

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

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

```
struct s_LevelLineConfig
```

```
{
    SCDatetime Date;
    SCString Label;
    uint32_t Color = 0;
    uint32_t Width = 0;
    float Value1 = 0;
    float Value2 = 0;
```

```
void Clear()
```

```
{
    Date.Clear();
    Label.Clear();
    Color = 0;
    Width = 0;
    Value1 = 0;
    Value2 = 0;
}
};
```

```
enum TradingLevelsDataLineTypeEnum : int//must be an integer
```

```
{
    DataLineTypeDateRepeatingValues = 0
    , DataLineTypeDateFormattingValue = 1
    , DataLineTypeDateFormattingTopBottomValues = 2
};
```

```
const char* HTTP_NO_DATA_RESPONSE = "NO_DATA";
```

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

```
int TradingLevelsStudy_RequestValuesFromServer
```

```
( SCStudyInterfaceRef sc
, const SCString& BaseWebsiteURL
, int& r_RequestState
, const char* AlternateSymbol
);
```

```
void TradingLevelsStudy_ResetStateForNextRequest
```

```
( SCStudyInterfaceRef sc
, int& r_RequestState
, SCDatetime& r_RequestDateTime
, std::vector<SCString>* p_ValuesLineForDates
, std::vector<std::vector<char*>>* p_PointersToValuesForDates
);
```

```
unsigned int TradingLevelsStudy_GetValuesForDate
```

```
( SCStudyInterfaceRef sc
, const SCDatetime& BarDate
, float* Values
, unsigned int ValuesArraySize
, std::vector<std::vector<char*>>* p_PointersToValuesForDates
, int& r_LastFoundIndex
);
```

```
unsigned int TradingLevelsStudy_GetFormattedValuesForDate
```

```
(SCStudyInterfaceRef sc
, const SCDatetime& BarDate
, s_LevelLineConfig* FormattedLevelValues
, unsigned int ArraySize
```

```

, std::vector<std::vector<char*>>* p_PointersToValuesForDates
, int& r_LastFoundIndex
);

```

```

void TradingLevelsStudyCore
( SCStudyInterfaceRef sc
, const SCString& RequestURL
, const int RequestIntervallInMinutes
, std::vector<SCString>* p_ValuesLineForDates
, std::vector<std::vector<char*>>* p_PointersToValuesForDates
, int& r_RequestState
, SCDateTime& r_RequestDateTime
, int& r_ClearAtMidnight
, bool AllowRequestLevelsFromServer
, int& r_AwaitingNextRequestForLevels
, const char* InAlternateSymbol
, int& r_LastFoundLevelsArrayIndex
, int& r_DataLineType
);

```

```

/*=====*/

```

```

SCSFExport scsf_TradingLevelsStudy(SCStudyInterfaceRef sc)
{

```

```

    SCInputRef Input_InVersion = sc.Input[0];
    SCInputRef Input_InRequestIntervallInMinutes = sc.Input[1];
    SCInputRef Input_InUseTimeRangeForLevelsRequest = sc.Input[2];
    SCInputRef Input_InTimeRangeTimeZone = sc.Input[3];
    SCInputRef Input_InRequestStartTime = sc.Input[4];
    SCInputRef Input_InRequestEndTime = sc.Input[5];
    SCInputRef Input_InAlternateSymbol = sc.Input[6];

```

```

    const SCString RequestURL = "https://www.sierrachart.com/API.php?Service=PriceLevels";

```

```

    if (sc.SetDefaults)
    {

```

```

        sc.GraphName = "Trading Levels";

```

```

        sc.GraphRegion = 0;
        sc.ScaleRangeType = SCALE_SAMEASREGION;
        sc.ValueFormat = VALUEFORMAT_INHERITED;
        sc.AutoLoop = 0;

```

```

        sc.StudyDescription = "Requests line level values from Web server in the format 'YYYY-MM-DD, [level_value_1], [level_value_2], [level_value_3], [level_value_4], and so on, where the level_value_# is equal to the value of the line. There can be up to SC_SUBGRAPHS_AVAILABLE level values.";

```

```

        for (int Index = 0; Index < SC_SUBGRAPHS_AVAILABLE; Index++)
        {
            SCString SubgraphName;
            SubgraphName.Format("Level %d", Index+1);
            sc.Subgraph[Index].Name = SubgraphName;
            sc.Subgraph[Index].DrawStyle = DRAWSTYLE_DASH;
            sc.Subgraph[Index].LineWidth = 2;
            sc.Subgraph[Index].PrimaryColor = RGB(0, 255, 0);
            sc.Subgraph[Index].DrawZeros = 0; // false
            sc.Subgraph[Index].LineLabel = LL_DISPLAY_VALUE | LL_VALUE_ALIGN_CENTER |
LL_VALUE_ALIGN_VALUES_SCALE;
        }

```

```

        Input_InRequestIntervallInMinutes.Name = "Request Interval in Minutes";
        Input_InRequestIntervallInMinutes.SetInt(60);
        Input_InRequestIntervallInMinutes.SetIntLimits(1, MINUTES_PER_DAY);

```

```

        Input_InUseTimeRangeForLevelsRequest.Name = "Use Time Range For Levels Request";

```

```

Input_InUseTimeRangeForLevelsRequest.SetYesNo(false);

Input_InTimeRangeTimeZone.Name = "Time Range Time Zone";
Input_InTimeRangeTimeZone.SetTimeZone(TIMEZONE_NEW_YORK);

Input_InRequestStartTime.Name = "Request Start Time";
Input_InRequestStartTime.SetTime(0);

Input_InRequestEndTime.Name = "Request End Time";
Input_InRequestEndTime.SetTime(SECONDS_PER_DAY - 1);

Input_InAlternateSymbol.Name = "Alternate Symbol";
Input_InAlternateSymbol.SetString("");

sc.TextInputName = "Request Identifier";

return;
}

std::vector<SCString>* p_ValuesLineForDates = reinterpret_cast<std::vector<SCString>*>(sc.GetPersistentPointer(1));

std::vector<std::vector<char*>>* p_PointersToValuesForDates = reinterpret_cast<std::vector<std::vector<char*>>*>(
sc.GetPersistentPointer(2));

int& r_RequestState = sc.GetPersistentInt(1);
int& r_ClearAtMidnight = sc.GetPersistentInt(2);
int& r_AwaitingNextRequestForLevels = sc.GetPersistentInt(3);
int& r_LastFoundLevelsArrayIndex = sc.GetPersistentInt(4);
int& r_DataLineType = sc.GetPersistentInt(5);

SCDateTime& r_RequestDateTime = sc.GetPersistentSCDateTime(2);

SCDateTime CurrentDateTimeInRequestTimeZone;

bool AllowRequestLevelsFromServer = true;

if (Input_InUseTimeRangeForLevelsRequest.GetYesNo())
{
    CurrentDateTimeInRequestTimeZone = sc.CurrentSystemDateTime;
    CurrentDateTimeInRequestTimeZone =
sc.ConvertDateTimeFromChartTimeZone(CurrentDateTimeInRequestTimeZone,
Input_InTimeRangeTimeZone.GetTimeZone());

    if (CurrentDateTimeInRequestTimeZone.GetTimeInSeconds() < Input_InRequestStartTime.GetTime()
        || CurrentDateTimeInRequestTimeZone.GetTimeInSeconds() > Input_InRequestEndTime.GetTime())
        AllowRequestLevelsFromServer = false;
}

if (sc.IsFullRecalculation)
{
    r_LastFoundLevelsArrayIndex = 0;
    r_DataLineType = DataLineTypeDateRepeatingValues;
}

TradingLevelsStudyCore
(
    sc
    , RequestURL
    , Input_InRequestIntervallInMinutes.GetInt()
    , p_ValuesLineForDates
    , p_PointersToValuesForDates
    , r_RequestState
    , r_RequestDateTime
    , r_ClearAtMidnight
    , AllowRequestLevelsFromServer

```

```

, r_AwaitingNextRequestForLevels
, Input_InAlternateSymbol.GetString()
, r_LastFoundLevelsArrayIndex
, r_DataLineType
);
}

```

```

/*=====*/

```

```

void TradingLevelsStudyCore
( SCStudyInterfaceRef sc
, const SCString& RequestURL
, const int RequestIntervallInMinutes
, std::vector<SCString>* p_ValuesLineForDates
, std::vector<std::vector<char*>>* p_PointersToValuesForDates
, int& r_RequestState
, SCDateTime& r_RequestDateTime
, int& r_ClearAtMidnight
, bool AllowRequestLevelsFromServer
, int& r_AwaitingNextRequestForLevels
, const char* InAlternateSymbol
, int& r_LastFoundLevelsArrayIndex
, int& r_DataLineType
)
{
    if(sc.LastCallToFunction)
    {
        if(p_ValuesLineForDates != NULL)
        {
            delete p_ValuesLineForDates;
            sc.SetPersistentPointer(1, NULL);
        }

        if(p_PointersToValuesForDates != NULL)
        {
            delete p_PointersToValuesForDates;
            sc.SetPersistentPointer(2, NULL);
        }

        return;
    }

    if (p_ValuesLineForDates == NULL)
    {
        p_ValuesLineForDates = new std::vector<SCString>;

        if(p_ValuesLineForDates != NULL)
            sc.SetPersistentPointer(1, p_ValuesLineForDates);
        else
        {
            sc.AddMessageToLog("Memory allocation error.", 1);
            return;
        }
    }

    if (p_PointersToValuesForDates == NULL)
    {
        p_PointersToValuesForDates = new std::vector<std::vector< char*> >;

        if(p_PointersToValuesForDates != NULL)
            sc.SetPersistentPointer(2, p_PointersToValuesForDates);
        else
        {
            sc.AddMessageToLog("Memory allocation error.", 1);
            return;
        }
    }
}

```

```

}

if (sc.IsFullRecalculation)
    r_ClearAtMidnight = false;

if (AllowRequestLevelsFromServer)
{
    //Request data on a full recalculation and also at the specified interval

    if (sc.UpdateStartIndex == 0 && r_RequestState == HTTP_REQUEST_RECEIVED)
    {
        TradingLevelsStudy_ResetStateForNextRequest(sc, r_RequestState, r_RequestDateTime,
p_ValuesLineForDates, p_PointersToValuesForDates);
    }
    else if (r_RequestDateTime.IsUnset()
        || ( (sc.CurrentSystemDateTime - r_RequestDateTime)
            >= SCDatetime::MINUTES(RequestIntervalInMinutes))
        )// Request interval has elapsed
    {
        TradingLevelsStudy_ResetStateForNextRequest(sc, r_RequestState, r_RequestDateTime,
p_ValuesLineForDates, p_PointersToValuesForDates);
    }

    if (TradingLevelsStudy_RequestValuesFromServer(sc, RequestURL, r_RequestState, InAlternateSymbol))
    {
        if (r_RequestState == HTTP_REQUEST_MADE)
            r_AwaitingNextRequestForLevels = false;

        return;//Return here since we need to wait for the response
    }
}

if (sc.HTTPRequestID != 0)//response received
{
    r_RequestState = HTTP_REQUEST_RECEIVED;

    if (sc.HTTPResponse == ACSIL_HTTP_REQUEST_ERROR_TEXT
        || sc.HTTPResponse == ACSIL_HTTP_EMPTY_RESPONSE_TEXT)
    {
        sc.AddMessageToLog("There was an error requesting data from the server.", true);
    }

    if (sc.UpdateStartIndex == 0 && sc.HTTPResponse == HTTP_NO_DATA_RESPONSE)
        sc.AddMessageToLog("There are no price levels for the given parameters.", false);

    if (strstr(sc.HTTPResponse.GetChars(), "CLEAR_AT_MIDNIGHT") != NULL)
    {
        sc.AddMessageToLog("Received clear at midnight command.", false);
        r_ClearAtMidnight = true;
    }
}

if (r_RequestState != HTTP_REQUEST_RECEIVED)
    return;

SCDateTime StartIndexDate = sc.BaseDateTimeIn[sc.UpdateStartIndex].GetDate();
SCDateTime LastBarDate = sc.BaseDateTimeIn[sc.ArraySize - 1].GetDate();

if (r_ClearAtMidnight && StartIndexDate != LastBarDate)
{
    //clear existing levels
    p_ValuesLineForDates->clear();
    p_PointersToValuesForDates->clear();
}

```

```

for (int BarIndex = sc.ArraySize - 1; BarIndex >= 0; --BarIndex)
{
    for (unsigned int LevelIndex = 0; LevelIndex < SC_SUBGRAPHS_AVAILABLE; LevelIndex++)
    {
        sc.Subgraph[LevelIndex][BarIndex] = 0;
    }
}

r_ClearAtMidnight = false; //Prevent this from happening on next calculation.
r_AwaitingNextRequestForLevels = true;

}

if (r_AwaitingNextRequestForLevels)
    return;

if (sc.HTTPResponse == HTTP_NO_DATA_RESPONSE)
    return;

bool FullRecalculate = false;

if (p_ValuesLineForDates->empty())
{
    r_LastFoundLevelsArrayIndex = 0;

    SCString DataLineTypeDateFormattingValueString("Date, Label, Color, Width, Value");
    SCString DataLineTypeDateFormattingTopBottomValuesString("Date, Label, Color, Width, TopValue,
BottomValue");

    if (sc.HTTPResponse.GetSubString(DataLineTypeDateFormattingValueString.GetLength(), 0)
        == DataLineTypeDateFormattingValueString)
    {
        r_DataLineType = DataLineTypeDateFormattingValue;
    }
    else if (sc.HTTPResponse.GetSubString(DataLineTypeDateFormattingTopBottomValuesString.GetLength(), 0)
        == DataLineTypeDateFormattingTopBottomValuesString)
    {
        r_DataLineType = DataLineTypeDateFormattingTopBottomValues;
    }

    sc.HTTPResponse.ParseLines(*p_ValuesLineForDates);

    SCString Message;
    Message.Format("Received %d price level lines of data from server.", p_ValuesLineForDates->size());
    sc.AddMessageToLog(Message, false);

    std::vector<char*> EmptyVector;

    for (int Index = 0; Index < static_cast<int>(p_ValuesLineForDates->size()); Index++)
    {
        p_PointersToValuesForDates->push_back(EmptyVector);
        p_ValuesLineForDates->at(Index).Tokenize(" ", p_PointersToValuesForDates->back());
    }

    FullRecalculate = true;
}

if (p_PointersToValuesForDates->empty())
    return;

if (FullRecalculate)
    sc.UpdateStartIndex = 0;

float LevelValues[SC_SUBGRAPHS_AVAILABLE] = {};

```

```

s_LevelLineConfig FormattedLevelValues[SC_SUBGRAPHS_AVAILABLE];

SCDateTime PriorDate;

uint32_t MaximumLevelsUsedPerDay = 0;
uint32_t NumberUsedLevels = 0;

for (int BarIndex = sc.UpdateStartIndex; BarIndex < sc.ArraySize; BarIndex++)
{
    SCDateTime BarIndexDate = sc.GetTradingDayDate(sc.BaseDateTimeln[BarIndex]);

    if (r_DataLineType == DataLineTypeDateRepeatingValues)
    {
        if (PriorDate != BarIndexDate)
        {
            NumberUsedLevels = TradingLevelsStudy_GetValuesForDate(sc, BarIndexDate, LevelValues,
SC_SUBGRAPHS_AVAILABLE, p_PointersToValuesForDates, r_LastFoundLevelsArrayIndex);

            if (NumberUsedLevels > MaximumLevelsUsedPerDay)
                MaximumLevelsUsedPerDay = NumberUsedLevels;

            PriorDate = BarIndexDate;
        }

        for (uint32_t LevelIndex = 0; LevelIndex < NumberUsedLevels; LevelIndex++)
            sc.Subgraph[LevelIndex][BarIndex] = LevelValues[LevelIndex];
    }
    else if (r_DataLineType == DataLineTypeDateFormattingValue)
    {
        if (PriorDate != BarIndexDate)
        {
            NumberUsedLevels = TradingLevelsStudy_GetFormattedValuesForDate(sc, BarIndexDate,
FormattedLevelValues, SC_SUBGRAPHS_AVAILABLE, p_PointersToValuesForDates, r_LastFoundLevelsArrayIndex);

            for (uint32_t LevelIndex = 0; LevelIndex < NumberUsedLevels; LevelIndex++)
            {
                sc.Subgraph[LevelIndex].Name = FormattedLevelValues[LevelIndex].Label;
                sc.Subgraph[LevelIndex].LineWidth = FormattedLevelValues[LevelIndex].Width;
                sc.Subgraph[LevelIndex].PrimaryColor = FormattedLevelValues[LevelIndex].Color;
            }

            if (NumberUsedLevels > MaximumLevelsUsedPerDay)
                MaximumLevelsUsedPerDay = NumberUsedLevels;

            PriorDate = BarIndexDate;
        }

        for (uint32_t LevelIndex = 0; LevelIndex < NumberUsedLevels; LevelIndex++)
            sc.Subgraph[LevelIndex][BarIndex] = FormattedLevelValues[LevelIndex].Value1;
    }
    else if (r_DataLineType == DataLineTypeDateFormattingTopBottomValues)
    {
        if (PriorDate != BarIndexDate)
        {
            NumberUsedLevels = TradingLevelsStudy_GetFormattedValuesForDate(sc, BarIndexDate,
FormattedLevelValues, SC_SUBGRAPHS_AVAILABLE, p_PointersToValuesForDates, r_LastFoundLevelsArrayIndex);

            if (NumberUsedLevels > (SC_SUBGRAPHS_AVAILABLE / 2))
                NumberUsedLevels = (SC_SUBGRAPHS_AVAILABLE / 2);

            for (uint32_t LevelIndex = 0; LevelIndex < NumberUsedLevels; LevelIndex++)

```

```

{
    int SubgraphIndex = LevelIndex * 2;
    sc.Subgraph[SubgraphIndex].Name = FormattedLevelValues[LevelIndex].Label;
    sc.Subgraph[SubgraphIndex].Name += " Top";
    sc.Subgraph[SubgraphIndex].LineWidth = FormattedLevelValues[LevelIndex].Width;
    sc.Subgraph[SubgraphIndex].PrimaryColor = FormattedLevelValues[LevelIndex].Color;

    uint16_t& r_DrawStyleTop = sc.Subgraph[SubgraphIndex].DrawStyle;
    uint16_t& r_DrawStyleBottom = sc.Subgraph[SubgraphIndex + 1].DrawStyle;

    if (r_DrawStyleTop != DRAWSTYLE_FILL_RECTANGLE_TOP &&
        r_DrawStyleTop != DRAWSTYLE_TRANSPARENT_FILL_RECTANGLE_TOP)
    {
        r_DrawStyleTop = DRAWSTYLE_TRANSPARENT_FILL_RECTANGLE_TOP;
        r_DrawStyleBottom = DRAWSTYLE_TRANSPARENT_FILL_RECTANGLE_BOTTOM;
    }

    if (r_DrawStyleTop == DRAWSTYLE_FILL_RECTANGLE_TOP)
        r_DrawStyleBottom = DRAWSTYLE_FILL_RECTANGLE_BOTTOM;
    else if (r_DrawStyleTop == DRAWSTYLE_TRANSPARENT_FILL_RECTANGLE_TOP)
        r_DrawStyleBottom = DRAWSTYLE_TRANSPARENT_FILL_RECTANGLE_BOTTOM;

    sc.Subgraph[SubgraphIndex + 1].Name = FormattedLevelValues[LevelIndex].Label;
    sc.Subgraph[SubgraphIndex + 1].Name += " Bottom";
    sc.Subgraph[SubgraphIndex + 1].LineWidth = FormattedLevelValues[LevelIndex].Width;
    sc.Subgraph[SubgraphIndex + 1].PrimaryColor = FormattedLevelValues[LevelIndex].Color;

}

//if (NumberUsedLevels > MaximumLevelsUsedPerDay)
// MaximumLevelsUsedPerDay = NumberUsedLevels;

PriorDate = BarIndexDate;
}

for (uint32_t LevelIndex = 0; LevelIndex < NumberUsedLevels; LevelIndex++)
{
    int SubgraphIndex = LevelIndex * 2;
    sc.Subgraph[SubgraphIndex][BarIndex] = FormattedLevelValues[LevelIndex].Value1;
    sc.Subgraph[SubgraphIndex + 1][BarIndex] = FormattedLevelValues[LevelIndex].Value2;
}
}
}

if (FullRecalculate && r_DataLineType != DataLineTypeDateFormattingTopBottomValues)
{
    for (uint32_t Index = 0; Index < SC_SUBGRAPHS_AVAILABLE; Index++)
    {
        if (Index < MaximumLevelsUsedPerDay
            && !sc.IsVisibleSubgraphDrawStyle(sc.Subgraph[Index].DrawStyle))
        {
            sc.Subgraph[Index].DrawStyle = DRAWSTYLE_DASH;
            sc.Subgraph[Index].DisplayNameValueInDataLine = true;
            sc.Subgraph[Index].DisplayNameValueInWindowsFlags = true;
        }
        else if (Index >= MaximumLevelsUsedPerDay)
        {
            sc.Subgraph[Index].DrawStyle = DRAWSTYLE_IGNORE;
            sc.Subgraph[Index].DisplayNameValueInDataLine = false;
            sc.Subgraph[Index].DisplayNameValueInWindowsFlags = false;
        }
    }
}
}
}

```



```

/*=====*/
//Returns 1 if request has been made. Returns 0 if request has not been made.
int TradingLevelsStudy_RequestValuesFromServer
( SCStudyInterfaceRef sc
, const SCString& BaseWebsiteURL
, int& r_RequestState
, const char* AlternateSymbol
)
{
    SCString FullURL;

    if (r_RequestState != HTTP_REQUEST_NOT_SENT)
        return 0;

    if (sc.TextInput.IsEmpty())
        return 0;

    const char* RequestSymbol = NULL;

    if (AlternateSymbol != NULL && AlternateSymbol[0] != '\0')
        RequestSymbol = AlternateSymbol;
    else
        RequestSymbol = sc.Symbol.GetChars();

    FullURL.Format
    ("%s&Username=%s&Symbol=%s&SCDLLName=%s"
    , BaseWebsiteURL.GetChars()
    , sc.UserName().GetChars()
    , RequestSymbol
    , sc.TextInput.GetChars()
    );

    if (!sc.MakeHTTPRequest(FullURL))
    {
        sc.AddMessageToLog("Error making HTTP Request.", true);
        r_RequestState = HTTP_REQUEST_ERROR;
    }
    else
    {
        r_RequestState = HTTP_REQUEST_MADE;
        SCString LogMessage("Requesting data from Trading Levels server: ");
        LogMessage += FullURL;
        sc.AddMessageToLog(LogMessage, false);
    }

    return 1;
}

/*=====*/
void TradingLevelsStudy_ResetStateForNextRequest
( SCStudyInterfaceRef sc
, int& r_RequestState
, SCDateTime& r_RequestDateTime
, std::vector<SCString>* p_ValuesLineForDates
, std::vector<std::vector<char*>>* p_PointersToValuesForDates
)
{
    p_ValuesLineForDates->clear();
    p_PointersToValuesForDates->clear();

    r_RequestState = HTTP_REQUEST_NOT_SENT;
    r_RequestDateTime = sc.CurrentSystemDateTime;
}

```

```

}
/*=====*/
unsigned int TradingLevelsStudy_GetValuesForDate
(SCStudyInterfaceRef sc
, const SCDateTime& BarDate
, float* Values
, unsigned int ValuesArraySize
, std::vector<std::vector<char*>>* p_PointersToValuesForDates
, int& r_LastFoundIndex
)
{
    //Clean array
    memset(Values, 0, sizeof(float) * ValuesArraySize);

    if (p_PointersToValuesForDates->empty())
        return 0;

    unsigned int NumberLevels = 0;
    const int EndIndex = static_cast<int>(p_PointersToValuesForDates->size());

    for(; r_LastFoundIndex < EndIndex; r_LastFoundIndex++)
    {
        std::vector<char*>& Fields = p_PointersToValuesForDates->at(r_LastFoundIndex);

        if (Fields.empty())
            continue;

        SCDateTime SourceDataLineDate = sc.DateStringToSCDateTime(Fields[0]);

        if (SourceDataLineDate < BarDate)
            continue;

        if (SourceDataLineDate > BarDate)
            return NumberLevels;//r_LastFoundIndex will be used on the next entry and continue from the correct index.

        int FieldsSize = static_cast<unsigned int>(Fields.size());

        for (int ItemIndex = 1; ItemIndex < FieldsSize; ItemIndex++)
        {
            Values[NumberLevels] = static_cast<float>(sc.StringToDouble(Fields[ItemIndex]));
            NumberLevels++;

            if (NumberLevels >= ValuesArraySize)
                break;
        }

        break;
    }

    return NumberLevels;
}
/*=====*/
unsigned int TradingLevelsStudy_GetFormattedValuesForDate
(SCStudyInterfaceRef sc
, const SCDateTime& BarDate
, s_LevelLineConfig* FormattedLevelValues
, unsigned int ArraySize
, std::vector<std::vector<char*>>* p_PointersToValuesForDates
, int& r_LastFoundIndex
)
{
    //Clean array
    for (uint32_t Index = 0; Index < ArraySize; Index++)
    {

```

```

    FormattedLevelValues[Index].Clear();
}

if (p_PointersToValuesForDates->empty())
    return 0;

unsigned int LevelIndex = 0;
const int EndIndex = static_cast<int>(p_PointersToValuesForDates->size());
for (; r_LastFoundIndex < EndIndex; r_LastFoundIndex++)
{
    std::vector<char*> Fields = p_PointersToValuesForDates->at(r_LastFoundIndex);

    if (Fields.empty())
        continue;

    //"Date, Label, Color, Width, Value"
    //"Date, Label, Color, Width, TopValue, BottomValue"
    SCDateTime SourceDataLineDate = sc.DateStringToSCDateTime(Fields[0]);

    if (SourceDataLineDate < BarDate)
        continue;

    if (SourceDataLineDate > BarDate)
        return LevelIndex; //r_LastFoundIndex will be used on the next entry and continue from the correct index.

    if (Fields.size() < 6)
        continue;

    FormattedLevelValues[LevelIndex].Date = SourceDataLineDate;
    FormattedLevelValues[LevelIndex].Label = Fields[1];
    FormattedLevelValues[LevelIndex].Color = atoi(Fields[2]);
    const int ColorValue = FormattedLevelValues[LevelIndex].Color;
    uint8_t Red = (ColorValue & 0xff0000) >> 16;
    uint8_t Green = (ColorValue & 0xff00) >> 8;
    uint8_t Blue = ColorValue & 0xff;
    FormattedLevelValues[LevelIndex].Color = RGB(Red, Green, Blue);

    FormattedLevelValues[LevelIndex].Width = atoi(Fields[3]);
    FormattedLevelValues[LevelIndex].Value1 = static_cast<float>(atof(Fields[4]));

    if (Fields.size() >= 6)
        FormattedLevelValues[LevelIndex].Value2 = static_cast<float>(atof(Fields[5]));

    LevelIndex++;

    if (LevelIndex >= ArraySize)
        break;
}

return LevelIndex; //Num levels
}

```