



MultiCharts, LLC

MultiCharts .NET Programming Guide

Version 1.2

5/8/2015

Contents

1	Basic Skills.....	4
2	Basic Definitions	4
2.1	Indicator	4
2.2	Signal/Strategy	5
2.3	Inputs.....	6
3	Tutorial.....	7
3.1	How to Create a C# Indicator	7
3.2	How to Create a C# Strategy	11
3.3	Debugging	13
4	Understanding PowerLanguage .NET	14
4.1	How Indicators and Signals are Calculated	14
4.2	Data Access	19
4.3	Plotting on the Chart	23
4.3.1	Plot and PlotPaintBar	23
4.3.2	Drawings.....	28
4.4	Output, Alerts and Expert Commentary.	30
4.4.1	Output	30
4.4.2	Alerts	31
4.4.3	Expert Commentary	33
4.5	Functions and Special Variables	35
4.5.1	Functions	35
4.5.2	Variables.....	35
4.5.3	Barsback	36
4.6	Strategies.....	38
4.6.1	Orders.....	38

4.6.2	Special Orders	41
4.6.3	Strategy Performance	44
4.6.4	Backtest and Optimization	47
4.6.5	Real-time	48
4.6.6	Calculation per Bar	49
4.6.7	Price Movement Emulation within the Bar at Backtest and Optimization	51
4.6.8	Commissions and Slippage	54
4.6.9	Advanced. How The Strategy Backtesting Engine works	54
4.6.10	Advanced. Portfolio	58
4.6.11	Advanced. AutoTrading	63
4.7	Advanced. Special Features	68
4.7.1	Chart Custom Draw	68
4.7.2	Chart ToolBar	70
4.7.3	Access to the data outside the chart window. DataLoader & SymbolStorage	71
4.7.4	Receiving the data for any symbol and any resolution. DataLoader.	72
4.7.5	Extended Trading Possibilities. TradeManager.	77
4.7.5.1	<i>Basic statements.</i>	77
4.7.6	Volume Profile	84
5	Compilation. Modules and assembly handling.	87
6	Integration with Microsoft Visual Studio 2010/2012/Express	88
6.1	Debugging with Microsoft Visual Studio 2010/2012	89
6.2	Debugging with Microsoft Visual Studio Express	90

1 Basic Skills

Indicators and signals for MultiCharts .NET can be written using the C# and VB.NET programming languages. The following information requires that the reader has basic knowledge of any of the C# or VB.NET programming languages. If not, then we recommend reviewing the following [reference](#) before continuing.

This Programming Guide describes the process of creating and applying studies in MultiCharts .NET. For general information on how to use MultiCharts .NET (i.e. how to create charts, scanner and DOM windows etc.) please refer to [MultiCharts Wiki](#).

Description of all PowerLanguage .NET methods and classes can be found in [Power Language .NET Help](#) (in PoweLanguage .NET main menu select **Help** and click **PowerLanguage .NET Help**).

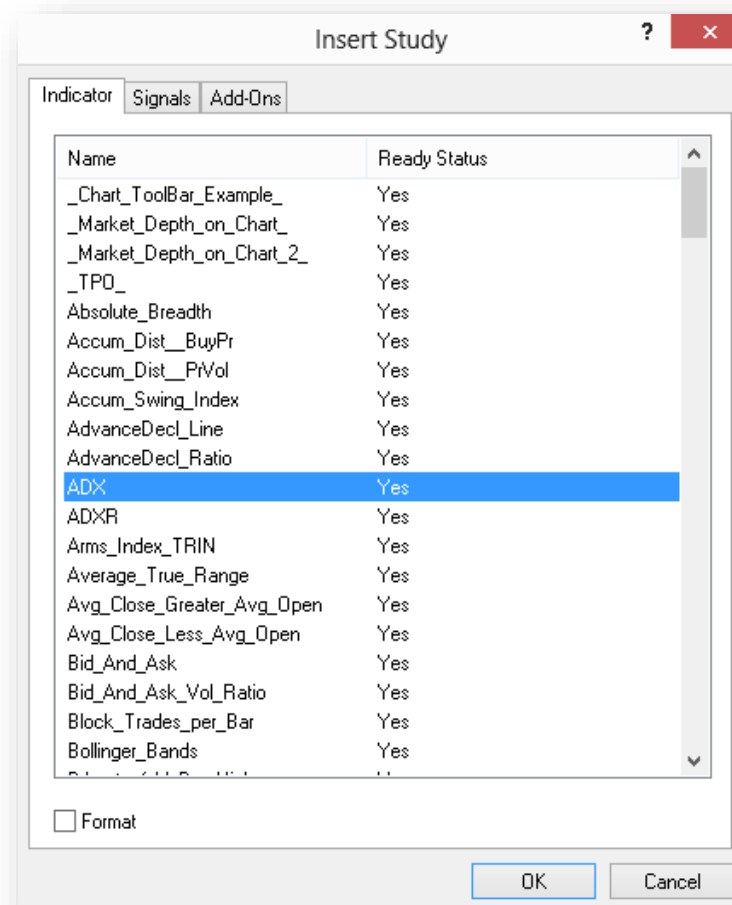
2 Basic Definitions

NOTE: Both indicators and signals can be referred to as studies or techniques.

2.1 Indicator

Indicators in MultiCharts .NET allow the user to analyze price and trade information and to graphically display it on the chart. To add an Indicator on a chart, right-click on the chart, click Insert Study, select the Indicator tab, then select the required Indicator and click OK.

For example, the ADX indicator calculation results will appear in a graph displayed at the bottom of the chart.



In MultiCharts .NET indicators are objects of the class, inherited from `IndicatorObject` class. Indicators, unlike other types of studies (functions, signals), can create and use plot objects (`IPlotObject`). These objects allow indicators to draw graphs, histograms, bars, etc. Before using plots, they should be created in the **Create ()** method of indicator class using the **AddPlot()** method.

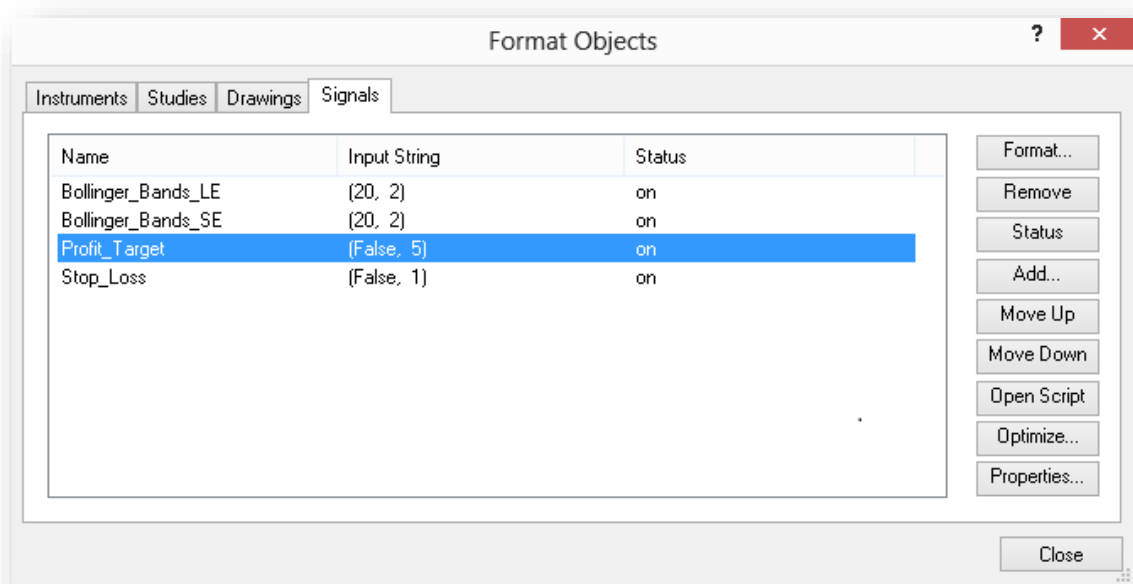
Here is an example of a simple indicator, which draws a gaps graph of day bars (the difference between the current bar open and the previous bar close in percent) in the form of histogram at the bottom of the chart:

```
public class DAY_GAP : IndicatorObject {
    public DAY_GAP(object _ctx):base(_ctx){}
    private IPlotObject plot_gap;//Plot object for graph drawing
    protected override void Create() {
        //Adds a graph to the chart
        plot_gap = AddPlot(new PlotAttributes("GAP",
EPlotShapes.Histogram, Color.Red));
    }
    protected override void CalcBar(){
        // Determines next bar plot value
        plot_gap.Set(100*((Bars.Open[0] -
Bars.Close[1])/Bars.Open[0]));
    }
}
```

2.2 Signal/Strategy

Trade strategies in MultiCharts .NET consist of one or several signals which are algorithms for order generation based on the analysis of price information on the chart.

For example, a strategy consists of one or more signals (algorithms) as shown in the chart below.



Bollinger_Bands_LE signal is an algorithm of order generation for long position entry.

Bollinger_Bands_SE signal is an algorithm of order generation for short position entry.

Profit_Target signal is an algorithm of profit fixation.

Stop_Loss is an algorithm protecting the position from a loss.

A signal is able to generate orders in accordance with a built-in algorithm, which will be executed by the strategy.

Signals have access to the information about the current condition and performance of a strategy.

To add a strategy with one or several signals to a chart, right click on the chart, select Insert Study, then select the Signals tab.

After the strategy calculation you can see markers of orders executed by the strategy. A report upon the results of strategy calculation can be found in the View menu in the Strategy Performance Report.

In MultiCharts .NET signals are objects of the class, inherited from [SignalObject](#).

In the process of the signal calculation orders will be generated according to the algorithm as described in the **CalcBar ()** method. These orders will subsequently be executed by the strategy.

Here is an example of a simple signal:

```
public class TestSignal : SignalObject
{
    public TestSignal(object _ctx):base(_ctx){}

    // Declaring orders-objects
    private IOrderMarket buy_order;

    protected override void Create() {
        buy_order = OrderCreator.MarketNextBar(new
        SOrderParameters(Contracts.Default, EOrderAction.Buy));
    }

    protected override void CalcBar(){
        // Signal's algorithm
        //.....
        //Order Generation
        buy_order.Send();
    }
}
```

2.3 Inputs

Inputs in MultiCharts .NET are user-defined options of the studies. Almost no indicators or signals exist without them. Inputs in PowerLanguage .Net are a public property of a study class, marked with an attribute [Input]. There are different types of inputs: [int](#), [double](#), [string](#), [DateTime](#), [Color](#), [Enum](#).

An example of integral type input in the study:

```
[Input]
public int length { get; set; }
```

Now this input will be available in Format Study dialog on the Inputs tab under the name “length”. Input values, chosen by the user in “Format Study” dialog, become available in the `protected override void StartCalc()` class study method.

Default values can be set in study constructor.

Example:

A study has one adjustable double type parameter:

```
public class Test : IndicatorObject {
    public Test(object _ctx):base(_ctx) {
        Percent = 10; // Default value
    }

    [Input]
    public double Percent {get; set;} //Input in percent.
}
```

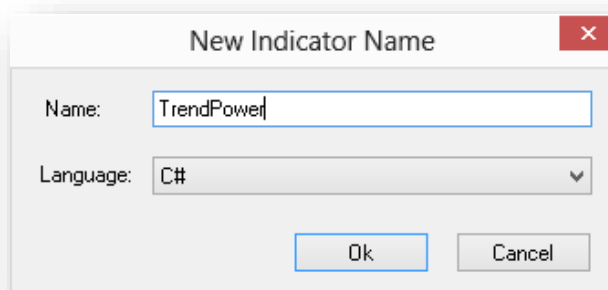
3 Tutorial

3.1 How to Create a C# Indicator

Let’s say that we want to write an indicator showing trend strength (please note that it is just an example and this indicator does not show true trend strength) in the form of a histogram. It will be calculated on the average price difference for a fast and a slow period and expressed in percent from the Close price of the current bar.

To create a new indicator, it is necessary to start PowerLanguage .NET Editor. In the editor main menu, select File, then select New, then New Indicator.

In the dialog window below:



Choose C# language and enter the name of the new indicator, i.e. TrendPower.

The PowerLanguage Editor will automatically create a template of an indicator class with one plot.

Let's rename it for our convenience:

```
private IPlotObject power_line;
```

In the **Create()** method choose the histogram style, color and add a text marker:

```
protected override void Create() {  
    power_line = AddPlot(new PlotAttributes("Power",  
    EPlotShapes.Histogram, Color.Cyan));  
}
```

Two inputs will also be necessary for setting the quantity of bars for the calculation of the average price for fast and slow periods:

```
[Input]  
public int fastLength { get; set; }  
  
[Input]  
public int slowLength { get; set; }
```

Assume that fastLength = 7, slowLength = 14 and are set as the default values. This is done in the indicator constructor:

```
public TrendPower(object _ctx):base(_ctx)  
{  
    fastLength = 7;  
    slowLength = 14;  
}
```

Write a private function that calculates average median price for the period:

```
private double AvgVal( int length )  
{  
    double aval = 0.0;  
    for (int i = 0; i < length; ++i)  
        aval += Bars.AvgPrice(i);  
    return aval / length;  
}
```

The logic of plot value setting on each bar should be described in the **CalcBar()** method:

```
protected override void CalcBar(){  
    double afast = AvgVal(fastLength); //Average price for a short period  
    double aslow = AvgVal(slowLength); //Average price for a long period  
    //Calculation of trend strength in percent from a close price of the  
current bar  
    double power = Math.Abs(100*(afast - aslow)/Bars.Close[0]);  
    power_line.Set(power); //Plot value setting for the current bar.  
}
```

As a result, we have an indicator, plotting a trend strength histogram upon two averages:

```
public class TrendPower : IndicatorObject {
```



```

public TrendPower(object _ctx):base(_ctx)
{
    fastLength = 7;
    slowLength = 14;
}

private IPlotObject power_line;

[Input]
public int fastLength { get; set; }

[Input]
public int slowLength { get; set; }

private double AvgVal( int length )
{
    double aval = 0.0;
    for (int i = 0; i < length; ++i)
        aval += Bars.AvgPrice(i);
    return aval / length;
}

protected override void Create() {
    power_line = AddPlot(new PlotAttributes("Power",
EPlotShapes.Histogram, Color.Cyan));
}

protected override void CalcBar(){
    double afast = AvgVal(fastLength);
    double aslow = AvgVal(slowLength);
    double power = Math.Abs(100*(afast - aslow)/Bars.Close[0]);
    power_line.Set(power);
}
}

```

Press F7 button for compilation of the indicator. It can now be plotted on the chart. Choose it in the **Insert Study** dialog window on the **Indicator** tab.



Next, we will highlight the areas of the histogram with a strong trend. Once the limits for what is to be considered as a strong trend and what is to be considered as a weak trend and their corresponding colors are determined inputs can be created as follows.

```
[Input]
public double strongLevel { get; set; }
[Input]
public Color strongColor { get; set; }
[Input]
public Color weakColor { get; set; }
```

In the indicator logic the plot color will be chosen depending on the trend strength:

```
protected override void CalcBar() {
    double afast = AvgVal(fastLength);
    double aslow = AvgVal(slowLength);
    double power = Math.Abs(100*(afast -
aslow)/Bars.Close[0]);
    Color pl_col = weakColor;
    if (power >= strongLevel ) pl_col = strongColor;

    power_line.Set(power, pl_col);
}
```

Now the indicator highlights red areas with a strong trend:



3.2 How to Create a C# Strategy

The indicator, which shows the strength trend has been created and defined. Now, on the basis of this indicator, we will build a strategy that will open a position at the beginning of a strong trend and close the position when the trend weakens.

In the PowerLanguage .NET Editor main menu, select File, then select New, then click New Signal and create a new signal with the name TrendPower. Next, copy the trend strength calculation logic with inputs from the previous indicator. As a result, the following signal will be created:

```
public class TrendPower : SignalObject {
    public TrendPower(object _ctx):base(_ctx)
    {
        fastLength = 7;
        slowLength = 14;
        strongLevel = 0.9;
    }

    [Input]
    public int fastLength { get; set; }

    [Input]
    public int slowLength { get; set; }

    [Input]
    public double strongLevel { get; set; }

    protected override void Create() {
    }

    private double AvgVal( int length )
    {
        double aval = 0.0;
        for (int i = 0; i < length; ++i)
            aval += Bars.AvgPrice(i);
        return aval / length;
    }

    protected override void StartCalc() {
    }

    protected override void CalcBar(){
        double afast = AvgVal(fastLength);
        double aslow = AvgVal(slowLength);
        double power = Math.Abs(100*(afast - aslow
/Bars.Close[0]));
    }
}
```

Add strong trend direction definition method. The method returns:

1 if High and Low of the current bar are higher. In this case, the trend is considered to be an uptrend,

-1 if High and Low of the current bar are lower. In this case the trend is considered to be a downtrend.

0 if the trend is not specified.

```
private int TrendDir() {
    if ((Bars.Low[0] < Bars.Low[1]) && (Bars.High[0] <
Bars.High[1])) return -1;
    if ((Bars.Low[0] > Bars.Low[1]) && (Bars.High[0] >
Bars.High[1])) return 1;
    return 0;
}
```

A long position will be opened by trend_LE market order, a short position by trend_SE market order. The position will be closed by one of the following orders: long position by trend_LX, short position by trend_SX.

Let's create these objects in our signal:

```
private IOrderMarket trend_LE;
private IOrderMarket trend_SE;
private IOrderMarket trend_LX;
private IOrderMarket trend_SX;

protected override void Create() {
    trend_LE = OrderCreator.MarketNextBar(new
SOrderParameters(Contracts.Default, "Tend LE", EOrderAction.Buy));
    trend_SE = OrderCreator.MarketNextBar(new
SOrderParameters(Contracts.Default, "Tend SE", EOrderAction.SellShort));
    trend_LX = OrderCreator.MarketNextBar(new
SOrderParameters(Contracts.Default, "Tend LX", EOrderAction.Sell));
    trend_SX = OrderCreator.MarketNextBar(new
SOrderParameters(Contracts.Default, "Tend SX", EOrderAction.BuyToCover));
}
```

The logic of this signal should be written in the **CalcBar()** method. If the trend is weak and becomes strong or if the trend is strong and becomes weak a long or short position will be opened depending upon the direction of the trend with the option to close any position. A Local variable should be created to save the previous strength trend value and its value will be reset before the beginning of every calculation of the strategy:

```
double old_power;
protected override void StartCalc() {
    old_power = 0;
}
protected override void CalcBar() {
    double afast = AvgVal(fastLength);
    double aslow = AvgVal(slowLength);
    double power = Math.Abs(100*(afast - aslow)/Bars.Close[0]);
    if ( (power >= strongLevel) && (old_power < strongLevel) ) {
        switch(TrendDir()) {
            case -1:
                trend_SE.Send();
                break;
            case 1:
                trend_LE.Send();
                break;
        }
    }
    if ((CurrentPosition.Side != EMarketPositionSide.Flat)
```

```

        &&(old_power >= strongLevel)
        &&(power < strongLevel))
    {
        trend_LX.Send();
        trend_SX.Send();
    }
    old_power = power;
}

```

To compile the signal, press F7. Once this is complete you can apply the signal to the chart. After the calculation you will see the order markers on the chart.



The results of the strategy can be seen in Strategy Performance Report by selecting View in the main menu and clicking on the Strategy Performance Report.

3.3 Debugging

Sometimes an indicator or signal may not work properly. So it will require some additional debugging. This can be done in two ways:

- 1) **Using Microsoft Visual Studio.** It is a good professional tool and it is much easier to debug the code there. The procedure is described in the following chapter: [Integration with Microsoft Visual Studio 2010/2012/Express](#).
- 2) **Without using a special debugger.** In this case the default PLEditor .NET Editor can be used. It is possible to trace necessary values of variables to either a previously created file or to the Output of the editor by means of scripting language.

```

#if DEBUG
    Output.WriteLine("CurrentBar = {0}; Close = {1}",
        Bars.CurrentBar, Bars.CloseValue );
#endif

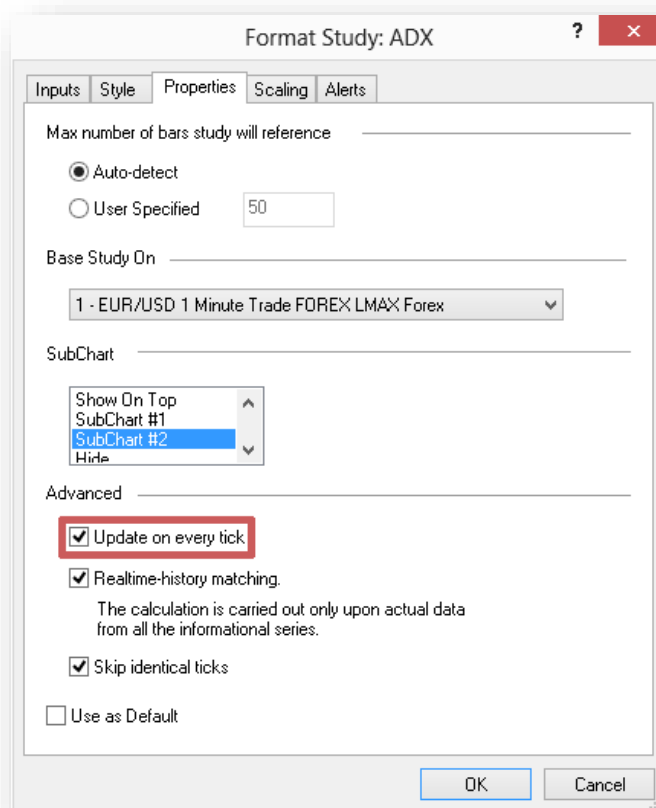
```

4 Understanding PowerLanguage .NET

4.1 How Indicators and Signals are Calculated

4.1.1 Basics

In MultiCharts .NET all studies (indicators and signals) are applied to the chart via the Insert Study window. Indicators can be applied to any chart symbol by clicking the right mouse button on the subchart to see the shortcut menu and selecting Insert Study. The constructor and the **Create()** method for study classes will be executed once when a study is selected, now indicators will create plots and signals will create orders. The **StarCalc()** method of study class is called every time before the calculation starts. The script begins to work through all of the historic bars of the symbol and on each bar the **CalcBar()** method is called. After the calculation of all historic bars, studies switch to the Real-time calculation on Close of every bar mode. For indicators, calculation on every tick received in real time can be selected. To do so, right-click on the chart to see the shortcut menu, select Format Indicators, select the indicator, then select Format, select the Properties tab and check the Update on Every Tick check box.



4.1.2 For Advanced Users

MultiCharts. NET is a multithread application and it is important to know in which threads studies are created, executed and deleted. There is a main thread (UI) and working threads, where the calculation of indicators and signals is executed.

Important! Please keep in mind that most methods and basic class properties of indicators, signals and functions are not thread-safe. It means that simultaneous work from several threads (which are created explicitly or implicitly) with methods and properties may lead to incorrect inner data and as the result to crashes in program. It is done in order to speed up calculation.

Starting from the moment of applying a study to a chart, scanner or portfolio, several stages occur before receiving the calculation results.

4.1.2.1 Construction. The **Create()** method

During this stage a new study object is created. Study object is created in the main thread and includes constructor and the **Create()** method calling. This process can be managed by the user through declaring and implementation of the method:

```
protected override void Create();
```

The **Create()** method is called first and only one time, immediately after the creation of a new study object. Here, the following is available: variables creation methods, adding plots function for indicators and strategy order creation factory:

- a) PowerLanguage.CStudyControl.CreateSimpleVar and PowerLanguage.CStudyControl.CreateSeriesVar for variable creation (please see description of these variables below).

```
private VariableObject<string> m_testVar;  
private VariableSeries<Double> m_seriesVar;  
private VariableSeries<Double> m_seriesVar2;  
  
protected override void Create()  
{  
    m_testVar = (VariableObject<string>)CreateSimpleVar<string>();  
    m_seriesVar = (VariableSeries<double>)CreateSeriesVar<double>();  
    m_seriesVar2 = new VariableSeries<Double>(this);  
}
```

- b) Adding plots to an indicator, for example, for MACD indicator:

```
private IPlotObject Plot1;  
  
protected override void Create() {  
    Plot1 =  
        AddPlot(new PlotAttributes("MACD", 0, Color.Cyan,  
                                   Color.Empty, 0, 0, true));  
}
```

- c) Strategy and signal order creation, using the **OrderCreator** property, for example, for RSI signals:

```
private RSI m_RSI;  
private VariableSeries<Double> m_myrsi;
```

```

private IOrderMarket m_RsiLE;

protected override void Create() {
    m_RSI = new RSI(this);
    m_myrsi = new VariableSeries<Double>(this);
    m_RsiLE = OrderCreator.MarketNextBar(new
SOrderParameters(Contracts.Default, "RsiLE", EOrderAction.Buy));
}

```

Please note, that all the listed methods and properties will be available ONLY at the stage of study construction (the **Create()** method). If they are used in the initialization or calculation stages, the study will compile and will be ready for application, but if applied to a chart or scanner window, they will generate an error message.

For example, plot addition is taken out of the **Create()** method in **StarCalc()**:

```

private IPlotObject Plot1;

protected override void StartCalc() {
    Plot1 =
        AddPlot(new PlotAttributes("MACD", 0, Color.Cyan,
                                Color.Empty, 0, 0, true));
}

```

If this study is applied to a chart or a scanner, a new study will be created, but when it comes to the initialization stage (the **StarCalc()** method), the AddPlot instruction, which can be applied only in the **Create()** method, will be detected and the study will switch to offline mode. The following error message will appear:

```

Error message.
Caption: "Error in study "MACD" :";
Message: "Unaccessible property(method): AddPlot. Construct only."

```

There are properties, available from the moment of object construction through its full life cycle:

- Graphic renderers of the following objects: texts, trend lines, arrows – these are the properties of DrwText, Drw, TrendLine, DrwArrow.
- The read-only **Output** property gives access to the Output window that displays study logs.
- The read-only **Alerts** property gives access to alerts (pop-up messages, sound signals and e-mails).
- The read-only **Name** property is the study name.

4.1.2.2 Initialization. The **StartCalc()** method

The **StartCalc()** method is called only once, before the every calculation.

The **StartCalc()** method is executed in the working thread and can be called repeatedly during the object life cycle (unlike the **Create()** method). To do this you must initialize the start values for study variables, the inputs for functions used in it and other custom objects in the **StartCalc()** method.

We can take the ADX indicator as the simplest example. One function, one serial variable and one plot are used here:

```
private Function.ADX m_adx1;
private VariableSeries<Double> m_adxvalue;
private IPlotObject Plot1;
```

In object creation stage (Create), we create the [the](#) objects of the function and variable and also add a new plot object. In initialization stage (StartCalc) we create the initial value of this variable and initialize the function input with a user input of the indicator itself:

```
[Input]
public int length { get; set; }

protected override void Create() {
    m_adx1 = new Function.ADX(this);
    m_adxvalue = new VariableSeries<Double>(this);
    Plot1 =
        AddPlot(new PlotAttributes("ADX", 0, Color.Cyan,
                                   Color.Empty, 0, 0, true));
}

protected override void StartCalc() {
    m_adx1.Length = length;
}
```

In initialization stage there is access to the price data through the **Bars** property and the **BarsOfDate ()** method.

Also, in this stage other indicator and strategy properties become available:

- a) The read-only **Environment** property – environment information
- b) The read-only **MaxDataStream** property – maximum number of data series, used by the study
- c) The bool **IsExist** method defines whether data series with a specified number exists on the chart (or in portfolio)
- d) The read-only **ExecInfo** property – current calculation context information (calculation data series number, execution offset, MaxBarsBack and MaxBarsForward properties)
- e) The read-only **ExecControl** property – allows the user to interrupt the calculation, recalculate the study, or force the study to calculate over a given period of time
- f) The read-only **VolumeProfile** and **VolumeProfileOfDataStream** properties – access to volume profile
- g) The read-only **TradeManager** property – access to trade information

- h) The read-only **StrategyInfo** property – access to strategy information
- i) The read-only **Portfolio** property – access to portfolio performance
- j) The read-only **Commission, InitialCapital, InterestRate, Account, Profile, Slippage** properties – current strategy properties
- k) The write-only **CustomFitnessValue** property sets custom value for optimization
- l) The **ChangeMarketPosition** method – market position change without sending an order.

All of the above listed properties and methods are available from the moment of initialization and during the calculation. Calling on them in the construction stage (the **Create ()** method) will result in an error message being generated. For example:

```
Message: "Unaccessible property (method): ExecControl. Initialize or
Execute."
```

4.1.2.3 Study Calculation on Data Series. The **CalcBar()** method

After the previous two stages (**Create, StartCals**) have finished their operations, let's move to bar-by-bar calculation of the study. The **CalcBar()** method will be called on for each bar of the data series, on which the study is applied, from left to right, in its turn. On each bar we have access to the data for that bar and for the previous bars on which the calculation was made (the **CalcBar()** method was called).

NOTE: All price data series where FullSymbolData of the **IInstrument** property was used will be available; this will be discussed separately.

Here is a simple example:

```
namespace PowerLanguage.Indicator {
    public class _plotAvgPrice : IndicatorObject {
        public _plotAvgPrice(object ctx) : base(ctx) {}
        private IPlotObject Plot1;

        protected override void Create() {
            Plot1 = AddPlot(new PlotAttributes("Close"));
        }

        protected override void CalcBar() {
            Plot1.Set(0, (Bars.HighValue + Bars.LowValue) / 2 );
        }
    }
}
```

The **CalcBar()** method is called starting from the first bar of the calculation and all following bars. This means the study calculation is in the context of the current bar, that is why the expression $(\text{Bars.HighValue} + \text{Bars.LowValue}) / 2$ will be calculated for the current bar, and `Plot1.Set` instruction will draw the calculated value also on the current bar. After the calculation on the last bar the final graph of average prices for the whole chart will be generated (scanner will show the last calculated value).

4.1.2.4 Calculation Stop. The **StopCalc()** method.

This method is called when the study calculation has finished, the study is in off mode or deleted from the price data series. The **StopCalc()** method can be called in the main or working thread.

4.1.2.5 Object Removal. The **Destroy()** method.

Study removal is executed in the main thread. The **Destroy()** method will be called before the removal. This method is called only once during the life circle of an indicator object (signal).

In the calculation process various .NET exceptions can be generated. It can be easily seen during debugging. You should not intercept any exceptions other than yours, as many mechanisms, for example, `BarsBack` auto detect, are realized through the generation and further processing of a specific exception. Please, remember, that an exception is NOT always an ERROR FLAG!

4.2 Data Access

Any study, whether it is an indicator or a signal, is an object that is applied to a price data series. Consequently, data and attributes of this data series are available from any study. Therefore, it is not important, which type of application we are working with in MultiCharts .NET: a chart, a scanner or a portfolio.

Data. A basis which all traders and market analysts are working with is price and volume data of various instruments. Visually, this data can be introduced as bars, which displays price movement for a particular period of time.

Access to the data from the study is executed through:

```
IInstrument Bars { get; }  
IInstrument BarsOfData(int dataStream);
```

All the information about price data series bars from the current calculation point back to history can be extracted by means of this property and method. The **BarsOfData()** method allows to get data upon the secondary data series, applied to the chart or portfolio.

At creation any study is connected to a particular data stream, the one which it is applied to. This data stream is main source for this data series. The **Bars** property returns a reference to the object implemented via the **Instrument** interface and is related to the basic data series of the study.

There can be only one data stream in the scanner; it will be the main source for the study. Only indicators can be applied to scanner instruments.

In portfolio there can be several data streams, but only the first one can be basic for the studies, only strategies can be applied to portfolio.

There can be several data streams on the chart but for signals, like for the portfolio, only the first one will be the main source. For indicators, it can be any of applied to the chart.

The main data stream can be chosen in the indicator settings (click Format Study ..., select Properties tab).

The **BarsOfData(int dataNum)** method should be used in order to access other data streams data, to be applied to the chart or (portfolio). The reference to the object, providing data of the specified instrument, will be returned according to the number of the data stream.

Data available through the **IInstrument** interface is described below.

4.2.1 Instrument Data.

- a) Prices: open, high, low, close of the current and previous count bars, for example:

```
double curClose = Bars.CloseValue; - get current close price
```

```
double prevBarOpen = BarsOfData(2).Open[1]; - get open price of previous bar from  
second data stream
```

- b) Volumes, ticks through the properties: Volume, Ticks, UpTicks, DownTicks, VolumeValue, TicksValue, UpTicksValue, DownTicksValue, for example:

```
double curVolume = Bars.VolumeValue; - get current bar volume
```

- c) Time: Bars.Time, Bars.TimeValue, for example:

```
DateTime prevBarTime = Bars.Time[1]; - get time of previous bar
```

- d) Depth of Market data: the **DOM** property, for example:

```
if ( Bars.DOM.Ask[0].Size > 100 ) buy_order.Send(); - send buy order if first DOM  
level ask size more than 100
```

- e) Status Line data: status line data when working either on the chart or scanner, for example:

```
double dailyLow = StatusLine.Low; - get daily low from status line.
```

4.2.2 Data for Calculation

- a) Current calculation bar: the **CurrentBar** property, for example:

```
double medPrice = 0;
```

```
if ( Bars.CurrentBar > 5 ) medPrice = ( Bars.Close[5] + Bars.CloseValue  
) / 2;
```

If current bar number more than 5 calculate median price between close 5 bars ago and current bar close.

b) Current calculation bar status: the **Status** property, for example:

```
if ( Bars.Status == EBarState.Close ) { /*check if we calculating on  
bar close */ }
```

c) Indicators of the last calculation bar and the last session bar, properties LastBarOnCharts, LastBarInSession, for example:

```
if (Bars.LastBarOnChart) plot1.Set(Bars.CloseValue); /*start plotting  
close value of bars since last bar */}
```

4.2.3 Properties of the Data Series, which a study is applied to

Attributes of Symbol, which the data series was built on, are available through the following property:

```
IInstrumentSettings Info { get; }
```

Instrument properties:

```
MTPA_MCSymbolInfo2 ASymbolInfo2 { get; }
```

```
string Name { get; } - Instrument name, for example, ESU3, EUR/USD;
```

```
string DataFeed { get; } - Data source, for example, CQG, eSignal;
```

```
string Description { get; } - Instrument description in words, for example, for ESU3 "S&P  
500 E-mini Futures Sep13";
```

```
ESymbolCategory Category { get; } - Value from the list of categories, for example, futures,  
Forex or stock;
```

```
string Exchange { get; } - Exchange, where the instrument is trading;
```

```
double PriceScale { get; } - Instrument price scale, for example, for ES=1/100 ;
```

```
double PointValue { get; } - Nominal value for one point for the instrument, for example,  
for ES=0.01;
```

```
double BigPointValue { get; } - Profit/loss money equivalent in the instrument currency,  
when the price moves by one minimum increment, for example, for ES = 50 ;
```

```
double MinMove { get; } - Minimum quantity of points, when the price moves by one  
minimum increment, for example, for ES = 25;
```

```
Resolution Resolution { get; } - For example, for created ES 5 of minute chart,  
Resolution.ChartType = ChartType.Regular; Resolution.Type = EResolution.Minute; Resolution.Size =  
5;
```

`double DailyLimit { get; } - Maximum price movement allowed during a session (currently is not used in MultiCharts .NET);`

`DateTime Expiration { get; } - Expiration date of the instrument (for futures), for example, ESU3 09/19/2013;`

`RequestTimeZone TimeZone { get; } - Local, Exchange or GMT;`

`RequestQuoteField QuoteField { get; } - Ask, Bid, or Trade ;`

`string SessionName { get; } - Session name (Check or add, or change session settings in QuoteManager) ;`

`double StrikePrice { get; } - For options;`

`OptionType OptionType { get; } - For options, Put or Call;`

`DataRequest Request { get; } - Request information, by which the symbol was built;`

4.2.4 StatusLine

When a study is applied to the chart or scanner, if the data source supports subscription to the status line data its data will be available from the study through the **StatusLine** property of the **IInstrument** interface. StatusLine itself provides the following information:

- a) the **Ask** and **Bid** read-only properties– return current Ask, Bid
- b) the **Time** read-only property – returns snapshot time of the status line
- c) **Open, High, Low, Close** read-only properties – return open, high, low, close of the current day (current trade session, depending on the data source)
- d) the **Last** read-only property – returns the last price for the instrument;
- e) the **PrevClose** read-only property – returns the close price of the previous day (session);
- f) the **TotalVolume** and **DailyVolume** read-only properties – return corresponding volumes of the status line;
- g) the **OpenInt** read-only property – returns open interest.

4.2.5 Random Data Access.

Starting from the initialization stage, the user gets access to all the data series that are used for calculation. It can be done by executing the **FullSymbolData** property of the **IInstrument** interface.

Example. Let's make the average price calculation of the whole data series on the initialization stage of the indicator and display the result during the calculation:

```
using System;
```

```

namespace PowerLanguage.Indicator
{
    public class ___test : IndicatorObject
    {
        private IPlotObject Plot1;

        public ___test(object ctx)
            : base(ctx)
        {
            m_chartAvgPrice = 0;
        }

        private double m_chartAvgPrice;

        protected override void Create()
        {
            Plot1 = AddPlot(new PlotAttributes("Test"));
        }

        protected override void StartCalc()
        {
            int totalBars = Bars.FullSymbolData.Count;

            double totalPrice = 0;
            for (int idx = 0; idx < totalBars; idx++)
                totalPrice += Bars.FullSymbolData.Close[-idx];
            m_chartAvgPrice = totalPrice / totalBars;
        }

        protected override void CalcBar()
        {
            Plot1.Set(0, m_chartAvgPrice);
        }
    }
}

```

Please note that bar indexation is executed, starting from the current bar, leftwards with positive indices, and rightwards with negative ones.

4.3 Plotting on the Chart

4.3.1 Plot and PlotPaintBar

There are two types of plots in MultiCharts .Net: graphic and text. And both are inherited from `IPlotObjectBase`:

```

public interface IPlotObjectBase<T>
{
    Color BGColor { get; set; } //Background color
    string Name { get; } //Plot name (set when the object is created)
}

```

```

void Reset();//Not to draw the plot on this bar
void Set(T val);//Set value on the current bar
void Set(T val, Color color); //Set value and color
void Set(T val, KnownColor color); //Set value and color
}

```

1. **Graphic Plots** are executed by the following interface:

```

public interface IPlotObject : IPlotObjectBase<double>
{
    IPlotAttributesArray<Color> Colors { get; } //Plot color
    IPlotAttributesArray<double> Values { get; } //Plot value
    IPlotAttributesArray<int> Widths { get; } //Thickness
    void Set(int barsAgo, double val); //Set value with displacement
    void Set(double val, Color color, int width); //Set value, color, thickness
    void Set(double val, KnownColor color, int width); //Set value, color, thickness
    void Set(int barsAgo, double val, Color color); //Set value, color with displacement
    void Set(int barsAgo, double val, KnownColor color); //Set value, color with displacement
    void Set(int barsAgo, double val, Color color, int width); //Set value, color, thickness with displacement
    void Set(int barsAgo, double val, KnownColor color, int width); //Set value, color, thickness with displacement
}

```

Displacement barsAgo = 0 for the current bar. barsAgo > 0 displacement to the history(barsAgo = 1 – the previous bar.) barsAgo < 0 – displacement to the future(barsAgo = -1 –the next bar).

Graphic indicators are created in the **Create()** study class method through the following methods:

AddPlot() or AddPlot(PlotAttributes info).

In PlotAttributes plot style can be set:

```

PlotAttributes(int plotNum); // Plot number (0-99)
PlotAttributes(string name); // Plot string name
PlotAttributes(string name, int plotNum); //Plot name and number
PlotAttributes(string name, EPlotShapes type, Color fgColor); //Name, type and color
PlotAttributes(string name, EPlotShapes type, Color fgColor, Color bgColor, int width, int style, bool showLastPriceMarker); //Name, type, color, background color, thickness, style, show/hide plot marker on the chart.

```

Plot types:

```

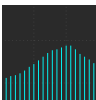
public enum EPlotShapes
{

```

Line = 0,



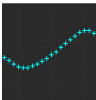
Histogram = 1,



Point = 2,

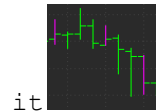


Cross = 3,



BarHigh = 4, //High of the bar (for PaintBar), BarLow is obligatory to use it

BarLow = 5, //Low of the bar (for PaintBar), BarHigh is obligatory to use it



LeftTick = 6, //tick to the left from the bar (for PaintBar)



RightTick = 7, //tick to the right from the bar (for PaintBar)



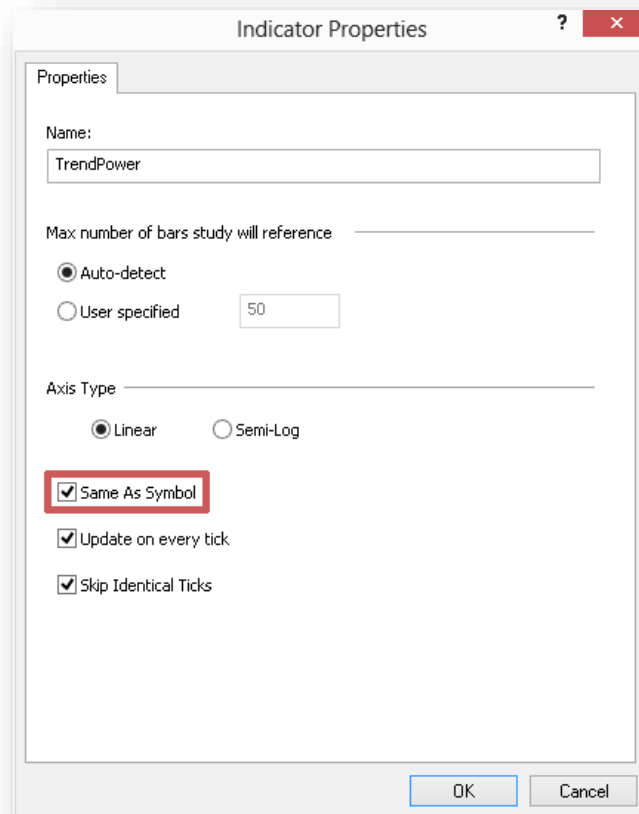
}

BarHigh, BarLow, LeftTick, RightTick allow to draw a complete bar. To build a bar OHLC type, four plots are necessary.

The **Style** property (line styles):

- 0 ————— (full line)
- 1 — — — — (dashes)
- 2 - - - - - (dots)
- 3 - . - . - (dash-dot)
- 4 - - - - - (dash-two dots)
- 5 (no line)

Tip: If it is necessary to paint bars in the indicator, then click the checkbox Same As Symbol in dialog window of indicator settings, or for the indicator class, set the `[SameAsSymbol(true)]` attribute.



This is an example of the indicator that paints uptrend bars in blue and downtrend bars in red:

```
[SameAsSymbol(true)]
public class SB_Up_Dn : IndicatorObject {
    public SB_Up_Dn(object _ctx):base(_ctx){}
    private IPlotObject plotH;
    private IPlotObject plotL;
    private IPlotObject plotO;
    private IPlotObject plotC;
    protected override void Create() {
        plotH = AddPlot(new PlotAttributes("", EPlotShapes.BarHigh,
Color.Red));
        plotL = AddPlot(new PlotAttributes("", EPlotShapes.BarLow,
Color.Red));
        plotO = AddPlot(new PlotAttributes("", EPlotShapes.LeftTick,
Color.Red));
        plotC = AddPlot(new PlotAttributes("", EPlotShapes.RightTick,
Color.Red));
    }
    protected override void CalcBar(){
        if (Bars.Close[0] > Bars.Open[0]){
            plotH.Set(Bars.High[0], Color.Blue);
            plotL.Set(Bars.Low[0], Color.Blue);
            plotO.Set(Bars.Open[0], Color.Blue);
            plotC.Set(Bars.Close[0], Color.Blue);
        }
    }
}
```

```

        if (Bars.Close[0] < Bars.Open[0]){
            plotH.Set(Bars.High[0], Color.Red);
            plotL.Set(Bars.Low[0], Color.Red);
            plotO.Set(Bars.Open[0], Color.Red);
            plotC.Set(Bars.Close[0], Color.Red);
        }
    }
}

```

Now the indicator will be drawn, not on a separate subchart, but right above the symbol that it is applied to.



2. **Text plots** are used to display text information in the indicator status line and are represented by the following interface:

```

public interface IPlotObjectStr : IPlotObjectBase<string>
{
}

```

Let's illustrate it with an example. The indicator will output values to the status line: "UP" when Close > Open and "DN" when < Open :

```

public class UpDn : IndicatorObject {
    public UpDn(object _ctx):base(_ctx){}
    private IPlotObjectStr plot_str;
    protected override void Create() {
        plot_str = AddPlot(new StringPlotAttributes(1));
    }
    protected override void CalcBar() {

```

```

        plot_str.Set("FLAT", Color.Cyan);
        if (Bars.OpenValue < Bars.CloseValue)
            plot_str.Set("UP", Color.Yellow);
        if (Bars.OpenValue > Bars.CloseValue)
            plot_str.Set("DN", Color.Red);
    }
}

```

The result:



4.3.2 Drawings

There are three types of drawings in MultiCharts .NET, which can be created from a study: TrendLine, Text, and Arrow. Every drawing type is created by the corresponding factory, which can be accessed through the following properties:

```

ITextContainer DrwText { get; }
ITrendLineContainer DrwTrendLine { get; }
IArrowContainer DrwArrow { get; }

```

Each property allows accessing the collection of drawings of a particular type for the given script. All of the drawings are created by calling the **Drawing's Create ()** method (IArrowContainer.Create(), ITextContainer.Create(), ITrendLineContainer.Create()).

Example:

```

protected override void CalcBar()
{
    if (Bars.CurrentBar == 1)
    {
        ChartPoint top = new ChartPoint(Bars.TimeValue, Bars.HighValue);
        ChartPoint bottom = new ChartPoint(Bars.TimeValue,
Bars.LowValue);

        DrwTrendLine.Create(bottom, top);
        DrwArrow.Create(bottom, false);
        ITextObject textObj = DrwText.Create(top, "1");
        textObj.VStyle = ETextStyleV.Above;
    }
}

```

```
    }
}
```

This script will mark the first bar of the data series with one trendline from Low to High, then it will draw an arrow below the bar and a text with the bar number above it.

The **Create()** method will form a corresponding type of drawing and bring back the reference to the interface which can be used to manipulate the object: it can be moved and its properties and styles can be changed.

We specify a point on the chart to which the drawing is attached when creating a drawing (two points for the trendline: start and end points). When creating a drawing there is also an opportunity to specify the data stream number and the drawing will be created according to its scale. Finally, if the study has a separate subchart, it is possible to specify that the drawing should be plotted on the subchart, where the study is applied. When arrow objects are created, it is possible to specify the arrow direction (up/down), and when text objects are created it is possible to specify the displayed text.

In every drawing collection there is a possibility to get access to the active object on the chart. Active property returns the reference to the corresponding object type. For example:

```
IArrowObject activeObj = DrwArrow.Active;
if (activeObj != null) { /* do something */ }
```

Drawing is considered to be an active object on the chart, if it was created or moved last, or was selected by mouse click.

Object movement is done by changing its coordinates. For a trendline object, they are Begin and End properties, and for text objects and arrows, it is Location property.

All types of objects have the Color property. Script can manage the following individual object properties:

1. Trendline has ExtRight and ExtLeft properties. If both properties are set as false they will provide a segment, if both are set as true they will provide a straight line and when one property is set as false and the other is set as true a beam be provided. Property size is used to get/set the line thickness. Alert property is used to enable/disable the signals of the trendline cross by the price data series. Property Style represents line style (solid, dashed, dotted, dashed2, dashed3). There is also an arbitrary trendline point price receiving method:

```
double PriceValue(DateTime dateTime);
```

2. BSColor property of the text object allows you to get/set a background color. HStyle and VStyle properties are used for text position management, the horizontal and vertical text position, regarding the text drawing point (Location). Border property is used to enclose the text into a frame. FontName property sets the font. Font styles are applied through the **SetFont()** method. Style availability can be checked through the **HaveFont()** method. Text property returns/sets the text.

3. The direction of arrow objects can be received/set through Direction property. Size property determines the size of the arrow. Text property allows you to get/set the text to an arrow object. FontName, TextSize, TextColor, TextBGColor properties and **HaveFont()**, **SetFont()** methods are used to manage text attributes.

Drawing removal can be done through the **Delete()** method for the corresponding object.

Note. If a drawing was created as the result of the script calculation made not at the bar Close, it will be DELETED after the calculation. To change this behavior (which may be considered strange by many programmers) and to not delete such drawings the following study type attribute should be set:

```
[RecoverDrawings(false)].
```

4.4 Output, Alerts and Expert Commentary.

4.4.1 Output

To open Output Window in PLEditor .Net select View and click Output Window.

Output Window can be accessed from any study during any calculation stage. It can be used for initial script debugging through logging the actions and printing the logs to Output window.

Example:

```
using System;

namespace PowerLanguage.Indicator
{
    public class ___test : IndicatorObject
    {
        public ___test(object ctx) : base(ctx) { }
        private IPlotObject Plot1;

        IVar<int> m_myPlotVal;

        protected override void Create()
        {
            Output.Clear();
            Output.WriteLine("___test Construct.");

            Plot1 = AddPlot(new PlotAttributes("Close"));
            m_myPlotVal = CreateSimpleVar<int>();

            Output.WriteLine("___test Construct finished.");
        }

        protected override void StartCalc()
        {
            Output.WriteLine("___test Init.");

            m_myPlotVal.Value = 0;
        }
    }
}
```

```

        Output.WriteLine("___test Init finished.");
    }

    protected override void CalcBar()
    {
        try
        {
            int pointsHL = (int)((Bars.HighValue - Bars.LowValue) /
Bars.Point);
            int pointsCO = (int)((Bars.CloseValue - Bars.OpenValue) /
Bars.Point);
            m_myPlotVal.Value = pointsHL / pointsCO;
            Plot1.Set(0, m_myPlotVal.Value);
        }
        catch (Exception exc)
        {
            Output.WriteLine("___test !!! Attention. Exception on bar
{0}, message: {1}", Bars.CurrentBar, exc.Message);
            Plot1.Set(0, -1);
        }
    }
}

```

The following text can appear in the Output window as a result of such script calculation:

```

___test Construct.
___test Construct finished.
___test Init.
___test Init finished.
___test !!! Attention. Exception on bar 358, message: Attempted to divide by
zero.
___test !!! Attention. Exception on bar 484, message: Attempted to divide by
zero.
___test !!! Attention. Exception on bar 569, message: Attempted to divide by
zero.

```

Write() and **WriteLine()** methods accept the formatted line, the difference between them is that the **WriteLine()** method adds a line ending symbol.

4.4.2 Alerts

Alert is the means by which a user is notified about some important event occurring on a data series by a text message, sound and e-mail. Studies can access alerts through the **Alerts** property:

```

public interface IAlert
{
    bool CheckAlertLastBar { get; } // true - if it is the last bar now
    bool Enabled { get; } //true - if alerts are on
    void Alert(); //to generate Alert with an empty text
    void Alert(string format, params object[] _args); //generate Alert
with a message.

```

```

        void Cancel();//finish Alert generation on this bar.
    }

```

To enable/disable strategy alerts, right-click on the chart to see the shortcut menu, click Properties, select the Alerts tab and check/uncheck the Enable Alerts check box.

To enable/disable Indicator alerts, right-click on the chart to see the shortcut menu, click Format Indicators, select the Indicator and click Format., then select the Alerts tab and check/uncheck the Enable Alerts check box.

The list of all generated alerts can be seen on the Alerts tab of the Orders and Position Tracker:

Date/Time	Source	Instru...	Resolution	Price	
13.03.2013 18:26:20	ADX	EUR/CAD	10 Second	1,33	Indicator turning down
13.03.2013 18:24:30	ADX	EUR/CAD	10 Second	1,33	Indicator turning up

Example:

Let's configure the indicator to send an alert every time when the Open of the new bar is higher than the High or lower than the Low of the previous bar.

Here is the script:

```

public class AlertOHOL : IndicatorObject {
    public AlertOHOL(object _ctx):base(_ctx){}
    private IPlotObject plotUpH;
    private IPlotObject plotUpL;
    private IPlotObject plotDnH;
    private IPlotObject plotDnL;
    protected override void Create() {
        plotUpH = AddPlot(new PlotAttributes("", EPlotShapes.BarHigh,
        Color.Red));
        plotUpL = AddPlot(new PlotAttributes("", EPlotShapes.BarLow,
        Color.Red));
        plotDnH = AddPlot(new PlotAttributes("", EPlotShapes.BarHigh,
        Color.Blue));
        plotDnL = AddPlot(new PlotAttributes("", EPlotShapes.BarLow,
        Color.Blue));
    }

    protected override void CalcBar(){
        if (Bars.Open[0] > Bars.High[1])
        {

```



```

        plotUpH.Set(Bars.High[0]);
        plotUpL.Set(Bars.Low[0]);
        Alerts.Alert("Open highest High");
    }
    if (Bars.Open[0] < Bars.Low[1])
    {
        plotDnH.Set(Bars.High[0]);
        plotDnL.Set(Bars.Low[0]);
        Alerts.Alert("Open lowest Low");
    }
}
}

```

Then, compile the indicator and apply it to the chart to see the result:



4.4.3 Expert Commentary

Alerts work in real-time but, if events that happened on historical bars should be commented on, Expert Commentary should be used. To add a commentary to the bar, it is necessary to use one of ExpertCommentary functions in the script:

```

public interface IExpertCommentary
{

```

```

        bool AtCommentaryBar { get; } // returns true, if in Expert
        Comentary mode a user has chosen this bar.
        bool Enabled { get; } //Returns true, if Expert Comentary Mode is
        enabled
        void Write(string format, params object[] _args); //Adds text to the
        comment
        void WriteLine(string format, params object[] _args); //Adds a line
        in the comment
    }

```

Example

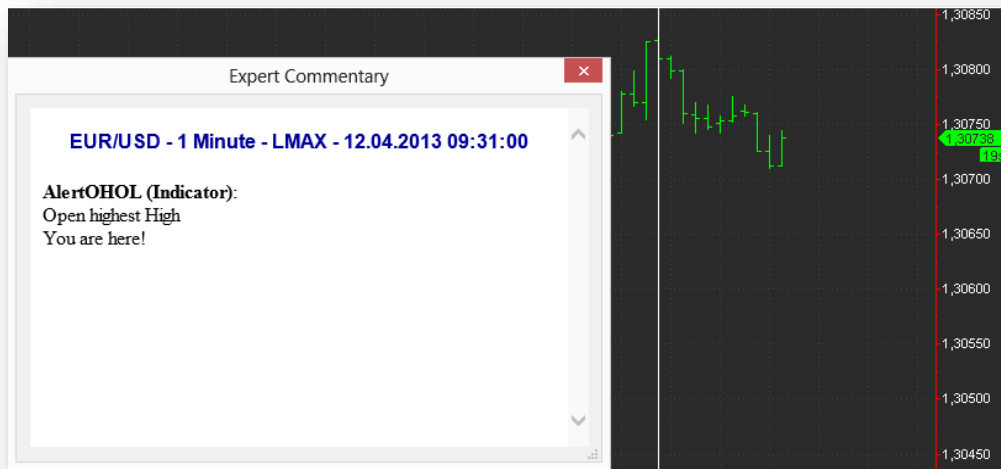
Let's add comments to the previous sample indicator:

```

protected override void CalcBar(){
    if (Bars.Open[0] > Bars.High[1])
    {
        plotUpH.Set(Bars.High[0]);
        plotUpL.Set(Bars.Low[0]);
        Alerts.Alert("Open highest High");
        ExpertCommentary.WriteLine("Open highest High");
    }
    if (Bars.Open[0] < Bars.Low[1])
    {
        plotDnH.Set(Bars.High[0]);
        plotDnL.Set(Bars.Low[0]);
        Alerts.Alert("Open lowest Low");
        ExpertCommentary.WriteLine("Open lowest Low");
    }
    if (ExpertCommentary.AtCommentaryBar)
    {
        ExpertCommentary.WriteLine("You are here!");
    }
}

```

Then, let's compile this indicator and apply it to the chart. Enable Expert Commentary (in the main menu of MultiCharts .NET select View and click Expert Commentary) and click on the necessary bar:



4.5 Functions and Special Variables

4.5.1 Functions

Simple functions execute some operation over the sent arguments and return the results. If the arguments are identical, the result will always be the same. These are so-called “pure” functions. Examples of these functions are: `Math.Max()`, `Highest()` etc. But there are functions, which store certain information, for example, results of previous calculations. We can take `XAverage` (Exponential Moving Average) as an example, where the current value is calculated using the previous value of the function and the arguments. These are so-called functions with state or functional objects. Class-functions, inherited from [FunctionSeries<T>](#) or [FunctionSimple<T>](#) are such functional objects.

In PL.NET there is a variety of such built-in functional objects. The standard model of their usage is the creation of an instance of the function object, initializing it with actual arguments and addressing the function value during the calculation process.

4.5.2 Variables

Those, who are familiar with C# or other OOP programming languages know such things as class fields. Together they form the state of the object of the given class. In PL.NET there are special classes of variables which make the writing logic of functions, indicators and signals easier regardless of whether the calculation is executed based on historical or real-time data. When calculating a study on historical data, `CalcBar()` method will be called once for each bar. This is an example of such a version of `CurrentBar`:

```
public class CurrentBar : FunctionSimple<int>
{
    public CurrentBar(CStudyControl master) : base(master) {}

    private int m_current_bar;
    protected override int CalcBar() {
        return ++m_current_bar;
    }
    protected override void StartCalc() {
```

```

        m_current_bar = 0;
    }
}

```

When calculating on historical data this function will return correct values only if it is called once per bar. In real-time calculation, CalcBar will be called on each tick of bar, the function will not work correctly. To make it work correctly, its logic should be changed. For example:

```

protected override int CalcBar() {
    var _result = m_current_bar + 1;
    if (Bars.Status==EBarState.Close)
        ++m_current_bar;
    return _result;
}

```

As we can see, the value of the variable has to be incremented only at the bar close. For these cases specific variables exist: `VariableSeries<T>` and `VariableObject<T>`. `VariableSeries<T>` should be used when it is necessary to call for the values of the previous bars variable. Considering this, the example would look like this:

```

public class CurrentBar : FunctionSimple<int>
{
    public CurrentBar(CStudyControl master) : base(master) {}

    private VariableObject<int> m_current_bar;
    protected override int CalcBar() {
        return ++m_current_bar.Value;
    }
    protected override void Create() {
        m_current_bar = new VariableObject<int>(this);
    }
}

```

And it will equally work either at historical calculation, or real-time calculation. In addition to this, it will also work correctly, even if we call this function several times during the bar.

In summarizing the information about functions and variables, our example with `CurrentBar` can be simplified:

```

public class CurrentBar : FunctionSeries<int>
{
    public CurrentBar(CStudyControl master) : base(master) {}

    protected override int CalcBar() {
        return this[1] + 1;
    }
}

```

4.5.3 Barsback

Barsback is the offset on the left of the price data series measured in bars and is necessary for the study calculation.

Let's consider the following example:

```
using System;

namespace PowerLanguage.Indicator
{
    public class ITest : IndicatorObject
    {
        private VariableSeries<Double> m_plotVal;

        private IPlotObject Plot1;

        public ITest(object ctx) : base(ctx) {}

        protected override void Create() {
            Plot1 = AddPlot(new PlotAttributes("Test"));
            m_plotVal = new VariableSeries<Double>(this);
        }

        protected override void CalcBar() {
            m_plotVal.Value = Bars.Close[10] - Bars.CloseValue;
            Plot1.Set(0, m_plotVal.Value);
        }
    }
}
```

This simple example shows, that to calculate this study, at least eleven bars of the price data series are necessary (Bars.Close[10] call considers Close price removal ten bars back from the current calculation bar).

Barsback can be set by the user manually (in indicator properties) or it can be calculated automatically. In this simple case, the maximum quantity of bars back =11, can be set manually. It is not always clear to the user how many bars of price data series it should be used, in this case it is best to have it automatically calculated. Let's change the initial example:

```
using System;

namespace PowerLanguage.Indicator
{
    public class ITest : IndicatorObject
    {
        private VariableSeries<Double> m_plotVal;
        private IPlotObject Plot1;

        public ITest(object ctx) : base(ctx)
        {
            length = 10;
        }

        [Input]
        public int length { get; set; }

        protected override void Create()
        {

```

```

        Plot1 = AddPlot(new PlotAttributes("Test"));
    }

    protected override void CalcBar() {
        m_plotVal.Value = Bars.Close[length] - Bars.CloseValue;
        Plot1.Set(0, m_plotVal.Value);
    }
}

```

It is not known in advance which input a user will set. But the number of data series bars, necessary for the indicator calculation, depends on it (determined by `Bars.Close[length]` expression). In this case, you can select the Auto-detect option for the “Max number of bars study will reference” setting. Then Barsback for the study will be selected automatically during the calculation. It works as follows:

1. During the first calculation by default it is considered that one bar is enough for the calculation.
2. When, during the calculation at the first bar `Bars.Close[length]` (`length > 0`) expression is executed, an exception will be generated inside the calculation module causing the study to stop the calculation (`StopCalc`).
3. The exception is intercepted and according to the specific algorithm, barsback value is incremented. Then, the study is initialized (`StartCalc`) and the new calculation starts (from the bar with the barsback number, which is the new value of the minimum number of bars, necessary for the calculation).
4. If at the execution of `Bars.Close[length]` expression the quantity of bars is not sufficient (`length > barsback`), an exception will be generated, the study stops the calculation (`StopCalc`). Then, it will go to the point 3. And so on, until the quantity of bars is enough.

Note. If the built data series does not have the necessary quantity of bars, the indicator does not switch off. It stays at Calculating mode... and waits, until the quantity of bars of the data series is sufficient.

4.6 Strategies

4.6.1 Orders

Order objects can be created and generated only by signals. All orders inherit from the basic interface.

```

public interface IOrderObject
{
    int ID { get; } //unique order number
    Order Info { get; } //Basic order information (direction, type,
category etc.)
}

```

There are three types of orders:

- 1) Market orders executed at current market price.

```
public interface IOrderMarket : IOrderObject
{
    void Send();
    void Send(int numLots);
}
```

- 2) Price orders executed when market price touches or crosses order price level.

```
public interface IOrderPriced : IOrderObject
{
    void Send(double price);
    void Send(double price, int numLots);
}
```

- 3) Algorithmic Stop-Limit orders executed as price Limit orders, but only after market price touches or crosses the stop price.

```
public interface IOrderStopLimit : IOrderObject
{
    void Send(double stopPrice, double limitPrice);
    void Send(double stopPrice, double limitPrice, int numLots);
}
```

Order objects are created only in **Create()** method using OrderCreator:

- 1) `IOrderMarket OrderCreator.MarketNextBar(SOrderParameters orderParams);` - creates a Market order that should be sent at the Open¹ of the next bar, the one after the bar where the order was generated.
- 2) `IOrderMarket OrderCreator.MarketThisBar(SOrderParameters orderParams);` - creates a Market order that should be sent on the Close of the bar where it was generated.
- 3) `IOrderPriced OrderCreator.Limit(SOrderParameters orderParams);` - creates a price Limit order that should be sent at the Open² of the next bar, the one after the bar where the order was generated. A Limit order can be filled at the price set, or better, in **Send()** method when generating the order where buy equals order price and lower and sell equals order price and higher.
- 4) `IOrderPriced OrderCreator.Stop(SOrderParameters orderParams);` - creates a price Stop order that should be sent at the Open³ of the next bar, the one after the bar where the order was generated. Stop order can be filled at the price set, or worse, in **Send ()** when generating the order buy equals order price and lower and sell equals order price and higher.
- 5) `IOrderStopLimit OrderCreator.StopLimit(SOrderParameters orderParams);` - creates algorithmic Limit order with a Stop condition that should be sent at the Open⁴ of the next bar, the one after the bar where the order was generated.

¹If the signal is calculated in Intra-Bar Order Generation mode, the order is sent on the tick that follows one where order has been generated.

² If the signal is calculated in Intra-Bar Order Generation mode, the order is sent on the tick that follows one where order has been generated.

³ If the signal is calculated in Intra-Bar Order Generation mode, the order is sent on the tick that follows one where order has been generated.

⁴ If the signal is calculated in Intra-Bar Order Generation mode, the order is sent on the tick that follows one where order has been generated.

The following structure is used to create an order object in all the **OrderCreator()** methods:

```
SOrderParameters:
SOrderParameters(EOrderAction action);
SOrderParameters(Contracts lots, EOrderAction action);
SOrderParameters(EOrderAction action, string name);
SOrderParameters(Contracts lots, EOrderAction action, OrderExit exitInfo);
SOrderParameters(Contracts lots, string name, EOrderAction action);
SOrderParameters(Contracts lots, string name, EOrderAction action, OrderExit
exitInfo);
```

Settings:

EOrderAction:

Buy – long position entry (buying). If the short position is currently opened, then it will be closed and the long one will be opened.

Sell – long position exit (selling). This order cannot be sent, if the long position is not opened. Also, this position cannot be reversed into short.

SellShort – short position entry (selling). If the long position is currently opened, then it will be closed, and the short one will be opened.

BuyToCover – short position exit (buying). This order cannot be sent, if the short position is not opened.

Contracts:

Default – the quantity of contracts is calculated, according to the strategy settings in Strategy Properties window.

CreateUserSpecified(int num) – the default quantity of contracts for the order is set in the method settings.

name – Custom name of the order.

OrderExit:

FromAll – closes all entries, which opened the position. If the quantity of contracts is specified, then close each entry partially on the quantity of contracts specified.

Total – closes the position on ONLY the specified quantity of contracts. The entries will be closed accordingly, starting from the first one, up to the last one, until the exit is executed upon the quantity of contracts specified.

FromEntry(IOrderObject entry) – closes the particular entry, in the entry setting, created earlier, in the same signal.

4.6.2 Special Orders

The signal can generate special exit order types, besides price and market orders such as:

`ProfitTarget`, `StopLoss`, `BreakEven`, `DollarTrailing`, `PercentTrailing`, `ExitOnClose`.

Special order types can work in one of two modes (if the mode is set in several signals, then the mode that was set last is used for all special orders):

CurSpecOrdersMode = PerContract – special exits are attached to each entry that opened the position. Average entry order fill price is used to calculate the special order. The Amount should be specified in currency per one contract in the special order settings.

CurSpecOrdersMode = PerPosition – one exit is generated per position which closes it completely. The average open position price is used for the calculation of special order prices. In the special order settings, the Amount should be specified in currency per entire position.

NOTE: Amount is one of the parameters used for special orders generation. It sets potential profit value after reaching which the special order becomes active.

Commands to generate special order types are:

GenerateExitOnClose() – to generate a market order which closes the current position upon the session close. Please note, that during automated trading, this order may not be executed, because at time the order is sent the trading session may already be closed and orders may not be accepted. To use this type of order in automated trading correctly, it is recommended to set the sessions so they end, for example, a minute before the exchange trading session closes.

GenerateProfitTarget(double amount) – to generate a Limit order with a price where the profit (excluding commission) from a position (`CurSpecOrdersMode = PerPosition`) or from a contract for each entry (`CurSpecOrdersMode = PerContract`), should not be lower than the amount set in the Amount parameter.

Example:

The long position was opened at price 12,25 for 100 contracts.

`BigPointValue=10`, `MinMove=25`, `PriceScale=0,01`.

Signal code:

```
CurSpecOrdersMode = ESpecOrdersMode.PerPosition;  
GenerateProfitTarget(5);
```

Sell Limit order will be generated at price = $\text{EntryPrice} + (\text{Ammount}/\text{Contracts}) / \text{BigPointValue} = 12,25 + 5/100/10 = 12,255$;

Because the price does not correspond to the minimum price increment, it will be rounded to BETTER price, according to the step $\text{MinMove} * \text{PriceScale} = 25 * 0,01 = 0,25$ to 12,50.

The final ProfitTarget order: Sell 100 contracts at 12,50 Limit;

GenerateStopLoss (double amount) – to generate a Stop order with the price to set the loss (excluding commission) from a position (CurSpecOrdersMode = PerPosition) or a contract for each entry (CurSpecOrdersMode = PerContract) specified in the Amount parameter.

Example:

The symbol has the following parameters: BigPointValue=100, MinMove=2, PriceScale=0,01.
Assume the short position is opened at the average price 0,982 for 10000 contracts by the two orders SellShort1 1000 at 1,00 and SellShort2 9000 at 0,98;

Signal code:

```
CurSpecOrdersMode = ESPECOrdersMode.PerContract;  
GenerateStopLoss (5);
```

GenerateStopLoss(5) in PerContract mode will generate 2 Stop orders (one exit per each entry).

StopLoss order price for the first entry = EntryPrice + Ammount / BigPointValue = 1,00 + 5/100 = 1,00 + 0,05 = 1,05; considering the price step (0,02) is rounded to the WORSE = 1,06.

StopLoss order price for the second entry = EntryPrice + Ammount / BigPointValue = 0,98 + 5/100 = 0,98 + 0,05 = 1,03; considering the price step (0,02) is rounded to the WORSE = 1,04.

Final sending orders:

BuyToCover 1000 contracts from SellShort1 at 1,06 Stop;

BuyToCover 9000 contracts from SellShort1 at 1,04 Stop;

GenerateBreakEven (double amount) – to generate a stop order at the break-even level (calculated excluding commission) after achieving profit per position (in PerPosition mode) or per contract (PerContract), including commission. Break-even level is calculated excluding the commission and is equal to the execution of the entry order in PerContract mode or the average open price of the position in PerPosition mode.

In PerPosition mode, the stop order sending condition is:

- Positions[0].MaxRunUp >= **amount**;

In PerContract mode, the stop order sending condition for the entry is:

- TradeRunUp >= **amount**;

GenerateDollarTrailing (double amount) – to generate a trailing stop order with the price to close the position (RepPosition/entry(PerContract)), when the potential profit decreases from the maximum value by a particular amount.

Stop order(s) price calculation is executed tick by tick.

Example:

The symbol has the following parameters: BigPointValue=1, MinMove=1, PriceScale=0,01.

The long position entry was at price 34,14 for 100 contracts.

The price after entry has reached maximum MaxProfitPrice = 35,01.

Signal code:

```
CurSpecOrdersMode = ESPECOrdersMode.PerPosition;  
GenerateDollarTrailing(5);
```

TrailingStop price = MaxPositionProfitPrice – amount/contracts/BigPointValue = 35,01 – 5/100/1 = 35,01 – 0,05 = 34,96. After rounding to the minimum price step to the **WORST**, we get the price 34,96.

The final order: Sell 100 contracts at 34,96 Stop.

GeneratePercentTrailing(double amount, double percentage) – to generate a trailing stop order that closes the position (trade) after the position(PerPosition)/trade(PerContract) reaches the profit that is greater or equal to the price set in the amount parameter. The price will close the position/trade when the profit goes down by a percentage% from the maximum value.

Stop order(s) price calculation is executed tick by tick.

Example:

The symbol has the following parameters: BigPointValue=1, MinMove=1, PriceScale=0,01.

The long position entries were at price 24,10 for 100 contracts and 24,56 for 50 contracts.

Maximum price after the first entry was 24,60, after the second one was 24,58.

Signal code:

```
CurSpecOrdersMode = ESPECOrdersMode.PerContract;  
GeneratePercentTrailing(0.30, 25);
```

Sending of trailing stop for the first entry possibility is verified:

MaxProfitPerContract = (MaxPrice – EntryPrice)*BigPointValue = (24,60 - 24,10)*1 = 0,50; Yes, the condition for stop sending is executed MaxProfitPerContract >= amount. Let's count the price:

StopPrice = MaxPrice – (MaxPrice- EntryPrice)*percentage/100 = 24,60 - (24,60 - 24,10)*25/100 = 24,60 – 0,50*0,25 = 24,475. Let's round the price to the minimum step to the **WORST**, we get the price = 24,47.

Sending of trailing stop for the first entry possibility is verified:

MaxProfitPerContract = (MaxPrice – EntryPrice)*BigPointValue = (24,58 - 24,56)*1 = 0,02;

MaxProfitPerContract = (MaxPrice – EntryPrice)*BigPointValue = (24,58 - 24,56)*1 = 0,02;

MaxProfitPerContract condition >= amount for this entry was not executed, so the stop order for it will not be sent at this calculation. It will be generated at the moment, when the price is reached = EntryPrice + amount/BigPointValue = 24,56 + 0,30/1 = 24,86.

The final order will be only one: Sell 100 contracts from Entry1 at 24,47 Stop;

CurSpecOrdersMode – affects all special orders in the strategy. The last set value of this marker is always used for price calculations.

4.6.3 Strategy Performance

Signals as well as functions and indicators can get key information upon the current strategy state during their calculation through `StrategyInfo`:

`StrategyInfo.AvgEntryPrice` is the average open price of the position on the chart. If the position is not opened, then 0 is returned;

`StrategyInfo.MarketPosition` is the current market position. For example, we are at the short position for 100 contracts, then `StrategyInfo.MarketPosition` will return "-100". For example, if we have a long position for 50 contracts, the `StrategyInfo.MarketPosition` will return the value "50".

`StrategyInfo.ClosedEquity` – is the current NetProfit of the strategy on this bar.

`StrategyInfo.OpenEquity` = current NetProfit + current OpenPL of the strategy on this bar.

Following keywords can be used only in signals.

More detailed information upon current strategy parameters can be obtained through `CurrentPosition(IMarketPosition)`:

`CurrentPosition.Value` – the same as `StrategyInfo.MarketPosition`.

`CurrentPosition.Side` – `EMarketPositionSide.Flat` – there is no open position, `EMarketPositionSide.Long` – long position is opened, `EMarketPositionSide.Short` – short position is opened.

`CurrentPosition.Profit` – always 0 (open position has no realized profit/loss).

`CurrentPosition.OpenProfit` – current OpenPL upon the open position or 0, if the position is closed.

`CurrentPosition.ProfitPerContract` – current "OpenPL" for the open position, returns 0 if position is closed.

`CurrentPosition.OpenLots` – the amount of not closed contracts in the position.

`CurrentPosition.MaxDrawDown` – minimum OpenPL for the current position, for the whole period of time, while the position was opened.

`CurrentPosition.MaxRunUp` – maximum OpenPL for the current position, for the whole period of time, while the position was opened.

`CurrentPosition.ClosedTrades[]` – the list of closed trades (`ITrade`) in the current position.

`CurrentPosition.OpenTrades[]` – the list of open trades (`ITrade`) in the current position.

`CurSpecOrdersMode` – current mode of special orders (`ESpecOrdersMode.PerContract`, `ESpecOrdersMode.PerPosition`)

`PositionSide` – current position: `EMarketPositionSide.Flat` – position is not opened, `EMarketPositionSide.Long` – long position, `EMarketPositionSide.Short` – short position.

In signals, the history of positions is available (`IMarketPosition`), either opened or closed by the strategy through `Positions[] (IMarketPosition)`.

`Position[0]` – current position (the same as `CurrentPosition`), `Position[1]` – the previous opened position, etc.

Closed positions information is available through the **IMarketPosition** interface:

`IMarketPosition.Value` – 0, as there are no open trades in the close position.
`IMarketPosition.Side` – `EMarketPositionSide.Long` – long position.
`EMarketPositionSide.Short` – short position.
`IMarketPosition.Profit` - total profit upon all the position trades.
`IMarketPosition.ProfitPerContract` – profits upon all the position trades per contract.
`IMarketPosition.OpenLots` = 0 for the closed positions, as by default, all the contracts were closed.
`IMarketPosition.MaxDrawDown` – maximum potential loss for the position, for the whole period, while the position was opened.
`IMarketPosition.MaxRunUp` – maximum OpenPL for the position, for the whole period, while the position was opened.
`IMarketPosition.ClosedTrades[]` – list of trades (`ITrade`) in the position.
`IMarketPosition.OpenTrades[]` – list of opened trades for the closed position, that is empty.

Example:

Initially, `CurrentPosition.Value` = 0

```
CurrentPosition.Side = EMarketPositionSide.Flat
CurrentPosition.OpenLots = 0
CurrentPosition.ClosedTrades[] – empty
CurrentPosition.OpenTrades[]
```

The strategy has generated and executed a Buy order for 100 contracts.
Now our position is the following:

```
CurrentPosition.Value = 100
CurrentPosition.Side = EMarketPositionSide.Long
CurrentPosition.OpenLots = 100
CurrentPosition.ClosedTrades[] – empty
CurrentPosition.OpenTrades[] – one element (EntryOrder = Buy 100, ExitOrder = null)
```

The strategy has generated and executed a Sell total order for 10 contracts.
Now our position is the following:

```
CurrentPosition.Value = 90
CurrentPosition.Side = EMarketPositionSide.Long
CurrentPosition.OpenLots = 90
CurrentPosition.ClosedTrades[] – one element (EntryOrder = Buy 10, ExitOrder = Sell 10)
CurrentPosition.OpenTrades[] – one element (EntryOrder = Buy 90, ExitOrder = null)
```

The strategy has generated and executed a SellShort order for 50 contracts.
Now our position is the following:

```
CurrentPosition.Value = -50
CurrentPosition.Side = EMarketPositionSide.Short
CurrentPosition.OpenLots = 50
CurrentPosition.ClosedTrades[] – empty
CurrentPosition.OpenTrades[] – one element (EntryOrder = SellShort 50, ExitOrder = null)
```

```

Position[1].Value = 0
Position[1].Side = EMarketPositionSide.Long
Position[1].OpenLots = 0
Position[1].ClosedTrades[] - two elements:
    1) Entry = Buy 10 Exit = Sell 10
    2) Entry = Buy 90 Exit = SellShort 90
Position[1].OpenTrades[] - empty

```

The following main settings of the strategy are available:

Commission – commission of the first level of the commission rule.
 InitialCapital – strategy setting «Init Capital (\$)». Does not change during the calculation process.
 InterestRate – «Interest Rate (%)» strategy setting.
 Slippage – «Slippage (\$)» strategy setting.

Detailed information about the strategy performance:

GrossLoss – Profit amount upon losing trades
 GrossProfit – Profit amount upon profitable trades
 NetProfit – net profit upon the strategy (GrossProfit+ GrossLoss)

NumEvenTrades – amount of zero trades with Profit = \$0.00.
 NumLosTrades – amount of losing trades with Profit < \$0.00.
 NumWinTrades – amount of profitable trades with Profit > 0.00\$.
 TotalBarsEvenTrades – amount of bars in zero trades.
 TotalBarsLosTrades – amount of bars in losing trades.
 TotalBarsWinTrades – amount of bars in profitable trades.

AvgBarsEvenTrade – average number of bars in zero trades.
 AvgBarsLosTrade – average number of bars in losing trades.
 AvgBarsWinTrade – average number of bars in profitable trades.
 LargestLosTrade – Profit of the most losing strategy trade.
 LargestWinTrade – Profit of the most profitable strategy trade.
 MaxConsecLosers – maximum number of losing trades in a row (Profit < 0).
 MaxConsecWinners – maximum number of profitable trades in a row (Profit > 0)
 MaxDrawDown – maximum draw down StrategyInfo.OpenEquity for all previous trade period.
 MaxLotsHeld – maximum number of possessing contracts for all previous trade period.
 TotalTrades – total number of trades for the whole trade period.
 PercentProfit – number of profitable trades with respect to the total number of trades in percent.

Trade information (ITrade) (entry + closing exit):

```

public interface ITrade
{
    double CommissionValue { get; } - commission of the trade
    ITradeOrder EntryOrder { get; } - the order, that opened the trade -
entry
    ITradeOrder ExitOrder { get; } - the order, that closed the trade -
exit
    bool IsLong { get; } - Means, that the entry was Buy (true) or
SellShort (false)
    bool IsOpen { get; } - Means, that the entry was not closed yet

```

```

        double Profit { get; } - OpenPL, if the trade is opened or
        TradeProfit - if closed.
    }

    Trade order information ITradeOrder:
    public interface ITradeOrder
    {
        EOrderAction Action { get; } - EOrderAction.Buy, EOrderAction.Sell,
        EOrderAction.SellShort, EOrderAction.BuyToCover.
        int BarNumber { get; } - number of the bar, on which the order
        occurred (Bars.CurrentBar, on which the order occurred)
        OrderCategory Category { get; } - OrderCategory.Market,
        OrderCategory.Limit, OrderCategory.Stop, OrderCategory.StopLimit
        int Contracts { get; } - number of contracts in the filled order
        string Name { get; } - order name
        double Price { get; } - execution price
        DateTime Time { get; } - time of the bar where the order was
        executed
    }

```

4.6.4 Backtest and Optimization

Backtest

By default strategies are calculated in Backtest mode and calculated on historic chart bars.

To see the result of strategy calculation on historic data, in the MultiCharts .NET main menu select View and click Strategy Performance Report. In this report you will find detailed information concerning the performance and results of the strategy calculation.

In Backtest mode the strategy handles all the chart bars one by one. And at each bar, the calculation of all the strategy signals that are specified in Format Object dialog window Signals tab are called. After the strategy has calculated all historic bars of the chart symbol, it proceeds to Realtime mode calculation.

Strategy calculation on the next bar in sequence will result in a list of orders generated by the signal that includes all order for which the **Send()** method was called during the calculation. Then the orders process to the broker emulator to be filled based upon the current strategy and price conditions of the bar.

Optimization

Usually, a strategy consists of several signals, and each signal has its **own** inputs for each symbol, which affect the strategy operating result. It is possible to manually select the input values for each symbol, **to** achieve maximum strategy effectiveness, but this is timing consuming and inconvenient to do. In MultiCharts .NET you can optimize signal inputs automatically and receive a report with the optimization results, displaying the strategy key performance indices for each inputs combination (Net Profit, Total Trades, %Profitable, etc.). To do this, open the Format Object dialog window, select the Signals tab and then click on the Optimize button. There you can choose optimization mode:

1. Exhaustive –calculation of all input values in the specified limits with the specified step. This optimization mode requires a considerable amount of time, however, none of the input

combinations will be missed. It is also helpful, if the optimization results are analyzed by 3D Optimization Chart.

2. Genetic – searches for the best input values of a genetic algorithm that helps to find the best inputs for the minimum amount of time. Also chooses the best range and steps for signal inputs calculations.

The optimization results are available in MultiCharts .NET main menu: click View and select Strategy Optimization Report. To apply inputs, you need to double click on the line in the optimization report and set the input combination which will be applied to the strategy.

Optimization can be done upon one of the several standard criteria: Net Profit, Total Trades, Profit factor ... and others. Among these parameters there is the Custom Fitness Value parameter. It is a manual parameter, which can be calculated right in the signal script during the strategy calculation. The strategy optimization will then be executed based upon this parameter.

The value of the Custom Fitness Value parameter is set through the `double CustomFitnessValue { set; }` signal property. The last value is used during the optimization, the value that was set by the signal to the end of the strategy calculation on all the data series. If CustomFitnessValue is calculated by several strategy signals, then as the calculation result the value, set by the signal, that was calculated last, is used.

During the strategy calculation, in optimization mode, there are several features, which should be noted, when you write signal scripts:

1. Several strategy instances are created in parallel during the optimization. And they are calculated in different threads, that is why global (static) variables should be used with care. MultiCharts .NET for optimization creates the amount of threads equal to the amount of cores of your computer. If it is necessary to do the optimization in one thread, you can use the following key: «NumberOfThreadsOnOptimization», upon the register path «HKEY_CURRENT_USER\Software\TS Support\Power Language\StudyRunner».
2. Signals cannot draw and get access to drawings during the optimization (`DrwTrendLine`, `DrwArrow` and `DrwText`).
3. Also, it is not recommended to use input-output during optimization (console operations, Output, files, etc.), as it can significantly slow down the optimization process.

Information, regardless of whether a signal is in optimization mode or not, can be seen in the **Environment.Optimizing** property, if true is returned the calculation is in optimization mode if false is returned the calculation is not in optimization mode.

4.6.5 Real-time

After the strategy calculation of all signal bars on the historical data, the strategy proceeds to calculation in RealTime mode. In case of the historical calculation: `Environment.IsRealTimeCalc = false`, and in RealTime: `Environment.IsRealTimeCalc = true`. The key difference between RealTime calculation

and historic calculation is that the sent orders at historic calculation can be executed at any possible price within the bar. In RealTime it happens only on the ticks received for the symbol.

Strategy signals calculation in RealTime is executed in the same way as historic calculation, upon the close of the next bar in sequence.

4.6.6 Calculation per Bar

Historic calculation per bar can be executed in one of the two modes:

1. Calculation upon Close of each bar and filling of the generated orders collection on the next bar. This is the default mode.
2. Calculation upon Open of each bar in the context of the previous bar and filling of the generated orders collection on this bar. This mode is enabled by the signal class attribute

```
[CalcAtOpenNextBarAttribute(true)]:
namespace PowerLanguage.Strategy {
    [CalcAtOpenNextBarAttribute(true)]
    public class Test : SignalObject {
        public Test(object _ctx):base(_ctx){}
        private IOrderMarket buy_order;
    }
}
```

Open and Time values for the next bar become available through the following functions:

Bars.OpenNextBar() or Bars.Open[-1], and Bars.TimeNextBar() or Bars.Time[-1].

But the calculation is still executed in the context of the current bar close (for example, Bars.Close[0] – will still return Close of the current bar, despite the calculation being executed at the Open of the next bar).

Example:

```
[CalcAtOpenNextBar(true)]
public class TestSignal : SignalObject {
```

Now, Open of the opened bar is available using the index [-1] or Bars.OpenNextBar():

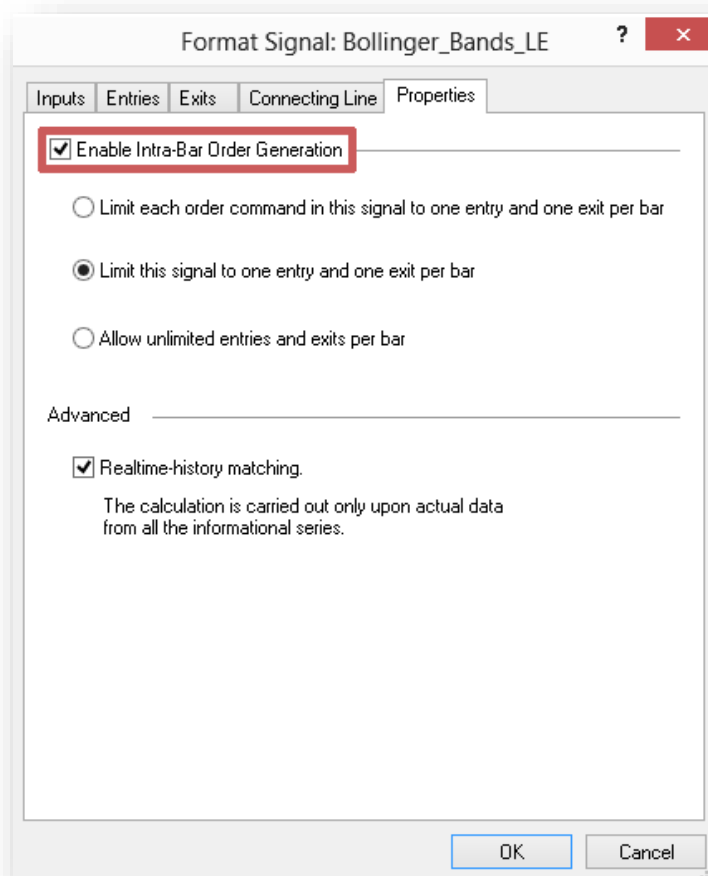
```
double o_next_bar = Bars.Open[-1]; or double o_next_bar = Bars.OpenNextBar();
```

Using index [0] the information upon the closed bar is still available.

Now, the following trading strategy can be implemented: to buy if opened with the gap down and sell if opened with the gap up:

```
protected override void CalcBar() {
    double o_next_bar = Bars.Open[-1]; //Open of the opened bar
    double h = Bars.High[0]; //High of the closed calculation bar
    double l = Bars.Low[0]; //Low of the closed calculation bar
    if (o_next_bar > h) sellshort_order.Send(); //Gap up - sell
    if (o_next_bar < l) buy_order.Send(); //Gap down - buy
}
```

If in the signal it is necessary to react at every price change within the bar, then Intra-Bar Order Generation (IOG) mode can be enabled for the signal with `[IOGMode (IOGMode.Enabled)]` the attribute or through the Format Signal dialog window at the Properties tab:



For the signal in IOG mode, the **CalcBar()** method will be called upon for every tick received for the symbol on the chart in real-time or at Open, High, Low, Close of each bar at historic calculation. If Bar Magnifier mode is enabled for the strategy, then CalcBar() will be called upon for each detailed tick. In IOG mode limits for the filling of orders can be specified for the signal:

- Limit each order command in this signal to one entry and one exit per bar, the strategy can fill each signal order once per bar.
- Limit this signal to one entry and one exit per bar, the strategy can fill only one entry order and one exit from this signal on the bar.
- Allow unlimited entries and exits per bar, the strategy can execute each signal order unlimited number of times on the bar.

Information on whether one or several strategy signals are calculated in IOG mode, can be obtained through the **Environment.IOGEnabled** property (true – IOG mode is enabled).

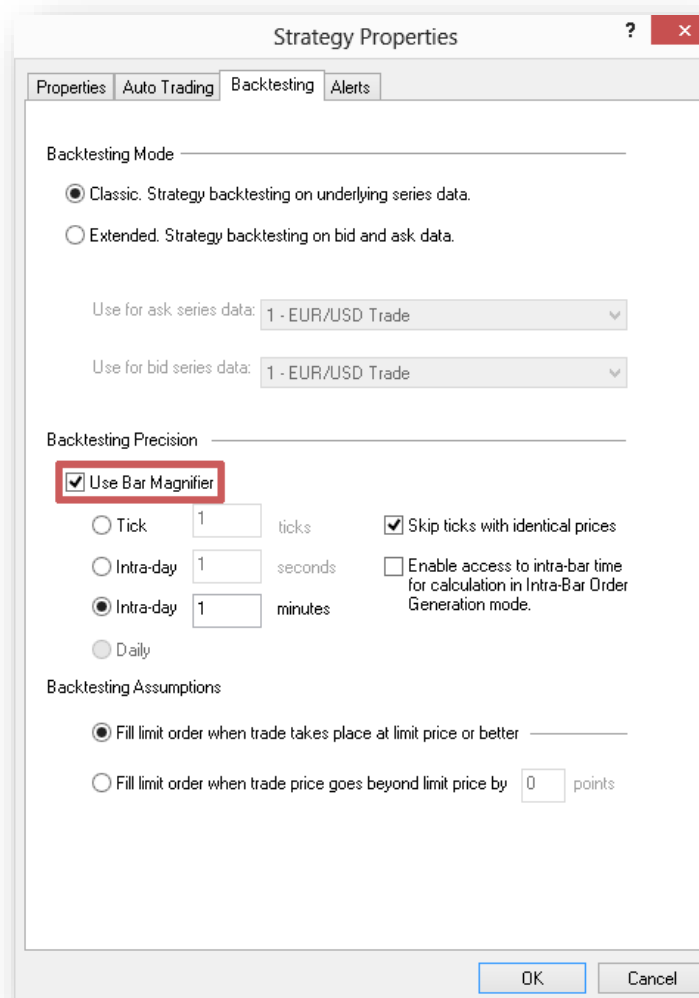
4.6.7 Price Movement Emulation within the Bar at Backtest and Optimization

During historic calculation MultiCharts .NET emulates the movement within the bar, tick by tick, upon which the order can be filled. The emulation of the price movement within the bar, by default, is executed with the [Intra-bar Price Movement Assumptions](#), upon all possible ticks.

Where the price moved at the beginning, towards High or Low of the bar, is specified by the proximity of Open to High or Low: if Open is closer to High, then ticks will be emulated upon [Intra-bar Price Movement Assumptions](#): -> High -> Low -> Close, or: Open -> Low -> High -> Close. At all the sectors of it, all possible ticks are calculated with the minimum price step. But in real-time the emulation with [Intra-bar Price Movement Assumptions](#) does not occur, every tick, received from the data source, is used.

To emulate the price movement within the bar more precisely, and to be more precise on historic calculation to real-time calculation, there is Bar Magnifier mode in MultiCharts .NET. To enable Bar Magnifier mode, select Strategy Properties, at this dialog window, click on Backtesting tab and select Use Bar Magnifier at Backtesting Precision section.

In this window you can select by tick, by second, by minute or by day for greater precision.

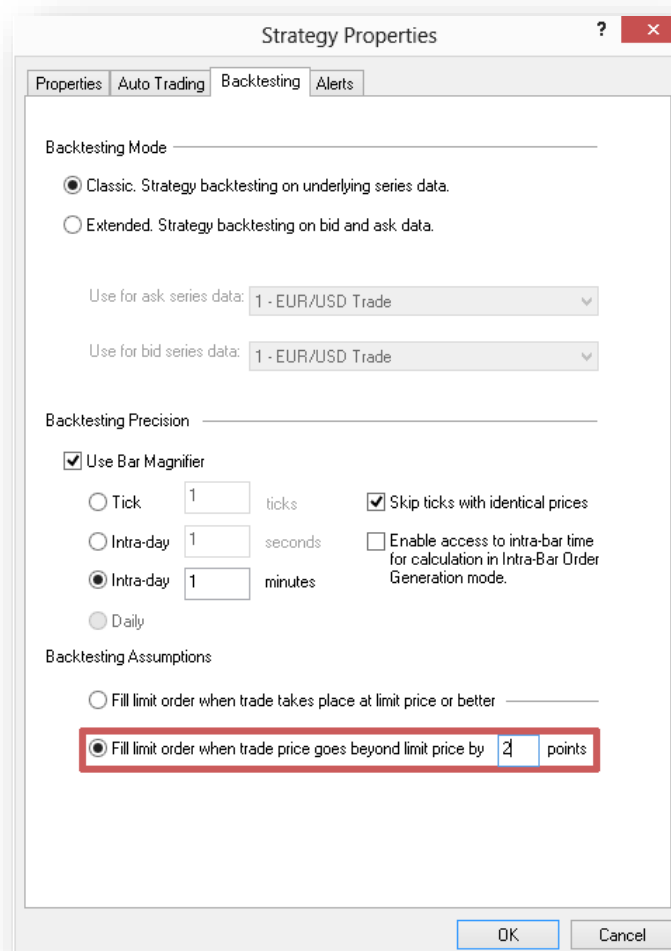


In Bar Magnifier mode, the price movement within the bar will be emulated by ticks Open-High-Low-Close or Open-Low-High-Close (with [Intra-bar Price Movement Assumptions](#), whether the Order is closer to High or Low) the smallest resolution within the detailed bar. In Bar Magnifier mode, it is considered that there are no ticks between the detailed resolution ticks.

There is also an Extended Backtesting mode in MultiCharts .NET, in which, the strategy still analyzes the signal data series (Data Series 1) to which the strategy is applied, but also executes upon the prices of an additional bid/ask data series (this can be any subsequent data series such as Data Series 2, 3, 4, 5, etc.) on the same chart that may be needed for the real trade emulation using symbols with big spread (for example, Forex). In Bar Magnifier Mode the main data series, Data Series 1 and the additional bid/ask data series; in this case Data Series 2 will be detailed. The calculation will be done using only Data Series 1 bars. The orders will be filled using the bid/ask ticks of Data Series 2 with the buy orders executed using the ask prices and the sell orders executed using the bid prices. The OpenPL will also be calculated using bid/ask prices of Data Series 2 and not the bar prices of Data Series 1.

4.6.7.1 Price Movement Assumptions

To emulate the lack of volume for the immediate Limit orders filling (if we want to emulate the trade on an illiquid instrument), the following rule can be specified: to fill Limit orders when the price crosses the Limit level by a specified number of points:



Using this mode, a Limit order will be filled upon Limit price and better, only if there will be the price on the bar that is better than the Limit by specified number of points.

Example 1:

Parameters of the symbol MinMove= 1, PriceScale=0.01, Backtesting Assumptions –classic, when the volume is always enough for the Limit application execution (Fill limit order when trade takes place at limit price or better) or Fill limit order when trade price goes beyond limit price by 0 point.

Let's try to fill Sell Limit order with the price 12.44 on the bar with the prices: Open=12.23 High=12.45 Low=11.05 Close=11.96.

Then, check if the order can be filled at Open. As $\text{Open} < \text{Limit price}$, then this order cannot be filled at this price.

Let's see, if the order can be filled at all the ticks from Open (12.23) to High (12.45), as Open is closer to High, than to Low.

12.23 – does not satisfy.

12.24 – does not satisfy.

...

12.43 – does not satisfy.

12.44 – does not satisfy this order. This order is filled at price 12,44.

Example 2:

Parameters of the symbol MinMove=1, PriceScale=0,01, Backtesting Assumptions – lack of volume emulation for the immediate Limit order filling (Fill limit order when trade price goes beyond limit price by 2 points).

Let's try to fill the Sell Limit order at the price 12.44 on the bar with the prices: Open=12.23 High=12.45 Low=11.05 Close=11.96.

1. Check, if it is possible to fill the order at Open. As $\text{Open} < \text{Limit price}$, then the order cannot be filled at this price.
2. Check, if it is possible to fill the orders at all the ticks from Open (12.23) to High (12.45), as Open is closer to High than to Low.
12.23 – does not satisfy.
12.24 – does not satisfy.
...
12.43 – does not satisfy.
12.44 – the price satisfies, but the order will be filled at this price only in case, if the price on the given bar is better by 2 points.
12.45 – the price satisfies the order, but it is only one point higher than the Limit price. The order at 12.44 is not filling yet.
3. Let's check if the order can be filled at the sector of High->Low of the bar.
12.45 – the price satisfies the order, but it is only one point higher than the Limit price. The order at 12.44 is not filling yet.
12.44 – the price satisfies, but the order will be filled at this price only in case, if the price on the given bar is better by 2 points.
12.43 – does not satisfy.
...

So, the order on this bar cannot be filled according to the chosen Limit orders execution emulation settings.

4.6.8 Commissions and Slippage

As during the live trading, Stop and Market orders are generally filled at a worse price than they were placed. In MultiCharts .NET there is slippage for Stop and Market orders. This parameter is set in Strategy Properties, open the Strategy Properties dialog window and select Slippage field. This parameter can work in two modes: per Contract or per Trade. Slippage does not affect the order execution price, it affects only the trade profit (entry+exit):

For example, we bought 100 contracts with the Market order at the price 12.5 (BigPointValue = 10), and then closed the position with the Limit order at the price 12.7.

The profit, excluding slippage, is : $(12.7 - 12.5) * 10 \text{ contracts} * \text{BigPointValue} = 0.2 * 100 * 10 = \200 .

If the Slippage is specified \$1 per Trade, then the net profit of this trade is \$200 - \$1 (from the entry, as it is the Market) - \$0 (from the exit, as it is the Limit and cannot be filled worse) = \$199. If the slippage is set at \$0.1 per Contract, then the net profit of this trade will be \$200 - \$0.1*100 contracts (from the entry, as it is the Market) - \$0*100 contracts (from the exit, as it is the Limit and cannot be filled worse) = \$190.

In real trading, the broker and the exchange, as a rule, charge commissions for the trading operations. In MultiCharts .NET trading operations commission charge emulation also exists. Commission charging rule is selected in the Strategy Properties dialog window in the dropdown Commissions list. If there is no necessary commission charging rule in that list, click Manage Commission Rule button and create the new commission rule or edit the existing one. After it, the rule will become available in the dropdown menu. The commission in MultiCharts .NET is always taken from every order, regardless its type (Stop, Limit, Market, StopLimit), when it is filled. Commission does not affect the order filling price but does affect the trade profit (entry+exit), just like Slippage.

4.6.9 Advanced. How The Strategy Backtesting Engine works

At historic calculation, the Backtesting Engine calculates every bar of the chart in order, it opens, emulates real-time within the bar and closes the bar. At the bar close the Backtesting Engine calculates all the strategy signals. The strategy calculation results in the collection of orders generated by the signals during the next calculation. MarketThisBar orders are also executed here at the bar close. Then, the new bar opens, Backtesting Engine fills MarketNextBar orders using Open of the bar, and after that, upon every tick within the bar it fills price and algorithm orders. Then, Backtesting Engine closes the bar, calculates the signals and the whole process repeats once again. In Open Next Bar mode all the signals are calculated at the bar open in the context of the closed bar. In IOG mode the strategy calculates signals without IOG attribute, as usual, and signals with IOG – at every tick within the bar and every tick in real-time.

Orders, generated by the signals without IOG, are active from open to close of the bar. Orders generated in IOG mode - are active from the signal calculation to the signal calculation.

If at least one strategy signal applies to other chart symbols, then the strategy signal calculation is executed as usual more often, then it can be suggested. It is connected with the bars that are at different price data series can be closed at different time. Also, in real-time, bars at different price data series can come with

different frequency that is why the strategy signals calculation is executed at every bar close at any used calculation data series. With IOG enabled, the calculation is executed on every tick of any used data series. But not all these calculations result in actual sending of the orders for execution in Backtesting Engine when IOG is disabled. Order sending is executed only after the Open tick of the main data series comes. And the last calculation before the orders sending at Open is considered valid.

In Backtest mode several orders can be filled on the same tick in order of priority.

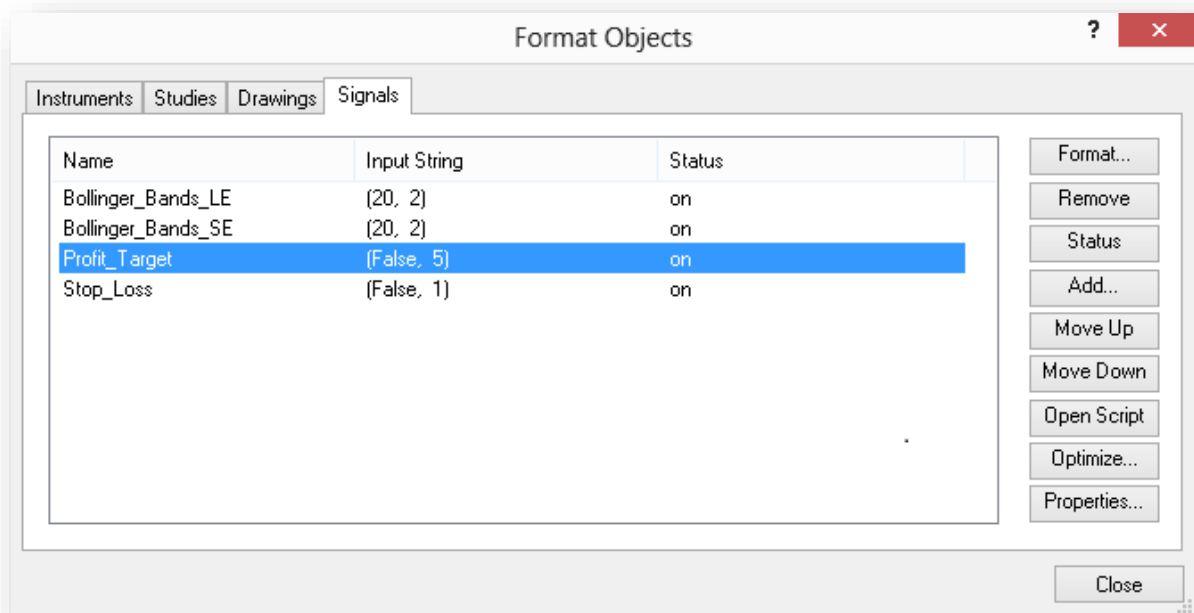
Order priority, depending on the current position, can be shown in the following priority table:

Order type	Short	Flat	Long
Buy (long position entry)	highest priority	lower priority	lower priority or won't be sent (*)
SellShort (short position entry)	lower priority or won't be sent (*)	lower priority	highest priority
Sell (long position exit)	won't be sent	won't be sent	lower priority
BuyToCover(short position exit)	lower priority	won't be sent	won't be sent

(*) – 1- if pyramiding property allows making repeated position entry and 0-if according to the strategy settings, no more entries allowed.

If the priority of the two orders is the same, then the preference will be given to the one that was generated first by the strategy.

Signal calculation priority in a strategy is determined by their order in Format Objects dialog window under the Signals tab.



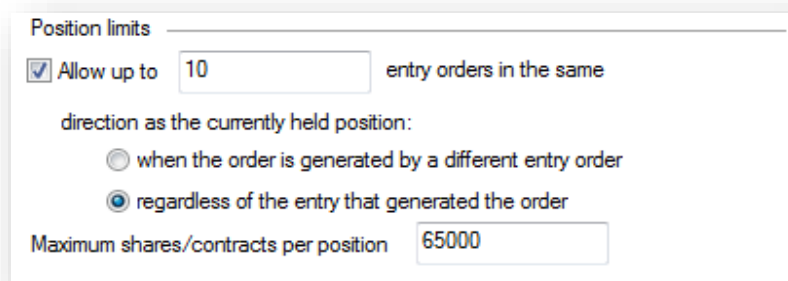
Signal calculation order can be changed by clicking Move Up and Move Down buttons.

A number of simple rules are used for the orders execution. The rules are built in the Backtesting Engine:

1. Time in force of the entry order ends right after its execution and cannot be executed again until its next generation by the signal.
2. Reverse order always closes the previous position and opens the new one per a specified or automatically calculated number of contracts.
3. Time in force of the exit order continues after its execution* (it can be executed again, for another entry, even if it was already executed). One and the same exit can be applied to one and the same entry only once*, otherwise, partial entry execution will not be possible.

(*) – exit mode for the signal can be changed using the attribute `[ExitFromOneEntryOnce(ExitFromOneEntryOnce.FillAndKillExitOrder)]`. In this mode time in force of the exit ends right after the execution, but can be applied to the same entry several times. This mode for the exits can be useful for the close in parts position strategy simplification without applying a big number of exit orders.

4. By default, the Backtesting Engine forbids repeated entry of the position, but in the Strategy Properties dialog window repeated entry can be allowed in one of the two pyramiding modes by selecting Allow up to XXX entry orders in the same.
 - Different orders are allowed to repeatedly enter only when the order is generated by a different entry order.
 - Regardless of the entry that generated the order positions increasing is allowed by one and the same order, repeatedly.



5. The number of contracts in the open position cannot exceed the Maximum shares/contracts per position settings. If the next executed entry exceeds this limit, the Backtesting Engine will automatically cut the quantity of contracts to the level specified in the strategy settings. For example, we have an open position for 60,000 contracts. One more entry for 10,000 contracts is executed. The Backtesting Engine will cut the number of contracts of this order to 5,000 and the limit condition will be executed. The following entries to the same position cannot be executed until at least one exit is executed even though the pyramiding limit is not been reached.

The Backtesting Engine also calculates the number of contracts for the orders if it was not specified at the generation of the order in the **Send()** method.

Calculation of contracts for the entry

The calculation of the number of contracts for the entries, by default, is executed according to the strategy settings (Strategy Properties dialog window, at Properties tab, Trade size sector).

Contracts calculation for the entry can be executed upon one of the two algorithms:

- 1) Fixed Share/Contracts – fixed number of entry contracts is set.
- 2) Dollars per Trade XXX round down to nearest YYY shares/contracts. Minimum number shares/contracts ZZZ (enter at \$ XXX with lots of YYY contracts. The minimum pack is ZZZ of contracts).

For example: enter at \$1000 with lots of 10 shares, but the minimum is 20 shares. As the calculation price of the contracts number, the maximum price from the last known received price that satisfies the order is used (in general Close case, or for IOG – the last tick, at which the calculation was executed) and the order price (if it is a price order). For a StopLimit order, the Limit price is used for the calculation.

For entries where there is a limit for the position in the strategy setting: Maximum shares/contracts per position, if the next entry at its execution exceeds the specified limit of contracts per position, the number of contracts for this entry will be cut to the limit, when the limit is satisfied.

For example: the limit is 65,000 contracts per position. The position for 60,000 contracts is opened already and the strategy tries to execute an entry for 10,000 more contracts. In this case, the strategy will automatically cut the number of contracts for this entry to 5,000.

A reverse order is the entry of the opposite position to the opened one. The number of contracts specified for the entry, in this case, contracts for the current position close are added.

Calculation of contracts for the exits

If the number of contracts is not specified for the order, in [OrderExit.FromEntry](#), then by default the exit always contains the number of contracts necessary for the position to close or enter. If the specified number of contracts to enter is larger than is necessary for the position to close or open, the number of contracts will be cut to the required level.

Exit types:

- 1) Exits by default have [OrderExit.FromAll](#) type. In this case, the order should fill each entry at the specified number of contracts. It works as following:

buy1 5 contracts

buy2 1 contract

buy3 2 contracts

The script sends the exit Sell FromAll for 2 contracts. The final order will be:
Sell 5 contracts:

2 contracts from buy1

1 contract from buy2 (this entry is always open at 1 contract and it is not possible to exit from it at 2 contracts)

2 contracts from buy3

After the exit execution only one entry remains open, Buy1 at 3 contracts. Other entries will be closed by that exit.

If the classic exit calculation mode is enabled, it will not be possible to close Buy1 with this order because it was already applied to this entry. Therefore, another exit order should be used to close this entry or to enable exit calculation mode, [ExitFromOneEntryOnce.FillAndKillExitOrder](#), then on the next bar that entry can be completely closed with the same order.

- 2) [OrderExit.Total](#). closes the entries in order, starting from the first one to the last one, until the specified number of contracts for the exit is executed or there will be no exit at all. It works as following:

There are three entries:

Buy1 2 contracts

Buy2 1 contracts

Buy3 5 contracts

The script sends Sell Total order at 5 contracts. The final order will be Sell at 5 contracts:

2 contracts from buy1 and it will close completely with 3 contracts remaining.

1 contract from buy2 and it closes completely with 2 contracts remaining.

2 contracts from buy3 but it still remain open at 3 contracts as all 5 exit contracts were executed.

After its execution, buy3 at 3 contracts will remain open.

- 3) [OrderExit.FromEntry](#). The particular entry order at the specified number of contracts is closed or cut down to the remaining number of contracts.
- 4) Special exit types: In PerPosition mode, one exit is sent with the number of contracts equal to the number of opened contracts per position.
In PerContracts mode the same number of exits is sent as the number of open entries in the position at the number of contracts which equal the number of not closed entry contracts.

4.6.10 Advanced Portfolio

In Portfolio Backtester, unlike trading from the chart, several strategies can be calculated simultaneously, each one with its own tool set and its own signal set. Thus, the calculation of all the strategies upon all the data series in them is synchronized.

True portfolio backtesting means that strategies are calculated across all price series, one bar at a time. For example, if you have one minute bars for five symbols, the strategy will first calculate on the first minute of each symbol, then the second, and so on.

All of the signals, even those where additional data series are used, are calculated upon the calculation rules at several DataStreams. [The orders are filled at the opening of the next bar on the Data1, the same as in the charts.](#)

Let's consider the following example and create a simple signal:

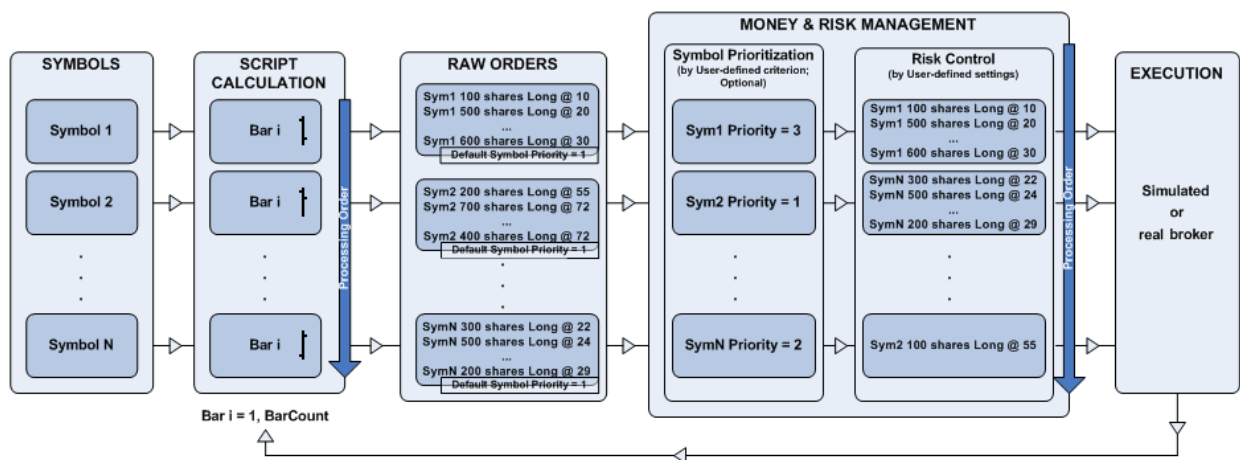
```
public class pf_test : SignalObject {
    public pf_test(object _ctx):base(_ctx){}
    protected override void CalcBar() {
        Output.WriteLine("Symbol:{0} Resolution:{1} CurrentBar:{2}
Date:{3}", Bars.Request.Symbol, Bars.Request.Resolution.Size,
Bars.CurrentBar, Bars.Time.Value);
    }
}
```

The above example will write the Symbol, resolution, bar and bar time at which the signal is calculated.

Now, let's apply several instruments with different resolutions in the portfolio in Data1 column, for example: MSFIT 1,2,3,4 days, then, calculate the portfolio and look in Output Power Language .NET:

```
Symbol:MSFT Resolution:1 CurrentBar:1 Date:06.02.2013 16:00:00
Symbol:MSFT Resolution:1 CurrentBar:2 Date:07.02.2013 16:00:00
Symbol:MSFT Resolution:1 CurrentBar:3 Date:08.02.2013 16:00:00
Symbol:MSFT Resolution:2 CurrentBar:1 Date:08.02.2013 16:00:00
Symbol:MSFT Resolution:3 CurrentBar:1 Date:08.02.2013 16:00:00
Symbol:MSFT Resolution:4 CurrentBar:1 Date:08.02.2013 16:00:00
Symbol:MSFT Resolution:1 CurrentBar:4 Date:11.02.2013 16:00:00
Symbol:MSFT Resolution:1 CurrentBar:5 Date:12.02.2013 16:00:00
Symbol:MSFT Resolution:2 CurrentBar:2 Date:12.02.2013 16:00:00
Symbol:MSFT Resolution:1 CurrentBar:6 Date:13.02.2013 16:00:00
Symbol:MSFT Resolution:3 CurrentBar:2 Date:13.02.2013 16:00:00
Symbol:MSFT Resolution:2 CurrentBar:3 Date:14.02.2013 16:00:00
Symbol:MSFT Resolution:4 CurrentBar:2 Date:14.02.2013 16:00:00
```

As we can see, the strategy on the symbol with lower resolution is counted more often than on other symbols, as a result all of the data series are calculated in sequence, from left to right and from the top to the bottom in chronological order.



Additionally, in portfolio the new number of default entry contracts calculation algorithm appeared: Percent of Equity:

Equity = Portfolio Initial Capital + Portfolio Net Profit – Portfolio Invested Capital.

The amount for the entry is calculated as following:

DollarsPerEntry = (PercentOfEquity / 100) * Equity;

The number of contracts for the entry EntryContracts = DollarsPerEntry / (EntryPrice*BigPointValue).

In the portfolio, unlike the strategy calculation on the chart, there are additional limits for the maximum number of contracts. This can be set in the Portfolio Properties tab:

Initial Portfolio Capital \$ - the initial capital for the whole portfolio common for all strategies in the portfolio.

Exposure % - the amount that we can risk in the whole portfolio.

Max % of Capital at Risk per Position % - the amount that we can risk upon separate positions in the portfolio.

Margin per Contract % - the amount in percent from the investment capital that will be frozen as a warranty provision.

Potential Loss per Contract – the amount in percent or money per contract from the investment capital that will be frozen for the potential loss. This parameter can be set individually for the trade symbol straight from the script and can be changed dynamically upon the calculation:

Example:

```
protected override void CalcBar() {
    // strategy logic
    Portfolio.SetMaxPotentialLossPerContract(Bars.High[0] - Bars.Low[0]);
}
```

The parameter of the SetMaxPotentialLossPerContract function can be expressed either in money or in percent depending upon how it was set for the portfolio in the settings of the Portfolio Settings tab.

Please note that this parameter, once set in the SetMaxPotentialLossPerContract function, will be applied to the following calculation of this symbol until it is changed again in the SetMaxPotentialLossPerContract function. If the function is not called at all, then the Potential Loss per Contract set in the Portfolio Settings is used.

In the same interface, **IPortfolioPerformance**, and many other useful properties and functions can be found. With the help of these properties and functions you can learn the current condition of the whole portfolio and evaluate risks from the script, CalcMaxPotentialLossForEntry (int side, int contracts, double price).

Example:

We have 2 symbols in the portfolio.

InitialCapital = \$100, 000

Exposure = 50%

MaxCapitalRiskPerPosition = 30%

Margin = 10%

PotentialLoss = \$1 per contract

Commission = \$10 per order.

Slippage = \$0.

At the moment of the calculation:

NetProfit = +\$400

OpenProfit = -\$100

At the first symbol (BigPointValue = 10) the position was opened with 2 orders:

Buy 100 contracts at \$26.34. Position\$ = $\$26.34 * 100 * 10 = \$26,340.00$. Let's count the amount of money to be frozen for margin provision and potential loss: FreezeMargin = $\$26,340.00 * (10\%/100) = \$2,634.00$; FreezePotLoss = $\$1 * 100 = \100

Buy 200 contracts for \$24.12 Position\$ = $\$24.12 * 200 * 10 = \$48,240.00$ Frozen: FreezeMargin = $\$48,240.00 * (10\%/100) = \$4,824.00$; FreezePotLoss = $\$1.00 * 200 = \200.00

In total, the buying power decreased by: $\$2,634.00 + \$100.00 + \$4,824.00 + \$200 + \$1.00$ (Commission entry1) + \$1 (Commission entry2) = \$7.760.00

At the second symbol (BigPointValue = 1) the position was opened with 1 order:

SellShort 100 contracts for \$112.40. Position\$ = $\$112.40 * 100 * 1 = \$11,240.00$ Frozen: FreezeMargin = $\$11,240.00 * (10\%/100) = \$1,124.00$; FreezePotLoss = $\$1.00 * 100 = \100.00

In total, the buying power is decreased more by: $\$1,124.00 + \$100.00 + \$1.00$ (Commission) = \$1,225.00

Altogether, the buying power is decreased by: PortfolioFreeze = $\$7,760.00 + \$1,225.00 = \$8,985.00$

The capital at the moment of calculation = InitialCapital+NetProfit+OpentProfit = $\$100,000.00 + \$400.00 - \$100.00 = \$100,300.00$.

Because part of the money is frozen for the warranty provision and for the potential loss and considering the maximum risk (Exposure %), the available funds in the portfolio (buying power) are:
 $\$100,300.00 * (50/100) - \text{PortfolioFreeze} = \$100,300.00 * 0.5 - \$8,985.00 = \$41,165.00$

After the calculation of all the strategy signals in the portfolio, for each strategy (the line in the portfolio) there is its own orders collection (which is the same as the strategy calculation on the chart). Then, for each strategy where there are entries the priority is calculated (expression), which can be specified in the signal through the **PortfolioEntriesPriority** property.

Example:

```
protected override void CalcBar() {  
    PortfolioEntriesPriority = new Lambda<double>(Math.Abs(Bars.Close -  
    Bars.Close[1])/Bars.Close[1]);  
}
```

If the expression is not set the priority for the strategy is calculated as 0 and the strategies are sorted according to their priority. That is, if two strategies have the same priority, the strategy that is higher in the list will be calculated first and the number of contracts for the entry orders (Buy, SellShort) is calculated according to the Money Management parameters and the current portfolio condition:

For example, on the first symbol, we calculated PortfolioEntriesPriority = 0,023; on the second, PortfolioEntriesPriority = 0,027

It means that, at the beginning the entries from the second signal are calculated as follows:

On the second symbol, we try to reverse from the short position, 100 contracts, into the long position for 300 contracts, using the order buy 300 at \$110.50:

As the order is a reverse, the short position will be completely closed at 100 contracts and the long position will be opened at 300 contracts.

The previous position will be closed at the price of \$110.50 at 100 contracts.
But at the order execution the commission charged was \$1.00.

So, potentially at Equity in the portfolio, the value will change at the commission of \$1.00.

The frozen funds for Short position will be released and for the long position entry we will have:
 $(\$100, 300.00 - \$1.00) * (30\%/100) = \$30,089.70$
For every bought contract will be frozen:

For warranty provision: $\$110.50 * \text{BigPointValue} = \$110.50 * 1 = \$110.50$

For potential loss: \$1.

Overall, for each contract at the price of \$110.50 the following will be frozen: $\$110.50 + 1.00\$ = \111.50 .

Maximum number of contracts that can be purchase at \$110.50: $\$30,089.70 / \$111.50 = 269$ contracts.

So, the entry will be cut from 300 contracts to 269 and the final order will be executed at $100 + 269 = 369$ contracts, instead of 400.

Additionally, in case of that order execution, for the warranty provision and potential loss will be frozen:
Position\$ = $\$110.50 * 269 * 1 = \$29,724.50$ Frozen: FreezeMargin = $\$29,724.50 * (10\%/100) = \$2,972.45$;
FreezePotLoss = $\$1.00 * 269.00 = \269.00 and commission charged \$1.00.

Totally, in case of this order execution, on the second symbol, we will risk: $\$2,972.45 + \$269.00 + \$1.00 = \$3,242.45$.

The maximum risk is 50%(Exposure%) from that amount for the portfolio:
 $\$100,300.00 * (50\%/100) = \$50,150.00$;

From it, \$7,760.00 are already frozen from the first symbol and on the second symbol \$3242.45 will be frozen.

Available capital for the portfolio:
 $\$50,150.00 - \$7,760.00 - \$3,242.45 = \$39,147.55$

Now, the entries on the first symbol will be calculated:

The maximum capital for this symbol according to the settings is:
MaxCapitalRiskPerPosition = 30%:

$\$100,300.00 * (30/100) - \$7,760.00 = \$22,330.00$

For every bought contract will be frozen:

For warranty provision: $\$25.00 * \text{BigPointValue} = \$25.00 * 10 = \$250.00$

For the potential loss: \$1.00

Also, the commission charged at the order execution is \$1.00.

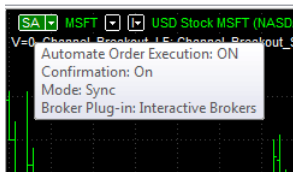
For each contract at the price of \$250.00 will be frozen - $\$250.00 + \$1.00 = \$251.00$.

We can buy more at price the \$25.00, considering the limit per position: $(\min(\$39,147.55, \$22,330.00) - \$1.00(\text{commission})) / \$251.00 = \$22,329.00 / \$251.00 = 88$ contracts.

The final order, instead of buy for 100 contracts, will be buy for 88 contracts.

If there are no funds available for the entry then this entry will not be executed at the current calculation. The funds can become available in case of partial or complete position(s) close of the portfolio. Trading with borrowed funds in cannot be executed in the portfolio.

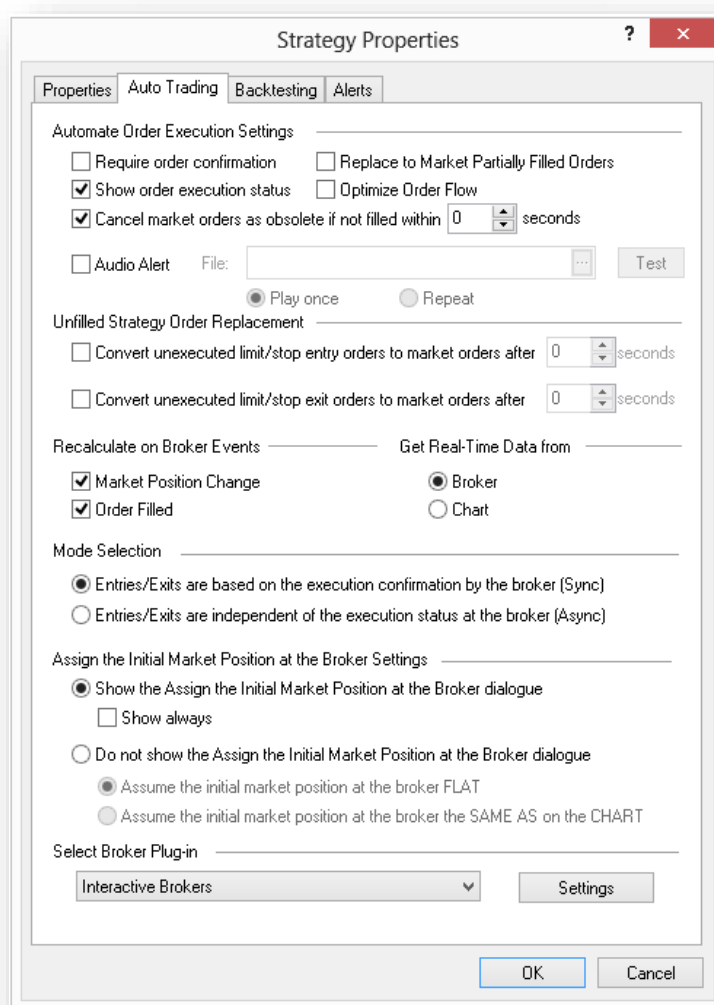
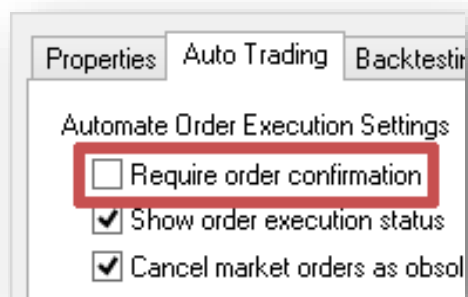
4.6.11 Advanced. AutoTrading



Using Backtesting, you can test your trading ideas and manually send orders for execution to a broker, but it is neither convenient nor practical to do this. In MultiCharts .NET you can automatically send the orders for execution to the broker according to the trading strategy. After completing the writing of the strategy, debugging it and optimizing it the strategy can be activated by clicking on the SA/AA button in the top left corner of the chart. The strategy will now start sending trade orders to the broker according to the applied algorithm. By default the synchronous auto-trading mode, SA, will be enabled and will display a dialog window showing the list of orders waiting for confirmation to be sent to the broker.

To set the automated trading mode and also to choose the broker profile for automated trading, go to the Strategy Properties dialog window and select the Auto Trading tab.

Here you can disable the order sending confirmation mode by making sure the check box next to Require order confirmation is empty.



Strategy calculation in auto-trading mode can be significantly different from the similar calculation in BackTesting and live trading. The key difference between BackTesting and live trading is that the orders in

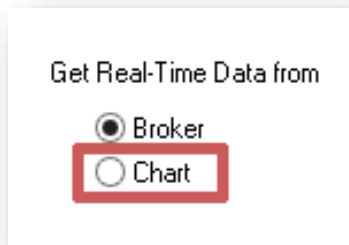
live trading cannot be physically executed immediately. In live trading the order can be executed or not executed at all until the end of its time in force, even if the price on the chart has reached its set execution price.

In Backtesting all possible market orders are executed strictly at the price at which they were sent, in Auto Trading, only one order is sent and executed, the market order with the highest priority. If market orders are followed by price orders then the command to cancel is sent to the market order immediately or according to the Cancel market orders settings.

In live trading, the price during the time of submitting a market or stop order, can change from the original order price at the Open(Close) of the bar. In reality, a market order is not generally executed at one price but at several prices. This is why its execution price may not match the minimum price step on the chart.

In auto-trading all of the price orders, which potentially can be executed, are chosen (priority > 0) and sent to the broker in OCO-Cancel group. This means that if any order starts to execute or cancel, then all other orders of the group will immediately be cancelled. This guarantees the execution of only one order in the group if a broker supporting native OCO is used. Otherwise, the group is emulated on the MultiCharts .NET side and the execution of several orders in the group becomes possible if the prices are close which may result in all orders of the group not being cancelled immediately. If the group contains only the exits they are sent in the OCO Reduce Size group upon the execution of one of the orders, other orders will have the number of contracts reduced automatically, that will allow to hold simultaneously at the broker, for example, ProfitTarget and to insure the loss with StopLoss order. Until the position completely closes, it will be always secured.

Also, the Auto Trading Engine and BackTesting Engine calculate the current strategy condition independently (MarketPosition, AvgPositionPrice, OpenPositionProfit, TradeProfit. Thus, OpenPositionProfit is calculated in relation to the Bid/Ask prices of the broker. Because the prices on the chart can differ from the broker bid/ask prices, MultiCharts .NET transfers the prices from the chart to Auto Trading by default.



Optimization of orders sent to a broker works during automated trading. If after subsequent strategy calculation the same collection of the orders were generated or the set of the orders was left the same but only some parameters were changed (price, number of contracts), then the group will not be replaced and the parameters of changed orders are only modified. Also, price orders can remain on the broker side if on a subsequent calculation no more market orders are generated. But if the number of orders changes after generation, the whole group will be canceled and replaced due to the fact that it is impossible to add or remove the orders from the group. If one or more orders are partially filled within a single group, these orders will be considered by the auto trading system as filled and another order generation will be performed.

NOTE: Partially filled orders may not fill completely within cancellation. To guarantee that an order is filled completely in automated trading, MultiCharts .NET has feature called, "Replace to Market Partially Filled Orders", that converts partially filled orders into market orders. This feature can be enabled in the Auto Trading tab of the Strategy Properties dialogue box. The Strategy calculation scheme within automated trading is the same as the calculation in Backtesting Real-time.

Within automated trading there are additional events per calculation:

1. Market Position Change -the average position price and/or number of open contracts at the broker has changed.
2. Order Filled- a broker returns an event indicating that an order sent by a strategy has been filled. This event will not be returned for orders not sent by a strategy.

To make a strategy react on these events it is necessary to enable the corresponding checkboxes on the Auto Trading tab of the Strategy Properties dialogue box. By default these two calculation events are enabled. The reason for calling this method can be checked using `Environment.CalcReason`.

When the signal is calculated on the broker order execution event: `Environment.CalcReason = CalculationReason.OrderFilled`.

When the signal is calculated on the broker market position change: `Environment.CalcReason = CalculationReason.MarketPositionChange`.

The parameters of the order execution at the broker shown under "OrderFilled" event can be checked using the reloaded method `protected override void OnBrokerStrategyOrderFilled(bool is_buy, int quantity, double avg_fill_price)`.

It is possible to use the override method for market position processing and changing at the broker in the same way:

```
void OnBrokerPositionChange();
```

Example:

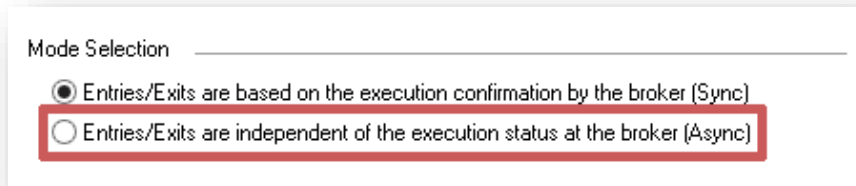
```
protected override void OnBrokerPositionChange() {  
    Output.WriteLine("MP={0}", Avg={1}"),  
    this.StrategyInfo.MarketPositionAtBroker,  
    this.StrategyInfo.AvgEntryPriceAtBroker);  
}
```

Automated trading in MultiCharts .NET is possible in one of the following modes: Synchronous, enabled by default, and Asynchronous.

In Synchronous automated trading mode both the BackTesting Engine and the Auto Trading Engine start trading with a zero (flat) market position and live transmission of the orders to the broker starts (only in real-time) after the historical calculation has ended. The BackTesting Engine does not execute any orders and only takes the orders filled at the broker which is why no orders can be executed on the historical calculation and the market position in both BackTesting and Auto Trading will be zero (flat) at the start of live trading. If at the start of auto trading is it desired to have the market position not set at than zero, select the "Show the Assign the Initial Market Position" on the Broker dialogue box and check the "Show Always" checkbox in Auto Trading settings of Strategy Properties. Now, when auto trading starts it will assign the

initial market position and current max profit (for special orders) on the current position. The BackTesting and Auto Trading Engine will start real time trading from the assigned market position.

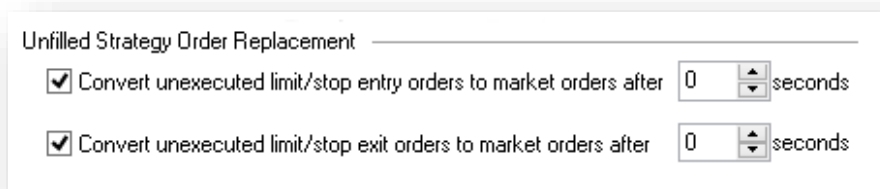
In Synchronous auto trading mode there are no orders on the historical part of the traded instrument. If the strategy depends on previous fills (for example, exits from particular entries and at the previous auto trading session the exits were not closed) the Asynchronous auto trading mode must be used. To select Asynchronous auto trading mode click on “Entries/Exits are independent of the execution status at the broker (Async)” found under Mode Selection on the Auto Trading Tab of the Strategy Properties dialog window.



To make Auto Trading automatically check the current state of the strategy, open entries, OpenPL, etc., by the end of the historical calculation without any additional dialogue windows, select “Do not show the Assign the Initial Market Position at the Broker dialogue” and set “Assume the initial market position at the broker the SAME AS on the CHART” mode found on the Auto Trading Tab of the Strategy Properties dialog window. After the historical calculation, the automated trading system will receive the information about the open position and the orders that formed it.

In Asynchronous auto trading mode, if the price level is reached the order may not be filled. That’s why the position on the chart in the BackTesting Engine may change but will not in Auto Trading.

There is another case when BackTesting may fill the order completely but a broker may not. It can lead to the strategy position and the broker position mismatch. To handle this situation, it is necessary to enable the “Convert unexecuted limit/stop orders to market orders” feature (to guarantee that the order will be executed for sure):



This conversion mode works in Synchronous mode in the same way. Order execution in this case is tracked by the broker’s prices or chart depending on the auto trading setting “Get Real-Time Data from”. If the price reaches the level of the order and the order is not executed within a set period of time, the order will be cancelled and a similar market order will be sent in place of the cancelled order. The Auto Trading system waits for its execution before continuing calculations.

It is possible to check whether a strategy is calculated in real time via the **Environment.IsAutoTradingMode** property.

If the strategy is automated: `Environment.IsAutoTradingMode=true`.

In automated trading mode a signal can access three market positions via `StrategyInfo`:

1. Current position on the chart in the BackTesting Engine:

```
StrategyInfo.MarketPosition;  
StrategyInfo.AvgEntryPrice;
```

2. Current position in AutoTrading system, for the current strategy:

```
StrategyInfo.MarketPositionAtBrokerForTheStrategy;  
StrategyInfo.AvgEntryPriceAtBrokerForTheStrategy;
```

3. Current position for the traded instrument at the broker:

```
StrategyInfo.MarketPositionAtBroker;  
StrategyInfo.AvgEntryPriceAtBroker;
```

By noticing the differences between position on the chart and in the Auto Trading System it is possible to notice the mistiming in Asynchronous mode. Each order execution affects both position in the Auto Trading System and the position on the chart in the BackTesting Engine. To synchronize the broker position to the position on the chart there is a command used for sending a market order into the Auto Trading System only, not to the chart:

```
GenerateATMarketOrder(bool buy, bool entry, int lots).
```

This command is executed right at the script execution that allows the script to react to the mistiming between the chart and broker position in a timely manner.

An example of such synchronization can be found in a pre-built signal:

```
From_Strategy_To_Broker_MP_Synchronizer.
```

Also in MultiCharts .NET it is possible to change the position on the chart while not changing the AutoTrading System position and without placing odd orders at the broker:

```
void ChangeMarketPosition(int mp_delta, double fill_price, string _name).
```

This command, like `GenerateATMarketOrder`, is executed within the current calculation of the script. An example of synchronization with the `ChangeMarketPosition` keyword can be found in a pre-built signal `From_Broker_To_Strategy_MP_Synchronizer`.

4.7 Advanced. Special Features

4.7.1 Chart Custom Draw

Every study can plot a custom drawing in a chart window. This is possible using the **IChartCustomDrawRegistrator** interface that is available through the **ChartCustomDraw** property.

This interface contains methods for registering/unregistering a custom drawer, the initiation of the drawing of the entire chart window and a property for accessing the data that is required during the drawing. It is possible to register/unregister the custom drawer from any thread. Re-plotting initiation is also available from any thread and will be started asynchronously in the main thread, i.e. not as soon the method is called.

Custom drawer is an object that implements the **IChartCustomDrawer** interface with the only one **void Draw(DrawContext context, EDrawPhases phase)** method. This method will be called several times within one process of chart window plotting, once per each phase of plotting. The plotting process is performed on a phased basis:

- 1) background is filled,
- 2) grid is plotted,
- 3) background figures are plotted (rectangles, ellipses, etc)
- 4) bar series is plotted, the chart for an instrument itself
- 5) foreground figures are plotted (trend lines, Fibonacci levels, etc)

The current phase of plotting can be checked in the `phase` argument. It allows plotting at the required moment, for example, either above or below the bar series, at the background, or in front of all the objects of the chart.

The first argument of the **IChartCustomDrawer.Draw()** method context is a wrapper class that contains all the necessary data and means for the plotting, they are:

`public readonly Color Background;` - chart background color.

`public RectangleF DirtyRect;` - Custom drawer can require to re-plot a bigger space than is currently being plotted and can notify the caller about that by setting the required space in this property. If on exiting from the **IChartCustomDrawer.Draw()** method this space is bigger than the one set in `DrawRect`, then the caller will initiate the plotting procedure on all phases one more time. This double plotting allows the receipt all the required information about the objects that need to be plotted in a bigger space, thus it is being re-plotted without any visual artifacts (flashing).

`public readonly RectangleF DrawRect;` - contains the space that is required to be re-plotted. To speed up the plotting process, the whole chart window does not need to be re-plotted, only the space where the indicated objects were changed. For example, the last bar has been updated in real time, so in this case only a little space containing this bar should be re-plotted and the **DrawRect** property will contain this space.

`public readonly IDrawDataEnvironment Environment;` - this interface gives access to the data on the charts bars and status line. Also it contains the methods for conversion of the coordinates from one type to another. For example:

- a. from bar number/index to screen coordinate X
- b. from the point, date-time, price to screen coordinate point

`public readonly bool FinalDraw;` - returns TRUE when the **IChartCustomDrawer.Draw()** method will not be called within the current plotting and phase, regardless of the values of the `DirtyRect` and `ReDraw`

fields. If nothing is currently being plotted the custom drawing will not be indicated on the chart (at least for 1 fps).

`public readonly RectangleF FullRect;` contains the whole chart space where the plotting is possible. Basically, this is the whole internal space of a chart window.

`public readonly Graphics graphics;` -plotting is performed by means of GDI/GDI+ with `System.Drawing.Graphics. graphics` class, this is a reference to the object of this type. All the plots are created by calling methods of this class. More information about methods and properties of `System.Drawing.Graphics` class can be found on [msdn](#).

`public bool Redraw;` -should be set as TRUE if the whole chart window should be re-plotted.

NOTE: A pre-built indicator `_Market_Depth_on_Chart_2_` uses custom drawings and can be used as an example.

4.7.2 Chart ToolBar

The studies can activate and access the toolbar for the chart. Such a toolbar can contain custom controls. Thus it is possible to let the user set the necessary values, colors, styles, etc., for the data shown by an indicator in a fast and convenient manner. Access to the toolbar is performed via the **ChartToolBar** property that is returned by the **ICustomToolBar** interface. This interface contains methods for accessing the toolbar and the property for manipulating visibility of this toolbar. Access to the toolbar is performed by calling one of the following methods:

1. `void AccessToolBar(ToolBarAction action);`
2. `void AccessToolBarAsync(ToolBarAction action);`

The difference between these two methods is the delegate of public delegate `void ToolBarAction(ToolStrip tb)` type will be executed synchronously when `void AccessToolBar(ToolBarAction action);` and asynchronously when `void AccessToolBarAsync(ToolBarAction action)` is used.

The reason for using a delegate instead of calling the toolbar directly is simple: the study is calculated in its thread while chart toolbar as well chart controls are working in the main thread (UI).

The delegate accepts a single argument of `System.Windows.Forms.ToolStrip`, the type that is familiar to those who have dealt with toolbar creation for WinForms windows. More information about this type and its methods can be found on [msdn](#).

NOTE: A pre-built indicator `_Chart_ToolBar_Example_` and a signal `_ChartToolBar_Trading_` may be used as examples.

4.7.3 Access to the data outside the chart window. DataLoader & SymbolStorage

4.7.3.1 Access to the symbols from QuoteManager. SymbolStorage

The studies can access the list of symbols available in the database. Access to this service is performed via the **SymbolStorage** property returned by the **ISymbolStorage** interface. This interface contains methods for requesting the symbols under different criteria.

```
MTPA_MCSymbolInfo2[] GetSymbols(string data_feed);
```

Returns an array of `MTPA_MCSymbolInfo2` structures corresponding to all the instruments added into the database from a data source set as a parameter in the `data_feed`.

Example:

```
MTPA_MCSymbolInfo2[] IQFeedSymbols =  
SymbolStorage.GetSymbols("IQFeed");
```

Will return an array of all the symbols added from `IQFeed` that are available in database that can be found in the QuoteManager.

```
MTPA_MCSymbolInfo2[] GetSymbols(string data_feed, string name);
```

Will return an array of `MTPA_MCSymbolInfo2` structures corresponding to all the instruments available in the database added from the `data_feed` source and the symbol name = `name`.

Example:

```
MTPA_MCSymbolInfo2[] IQFeedSymbolsMSFT=SymbolStorage.GetSymbols("IQFeed",  
"MSFT");
```

Will return an array of all the symbols from the database with `IQFeed` as data source and the symbol name is `MSFT`.

```
MTPA_MCSymbolInfo2[] GetSymbols(string data_feed, string name,  
ESymbolCategory cat);
```

Will return an array of `MTPA_MCSymbolInfo2` structures corresponding to all the instruments in the base added from the `data_feed` source with the `cat` category.

Example:

```
MTPA_MCSymbolInfo2[] IQFeedSymbolsMSFTStock =  
SymbolStorage.GetSymbols("IQFeed", "MSFT", ESymbolCategory.Stock);
```

Will return an array of all the symbols in the database added from `IQFeed` with the name `MSFT` and the category is `Stock`.

```
MTPA_MCSymbolInfo2[] GetSymbols(string data_feed, string name,  
ESymbolCategory _cat, string Exchange);
```

Returns an array of `MTPA_MCSymbolInfo2` structures corresponding to all the instruments in database added from the `data_feed` source with `name` symbol name, `cat` category and `Exchange` exchange.

Example:

```
MTPA_MCSymbolInfo2[] IQFeedSymbolsMSFTStockCME =  
SymbolStorage.GetSymbols("IQFeed", "MSFT", ESymbolCategory.Stock, "CME");
```

Will return an array of all the instruments from the database added from IQFeed, with MSFT as the symbol name, Stock as the category and CME as the exchange.

This service can be used for obtaining all the attributes of the symbol based on its key parameters only.

4.7.4 Receiving the data for any symbol and any resolution. DataLoader.

The studies can download the data for any symbol that is available in the database even if a chart for this instrument is not created in MultiCharts .NET. This service is available via the **DataLoader** property returned by the **IDataLoader** interface. This interface contains the methods for starting and stopping the data downloading process.

To start the data downloading process it is necessary to call the following method:

```
IDataLoaderResult BeginLoadData(InstrumentDataRequest Request,  
LoadDataCallback Sink, object State)
```

where `InstrumentDataRequest` is the structure that determines the data request parameters. It contains the following fields:

```
public string Symbol - instrument name;  
  
public string DataFeed - data feed name;  
  
public string Exchange - exchange name;  
  
public ESymbolCategory Category - symbol category (for example Future, Stock, Index, etc.);  
  
public Resolution Resolution - request resolution
```

Resolution structure helps to determine both regular and irregular resolutions.

- Example of determining a regular resolution:

```
Resolution _res = new Resolution(EResolution.Tick, 10);
```

If the request resolution is set in this way then 10-tick data will be requested.

- Example of determining an irregular resolution:

```
Resolution _res = Resolution.CreatePointAndFigure(EResolution.Tick, 10,  
0.001, 3, PointAndFigureBasis.HighLow, true);
```


If the request resolution is set in this way, then the PointAndFigure resolution based on a 10-tick resolution, BoxSize = 0.001, Reversal = 3, HighLow Basis and BreakOnSession enabled will be returned.

`public DateTime From` - the start date of requested data. This field is outdated and it is better to use the `Range` field.

`public DateTime To` - the end date of requested data. This field is outdated and it is better to use the `Range` field.

`public DataRequest Range` - the range of requested data

`public RequestTimeZone TimeZone` - the time zone that will be applied to the requested data. The available values are: Local, Exchange and GMT. The Default value is Local.

`public RequestQuoteField QuoteField` - the quote field of the requested data. The available values are: Bid, Ask and Trade. The Default value is Trade.

`public string SessionName` - the name of the session that will be applied to the requested data.

`public bool FilterOutOfRangeData` - this indicates if “odd” data should be cut off. Here the “odd” data is defined as the data before the request start time and the data after the request end time.

`public bool Subscribe2RT` - this indicates whether the real time data subscription is required or if only historical data is needed.

`LoadDataCallback` - is a delegate with the following signature:

```
public delegate void LoadDataCallback(IDataLoaderResult Result);
```

It is used to call back the data request results. This function is called in a separate thread, different from one where [BeginLoadData](#) is called.

`IDataLoaderResult` - the interface that provides information in the data request results. It contains the following fields:

`bool IsCompleted { get; }` - indicates whether the data downloading process is complete or not.

`object State { get; }` - returns the current state that was set in `BeginLoadData` method.

`Bar[] Data { get; }` - returns the downloaded data as an array of bars.

`BarPriceLevels[] Data2 { get; }` - returns an array of downloaded price levels.

`InstrumentDataRequest Request { get; }` - is used in `DataLoader`. This attribute helps to make the same request for `DataLoader` that is used for the instrument bars.

`Bar? RTData { get; set; }` - returns realtime data (if available).

`BarPriceLevels? RTData2 { get; set; }` - returns realtime price levels (if available)

`DataLoadedEvent` Event { `get`; } – returns a reason for calling the callback function

`LoadDataCallback`. Available values are None, History, RTUpdateLastBar, and RTNewBar.

To cancel data downloading the **void EndLoadData(IDataLoaderResult Result)** method should be called. The Result of BeginLoadData call should be put into the argument as an attribute.

Dataloader Usage Example

The example indicator will calculate and plot the spreads on the symbols different from ones that are used on the chart where an indicator is applied. Assume that a EUR/USD chart is created and it is necessary to plot the spreads between the Ask price of EUR/JPY and the Bid price of EUR/AUD. This means that an additional data series will not be inserted into the EUR/USD chart and the the indicator will get all required information from the main data source.

1. Create the new indicator with the standard pattern:

```
namespace PowerLanguage.Indicator{
    public class DataLoader : IndicatorObject {
        public DataLoader(object _ctx):base(_ctx){}
        protected override void Create() {}
        protected override void StartCalc() {}
        protected override void CalcBar(){}
    }
}
```

2. Add the plot statements that will plot spread values:

```
private IPlotObject m_spreadPlot;
```

3. Then create a corresponding object. This should be done in Create() function:

```
protected override void Create() {
    m_spreadPlot
    = AddPlot(new PlotAttributes("", EPlotShapes.Line, Color.Red));
}
```

4. Now it is necessary to form the objects for data requests. To do that, `InstrumentDataRequest` is to be used but, there's an easier workaround for this example indicator. Remember, that the **Bars.Request** property returns the reference to the object of this structure and its fields are filled in accordance with the main data series settings. So in this example, several parameters should be changed while the rest parameters can be left unchanged.

```
InstrumentDataRequest Request = Bars.Request;

Request.Subscribe2RT = true;
```

It means that both historical and real-time data should be received.

5. Specify the request range. Here the bars back range is specified. Number of bars back in this example is equal to the number of all complete bars on the [main data series](#) chart.

```
DataRequest _DataRequest = new DataRequest();
_DataRequest.RequestType = DataRequestType.BarsBack;
_DataRequest.Count = Bars.FullSymbolData.Count;
Request.Range = _DataRequest;
```

6. Two separate requests should be done, the first request will accumulate the Ask price of EUR/JPY and the second the Bid price of EUR/AUD.

For EUR/JPY the request will be as follows:

```
Request.QuoteField = RequestQuoteField.Ask;
Request.Symbol = "EUR/JPY";
```

For EUR/AUD the request will be as follows:

```
Request.QuoteField = RequestQuoteField.Bid;
Request.Symbol = "EUR/AUD";
```

7. Before requesting the data, it is necessary to declare the call-back function that will be used for receiving the data into study:

```
public void OnData(IDataLoaderResult Result){ }
```

It will be implemented later.

8. Having combined all of the above criteria, the following StartCalc() function can be retrieved. It will perform the creating of the requests and the requests themselves:

```
protected override void StartCalc() {
    InstrumentDataRequest Request = Bars.Request;
    Request.Subscribe2RT = true;

    DataRequest _DataRequest = new DataRequest();
    _DataRequest.RequestType = DataRequestType.BarsBack;
    _DataRequest.Count = Bars.FullSymbolData.Count;
    _DataRequest.To = DateTime.Now;
    Request.Range = _DataRequest;

    Request.QuoteField = RequestQuoteField.Ask;
    Request.Symbol      = "EUR/JPY";
    Request.Exchange    = "FOREX";
    Request.DataFeed    = "LMAX";
    Request.Category    = ESymbolCategory.Forex;

    DataLoader.BeginLoadData(Request, OnData, null);

    Request.QuoteField = RequestQuoteField.Bid;
    Request.Symbol     = "EUR/AUD";

    DataLoader.BeginLoadData(Request, OnData, null);
}
```

For requesting data, as seen in the example above, it is necessary to call the `BeginLoadData` function, which contains formed request object, call-back function pointer and optional State object that will be returned into call-back function specifying its parameters.

9. Next step is to process the data. Accumulation logic should be done in the `OnData()` function. First, add a reference to the `System.Collections.Generic` namespace:

```
using System.Collections.Generic;
```

Container fields where data will be stored should be added into indicator class:

```
private List<Bar> m_EURJPY = new List<Bar>();  
private List<Bar> m_EURAUD = new List<Bar>();
```

The **`IDataLoaderResult`** interface provides access to the call-back function of `OnData()`

The reason for call-back call can be checked in `IDataLoaderResult::Event` value. The value `DataLoadedEvent.History` means that historical data has been received. The value `DataLoadedEvent.RTNewBar` means that the new real-time bar has been received.

The data processing function can be written in the following manner:

```
private void ProcessData(List<Bar> Container, IDataLoaderResult  
Result)  
{  
    switch (Result.Event) {  
        case DataLoadedEvent.History: {  
            Container.AddRange(Result.Data);  
            break;  
        }  
        case DataLoadedEvent.RTNewBar: {  
            if (Result.RTData.HasValue)  
                Container.Add(Result.RTData.Value);  
            break;  
        }  
    }  
}
```

Please note that the call-back function call is to be performed in a thread separate from one where `BeginLoadData` has been called.

`OnData` function will be implemented the following:

```
public void OnData(IDataLoaderResult Result){  
    if (Result.Request.Symbol == "EUR/JPY") {  
        ProcessData(m_EURJPY, Result);  
    }  
  
    if (Result.Request.Symbol == "EUR/AUD") {  
        ProcessData(m_EURAUD, Result);  
    }  
}
```

The call of the **ProcessData()** function of data processing for a required container will be initiated depending on the request for which the call-back call was received.

10. The final step is to make the plotting of the spreads once all of the required information has been obtained:

```
protected override void CalcBar(){
    if (Bars.Status == EBarState.Close){
        int index =
            Math.Min(Bars.CurrentBar,
                Math.Min(m_EURAUD.Count - 1,
                    m_EURJPY.Count - 1));
        if(index > 0)
        {
            m_spreadPlot.Set(
                Math.Abs(m_EURAUD[index].Close -
                    m_EURJPY[index].Close));
        }
    }
}
```

NOTE: If either VolumeDelta or CumulativeDelta are specified as ChartType in the request, then the information on Data2 will be relevant for the resulting object, if real-time is requested then RTData2 is required. Data2 is an array of BarPriceLevels.

```
public struct BarPriceLevels
{
    public Bar Bar { get; set; }
    public PriceLevel[] Levels { get; set; }
    public int Index { get; set; }
    public DateTime Time { get; set; }
    public uint TickID { get; set; }
}
```

The **Levels** property contains VolumeProfile/Volume Delta data. PriceLevel[] is an array of the price levels where Index equals bar index, Time equals bar time, and TickID equals its ID.

4.7.5 Extended Trading Possibilities. TradeManager.

4.7.5.1 Basic statements.

With the classic strategy it was possible to trade only one instrument within one trading profile. Trade Manager allows trading on several instruments, accounts or profiles; it also obtains the relevant data about the orders on the broker's side, real open positions, and account(s) information directly from the script of the study. It provides access to all information available in the Order's & Position's Tracker as well as the ability to send the orders, both regular and OCO ones, to modify them and to cancel them.

Access to Trade Manager functionality is available via the **TradeManager** property returned by the **ITradeManager** interface. This interface contains:

- 1) `void ProcessEvents();`
- 2) `void ProcessEvents(TimeSpan max_time);`

These methods initiate the processing of queued events.

- 3) `ITradingData TradingData { get; }` this provides access to orders, positions, accounts and events on these elements changes.
- 4) `ITradingProfile[] TradingProfiles { get; }` provides access to trading profiles available in MultiCharts .NET.

Each of these properties is described details below.

4.7.5.2 Receiving Data. *TradingData*.

The **ITradingData** interface contains 4 properties that provide access to different data:

```
public interface ITradingData
{
    IAccounts Accounts { get; } - provides access to account information
    which can be found at the Accounts tab of the Order and Position Tracker
    Window.
    ILogs Logs { get; } - provides access to the event list information
    which can be found at the Logs tab of the Order and Position Tracker
    Window.
    IOrders Orders { get; } - provides access to the orders list
    information which can be found at the Orders tab of the Order and
    Position Tracker Window.
    IPositions Positions { get; } - provides access to the positions
    list information which can be found at the Positions tab of the Order and
    Position Tracker Window.
}
```

Each of these properties returns a reference to the interface of the corresponding list or collection. Each of these interfaces has a single basic interface - [IDataSource](#)<T>, where T is the certain type that describes the order, account, position, etc.

The **IDataSource** interface contains:

```
public interface IDataSource<T>
{
    event TItemsChanged<T> Added; - an event that is generated when the new elements are added
    into the collection. For example, for orders it means that a new order has been placed.

    event TItemsChanged<T> Changed; - an event that is generated when the current components
    of the collection are changed. For example, for orders it means that the existing order has been changed. It
    can be a modification of its status, price, number of contracts or any other attributes.

    event TItemsChanged<T> Deleted; - an event that is generated when the current elements of
    the collection have been removed. For example, for positions it means that it has been closed or has
```

become Flat.

event [EventHandler](#) [FinishChanging](#); - an event that is generated at the end of batch changes.

event [EventHandler](#) [ReloadFinished](#); - an event that is generated after the collection has been completely changed.

event [EventHandler](#) [ReloadStarted](#); - an event that is generated before the collection is changed. It can be generated, for example, if a filter is applied.

event [EventHandler](#) [StartChanging](#); - an event that is generated before batch modifications start. Usually, "batch modification" means a single generation of each event such as Added, Changed, and Deleted events.

T[] [Items](#) { get; } – is a property for accessing all of the available components of the collection.

Please NOTE, that this property returns a copy of the data but not the reference to the place where it is stored. }

All of the above mentioned events are generated at strictly defined moments, for example, when a position or order update event is received from the broker. Respectively, the updating of the collection elements that are accessed via the **Items** property is also performed at a strictly defined moment. This moment is defined by calling the **ITradeManager.ProcessEvents()** method.

All collected events, elements collection changes, events from **IDataSource<T>** interface that start being generated, on all the elements for a current moment start being applied within this call and the current thread. The chronological order of the events within a single collection stays the same. The reason for this is that the indicator/signal is not a stand-alone application but a component that should be executed in certain conditions and environment.

All of the collection interfaces, **ILogs**, **IOrders**, and **IPositions** except for **IAccounts**, have the properties for accessing the filters. The filters allow you to exclude unnecessary elements that do not correspond to the filters conditions. There are two types of filters:

- 1) Time-based. This filter type is available in the following interface:

```
interface IDateTimeFilter
{
    DateTime From { get; set; }
    DateTime To { get; set; }
}
```

Where the "From To" properties set the complete (closed) range of the time period [From, To].

- 2) List-based. This filter type is available in the following interface:

```
interface IBaseFilter<T>
{
    bool CurrentIsAll { get; } – If TRUE then the filter value has not been set.
    T CurrentValue { get; set; } – a current filter value.
    T[] Values { get; } – the list of all the possible filter values.
```

} where T can be either a string or an enumeration.

As soon as the new value is set for any property, the collection is filtered on the new value of the filter: synchronously if the new value of the filter is deemed to be a subset of the previous value or asynchronously in all other cases.

4.7.5.2.1 *Collections elements.*

Below the elements of IAccounts, ILogs, IOrders, and IPositions collections will be described.

IAccounts collection contains the following elements:

```
public struct Account
{
    public string ID { get; } – the unique identifier of the account. Returned as Profile + Name.

    public string Profile { get; private set; } – Profile name.

    public string Name { get; private set; } – Account name.

    public double? Balance { get; private set; }
    public double? Equity { get; private set; }
    public double? OpenPL { get; private set; }
    public bool BalanceInPrcent { get; private set; } – Returns true, if Balance
returns values in percent.
    public bool EquityInPrcent { get; private set; } – Returns true, if Equity
returns values in percent.
    public double? Margin { get; private set; } – Absolute value of margin.
    public double? MarginAsPrcent { get; private set; } – Margin value in percent.
    public double? AvailableToTrade { get; private set; }
}
```

Nullable<T> fields (for example, double?) may be == null if the broker does not provide corresponding information.

ILogs collection contains the elements:

```
public struct Log
{
    public DateTime Time { get; private set; } – event time
    public string Symbol { get; private set; } – symbol name
    public string Profile { get; private set; } – profile name
    public string Text { get; private set; } – message text
    public ETM MessageCategory Category { get; private set; } – message
category (info, warning, error)
    public string Source { get; private set; } – event source name
    public string ID { get; private set; } – Event identifier, this is a number.
}
```


IOrders collection contains the following elements:

```
public struct Order
{
    public bool IsManual { get; private set; } - returns true if the order was
    generated manually, false if the order was generated by an automated system.
    public DateTime GeneratedDT { get; private set; } - time of the order creation,
    i.e. the time when the order appeared in MultiCharts .NET during the current connection.
    public DateTime? FinalDT { get; private set; } - time when the order moves to
    the final state.
    public string Account { get; private set; } - account
    public string Profile { get; private set; } - profile
    public string Symbol { get; private set; } - symbol
    public string Resolution { get; private set; } - resolution, this should not be
    empty when Automated Trading is being used.
    public string Name { get; private set; } - order name, when Automated
    Trading is being used this contains the order name generated by the strategy. If Semi-Automated
    strategies for manual trading are being used this contains the order names generated by them.
    public string StrategyName { get; private set; } - strategy name, this should
    not be empty when Automated Trading is being used.
    public string Workspace { get; private set; } - workspace name, this should
    not be empty when Automated Trading is being used.
    public ETM_OrderAction Action { get; private set; } - action, this is buy or
    sell

    public ETM_OrderState State { get; private set; } - order state, this is filled,
    cancelled, sent, etc.
    public ETM_OrderCategory Category { get; private set; } - category this
    market, stop, limit, etc.
    public ETM_OrderTIF TIF { get; private set; } - Time In Force of the order
    public DateTime? TIFDate { get; private set; } Date for GTD Time In Force
    public int Contracts { get; private set; } - number of contracts
    public int FilledContracts { get; private set; } - number of filled contracts
    public double? StopPrice { get; private set; } - stop price (!=null for stop &
    stoplimit orders)
    public double? LimitPrice { get; private set; } - limit price (!=null for limit &
    stoplimit orders)
    public double? ExecPrice { get; private set; } - average order execution price
    public string BrokerID { get; private set; } - order ID returned by the broker
    public int OrderID { get; private set; } - internal order ID that is generated by
    MultiCharts and is used for internal calculation for Automated Trading.
    public string OCOGroupID { get; private set; } - OCO group ID. If the order is
    not a member of OCO group this field will be empty.
    public string ID { get; } - unique order identifier, which is - Profile +
BrokerID.
}
```

Order statuses:

```
public enum ETM OrderState
{
    eTM OS Cancelled = 2, - cancelled, Final status.
    eTM OS Filled = 3, - filled, Final status.
    eTM OS Ignored = 9, - user cancelled order transmission within Automated Trading in
the configuration dialogue, Final status
    eTM OS PartiallyFilled = 7, - partially filled, will continue to be filled and will turn to
eTM OS Filled state.
    eTM OS PreCancelled = 6, - cancellation command has been sent.
    eTM OS PreChanged = 8, - modification command has been sent.
    eTM OS PreSubmitted = 0, - sent to the Exchange.
    eTM OS Rejected = 4, - rejected by broker, Final Status.
    eTM OS Saved = -10, - artificial status, that is set by MultiCharts .NET for active orders
when the broker profile is disconnected or MultiCharts .NET is close, Final status.
    eTM OS Sent = 5, - sent to the broker.
    eTM OS Submitted = 1 - submitted at the Exchange.
}
Final Status means that the order turned in its final state and will not be modified.
```

IPositions collection contains the methods:

```
public struct Position
{
    public string Account { get; private set; } - account
    public string Profile { get; private set; } - profile
    public string Symbol { get; private set; } - symbol
    public int Value { get; private set; } - market position(0== if closed, 0< if long
and 0> if short).
    public double AvgPrice { get; private set; } - average entry price
    public double OpenPL { get; private set; } - current unrealized profit
    public string ID { get; } - identifier, which is Profile + Account + Symbol
}
```

4.7.5.3 Trading Functionality. *TradingProfiles*

The **ITradeManager** interface contains the **TradingProfiles** property that returns an array of profiles represented as the **ITradingProfile** interface:

```
public interface ITradingProfile
{
    // Events
    event TConnection Connection; - By subscribing to this event it is possible get the
notifications about disconnects and reconnects.

    event TRealtime RT; - Real-time

    // Order operation methods:
    void CancelOrder(int order_id); - cancellation of the order by its ID.
```

`void ModifyOrder(int order_id, OrderParams -order, MTPA ModifiableOrderFields fields_for_modify);` - modifications of the order fields by its ID

`int[] PlaceOCOOrders(MTPA OCOGroupType oco_type, params OrderParams [] orders);` - submission of several orders within OCO group. This returns a list of successfully submitted order identifiers (IDs). The OCO group can be of two types:

- a) One Cancel Others means that the cancellation or execution of one order cancels all other orders.
- b) Reduce Size means that when Order#1 is filled for X contracts the amount of the contracts of the other orders in the group is reduced by X number of contracts.

`int PlaceOrder(OrderParams order);` - submission of one order. If submission is successful the identifier (ID) of the order will be returned. Otherwise, an exception like "System.Runtime.InteropServices.COMException." will be generated.

`// Properties`
`string[] Accounts { get; }` – returns the list of accounts for the current profile.
`ETM ConnectionChanged ConnectionState { get; }` – the current connection state.
`string CurrentAccount { get; set; }` – a property that sets the current account that is used for trading.

`MCSymbolInfo CurrentSymbol { get; set; }` – a property that sets the current symbol that is used for trading and that is updated in real-time on RT event.

`MTPA ModifiableOrderFields ModifiableFields { get; }` – Returns attribute indicating what fields are available for modification on a brokers side.

`string Name { get; }` – Profile name. This is the name specified by a user when creating a broker profile. The same broker can have several profiles with different settings for each one.

`string PluginName { get; }` Trading plug-in name that correlates with the broker (1-to-1).
`}`

The **CurrentSymbol** property has the following type:

```
struct MCSymbolInfo
{
    public MTPA MCSymbolInfo2 symbol; – the symbol itself.

    public string data feed; - data source name (Interactive Brokers, eSignal, CQG etc).
}
```

MTPA MCSymbolInfo2 structure is a COM-structure with the following fields:

```
public struct MTPA MCSymbolInfo2
{
    public string SymbolName; – name of the instrument, for example ESU3, EUR/USD
    public string SymbolRoot; – symbol root (for futures), for example ES, CL
}
```

`public string SymbolDescription`; - Instrument description in words, for example, for ESU3 "S&P 500 E-mini Futures Sep13". Some data sources save additional info in this field.

`public string SymbolExchange`; - exchange name, for example GLOBEX, CME

`public MTPA MCSymbolCategories SymbolCategory`; - symbol category, for example, futures, Forex or stock.

`public MTPA MCSymbolCurrency SymbolCurrency`; - symbol currency, for example USD, EUR, JPY.

`public DateTime SymbolExpiry`; - expiration date(for futures), for example ESU3 09/19/2013.

`public int MinMove`; - minimum quantity of points, when the price moves by one minimum price increment, for example for ES MinMove = 25

`public double PriceScale`; - price scale, for example for ES = 1/100

`public double BigPointValue`; - Big Point Value. Profit/loss money equivalent in currency of the instrument , when the price moves by one minimum increment, for example, for ES = 50

`public short PutOrCall`; - for options

`public double StrikePrice`; - for options

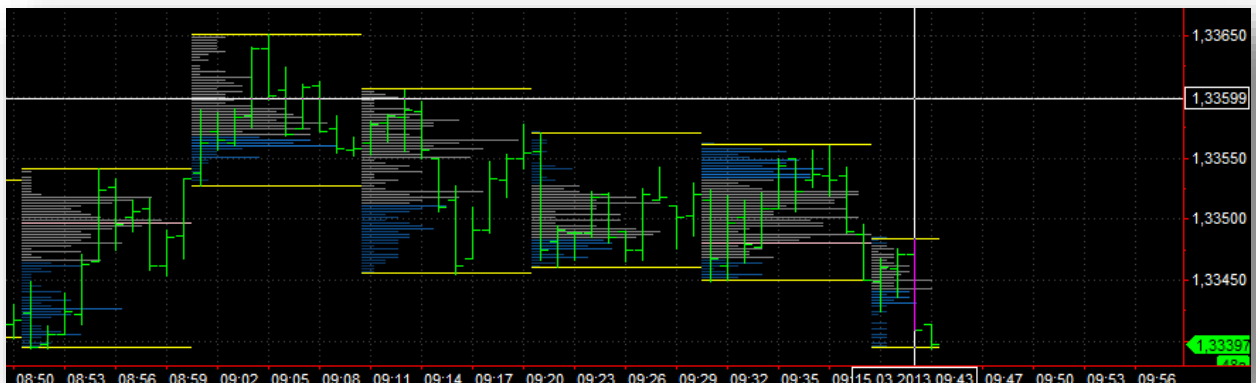
`public MTPA SymbolFields UnsupportedFields`; - instrument fields, which are not provided by the source
}

This structure can be received through the properties of Bars.Info.ASymbolInfo2 for the current symbol.

4.7.6 Volume Profile

All indicators and signals have access to Volume Profile which can be plotted on the chart for the instrument. Profile collection for the main symbol is available via the **VolumeProfile** property or any symbol through a data stream via the **VolumeProfileByDataStream(int DataNumber)** method.

Example: The Volume Profile is plotted for the main symbol. Here Volume Profile consists of profiles plotted for every 10 bars of the symbol:



Profile collection is presented by the following interface:

```
public interface IProfilesCollection : IDisposable
{
    ICollection<IProfile> Items { get; } //Profiles list

    bool Visible { get; } //true - profiles are plotted, false - no
    profiles and collection is empty

    event TProfilesChanged EChanged; //event for receiving the
    notifications about modification of profile collection
    IProfile ItemForBar(int bar); // returns the profile on bar number.
    Bar number should be absolute, starting with 0 (0-based). Acces to the
    absolute bar number can be gained via Bars.FullSymbolData.Current-1
}
```

Profile modification Event handler:

```
namespace PowerLanguage.VolumeProfile
{
    public delegate void TProfilesChanged(bool _full);
}

Example: the study will be recalculated if volume profile has changed:
private void VolumeProfileOnEChanged(bool full)
{
    if (full)
        this.ExecControl.Recalculate();
}

protected override void StartCalc() {
    VolumeProfile.EChanged += VolumeProfileOnEChanged;
}
```

Interface with the Profile information:

```
public interface IProfile
{
    decimal AboveVAValue { get; } //Cumulative volume on all the levels
    higher then the highest limit of Value Area(VA)
    Pair<int> BarsInterval { get; } //The numbers of start and end bars
    of the profile (first, second]
    decimal BelowVAValue { get; } //Volumes sum on the levels lower than
    the lowest limit of the VA
    Price Close { get; } //Profile closing price
    bool Empty { get; } //true - then Profile is empty
    ILevel MaxDelta { get; } //Level with the maximal difference between
    Buy - Sell (ASK traded - BID traded)
    ILevel MinDelta { get; } // Level with the minimal difference
    between Buy - Sell (ASK traded - BID traded)
    Price Open { get; } //Profile opening price
    ILevel POC { get; } // POC level (Point Of Control)
    decimal TotalValue { get; } //Cumulative volume on the profile
    ICollection<ILevel> VA { get; } //Collection of levels between the
    highest and the lowest limits of VA
    ICollection<ILevel> Values { get; } //Profile levels collection
}
```

```

        decimal VAValue { get; } //Cumulative volume of the levels between
the highest and the lowest limits of VA

        Price HighVAForBar(int bar); //Price of the highest VA level for the
bar
        Price LowVAForBar(int bar); //Price of the lowest VA level for the
bar
        Price POCForBar(int bar); //POC level price for the bar
    }

```

The Profile level description interface:

```

public interface ILevel : IEquatable<ILevel>
{
    decimal AskTradedValue { get; } // Cumulative volume of all ASK
traded trades for the level
    decimal BidTradedValue { get; } // Cumulative volume of all BID
traded trades for the level
    Price Price { get; } //Level price
    decimal TotalValue { get; } //Cumulative volume of the level
}

```

An example of indicator that plots the minimal and the maximal VA levels:

```

[SameAsSymbol(true)]
public class VA_Min_Max : IndicatorObject {
    public VA_Min_Max(object _ctx):base(_ctx){}

    private IPlotObject max_VA;
    private IPlotObject min_VA;

    protected override void Create() {
        max_VA = AddPlot(new PlotAttributes("MaxVA", EPlotShapes.Line,
Color.Red, Color.Black, 2, 0, true));
        min_VA = AddPlot(new PlotAttributes("MinVA", EPlotShapes.Line,
Color.Blue, Color.Black, 2, 0, true));
    }
    protected override void StartCalc() {
        //subscribing for profile changes
        VolumeProfile.EChanged += VolumeProfileOnEChanged;
    }

    protected override void CalcBar()
    {
        int bn = Bars.FullSymbolData.Current-1;
        var vp = VolumeProfile.ItemForBar(bn);
        if (vp != null)
        {
            max_VA.Set((double)vp.HighVAForBar(bn).Dbl );
            min_VA.Set((double)vp.LowVAForBar(bn).Dbl );
        }
    }
    private void VolumeProfileOnEChanged(bool full)
    {
        //Recalculate if the profile has been completely changed.
        if (full)

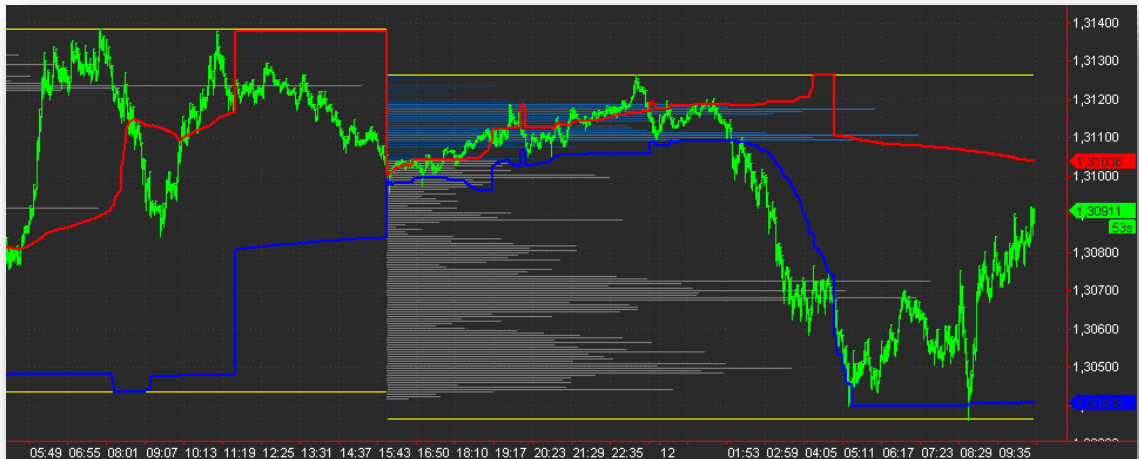
```

```

        this.ExecControl.Recalculate();
    }
}

```

After compilation, enable Volume Profile for the instrument on the chart. Then, insert this indicator into the chart. Now it's possible to see how the maximal and minimal VA levels changed within Volume Profile.



5 Compilation. Modules and assembly handling.

PL.NET supports studies written on 2 programming languages, C# and VB.NET. It is well known that the .NET Framework contains the compilers for both of these languages. But it is also known that it is impossible to compile source codes written on two different languages into a single assembly. That's why all C# studies should be compiled into a single .netmodule, and all VB.NET studies should be compiled into a separate .netmodule, only then can these two netmodules be linked into a single assembly.

Below are some advantages that can be gained with the implementation described above:

- 1) It is possible to organize data transfer between several separate studies within a single MultiCharts .NET process by using static fields.
- 2) It is possible to create source code storage with common types. To do that:
 - a. Create a new function with any name.
 - b. Delete the WHOLE code (this function will always be colored in red (not compiled), but it will be required for code storage only)
 - c. Fill this function with any info that can be necessary common classes including basic classes of indicators, signals, methods, etc.
 - d. This common code can be used further in any other signals and indicators

When the compilation of a study is initialized, either after pressing the “Compile” button or because of changes to a study, it will check to see if this study is already applied to a chart or a scanner window. If it is already being used, after compilation has been completed successfully, MultiCharts .NET will be informed that these exemplars should be re-created from a new assembly, re-initialize their inputs with the values set by user, and recalculate.

IMPORTANT NOTE: After such a partial re-compilation (Compile->Compile (F7)), the studies with modified code will be already from another assembly (with a different name), and unmodified studies will be still from the previous assembly. It should be noted if the same static variables with several classes are used (for example for data transfer). The workaround for this is to re-compile the studies completely (Compile->Recompile All (Ctrl+F7)). In this case, ALL the studies will be re-compiled and re-created. This compilation mode is useful for **development** but can lead to **undesirable results**, such as **disabling of automated trading** performed by an unmodified signal.

6 Integration with Microsoft Visual Studio 2010/2012/Express

In MultiCharts .NET it is possible to develop indicators and signals not only in a pre-built editor, PLEditor.NET.exe, but in any other editor, for example, in a notepad, or in a more suitable editor such as MS VS 2010 and its freeware version, [MS VS Express](#).

The Integration process between MultiCharts .NET and MS VS is very simple. The script for each study is stored in a separate file (*.cs or *.vb). Below are the default paths of the folders where these scripts are stored:

- 1) For **MultiCharts64 .NET** - %allusersprofile%\Application Data\TS Support\MultiCharts64 .NET\StudyServer\Techniques\CS\
- 2) For **MultiCharts .NET** - %allusersprofile%\Application Data\TS Support\MultiCharts .NET\StudyServer\Techniques\CS\

The Source file name is important. It is formed from the name of the study, its type and extension (cs or vb). **Example:** “_TPO_.Indicator.CS” is the indicator with “_TPO_” name written in C#.

If the file is changed and these changes are saved, MultiCharts .NET will track them, accept the changes and compile the study. If this study was applied to the chart, then after successful compilation, the study will be re-created on the chart and the result of the changes will be seen immediately. Such an implementation allows the editing of source codes from anywhere.

Let’s discuss MS VS. You may want to ask why do you need MS VS if any editor can be used to work with the script. The answer is MS VS provides additional features:

- 1) Debugging. In MultiCharts .NET there is no pre-built debugger for managed applications, that is why MS VS is required if debugging is essential for a developer.
- 2) IntelliSense. PLEditor .NET supports highlighting of syntax, lists of methods and classes. As opposed to a pre-built PLEditor .NET, MS VS supports builds with target framework 4.0 and higher.

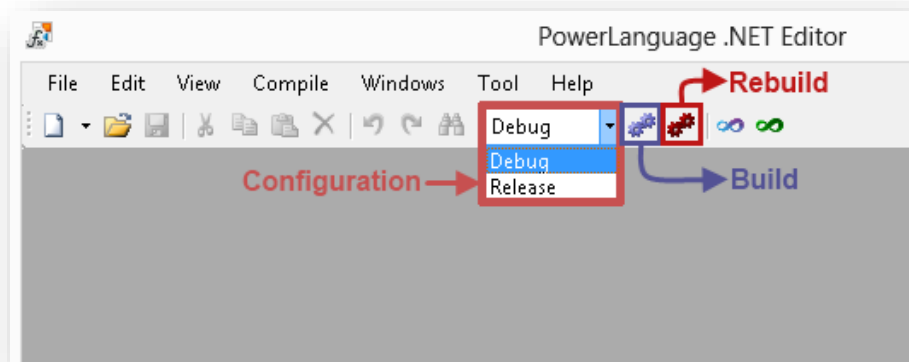
- 3) Version Control System. MS VS supports numerous version control systems through special plug-ins. If a folder with studies is put under source control, then an already-made professional development environment can be used.

There are some cases when it is necessary to use a pre-built PLEditor .NET:

- 1) Adding references to external assembly. For any study it is possible to add a reference to any other assembly and use its functionality. If the reference has been added in MS VS but has not been added in PLEditor .NET, then this study will be compiled without that reference. In this case the compilation will fail.

NOTE: Any reference to the other assemblies should be added ONLY via PLEditor .NET.


- 2) Compilation mode. PLEditor .NET has 2 configurations (Debug, Release).



Debug configuration is required for development and debugging of the studies. Release configuration is required for the final assembly.


When the compilation is started in editor, MultiCharts .NET remembers what compilation mode was used. With future background compilations, when studies scripts are changed and saved from MS VS or any other editor, the most recent mode parameters will be used.

6.1 Debugging with Microsoft Visual Studio 2010/2012

To debug indicators and signals in MultiCharts .NET open the solution with the scripts of these studies from PLEditor .NET by clicking the  button on the toolbar. When MS VS 2010 is open, select Debug > Attach to Process from the main menu. In the opened window select:

1. Attach to: Managed (v4.0) code
2. In Available Process list select MultiCharts.exe or MultiCharts64.exe process depending on the version used.
3. Click the Attach button.
4. Proceed with debugging.

6.2 Debugging with Microsoft Visual Studio Express

C# Express and VB.NET Express versions are distributed separately. The Information below can be applied only to “**Microsoft Visual Studio C# 2010 Express**”. This free version of MS VS has one drawback: it does not have the **Attach To Process** menu. However, we created a workaround: click the  button on the tool bar of PLEditor .NET and MS VS C# Express will be opened. It will contain one C# project with the scripts of all studies. To start the debugging process run MultiCharts .NET from MS VS. To do that:

1. Click **Debug** in the main menu.
2. Select **Start Debugging**.

After that, MultiCharts .NET should start.