R2 - R1);

    S1 = 2 * PivotPoint - PriorHigh;
    S2 = PivotPoint - (PriorHigh - PriorLow);
    S3 = S1 - (PriorHigh - PriorLow);
    S4 = S3 - (S1 - S2);

    S3_5 = (S4 + S3)/2;
    S2_5 = (S3 + S2)/2;
    S1_5 = (S2 + S1)/2;
    S_5 = (S1 + PivotPoint)/2;
    R_5 = (R1 + PivotPoint)/2;
    R1_5 = (R2 + R1)/2;
    R2_5 = (R3 + R2)/2;
    R3_5 = (R4 + R3)/2;
}
else if (FormulaType == 11)
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose)/3;
    float DailyRange = PriorHigh - PriorLow;

    R_5 = PivotPoint + 0.5f*DailyRange;
    R1 = PivotPoint + 0.618f*DailyRange;
    R1_5 = PivotPoint + DailyRange;
    R2 = PivotPoint + 1.272f*DailyRange;
    R2_5 = PivotPoint + 1.618f*DailyRange;
    R3 = PivotPoint + 2*DailyRange;
    R4 = PivotPoint + 2.618f*DailyRange;

    S_5 = PivotPoint - 0.5f*DailyRange;
    S1 = PivotPoint - 0.618f*DailyRange;
    S1_5 = PivotPoint - DailyRange;
    S2 = PivotPoint - 1.272f*DailyRange;
    S2_5 = PivotPoint - 1.618f*DailyRange;
    S3 = PivotPoint - 2*DailyRange;
    S4 = PivotPoint - 2.618f*DailyRange;

}
else if (FormulaType == 12)
{
    PivotPoint = CurrentOpen;

    R_5 = PivotPoint;
    R1 = PivotPoint;
    R1_5 = PivotPoint;
    R2 = PivotPoint;
    R2_5 = PivotPoint;

```



```

R3 = PivotPoint;
R4 = PivotPoint;

S_5 = PivotPoint;
S1 = PivotPoint;
S1_5 = PivotPoint;
S2 = PivotPoint;
S2_5 = PivotPoint;
S3 = PivotPoint;
S4 = PivotPoint;
}
else if (FormulaType == 13) // Fibonacci Pivot Points
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose)/3;
    float DailyRange = PriorHigh - PriorLow;

    R1 = PivotPoint + 0.382f*DailyRange;
    R2 = PivotPoint + 0.618f*DailyRange;
    R3 = PivotPoint + 1*DailyRange;

    S1 = PivotPoint - 0.382f*DailyRange;
    S2 = PivotPoint - 0.618f*DailyRange;
    S3 = PivotPoint - 1*DailyRange;

    S2_5 = (S3 + S2)/2;
    S1_5 = (S2 + S1)/2;
    S_5 = (S1 + PivotPoint)/2;
    R_5 = (R1 + PivotPoint)/2;
    R1_5 = (R2 + R1)/2;
    R2_5 = (R3 + R2)/2;

}
else if (FormulaType == 14)
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;

    S1 = (2 * PivotPoint) - PriorHigh;
    R1 = (2 * PivotPoint) - PriorLow;

    S2 = PivotPoint - (R1 - S1);
    R2 = (PivotPoint - S1) + R1;

    S3 = PivotPoint - (R2 - S1) ;
    R3 = (PivotPoint - S1) + R2;
}
else if (FormulaType == 15) // Fibonacci Zone Pivots
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;

    R1 = PivotPoint + (PriorHigh - PriorLow)/2;
    R2 = PivotPoint + (PriorHigh - PriorLow);

    S1 = PivotPoint - (PriorHigh - PriorLow)/2;
    S2 = PivotPoint - (PriorHigh - PriorLow);

    // 0.5 to 0.618 defines resistance/support band 1
    // 1.0 to 1.382 defines resistance/support band 2
    // the bands could start as lines and if the user
    // wishes they can use Fill Rect top and bottom
    // or the transparent versions.
    R1_5 = PivotPoint + (PriorHigh - PriorLow)*0.618f;
    R2_5 = PivotPoint + (PriorHigh - PriorLow)*1.382f;

```

```

S1_5 = PivotPoint - (PriorHigh - PriorLow)*0.618f;
S2_5 = PivotPoint - (PriorHigh - PriorLow)*1.382f;
}
else if (FormulaType == 16)
{
/*
THE CENTRAL PIVOT RANGE FORMULA:

TC (TOPCENTRAL) = (PIVOT-BC)+PIVOT
PIVOT = (HIGH+LOW+CLOSE)/3
BC (BOTTOM CENTRAL) =(HIGH+LOW)/2
*/
PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;
S1 = (PriorHigh + PriorLow) / 2;
R1 = (PivotPoint - S1) + PivotPoint;
}
else if (FormulaType == 17)
{
PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3; // PP = (H+L+C)/3

// H: high, L: low, C: close

// Previous Day H: high, L: low, C: close
R3 = PriorHigh+2*(PivotPoint- PriorLow); //R3 = H+2*(PP-L)
R2 = PivotPoint+(PriorHigh-PriorLow); //R2 = PP+(H-L)
R2_5 = (R2+R3)/2; //M5 = (R2+R3)/2

S1 = 2*PivotPoint - PriorHigh; // S1 = (2*PP)-H

S2 = PivotPoint - (PriorHigh - PriorLow); // S2 = PP-(H-L)

S3 = PriorLow - 2 * (PriorHigh - PivotPoint); // S3 = L-2*(H-PP)

S2_5 = (S2 + S3)/2; // M0 = (S2+S3)/2

R1 = 2*PivotPoint - PriorLow; // R1 = (2*PP)-L
R_5 = (PivotPoint + R2)/2; // M3 = (PP+R2)/2
R1_5 = (R_5 + R2)/2; // M4 = (M3+R2)/2
S_5 = (PivotPoint + S2)/2; // M2 = (PP+S2)/2
S1_5 = (S_5 + S2)/2; // M1 = (M2+S2)/2
}
else if (FormulaType == 18)
{
PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;

R1 = PivotPoint + (PriorHigh - PriorLow) * 0.75f;
S1 = PivotPoint - (PriorHigh - PriorLow) * 0.75f;
}
else if (FormulaType == 19)
{
/*
Pivot Point (PP) = (Yesterday's High + Yesterday's Low + Current Open + Current Open) / 4

Resistance Level (R1) = PivotPoint + 2 * (Yesterday's High - Yesterday's Low)

Support Level (S1) = PivotPoint - 2 * (Yesterday's High - Yesterday's Low)
*/
PivotPoint = (PriorHigh + PriorLow + 2 * CurrentOpen) / 4;
R1 = PivotPoint + 2 * (PriorHigh - PriorLow);
S1 = PivotPoint - 2 * (PriorHigh - PriorLow);
}
else if (FormulaType == 20)
{

```

```

/* Calculations for ACD method
*/
PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;
float HighLowAverage = (PriorHigh + PriorLow)/2;
float Difference = fabs (HighLowAverage - PivotPoint);
S1 = PivotPoint -Difference;
R1 =PivotPoint +Difference;
}
else if (FormulaType == 21)
{
//////////Fibonacci Calculation//////////

//R10 = OP + ((HighD(1) - LowD(1)) * 3.618);
//R9 = OP + ((HighD(1) - LOWD(1))* 3.00);
//R8 = OP + ((HighD(1) - LowD(1)) * 2.618);
//R7 = OP + ((HighD(1) - LOWD(1))* 2.00);
//R6 = OP + ((HighD(1) - LowD(1)) * 1.618);
//R5 = OP + ((HighD(1) - LowD(1)) * 1.27);
//R4 = OP + ((HighD(1) - LowD(1)) * 1.000);
//R3 = OP + ((HighD(1) - LowD(1)) * .786);
//R2 = OP + ((HighD(1) - LowD(1)) * .618);
//R1 = OP + ((HighD(1) - LowD(1)) * .382);
//OP = (OpenD(0));// (HighD(1) + LowD(1) + CloseD(1)) / 3;
//S1 = OP - ((HighD(1) - LowD(1)) * .382);
//S2 = OP - ((HighD(1) - LowD(1)) * .618);
//S3 = OP - ((HighD (1) - LowD (1)) * .786);
//S4 = OP - ((HighD(1) - LowD(1)) * 1.000);
//S5 = OP - ((HighD(1) - LowD(1)) * 1.27);
//S6 = OP - ((HighD(1) - LowD(1)) * 1.618);
//S7 = OP - ((HighD(1) - LowD(1)) * 2.00);
//S8 = OP - ((HighD(1) - LowD(1)) * 2.618);
//S9 = OP - ((HighD(1) - LowD(1)) * 3.00);
//S10 = OP - ((HighD(1) - LowD(1)) * 3.618);

PivotPoint = CurrentOpen;

R_5 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 0.382));
R1 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 0.618));
R1_5 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 0.786));
R2 = static_cast<float>( PivotPoint + ((PriorHigh - PriorLow) * 1.0));
R2_5 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 1.27));
R3 = static_cast<float>( PivotPoint + ((PriorHigh - PriorLow) * 1.618));
R3_5 = static_cast<float>( PivotPoint + ((PriorHigh - PriorLow) * 2.0));
R4 = static_cast<float>( PivotPoint + ((PriorHigh - PriorLow) * 2.618));
R5 = static_cast<float>( PivotPoint + ((PriorHigh - PriorLow) * 3.0));

S_5 = static_cast<float>( PivotPoint - ((PriorHigh - PriorLow) * 0.382));
S1 = static_cast<float>( PivotPoint - ((PriorHigh - PriorLow) * 0.618));
S1_5 = static_cast<float>( PivotPoint - ((PriorHigh - PriorLow) * 0.786));
S2 = static_cast<float>( PivotPoint - ((PriorHigh - PriorLow) * 1.0));
S2_5 = static_cast<float>( PivotPoint - ((PriorHigh - PriorLow) * 1.27));
S3 = static_cast<float>( PivotPoint - ((PriorHigh - PriorLow) * 1.618));
S3_5 = static_cast<float>( PivotPoint - ((PriorHigh - PriorLow) * 2.0));
S4 = static_cast<float>( PivotPoint - ((PriorHigh - PriorLow) * 2.618));
S5 = static_cast<float>( PivotPoint - ((PriorHigh - PriorLow) * 3.0));
}
else if (FormulaType == 22)
{
PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;
float UDIFF = PriorHigh - PivotPoint;
float LDIFF = PivotPoint - PriorLow;

R1 = PivotPoint + .382f * LDIFF;
R2 = PivotPoint + .618f * LDIFF;
R3 = PivotPoint + LDIFF;

```

```

R4 = PivotPoint + 1.272f * LDIFF;
R5 = PivotPoint + 1.382f * LDIFF;
R6 = PivotPoint + 1.618f * LDIFF;
R7 = PivotPoint + 2.0f * LDIFF;
R8 = PivotPoint + 2.272f * LDIFF;
R9 = PivotPoint + 2.382f * LDIFF;
R10 = PivotPoint + 2.618f * LDIFF;

S1 = PivotPoint - .382f * UDIFF;
S2 = PivotPoint - .618f * UDIFF;
S3 = PivotPoint - UDIFF;
S4 = PivotPoint - 1.272f * UDIFF;
S5 = PivotPoint - 1.382f * UDIFF;
S6 = PivotPoint - 1.618f * UDIFF;
S7 = PivotPoint - 2.0f * UDIFF;
S8 = PivotPoint - 2.272f * UDIFF;
S9 = PivotPoint - 2.382f * UDIFF;
S10 = PivotPoint - 2.618f * UDIFF;
}
else if (FormulaType == 23)
{
    // Fibonacci Zone Pivot Points calculation
    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;
    // Daily Range
    const float DR = PriorHigh - PriorLow;

    R1 = PivotPoint + .5f * DR;
    R2 = PivotPoint + .618f * DR;
    R3 = PivotPoint + DR;
    R4 = PivotPoint + 1.382f * DR;
    R5 = PivotPoint + 1.618f * DR;
    R6 = PivotPoint + 2 * DR;
    R7 = PivotPoint + 2.618f * DR;

    S1 = PivotPoint - 0.5f * DR;
    S2 = PivotPoint - 0.618f * DR;
    S3 = PivotPoint - DR;
    S4 = PivotPoint - 1.382f * DR;
    S5 = PivotPoint - 1.618f * DR;
    S6 = PivotPoint - 2 * DR;
    S7 = PivotPoint - 2.618f * DR;
}
else if (FormulaType == 24) // Advanced Camarilla Pivot Points calculation
{
    const float DailyRange = PriorHigh - PriorLow;

    R1 = PriorClose + .0916f * DailyRange;
    R2 = PriorClose + .183f * DailyRange;
    R3 = PriorClose + .275f * DailyRange;
    R4 = PriorClose + .555f * DailyRange;
    R5 = PriorClose + .8244f * DailyRange;
    R6 = PriorClose + 1.0076f * DailyRange;
    PivotPoint = PriorClose;
    S1 = PriorClose - .0916f * DailyRange;
    S2 = PriorClose - .183f * DailyRange;
    S3 = PriorClose - .275f * DailyRange;
    S4 = PriorClose - .55f * DailyRange;
    S5 = PriorClose - .8244f * DailyRange;
    S6 = PriorClose - 1.0992f * DailyRange;
}
else if (FormulaType == 25)
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;

```

```

S1 = (PriorHigh + PriorLow) / 2;
R1 = PivotPoint + (PivotPoint - S1);
S2 = PriorLow - (PriorHigh - PriorLow)*0.25f;
S3 = PriorLow - (PriorHigh - PriorLow)*0.5f;
S4 = PriorLow - (PriorHigh - PriorLow)*0.75f;
S5 = PriorLow - (PriorHigh - PriorLow)*1.0f;
R2 = PriorHigh + (PriorHigh - PriorLow)*0.25f;
R3 = PriorHigh + (PriorHigh - PriorLow)*0.5f;
R4 = PriorHigh + (PriorHigh - PriorLow)*0.75f;
R5 = PriorHigh + (PriorHigh - PriorLow);
}
else if (FormulaType == 26)
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;
    const float DailyRange = PriorHigh - PriorLow;
    R1 = PivotPoint + 0.382f*(DailyRange);
    R2 = PivotPoint + 0.618f*(DailyRange);
    R3 = PivotPoint + 0.786f*(DailyRange);
    R4 = PivotPoint + 1.00f*(DailyRange);
    R5 = PivotPoint + 1.382f*(DailyRange);
    R6 = PivotPoint + 1.618f*(DailyRange);
    R7 = PivotPoint + 2.00f*(DailyRange);
    S1 = PivotPoint - 0.382f*(DailyRange);
    S2 = PivotPoint - 0.618f*(DailyRange);
    S3 = PivotPoint - 0.786f*(DailyRange);
    S4 = PivotPoint - 1.00f*(DailyRange);
    S5 = PivotPoint - 1.382f*(DailyRange);
    S6 = PivotPoint - 1.618f*(DailyRange);
    S7 = PivotPoint - 2.00f*(DailyRange);
}
else if (FormulaType == 27)
{
    PivotPoint = CurrentOpen;

    R_5 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 0.236));
    R1 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 0.382));
    R1_5 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 0.5));
    R2 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 0.618));
    R2_5 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 0.786));
    R3 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 1.0));
    R3_5 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 1.27));
    R4 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 1.618));
    R5 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 2.0));
    R6 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 2.618));
    R7 = static_cast<float>(PivotPoint + ((PriorHigh - PriorLow) * 3.0));

    S_5 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 0.236));
    S1 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 0.382));
    S1_5 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 0.5));
    S2 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 0.618));
    S2_5 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 0.786));
    S3 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 1.0));
    S3_5 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 1.27));
    S4 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 1.618));
    S5 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 2.0));
    S6 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 2.618));
    S7 = static_cast<float>(PivotPoint - ((PriorHigh - PriorLow) * 3.0));
}
else if (FormulaType == 28)
{
    PivotPoint = (PriorHigh + PriorLow + PriorClose) / 3;

    float d = (PriorHigh + PriorLow) / 2.0f;

```

```

float r = PivotPoint - d;

PivotPointHigh = PivotPoint + r;
PivotPointLow = PivotPoint - r;

S1 = (2 * PivotPoint) - PriorHigh;
R1 = (2 * PivotPoint) - PriorLow;

S2 = PivotPoint - (R1 - S1);
R2 = PivotPoint + (R1 - S1);

S3 = S1 - (PriorHigh - PriorLow);
R3 = R1 + (PriorHigh - PriorLow);

S4 = S3 - (S1 - S2);
R4 = R3 + (R2 - R1);
}

return 1;
}

/*=====*/
int CalculateDailyOHLC
( SCStudyInterfaceRef sc
, const SCDateTimeMS& CurrentBarTradingDayDate
, int InNumberOfDaysBack
, int InNumberOfDaysToCalculate
, int InUseSaturdayData
, int InUseThisIntradayChart
, int InDailyChartNumber
, SCGraphData& DailyChartData
, SCDateTimeArray& DailyChartDateTimes
, int UseDaySessionOnly
, float& Open
, float& High
, float& Low
, float& Close
, float& Volume
, int InIncludeFridayEveningSessionWithSundayEveningSession
, int InUseSundayData
)
{
    if (InUseThisIntradayChart) // Use this Chart
    {
        int ArraySize = sc.ArraySize;
        if (ArraySize <= 1)
            return 0;

        const int LastIndex = ArraySize - 1;

        SCDateTimeMS LastTradingDayDateInChart = sc.GetTradingDayDate(sc.BaseDateIn[LastIndex]);

        // Return and do not calculate if based upon trading day dates, that the current bar is earlier than the number of days
        // to calculate.
        if (CurrentBarTradingDayDate <=
SCDateTimeMS(LastTradingDayDateInChart).SubtractDays(InNumberOfDaysToCalculate))
        {
            return 0;
        }

        SCDateTimeMS StartDateTimeForTradingDate = sc.GetStartDateTimeForTradingDate(CurrentBarTradingDayDate);

        for (int DaysBackCount = 1; DaysBackCount <= InNumberOfDaysBack; DaysBackCount++)
        {
            StartDateTimeForTradingDate.SubtractDays(1);

```

```

SCDateTimeMS TradingDayDate = sc.GetTradingDayDate(StartDateTimeForTradingDate);

if (!InUseSundayData && TradingDayDate.IsSunday())
{
    StartDateTimeForTradingDate.SubtractDays(1);
}

TradingDayDate = sc.GetTradingDayDate(StartDateTimeForTradingDate);

if (!InUseSaturdayData && TradingDayDate.IsSaturday())
{
    StartDateTimeForTradingDate.SubtractDays(1);
}
}

SCDateTimeMS IntendedTradingDayDate = sc.GetTradingDayDate(StartDateTimeForTradingDate);

SCDateTimeMS ContainingIndexTradingDayDate;

for(int DayCount = 0; DayCount < DAYS_PER_WEEK; DayCount++)
{
    int ContainingIndex = sc.GetNearestMatchForSCDateTime(sc.ChartNumber, StartDateTimeForTradingDate);

    ContainingIndexTradingDayDate = sc.GetTradingDayDate(sc.BaseDateTimeIn[ContainingIndex]);

    if ((ContainingIndexTradingDayDate - IntendedTradingDayDate).GetDaysSinceBaseDate() <= -1)
    {
        ContainingIndex++;
        ContainingIndexTradingDayDate = sc.GetTradingDayDate(sc.BaseDateTimeIn[ContainingIndex]);
    }

    if ((ContainingIndexTradingDayDate - IntendedTradingDayDate).GetDaysSinceBaseDate() >= 1)
    {
        ContainingIndex--;
        ContainingIndexTradingDayDate = sc.GetTradingDayDate(sc.BaseDateTimeIn[ContainingIndex]);
    }

    if (!InUseSundayData && ContainingIndexTradingDayDate.IsSunday())
    {
        StartDateTimeForTradingDate.SubtractDays(1);
        continue;
    }

    if (!InUseSaturdayData && ContainingIndexTradingDayDate.IsSaturday())
    {
        StartDateTimeForTradingDate.SubtractDays(1);
        continue;
    }

    if (ContainingIndexTradingDayDate > CurrentBarTradingDayDate)
    {
        StartDateTimeForTradingDate.SubtractDays(1);
        continue;
    }

    if (InNumberOfDaysBack > 0
        && ContainingIndexTradingDayDate == CurrentBarTradingDayDate)
    {
        StartDateTimeForTradingDate.SubtractDays(1);
        IntendedTradingDayDate.SubtractDays(1);
        continue;
    }
}

```

```

    }

    break;
}

if(!UseDaySessionOnly)
{
    if (!sc.GetOpenHighLowCloseVolumeForDate(ContainingIndexTradingDayDate, Open, High, Low, Close, Volume,
InIncludeFridayEveningSessionWithSundayEveningSession))
    {
        return 0;
    }
}
else
{
    SCDatetimeMS StartDateTime;
    SCDatetimeMS EndDateTime;

    if(sc.StartTime1 <= sc.EndTime1)
    {
        StartDateTime.SetDate(ContainingIndexTradingDayDate);
        StartDateTime.SetTime(sc.StartTime1);

        EndDateTime.SetDate(ContainingIndexTradingDayDate);
        EndDateTime.SetTime(sc.EndTime1);
    }
    else
    {
        StartDateTime.SetDate(ContainingIndexTradingDayDate - 1);
        StartDateTime.SetTime(sc.StartTime1);

        EndDateTime.SetDate(ContainingIndexTradingDayDate);
        EndDateTime.SetTime(sc.EndTime1);
    }

    float NextOpen = 0;

    if(!sc.GetOHLCOfTimePeriod(StartDateTime, EndDateTime, Open, High, Low, Close, NextOpen))
        return 0;
}
}
else // Use Daily Chart
{
    SCDatetimeMS LastDateInDestinationChart = sc.GetTradingDayDate(sc.BaseDateTimeln[sc.ArraySize - 1]);

    // Return and do not calculate if based upon trading day dates, that the current bar is earlier than the number of days
to calculate.
    if (CurrentBarTradingDayDate <=
SCDatetimeMS(LastDateInDestinationChart).SubtractDays(InNumberOfDaysToCalculate))
    {
        return 0;
    }

    // Look for a matching date that is not a weekend

    int FirstIndexOfReferenceDay = 0;
    SCDatetimeMS ReferenceDay = CurrentBarTradingDayDate;
    ReferenceDay.SubtractDays(InNumberOfDaysBack);

    if (InNumberOfDaysBack == 0)
    {
        FirstIndexOfReferenceDay = sc.GetFirstNearestIndexForTradingDayDate(InDailyChartNumber,
ReferenceDay.GetDate());
    }
}

```



```

else
{
    while (ReferenceDay.IsWeekend(InUseSaturdayData != 0))
    {
        ReferenceDay.SubtractDays(1);
    }

    FirstIndexOfReferenceDay = sc.GetFirstNearestIndexForTradingDayDate(InDailyChartNumber,
ReferenceDay.GetDate());

    if (sc.GetTradingDayDate(DailyChartDateTimes[FirstIndexOfReferenceDay]) == CurrentBarTradingDayDate
        && FirstIndexOfReferenceDay >= 1)
    {
        --FirstIndexOfReferenceDay;
    }

#ifdef _DEBUG
    SCString DateString1 = sc.DateTimeToString(ReferenceDay, FLAG_DT_COMPLETE_DATETIME);
    SCString DateString2 = sc.DateTimeToString(DailyChartDateTimes[FirstIndexOfReferenceDay].GetAsDouble(),
FLAG_DT_COMPLETE_DATETIME);
    SCString LogMessage;
    LogMessage.Format("%s, %s", DateString1.GetChars(), DateString2.GetChars());
    sc.AddMessageToLog(LogMessage, 0);
#endif
}

    Open = DailyChartData[SC_OPEN][FirstIndexOfReferenceDay];
    High = DailyChartData[SC_HIGH][FirstIndexOfReferenceDay];
    Low = DailyChartData[SC_LOW][FirstIndexOfReferenceDay];
    Close = DailyChartData[SC_LAST][FirstIndexOfReferenceDay];
}

return 1;
}

int GetDailyChartIndexForDate(SCGraphData& DailyChartData, int NumberOfDaysToCalculate, SCStudyInterfaceRef&
sc, int DailyChartNumber, SCDateTimeArray& DailyChartDateTimeArray, int TargetDate)
{
    if (DailyChartData[0].GetArraySize() < 2)
        return -1;

    int DailyChartBarIndex = -1;
    int CountBack = DailyChartData[SC_OPEN].GetArraySize() - NumberOfDaysToCalculate - 1;
    for (int CurrentIndex = DailyChartData[SC_OPEN].GetArraySize() - 1; CurrentIndex >= CountBack && CurrentIndex >=
0; CurrentIndex--)
    {
        if (sc.GetTradingDayDateForChartNumber(DailyChartNumber, DailyChartDateTimeArray[CurrentIndex]) <
TargetDate)
        {
            DailyChartBarIndex = CurrentIndex;
            break;
        }
    }

    return DailyChartBarIndex;
}

/*=====*/
int CalculateDailyPivotPoints
( SCStudyInterfaceRef sc
, int IntradayChartDate
, int FormulaType
, int DailyChartNumber
, SCGraphData& DailyChartData

```

```

, SCDateTimeArray& DailyChartDateTimeArray
, int NumberOfDaysToCalculate
, int UseSaturdayData
, int UseThisChart
, int UseManualValues
, float UserOpen
, float UserHigh
, float UserLow
, float UserClose
, int UseDaySessionOnly
, float& PivotPoint
, float& PivotPointHigh
, float& PivotPointLow
, float& R_5
, float& R1, float& R1_5
, float& R2, float& R2_5
, float& R3
, float& S_5
, float& S1, float& S1_5
, float& S2, float& S2_5
, float& S3
, float& R3_5
, float& S3_5
, float& R4
, float& R4_5
, float& S4
, float& S4_5
, float& R5
, float& S5
, float& R6
, float& S6
, float& R7
, float& S7
, float& R8
, float& S8
, float& R9
, float& S9
, float& R10
, float& S10
, int UseDailyChartForSettlementOnly
)
{

    float Open = 0.0, High = 0.0, Low = 0.0, Close = 0.0, NextOpen = 0.0;

    if (UseThisChart != 0) // Use this Chart
    {
        int ArraySize = sc.BaseDateTimeIn.GetArraySize();
        if (ArraySize < 2)
            return 0;

        int LastIndex = ArraySize - 1;

        int CurrentDate = sc.GetTradingDayDate(sc.BaseDateTimeIn[LastIndex]);

        // Return and do not calculate pivots if the current bar to be calculated is further back than the number of days to
calculate
        if (CurrentDate - NumberOfDaysToCalculate >= IntradayChartDate)
            return 0;

        // Look for the last good date which is not a weekend and has data
        int FirstIndexOfPriorDay;
        int PreviousDay = IntradayChartDate;
        while (true)

```

```

{
    --PreviousDay;

    if (IsWeekend(PreviousDay, UseSaturdayData != 0))
        continue;

    // It is not a weekend
    FirstIndexOfPriorDay = sc.GetFirstIndexForDate(sc.ChartNumber, PreviousDay);
    if (sc.GetTradingDayDate(sc.BaseDateTimeIn[FirstIndexOfPriorDay]) == PreviousDay)
        break;

    if (FirstIndexOfPriorDay == 0)
    {
        // At the beginning of the date array, so we can't look back any farther
        break;
    }
}

if (sc.GetTradingDayDate(sc.BaseDateTimeIn[FirstIndexOfPriorDay]) != PreviousDay)
{
    // Previous day not found
    return 0;
}

if(!UseDaySessionOnly)
{
    if (IsC.GetOHLCForDate(PreviousDay, Open, High, Low, Close))
        return 0;

    float NextHigh, NextLow, NextClose; // The values returned for these are unused
    if (IsC.GetOHLCForDate(IntradayChartDate, NextOpen, NextHigh, NextLow, NextClose))
        NextOpen = Close; // If there is a failure, use the prior Close
}
else
{
    SCDatetime StartDateTime;
    SCDatetime EndDateTime;

    if(sc.StartTime1 <= sc.EndTime1)
    {
        StartDateTime.SetDate(PreviousDay);
        StartDateTime.SetTime(sc.StartTime1);

        EndDateTime.SetDate(PreviousDay);
        EndDateTime.SetTime(sc.EndTime1);
    }
    else
    {
        StartDateTime.SetDate(PreviousDay - 1);
        StartDateTime.SetTime(sc.StartTime1);

        EndDateTime.SetDate(PreviousDay);
        EndDateTime.SetTime(sc.EndTime1);
    }

    if(IsC.GetOHLCOfTimePeriod(StartDateTime, EndDateTime, Open, High, Low, Close, NextOpen))
        return 0;
}

if(UseDailyChartForSettlementOnly)
{

```

```

    int DailyChartBarIndex = GetDailyChartIndexForDate(DailyChartData, NumberOfDaysToCalculate, sc,
DailyChartNumber, DailyChartDateTimeArray, IntradayChartDate);

    if (DailyChartBarIndex == -1)
        return 0;

    Close = DailyChartData[SC_LAST][DailyChartBarIndex];
}
}
else // Use Daily Chart
{
    int DailyChartBarIndex = GetDailyChartIndexForDate(DailyChartData, NumberOfDaysToCalculate, sc,
DailyChartNumber, DailyChartDateTimeArray, IntradayChartDate);

    if(DailyChartBarIndex == -1)
        return 0;

    Open = DailyChartData[SC_OPEN][DailyChartBarIndex];
    High = DailyChartData[SC_HIGH][DailyChartBarIndex];
    Low = DailyChartData[SC_LOW][DailyChartBarIndex];
    Close = DailyChartData[SC_LAST][DailyChartBarIndex];

    if (DailyChartBarIndex == DailyChartData[0].GetArraySize() - 1 )
        NextOpen = DailyChartData[SC_LAST][DailyChartBarIndex];
    else
        NextOpen = DailyChartData[SC_OPEN][DailyChartBarIndex + 1];
}

// If we are calculating the last day, and the user has manually entered OHLC values, then use those
if (UseManualValues)
{
    int ArraySize = sc.BaseDateTimeln.GetArraySize();

    int LastIndex = ArraySize - 1;

    int CurrentDate = sc.GetTradingDayDate(sc.BaseDateTimeln[LastIndex]);

    if (CurrentDate == IntradayChartDate)
    {
        Open = (UserHigh + UserLow + UserClose) / 3.0f; // This is not normally used, but we need to set it to something
reasonable
        High = UserHigh;
        Low = UserLow;
        Close = UserClose;
        NextOpen = UserOpen;
    }
}

return CalculatePivotPoints
( Open
, High
, Low
, Close
, NextOpen
, PivotPoint
, PivotPointHigh
, PivotPointLow
, R_5
, R1, R1_5
, R2, R2_5
, R3
, S_5
, S1, S1_5
, S2, S2_5

```

```

, S3
, R3_5
, S3_5
, R4
, R4_5
, S4
, S4_5
, R5
, S5
, R6
, S6
, R7
, S7
, R8
, S8
, R9
, S9
, R10
, S10
, FormulaType
);
}

```

```

/*=====*/

```

```

SCFloatArrayRef Slope_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index)

```

```

{
    Out[Index] = In[Index] - In[Index - 1];

    return Out;
}

```

```

/*=====*/

```

```

SCFloatArrayRef CalculateAngle_S(SCFloatArrayRef InputArray, SCFloatArrayRef OutputArray, int Index, int Length,
float ValuePerPoint)

```

```

{
    if (ValuePerPoint == 0)
        ValuePerPoint = 1;

    float BackValue = InputArray[Index - Length];
    float CurrentValue = InputArray[Index];

    float PointChange = (CurrentValue - BackValue) / ValuePerPoint;

    OutputArray[Index] = static_cast<float>(atan2(static_cast<double>(PointChange), static_cast<double>(Length)) *
180.0 / M_PI);

    return OutputArray;
}

```

```

/*=====*/

```

```

SCFloatArrayRef DoubleStochastic_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Out, SCFloatArrayRef MovAvgIn,
SCFloatArrayRef MovAvgOut, SCFloatArrayRef MovAvgIn2, SCFloatArrayRef Unused, int Index, int Length, int
MovAvgLength, int MovAvgType)

```

```

{
    /*-----
    MovingAvgX (((PI - Lowest (PI , 10)) / Deler2) * 100) , 3 , False)

```

where:

```

PL= MovingAvgX (((Close - Lowest Low (10)) / Deler) * 100 , 3 , False)
Deler= IFF (highest_minus_lowest > 0 , highest_minus_lowest , 1)
Deler2= IFF (highest_minus_lowest#2 > 0 , highest_minus_lowest#2 , 1)
Highest_minus_lowest= Highest High (10) - Lowest Low (10)
Highest_minus_lowest#2= Highest High (5) - Lowest Low (5)

```

```

-----*/

```

```

float HighestMinusLowest1 = GetHighest(BaseDataIn[SC_HIGH], Index, Length) - GetLowest(BaseDataIn[SC_LOW],
Index, Length);
float Calc1 = HighestMinusLowest1 > 0 ? HighestMinusLowest1 : 1;

MovAvgIn[Index] = (BaseDataIn[SC_LAST][Index] - GetLowest(BaseDataIn[SC_LOW], Index, Length))/Calc1*100.0f;

MovingAverage_S(MovAvgIn, MovAvgOut, MovAvgType, Index, MovAvgLength);

float HighestMinusLowest2 = GetHighest(MovAvgOut, Index, Length) - GetLowest(MovAvgOut, Index, Length);
float Calc2 = HighestMinusLowest2 > 0 ? HighestMinusLowest2 : 1;

MovAvgIn2[Index]= (MovAvgOut[Index] - GetLowest(MovAvgOut, Index, Length))/Calc2*100.0f;

MovingAverage_S(MovAvgIn2, Out, MovAvgType, Index, MovAvgLength);

return Out;
}

/*=====*/
double GetStandardError(SCFloatArrayRef In, int Index, int Length)
{
    if (Length <= 1)
        return 0.0;

    if (Index < Length - 1)
        return 0.0;

    double AvgX = (Length - 1) * 0.5f;

    double AvgY = 0;
    for (int Offset = 0; Offset < Length; Offset++)
        AvgY += In[Index - Offset];

    AvgY /= Length;

    double SumDxSqr = 0;
    double SumDySqr = 0;
    double SumDxDy = 0;
    for (int Offset = 0; Offset < Length; Offset++)
    {
        double Dx = Offset - AvgX;
        double Dy = In[Offset + Index - Length + 1] - AvgY;
        SumDxSqr += Dx * Dx;
        SumDySqr += Dy * Dy;
        SumDxDy += Dx * Dy;
    }

    return sqrt((SumDySqr - (SumDxDy * SumDxDy) / SumDxSqr) / (Length - 2));
}

/*=====*/
SCFloatArray& StandardError_S(SCFloatArrayRef In, SCFloatArray& Out, int Index, int Length)
{
    Out[Index] = static_cast<float>(GetStandardError(In, Index, Length));

    return Out;
}

/*=====*/
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;
}

/*=====*/
SCFloatArrayRef CumulativeDeltaVolume_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Open, SCFloatArrayRef High, SCFloatArrayRef Low, SCFloatArrayRef Close, int Index, int ResetCumulativeCalculation)
{
    SCFloatArrayRef BidVolume    = BaseDataIn[SC_BIDVOL];
    SCFloatArrayRef AskVolume    = BaseDataIn[SC_ASKVOL];
    SCFloatArrayRef DifferenceHigh = BaseDataIn[SC_ASKBID_DIFF_HIGH];
    SCFloatArrayRef DifferenceLow  = BaseDataIn[SC_ASKBID_DIFF_LOW];

    if (Index == 0 || ResetCumulativeCalculation)
    {
        Open[Index] = 0;
        High[Index] = DifferenceHigh[Index];
        Low[Index] = DifferenceLow[Index];
        Close[Index] = AskVolume[Index] - BidVolume[Index];

        if (Open[Index] > High[Index])
            Open[Index] = High[Index];

        if (Open[Index] < Low[Index])
            Open[Index] = Low[Index];
    }
    else
    {
        Open[Index] = Close[Index-1];
        High[Index] = Close[Index-1] + DifferenceHigh[Index];
        Low[Index] = Close[Index-1] + DifferenceLow[Index];
        Close[Index] = Close[Index-1] + (AskVolume[Index] - BidVolume[Index]);

        if (Open[Index] > High[Index])
            Open[Index] = High[Index];

        if (Open[Index] < Low[Index])
            Open[Index] = Low[Index];
    }

    return Close;
}

/*=====*/
SCFloatArrayRef CumulativeDeltaTicks_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Open, SCFloatArrayRef High, SCFloatArrayRef Low, SCFloatArrayRef Close, int Index, int ResetCumulativeCalculation)

```

```

{
    SCFloatArrayRef BidNumberTrades = BaseDataIn[SC_BIDNT];
    SCFloatArrayRef AskNumberTrades = BaseDataIn[SC_ASKNT];
    SCFloatArrayRef DifferenceHigh = BaseDataIn[SC_ASKBID_NUM_TRADES_DIFF_HIGH];
    SCFloatArrayRef DifferenceLow = BaseDataIn[SC_ASKBID_NUM_TRADES_DIFF_LOW];

    if (Index == 0 || ResetCumulativeCalculation)
    {
        Open[Index] = 0;
        High[Index] = DifferenceHigh[Index];
        Low[Index] = DifferenceLow[Index];
        Close[Index] = AskNumberTrades[Index] - BidNumberTrades[Index];

        if (Open[Index] > High[Index])
            Open[Index] = High[Index];

        if (Open[Index] < Low[Index])
            Open[Index] = Low[Index];
    }
    else
    {
        Open[Index] = Close[Index-1];
        High[Index] = Close[Index-1] + DifferenceHigh[Index];
        Low[Index] = Close[Index-1] + DifferenceLow[Index];
        Close[Index] = Close[Index-1] + (AskNumberTrades[Index] - BidNumberTrades[Index]);

        if (Open[Index] > High[Index])
            Open[Index] = High[Index];

        if (Open[Index] < Low[Index])
            Open[Index] = Low[Index];
    }

    return Close;
}

/*=====*/
SCFloatArrayRef CumulativeDeltaTickVolume_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef Open, SCFloatArrayRef High, SCFloatArrayRef Low, SCFloatArrayRef Close, int Index, int ResetCumulativeCalculation)
{
    SCFloatArrayRef UpTickVolume = BaseDataIn[SC_UPVOL];
    SCFloatArrayRef DownTickVolume = BaseDataIn[SC_DOWNVOL];
    SCFloatArrayRef DifferenceHigh = BaseDataIn[SC_UPDOWN_VOL_DIFF_HIGH];
    SCFloatArrayRef DifferenceLow = BaseDataIn[SC_UPDOWN_VOL_DIFF_LOW];

    if (Index == 0 || ResetCumulativeCalculation)
    {
        Open[Index] = 0;
        High[Index] = DifferenceHigh[Index];
        Low[Index] = DifferenceLow[Index];
        Close[Index] = UpTickVolume[Index] - DownTickVolume[Index];

        if (Open[Index] > High[Index])
            Open[Index] = High[Index];

        if (Open[Index] < Low[Index])
            Open[Index] = Low[Index];
    }
    else
    {
        Open[Index] = Close[Index-1];
        High[Index] = Close[Index-1] + DifferenceHigh[Index];
        Low[Index] = Close[Index-1] + DifferenceLow[Index];
        Close[Index] = Close[Index-1] + (UpTickVolume[Index] - DownTickVolume[Index]);
    }
}

```



```

        if (Open[Index] > High[Index])
            Open[Index] = High[Index];

        if (Open[Index] < Low[Index])
            Open[Index] = Low[Index];
    }

    return Close;
}

/*=====*/
// ZigZagValues: value of zig zag line
// ZigZagPeakType: +1=high peak, -1=low peak, 0=not peak
// ZigZagPeakIndex: index of current peak, -1=no peak yet
SCFloatArrayRef ResettableZigZag_S(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow,
SCFloatArrayRef ZigZagValues, SCFloatArrayRef ZigZagPeakType, SCFloatArrayRef ZigZagPeakIndex, int StartIndex,
int Index, float ReversalPercent, float ReversalAmount, SCStudyInterfaceRef sc)
{
    if (Index == StartIndex)
    {
        for(int ZeroIndex = StartIndex ; ZeroIndex < ZigZagValues.GetArraySize(); ZeroIndex++)
        {
            ZigZagValues[ZeroIndex] = 0;
            ZigZagPeakType[ZeroIndex] = 0;
            ZigZagPeakIndex[ZeroIndex] = 0;
        }
        ZigZagPeakIndex[StartIndex] = -1;
        return ZigZagValues;
    }

    if (ZigZagPeakIndex[Index] == 0)
    {
        if (ZigZagPeakIndex[Index-1] != -1)
        {
            // new bar, copy values forward
            ZigZagPeakIndex[Index] = ZigZagPeakIndex[Index-1];
            ZigZagValues[Index] = 0;
            ZigZagPeakType[Index] = 0;
        }
        else
        {
            // still looking for initial trend
            if (InputDataHigh[Index] > InputDataHigh[StartIndex] && InputDataLow[Index] > InputDataLow[StartIndex])
//bullish trend
            {
                ZigZagPeakIndex[Index] = static_cast<float>(Index);
                ZigZagValues[Index] = InputDataHigh[Index];
                ZigZagPeakType[Index] = 1;

                ZigZagPeakIndex[StartIndex] = static_cast<float>(StartIndex);
                ZigZagValues[StartIndex] = InputDataLow[StartIndex];
                ZigZagPeakType[StartIndex] = -1;

                float Increment = (InputDataHigh[Index] - InputDataLow[StartIndex])/(Index-StartIndex);

                for (int ArrayIndex=StartIndex+1, Count=1; ArrayIndex<Index; ArrayIndex++, Count++)
                {
                    ZigZagValues[ArrayIndex] = InputDataLow[StartIndex] + Count*Increment;
                    ZigZagPeakIndex[ArrayIndex] = static_cast<float>(StartIndex);
                    ZigZagPeakType[ArrayIndex] = 0;
                }
            }
            else if (InputDataHigh[Index] < InputDataHigh[StartIndex] && InputDataLow[Index] < InputDataLow[StartIndex])

```

```
//bearish trend
```

```
{
    ZigZagPeakIndex[Index] = static_cast<float>(Index);
    ZigZagValues[Index] = InputDataLow[Index];
    ZigZagPeakType[Index] = -1;

    ZigZagPeakIndex[StartIndex] = static_cast<float>(StartIndex);
    ZigZagValues[StartIndex] = InputDataHigh[StartIndex];
    ZigZagPeakType[StartIndex] = 1;

    float Increment = (InputDataHigh[StartIndex] - InputDataLow[Index])/(Index-StartIndex);

    for (int ArrayIndex=StartIndex+1, Count=1; ArrayIndex<Index; ArrayIndex++, Count++)
    {
        ZigZagValues[ArrayIndex] = InputDataHigh[StartIndex] - Count*Increment;
        ZigZagPeakIndex[ArrayIndex] = static_cast<float>(StartIndex);
        ZigZagPeakType[ArrayIndex] = 0;
    }
}
else
{
    ZigZagPeakIndex[Index] = -1;
    ZigZagValues[Index] = 0;
    ZigZagPeakType[Index] = 0;
}

return ZigZagValues;
}
}
```

  

```
int CurrentPeakIndex = static_cast<int>(ZigZagPeakIndex[Index]);

if (CurrentPeakIndex > 0)
{
    int CurrentPeakType = static_cast<int>(ZigZagPeakType[CurrentPeakIndex]);
    float CurrentPeakValue = ZigZagValues[CurrentPeakIndex];

    int PriorPeakIndex = static_cast<int>(ZigZagPeakIndex[CurrentPeakIndex-1]);
    float PriorPeakValue = ZigZagValues[PriorPeakIndex];

    bool CurrentBarIsPeak = Index == CurrentPeakIndex;

    float BarLow = InputDataLow[Index];
    float BarHigh = InputDataHigh[Index];

    if (CurrentPeakType > 0)
    {
        float ReversalPrice;
        if (ReversalPercent == 0.0f)
            ReversalPrice = CurrentPeakValue - ReversalAmount;
        else
            ReversalPrice = CurrentPeakValue - (CurrentPeakValue*ReversalPercent);

        //if (!CurrentBarIsPeak && BarLow < ReversalPrice)
        if (!CurrentBarIsPeak && sc.FormattedEvaluate(BarLow, sc.BaseGraphValueFormat, LESS_OPERATOR,
ReversalPrice, sc.BaseGraphValueFormat))
        {
            ZigZagPeakIndex[Index] = static_cast<float>(Index);
            ZigZagValues[Index] = BarLow;
            ZigZagPeakType[Index] = -1;

            float Increment = (CurrentPeakValue - BarLow)/(Index - CurrentPeakIndex);
```

```

    for (int n = CurrentPeakIndex+1, Count=1; n < Index; n++, Count++)
    {
        ZigZagValues[n] = CurrentPeakValue - Count*Increment;
    }
}
//else if (BarHigh > CurrentPeakValue)
else if (sc.FormattedEvaluate(BarHigh, sc.BaseGraphValueFormat, GREATER_EQUAL_OPERATOR,
CurrentPeakValue, sc.BaseGraphValueFormat))
{
    ZigZagPeakIndex[Index] = static_cast<float>(Index);
    ZigZagValues[Index] = BarHigh;
    ZigZagPeakType[Index] = 1;

    float Increment = (BarHigh - PriorPeakValue)/(Index - PriorPeakIndex);

    for (int n = PriorPeakIndex+1, Count=1; n < Index; n++, Count++)
    {
        ZigZagValues[n] = PriorPeakValue + Count*Increment;
        ZigZagPeakIndex[n] = static_cast<float>(PriorPeakIndex);
        ZigZagPeakType[n] = 0;
    }
}
else if (CurrentPeakType < 0)
{
    float ReversalPrice;
    if (ReversalPercent == 0.0f)
        ReversalPrice = CurrentPeakValue + ReversalAmount;
    else
        ReversalPrice = CurrentPeakValue + (CurrentPeakValue*ReversalPercent);

    //if (!CurrentBarIsPeak && BarHigh > ReversalPrice)
    if (!CurrentBarIsPeak && sc.FormattedEvaluate(BarHigh, sc.BaseGraphValueFormat, GREATER_OPERATOR,
ReversalPrice, sc.BaseGraphValueFormat))
    {
        ZigZagPeakIndex[Index] = static_cast<float>(Index);
        ZigZagValues[Index] = BarHigh;
        ZigZagPeakType[Index] = 1;

        float Increment = (BarHigh - CurrentPeakValue)/(Index - CurrentPeakIndex);

        for (int n = CurrentPeakIndex+1, Count=1; n < Index; n++, Count++)
        {
            ZigZagValues[n] = CurrentPeakValue + Count*Increment;
        }
    }
    //else if (BarLow < CurrentPeakValue)
    else if (sc.FormattedEvaluate(BarLow, sc.BaseGraphValueFormat, LESS_EQUAL_OPERATOR,
CurrentPeakValue, sc.BaseGraphValueFormat))
    {
        ZigZagPeakIndex[Index] = static_cast<float>(Index);
        ZigZagValues[Index] = BarLow;
        ZigZagPeakType[Index] = -1;

        float Increment = (PriorPeakValue - BarLow)/(Index - PriorPeakIndex);

        for (int n = PriorPeakIndex+1, Count=1; n < Index; n++, Count++)
        {
            ZigZagValues[n] = PriorPeakValue - Count*Increment;
            ZigZagPeakIndex[n] = static_cast<float>(PriorPeakIndex);
            ZigZagPeakType[n] = 0;
        }
    }
}
}
}

```

```

    return ZigZagValues;
}

/*=====*/
// ZigZagPeakType : +1=high peak, -1=low peak, 0=not peak
// ZigZagPeakIndex: index of current peak, -1=no peak yet
SCFloatArrayRef ResettableZigZag2_S(SCFloatArrayRef InputDataHigh, SCFloatArrayRef InputDataLow,
SCFloatArrayRef ZigZagValues, SCFloatArrayRef ZigZagPeakType, SCFloatArrayRef ZigZagPeakIndex, int StartIndex,
int Index, int NumberOfBars, float ReversalAmount, SCStudyInterfaceRef sc)
{
    if (Index == StartIndex)
    {
        for(int ZeroIndex = StartIndex ; ZeroIndex < ZigZagValues.GetArraySize(); ZeroIndex++)
        {
            ZigZagValues[ZeroIndex] = 0;
            ZigZagPeakType[ZeroIndex] = 0;
            ZigZagPeakIndex[ZeroIndex] = 0;
        }
        ZigZagPeakIndex[StartIndex] = -1;
        return ZigZagValues;
    }

    if (ZigZagPeakIndex[Index] == 0)
    {
        if (ZigZagPeakIndex[Index-1] != -1)
        {
            // new bar, copy values forward
            ZigZagPeakIndex[Index] = ZigZagPeakIndex[Index-1];
            ZigZagValues[Index] = 0;
            ZigZagPeakType[Index] = 0;
        }
        else
        {
            // still looking for initial trend
            if (InputDataHigh[Index] > InputDataHigh[StartIndex] && InputDataLow[Index] > InputDataLow[StartIndex])
//bullish trend
            {
                ZigZagPeakIndex[Index] = static_cast<float>(Index);
                ZigZagValues[Index] = InputDataHigh[Index];
                ZigZagPeakType[Index] = 1;

                ZigZagPeakIndex[StartIndex] = static_cast<float>(StartIndex);
                ZigZagValues[StartIndex] = InputDataLow[StartIndex];
                ZigZagPeakType[StartIndex] = -1;

                float Increment = (InputDataHigh[Index] - InputDataLow[StartIndex])/(Index-StartIndex);

                for (int n=StartIndex+1, Count=1; n<Index; n++, Count++)
                {
                    ZigZagValues[n] = InputDataLow[StartIndex] + Count*Increment;
                    ZigZagPeakIndex[n] = static_cast<float>(StartIndex);
                    ZigZagPeakType[n] = 0;
                }
            }
            else if (InputDataHigh[Index] < InputDataHigh[StartIndex] && InputDataLow[Index] < InputDataLow[StartIndex])
//bearish trend
            {
                ZigZagPeakIndex[Index] = static_cast<float>(Index);
                ZigZagValues[Index] = InputDataLow[Index];
                ZigZagPeakType[Index] = -1;

                ZigZagPeakIndex[StartIndex] = static_cast<float>(StartIndex);

```

```

ZigZagValues[StartIndex] = InputDataHigh[StartIndex];
ZigZagPeakType[StartIndex] = 1;

float Increment = (InputDataHigh[StartIndex] - InputDataLow[Index])/(Index-StartIndex);

for (int n=StartIndex+1, Count=1; n<Index; n++, Count++)
{
    ZigZagValues[n] = InputDataHigh[StartIndex] - Count*Increment;
    ZigZagPeakIndex[n] = static_cast<float>(StartIndex);
    ZigZagPeakType[n] = 0;
}
}
else
{
    ZigZagPeakIndex[Index] = -1;
    ZigZagValues[Index] = 0;
    ZigZagPeakType[Index] = 0;
}
return ZigZagValues;
}
}

int CurrentPeakIndex = static_cast<int>(ZigZagPeakIndex[Index]);

if (CurrentPeakIndex > 0)
{
    int CurrentPeakType = static_cast<int>(ZigZagPeakType[CurrentPeakIndex]);
    float CurrentPeakValue = ZigZagValues[CurrentPeakIndex];

    int PriorPeakIndex = static_cast<int>(ZigZagPeakIndex[CurrentPeakIndex-1]);
    float PriorPeakValue = ZigZagValues[PriorPeakIndex];

    bool CurrentBarIsPeak = Index == CurrentPeakIndex;

    float BarLow = InputDataLow[Index];
    float BarHigh = InputDataHigh[Index];

    if (CurrentPeakType > 0)
    {
        //if (BarHigh >= CurrentPeakValue)
        if (sc.FormattedEvaluate(BarHigh, sc.BaseGraphValueFormat, GREATER_EQUAL_OPERATOR,
CurrentPeakValue, sc.BaseGraphValueFormat))
        {
            ZigZagPeakIndex[Index] = static_cast<float>(Index);
            ZigZagValues[Index] = BarHigh;
            ZigZagPeakType[Index] = 1;

            float Increment = (BarHigh - PriorPeakValue)/(Index - PriorPeakIndex);

            for (int n = PriorPeakIndex+1, Count=1; n < Index; n++, Count++)
            {
                ZigZagValues[n] = PriorPeakValue + Count*Increment;
                ZigZagPeakIndex[n] = static_cast<float>(PriorPeakIndex);
                ZigZagPeakType[n] = 0;
            }
        }
        else
        {
            int SkipCount = Index - CurrentPeakIndex;

            if (
                //BarLow < PriorPeakValue ||
                sc.FormattedEvaluate(BarLow, sc.BaseGraphValueFormat, LESS_OPERATOR, PriorPeakValue,

```

```
sc.BaseGraphValueFormat) ||
```

```
    //(SkipCount >= NumberOfBars && CurrentPeakValue - BarLow > ReversalAmount)
    (SkipCount >= NumberOfBars && sc.FormattedEvaluate(CurrentPeakValue - BarLow,
sc.BaseGraphValueFormat, GREATER_OPERATOR, ReversalAmount, sc.BaseGraphValueFormat))

    ) //change to bearish
    {
        ZigZagPeakIndex[Index] = static_cast<float>(Index);
        ZigZagValues[Index] = BarLow;
        ZigZagPeakType[Index] = -1;

        float Increment = (CurrentPeakValue - BarLow)/(Index - CurrentPeakIndex);

        for (int n = CurrentPeakIndex+1, Count=1; n < Index; n++, Count++)
        {
            ZigZagValues[n] = CurrentPeakValue - Count*Increment;
        }
    }
}
else if (CurrentPeakType < 0)
{
    //if (BarLow <= CurrentPeakValue)
    if (sc.FormattedEvaluate(BarLow, sc.BaseGraphValueFormat, LESS_EQUAL_OPERATOR, CurrentPeakValue,
sc.BaseGraphValueFormat))
    {
        ZigZagPeakIndex[Index] = static_cast<float>(Index);
        ZigZagValues[Index] = BarLow;
        ZigZagPeakType[Index] = -1;

        float Increment = (PriorPeakValue - BarLow)/(Index - PriorPeakIndex);

        for (int n = PriorPeakIndex+1, Count=1; n < Index; n++, Count++)
        {
            ZigZagValues[n] = PriorPeakValue - Count*Increment;
            ZigZagPeakIndex[n] = static_cast<float>(PriorPeakIndex);
            ZigZagPeakType[n] = 0;
        }
    }
    else
    {
        int SkipCount = Index - CurrentPeakIndex;

        if (
            //BarHigh > PriorPeakValue ||
            sc.FormattedEvaluate(BarHigh, sc.BaseGraphValueFormat, GREATER_OPERATOR, PriorPeakValue,
sc.BaseGraphValueFormat) ||

            //(SkipCount >= NumberOfBars && BarHigh - CurrentPeakValue > ReversalAmount)
            (SkipCount >= NumberOfBars && sc.FormattedEvaluate(BarHigh - CurrentPeakValue,
sc.BaseGraphValueFormat, GREATER_OPERATOR, ReversalAmount, sc.BaseGraphValueFormat))

            ) //change to bullish
        {
            ZigZagPeakIndex[Index] = static_cast<float>(Index);
            ZigZagValues[Index] = BarHigh;
            ZigZagPeakType[Index] = 1;

            float Increment = (BarHigh - CurrentPeakValue)/(Index - CurrentPeakIndex);

            for (int n = CurrentPeakIndex+1, Count=1; n < Index; n++, Count++)
            {
                ZigZagValues[n] = CurrentPeakValue + Count*Increment;
            }
        }
    }
}
```

```

    }
    }
}
return ZigZagValues;
}

```

```

/*=====*/
void Vortex_S(SCBaseDataRef BaseDataIn, SCFloatArrayRef TrueRangeOut, SCFloatArrayRef VortexMovementUpOut,
SCFloatArrayRef VortexMovementDownOut, SCFloatArrayRef VIPlusOut, SCFloatArrayRef VIMinusOut, int Index, int
VortexLength)
{
    TrueRange_S(BaseDataIn, TrueRangeOut, Index);

    if (Index == 0)
    {
        VortexMovementUpOut[Index] = abs(BaseDataIn[SC_HIGH][Index] - BaseDataIn[SC_LOW][Index]);

        VortexMovementDownOut[Index] = abs(BaseDataIn[SC_LOW][Index] - BaseDataIn[SC_HIGH][Index]);
    }

    else
    {
        VortexMovementUpOut[Index] = abs(BaseDataIn[SC_HIGH][Index] - BaseDataIn[SC_LOW][Index - 1]);

        VortexMovementDownOut[Index] = abs(BaseDataIn[SC_LOW][Index] - BaseDataIn[SC_HIGH][Index - 1]);
    }

    float VMUpSum = 0;
    float VMDownSum = 0;
    float TrueRangeSum = 0;

    for (int Iteration = 0; Iteration < VortexLength && Index >= Iteration; ++Iteration)
    {
        VMUpSum += VortexMovementUpOut[Index - Iteration];
        VMDownSum += VortexMovementDownOut[Index - Iteration];
        TrueRangeSum += TrueRangeOut[Index - Iteration];
    }

    if (TrueRangeSum != 0)
    {
        VIPlusOut[Index] = VMUpSum / TrueRangeSum;
        VIMinusOut[Index] = VMDownSum / TrueRangeSum;
    }
}
/*=====*/

```

```

void HeikinAshi_S(SCBaseDataRef BaseDataIn,
int Index,
int Length,
SCFloatArrayRef OpenOut,
SCFloatArrayRef HighOut,
SCFloatArrayRef LowOut,
SCFloatArrayRef LastOut
, int SetCloseToCurrentPriceAtLastBar)
{
    const float OpenVal = BaseDataIn[SC_OPEN][Index];
    const float HighVal = BaseDataIn[SC_HIGH][Index];
    const float LowVal = BaseDataIn[SC_LOW][Index];
    const float LastVal = BaseDataIn[SC_LAST][Index];

    if (Index == BaseDataIn.GetArraySize() - 1 && SetCloseToCurrentPriceAtLastBar != 0)
        LastOut[Index] = LastVal;
    else
        LastOut[Index] = (OpenVal + HighVal + LowVal + LastVal) / 4.0f;
}

```

```

    if (Index == 0)
        OpenOut[Index] = OpenVal;
    else
    {
        OpenOut[Index] = (OpenOut[Index - 1] + LastOut[Index - 1]) / 2.0f;
    }

    HighOut[Index] = max(HighVal, OpenOut[Index]);
    LowOut[Index] = min(LowVal, OpenOut[Index]);
}

/*=====*/
void InverseFisherTransform_S(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef CalcArray1,
SCFloatArrayRef CalcArray2, int Index, int HighestLowestLength, int MovingAverageLength, int MovAvgType)
{
    float Highest = GetHighest(In, Index, HighestLowestLength);
    float Lowest = GetLowest(In, Index, HighestLowestLength);
    float Range = (Highest - Lowest);

    if (Range != 0)
    {
        float Ratio = 10.0f / Range;
        float CalcValue2 = ((In[Index] - Lowest) * Ratio) - 5.0f;
        CalcArray2[Index] = CalcValue2;

        MovingAverage_S(CalcArray2, CalcArray1, MovAvgType, Index, MovingAverageLength);
        float CalcValue1 = CalcArray1[Index];
        float RefData = CalcValue1;

        //http://en.wikipedia.org/wiki/Exponential_function
        Out[Index] = (exp(2 * RefData) - 1) / (exp(2 * RefData) + 1);

        if (!_isnan(Out[Index]))
        {
            int A = 1;
        }
    }
    else
        Out[Index] = Out[Index - 1];
}

/*=====*/
void InverseFisherTransformRSI_S(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)
{
    RSI_S(In, RSIArray1, RSIArray2, RSIArray3, RSIArray4, RSIArray5, Index, InternalRSIMovAvgType, RSILength);

    //put RSI calculation into an Array to be used with the moving average, since an array is required.
    CalcArray1[Index] = static_cast<float>((RSIArray1[Index] - 50)*0.1);

    MovingAverage_S(CalcArray1, CalcArray2, MovingAverageOfRSIType, Index, RSIMovingAverageLength);

    float WMA = CalcArray2[Index];

    Out[Index] = (exp(2 * WMA) - 1) / (exp(2 * WMA) + 1);
}

/*=====*/

```



```

void MovingAverageCumulative_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index)
{
    if (Index == 0)
    {
        Out[0] = In[0];
        return;
    }

    Out[Index] = (In[Index] + Index * Out[Index - 1]) / (Index + 1);
}

/*=====*/
void CalculateCumulativeLogLogRegressionStatistics_S(SCFloatArrayRef In, double &Slope, double &Y_Intercept,
SCFloatArrayRef Array_Sum_x, SCFloatArrayRef Array_Sum_x2, SCFloatArrayRef Array_Sum_x_2, SCFloatArrayRef
Array_Sum_y, SCFloatArrayRef Array_Sum_xy, int Index)
{
    if (Index == 0 || Index == 1) //The slope is not defined until Index = 2.
        return;

    //The sums are not defined at Index = 0, so we will simply define them to be 0 there.
    if (Index == 0)
    {
        Array_Sum_x[0] = 0;
        Array_Sum_x2[0] = 0;
        Array_Sum_x_2[0] = 0;
        Array_Sum_y[0] = 0;
        Array_Sum_xy[0] = 0;
    }

    //Initialize the sums at Index = 1. Since x[1] = 1, and log(1) = 0, all sums involving x are 0 at Index = 1.
    if (Index == 1)
    {
        Array_Sum_x[1] = 0;
        Array_Sum_x2[1] = 0;
        Array_Sum_x_2[1] = 0;
        Array_Sum_y[1] = log(In[1]);
        Array_Sum_xy[1] = 0;
    }

    //Compute the sums recursively.
    Array_Sum_x[Index] = static_cast<float>(log(Index) + Array_Sum_x[Index - 1]);
    Array_Sum_x2[Index] = static_cast<float>(log(Index)*log(Index) + Array_Sum_x2[Index - 1]);
    Array_Sum_x_2[Index] = Array_Sum_x[Index]*Array_Sum_x[Index];
    Array_Sum_y[Index] = log(In[Index]) + Array_Sum_y[Index - 1];
    Array_Sum_xy[Index] = static_cast<float>(log(Index)*log(In[Index]) + Array_Sum_xy[Index - 1]);

    //Compute the log-log regression statistics.
    double b_numerator = (Index * Array_Sum_xy[Index] - Array_Sum_x[Index] * Array_Sum_y[Index]);
    double b_denominator = Index * Array_Sum_x2[Index] - Array_Sum_x_2[Index];
    Slope = b_numerator / b_denominator;

    //Y-Intercept is at Index == 0.
    Y_Intercept = (Array_Sum_y[Index] - Slope * Array_Sum_x[Index]) / Index;
}

/*=====*/
SCFloatArrayRef CumulativeStdDev_S(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef Array_XSquared,
SCFloatArrayRef Array_Mean, SCFloatArrayRef Array_MeanOfSquares, int Index)
{
    Array_XSquared[Index] = In[Index] * In[Index];

    MovingAverageCumulative_S(In, Array_Mean, Index);
    MovingAverageCumulative_S(Array_XSquared, Array_MeanOfSquares, Index);
}

```

```

float CumulativeVariance = Array_MeanOfSquares[Index] - Array_Mean[Index]*Array_Mean[Index];

Out[Index] = sqrt(CumulativeVariance);

return Out;
}

/*=====*/
SCFloatArrayRef CumulativeHurstExponent_S(SCFloatArrayRef In, SCFloatArrayRef Out, SCFloatArrayRef
Array_Average, SCFloatArrayRef Array_MeanAdjustedData, SCFloatArrayRef Array_CumulativeDeviation,
SCFloatArrayRef Array_MinMax, SCFloatArrayRef Array_StdDev, SCFloatArrayRef Array_RescaledRange,
SCFloatArrayRef Array_XSquared, SCFloatArrayRef Array_Mean, SCFloatArrayRef Array_MeanOfSquares,
SCFloatArrayRef Array_Sum_x, SCFloatArrayRef Array_Sum_x2, SCFloatArrayRef Array_Sum_x_2, SCFloatArrayRef
Array_Sum_y, SCFloatArrayRef Array_Sum_xy, int Index)
{

//Calculate the mean-adjusted data.
MovingAverageCumulative_S(In, Array_Average, Index);

Array_MeanAdjustedData[Index] = In[Index] - Array_Average[Index];

//Calculate the cumulative deviations Z_i.
if (Index == 0)
{
    Array_CumulativeDeviation[Index] = Array_MeanAdjustedData[Index];
    Array_MinMax[0] = FLT_MAX;//Minimum cumulative deviation value
    Array_MinMax[1] = -FLT_MAX;//Maximum cumulative deviation value
}
else
    Array_CumulativeDeviation[Index] = Array_MeanAdjustedData[Index] + Array_CumulativeDeviation[Index - 1];

if(Array_CumulativeDeviation[Index] > Array_MinMax[1])
{
    Array_MinMax[1]=Array_CumulativeDeviation[Index];
}

if(Array_CumulativeDeviation[Index] < Array_MinMax[0])
{
    Array_MinMax[0] = Array_CumulativeDeviation[Index];
}
//Calculate the ranges R_t = max{Z_0,Z_1,...,Z_t} - min{Z_0,Z_1,...,Z_t}.

float Range = Array_MinMax[1] - Array_MinMax[0];

//Calculate the standard deviations. THE INPUT NEEDS TO BE THE BASE DATA.
CumulativeStdDev_S(In, Array_StdDev, Array_XSquared, Array_Mean, Array_MeanOfSquares, Index);

//Calculate the rescaled ranges.
if (Array_StdDev[Index] != 0)
    Array_RescaledRange[Index] = Range / Array_StdDev[Index];

//Calculate the log-log slope, which is the Hurst exponent.
double slope = 0;
double y_intercept = 0;

if(Index >= 2) //The Hurst exponent is only defined when Index >= 2.
{
    CalculateCumulativeLogLogRegressionStatistics_S(Array_RescaledRange, slope, y_intercept, Array_Sum_x,
Array_Sum_x2, Array_Sum_x_2, Array_Sum_y, Array_Sum_xy, Index);
}

```

```

}

Out[Index] = static_cast<float>(slope);

return Out;
}

/*=====*/
void CalculateHurstExponent(SCFloatArrayRef In_X, SCFloatArrayRef In_Y, double &HurstExponent, int Index, int
Length)
{
    HurstExponent = 0.0;

    double sum_y = 0, sum_x = 0, sum_x2 = 0, sum_x_2 = 0, sum_xy = 0;

    if (Index < (Length - 1))
    {
        Index = Length - 1;
    }

    sum_x = GetSummation(In_X, Index, Length);
    sum_x_2 = sum_x * sum_x;
    sum_y = GetSummation(In_Y, Index, Length);
    for (int Offset = 0; Offset < Length; Offset++)
    {
        sum_x2 += In_X[Index - Offset] * In_X[Index - Offset];
        sum_xy += In_X[Index - Offset] * In_Y[Index - Offset];
    }

    double H_numerator = (Length * sum_xy - sum_x * sum_y);
    double H_denominator = Length * sum_x2 - sum_x_2;

    if (H_denominator != 0)
        HurstExponent = H_numerator / H_denominator;
}

/*=====*/
SCFloatArrayRef HurstExponentNew_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int LengthIndex)
{
    int CalculationStartIndex = 0;

    int ArrayIterations = 0;
    int CurrentSubsetLength = 8; // This is set to the initial subset length and is doubled with each iteration.
    int CurrentNumberOfSubsets = 0;
    if (LengthIndex == 0) // 32
    {
        if (Index < 31)
            return Out;

        CalculationStartIndex = Index - 31;

        ArrayIterations = 3;
        CurrentNumberOfSubsets = 32 / CurrentSubsetLength; // Starts with 4 subsets of 8, then 2 subsets of 16, then 1
subset of 32
    }
    else if (LengthIndex == 1) // 64
    {
        if (Index < 63)
            return Out;

        CalculationStartIndex = Index - 63;

        ArrayIterations = 4;
        CurrentNumberOfSubsets = 64 / CurrentSubsetLength; // 8
    }
}

```

```

else if (LengthIndex == 2)//128
{
    if (Index < 127)
        return Out;

    CalculationStartIndex = Index - 127;

    ArrayIterations = 5;
    CurrentNumberOfSubsets = 128 / CurrentSubsetLength;
}
else
    return Out;

float MeanAdjustedSeries[128] = {};
float CumulativeVariateSeries[128] = {};

//This is five pairs of values. Each pair is 'subsets per iteration' and 'rescaled range average'
float RescaledRangeAverageDataPairForIteration[5][2] = {};

for (int IterationIndex = 0; IterationIndex < ArrayIterations; IterationIndex++)
{
    float RescaledRangeSum = 0;
    for (int SubsetIndex = 0; SubsetIndex < CurrentNumberOfSubsets; SubsetIndex++)
    {
        const int SubsetStartBarIndex = CalculationStartIndex + (CurrentSubsetLength * SubsetIndex);

        float SumForMean = 0;
        for (int BarIndex = SubsetStartBarIndex; BarIndex < SubsetStartBarIndex + CurrentSubsetLength; BarIndex++)
        {
            SumForMean += ln[BarIndex];
        }

        float Mean = SumForMean / CurrentSubsetLength;

        int SeriesIndex = 0;
        for (int BarIndex = SubsetStartBarIndex; BarIndex < SubsetStartBarIndex + CurrentSubsetLength; BarIndex++)
        {
            MeanAdjustedSeries[SeriesIndex] = ln[BarIndex] - Mean;
            SeriesIndex++;
        }

        //calculate the cumulative variate series
        for (SeriesIndex = 0; SeriesIndex < CurrentSubsetLength; SeriesIndex++)
        {
            float SumForSeries = 0;
            for (int MeanAdjustedSeriesIndex = 0; MeanAdjustedSeriesIndex < SeriesIndex + 1;
MeanAdjustedSeriesIndex++)
            {
                SumForSeries += MeanAdjustedSeries[MeanAdjustedSeriesIndex];
            }

            CumulativeVariateSeries[SeriesIndex] = SumForSeries;
        }

        //calculate the range of the subset
        float MinValue;
        float MaxValue;
        GetMinMaxValuesFromArray(CumulativeVariateSeries, CurrentSubsetLength, MinValue, MaxValue);

        float Range = MaxValue - MinValue;

        float StandardDeviationOutput = 0;
        int StandardDeviationStartIndex = SubsetStartBarIndex + CurrentSubsetLength - 1;
        GetStandardDeviation(ln, StandardDeviationOutput, StandardDeviationStartIndex, CurrentSubsetLength);
    }
}

```

```

    float RescaledRange = 0;
    if(StandardDeviationOutput != 0)
        RescaledRange = Range / StandardDeviationOutput;

    RescaledRangeSum += RescaledRange;
}

float RescaledRangeAverage = RescaledRangeSum / CurrentNumberOfSubsets;
RescaledRangeAverageDataPairForIteration[IterationIndex][0] = static_cast<float>(CurrentSubsetLength);
RescaledRangeAverageDataPairForIteration[IterationIndex][1] = RescaledRangeAverage;

CurrentSubsetLength *= 2;
CurrentNumberOfSubsets /= 2;

}

float SubsetLengthLogArray[5] = {};
float RescaledRangeAverageLogArray[5] = {};
for (int IterationIndex = 0; IterationIndex < ArrayIterations; IterationIndex++)
{
    SubsetLengthLogArray[IterationIndex] = log(RescaledRangeAverageDataPairForIteration[IterationIndex][0]); //
    CurrentSubsetLength;

    RescaledRangeAverageLogArray[IterationIndex] = log(RescaledRangeAverageDataPairForIteration[IterationIndex]
    [1]); // RescaledRangeAverage;

}

SCFloatArray SubsetLengthLogSCFloatArray;
SubsetLengthLogSCFloatArray.InternalSetArray(SubsetLengthLogArray, ArrayIterations);

SCFloatArray RescaledRangeAverageLogArraySCFloatArray;
RescaledRangeAverageLogArraySCFloatArray.InternalSetArray(RescaledRangeAverageLogArray, ArrayIterations);

double HurstExponent;
CalculateHurstExponent(SubsetLengthLogSCFloatArray, RescaledRangeAverageLogArraySCFloatArray,
HurstExponent, ArrayIterations - 1, ArrayIterations);

if (isnan(HurstExponent))
{
    HurstExponent = 0;
}

Out[Index] = static_cast<float>(HurstExponent);

return Out;
}

/*=====*/
void GetMinMaxValuesFromArray(float* Array, int ArrayLength, float &MinValue, float &MaxValue)
{
    MinValue = FLT_MAX;
    MaxValue = -FLT_MAX;

    for (int Index = 0; Index < ArrayLength; Index++)
    {
        if (MinValue > Array[Index])
            MinValue = Array[Index];

        if (MaxValue < Array[Index])
            MaxValue = Array[Index];
    }
}

```

```

    }

}

/*=====*/
SCFloatArrayRef T3MovingAverage_S
( SCFloatArrayRef InputArray
, SCFloatArrayRef OutputArray
, SCFloatArrayRef CalcArray0
, SCFloatArrayRef CalcArray1
, SCFloatArrayRef CalcArray2
, SCFloatArrayRef CalcArray3
, SCFloatArrayRef CalcArray4
, SCFloatArrayRef CalcArray5
, float Multiplier
, int Index
, int Length)
{
    MovingAverage_S(InputArray, CalcArray0, MOVAVGTYPE_EXPONENTIAL, Index, Length);

    MovingAverage_S(CalcArray0, CalcArray1, MOVAVGTYPE_EXPONENTIAL, Index, Length);

    MovingAverage_S(CalcArray1, CalcArray2, MOVAVGTYPE_EXPONENTIAL, Index, Length);

    MovingAverage_S(CalcArray2, CalcArray3, MOVAVGTYPE_EXPONENTIAL, Index, Length);

    MovingAverage_S(CalcArray3, CalcArray4, MOVAVGTYPE_EXPONENTIAL, Index, Length);

    MovingAverage_S(CalcArray4, CalcArray5, MOVAVGTYPE_EXPONENTIAL, Index, Length);

    float Mult2 = Multiplier * Multiplier; // s^2

    float Mult3 = Mult2 * Multiplier; // s^3

    float c1 = -Mult3; // -s^3

    float c2 = 3 * Mult3 + 3 * Mult2; // 3s^3 + 3s^2

    float c3 = -3 * Mult3 - 6 * Mult2 - 3 * Multiplier; // -3s^3 - 6s^2 - 3s

    float c4 = Mult3 + 3 * Mult2 + 3 * Multiplier + 1; // s^3 + 3s^2 + 3s + 1

    OutputArray[Index] = c1*CalcArray5[Index] + c2*CalcArray4[Index] +
        c3*CalcArray3[Index] + c4*CalcArray2[Index];

    return OutputArray;
}

/*=====*/
SCFloatArrayRef ExampleFunction_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    return Out;
}

/*=====*/
SCFloatArrayRef ArnaudLegouxMovingAverage_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length, float
Sigma, float Offset)
{
    float Numerator = 0;
    float Denominator = 0;

    int OffsetTerm = static_cast<int>((Offset) * (Length - 1));
    float StdDev = static_cast<float>(Length) / Sigma;

```

```

for (int j = 0; j < Length; j++)
{
    Numerator += exp(-(j - OffsetTerm) * (j - OffsetTerm) / (2.0f * StdDev * StdDev)) * ln[Index - Length + 1 + j];
    Denominator += exp(-(j - OffsetTerm) * (j - OffsetTerm) / (2.0f * StdDev * StdDev));
}

Out[Index] = static_cast<float>(Numerator / Denominator);

return Out;
}

/*=====*/
SCFloatArrayRef ExponentialRegressionIndicator_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Index >= In.GetArraySize())
        return Out;

    double GrowthConstant = 0;
    double Coefficient = 0;

    double sum_logy = 0, sum_x = 0, sum_x2 = 0, sum_x_2 = 0, sum_xlogy = 0;

    if (Index < (Length - 1))
    {
        Index = Length - 1;
    }

    sum_x = static_cast<float>((Length * (Length + 1)) / 2.0);

    sum_x_2 = sum_x * sum_x;

    sum_x2 = (Length + 1) * Length * (2 * Length + 1) / 6.0f;

    for (int i = Index - Length + 1; i <= Index; i++)
    {
        if (ln[i] != 0)
        {
            sum_logy += log(ln[i]);

            sum_xlogy += (i - Index + Length) * log(ln[i]);
        }
    }

    double r_numerator = (Length * sum_xlogy - sum_x * sum_logy);
    double r_denominator = Length * sum_x2 - sum_x_2;

    GrowthConstant = r_numerator / r_denominator;
    Coefficient = exp((sum_logy - GrowthConstant * sum_x) / Length);

    Out[Index] = static_cast<float>(Coefficient * exp(GrowthConstant * Length));

    return Out;
}

/*=====*/
SCFloatArrayRef InstantaneousTrendline_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    float n = static_cast<float>(Length);

    float a = 2 / (n + 1);

    if (Index < 6)
        Out[Index] = 0.25f * (ln[Index] + 2 * ln[Index - 1] + ln[Index - 2]);

```

```

else
    Out[Index] = (a - 0.25f * a * a) * ln[Index] + 0.50f * a * a * ln[Index - 1] - (a - 0.75f * a * a) * ln[Index - 2] + 2 * (1.0f - a)
    * Out[Index - 1] - (1.0f - a) * (1.0f - a) * Out[Index - 2];

return Out;
}

/*=====*/
SCFloatArrayRef CyberCycle_S(SCFloatArrayRef In, SCFloatArrayRef Smoothed, SCFloatArrayRef Out, int Index, int
Length)
{
    float n = static_cast<float>(Length);

    float a = 2 / (n + 1);

    if (Index < 6)
        Out[Index] = 0.25f * (ln[Index] - 2 * ln[Index - 1] + ln[Index - 2]);
    else
        Out[Index] = (1 - 0.5f * a) * (1 - 0.5f * a) * (Smoothed[Index] - 2.0f * Smoothed[Index - 1] + Smoothed[Index - 2]) +
        2.0f * (1.0f - a) * Out[Index - 1] - (1.0f - a) * (1.0f - a) * Out[Index - 2];

    return Out;
}

/*=====*/
SCFloatArrayRef FourBarSymmetricalFIRFilter_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index)
{
    Out[Index] = (ln[Index] + 2.0f * ln[Index - 1] + 2.0f * ln[Index - 2] + ln[Index - 3]) / 6.0f;

    return Out;
}

/*=====*/
SCFloatArrayRef SuperSmoother2Pole_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Index < 2)
        Out[Index] = ln[Index];
    else
    {
        float arg = sqrt(2.0f) * static_cast<float>(M_PI) / Length;
        float a1 = exp(-1.0f * arg);
        float b1 = 2 * a1 * cos(arg);
        float k2 = b1;
        float k3 = -1.0f * a1 * a1;
        float k1 = 1.0f - k2 - k3;

        Out[Index] = k1 * ln[Index] + k2 * Out[Index - 1] + k3 * Out[Index - 2];
    }

    return Out;
}

/*=====*/
SCFloatArrayRef SuperSmoother3Pole_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Index < 3)
        Out[Index] = ln[Index];
    else
    {
        float arg = static_cast<float>(M_PI) / Length;
        float a1 = exp(-1.0f * arg);
        float b1 = 2 * a1 * cos(1.738f * arg);
        float c1 = a1 * a1;
        float k2 = b1 + c1;
        float k3 = -1.0f * (c1 + b1 * c1);
    }

```



```

float k4 = c1 * c1;
float k1 = 1.0f - k2 - k3 - k4;

Out[Index] = k1 * In[Index] + k2 * Out[Index - 1] + k3 * Out[Index - 2] + k4 * Out[Index - 3];
}

return Out;
}

/*=====*/
SCFloatArrayRef ZeroLagEMA_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    int Lag = static_cast<int>(round((Length - 1) / 2.0f));

    double Multiplier1 = 2.0f / (Length + 1);
    double Multiplier2 = 1.0f - Multiplier1;
    double DeLaggedData = 2 * In[Index] - In[Index - Lag];

    if(Index == 0)
        Out[Index] = 2 * In[Index] - In[Index - Lag];
    else
        Out[Index] = static_cast<float>((Multiplier1 * DeLaggedData) + (Multiplier2 * Out[Index - 1]));

    return Out;
}

/*=====*/
SCFloatArrayRef Butterworth2Pole_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Index < 2)
        Out[Index] = In[Index];
    else
    {
        float arg = sqrt(2.0f) * static_cast<float>(M_PI) / Length;
        float a1 = exp(-1.0f * arg);
        float b1 = 2.0f * a1 * cos(arg);
        float k2 = b1;
        float k3 = -1.0f * a1 * a1;
        float k1 = (1.0f - k2 - k3)/4.0f;

        Out[Index] = k1 * (In[Index] + 2.0f*In[Index - 1] + In[Index - 2]) + k2 * Out[Index - 1] + k3 * Out[Index - 2];
    }

    return Out;
}

/*=====*/
SCFloatArrayRef Butterworth3Pole_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Index < 3)
        Out[Index] = In[Index];
    else
    {
        float arg = static_cast<float>(M_PI) / Length;
        float a1 = exp(-1.0f * arg);
        float b1 = 2.0f * a1 * cos(1.738f * arg);
        float c1 = a1 * a1;
        float k2 = b1 + c1;
        float k3 = -1.0f * (c1 + b1 * c1);
        float k4 = c1 * c1;
        float k1 = (1 - b1 + c1)*(1 - c1)/8.0f;

        Out[Index] = k1 * (In[Index] + 3.0f*In[Index - 1] + 3.0f*In[Index - 2] + In[Index - 3]) + k2 * Out[Index - 1] + k3 *
Out[Index - 2] + k4 * Out[Index - 3];

```

```

}

return Out;
}

/*=====*/
SCFloatArrayRef DominantCyclePeriod_S(SCFloatArrayRef In, SCFloatArrayRef InstPeriod, SCFloatArrayRef Q1,
SCFloatArrayRef I1, SCFloatArrayRef PhaseChange, SCFloatArrayRef Temp, SCFloatArrayRef MedianPhaseChange,
SCFloatArrayRef DominantCycle, SCFloatArrayRef Out, int Index, int MedianLength)
{
    //Initialize Instantaneous Period and Cycle Period
    if (Index == 0)
    {
        InstPeriod[Index] = 0.0f;
        Out[Index] = 0.0f;
    }

    // Compute Quadrature and In Phase Components.
    Q1[Index] = (0.0962f * In[Index] + 0.5769f * In[Index - 2] - 0.5769f * In[Index - 4] - 0.0962f * In[Index - 6])*(0.5f +
0.08f*InstPeriod[Index - 1]);
    I1[Index] = In[Index - 3];

    // Compute Phase Change.
    float PhaseChangeRaw;

    if (Q1[Index] != 0.0f && Q1[Index - 1] != 0.0f)
        PhaseChangeRaw = (I1[Index] / Q1[Index] - I1[Index - 1] / Q1[Index - 1]) / (1 + (I1[Index] * I1[Index - 1]) / (Q1[Index]
* Q1[Index - 1]));
    else
        PhaseChangeRaw = 0.0f;

    if (PhaseChangeRaw > 1.1f)
        PhaseChange[Index] = 1.1f;
    else if (PhaseChangeRaw < 0.1f)
        PhaseChange[Index] = 0.1f;
    else
        PhaseChange[Index] = PhaseChangeRaw;

    MovingMedian_S(PhaseChange, MedianPhaseChange, Temp, Index, MedianLength);

    if (MedianPhaseChange[Index] == 0)
        DominantCycle[Index] = 15.0f;
    else
    {
        DominantCycle[Index]
            = static_cast<float>(2.0f * M_PI / MedianPhaseChange[Index] + 0.5f);
    }

    InstPeriod[Index] = 0.33f * DominantCycle[Index] + 0.67f * InstPeriod[Index - 1];

    Out[Index] = 0.15f * InstPeriod[Index] + 0.85f * Out[Index - 1];

    return Out;
}

/*=====*/
SCFloatArrayRef LaguerreFilter_S(SCFloatArrayRef In, SCFloatArrayRef L0, SCFloatArrayRef L1, SCFloatArrayRef L2,
SCFloatArrayRef L3, SCFloatArrayRef Out, int Index, float DampingFactor)
{
    if (Index == 0)
    {
        L0[Index] = (1.0f - DampingFactor) * In[Index];
        L1[Index] = -1.0f * DampingFactor * L0[Index];
        L2[Index] = -1.0f * DampingFactor * L1[Index];
        L3[Index] = -1.0f * DampingFactor * L2[Index];
    }

```

```

    }
    else
    {
        L0[Index] = (1.0f - DampingFactor) * In[Index] + DampingFactor * L0[Index - 1];
        L1[Index] = -1.0f * DampingFactor * L0[Index] + L0[Index - 1] + DampingFactor * L1[Index - 1];
        L2[Index] = -1.0f * DampingFactor * L1[Index] + L1[Index - 1] + DampingFactor * L2[Index - 1];
        L3[Index] = -1.0f * DampingFactor * L2[Index] + L2[Index - 1] + DampingFactor * L3[Index - 1];
    }

    Out[Index] = (L0[Index] + 2.0f * L1[Index] + 2.0f * L2[Index] + L3[Index]) / 6.0f;

    return Out;
}

/*=====*/
SCFloatArrayRef DominantCyclePhase_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index)
{
    //Declare Period and Real and Imaginary Parts of Dominant Cycle Phase
    int DCPeriod = static_cast<int>(In[Index]);
    float Re = 0.0f;
    float Im = 0.0f;

    for (int i = 0; i <= DCPeriod - 1; i++)
    {
        float angle;

        if (DCPeriod == 0)
            angle = 0;
        else
            angle = static_cast<float>((360.0f * i / DCPeriod) * (M_PI/180));

        Re += sin(angle) * In[Index - i];
        Im += cos(angle) * In[Index - i];
    }

    if (fabs(Im) > 0.001)
        Out[Index] = static_cast<float>(atan(Re / Im) * (180 / M_PI) + 90);
    else
    {
        if (Re == 0)
            Out[Index] = 0.0f;
        else
            Out[Index] = 90 * fabs(Re) / Re + 90;
    }

    if (Im < 0)
        Out[Index] += 180;
    if (Im > 315)
        Out[Index] -= 360;

    return Out;
}

/*=====*/
SCFloatArrayRef LinearRegressionSlope_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Index >= In.GetArraySize())
        return Out;

    double Slope = 0;
    double Y_Intercept = 0;

    CalculateRegressionStatistics(In, Slope, Y_Intercept, Index, Length);

    Out[Index] = (float)Slope;

```

```

    return Out;
}

/*=====*/
SCFloatArrayRef LinearRegressionIntercept_S(SCFloatArrayRef In, SCFloatArrayRef Out, int Index, int Length)
{
    if (Index >= In.GetArraySize())
        return Out;

    double Slope = 0;
    double Y_Intercept = 0;

    CalculateRegressionStatistics(In, Slope, Y_Intercept, Index, Length);

    Out[Index] = (float)Y_Intercept;

    return Out;
}

```