

MQL5

Programming Language



**Advanced use of the
trading platform MetaTrader 5
Creating trading robots
and indicators**

Second Edition, Revised & Updated

Timur Mashnin

**MQL5 programming language
Advanced use of the trading
platform MetaTrader 5
Creating trading robots and
indicators**

Second Edition, Revised & Updated

Timur Mashnin

Source Code

The source code for this book can be downloaded at
<https://github.com/novts/MetaTrader-5-Creating-Trading-Robots-and-Indicators-with-MQL5>.

Introduction

I hope you all have already read the MQL5 Reference on the site <https://www.mql5.com/en/docs>.

<https://www.mql5.com/en/docs>



The screenshot shows the MQL5 Reference website. On the left, there is a navigation menu titled "MQL5 Reference" with a list of topics including Language Basics, Constants, Enumerations and Structures, MQL5 programs, Predefined Variables, Common Functions, Array Functions, Conversion Functions, Math Functions, String Functions, Date and Time, Account Information, Checkup, Event Handling, Market Info, Timeseries and Indicators Access, Custom Symbols, Chart Operations, Trade Functions, Trade Signals, Network Functions, Global Variables of the Terminal, File Functions, Custom Indicators, Object Functions, Technical Indicators, and Working with Optimization Results. The main content area features the "MQL5" logo in large letters. Below the logo, there is a brief introduction to the MQL5 language, mentioning its use for developing trading robots and utility applications, and its similarity to C++. It also mentions the MetaEditor IDE, MQL5 Storage, and the MQL5 community website. Further down, it discusses trading functions, event handlers, and technical indicators.

Here we will not retell this document but focus on its practical use.

From time to time we will allow ourselves only its citing.

As stated in the introduction to the Reference:

MQL5 programs are divided into four specialized types based on the trading automation tasks that they implement.

And then comes a list of Expert Advisor, Custom Indicator, Script, Library and Include File.

Scripts are used to perform one-time actions, processing only the event of its start, and therefore will not be interesting to us here.

Also, we will not be interested in libraries, since the use of include files is more preferable to reduce overhead.

Therefore, we will focus on creating expert advisors and indicators using included files. This is our goal in using the MQL5 programming language, which the syntax is, of course, interesting, but it will only help us.

In fact, MQL5 programming refers to the field of event-oriented programming, since all the code of an MQL5 application is based on overriding of callback functions — handlers for client terminal and user events.

And already in the code of callback functions, you can use either procedural programming or object-oriented programming.

Here we look at both of these approaches.

Getting Started

To get started, we will select some agent to connect to its server and receive real market quotes for developing and testing our MQL5 applications.

货币	国家/地区	买入价	卖出价
港元	Hong Kong	5.08	5.93
Malaysian Ringgit	Malaysia	4.43	4.74
EUR	Euro	7.48	8.75
Australian Dollar	Australia	37.25	39.44
Pound sterling	England	24.13	26.42
대한민국 원 (: 1000)	Korea	52.84	55.76
New Zealand Dollar	New Zealand	25.50	42.60
		22.76	24.41
		36.65	

As an agent, we mean a professional market participant who has the right to perform operations in the market on behalf of a client and at his expense or on agent's behalf and at the expense of a client on the basis of compensable contracts with a client.

Now, what is the market?

There are different types of markets.

There is a foreign exchange market, there is a stock market or securities market, there is a commodity market, and there is a futures and options market.

We will focus on the currency market or forex market.

What is the forex market?

FOREX is an abbreviation for two words Foreign Exchange, which means Currency Exchange.

Unlike other markets where trading takes place on the stock exchanges, the forex market is an over-the-counter market for interbank currency exchange

without any centralized platform.

Participants in the forex market are central banks, commercial banks, investment banks, brokers and dealers, pension funds, insurance companies, transnational corporations, etc.

Really, the majority of transactions involving the exchange of one currency for another takes place in the over-the-counter market between large international banks using the Interbank Information Network.

And very large sums of money are traded. The minimum lot is the sum of 1 million dollars or euros, the standard lot is the sum of 5 or 10 million dollars.

Firstly, such currency trading provides export-import operations of bank customers, and secondly, it provides interests of trading and investment departments of international banks.

And banks make transactions both on the interbank over-the-counter market and on the currency exchanges.

Where do quotes in the Forex market come from?

If you take, for example, a stock market, then there is a special organization - the exchange, where certain securities are traded (only there and nowhere else), and this exchange itself acts as a single center for the distribution of quotations to other participants, including dealing centers.

In the case of Forex, such a center does not exist, the market does not have a single place of trade and Forex unites all participants through modern means of data transfer.

Since the bulk of trading operations is carried out through banks, the Forex market is called as the international interbank market.

All the largest participants in this market, international banks, make quotations and act as a kind of "market drivers", making deals either with other banks or with clients - investment funds, companies, individuals.

All other participants in the Forex market request a quote from the largest participants and conduct its operations based on these quotes.

Quotations for currency pairs are set by international banks, as a rule, in the electronic mode.

And quotations are formed both on the basis of requests from other participants, and in streaming mode (indicative), when a bank sets a "reference" rate, in which it is ready to make a deal, but will not be obliged to

do so.

A final price of the transaction depends on the amount of the transaction, the status of the participant, a current position in the market and other factors.

Indicative and real quotes come into the global information systems (Reuters, Bloomberg, Dow Jones, etc.), from where they are received by other users, including dealing centers.

It is the quotes received from the serving dealing center that a trader sees in his trading terminal, which he uses in the trading process.

Thus, if we compare Forex with an exchange market, then there is no price, which is the same for all participants without exception.

Often, transactions are performed at different prices, and the price will be more profitable for minor participants who have established contacts with the main participants - banks, as well as participants trading large amounts of currency.

At the same time, due to the high liquidity of the market, quotes in most cases differ only by 1-2 points, which makes spatial arbitration almost impossible when a participant buys currency from one seller at a price, knowing at the same time that it can sell currency to another buyer on more favorable terms.

Now, what is a dealing center?

A dealing center is a non-banking organization that provides customers with small amounts of trading capital to enter into speculative transactions on the terms of margin trading.

And naturally, before transferring quotes to its customers, a dealing center imposes its own filter on the quotes, including, among other things, a spread that will make up its earnings.

Now, in that way, a dealing center provides an opportunity for clients with small amounts of trading capital to enter into speculative transactions on the basis of margin trading.

As the dealing centers explain themselves, they send not all client orders to the real over-the-counter market, but only an aggregate order that exceeds a certain amount. The remaining orders are dealt with by dealing with opposite orders received from other clients.

In fact, as a rule, there are not dealing centers that put deals of its customers to the open market, because they know that the conditions of the game are such that any client will lose sooner or later. Therefore, there is no need to

bring transactions to the market.

Thus, a client or trader trades not against the market, but against the dealing center.

In the beginning, we said that we need some kind of intermediary to connect to its server and get real market quotes for developing and testing our MQL5 applications.

Since we do not have millions of dollars to trade directly in Forex, and we cannot afford to establish our own interbank information and trading terminal, we will choose a dealing center as the intermediary.

The screenshot shows the FxPro website homepage. At the top, there is a red header bar with the FxPro logo and the tagline "Trade Like a Pro". Below the header, there are navigation links for Markets, Tools, Pricing, Company, and a search bar. On the right side of the header, there are language selection (EN), account sign-in ("Hello, Sign in Your Account"), and a "Register" button. The main content area features a large, colorful nebula-like background image. Overlaid on this image is the text "Your trusted trading partner" and "In every order. In every execution. In every trade.". A call-to-action box on the left side contains icons for mobile devices and computers, with the text "Get started for free" and a "Create Account" button. Below this box, there is a link "Get FREE platforms today." and a "Invest later" button. At the bottom of the page, there is a cookie policy notice, a "Cookie Disclosure" link, and a "CLOSE" button. A footer section displays current market prices for GOLD (1342.85), EURUSD (1.13402), GBPUSD (1.30552), USOIL (56.506), AAPL (171.21), and S&P500 (2776.50). It also includes a warning about "Invest Responsibly: Trading CFDs involves significant risks." and a "Get Started" button.

Let's choose, for example, the FxPro dealing center.

I am not a fan of this company, it is just for our coding.

For real trading, it is better to choose, perhaps, a bank.

Register and create a demo account for the MetaTrader 5 platform.

Usually, a dealing center offers two types of accounts: Instant Execution and Market Execution.

What about Instant Execution.

In this mode, a market order is executed at a proposed price.

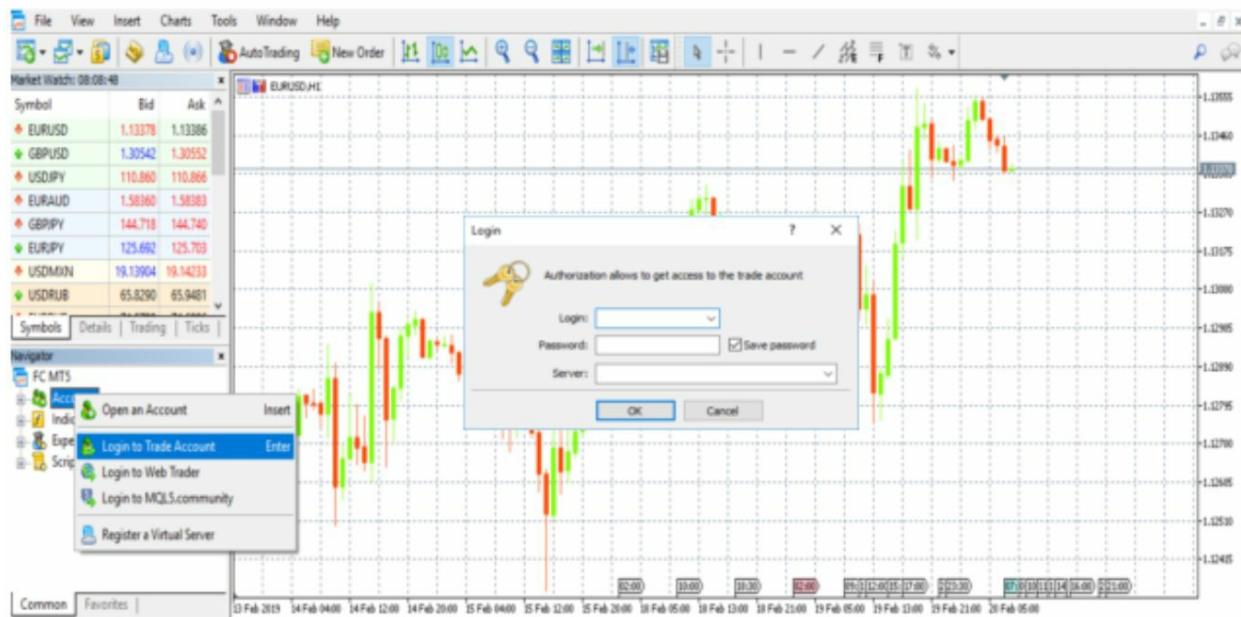
When sending a request for execution, the platform automatically inserts current prices into the order.

And what about Market Execution.

In this mode to execute a market order, the decision about a price of execution is taken by the dealing center without additional coordination with a trader.

We will open an account with Instant Execution.

Next, download and install the MetaTrader 5 platform.



And to connect to the FxPro server, you should use a login and password of the demo account.

Next, right-click on the chart and go to properties, adjust the appearance of the chart as you like.



The MetaTrader 5 multi-market platform allows trading on Forex, stock exchanges and futures.

Using MetaTrader 5, you can also do technical analysis of quotes, work with trading robots and copy trades of other traders.

<https://www.metatrader5.com/en/terminal/help>

MetaTrader 5 Help

[Getting Started](#)
[Trading Operations](#)
[Price Charts, Technical and Fundamental Analysis](#)
[Algorithmic Trading](#)
[Trading Robots](#)
[Trading Signals and Copy Trading](#)
[Market App Store](#)

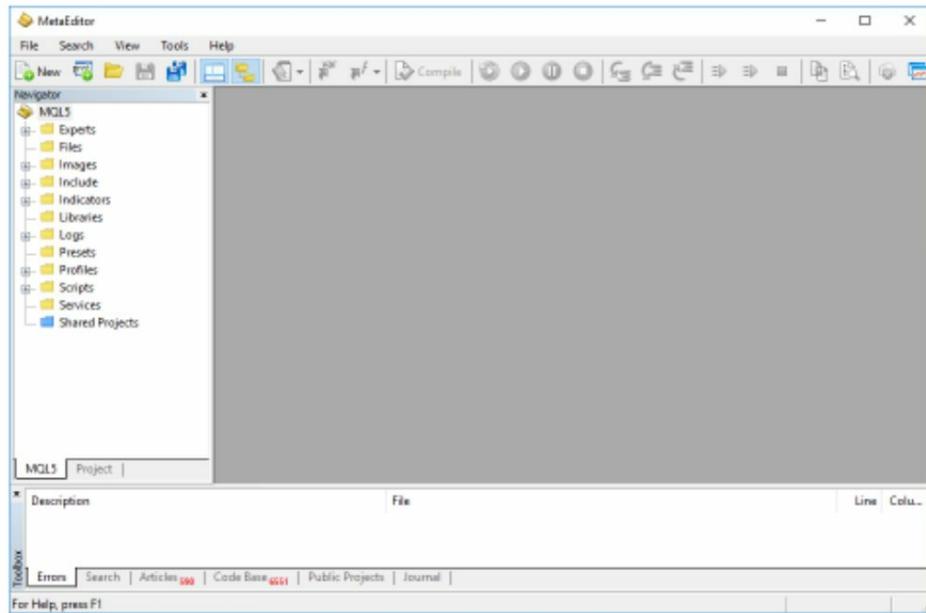
Trading Platform — User Manual

The Trading Platform is the trader's working tool, providing all the necessary features for a successful online trading. It includes [trading](#), [technical analysis of prices](#) and [fundamental analysis](#), [automated trading](#) and [trading from mobile devices](#). In addition to Forex symbols, options, futures and stocks can be traded from the platform.

[All Types of Orders, Price Charts, Technical and Fundamental Analysis, Algorithmic and Mobile Trading](#)

You can read more about the MetaTrader 5 platform and its interface in the corresponding help.

We will not retell this manual since it would be too brazen to take money for lectures in which the publicly available document is retold.

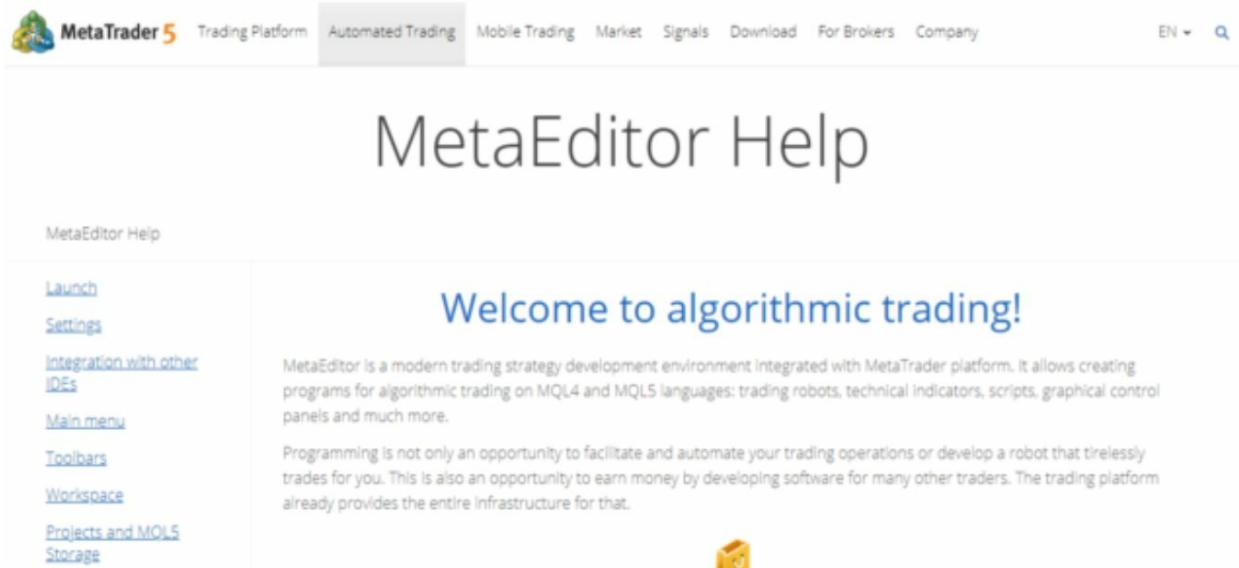


In addition to the MetaTrader 5 terminal, we are interested in the MQL5 editor, which can be opened either via a shortcut or in the MetaTrader 5 terminal menu Tools.

The MetaEditor is a modern trading strategy development environment integrated with the MetaTrader platform.

With MetaEditor, you can create trading robots, technical indicators, scripts, graphical control panels, and much more.

<https://www.metatrader5.com/en/metaeditor/help>



The screenshot shows the 'MetaEditor Help' page. At the top, there's a navigation bar with links for 'Trading Platform', 'Automated Trading', 'Mobile Trading', 'Market', 'Signals', 'Download', 'For Brokers', and 'Company'. The 'EN' language dropdown and a search icon are also at the top right. Below the header, the main title 'MetaEditor Help' is centered. On the left, a sidebar contains links: 'Launch', 'Settings', 'Integration with other IDEs', 'Main menu', 'Toolbars', 'Workspace', 'Projects and MQL5 Storage'. The main content area features a large blue heading 'Welcome to algorithmic trading!'. Below it, a paragraph explains that MetaEditor is a modern trading strategy development environment integrated with MetaTrader platform, allowing for creating programs for algorithmic trading on MQL4 and MQL5 languages. It highlights the ability to create trading robots, technical indicators, scripts, graphical control panels, and more. Another paragraph states that programming is an opportunity to facilitate and automate trading operations or develop a robot that trades for you, and that the trading platform provides the entire infrastructure for that. A small orange icon of a folder with a document is positioned to the right of the text.

MetaEditor Help

[Launch](#)
[Settings](#)
[Integration with other IDEs](#)
[Main menu](#)
[Toolbars](#)
[Workspace](#)
[Projects and MQL5 Storage](#)

Welcome to algorithmic trading!

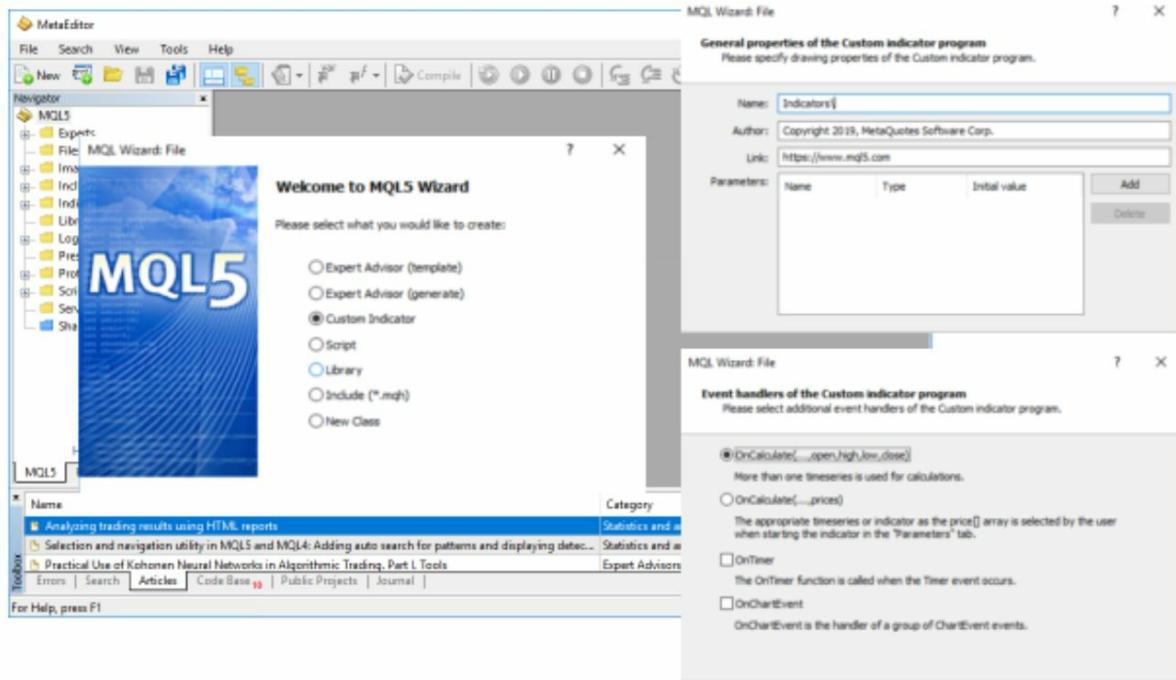
MetaEditor is a modern trading strategy development environment integrated with MetaTrader platform. It allows creating programs for algorithmic trading on MQL4 and MQL5 languages: trading robots, technical indicators, scripts, graphical control panels and much more.

Programming is not only an opportunity to facilitate and automate your trading operations or develop a robot that tirelessly trades for you. This is also an opportunity to earn money by developing software for many other traders. The trading platform already provides the entire infrastructure for that.

For the MetaEditor, there is also a detailed help, which we will not retell also. We'd better deal with practical coding right away.

Indicator General Structure

To create a base code of a custom indicator, you can use the MetaEditor editor.



You can press the menu button “New” and select the Custom Indicator in the wizard window.

Further, click the button “Next”, enter a name of the created indicator, click “Next” and check functions that the wizard should generate, and in the next window click the “Finish”.

```

1 //+-----+
2 //| InDemo.mq5 |
3 //| Copyright 2019, MetaQuotes Software Corp. |
4 //| https://www.mql5.com |
5 //+-----+
6 #property copyright "Copyright 2019, MetaQuotes Software Corp."
7 #property link "https://www.mql5.com"
8 #property version "1.00"
9 #property indicator_chart_window
10 //+-----+
11 //| Custom indicator initialization function |
12 //+-----+
13 int OnInit()
14 {
15 //--- indicator buffers mapping
16
17 //---
18     return(INIT_SUCCEEDED);
19 }
20 //+-----+
21 //| Custom indicator iteration function |
22 //+-----+
23 int OnCalculate(const int rates_total,
24                 const int prev_calculated,
25                 const datetime &time[],
26                 const double &open[],
27                 const double &high[],
28                 const double &low[],
29                 const double &close[],
30                 const long &tick_volume[],
31                 const long &volume[],
32                 const int &spread[])
33 {
34 //--- return value of prev_calculated for next call
35     return(rates_total);
36 }
37 //+-----+
38 //| ChartEvent function |
39 //+-----+
40 void OnChartEvent(const int id,
41                     const long &param,
42                     const double &param,
43                     const string &param)
44 {
45 //---
46 }
47 //+-----+
48 void OnDeinit(const int reason)
49 {
50 //+-----+
51 Print(__FUNCTION__,"Deinitialization reason code= ",reason);
52
53 }
54
55

```

As a result, it will create an indicator's base code.

The indicator code begins with a block of a declaration of indicator properties and various objects used by an indicator, such as arrays of indicator buffers, input parameters, global variables, handles of used technical indicators, and constants.

This code block is executed by the MetaTrader 5 Trading Platform immediately when an indicator is attached to a symbol chart.

After a declaration block of the indicator properties, its parameters and variables, you can see callback functions, which the terminal calls receiving such events as indicator initialization after an indicator is loaded before an indicator is deinitialized, when price data are changed, and when a symbol chart is changed by a user.

To handle the above events, you need to define such functions as the OnInit, OnDeinit, OnCalculate, and OnChartEvent.

In the OnInit function of an indicator, as a rule, arrays declared in the initial block are associated with indicator buffers.

Also, the OnInit function defines indicator's output values, specifies indicator colors, specifies the number of digits after the decimal point to visualize indicator values, indicator's labels, and other indicator display parameters.

In addition, in the OnInit function of an indicator, it can get handles of used

technical indicators and calculate other variables to be used.

In the OnDeinit function of an indicator, as a rule, it deletes graphic objects of the indicator and removes objects from a symbol chart, and, also it deletes handles of the technical indicators to be used.

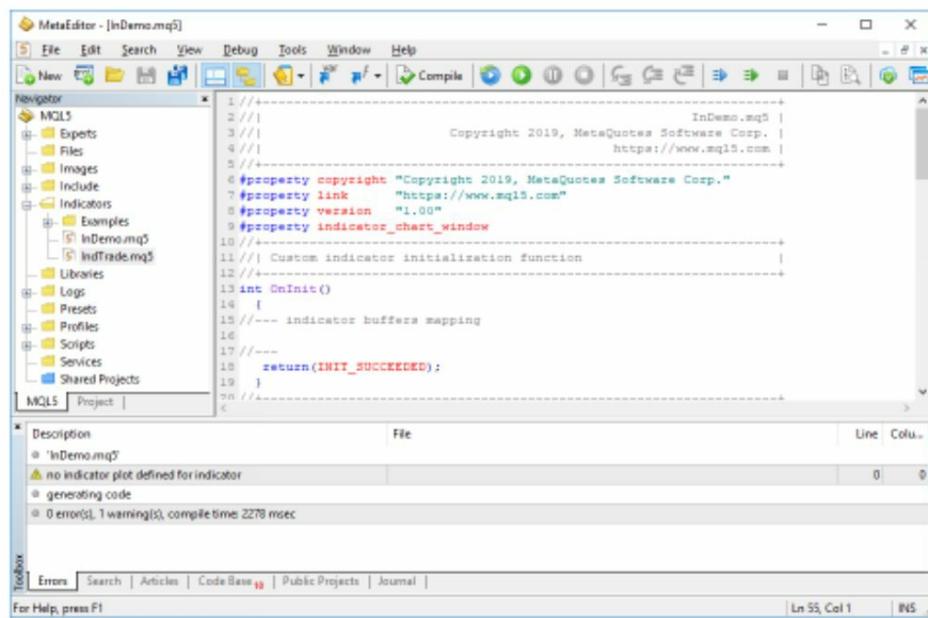
Actually, in the OnCalculate function, it calculates indicator values, filling arrays declared in the initial block with calculated values.

Remember, these arrays were associated with indicator buffers in the OnInit function of an indicator, and data from these arrays are taken by the terminal to draw the indicator.

In addition, the OnCalculate function can change colors of an indicator and other parameters of its displaying.

The OnChartEvent function can handle events generated by other indicators on a chart, as well as by a user when deleting a graphical object of an indicator and other events that occur when a user works with the chart.

This is where the indicator code ends, although custom functions can also be defined there to call from the OnInit, OnDeinit, OnCalculate and OnChartEvent functions.

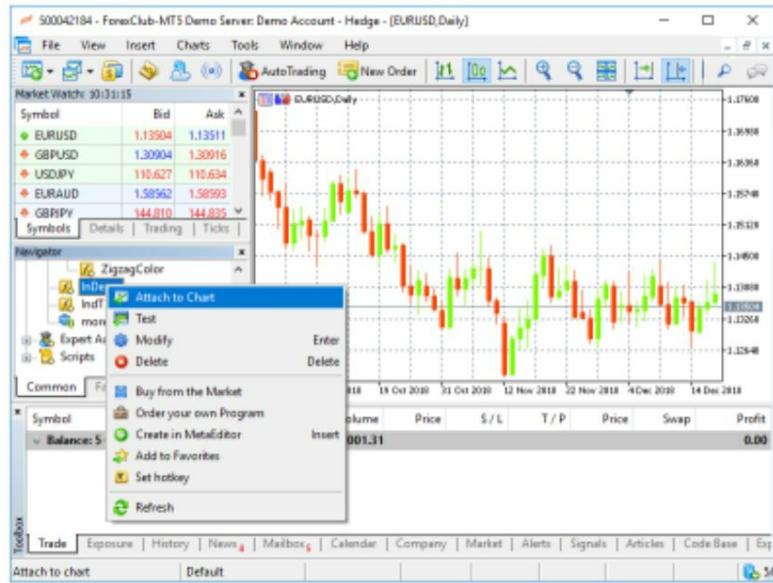


The screenshot shows the MetaEditor interface with the file 'InDemo.mq5' open. The code editor displays the following MQL5 script:

```
1 //+------------------------------------------------------------------+
2 //| Copyright 2019, MetaQuotes Software Corp. |
3 //| https://www.mql5.com |
4 //+----+
5 //+-----[REDACTED]-----+
6 #property copyright "Copyright 2019, MetaQuotes Software Corp."
7 #property link "https://www.mql5.com"
8 #property version "1.00"
9 #property indicator_chart_window
10 //+-----[REDACTED]-----+
11 //| Custom indicator initialization function |
12 //+-----[REDACTED]-----+
13 int OnInit()
14 {
15 //--- indicator buffers mapping
16
17 //---
18 return(INIT_SUCCEEDED);
19 }
```

The Navigator panel on the left shows the project structure with 'InDemo.mq5' selected. The bottom status bar indicates 'Ln 55, Col 1' and 'INS'.

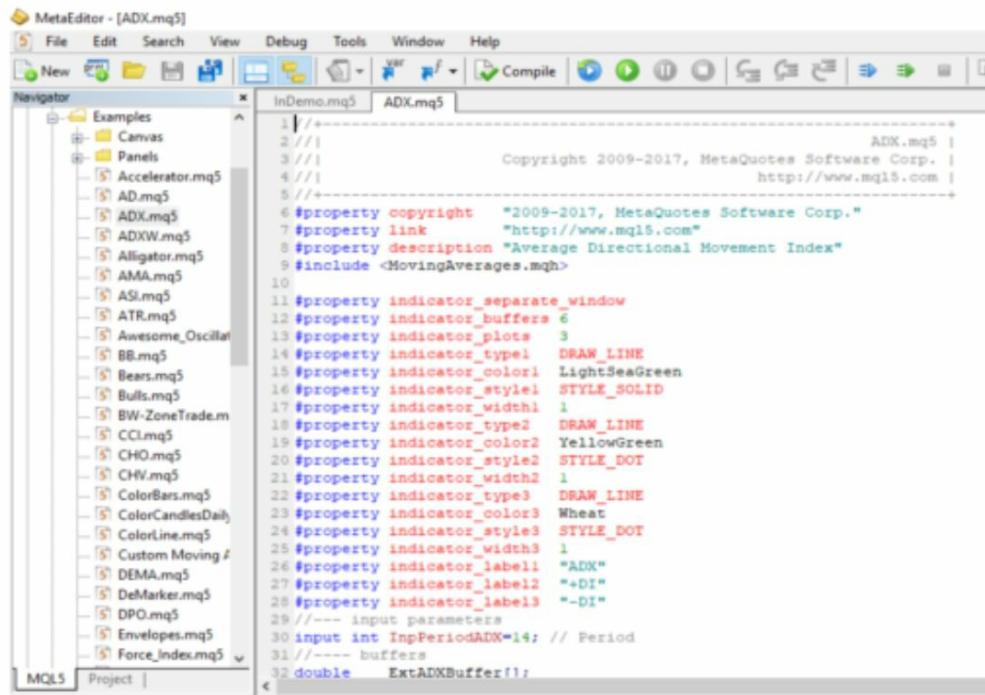
To compile our indicator, click the Compile button of the MetaEditor, and a result of the compilation will be displayed in the bottom window.



After compilation, our indicator will automatically appear in the trading terminal and we will be able to attach it to a chart of the financial instrument.

Indicator Properties

Let's take a closer look at the properties of an indicator.



The screenshot shows the MetaEditor interface with the file 'ADX.mq5' open. The left pane displays a tree view of included files under 'Examples'. The right pane shows the source code of 'ADX.mq5'. The code includes several '#property' directives defining indicator properties like type, color, and style.

```
1 //+
2 /**
3 /**
4 /**
5 /**
6 #property copyright "2009-2017, MetaQuotes Software Corp."
7 #property link "http://www.mql5.com"
8 #property description "Average Directional Movement Index"
9 #include <MovingAverages.mqh>
10
11 #property indicator_separate_window
12 #property indicator_buffers 6
13 #property indicator_plots 3
14 #property indicator_type1 DRAW_LINE
15 #property indicator_color1 LightSeaGreen
16 #property indicator_style1 STYLE_SOLID
17 #property indicator_width1 1
18 #property indicator_type2 DRAW_LINE
19 #property indicator_color2 YellowGreen
20 #property indicator_style2 STYLE_DOT
21 #property indicator_width2 1
22 #property indicator_type3 DRAW_LINE
23 #property indicator_color3 Wheat
24 #property indicator_style3 STYLE_DOT
25 #property indicator_width3 1
26 #property indicator_label1 "ADX"
27 #property indicator_label2 "+DI"
28 #property indicator_label3 "-DI"
29 //--- input parameters
30 input int InpPeriodADX=14; // Period
31 //---- buffers
32 double ExtADXBuffer[];
```

Quote from the MQL5 Reference:

Every mql5-program allows specifying additional specific parameters named #property that help client terminal in proper servicing for programs without the necessity to launch them explicitly. This concerns the external settings of indicators, first of all. Properties described in included files are completely ignored. Properties must be specified in the main mq5-file. #property identifier value

An included file is specified using the #include keyword, followed by the path to the included file.

An included file is a frequently used block of code.

Such files may be included in a source code of experts, scripts, custom indicators and libraries at the compilation stage.

The use of included files is more preferable than the use of libraries, due to the additional overhead when calling library functions.

Included files can be placed in the same directory as a source file, in this case, the #include directive with double quotes is used.

Another place to store the included files is in the <terminal_directory> \ MQL5 \ Include directory, in this case, the #include directive with angle brackets is used.

As a first indicator property, as a rule, you specify the name of the developer, for example:

```
#property copyright
```

The following is a link to a developer's site:

```
#property link
```

After that there is a description of an indicator, each line of which is marked by the identifier description, for example:

```
#property description "Average Directional Movement Index"
```

Further, you can specify an indicator's version:

```
#property version "1.00"
```

At this, as a rule, the declaration of the indicator's general properties ends.

An indicator can appear in the terminal window in two ways - on a symbol's chart or in a separate window under a symbol's chart.

And the property:

```
#property indicator_chart_window
```

defines the indicator drawing on a symbol's chart.

And the property:

```
#property indicator_separate_window
```

defines the indicator drawing in a separate window.

One of the most important properties of an indicator is the number of buffers for calculating an indicator, for example:

```
#property indicator_buffers 6
```

This property is closely related to two other properties of the indicator - a number of plots and a type of plots.

A number of plots is the number of color diagrams that make up an indicator.

For example, for the ADX indicator:

```
#property indicator_plots 3
```

The indicator consists of three diagrams (lines) —the directivity indicator + DI, the directivity indicator -DI, and the ADX indicator itself.

A type of plots is the graphical shape from which an indicator is plotted.

For example, for the ADX indicator:

```
#property indicator_type1 DRAW_LINE
```

```
#property indicator_type2 DRAW_LINE
```

```
#property indicator_type3 DRAW_LINE
```

Thus, each chart of the ADX indicator is a line.

A graphical shape is matched to a graphic plot using the plot number following the indicator_type.

Color of each plot of an indicator is set by the indicator_colorN property.

For example, for the ADX indicator:

```
#property indicator_color1 LightSeaGreen
```

```
#property indicator_color2 YellowGreen
```

```
#property indicator_color3 Wheat
```

Color is matched to a plot using the plot number, following the indicator_color.

<https://www.mq5.com/en/docs/constants/objectconstants/webcolors>

[MQL5 Reference](#) · [Constants, Enumerations and Structures](#) · [Objects Constants](#) · [Web Colors](#)

Web Colors								
The following color constants are defined for the <code>color</code> type:								
<code>clrBlack</code>	<code>clrDarkGreen</code>	<code>clrDarkSlateGray</code>	<code>clrOlive</code>	<code>clrGreen</code>	<code>clrTeal</code>	<code>clrNavy</code>	<code>clrPurple</code>	
<code>clrMaroon</code>	<code>clrIndigo</code>	<code>clrMidnightBlue</code>	<code>clrDarkBlue</code>	<code>clrDarkOliveGreen</code>	<code>clrSaddleBrown</code>	<code>clrForestGreen</code>	<code>clrOliveDrab</code>	
<code>clrSeaGreen</code>	<code>clrDarkGoldenrod</code>	<code>clrDarkSlateBlue</code>	<code>clrSienna</code>	<code>clrMediumBlue</code>	<code>clrBrown</code>	<code>clrDarkTurquoise</code>	<code>clrDimGray</code>	
<code>clrLightSeaGreen</code>	<code>clrDarkViolet</code>	<code>clrFirebrick</code>	<code>clrMediumVioletRed</code>	<code>clrMediumSeaGreen</code>	<code>clrChocolate</code>	<code>clrCrimson</code>	<code>clrSteelBlue</code>	
<code>clrGoldenrod</code>	<code>clrMediumSpringGreen</code>	<code>clrLawnGreen</code>	<code>clrCadetBlue</code>	<code>clrDarkOrchid</code>	<code>clrYellowGreen</code>	<code>clrLimeGreen</code>	<code>clrOrangeRed</code>	
<code>clrDarkOrange</code>	<code>clrOrange</code>	<code>clrGold</code>	<code>clrYellow</code>	<code>clrChartreuse</code>	<code>clrLime</code>	<code>clrSpringGreen</code>	<code>clrAqua</code>	
<code>clrDeepSkyBlue</code>	<code>clrBlue</code>	<code>clragenta</code>	<code>clrRed</code>	<code>clrGray</code>	<code>clrStateGray</code>	<code>clrPens</code>	<code>clrBlueViolet</code>	
<code>clrLightSlateGray</code>	<code>clrDeepPink</code>	<code>clrMediumTurquoise</code>	<code>clrOliveBlue</code>	<code>clrTurquoise</code>	<code>clrRoyalBlue</code>	<code>clrSteelBlue</code>	<code>clrKhaki</code>	
<code>clrIndianRed</code>	<code>clrMediumOrchid</code>	<code>clrGreenYellow</code>	<code>clrMediumAquamarine</code>	<code>clrDarkSeaGreen</code>	<code>clrTomato</code>	<code>clrTaupeBrown</code>	<code>clrOrchid</code>	
<code>clrMediumPurple</code>	<code>clrPaleVioletRed</code>	<code>clrCoral</code>	<code>clrCornflowerBlue</code>	<code>clrDarkGray</code>	<code>clrSandyBrown</code>	<code>clrMedium slateBlue</code>	<code>clrTan</code>	
<code>clrDarkSalmon</code>	<code>clrBurlyWood</code>	<code>clrHotPink</code>	<code>clrSalmon</code>	<code>clrViolet</code>	<code>clrLightCoral</code>	<code>clrSkyBlue</code>	<code>clrLightSalmon</code>	
<code>clrPlum</code>	<code>clrKhaki</code>	<code>clrLightGreen</code>	<code>clrAquaMarine</code>	<code>clrSilver</code>	<code>clrLightSkyBlue</code>	<code>clrLightSteelBlue</code>	<code>clrLightBlue</code>	
<code>clrPaleGreen</code>	<code>clrThistle</code>	<code>clrPowderBlue</code>	<code>clrPaleGoldenrod</code>	<code>clrPaleTurquoise</code>	<code>clrLightGray</code>	<code>clrWheat</code>	<code>clrNavajoWhite</code>	

In the MQL5 Reference, there is a table of Web colors for defining the color of a plot.

```

32 //---- buffers
33 double ExtADXBuffer[];
34 double ExtPDIBuffer[];
35 double ExtNDIBuffer[];
36 double ExtPDBuffer[];
37 double ExtNDBuffer[];
38 double ExtTmpBuffer[];
39 //--- global variables
40 int ExtADXPeriod;
41 //+-----+
42 //| Custom indicator initialization function |+
43 //+-----+
44 void OnInit()
45 {
46 //--- check for input parameters
47 if (InpPeriodADX > -100 || InpPeriodADX < -0)
48 {
49     ExtADXPeriod = 14;
50     printf("Incorrect value for input variable Period_AXD-%d. Indicator will",
51 }
52 else ExtADXPeriod = InpPeriodADX;
53 //---- indicator buffers
54 SetIndexBuffer(0, ExtADXBuffer);
55 SetIndexBuffer(1, ExtPDIBuffer);
56 SetIndexBuffer(2, ExtNDIBuffer);
57 SetIndexBuffer(3, ExtPDBuffer, INDICATOR_CALCULATIONS);
58 SetIndexBuffer(4, ExtNDBuffer, INDICATOR_CALCULATIONS);
59 SetIndexBuffer(5, ExtTmpBuffer, INDICATOR_CALCULATIONS);
60 //--- indicator digits
61 IndicatorSetInteger(INDICATOR_DIGITS, 2);

```

Let's now return to the number of buffers for calculating an indicator.

Since the data for constructing each indicator chart is taken from its indicator buffer, the number of declared indicator buffers cannot be less than the declared number of indicator plots.

Immediately the question arises - how a particular array representing an indicator buffer is matched to a specific plot of an indicator.

This is done in the OnInit callback function by calling the SetIndexBuffer function.

For example, for the ADX indicator:

SetIndexBuffer (0, ExtADXBuffer);

SetIndexBuffer (1, ExtPDIBuffer);

SetIndexBuffer (2, ExtNDIBuffer);

Where the first argument is the number of the plots.

Thus, the array is associated with the indicator diagram, and the diagram is associated with its shape and color.

However, with indicator buffers, everything is a little more complicated.

Their number can be declared more than a number of plots of an indicator.

What does this mean?

This means that some arrays representing indicator buffers are used not for

building indicator diagrams, but for intermediate calculations.

For example, for the ADX indicator:

```
SetIndexBuffer (3, ExtPDBuffer, INDICATOR_CALCULATIONS);
```

```
SetIndexBuffer (4, ExtNDBuffer, INDICATOR_CALCULATIONS);
```

```
SetIndexBuffer (5, ExtTmpBuffer, INDICATOR_CALCULATIONS);
```

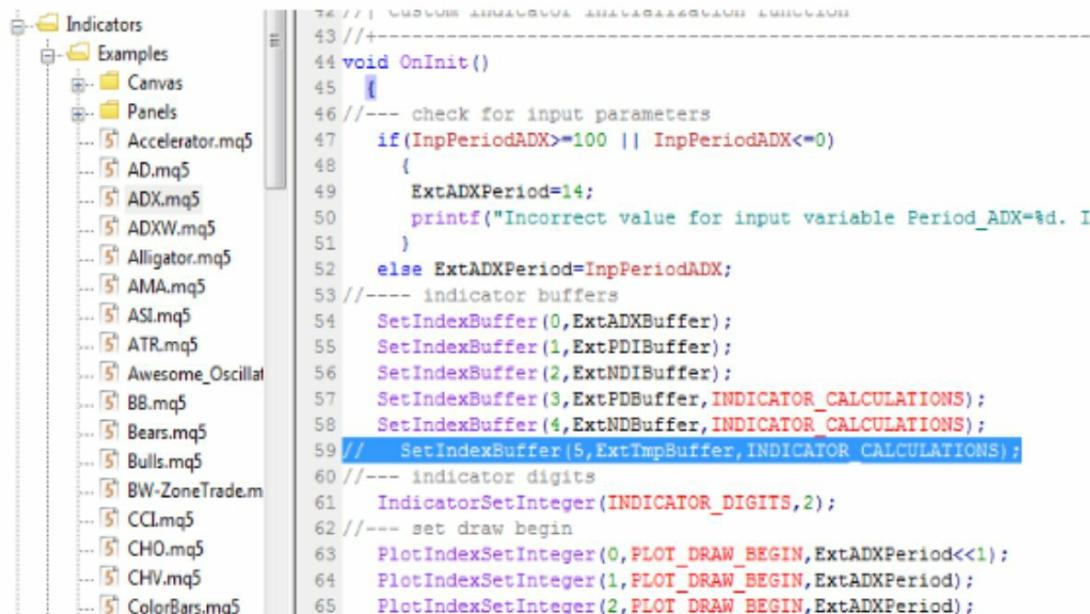
Such the array is determined using the third parameter INDICATOR_CALCULATIONS.

This gives the following:

It's all about the partial filling of the array.

If the array specified in the SetIndexBuffer function is dynamic, that is declared without specifying a size, but it is associated with an indicator buffer using the SetIndexBuffer function, the client terminal itself ensures that the size of such an array matches the price history.

let's look at it with the example of the indicator ADX.



```
76 //+
43 //+
44 void OnInit()
45 {
46 //--- check for input parameters
47 if(InpPeriodADX>=100 || InpPeriodADX<=0)
48 {
49     ExtADXPeriod=14;
50     printf("Incorrect value for input variable Period_AXD. I
51 }
52 else ExtADXPeriod=InpPeriodADX;
53 //--- indicator buffers
54 SetIndexBuffer(0,ExtADXBuffer);
55 SetIndexBuffer(1,ExtPDIBuffer);
56 SetIndexBuffer(2,ExtNDIBuffer);
57 SetIndexBuffer(3,ExtPDBuffer,INDICATOR_CALCULATIONS);
58 SetIndexBuffer(4,ExtNDBuffer,INDICATOR_CALCULATIONS);
59 // SetIndexBuffer(5,ExtTmpBuffer,INDICATOR_CALCULATIONS);
60 //--- indicator digits
61 IndicatorSetInteger(INDICATOR_DIGITS,2);
62 //--- set draw begin
63 PlotIndexSetInteger(0,PLOT_DRAW_BEGIN,ExtADXPeriod<<1);
64 PlotIndexSetInteger(1,PLOT_DRAW_BEGIN,ExtADXPeriod);
65 PlotIndexSetInteger(2,PLOT_DRAW_BEGIN,ExtADXPeriod);
```

In the MQL5 editor, in the Navigator window, in the Indicators-> Examples section, select and open the source code of the ADX indicator.

In the OnInit function, comment out the line:

```
// SetIndexBuffer (5, ExtTmpBuffer, INDICATOR_CALCULATIONS);
```

Now the ExtTmpBuffer array is simply a dynamic array.

Let's compile the indicator code and attach the indicator to a chart in the MetaTrader 5 terminal.



As a result, the terminal gives an error.

array out of range

It happened because before filling this array with values, we did not specify its size and did not reserve memory for it.

So its size was zero when we tried to write something into it.

Also, we cannot make this array static, that is declares it immediately with a defined size, since the values of such an intermediate array are calculated in the OnCalculate callback function based on the price history loaded into the OnCalculate function, namely, based on the open, high, low, and close arrays. But the exact size of the arrays open, high, low, and close is unknown, it is denoted only by the variable rates_total.

```

73 //+-----+
74 //| Custom indicator iteration function
75 //+-----+
76 int OnCalculate(const int rates_total,
77                 const int prev_calculated,
78                 const datetime &time[],
79                 const double &open[],
80                 const double &high[],
81                 const double &low[],
82                 const double &close[],
83                 const long &tick_volume[],
84                 const long &volume[],
85                 const int &spread[])
86 {
87     ArrayResize(ExtTmpBuffer, rates_total);
88 //--- checking for bars count
89     if(rates_total<ExtADXPeriod)
90         return(0);
91 //--- detect start position
92     int start;
93     if(prev_calculated>1) start=prev_calculated-1;
94     else
95     {
96         start=1;
97         ExtPDIBuffer[0]=0.0;
98         ExtNDIBuffer[0]=0.0;
99         ExtADXBuffer[0]=0.0;
100    }

```

Good, but we can use the `ArrayResize` function in the `OnCalculate` function to set the size of the array:

`ArrayResize(ExtTmpBuffer, rates_total);`

Passing the variable `rates_total` to the function as an argument - the number of bars on the chart on which an indicator is running.

Now, after compilation, the indicator will work as it should.

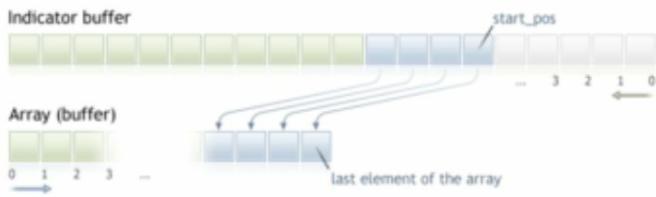
But the fact is that in the `OnCalculate` function, we first calculate the indicator for the entire price history, that is for the `rates_total` values, and then, when a new tick of the indicator's symbol arrives, and accordingly the `OnCalculate` function is called, we calculate the indicator value for this new symbol's tick and write the new indicator value into its buffer array.

To do this with an intermediate array, you need to carefully monitor its size and write the new value to the end of the array.

Instead of all this, it is easiest to bind the intermediate array to the indicator buffer using the `SetIndexBuffer` function and thus solve all these problems.

CopyBuffer

Gets data of a specified buffer of a certain indicator in the necessary quantity.



```
int CopyBuffer(
    int     indicator_handle,      // indicator handle
    int     buffer_num,           // indicator buffer number
    int     start_pos,            // start position
    int     count,                // amount to copy
    double  buffer[]             // target array to copy
);
```

A similar situation occurs when the values of such intermediate arrays are populated using the CopyBuffer function when we build a custom indicator based on other indicators.

The CopyBuffer function fits the size of the receiving array to a size of the copied data.

If the entire price history is copied, then there is no problem and in this case, it is not necessary to use the INDICATOR_CALCULATIONS.

If we want to copy only one new incoming value, the CopyBuffer function will determine the size of the receiving array as 1, and we will need to use this receiving array as another intermediary array from which to write the value to the intermediate indicator array.

And in this case, the ArrayResize function for the receiving array cannot simply solve the problem.

```

Навигатор ADX.mq5 ColorLine.mq5
-----
1 //+
2 //|
3 //| Copyright 2009-2017, MetaQuotes Software Corp. |
4 //| http://www.mql5.com |
5 //+
6 #property copyright "2009-2017, MetaQuotes Software Corp."
7 #property link "http://www.mql5.com"
8
9 #property indicator_chart_window
10 #property indicator_buffers 2
11 #property indicator_plots 1
12 //---- plot ColorLine
13 #property indicator_label1 "ColorLine"
14 #property indicator_type1 DRAW_COLOR_LINE
15 #property indicator_color1 Red,Green,Blue
16 #property indicator_style1 STYLE_SOLID
17 #property indicator_width1 3
18 //--- indicator buffers
19 double ExtColorLineBuffer[];
20 double ExtColorsBuffer[];
21 //---
22 int ExtMAHandle;
23 //+
24 //| Custom indicator initialization function
25 //+

```

Now what should we do if we want to color our indicator charts in different colors depending on a price?

Firstly, we must specify that the graphical shape of our plot is colored, for example:

```
#property indicator_type1 DRAW_COLOR_LINE
```

The word COLOR is added to the identifier of a graphical shape.

Next, a value of the #property indicator_buffers is increased by one and another array is declared for storing color.

```

10 #property indicator_buffers 2
11 #property indicator_plots 1
12 //---- plot ColorLine
13 #property indicator_label1 "ColorLine"
14 #property indicator_type1 DRAW_COLOR_LINE
15 #property indicator_color1 Red,Green,Blue
16 #property indicator_style1 STYLE_SOLID
17 #property indicator_width1 3
18 //--- indicator buffers
19 double ExtColorLineBuffer[];
20 double ExtColorsBuffer[];
21 //---
22 int ExtMAHandle;
23 //+-----+
24 //| Custom indicator initialization function |
25 //+-----+
26 void OnInit()
27 {
28 //--- indicator buffers mapping
29 SetIndexBuffer(0,ExtColorLineBuffer,INDICATOR_DATA);
30 SetIndexBuffer(1,ExtColorsBuffer,INDICATOR_COLOR_INDEX);
31 //--- get MA handle
32 ExtMAHandle=IMA(Symbol(),0,10,0,MODE_EMA,PRICE_CLOSE);
33 }

```

Using the SetIndexBuffer function, a declared additional array is matched to an indicator color buffer, for example:

`SetIndexBuffer (1, ExtColorsBuffer, INDICATOR_COLOR_INDEX);`

The #property indicator_color property of a graphic plotting color specifies several colors, for example:

`#property indicator_color1 Red, Green, Blue`

```

36 //+-----+
37 int getIndexOfColor(int i)
38 {
39     int j=i%300;
40     if(j<100) return(0);// first index
41     if(j<200) return(1);// second index
42     return(2); // third index
43 }

74     //--- now set line color for every bar
75     for(int i=0;i<rates_total && !IsStopped();i++)
76         ExtColorsBuffer[i]=getIndexOfColor(i);
77

```

And, finally, each element of the array that represents an indicator color buffer is assigned with a color number, defined in the #property indicator_color property.

In this case, it is 0, 1 and 2.

Now, when drawing a chart of an indicator, a value of the chart is taken from the buffer, and it is matched to the value of the color buffer at the position of the value, and the element of the chart becomes color.

```
12 //---- plot ColorLine
13 #property indicator_label1 "ColorLine"
14 #property indicator_type1 DRAW_COLOR_LINE
15 //#property indicator_color1 Red,Green,Blue
16 #property indicator_style1 STYLE_SOLID
17 #property indicator_width1 3
18 //--- indicator buffers
19 double ExtColorLineBuffer[];
20 double ExtColorsBuffer[];
21 //---
22 int ExtMAHandle;
23 //+
24 //| Custom indicator initialization function
25 //+
26 void OnInit()
27 {
28
29 PlotIndexSetInteger(0, PLOT_COLOR_INDEXES, 3);
30 PlotIndexSetInteger(0, PLOT_LINE_COLOR, 0, Red);
31 PlotIndexSetInteger(0, PLOT_LINE_COLOR, 1, Green);
32 PlotIndexSetInteger(0, PLOT_LINE_COLOR, 2, Blue);
33 }
```

Instead of the #property indicator_color property, colors of a plot can be set programmatically:

Set a number of color indices for the plot using the function:

```
PlotIndexSetInteger (0, PLOT_COLOR_INDEXES, 3);
```

And set the color for each index using the function:

```
PlotIndexSetInteger (0, PLOT_LINE_COLOR, 0, Red);
```

Where the first parameter is the index of the plot, respectively, the first plot has the index 0.

This is identical to the declaration:

```
#property indicator_color1 Red, Green, Blue
```

```
6 #property copyright "2009-2017, MetaQuotes Software Corp."
7 #property link "http://www.mql5.com"
8 #property description "Average Directional Movement Index"
9 #include <MovingAverages.mqh>
10
11 #property indicator_separate_window
12 //#property indicator_chart_window
13 #property indicator_buffers 6
14 #property indicator_plots 3
15 #property indicator_type1 DRAW_LINE
16 #property indicator_color1 LightSeaGreen
17 #property indicator_style1 STYLE_SOLID
18 #property indicator_width1 1
19 #property indicator_type2 DRAW_LINE
20 #property indicator_color2 YellowGreen
21 #property indicator_style2 STYLE_DOT
22 #property indicator_width2 1
23 #property indicator_type3 DRAW_LINE
24 #property indicator_color3 Wheat
25 #property indicator_style3 STYLE_DOT
26 #property indicator_width3 1
27 #property indicator_label1 "ADX"
28 #property indicator_label2 "+DI"
29 #property indicator_label3 "-DI"
```

Let's continue to look at the properties of an indicator.

A thickness of an indicator chart line is specified by the indicator_widthN property, where N is the number of the graphic plot, for example:

```
#property indicator_width1 1
```

You can also set a line style of an indicator chart - a solid line, dashed, dotted, dash-dotted, and dash - using the indicator_styleN property, where N is the number of the graphic plot, for example:

```
#property indicator_style1 STYLE_SOLID
```

And finally, the indicator_labelN property specified labels of an indicator's diagrams in the DataWindow, for example:

```
#property indicator_label1 "ADX"
```

```
#property indicator_label2 "+ DI"
```

```
#property indicator_label3 "-DI"
```

- Macro substitution (#define)
- Program Properties (#property)**
- Including Files (#include)
- Importing Functions (#import)
- Conditional Compilation (#ifdef, #ifndef, #else, #endif)

Program Properties (#property)

Every mql5-program allows to specify additional specific parameters named #property that help client terminal in proper servicing for programs without the necessity to launch them explicitly. This concerns external settings of indicators, first of all. Properties described in included files are completely ignored. Properties must be specified in the main mql5-file.

```
#property identifier value
```

The compiler will write declared values in the configuration of the module executed.

Constant	Type	Description
icon	string	Path to the file of an image that will be used as an icon of the EX5 program. Path specification rules are the same as for resources. The property must be specified in the main module with the MQL5 source code. The icon file must be in the ICO format.
link	string	Link to the company website
copyright	string	The company name
version	string	Program version, maximum 31 characters
description	string	Brief text description of a mql5-program. Several description can be present, each of them describes one line of the text. The total length of all description can not exceed 511 characters including line feed.
stacksize	int	MQL5 program stack size. The stack of sufficient size is necessary when executing function recursive calls. When launching a script or an Expert Advisor on the chart, the stack of at least 8 MB is allocated. In case of indicators, the stack size is always fixed and equal to 1 MB.

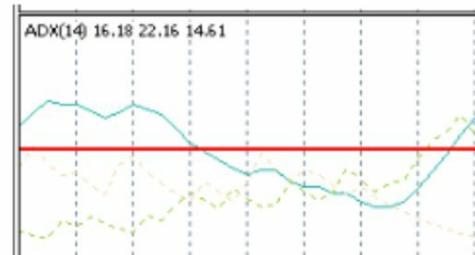
Other properties of the indicator can be viewed in the MQL5 reference.

However, it is important to note another group of properties that allows you to draw a horizontal level of an indicator in a separate window, for example:

```

11 #property indicator_separate_window
12 //#property indicator_chart_window
13 #property indicator_buffers 6
14 #property indicator_plots 3
15 #property indicator_type1 DRAW_LINE
16 #property indicator_color1 LightSeaGreen
17 #property indicator_style1 STYLE_SOLID
18 #property indicator_width1 1
19 #property indicator_type2 DRAW_LINE
20 #property indicator_color2 YellowGreen
21 #property indicator_style2 STYLE_DOT
22 #property indicator_width2 1
23 #property indicator_type3 DRAW_LINE
24 #property indicator_color3 Wheat
25 #property indicator_style3 STYLE_DOT
26 #property indicator_width3 1
27 #property indicator_label1 "ADX"
28 #property indicator_label2 "+DI"
29 #property indicator_label3 "-DI"
30
31 #property indicator_level1 30.0
32 #property indicator_levelcolor Red
33 #property indicator_levelstyle STYLE_SOLID
34 #property indicator_levelwidth 2

```

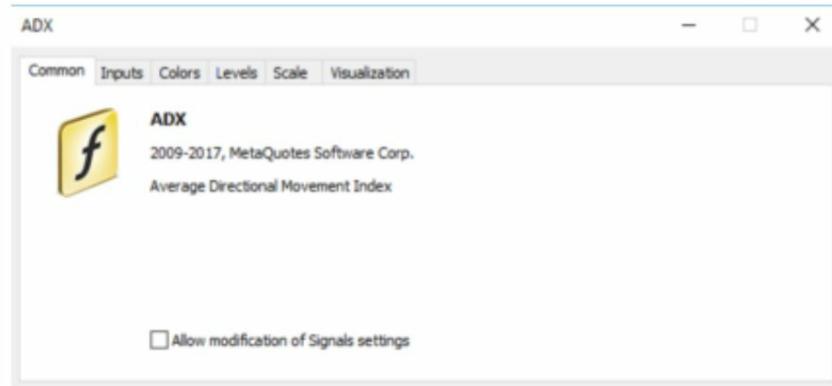


```
#property indicator_level1 30.0
#property indicator_levelcolor Red
```

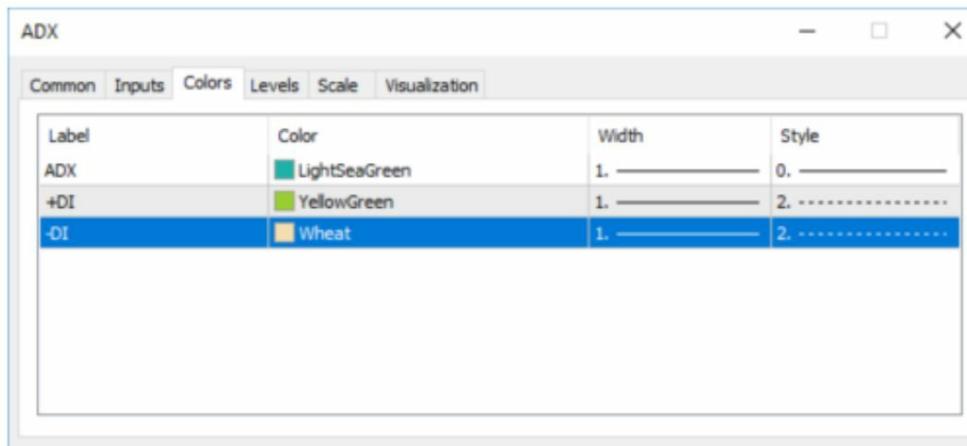
```
#property indicator_levelstyle STYLE_SOLID
```

```
#property indicator_levelwidth 2
```

As a result of adding these lines of code to the ADX indicator, it will have a horizontal level.



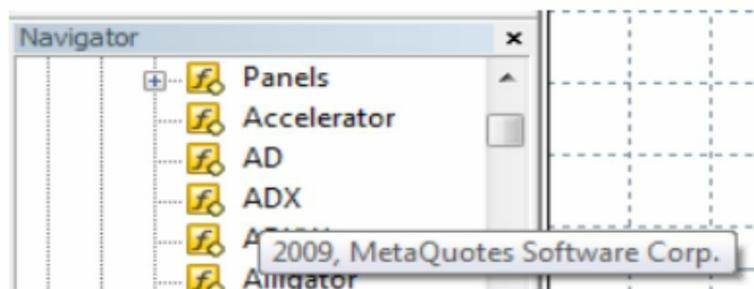
Now, using the example of the ADX indicator, when attaching the indicator to a chart in MetaTrader 5, firstly, the indicator dialog box opens, which displays values of the copyright, link and description properties in the Common tab.



And in the Colors tab, you can see values of the `indicator_label`, `indicator_color`, `indicator_width`, `indicator_style` properties.

A name of an indicator itself is determined by the name of the indicator file.

By the way, the indicator dialog box can also be opened after an indicator is attached to a chart using the context menu by right-clicking on the indicator and selecting the indicator properties.



When you hover the cursor over the name of an indicator in the Navigator window of the terminal, a prompt pops up that displays the copyright property.

After attaching an indicator, the property:

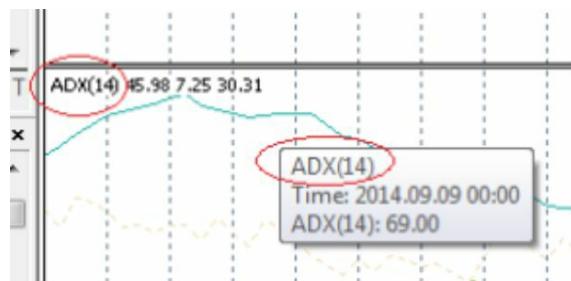
```
#property indicator_label1 "ADX"
```

will not work, since in the OnInit function, using the function call:

```
string short_name = "ADX (" + string(ExtADXPeriod) + ")";
```

```
IndicatorSetString(INDICATOR_SHORTNAME, short_name);
```

The indicator signature has been changed to ADX (14) - the indicator period.



```
#property indicator_label1 "ADX"
```

```
67 //--- indicator short name
68     string short_name="ADX("+string(ExtADXPeriod)+")";
69     IndicatorSetString(INDICATOR_SHORTNAME,short_name);
```

And by calling the function:

```
PlotIndexSetString(0, PLOT_LABEL, short_name);
```

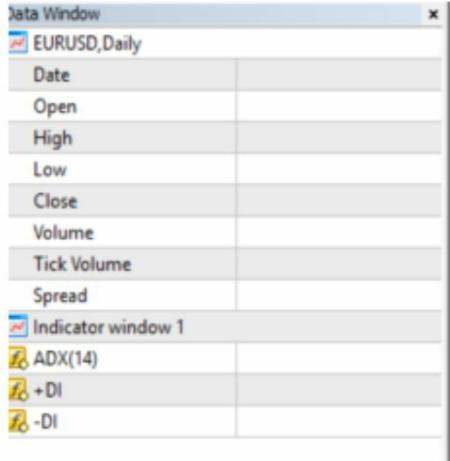
the indicator label has been changed in the Data Window that opens in the View menu of the terminal.

Values of the properties:

```
#property indicator_label2 "+ DI"
```

```
#property indicator_label3 "-DI"
```

are displayed, as it was determined, in the tooltips to the indicator diagrams and displayed in the Data Window.



```
70 //--- change 1-st index label  
71 PlotIndexSetString(0, PLOT_LABEL, short_name);
```

In the ADX indicator code, the declared number of indicator buffers is more than the number of plots:

The indicator_buffers property is 6

But the indicator_plots property is 3

This is done in order to use three indicator buffers for intermediate calculations:

These are the ExtPDBuffer, ExtNDBuffer and ExtTmpBuffer arrays.

In the OnCalculate indicator's function, values of the ExtPDBuffer, ExtNDBuffer, ExtTmpBuffer arrays are calculated based on the loaded price history, and then, the values of the ExtADXBuffer, ExtPDIBuffer, ExtNDIBuffer arrays, which are used to draw indicator's diagrams, are calculated based on them.

As already mentioned, indicator buffers for intermediate calculations are marked here with the INDICATOR_CALCULATIONS constant, since the size of the loaded price history is not known in advance.

```

5 //+
6 #property copyright "2009-2017, MetaQuotes Software Corp."
7 #property link "http://www.mql5.com"
8 #property description "Average Directional Movement Index"
9 #include <MovingAverages.mqh>
10
11 #property indicator_separate_window
12 //#property indicator_chart_window
13 #property indicator_buffers 6
14 #property indicator_plots 3
15 #property indicator_type1 DRAW_LINE
16 #property indicator_color1 LightSeaGreen
17 #property indicator_style1 STYLE_SOLID
18 #property indicator_width1 1
19 #property indicator_type2 DRAW_LINE
20 #property indicator_color2 YellowGreen
21 #property indicator_style2 STYLE_DOT
22 #property indicator_width2 1
23 #property indicator_type3 DRAW_LINE
24 #property indicator_color3 Wheat
25 #property indicator_style3 STYLE_DOT
26 #property indicator_width3 1
27 #property indicator_label1 "ADX"
28 #property indicator_label2 "+DI"
29 #property indicator_label3 "-DI"
30

```

```

54 //---- indicator buffers
55 SetIndexBuffer(0,ExtADXBuffer);
56 SetIndexBuffer(1,ExtPDIBuffer);
57 SetIndexBuffer(2,ExtNDIBuffer);
58 SetIndexBuffer(3,ExtPDBuffer,INDICATOR_CALCULATIONS);
59 SetIndexBuffer(4,ExtNDBuffer,INDICATOR_CALCULATIONS);
60 SetIndexBuffer(5,ExtTmpBuffer,INDICATOR_CALCULATIONS);

```

Now, in the description of the ADX indicator, it is said that:

A buy signal is generated when the + DI rises higher than the - DI and at the same time, the ADX itself grows.

At the moment when the + DI is located higher than the - DI, but the ADX itself begins to decline, the indicator gives a signal that the market is “overheated” and the time has come to take profits.

A sell signal is generated when the + DI drops below the - DI and at the same time the ADX grows.

At the moment when the + DI is located below the - DI, but the ADX itself starts to decline, the indicator gives a signal that the market is “overheated” and the time has come to take profits.

Let's modify the ADX indicator code in such a way as to color the ADX chart in four colors that correspond to the four trading signals described above.



As a first step, let's change the `indicator_type1` property to `DRAW_COLOR_LINE`.

Next, increase by one the value of the `indicator_buffers` property by the value 7.

Let's declare an array for the `ExtColorsBuffer` color buffer.

```

12 // #property indicator_chart_window
13 #property indicator_buffers 7
14 #property indicator_plots 3
15 #property indicator_type1 DRAW_COLOR_LINE
16 #property indicator_color1 LightSeaGreen
17 #property indicator_style1 STYLE_SOLID
18 #property indicator_width1 1
19 #property indicator_type2 DRAW_LINE
20 #property indicator_color2 YellowGreen
21 #property indicator_style2 STYLE_DOT
22 #property indicator_width2 1
23 #property indicator_type3 DRAW_LINE
24 #property indicator_color3 Wheat
25 #property indicator_style3 STYLE_DOT
26 #property indicator_width3 1
27 #property indicator_label1 "ADX"
28 #property indicator_label2 "+DI"
29 #property indicator_label3 "-DI"
30
31 //--- input parameters
32 input int InpPeriodADX=14; // Period
33 //--- buffers
34 double ExtADXBuffer[];
35 double ExtPDIBuffer[];
36 double ExtNDIBuffer[];
37 double ExtPDBuffer[];
38 double ExtNDBuffer[];
39 double ExtTmpBuffer[];
40 //By usera
41 double ExtColorsBuffer[];
```

And in the `OnInit` function, we associate the declared array with the color

buffer using the SetIndexBuffer function.

There is a trick here - the color buffer index should follow the index of the indicator value buffer.

If, for example, you associate the ExtColorsBuffer array with the buffer that has the index 6, then the indicator will not be correctly drawn.

```
47 void OnInit()
48 {
49 //--- check for input parameters
50 if(InpPeriodADX>=100 || InpPeriodADX<=0)
51 {
52     ExtADXPeriod=14;
53     printf("Incorrect value for input variable Period_AXD=%d.
54 }
55 else ExtADXPeriod=InpPeriodADX;
56 //---- indicator buffers
57 SetIndexBuffer(0,ExtADXBuffer);
58 SetIndexBuffer(1,ExtColorsBuffer,INDICATOR_COLOR_INDEX);
59 SetIndexBuffer(2,ExtPDIBuffer);
60 SetIndexBuffer(3,ExtNDIBuffer);
61 SetIndexBuffer(4,ExtPDBuffer,INDICATOR_CALCULATIONS);
62 SetIndexBuffer(5,ExtNDBuffer,INDICATOR_CALCULATIONS);
63 SetIndexBuffer(6,ExtTmpBuffer,INDICATOR_CALCULATIONS);
```

Let's add colors to indicator_color1 property.

And increase the line thickness using the indicator_width1 property.

```

11 #property indicator_separate_window
12 //#property indicator_chart_window
13 #property indicator_buffers 7
14 #property indicator_plots 3
15 #property indicator_type1 DRAW_COLOR_LINE
16 #property indicator_color1 LightSeaGreen, clrBlue, clrLightBlue, clrRed, clrLightPink
17 #property indicator_style1 STYLE_SOLID
18 #property indicator_width1 2
19 #property indicator_type2 DRAW_LINE
20 #property indicator_color2 YellowGreen
21 #property indicator_style2 STYLE_DOT
22 #property indicator_width2 1
23 #property indicator_type3 DRAW_LINE
24 #property indicator_color3 Wheat
25 #property indicator_style3 STYLE_DOT
26 #property indicator_width3 1
27 #property indicator_label1 "ADX"
28 #property indicator_label2 "+DI"
29 #property indicator_label3 "-DI"

```

In the OnCalculate function, in the end, before the closing bracket of the for loop, let's add the code for filling the color buffer with values according to the strategy we described.

```

155 ExtColorsBuffer[i]=0;
156 if(ExtPDIBuffer[i]>ExtNDIBuffer[i]&&ExtADXBuffer[i]>ExtADXBuffer[i-1]){
157 ExtColorsBuffer[i]=1;
158 }
159 if(ExtPDIBuffer[i]>ExtNDIBuffer[i]&&ExtADXBuffer[i]<ExtADXBuffer[i-1]){
160 ExtColorsBuffer[i]=2;
161 }
162 if(ExtPDIBuffer[i]<ExtNDIBuffer[i]&&ExtADXBuffer[i]>ExtADXBuffer[i-1]){
163 ExtColorsBuffer[i]=3;
164 }
165 if(ExtPDIBuffer[i]<ExtNDIBuffer[i]&&ExtADXBuffer[i]<ExtADXBuffer[i-1]){
166 ExtColorsBuffer[i]=4;
167 }

```

Let's compile the code and get the indicator with a visual display of buy and sell signals.



In the MQL5 editor, let's open another indicator RSI from the Examples folder.

This indicator has two key levels that define overbought and oversold areas.

In the indicator code, these levels are defined as properties:

```
#property indicator_level1 30
#property indicator_level2 70
```

Let's improve the display of these levels by adding colors and styles to them.

```

6 #property copyright "2009-2017, MetaQuotes Software Corp."
7 #property link "http://www.mql5.com"
8 #property description "Relative Strength Index"
9 //--- indicator settings
10 #property indicator_separate_window
11 #property indicator_minimum 0
12 #property indicator_maximum 100
13 #property indicator_level1 30
14 #property indicator_level2 70
15 #property indicator_buffers 3
16 #property indicator_plots 1
17 #property indicator_type1 DRAW_LINE
18 #property indicator_color1 DodgerBlue
19 //--- input parameters
20 input int InpPeriodRSI=14; // Period
21 //--- indicator buffers
22 double ExtRSIBuffer[];
23 double ExtPosBuffer[];
24 double ExtNegBuffer[];
25 //--- global variable
26 int ExtPeriodRSI;
27 //+-----
```

To do this, let's add the properties:

```
#property indicator_levelcolor Red
#property indicator_levelstyle STYLE_SOLID
#property indicator_levelwidth 1
```

```

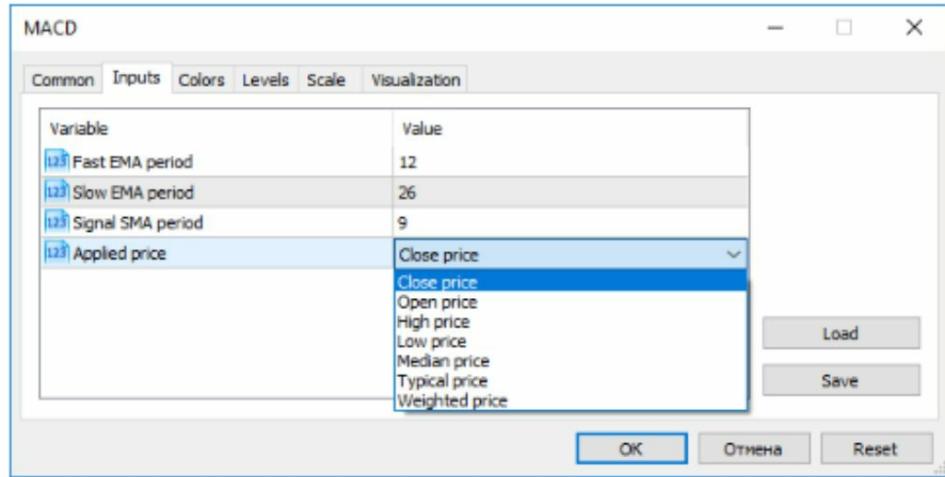
6 #property copyright "2009-2017, MetaQuotes Software Corp."
7 #property link "http://www.mql5.com"
8 #property description "Relative Strength Index"
9 //--- indicator settings
10 #property indicator_separate_window
11 #property indicator_minimum 0
12 #property indicator_maximum 100
13 #property indicator_level1 30
14 #property indicator_level2 70
15 #property indicator_buffers 3
16 #property indicator_plots 1
17 #property indicator_type1 DRAW_LINE
18 #property indicator_color1 DodgerBlue
19
20 #property indicator_levelcolor Red
21 #property indicator_levelstyle STYLE_SOLID
22 #property indicator_levelwidth 1
--
```

Now the indicator will look like this.



Input Parameters and Indicator Variables

Input parameters are the parameters of an indicator that are displayed to a user before attaching the indicator to a chart in the Inputs tab of the dialog box.



For example, for the MACD indicator:

These are periods of moving averages and a type of price applied.

Here a user can change the default parameters of an indicator, and the indicator will be attached to a chart with the already changed parameters.

A user can also change the parameters of an indicator after attaching the indicator to a chart, by right-clicking on the indicator and selecting the indicator properties.

```

6 #property copyright "2009-2017, MetaQuotes Software Corp."
7 #property link "http://www.mql5.com"
8 #property description "Moving Average Convergence/Divergence"
9 #include <MovingAverages.mqh>
10 //--- indicator settings
11 #property indicator_separate_window
12 #property indicator_buffers 4
13 #property indicator_plots 2
14 #property indicator_type1 DRAW_HISTOGRAM
15 #property indicator_type2 DRAW_LINE
16 #property indicator_color1 Silver
17 #property indicator_color2 Red
18 #property indicator_width1 2
19 #property indicator_width2 1
20 #property indicator_label1 "MACD"
21 #property indicator_label2 "Signal"
22 //--- input parameters
23 input int InpFastEMA=12; // Fast EMA period
24 input int InpSlowEMA=26; // Slow EMA period
25 input int InpSignalSMA=9; // Signal SMA period
26 input ENUM_APPLIED_PRICE InpAppliedPrice=PRICE_CLOSE; // Applied price
27 //--- indicator buffers
28 double ExtMacdBuffer[];
29 double ExtSignalBuffer[];
30 double ExtFastMaBuffer[];
31 double ExtSlowMaBuffer[];
32 //--- MA handles
33 int ExtFastMaHandle;
34 int ExtSlowMaHandle;
35 //-----

```

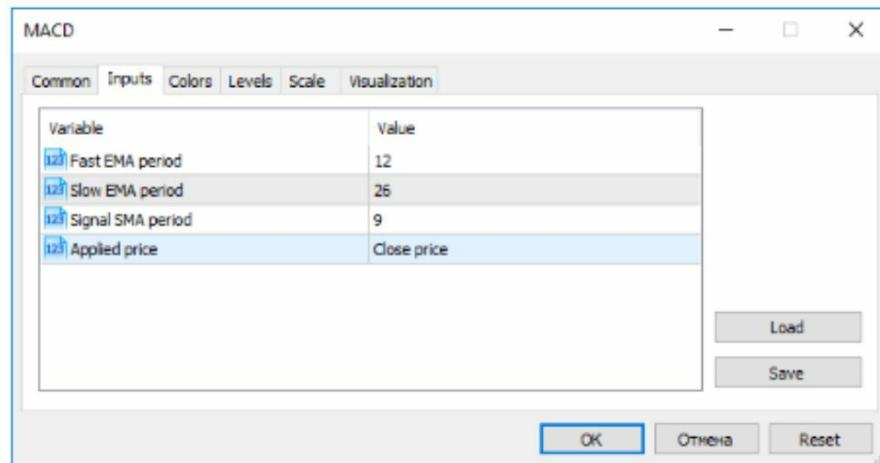
In the indicator code, such parameters are set by Input variables with the modifier “input”, which is specified before the data type. As a rule, input variables are declared immediately after indicator properties.

For example, for the MACD indicator, these are periods for
The exponential moving average with a short period of price.
The exponential moving average with a long period of price.
The smoothed moving average with a short period from the difference of the other two moving averages.

And a type of price applied.

Here it should be noted that in the dialog box for attaching the indicator to a chart, not names of the variables, but comments to them are displayed.

If you remove the comments, the input parameters are displayed as follows:



The variable names are displayed here.

As you have probably already guessed, comments are used for display in order to facilitate a user to understand their purpose.

Here you can see that the input parameters can be not only individual variables but also enumerations, which are displayed as drop-down lists.

```
22 //--- input parameters
23 input int           InpFastEMA=12;          // Fast EMA period
24 input int           InpSlowEMA=26;         // Slow EMA period
25 input int           InpSignalSMA=9;        // Signal SMA period
26 input ENUM_APPLIED PRICE InpAppliedPrice=PRICE_CLOSE; // Applied price
27 //--- indicator buffers
28 double              ExtMacdBuffer[];
29 double              ExtSignalBuffer[];
30 double              ExtFastMaBuffer[];
31 double              ExtSlowMaBuffer[];
```

For the MACD indicator, the built-in enum `ENUM_APPLIED_PRICE` is used, but you can also define your own enumeration.

In the MQL5 reference there is a corresponding example:

```
#property script_show_inputs
//-- day of week
enum dayOfWeek
{
    S=0, // Sunday
    M=1, // Monday
    T=2, // Tuesday
    W=3, // Wednesday
    Th=4, // Thursday
    Fr=5, // Friday,
    St=6, // Saturday
};
//-- input parameters
input dayOfWeek swapday=W;
```

In this example, the `#property script_show_inputs` command is used for scripts, and it can be omitted for indicators.

The main difference between input variables and other types of variables is that only a user can change their value in the indicator dialog box.

If you try to change the value of the input parameter in the indicator code, an error will occur during compilation.

The screenshot shows the MetaEditor interface. On the left is a file tree with several .mq5 files listed. In the center is a code editor window displaying C++ code related to moving averages. At the bottom is a status bar showing build results.

Description	File
• 'MACD.mq5'	
• 'MovingAverages.mqh'	
✖ 'InpFastEMA' - constant cannot be modified	MACD.mq5
• 1 error(s), 0 warning(s)	

Therefore, if you want to use a modified value of an input parameter in the calculations, you need to use an intermediate variable.

The screenshot shows the MetaEditor IDE interface. On the left, there is a file tree containing several MQL5 files: ColorLine.mq5, Custom Moving Average.mq5, DEMA.mq5, DeMarker.mq5, DPO.mq5, Envelopes.mq5, Force_Index.mq5, Fractals.mq5, and a file named 'Справка.mqh'. Below the file tree is a status bar with the text 'Description' and 'File'. To the right of the file tree is a code editor window displaying MQL5 code. The code includes initialization logic for moving averages and convergence calculations. At the bottom of the code editor, there is a toolbar with three icons.

Description	File
• 'MACD.mq5'	
• 'MovingAverages.mqh'	
• 0 error(s), 0 warning(s)	

```
53 ExtSlowMaHandle=iMA(NULL,0,I);
54 //--- initialization done
55 int tempInpFastEMA=InpFastEMA;
56 tempInpFastEMA=15;
57 }
58 //+-----+
59 //| Moving Averages Convergence.
60 //+-----+
61 int OnCalculate(const int rates
```

In addition to the input variables, the MQL5 code uses local variables, static variables, global variables, and extern variables.

Storage Classes

There are three storage classes: [static](#), [input](#) and [extern](#). These modifiers of a storage class explicitly indicate to the compiler that corresponding variables are distributed in a pre-allocated area of memory, which is called the global pool. Besides, these modifiers indicate the special processing of variable data. If a variable declared on a local level is not a [static](#) one, memory for such a variable is allocated automatically at a program stack. Freeing of memory allocated for a non-static array is also performed automatically when going beyond the visibility area of the block, in which the array is declared.

In principle, everything is clear with local variables - they are declared in a code block, for example, in a loop or function, initialized there, and, after the code block is executed, the memory allocated for local variables in the software stack is released.

Here it should be especially noted that for local objects created using the operator “new”, at the end of the code block, you must use the operator “delete” to free memory.

Global variables, as a rule, are declared after the indicator properties, input parameters, and indicator buffer arrays, but before functions.

Global variables are visible within the entire program, their value can be changed anywhere in a program, and the memory allocated for global variables outside the program stack is released when the program is unloaded.

Here you can see that the input variables are the same as global variables, with the exception of the option - their value cannot be changed anywhere in the program.

If you declare a global or local variable with the modifier “const”, it also does not allow changing the value of this variable during program execution.

Static variables are defined by the modifier “static”, which is specified before a data type.

With static variables, everything is a bit more complicated, but it is easiest to understand them by comparing static variables with local and global variables.

In principle, a static variable declared in the same place as a global variable is not different from a global variable.

A trick begins if a local variable is declared with the modifier “static”.

In this case, after executing a block of code, the memory allocated for a static variable is not released. And the next time you run the same block of code, the previous value of the static variable can be used.

Although the scope of such a static variable is limited to the same block of code in which it was declared.

Extern variables are analogous to static global variables. You cannot declare a local variable with the modifier “extern”.

The difference between extern variables and static global variables is easiest to demonstrate on the MACD indicator.

```

1 //-----+ MACD.mq5 |
2 //| Copyright 2009-2017, MetaQuotes Software Corp. |
3 //| http://www.mql5.com |
5 //+-----+
6 #property copyright "2009-2017, MetaQuotes Software Corp."
7 #property link "http://www.mql5.com"
8 #property description "Moving Average Convergence/Divergence"
9 #include <MovingAverages.mqh>
10 //--- indicator settings
11 #property indicator_separate_window
12 #property indicator_buffers 4
13 #property indicator_plots 2
14 #property indicator_type1 DRAW_HISTOGRAM
15 #property indicator_type2 DRAW_LINE
16 #property indicator_color1 Silver
17 #property indicator_color2 Red
18 #property indicator_width1 2
19 #property indicator_width2 1
20 #property indicator_label1 "MACD"
21 #property indicator_label2 "Signal"

```

The MACD indicator has the MovingAverages include file declared by the #include directive and located in the Include folder.

If you simultaneously declare an extern variable in the MovingAverages file and in the MACD file:

```
extern int a = 0;
```

then when compiling both files, everything will go well, and the variable can be used.

If in the file Moving

If you simultaneously declare a static global variable in the MovingAverages file and in the MACD file:

```
static int a = 0;
```

then when compiling both files, an error will occur:

The screenshot shows the MetaEditor interface. On the left is a file tree with the following structure:

- ...\\Object.mqh
- ...\\StdLibErr.mqh
- ...\\tradealgorithms.mqh
- ...\\VirtualKeys.mqh
- Indicators
- Examples
- Canvas

The main area is a code editor with the following code:

```
37
38 static| int a=0;
39 //+-----
40 //| Custom indica
41 //+-----
42 void OnInit()
```

The status bar at the bottom shows:

Description	File
» 'MACD.mq5'	
» 'MovingAverages.mqh'	
» 'a' - variable already defined	MACD.mq5
» 1 error(s), 0 warning(s)	

In addition to the #include command, the #define directive is also useful, which allows you to substitute an expression instead of an identifier, for example:

```
#define ABC          100
#define PI           3.14
#define COMPANY_NAME "MetaQuotes Software Corp."
...
void ShowCopyright()
{
    Print("Copyright 2001-2009, ",COMPANY_NAME);
    Print("https://www.metaquotes.net");
}
```

#define PI 3.14

And here, at compilation, everywhere where the PI will be found, it will substitute 3.14.

Indicator Handle

Let's start with a quote:

The HANDLE identifies an object that you can manipulate. Jeffrey RICHTER.

- [iAC](#)
- [iAD](#)
- [iADX](#)
- [iADXWilder](#)
- [iAlligator](#)
- **iMA**
- [iAO](#)
- [iATR](#)
- [iBearsPower](#)
- [iBands](#)

iMA

The function returns the handle of the *Moving Average* indicator. It has only one buffer.

```
int iMA(
    string          symbol,           // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             ma_period,        // averaging period
    int             ma_shift,         // horizontal shift
    ENUM_MA_METHOD ma_method,        // smoothing type
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

Variables of type handle represent a pointer to a certain system structure or index in a system table that contains an address of the structure.

Thus, having received a handle of some indicator, we can use its data to build a custom indicator.

The indicator handle is a variable of type “int” and it is declared, as a rule, after the declaration of indicator buffer arrays, together with global variables, for example, in the MACD indicator.

```

27 //--- indicator buffers
28 double ExtMacdBuffer[];
29 double ExtSignalBuffer[];
30 double ExtFastMaBuffer[];
31 double ExtSlowMaBuffer[];
32 //--- MA handles
33 int ExtFastMaHandle;
34 int ExtSlowMaHandle;
35 //+
36 //| Custom indicator initialization function
37 //+
38 void OnInit()
39 {
40 //--- indicator buffers mapping
41 SetIndexBuffer(0,ExtMacdBuffer,INDICATOR_DATA);
42 SetIndexBuffer(1,ExtSignalBuffer,INDICATOR_DATA);
43 SetIndexBuffer(2,ExtFastMaBuffer,INDICATOR_CALCULATIONS);
44 SetIndexBuffer(3,ExtSlowMaBuffer,INDICATOR_CALCULATIONS);
45 //--- sets first bar from what index will be drawn
46 PlotIndexSetInteger(1,PLOT_DRAW_BEGIN,InpSignalsSMA-1);
47 //--- name for Dindicator subwindow label
48 IndicatorSetString(INDICATOR_SHORTNAME,"MACD ("+string(InpFastEMA)+","+
49 //--- get MA handles
50 ExtFastMaHandle=iMA(NULL,0,InpFastEMA,0,MODE_EMA,InpAppliedPrice);
51 ExtSlowMaHandle=iMA(NULL,0,InpSlowEMA,0,MODE_EMA,InpAppliedPrice);
52 //--- initialization done
53 }

```

Here, two handles are declared.

int ExtFastMaHandle;

and int ExtSlowMaHandle;

Here, the indicator handles are pointers to the moving average indicators with different periods of 12 and 26.

Having declared these variables, we naturally really get nothing, since the indicator object does not yet exist, which we want to use data of.

You can create a copy of the corresponding technical indicator in the global cache of the client terminal and get a link to it in several ways.

If it is a standard indicator, the easiest way to get its handle is using a standard function for working with technical indicators.

iMA

The function returns the handle of the Moving Average indicator. It has only one buffer.

```
int iMA(
    string          symbol,           // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             ma_period,        // averaging period
    int             ma_shift,         // horizontal shift
    ENUM_MA_METHOD ma_method,        // smoothing type
    ENUM_APPLIED_PRICE applied_price // type of price or handle
);
```

The standard function for the moving average indicator is the iMA function. And in the MACD indicator, the moving average indicator handles are obtained by calling the iMA function in the OnInit function.

```
49 //--- get MA handles
50     ExtFastMaHandle=iMA(NULL,0,InpFastEMA,0,MODE_EMA,InpAppliedPrice);
51     ExtSlowMaHandle=iMA(NULL,0,InpSlowEMA,0,MODE_EMA,InpAppliedPrice);

22 //--- input parameters
23 input int             InpFastEMA=12;                      // Fast EMA period
24 input int             InpSlowEMA=26;                     // Slow EMA period
25 input int             InpSignalSMA=9;                    // Signal SMA period
26 input ENUM_APPLIED_PRICE InpAppliedPrice=PRICE_CLOSE; // Applied price
```

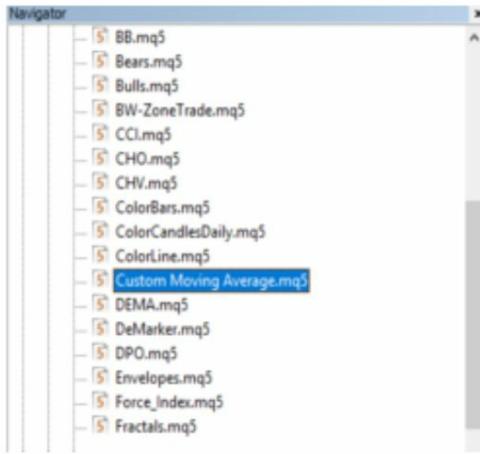
Where the indicator properties are used:

InpFastEMA

InpSlowEMA

And InpAppliedPrice

Suppose, we want to use a custom indicator, not a standard one.



```
1 //+-----+  
2 //| Copyright 2009-2017, MetaQuotes Software Corp. |  
3 //| http://www.mql5.com |  
4 //+-----+  
5 //+-----+  
6 #property copyright "2009-2017, MetaQuotes Software Corp."  
7 #property link      "http://www.mql5.com"  
8  
9 //--- indicator settings  
10 #property indicator_chart_window  
11 #property indicator_buffers 1  
12 #property indicator_plots   1  
13 #property indicator_type1  DRAW_LINE  
14 #property indicator_color1 Red  
15 //--- input parameters  
16 input int           InpMAPeriod=13;          // Period  
17 input int           InpMAShift=0;            // Shift  
18 input ENUM_MA_METHOD InpMAMethod=MODE_SMA; // Method  
19 //--- indicator buffers  
20 double              ExtLineBuffer[];
```

As you can see in the Indicators/Examples folder of the MQL5 editor, we have the indicator we need - this is the Custom Moving Average.mq5 file. To call that indicator, you can use the iCustom function.

iCustom

The function returns the handle of a specified custom indicator.

```
int iCustom(
    string          symbol,      // symbol name
    ENUM_TIMEFRAMES period,     // period
    string          name,        // folder/custom_indicator_name
    ...
);
```

In the OnInit function of the MACD indicator, we change the code, and there we use the iCustom function to get handles instead of the standard function.

```
51 // ExtFastMaHandle=iMA(NULL,0,InpFastEMA,0,MODE_EMA,InpAppliedPrice);
52 // ExtSlowMaHandle=iMA(NULL,0,InpSlowEMA,0,MODE_EMA,InpAppliedPrice);
53 ExtFastMaHandle=iCustom(NULL,0,"Examples\\Custom Moving Average", InpFastEMA,0,MODE_EMA,InpAppliedPrice);
54 ExtSlowMaHandle=iCustom(NULL,0,"Examples\\Custom Moving Average", InpSlowEMA,0,MODE_EMA,InpAppliedPrice);
```

After compiling the indicator, we can see that its display has not changed.



Another way to get a handle of a custom indicator is to use the `IndicatorCreate` function.

IndicatorCreate

The function returns the handle of a specified technical indicator created based on the array of parameters of [MqlParam](#) type.

```
int IndicatorCreate(
    string      symbol,           // symbol name
    ENUM_TIMEFRAMES period,       // timeframe
    ENUM_INDICATOR  indicator_type, // indicator type from the enumeration ENUM_INDICATOR
    int          parameters_cnt=0, // number of parameters
    const MqlParam& parameters_array[]=NULL, // array of parameters
);
```

In the OnInit function of the MACD indicator, we change the code where we use the IndicatorCreate function to get the handles.

```

51 // ExtFastMaHandle=iMA(NULL,0,InpFastEMA,0,MODE_EMA,InpAppliedPrice);
52 // ExtSlowMaHandle=iMA(NULL,0,InpSlowEMA,0,MODE_EMA,InpAppliedPrice);
53 // ExtFastMaHandle=iCustom(NULL,0,"Examples\\Custom Moving Average", InpFastEMA,0,MODE_EMA,InpAppliedPrice);
54 // ExtSlowMaHandle=iCustom(NULL,0,"Examples\\Custom Moving Average", InpSlowEMA,0,MODE_EMA,InpAppliedPrice);
55
56 MqlParam params[];
57 ArrayResize(params,5);
58 params[0].type      =TYPE_STRING;
59 params[0].string_value="Examples\\Custom Moving Average";
60 //--- set ma_period
61   params[1].type      =TYPE_INT;
62   params[1].integer_value=InpFastEMA;
63 //--- set ma_shift
64   params[2].type      =TYPE_INT;
65   params[2].integer_value=0;
66 //--- set ma_method
67   params[3].type      =TYPE_INT;
68   params[3].integer_value=MODE_EMA;
69 //--- set applied_price
70   params[4].type      =TYPE_INT;
71   params[4].integer_value=InpAppliedPrice;
72
73 ExtFastMaHandle=IndicatorCreate(NULL,NULL,IND_CUSTOM,4,params);
74 params[1].integer_value=InpSlowEMA;
75 ExtSlowMaHandle=IndicatorCreate(NULL,NULL,IND_CUSTOM,4,params);
76

```

After compiling the indicator, we again see that its display has not changed.
 After getting a handle of the indicator, if it is used in the code once, it is not bad to use the IndicatorRelease function to save memory.

IndicatorRelease

The function removes an indicator handle and releases the calculation block of the indicator, if it's not used by anyone else.

```
bool IndicatorRelease(
    int      indicator_handle // indicator handle
);
```

Here, we remove the handle of the indicator and releases the calculated part of the indicator.

Well, we got the indicator handle. How now to extract its data?

This is done in the OnCalculate function using the CopyBuffer function.

Call by the first position and the number of required elements

```
int CopyBuffer(
    int     indicator_handle,      // indicator handle
    int     buffer_num,           // indicator buffer number
    int     start_pos,            // start position
    int     count,                // amount to copy
    double  buffer[]             // target array to copy
);
```

Call by the start date and the number of required elements

```
int CopyBuffer(
    int     indicator_handle,      // indicator handle
    int     buffer_num,           // indicator buffer number
    datetime start_time,          // start date and time
    int     count,                // amount to copy
    double  buffer[]             // target array to copy
);
```

Call by the start and end dates of a required time interval

```
int CopyBuffer(
    int     indicator_handle,      // indicator handle
    int     buffer_num,           // indicator buffer number
    datetime start_time,          // start date and time
    datetime stop_time,            // end date and time
    double  buffer[]             // target array to copy
);
```

The function CopyBuffer adjusts a size of the receiving array to a size of the copied data.

Let's remind that this is working if the receiving array is simply a dynamic array.

If the receiving array is associated with an indicator buffer, then the client terminal itself takes care that the size of such an array matches a number of bars available to the indicator for calculation.

This is the situation in the MACD indicator.

```
35 //+-----+
36 //| Custom indicator initialization function
37 //+-----+
38 void OnInit()
39 {
40 //--- indicator buffers mapping
41 SetIndexBuffer(0,ExtMacdBuffer,INDICATOR_DATA);
42 SetIndexBuffer(1,ExtSignalBuffer,INDICATOR_DATA);
43 SetIndexBuffer(2,ExtFastMaBuffer,INDICATOR_CALCULATIONS);
44 SetIndexBuffer(3,ExtSlowMaBuffer,INDICATOR_CALCULATIONS);
```

The intermediate arrays ExtFastMaBuffer and ExtSlowMaBuffer are bound to the indicator buffers using the SetIndexBuffer function.

```
118 //--- get Fast EMA buffer
119     if(IsStopped()) return(0); //Checking for stop flag
120     if(CopyBuffer(ExtFastMaHandle,0,0,to_copy,ExtFastMaBuffer)<=0)
121     {
122         Print("Getting fast EMA is failed! Error",GetLastError());
123         return(0);
124     }
125 //--- get SlowSMA buffer
126     if(IsStopped()) return(0); //Checking for stop flag
127     if(CopyBuffer(ExtSlowMaHandle,0,0,to_copy,ExtSlowMaBuffer)<=0)
128     {
129         Print("Getting slow SMA is failed! Error",GetLastError());
130         return(0);
131     }
```

And the Moving Average indicator buffer is copied to these arrays based on its handles using the CopyBuffer function.

```
40 //--- indicator buffers mapping
41     SetIndexBuffer(0,ExtMacdBuffer,INDICATOR_DATA);
42     SetIndexBuffer(1,ExtSignalBuffer,INDICATOR_DATA);
43 //    SetIndexBuffer(2,ExtFastMaBuffer,INDICATOR_CALCULATIONS);
44 //    SetIndexBuffer(3,ExtSlowMaBuffer,INDICATOR_CALCULATIONS);
```

⌚ 2018.11.08 11:14:38.678 | MACD (EURUSD,H1) | array out of range in 'MACD.mq5' (139,39)

```
120 if(CopyBuffer(ExtFastMaHandle,0,0,to_copy,ExtFastMaBuffer)<=0)
121 {
122     Print("Getting fast EMA is failed! Error",GetLastError());
123     return(0);
124 }
```

If you remove the binding of the ExtFastMaBuffer and ExtSlowMaBuffer arrays to the indicator buffers, then the client terminal will generate an error. This happens because when the indicator is loaded, the value of the to_copy is equal to the size of the price history, and then the to_copy = 1 and a partial copy is made into the ExtFastMaBuffer and ExtSlowMaBuffer arrays, and their sizes become equal to 1.

In this case, using the ArrayResize function cannot solve the problem, since the CopyBuffer function will still reduce the size of the array to 1.

You can, of course, use another intermediary array in which to copy one element. And already from this intermediary array, you can copy to the intermediate array, but the easiest way, of course, is simply to bind the intermediate array to the indicator buffer.

The OnInit function

As already mentioned, the OnInit, OnDeinit, OnCalculate functions are called by the client terminal when corresponding events occur.

OnInit

The function is called in indicators and EAs when the [Init](#) event occurs. It is used to initialize a running MQL5 program. There are two function types.

The version that returns the result

```
int OnInit(void);
```

Return Value

[int](#) type value, zero means successful initialization.

The `OnInit()` call that returns the execution result is recommended for use since it not only allows for program initialization, but also returns an error code in case of an early program termination.

The version without a result return is left only for compatibility with old codes. It is not recommended for use

```
void OnInit(void);
```

Note

The `Init` event is generated immediately after loading an EA or an indicator. The event is not generated for scripts. The `OnInit()` function is used to initialize an MQL5 program. If `OnInit()` has a return value of [int](#) type, the non-zero return code means failed initialization and generates the [Deinit](#) event with the [REASON_INITFAILED](#) deinitialization reason code.

`OnInit()` function of [void](#) type always means successful initialization and is not recommended for use.

For optimizing the EA [inputs](#), it is recommended to use values from the [ENUM_INIT RETCODE](#) enumeration as a return code. These values are intended for establishing the optimization process management, including selection of the most suitable [test agents](#). It is possible to request data on agent configuration and resources (number of cores, free memory amount, etc.) using the [TerminalInfoInteger\(\)](#) function during the EA initialization before launching the test. Based on the obtained data, you can either allow using the test agent or ban it from optimizing the EA.

The `OnInit` function is called immediately after an indicator is loaded and accordingly it is used for its initialization.

The indicator initialization includes binding arrays to indicator buffers, initializing global variables, including initializing the handlers of the indicators used, as well as programmatically setting the indicator properties.

Let's look at some of these points in more detail.

As already shown, the binding of arrays to indicator buffers is performed using the `SetIndexBuffer` function.

SetIndexBuffer

The function binds a specified indicator buffer with one-dimensional dynamic array of the [double](#) type.

```
bool SetIndexBuffer(  
    int           index,      // buffer index  
    double        buffer[],   // array  
    ENUM_INDEXBUFFER_TYPE data_type // what will be stored  
,
```

Parameters

index

[in] Number of the indicator buffer. The numbering starts with 0. The number must be less than the value declared in [#property indicator_buffers](#).

buffer[]

[in] An array declared in the custom indicator program.

data_type

[in] Type of data stored in the indicator array. By default it is [INDICATOR_DATA](#) (values of the calculated indicator). It may also take the value of [INDICATOR_COLOR_INDEX](#); in this case this buffer is used for storing color indexes for the previous indicator buffer. You can specify up to 64 [colors](#) in the [#property indicator_colorN](#) line. The [INDICATOR_CALCULATIONS](#) value means that the buffer is used in intermediate calculations of the indicator and is not intended for drawing.

Where the `data_type` can be the [INDICATOR_DATA](#) (indicator data for drawing, and by default, it can be omitted), [INDICATOR_COLOR_INDEX](#) (indicator color), [INDICATOR_CALCULATIONS](#) (indicator intermediate calculation buffer).

After applying the `SetIndexBuffer` function to a dynamic array, its size is automatically adjusted equal to the number of bars available to the indicator for calculation.

Each index of an array of the type [INDICATOR_COLOR_INDEX](#) corresponds to the index of an array of the type [INDICATOR_DATA](#), and a value of an index of an array of the type [INDICATOR_COLOR_INDEX](#) determines a color of the display of an array's index of the type [INDICATOR_DATA](#).

An index value of an array of the type [INDICATOR_COLOR_INDEX](#), when it is set, is taken from the [#property indicator_colorN](#) property like a color index in the row.

An index of a buffer of the type [INDICATOR_COLOR_INDEX](#) should follow an index of a buffer of the type [INDICATOR_DATA](#).

After binding a dynamic array to an indicator buffer, you can change the order of access to the array from the end to the beginning, that is a value of an array with the index 0 will correspond to the last obtained indicator value.

This can be done using the `ArraySetAsSeries` function.

ArraySetAsSeries

The function sets the AS_SERIES flag to a selected [object of a dynamic array](#), and elements will be indexed like in [timeseries](#).

```
bool ArraySetAsSeries(
    const void* array[],      // array by reference
    bool flag                // true denotes reverse order of indexing
);
```

Parameters

array[]
[in][out] Numeric array to set.
flag
[in] Array indexing direction.

Return Value

The function returns true on success, otherwise - false.

When using the `ArraySetAsSeries` function, the physical storage of array data does not change, in memory, the array, as before, is stored in order from the first value to the last value.

The `ArraySetAsSeries` function changes only programmatic access to elements of the array — from the last element of the array to the first element of the array.

In the `OnInit` function, the input parameters can also be checked for correctness, since a user can enter anything.

In this case, the value of the input parameter is reassigned using a global variable, and further in the calculations, the value of the global variable is used.

For example, for the `ADX` indicator, it looks like this.

```
47 void OnInit()
48 {
49 //--- check for input parameters
50 if(InpPeriodADX>=100 || InpPeriodADX<=0)
51 {
52     ExtADXPeriod=14;
53     printf("Incorrect value for input variable Period_ADX=%d. Indicator will use value=%d for calculations.", 
54     InpPeriodADX,ExtADXPeriod);
55 }
56 else ExtADXPeriod=InpPeriodADX;
```

Here, the ExtADXPeriod is a global variable, and the InpPeriodADX is an input parameter.

When using indicator handles, you can specify a symbol (financial instrument) for which the indicator will be created.

In this case, such a symbol can be defined by a user.

In the OnInit function, it is also useful to check this input parameter for correctness.

```

35 //-----
36 input string symbol=" "; // Symbol
37 string name=symbol;
38 //+-----+
39
40 //--- delete the left and right spaces
41 StringTrimRight(name);
42 StringTrimLeft(name);
43 //--- if after this the length of the string name is zero
44 if(StringLen(name)==0)
45 {
46     //--- take a symbol from the chart on which the indicator is running
47     name=_Symbol;
48 }
49
50 ExtFastMaHandle=iMA(name,0,InpFastEMA,0,MODE_EMA,InpAppliedPrice);
51 ExtSlowMaHandle=iMA(name,0,InpSlowEMA,0,MODE_EMA,InpAppliedPrice);

```

For example, in the MACD indicator's code, let the input parameter be defined.

input string symbol = " ";

Let's declare a global variable:

string name = symbol;

In the OnInit function, let's check:

Remove the left and right spaces with the StringTrimRight function.

If after this the length of the string name is zero, we take a symbol from the chart on which the indicator is running.

Custom Indicators

This is the group functions used in the creation of custom indicators. These functions can't be used when writing Expert Advisors and Scripts.

Function	Action
SetIndexBuffer	Binds the specified indicator buffer with one-dimensional dynamic array of the double type
IndicatorSetDouble	Sets the value of an indicator property of the double type
IndicatorSetInteger	Sets the value of an indicator property of the int type
IndicatorSetString	Sets the value of an indicator property of the string type
PlotIndexSetDouble	Sets the value of an indicator line property of the type double
PlotIndexSetInteger	Sets the value of an indicator line property of the int type
PlotIndexSetString	Sets the value of an indicator line property of the string type
PlotIndexGetInteger	Returns the value of an indicator line property of the integer type

Programmatic settings properties of an indicator are implemented using the functions `IndicatorSetDouble`, `IndicatorSetInteger`, `IndicatorSetString`, `PlotIndexSetDouble`, `PlotIndexSetInteger`, `PlotIndexSetString`.

IndicatorSetDouble

The function sets the value of the corresponding indicator property. Indicator property must be of the double type. There are two variants of the function.

Call with specifying the property identifier.

```
bool IndicatorSetDouble(  
    int prop_id,           // identifier  
    double prop_value      // value to be set  
)
```

Call with specifying the property identifier and modifier.

```
bool IndicatorSetDouble(  
    int prop_id,           // identifier  
    int prop_modifier,     // modifier  
    double prop_value      // value to be set  
)
```

```
IndicatorSetDouble(INDICATOR_LEVELVALUE, 0, 50);
```

```
#property indicator_level1 50
```

The IndicatorSetDouble function allows you to programmatically define such indicator properties as the indicator_minimum, indicator_maximum, and indicator_levelN, for example:

IndicatorSetDouble (INDICATOR_LEVELVALUE, 0, 50);

is analogous to the:

```
#property indicator_level1 50
```

IndicatorSetInteger

The function sets the value of the corresponding indicator property. Indicator property must be of the int or color type. There are two variants of the function.

Call with specifying the property identifier.

```
bool IndicatorSetInteger(
    int prop_id,           // identifier
    int prop_value         // value to be set
);
```

Call with specifying the property identifier and modifier.

```
bool IndicatorSetInteger(
    int prop_id,           // identifier
    int prop_modifier,     // modifier
    int prop_value         // value to be set
)
```

The IndicatorSetInteger function allows you to programmatically define such indicator properties as the indicator_height, indicator_levelcolor, indicator_levelwidth, and indicator_levelstyle.

At the same time for the levels, it is necessary to define their number using the IndicatorSetInteger function. For example, for the RSI indicator, it looks like this.

```

9 //--- indicator settings
10 #property indicator_separate_window
11 #property indicator_minimum 0
12 #property indicator_maximum 100
13 //#property indicator_level1 30
14 //#property indicator_level2 70
15 #property indicator_buffers 3
16 #property indicator_plots 1
17 #property indicator_type1 DRAW_LINE
18 #property indicator_color1 DodgerBlue
19
20 //#property indicator_levelcolor Red
21 //#property indicator_levelstyle STYLE_SOLID
22 //#property indicator_levelwidth 1
23

57 IndicatorSetInteger(INDICATOR_LEVELS,2);
58 IndicatorSetDouble(INDICATOR_LEVELVALUE,0,30);
59 IndicatorSetDouble(INDICATOR_LEVELVALUE,1,70);
60 IndicatorSetInteger(INDICATOR_LEVELCOLOR,0,0xff0);
61 IndicatorSetInteger(INDICATOR_LEVELCOLOR,1,0xff0);
62 IndicatorSetInteger(INDICATOR_LEVELSTYLE,0,STYLE_SOLID);
63 IndicatorSetInteger(INDICATOR_LEVELSTYLE,1,STYLE_SOLID);
64 IndicatorSetInteger(INDICATOR_LEVELWIDTH,0,1);
65 IndicatorSetInteger(INDICATOR_LEVELWIDTH,1,1);

```

Here, we replace the indicator's properties associated with levels with the code using the `IndicatorSetInteger` function.

The `IndicatorSetInteger` function also allows you to define the accuracy of the indicator, for example:

`IndicatorSetInteger (INDICATOR_DIGITS, 2);`

As a result, only two decimal places of an indicator value will be displayed.

And the `IndicatorSetString` function cannot replace indicator's properties that set as `#property` at the beginning of the indicator's file.

IndicatorSetString

The function sets the value of the corresponding indicator property. Indicator property must be of the string type. There are two variants of the function.

Call with specifying the property identifier.

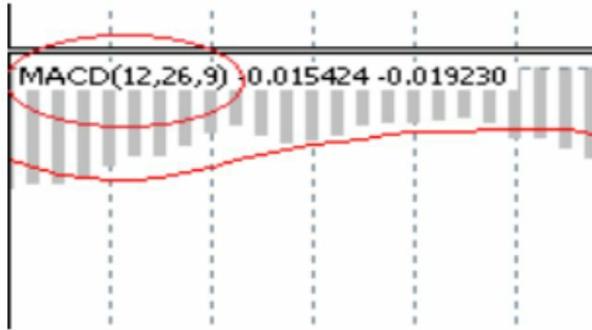
```
bool IndicatorSetString(  
    int prop_id,           // identifier  
    string prop_value      // value to be set  
)
```

Call with specifying the property identifier and modifier.

```
bool IndicatorSetString(  
    int prop_id,           // identifier  
    int prop_modifier,     // modifier  
    string prop_value      // value to be set  
)
```

Anyway, using the IndicatorSetString function, you can define a short indicator's name, for example, for the MACD indicator.

```
50 //--- name for Dindicator subwindow label  
51 IndicatorSetString(INDICATOR_SHORTNAME,"MACD("+string(InpFastEMA)+","+string(InpSlowEMA)+","+string(InpSignalsSMA)+")");
```



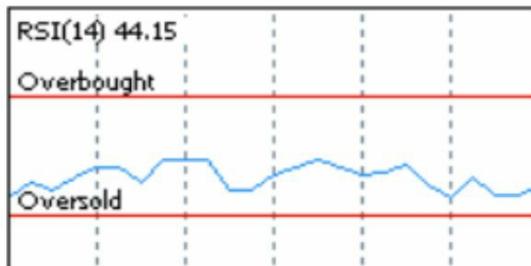
It looks like this.

And accordingly, the indicator's name will be displayed in the indicator's window as:

MACD (12, 26, 9)

In addition, the `IndicatorSetString` function allows you to set labels for indicator's levels, for example, for the RSI indicator.

```
IndicatorSetString(INDICATOR_LEVELTEXT, 0,"Oversold");
IndicatorSetString(INDICATOR_LEVELTEXT, 1,"Overbought");
```



Here, you can display labels for the Oversold and Overbought levels.

PlotIndexSetDouble

The function sets the value of the corresponding property of the corresponding indicator line. The indicator property must be of the double type.

```
bool PlotIndexSetDouble(
    int plot_index,      // plotting style index
    int prop_id,        // property identifier
    double prop_value   // value to be set
);
```

```
PlotIndexSetDouble(plot_index,
PLOT_EMPTY_VALUE, 0);
```

And using the PlotIndexSetDouble function, you can determine which indicator buffer value is empty and does not participate in the drawing of the indicator's diagram.

An indicator's diagram is drawn from one non-empty value to another non-empty value of the indicator's buffer, and empty values are skipped.

To indicate which value should be considered as "empty", it is necessary to define this value in the property PLOT_EMPTY_VALUE. For example, if the indicator should be drawn by non-zero values, then you need to specify a zero value as an empty indicator buffer value:

```
PlotIndexSetDouble (build_index, PLOT_EMPTY_VALUE, 0);
```

PlotIndexSetInteger

The function sets the value of the corresponding property of the corresponding indicator line. The indicator property must be of the int, char, bool or color type. There are 2 variants of the function.

Call indicating identifier of the property.

```
bool PlotIndexSetInteger(
    int plot_index,           // plotting style index
    int prop_id,              // property identifier
    int prop_value            // value to be set
);
```

Call indicating the identifier and modifier of the property.

```
bool PlotIndexSetInteger(
    int plot_index,           // plotting style index
    int prop_id,              // property identifier
    int prop_modifier,        // property modifier
    int prop_value            // value to be set
)
```

The PlotIndexSetInteger function allows you to programmatically, dynamically, set such properties of an indicator's chart as an arrow code for the DRAW_ARROW style, a vertical shift of arrows for the DRAW_ARROW style, a number of initial bars without drawing and values in the Data Window, a type of graphical construction, a sign of display of construction values in the DataWindow, a shift of indicator plotting along the time axis in bars, a drawing line style, the thickness of the drawing line, the number of colors, the index of a buffer containing the drawing color.

Let's analyze each of these properties in order using the example of the Custom Moving Average indicator.

```
9 //--- indicator settings
10 #property indicator_chart_window
11 #property indicator_buffers 1
12 #property indicator_plots 1
13 #property indicator_type1 DRAW_ARROW
14 #property indicator_color1 Red
```

154 PlotIndexSetInteger(0, PLOT_ARROW, 2);



Let's change the indicator_type1 property of the Custom Moving Average indicator as follows.

```
#property indicator_type1 DRAW_ARROW
```

In the OnInit function, let's add a call to the PlotIndexSetInteger function, defining different arrow code for the DRAW_ARROW style.

```
PlotIndexSetInteger (0, PLOT_ARROW, 2);
```

As a result, we get the following type of arrow.

```
PlotIndexSetInteger(0,PLOT_ARROW,3);
```



```
PlotIndexSetInteger(0,PLOT_ARROW,4);
```



```
PlotIndexSetInteger(0,PLOT_ARROW,5);
```



```
PlotIndexSetInteger(0,PLOT_ARROW,6);
```



With the value 3 of the property PLOT_ARROW, we get another arrow.
And so on, changing the value of the property, we will see different arrows.

```
154 PlotIndexSetInteger(0,PLOT_ARROW,7);  
155 PlotIndexSetInteger(0,PLOT_ARROW_SHIFT,100);|
```



In the OnInit function, let's add a call to the PlotIndexSetInteger function, defining the vertical shift of the arrows for the DRAW_ARROW style:
PlotIndexSetInteger (0, PLOT_ARROW_SHIFT, 100);
As a result, the indicator's chart has moved down.

```
131 void OnInit()
132 {
133 //--- indicator buffers mapping
134     SetIndexBuffer(0,ExtLineBuffer,INDICATOR_DATA);
135 //--- set accuracy
136     IndicatorSetInteger(INDICATOR_DIGITS, _Digits+1);
137 //--- sets first bar from what index will be drawn
138     PlotIndexSetInteger(0,PLOT_DRAW_BEGIN,InpMAPeriod);
139 //---- line shifts when drawing
140     PlotIndexSetInteger(0,PLOT_SHIFT,InpMASHift);
```

In the Custom Moving Average indicator, to define a number of initial bars without drawing, let's use the PlotIndexSetInteger function call:
PlotIndexSetInteger (0, PLOT_DRAW_BEGIN, InpMAPeriod);
where InpMAPeriod is a period of the moving average.

```
9 //--- indicator settings
10 #property indicator_chart_window
11 #property indicator_buffers 1
12 #property indicator_plots 1
13 //#property indicator_type1 DRAW_ARROW
14 #property indicator_color1 Red

131 void OnInit()
132 {
133     PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_ARROW);
```

The property identifier PLOT_DRAW_TYPE of the PlotIndexSetInteger function allows you to programmatically set the indicator_typeN property, for example:

PlotIndexSetInteger (0, PLOT_DRAW_TYPE, DRAW_ARROW);

Moreover, if the indicator_typeN property is set and at the same time a call to the PlotIndexSetInteger function with the identifier PLOT_DRAW_TYPE is made, the chart type will act specified by the PlotIndexSetInteger function.

```

141 //---- line shifts when drawing
142 PlotIndexSetInteger(0, PLOT_SHIFT, InpMAShift);
143
144 PlotIndexSetInteger(0, PLOT_SHOW_DATA, false);

```

Data Window	
Date	2018.11.08
Time	03:00
Open	1.14324
High	1.14344
Low	1.14299
Close	1.14323
Volume	0
Tick Volume	1760
Spread	11

InpMAShift=0



InpMAShift=10



You can remove a display of current values of an indicator's chart, when you hover the mouse cursor in the Data Window, by calling the PlotIndexSetInteger function with the identifier PLOT_SHOW_DATA.

PlotIndexSetInteger (0, PLOT_SHOW_DATA, false);

In the Custom Moving Average indicator, the PlotIndexSetInteger function is used to define a shift of indicator plotting along the time axis in bars:

PlotIndexSetInteger (0, PLOT_SHIFT, InpMAShift);

For example, when InpMAShift = 10, the indicator's shift along the time axis is visible here.

Such a shift is made to simulate the predictability of the indicator.

```
PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_LINE);
PlotIndexSetInteger(0, PLOT_LINE_STYLE, STYLE_DASHDOT);
PlotIndexSetInteger(0, PLOT_LINE_WIDTH, 2);
```



The PlotIndexSetInteger function identifier PLOT_LINE_STYLE allows you to programmatically set the indicator_styleN property, drawing line style, for example:

```
PlotIndexSetInteger (0, PLOT_LINE_STYLE, STYLE_DASHDOT);
```

The identifier PLOT_LINE_WIDTH of the PlotIndexSetInteger function allows you to programmatically set the indicator_widthN property, the thickness of the drawing line, for example:

```
PlotIndexSetInteger (0, PLOT_LINE_WIDTH, 2);
```

```

10 //--- indicator settings
11 #property indicator_separate_window
12 #property indicator_buffers 4
13 #property indicator_plots 2
14 #property indicator_type1 DRAW_HISTOGRAM
15 #property indicator_type2 DRAW_LINE
16 //#property indicator_color1 Silver
17 //#property indicator_color2 Red
18 #property indicator_width1 2
19 #property indicator_width2 1
20 #property indicator_label1 "MACD"
21 #property indicator_label2 "Signal"

41 void OnInit()
42 {
43 PlotIndexSetInteger(0,PLOT_COLOR_INDEXES,1);
44 PlotIndexSetInteger(0,PLOT_LINE_COLOR,0,Silver);
45 PlotIndexSetInteger(1,PLOT_COLOR_INDEXES,1);
46 PlotIndexSetInteger(1,PLOT_LINE_COLOR,0,Red);

```

A call of the `PlotIndexSetInteger` function with the identifiers `PLOT_COLOR_INDEXES` and `PLOT_LINE_COLOR` allows you to programmatically set the `indicator_colorN` property, for example, in the case of the MACD indicator:

```
#property indicator_color1 Silver
```

This is equivalent to

```
PlotIndexSetInteger (0, PLOT_COLOR_INDEXES, 1);
```

```
PlotIndexSetInteger (0, PLOT_LINE_COLOR, 0, Silver);
```

```
#property indicator_label1 "MACD"  
#property indicator_label2 "Signal"
```

```
PlotIndexSetString(0, PLOT_LABEL,  
"MACD");  
PlotIndexSetString(1, PLOT_LABEL,  
"Signal");
```

The `PlotIndexSetString` function allows you to programmatically set the `indicator_labelN` property. For example, for the MACD indicator, it will look like this:

```
#property indicator_label1 "MACD"  
#property indicator_label2 "Signal"
```

This is equivalent to

```
PlotIndexSetString (0, PLOT_LABEL, "MACD");  
PlotIndexSetString (1, PLOT_LABEL, "Signal");
```

The above functions of the programmatically setting the indicator's properties can be, of course, called in the `OnCalculate` callback function, but there is no deep meaning in this since they cannot be applied only to part of the indicator's diagram — they are applied directly to the entire indicator's diagram. Therefore, to save resources, it is best to call these functions in the `OnInit` callback function.

The OnDeinit function

Firstly, let's quote a reference:

The Deinit event is generated for Expert Advisors and indicators in the following cases:

- before reinitialization due to the change of a symbol or chart period, to which the mql5 program is attached;
- before reinitialization due to the change of input parameters;
- before unloading the mql5 program.

OnDeinit

The function is called in indicators and EAs when the [Deinit](#) event occurs. It is used to deinitialize a running MQL5 program.

```
int OnDeinit(
    const int reason          // deinitialization reason code
);
```

Parameters

`reason`
[in] Deinitialization reason code.

Return Value

No return value

Since the OnDeinit function is called during deinitialization, its main purpose is to release occupied resources.

The release of occupied resources for the indicator means cleaning a symbol's chart from additional graphical objects.

That is, in addition to the indicator's diagram, we can attach various objects to the chart of the symbol - lines, graphic figures - a triangle, a rectangle, and an ellipse, signs, labels, etc.

And we will discuss this later.

Accordingly, when the indicator is deinitialized, it would be nice to remove all these objects from the symbol's chart.

```
void Comment(  
    argument, // first value  
    ...       // next values  
);  
  
Comment("");  
  
bool ObjectDelete(  
    long chart_id, // chart identifier  
    string name    // object name  
);  
  
ObjectDelete(0,label_name);
```

Firstly, here, you can use the `Comment` function, which displays a user-defined comment in the upper left corner of the chart.

And empty comments are used to clear comments.

Next, you can use the `ObjectDelete` function that deletes an object with the specified name from the specified chart.

And later we will demonstrate the use of these functions.

The OnCalculate function

The OnCalculate function is called by the client terminal when a new tick is received by the symbol for which the indicator is calculated.

OnCalculate

The function is called in the indicators when the [Calculate](#) event occurs for processing price data changes. There are two function types. Only one of them can be used within a single indicator.

Calculation based on data array

```
int OnCalculate(
    const int      rates_total,           // price[] array size
    const int      prev_calculated,       // number of handled bars at the previous call
    const int      begin,                // index number in the price[] array meaningful data starts from
    const doubles   price[]);           // array of values for calculation
```

Calculations based on the current timeframe timeseries

```
int OnCalculate(
    const int      rates_total,           // size of input time series
    const int      prev_calculated,       // number of handled bars at the previous call
    const datetimes time[],              // Time array
    const doubles   open[],               // Open array
    const doubles   high[],               // High array
    const doubles   low[],                // Low array
    const doubles   close[],              // Close array
    const longs     tick_volume[],        // Tick Volume array
    const longs     volume[],             // Real Volume array
    const int4      spread[]);           // Spread array
```

Although the OnCalculate function has two versions - for an indicator that can be calculated based on only one of the price series and for an indicator that is calculated using several price time series.

And here, we will use the full version of the OnCalculate function as the most flexible and offers the most features.

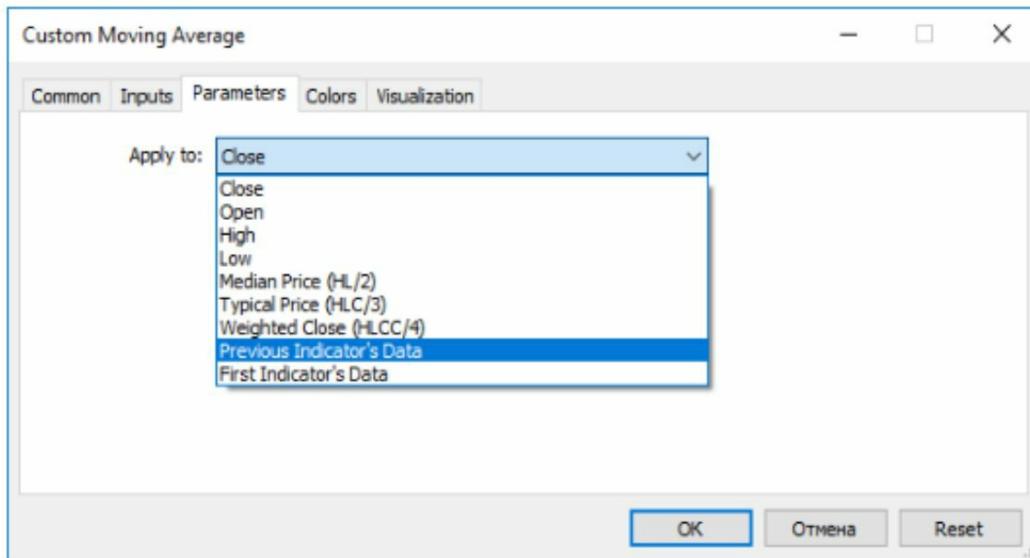
The only thing that we should note about the truncated OnCalculate function is that it has an option of using a calculated buffer of another indicator as to the price array.

Let's demonstrate this with the example of the MACD indicator and the Custom Moving Average indicator, which uses the truncated OnCalculate function.



Firstly, let's add the MACD indicator to the symbol's chart, and then let's drag the MA indicator from the Indicators - Trending folder into the MACD indicator window.

Then let's drag the Custom Moving Average indicator to the MACD indicator window, and as a result, it will open the Custom Moving Average indicator parameters window.



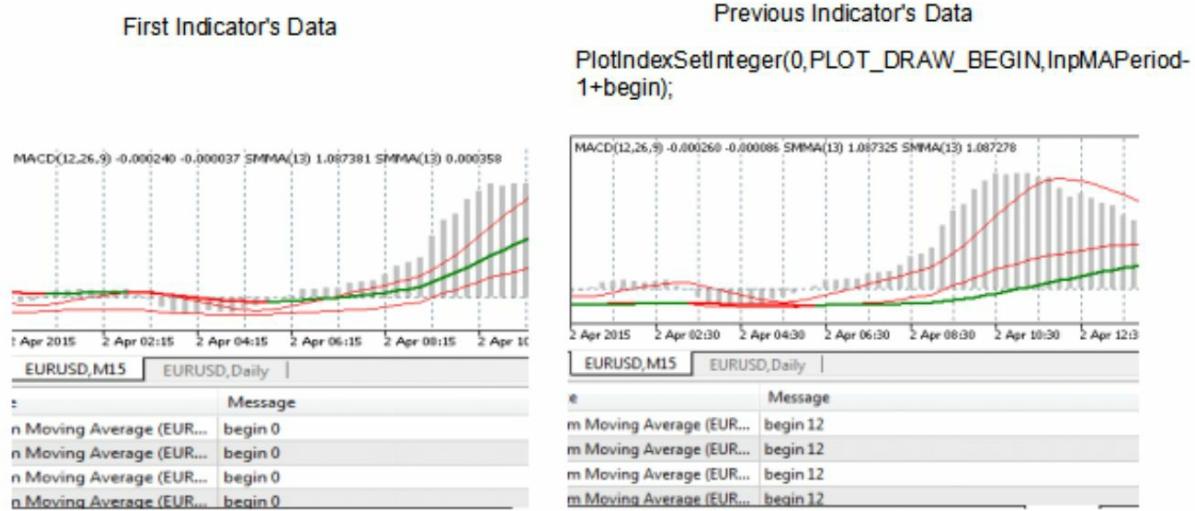
In the selection list, which you can use to select an item as the price array, you can see:

Previous indicator's data

First indicator's data

Here, the item "first indicator's data" means that the ExtMacdBuffer array of the MACD indicator's buffer will be used as the price array, and the item "previous indicator's data" means that the ExtLineBuffer array of the MA indicator's buffer will be used as the price array.

```
Print("begin ", begin);
```



If we add the call to the OnCalculate function of the Custom Moving Average indicator:

```
Print ("begin", begin);
```

Then, when you select the first indicator's data, it will be displayed:

Begin 0

And when selecting the previous indicator's data, it will be displayed:

Begin 12

In the first case, the begin = 0, since for the ExtMacdBuffer buffer of the MACD indicator, the PlotIndexSetInteger function with the PLOT_DRAW_BEGIN parameter is not called.

And in the second case, the begin = 12, since the PlotIndexSetInteger function is called for the MA ExtLineBuffer buffer of the MA indicator:

It says that the array of the ExtLineBuffer buffer of the MA indicator is filled starting from the InpMAPeriod-1 bar, respectively, the variable begin of the OnCalculate function of the Custom Moving Average indicator will also be InpMAPeriod-1.

Calculations based on the current timeframe timeseries

```
int OnCalculate(
    const int      rates_total,           // size of input time series
    const int      prev_calculated,       // number of handled bars at the previous call
    const datetime& time[],              // Time array
    const double&  open[],               // Open array
    const double&  high[],               // High array
    const double&  low[],                // Low array
    const double&  close[],              // Close array
    const long&    tick_volume[],        // Tick Volume array
    const long&    volume[],             // Real Volume array
    const int&     spread[]              // Spread array
);
```

Let's return to the full version of the OnCalculate function.

As a rule, the OnCalculate function code is designed so that when the indicator is loaded and the first time the OnCalculate function is called, the indicator's buffers are calculated based on the entire loaded price history.

And at the subsequent receipt of a new tick and calling the OnCalculate function, only one new value would be calculated, which is added to the end of the indicator buffer array.

But at the beginning of the OnCalculate function code, of course, you need to check whether the size of the price history was sufficiently got when the indicator was loaded.

To do this, you could check the value of the variable `rates_total` - the size of the input time series.

ADX

```
80 //-----+
81 //| Custom indicator iteration function |
82 //+-----+
83 int OnCalculate(const int rates_total,
84                 const int prev_calculated,
85                 const datetime &time[],
86                 const double &open[],
87                 const double &high[],
88                 const double &low[],
89                 const double &close[],
90                 const long &tick_volume[],
91                 const long &volume[],
92                 const int &spread[])
93 {
94 //--- checking for bars count
95 if(rates_total<ExtADXPeriod)
96     return(0);
```

As a rule, the value of the indicator period is taken as a threshold value for the rates_total, for example, for the ADX indicator:

rates_total <ExtADXPeriod

```
//--- checking the number of calculated values in the indicator  
  
int calculated=BarsCalculated(handle);  
  
    if(calculated<=0)  
    {  
        PrintFormat("BarsCalculated() return %d, error code  
%d",calculated,GetLastError());  
  
        return(0);  
    }
```

If in the calculation of the indicator's buffer, a handle of another indicator is involved, then the amount of calculated data for the used indicator is checked.

Using the BarsCalculated function, which takes an indicator handle as an argument, and after checking the initially loaded history for calculations, the size of the data is calculated that needs to be calculated in this call of the OnCalculate function.

As an example, let's analyze the OnCalculate function code block, which is given in the MQL5 reference, in the Technical Indicators section.

```

{
//--- number of values copied from the iMA indicator
int values_to_copy;
//--- determine the number of values calculated in the indicator
int calculated=BarsCalculated(handle);
if(calculated<=0)
{
    PrintFormat("BarsCalculated() returned %d, error code %d",calculated,GetLastError());
    return(0);
}
//--- if it is the first start of calculation of the indicator or if the number of values in the iMA indicator changed
//--- or if it is necessary to calculate the indicator for two or more bars (it means something has changed in the price history)
if(prev_calculated==0 || calculated!=bars_calculated || rates_total>prev_calculated+1)
{
    //--- if the iMABuffer array is greater than the number of values in the iMA indicator for symbol/period, then we don't copy everything
    //--- otherwise, we copy less than the size of indicator buffers
    if(calculated>rates_total) values_to_copy=rates_total;
    else                  values_to_copy=calculated;
}
else
{
    //--- it means that it's not the first time of the indicator calculation, and since the last call of OnCalculate()
    //--- for calculation not more than one bar is added
    values_to_copy=(rates_total-prev_calculated)+1;
}

//--- memorize the number of values in the Moving Average indicator
bars_calculated=calculated;
//--- return the prev_calculated value for the next call
return(rates_total);
}

```

Here, the variable `values_to_copy` is a number of calculated values in the call of the `OnCalculate` function.

The variable `prev_calculated` is how many bars were processed by the `OnCalculate` function during the previous call.

Thus, when loading an indicator, the `prev_calculated = 0`, and each time when a new tick arrives, the `prev_calculated = rates_total`.

The `prev_calculated` variable is also reset by the terminal if the value of the `rates_total` variable has suddenly changed.

The variable `bars_calculated` is a previous amount of calculated data for the used indicator, on the basis of which this indicator is calculated.

So the first check here is:

This is the `prev_calculated == 0` - that means the indicator has just loaded or the price history has changed.

The `calculated! = bars_calculated` - that means the amount of calculated data for the used indicator has changed.

The `rates_total > prev_calculated + 1` - it is necessary to calculate the indicator for two or more bars (this means that something has changed in history).

The last condition conflicts with the statement of the MQL5 reference:

If since the last call of OnCalculate price data has changed (a deeper history downloaded or history blanks filled), the value of the input parameter prev_calculated will be set to zero by the terminal.

If the history has changed, then the check `prev_calculated == 0` will work and the last condition check will be superfluous.

Now, if the first or second condition is triggered, then the number of calculated values is the size of the input time series or the amount of calculated data for the used indicator, on the basis of which this indicator is calculated.

And it will be one or the other depending on which of them will be less.

If the first or second condition does not work, then the number of calculated values

```
values_to_copy = (rates_total-prev_calculated) +1;
```

Again, there is an unnecessary code, since, judging from the MQL5 reference, the variable `prev_calculated` can take the value either 0 or `rates_total`.

Therefore, `values_to_copy = 1`.

Thus, when a new tick arrives, only one indicator's value for this new tick will be calculated.

Let's consider another implementation to calculate the size of the data that needs to be calculated in a call of the OnCalculate function.

```
109 //--- we can copy not all data
110 int to_copy;
111 if(prev_calculated>rates_total || prev_calculated<0) to_copy=rates_total;
112 else
113 {
114     to_copy=rates_total-prev_calculated;
115     if(prev_calculated>0) to_copy++;
116 }
```

For the MACD indicator, this is implemented as follows:

Again, judging by the MQL5 reference, only the code works here:

```
to_copy = rates_total-prev_calculated;
if (prev_calculated> 0) to_copy ++;
```

That is, when the indicator is loaded, then the `to_copy = rates_total`, and then `to_copy = 1`.

After calculating the size of the data to be calculated in a call of the `OnCalculate` function, the data are calculated and fill indicator's buffers.

If the indicator is calculated on the basis of a handle of another indicator, then the code copies from the buffers of the used indicator into the dynamic arrays of this indicator.

- [iAC](#)
- [iAD](#)
- **iADX**
- [iADXWilder](#)
- [iAlligator](#)
- [iAMA](#)
- [IAO](#)

iADX

The function returns the handle of the Average Directional Movement Index indicator.

```
int iADX(
    string          symbol,           // symbol name
    ENUM_TIMEFRAMES period,          // period
    int             adx_period       // averaging period
);
```

Here is how it is implemented in the MQL5 reference, in the example for the iADX function, where the ADX indicator is used.

```

//--- fill a part of the iADXBuffer array with values from the indicator buffer that has 0 index
if(CopyBuffer(ind_handle,0,0,amount,adx_values)<0)
{
    //--- if the copying fails, tell the error code
    PrintFormat("Failed to copy data from the iADX indicator, error code %d",GetLastError());
    //--- quit with zero result - it means that the indicator is considered as not calculated
    return(false);
}

//--- fill a part of the DI_plusBuffer array with values from the indicator buffer that has index 1
if(CopyBuffer(ind_handle,1,0,amount,DIplus_values)<0)
{
    //--- if the copying fails, tell the error code
    PrintFormat("Failed to copy data from the iADX indicator, error code %d",GetLastError());
    //--- quit with zero result - it means that the indicator is considered as not calculated
    return(false);
}

//--- fill a part of the DI_minusBuffer array with values from the indicator buffer that has index 2
if(CopyBuffer(ind_handle,2,0,amount,DIminus_values)<0)
{
    //--- if the copying fails, tell the error code
    PrintFormat("Failed to copy data from the iADX indicator, error code %d",GetLastError());
    //--- quit with zero result - it means that the indicator is considered as not calculated
    return(false);
}

```

Here, the `ind_handle` is the ADX indicator's handle, the second parameter is the buffer index of the indicator being used, from which copying is made, the third parameter is the starting position from which the copying starts.

Here, we remember that copying goes from the end to the beginning, and therefore the zero starting position is the latest data.

The fourth parameter is our data size, which needs to be calculated in a call of the `OnCalculate` function, and the last parameter is usually a dynamic array bound to the indicator's buffer where the copy is being made.

There is a question - how to bind the second parameter of the `CopyBuffer` function with the buffer index of the indicator used.

This is done by calling the `SetIndexBuffer` function in the indicator being used.

```
SetIndexBuffer(0,ExtADXBuffer);
SetIndexBuffer(1,ExtPDIBuffer);
SetIndexBuffer(2,ExtNDIBuffer);
```

For example, for the ADX indicator.

Here, the index 0 is associated with the buffer of the ADX indicator itself, the index 1 is associated with the directional index buffer + DI, the index 2 is associated with the directional index buffer -DI.

Thus, to bind the second parameter of the CopyBuffer function with the buffer index of the indicator used, you need to know the code of the indicator used.

```
for(int i=start; i<rates_total && !IsStopped(); i++)
{
    ...
}
```

Also, to fill the indicator buffer with values, a loop can be used, for example, the for loop.

As can be seen in the code of the indicators in the Examples folder of the MetaEditor.

Here, the start is the starting position from which the filling of the indicator's buffer begins.

If the prev_calculated = 0, the start is usually 0, and if the prev_calculated = rates_total, the start = prev_calculated-1.

If, before implementing the loop, to change the order of access to the indicator's buffer arrays and price arrays using the `ArraySetAsSeries` function, then the loop takes the following form.

```
for(int i=start; i>=0; i--){
    ...
}
```

Where the start = rates_total-1, if the prev_calculated = 0, and the start = 0, if the prev_calculated = rates_total.

Example of Creating Indicator

As an example, let's consider the creation of an indicator that will implement the Forex strategy "Impulse keeper" and show buy and sell signals on a chart.



This strategy uses four indicators:

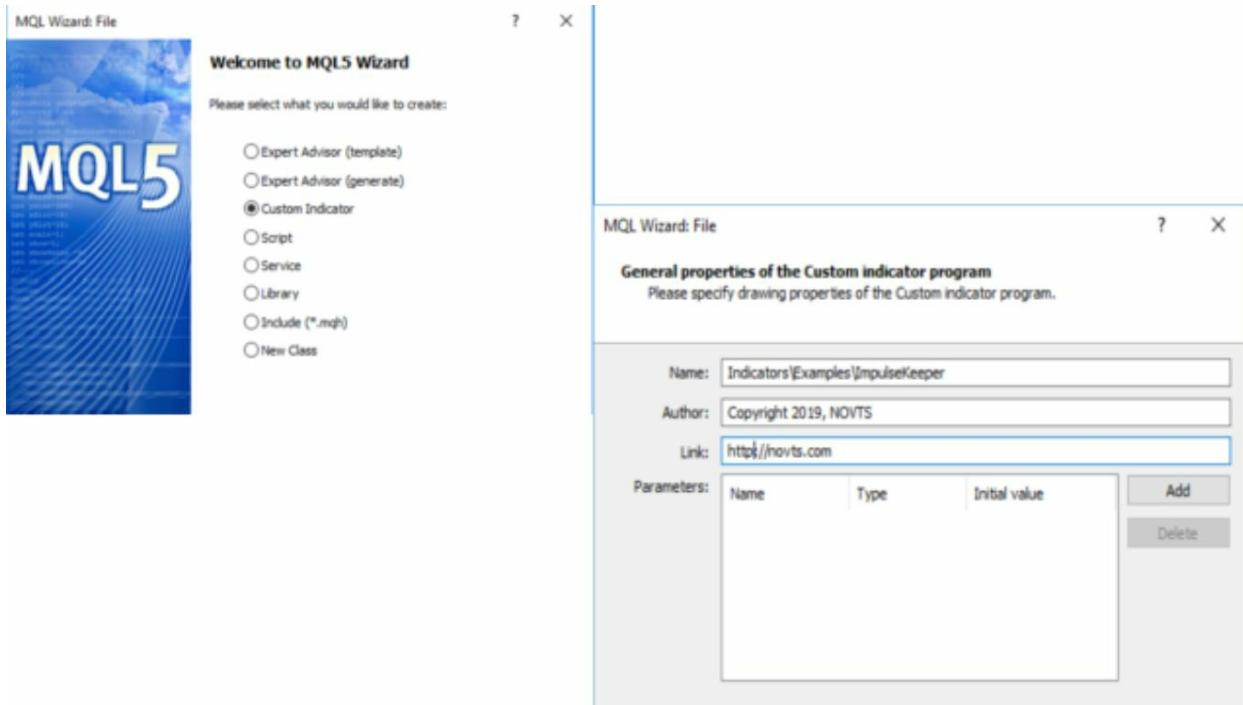
- Exponential moving average with period 34 for the High price.
- Exponential moving average with period 34 for the Low price.
- Exponential moving average with the period 125 for the Close price.
- Parabolic SAR.

The buy and sell signals in this strategy are described as follows.

Buy signal: a green candle closes above EMA34 High and EMA34 Low, and a green candle is above EMA125 and Parabolic SAR.

Sell signal: a red candle closes below EMA34 Low and EMA34 High, and a red candle is below EMA125 and Parabolic SAR.

Let's implement this strategy in the code that will display up and down arrows on a chart for buy and sell signals.



Let's open the MQL5 editor and select the New in the File menu.

In the MQL Wizard dialog box, let's select the Custom Indicator and click the Next button.

Let's enter a name of the indicator "Impulse keeper", a name of the author and a site link and click the Next twice and then the Finish.

```

6 #property copyright "Copyright 2018, NOVTS"
7 #property link      "http://novts.com"
8 #property version   "1.00"
9 #property indicator_chart_window
10 //+
11 //| Custom indicator initialization function
12 //+
13 int OnInit()
14 {
15 //--- indicator buffers mapping
16
17 //---
18     return(INIT_SUCCEEDED);
19 }
20 //+
21 //| Custom indicator iteration function
22 //+
23 int OnCalculate(const int rates_total,
24                 const int prev_calculated,
25                 const datetime &time[],
26                 const double &open[],
27                 const double &high[],
28                 const double &low[],
29                 const double &close[],
30                 const long &tick_volume[],
31                 const long &volume[],
32                 const int &spread[])
33 {
34 //---
35
36 //--- return value of prev_calculated for next call
37     return(rates_total);
38 }
39 //+

```

As a result, we get the indicator code with the empty OnInit and OnCalculate functions.

The creation of our indicator will begin with the definition of its properties.

```

6 #property copyright "Copyright 2018, NOVTS"
7 #property link      "http://novts.com"
8 #property version   "1.00"
9 #property indicator_chart_window
10
11 #property indicator_buffers 8
12 #property indicator_plots 2
13 #property indicator_color1 clrGreen, clrBlack
14 #property indicator_type1 DRAW_COLOR_ARROW
15 #property indicator_color2 clrRed, clrBlack
16 #property indicator_type2 DRAW_COLOR_ARROW
17
18 double IKBuyBuffer[];
19 double ColorIKBuyBuffer[];
20 double IKSellBuffer[];
21 double ColorIKSellBuffer[];
22 double EMA34HBuffer[];
23 double EMA34LBuffer[];
24 double EMA125Buffer[];
25 double PSARBuffer[];
26
27 int EMA34HHandle;
28 int EMA34LHandle;
29 int EMA125Handle;
30 int PSARHandle;

```

A number of indicator's buffers is 8.

Two buffers - data and color for buy signals.

Two buffers - data and color for sell signals.

And for intermediate calculated buffers for copied data from the EMA34 Low, EMA34 High, EMA125 and Parabolic SAR indicators:

#property indicator_buffers 8

Let's define a number of graphical constructions as the two plots, one construction for buy signals and another construction for sale signals the indicator_plots property.

And we define a color and a type for both graphical constructions with the indicator_color and indicator_type properties.

Next, we define the indicator buffer arrays and the handles of the indicators used.

```

35 int OnInit()
36 {
37 PlotIndexSetInteger(0,PLOT_ARROW,233);
38 PlotIndexSetDouble(0,PLOT_EMPTY_VALUE,0);
39 PlotIndexSetInteger(0,PLOT_ARROW_SHIFT,-10);
40
41 PlotIndexSetInteger(1,PLOT_ARROW,234);
42 PlotIndexSetDouble(1,PLOT_EMPTY_VALUE,0);
43 PlotIndexSetInteger(1,PLOT_ARROW_SHIFT,10);
44 //--- indicator buffers mapping
45 SetIndexBuffer(0,iKBuyBuffer,INDICATOR_DATA);
46 SetIndexBuffer(1,ColorIKBuyBuffer,INDICATOR_COLOR_INDEX);
47
48 SetIndexBuffer(2,iKSellBuffer,INDICATOR_DATA);
49 SetIndexBuffer(3,ColorIKSellBuffer,INDICATOR_COLOR_INDEX);
50
51 SetIndexBuffer(4,EMA34HBuffer,INDICATOR_CALCULATIONS);
52 SetIndexBuffer(5,EMA34LBuffer,INDICATOR_CALCULATIONS);
53 SetIndexBuffer(6,EMA125Buffer,INDICATOR_CALCULATIONS);
54 SetIndexBuffer(7,PSARBuffer,INDICATOR_CALCULATIONS);
55
56 EMA34HHandle=iMA(NULL,0,34,0,MODE_EMA,PRICE_HIGH);
57 EMA34LHandle=iMA(NULL,0,34,0,MODE_EMA,PRICE_LOW);
58 EMA125Handle=iMA(NULL,0,125,0,MODE_EMA,PRICE_CLOSE);
59 PSARHandle=iSAR(NULL,0,0.02, 0.2);
60
61 //--- end init
62     return(INIT_SUCCEEDED);
63 }
```

In the OnInit function, for the first graphical construction with the index 0, we define a type of the arrow - the up arrow, the blank value and the shift using the functions PlotIndexSetInteger and PlotIndexSetDouble.

For the second graphical construction with index 1, we define a type of the arrow - the down arrow, the blank value, and the shift.

And we associate arrays with the indicator's buffers using the SetIndexBuffer function.

And then we get the handles of the indicators used, using the standard functions of the technical indicators - iMA and iSAR.

```

32 int bars_calculated=0;
33

78 {
79 //---
80     int values_to_copy;
81     int start;
82     int calculated=BarsCalculated(EMAS4Handle);
83     if(calculated<=0)
84     {
85         return(0);
86     }
87     if(prev_calculated==0 || calculated!=bars_calculated)
88     {
89         start=1;
90         if(calculated>rates_total) values_to_copy=rates_total;
91         else                                values_to_copy=calculated;
92     }
93     else
94     {
95         start=rates_total-1;
96         values_to_copy=1;
97     }
98
99 bars_calculated=calculated;
100 //--- return value of prev_calculated for next call
101     return(rates_total);
102 }
103 //+-----+

```

In the OnCalculate function, we check a size of the available history for calculating the indicators used, determine the number of copied values of the used indicators values_to_copy and determine the starting position for calculating the indicator.

And we define the bars_calculated variable as the global variable int bars_calculated = 0; in the properties of the indicator.

```
99 if(!FillArrayFromMABuffer(EMA34HBuffer,0,EMA34HHandle,values_to_copy)) return(0);
100
101 if(!FillArrayFromMABuffer(EMA34LBuffer,0,EMA34LHandle,values_to_copy)) return(0);
102
103 if(!FillArrayFromMABuffer(EMA125Buffer,0,EMA125Handle,values_to_copy)) return(0);
104
105 if(!FillArrayFromPSARBuffer(PSARBuffer,PSARHandle,values_to_copy)) return(0);
```

Next, we copy from the buffers of the used indicators into the arrays of the buffers of our indicator.

Here, the FillArrayFromMABuffer and FillArrayFromPSARBuffer are the custom functions defined outside the OnCalculate function.

```

137 //+
138 bool FillArrayFromPSARBuffer(
139 double & sar_buffer[], // Parabolic SAR indicator's value buffer
140 int ind_handle, // iSAR indicator's handle
141 int amount // number of values to be copied
142 )
143 {
144     ResetLastError();
145     if(CopyBuffer(ind_handle,0,0,amount,sar_buffer)<0)
146     {
147         return(false);
148     }
149     return(true);
150 }
151 //+
152 bool FillArrayFromMABuffer(
153 double & values[], // indicator's buffer of Moving Average values
154 int shift, // shift
155 int ind_handle, // iMA indicator's handle
156 int amount // number of values to be copied
157 )
158 {
159     ResetLastError();
160     if(CopyBuffer(ind_handle,0,-shift,amount,values)<0)
161     {
162         return(false);
163     }
164     return(true);
165 }

```

The `FillArrayFromPSARBuffer` function is responsible for copying the Parabolic SAR indicator's data to the specified array using the `CopyBuffer` function.

And the `FillArrayFromMABuffer` function is responsible for copying the data of the Moving Average indicator to the specified array.

```

107 for(int i=start;i<rates_total && !IsStopped();i++)
108 {
109     IKBuyBuffer[i-1]=0;
110     ColorIKBuyBuffer[i-1]=1;
111
112     IKSellBuffer[i-1]=0;
113     ColorIKSellBuffer[i-1]=1;
114
115 if(close[i-1]>open[i-1]&&close[i-1]>EMA34HBuffer[i-1]&&close[i-1]>EMA34LBuffer[i-1]
116 &&low[i-1]>EMA125Buffer[i-1]&&low[i-1]>PSARBuffer[i-1]&&EMA125Buffer[i-1]<EMA34LBuffer[i-1]
117 &&EMA125Buffer[i-1]<EMA34HBuffer[i-1]){
118     IKBuyBuffer[i-1]=high[i-1];
119     ColorIKBuyBuffer[i-1]=0;
120 }
121
122 if(close[i-1]<open[i-1]&&close[i-1]<EMA34HBuffer[i-1]&&close[i-1]<EMA34LBuffer[i-1]
123 &&high[i-1]<EMA125Buffer[i-1]&&high[i-1]<PSARBuffer[i-1]&&EMA125Buffer[i-1]>EMA34LBuffer[i-1]
124 &&EMA125Buffer[i-1]>EMA34HBuffer[i-1]){
125     IKSellBuffer[i-1]=low[i-1];
126     ColorIKSellBuffer[i-1]=0;
127 }
128 }
```

Next, in the OnCalculate function, let's fill the indicator's buffers with data and color.

Here, we calculate the indicator on the previous bar, since, at the current bar, the Close price is the current price of a tick.

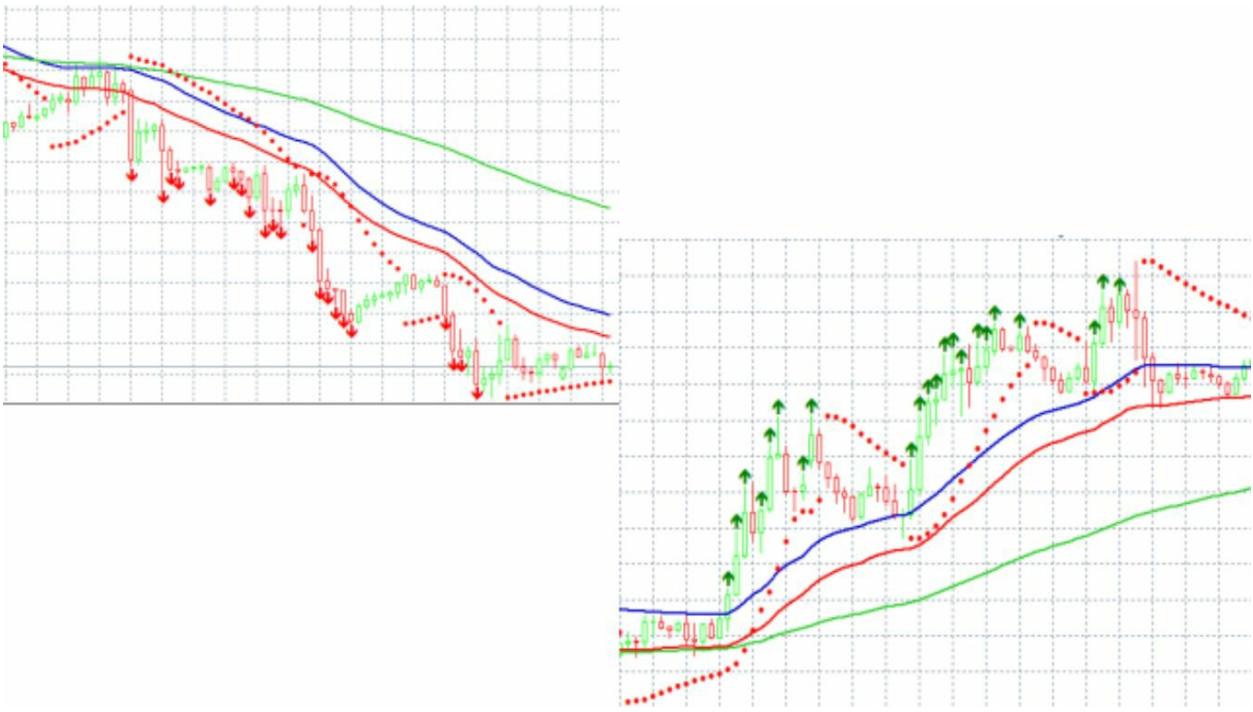
And here, in the loop for, passing through the price history, we first set a value of our indicator to the 0 and indicator's color as the black color.

At the same time, since using the PlotIndexSetDouble function, we set the indicator's zero value as a value that will not be drawn, the black color indicator' values will not be displayed.

Then we check if the conditions are in line with our strategy for buying or selling a financial instrument.

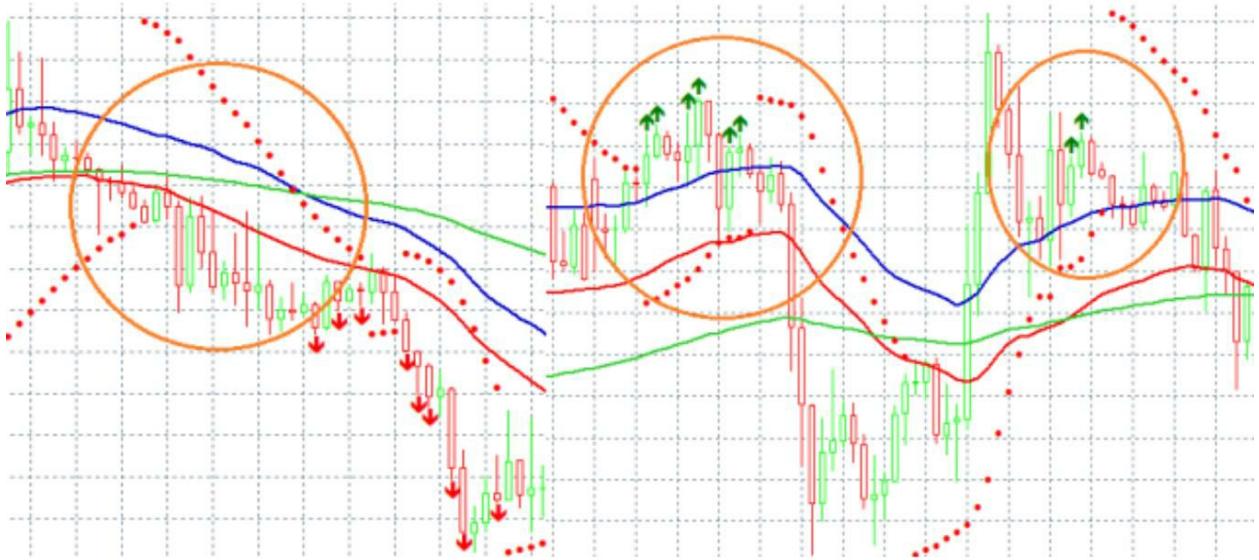
And if the conditions correspond to the purchase, then we set the indicator value as the high price of the bar, and its color as the green color.

If the conditions correspond to the sale, then we set the indicator value as the low price of the bar, and its color as the red color.



Let's compile the code and attach the indicator to a symbol's chart.

And we see that, in general, the indicator gives the right signals to sell and buy, although in some cases it is late and gives false signals.

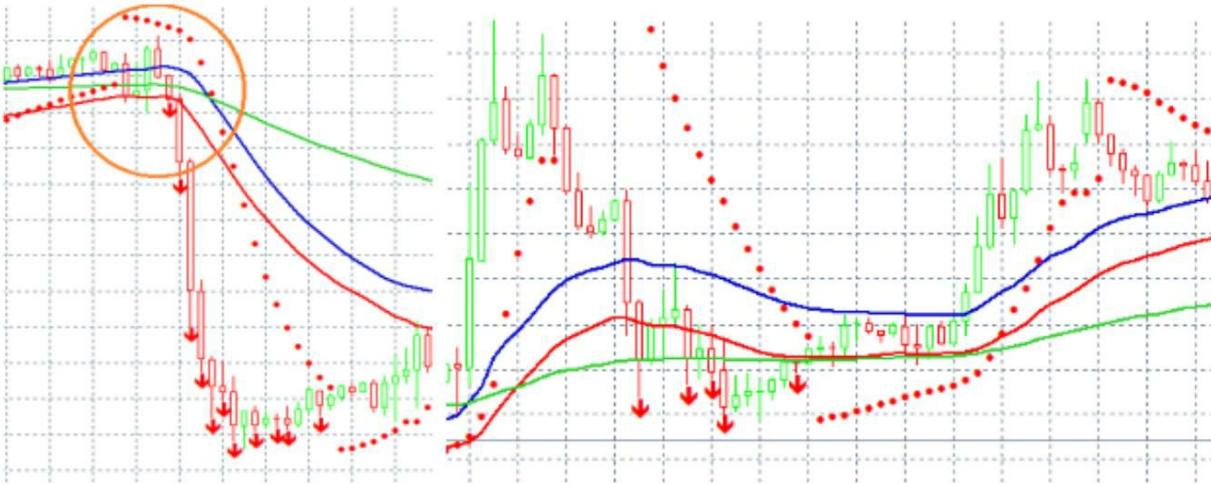


As we can see, this is due to the trend line EMA125.

Let's try to untie it from the current period and let's try to determine the trend, say, on the daily chart.

```

57 EMA34HHandle=iMA(NULL,0,34,0,MODE_EMA,PRICE_HIGH);
58 EMA34LHandle=iMA(NULL,0,34,0,MODE_EMA,PRICE_LOW);
59 //EMA125Handle=iMA(NULL,0,125,0,MODE_EMA,PRICE_CLOSE);
60 PSARHandle=iSAR(NULL,0,0.02, 0.2);
61
62 EMA125Handle=iMA(NULL,PERIOD_D1,125,0,MODE_EMA,PRICE_CLOSE);
```



In this case, the delay, of course, will be reduced, but the number of false signals will increase.

Apparently, to improve this strategy, you need to use additional indicators.

Let's try to make the same indicator, but not using graphical constructions, and placing graphical objects on a symbol's chart.

In the properties of the indicator, it is no longer necessary to declare data buffers and indicator colors, and graphical series for them.

```
5 //+-----  
6 #property copyright "Copyright 2018, NOVTS"  
7 #property link      "http://novts.com"  
8 #property version   "1.00"  
9 #property indicator_chart_window  
10  
11 #property indicator_buffers 4  
12  
13 double EMA34HBuffer[];  
14 double EMA34LBuffer[];  
15 double EMA125Buffer[];  
16 double PSARBuffer[];  
17  
18 int EMA34HHandle;  
19 int EMA34LHandle;  
20 int EMA125Handle;  
21 int PSARHandle;  
22  
23 int     bars_calculated=0;
```

We leave only the indicator's buffers for intermediate calculations and handles of the indicators used.

```
27 int OnInit()
28 {
29 //--- indicator buffers mapping
30 SetIndexBuffer(0,EMA34HBuffer,INDICATOR_CALCULATIONS);
31 SetIndexBuffer(1,EMA34LBuffer,INDICATOR_CALCULATIONS);
32 SetIndexBuffer(2,EMA125Buffer,INDICATOR_CALCULATIONS);
33 SetIndexBuffer(3,PSARBuffer,INDICATOR_CALCULATIONS);
34
35 EMA34HHandle=iMA(NULL,0,34,0,MODE_EMA,PRICE_HIGH);
36 EMA34LHandle=iMA(NULL,0,34,0,MODE_EMA,PRICE_LOW);
37 EMA125Handle=iMA(NULL,0,125,0,MODE_EMA,PRICE_CLOSE);
38 PSARHandle=iSAR(NULL,0,0.02, 0.2);
39 //---
40     return(INIT_SUCCEEDED);
41 }
```

In the OnInit function, respectively, we leave only the binding of arrays to the intermediate calculation buffers and getting handles of the indicators used.

```

81 | for(int i=start;i<rates_total && !IsStopped();i++)
82 |
83 if(close[i-1]>open[i-1]&&close[i-1]>EMA34HBuffer[i-1]&&close[i-1]>EMA34LBuffer[i-1]
84 &&low[i-1]>EMA125Buffer[i-1]&&low[i-1]>PSARBuffer[i-1]&&EMA125Buffer[i-1]<EMA34LBuffer[i-1]
85 &&EMA125Buffer[i-1]<EMA34HBuffer[i-1]){
86
87     if(!ObjectCreate(0,"Buy"+i,OBJ_ARROW,0,time[i-1],high[i-1]))
88     {
89         return(false);
90     }
91     ObjectSetInteger(0,"Buy"+i,OBJPROP_COLOR,clrGreen);
92     ObjectSetInteger(0,"Buy"+i,OBJPROP_ARROWCODE,233);
93     ObjectSetInteger(0,"Buy"+i,OBJPROP_WIDTH,2);
94     ObjectSetInteger(0,"Buy"+i,OBJPROP_ANCHOR,ANCHOR_UPPER);
95     ObjectSetInteger(0,"Buy"+i,OBJPROP_HIDDEN,true);
96     ObjectSetString(0,"Buy"+i,OBJPROP_TOOLTIP,""+close[i-1]);
97 }
98 if(close[i-1]<open[i-1]&&close[i-1]<EMA34HBuffer[i-1]&&close[i-1]<EMA34LBuffer[i-1]
99 &&high[i-1]<EMA125Buffer[i-1]&&high[i-1]<PSARBuffer[i-1]&&EMA125Buffer[i-1]>EMA34LBuffer[i-1]
100 &&EMA125Buffer[i-1]>EMA34HBuffer[i-1]){
101
102     if(!ObjectCreate(0,"Sell"+i,OBJ_ARROW,0,time[i-1],low[i-1]))
103     {
104         return(false);
105     }
106     ObjectSetInteger(0,"Sell"+i,OBJPROP_COLOR,clrRed);
107     ObjectSetInteger(0,"Sell"+i,OBJPROP_ARROWCODE,234);
108     ObjectSetInteger(0,"Sell"+i,OBJPROP_WIDTH,2);
109     ObjectSetInteger(0,"Sell"+i,OBJPROP_ANCHOR,ANCHOR_LOWER);
110     ObjectSetInteger(0,"Sell"+i,OBJPROP_HIDDEN,true);
111     ObjectSetString(0,"Sell"+i,OBJPROP_TOOLTIP,""+close[i-1]);
112 }
113 }

```

In the OnCalculate function, we define the creation of objects on a symbol's chart.

Here, the ObjectCreate function creates arrow objects tied to time and a maximum or minimum price.

And two different arrow objects are created depending on whether the conditions are appropriate for the purchase or sale of a financial instrument.

After creating the arrow object, the ObjectSetInteger function with the OBJPROP_COLOR property defines a color of the arrow.

The ObjectSetInteger function with the OBJPROP_ARROWCODE property defines a direction of the up or down of the arrow.

The ObjectSetInteger function with the OBJPROP_WIDTH property defines the size of the object.

And the ObjectSetInteger function with the OBJPROP_ANCHOR property defines the peg to the price from above or below in the center.

The ObjectSetInteger function with the OBJPROP_HIDDEN property as true defines the absence of created objects in the list of objects of a symbol's chart.

And the ObjectSetString function with the OBJPROP_TOOLTIP property defines the content of the tooltip when the mouse pointer is placed over an

object.

```
119 void OnDeinit(const int reason){  
120 ObjectsDeleteAll(0,-1,-1);  
121 }
```

Now, in the OnDeinit function, we remove all added graphical objects using the ObjectsDeleteAll function.

And in more detail about the creation of objects on a symbol's chart, we will talk later.



By the way, we used the `ObjectSetInteger` function with the `OBJPROP_HIDDEN` property as true, so as not to clutter up the list of objects of a symbol's chart with our created arrow objects.

Graphical Objects

As it was shown earlier, we can draw not only indicator' plots on a symbol's chart, but also we can add various graphical objects using the ObjectCreate function.

ObjectCreate

The function creates an object with the specified name, type, and the initial coordinates in the specified chart subwindow. During creation up to 30 coordinates can be specified.

```
bool ObjectCreate(
    long     chart_id,      // chart identifier
    string   name,          // object name
    ENUM_OBJECT type,        // object type
    sub_Window nwin,        // window index
    datetime time1,         // time of the first anchor point
    double   price1,        // price of the first anchor point
    ...
    datetime timeN=0,        // time of the N-th anchor point
    double   priceN=0,        // price of the N-th anchor point
    ...
    datetime time30=0,       // time of the 30th anchor point
    double   price30=0,       // price of the 30th anchor point
);
```

Here, the `sub_window` parameter is the index of the main window of a symbol's chart with a value 0 or an index of a sub-window of another indicator attached to a symbol's chart.

```

99     if(!ObjectCreate(0,"Buy1"+i,OBJ_ARROW,1,time[i-1],high[i-1]))
100    {
101       return(false);
102    }
103
104    ObjectSetInteger(0,"Buy1"+i,OBJPROP_COLOR,clrGreen);
105    ObjectSetInteger(0,"Buy1"+i,OBJPROP_ARROWCODE,233);
106    ObjectSetInteger(0,"Buy1"+i,OBJPROP_WIDTH,2);
107    ObjectSetInteger(0,"Buy1"+i,OBJPROP_ANCHOR,ANCHOR_UPPER);
108    ObjectSetInteger(0,"Buy1"+i,OBJPROP_HIDDEN,true);
109    ObjectSetString(0,"Buy1"+i,OBJPROP_TOOLTIP,close[i-1]);
110
111
128    if(!ObjectCreate(0,"Sell1"+i,OBJ_ARROW,1,time[i-1],low[i-1]))
129    {
130       return(false);
131    }
132
133    ObjectSetInteger(0,"Sell1"+i,OBJPROP_COLOR,clrRed);
134    ObjectSetInteger(0,"Sell1"+i,OBJPROP_ARROWCODE,234);
135    ObjectSetInteger(0,"Sell1"+i,OBJPROP_WIDTH,2);
136    ObjectSetInteger(0,"Sell1"+i,OBJPROP_ANCHOR,ANCHOR_LOWER);
137    ObjectSetInteger(0,"Sell1"+i,OBJPROP_HIDDEN,true);
138    ObjectSetString(0,"Sell1"+i,OBJPROP_TOOLTIP,close[i-1]);

```

For example, if in the previous example with the custom indicator Impulse Keeper, we change the code by adding arrow objects to the sub-window with the index 1, and we attach, say, the ADX indicator to a symbol's chart, r, we can see the following.



The sub-window numbering goes from top to bottom in the display order. The third parameter of the `ObjectCreate` function — a type of the displayed object specified by the `ENUM_OBJECT` enumeration, which can be viewed in the MQL5 reference.

Object Types

When a graphical object is created using the `ObjectCreate()` function, it's necessary to specify the type of object being created, which can be one of the values of the `ENUM_OBJECT` enumeration. Further specifications of object [properties](#) are possible using functions for working with [graphical objects](#).

`ENUM_OBJECT`

ID	Description
<code>OBJ_VLINE</code>	Vertical Line
<code>OBJ_HLINE</code>	— Horizontal Line
<code>OBJ_TREND</code>	/ Trend Line
<code>OBJ_TREND_BY_ANGLE</code>	△ Trend Line By Angle
<code>OBJ_CYCLES</code>	□ Cycle Lines
<code>OBJ_ARROWED_LINE</code>	↗ Arrowed Line
<code>OBJ_CHANNEL</code>	△ Equidistant Channel
<code>OBJ_STDEVCHANNEL</code>	△ Standard Deviation Channel
<code>OBJ_REGRESSION</code>	↖ Linear Regression Channel
<code>OBJ_PITCHFORK</code>	≡ Andrews' Pitchfork
<code>OBJ_GANNLINE</code>	/ Gann Line
<code>OBJ_GANNFAN</code>	△ Gann Fan
<code>OBJ_GANNGRID</code>	× Gann Grid

After adding graphical objects, do not forget to delete them in the `OnDeinit` callback function using the `ObjectDelete` function.

```
bool ObjectDelete(
    long chart_id, // chart identifier
    string name     // object name
);

int ObjectsDeleteAll(
    long chart_id,      // chart identifier
    int sub_window=-1, // window index
    int type=-1        // object type
);
```

Or using the ObjectsDeleteAll function, where the sub_window = -1 means all the subwindows of the chart, including the main window.

- [ObjectCreate](#)
- [ObjectName](#)
- [ObjectDelete](#)
- [ObjectsDeleteAll](#)
- [ObjectFind](#)
- [ObjectGetTimeByValue](#)
- [ObjectGetValueByTime](#)
- [ObjectMove](#)
- [ObjectsTotal](#)
- [ObjectSetDouble](#)
- [ObjectSetInteger](#)
- [ObjectSetString](#)
- [ObjectGetDouble](#)
- [ObjectGetInteger](#)
- [ObjectGetString](#)
- [TextSetFont](#)
- [TextOut](#)
- [TextGetSize](#)

Besides the above-mentioned ObjectCreate, ObjectDelete, and ObjectsDeleteAll functions, MQL5 offers a whole range of functions for working with graphical objects: ObjectName, ObjectFind, and others.

The ObjectName, ObjectFind, ObjectGetTimeByValue, ObjectGetValueByTime, ObjectsTotal, ObjectGetDouble, ObjectGetInteger, ObjectGetString, TextGetSize functions are functions that return some information.

The ObjectSetDouble, ObjectSetInteger, ObjectSetString, TextSetFont functions are functions that set the properties of an object.

The ObjectMove function moves an object in a window.

The TextOut function outputs text to a pixel array for displaying by an OBJ_BITMAP_LABEL or OBJ_BITMAP object.

After adding graphical objects, it is recommended to force a symbol's chart to be redrawn using the ChartRedraw function.

ChartRedraw

This function calls a forced redrawing of a specified chart.

```
void ChartRedraw(  
    long chart_id=0           // Chart ID  
);
```

Parameters

chart_id=0
[in] Chart ID. 0 means the current chart.

Note

Usually it is used after changing the [object properties](#).

See also

[Objects functions](#)

It should be noted that the ObjectCreate function allows you to programmatically create graphical objects that you can manually draw on a symbol's chart using the toolbar of the client terminal.

ObjectSetDouble

The function sets the value of the corresponding object property. The object property must be of the [double](#) type. There are 2 variants of the function.

Setting property value, without modifier

```
bool ObjectSetDouble(  
    long          chart_id,      // chart identifier  
    string        name,         // object name  
    ENUM_OBJECT_PROPERTY_DOUBLE prop_id, // property  
    double        prop_value   // value  
) ;
```

Setting a property value indicating the modifier

```
bool ObjectSetDouble(  
    long          chart_id,      // chart identifier  
    string        name,         // object name  
    ENUM_OBJECT_PROPERTY_DOUBLE prop_id, // property  
    int           prop_modifier, // modifier  
    double        prop_value   // value  
) ;
```

Using the ObjectSetDouble function, you can set properties of a graphical object, such as:

the OBJPROP_PRICE property — to change the price parameter of the ObjectCreate function,

the OBJPROP_LEVELVALUE property - to define levels for objects such as the Fibonacci and Andrews Fork tool objects,

the OBJPROP_SCALE property — to define a scale for objects such as the Gunn and Arc Fibonacci tool objects,

the OBJPROP_ANGLE property - to define an angle of an object, that is the ability to rotate an object that does not initially have a hard binding, for example, to rotate text,

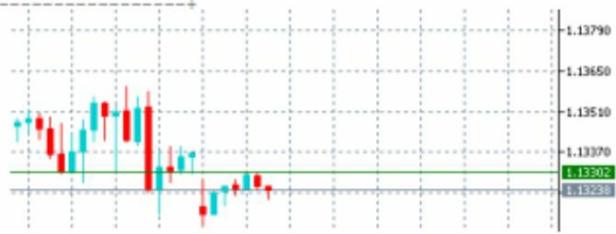
The OBJPROP_DEVIATION property — to define a deviation for the Standard Deviation Channel object.

And let's see an example of using the OBJPROP_PRICE property.

```

35 ArraySetAsSeries(time, true);
36 ArraySetAsSeries(high, true);
37 ArraySetAsSeries(low, true);
38 ArraySetAsSeries(close, true);
39 ObjectDelete(0,"Price");
40     if(!ObjectCreate(0,"Price",OBJ_HLINE,0,time[1],close[1]))
41     {
42         return(false);
43     }
44     ObjectSetInteger(0,"Price",OBJPROP_COLOR,clrGreen);
45     ObjectSetInteger(0,"Price",OBJPROP_WIDTH,1);
46     ObjectSetString(0,"Price",OBJPROP_TOOLTIP,close[1]);
47     if (open[1]>close[1])
48     ObjectSetDouble(0,"Price",OBJPROP_PRICE,low[1]);
49     if (open[1]<close[1])
50     ObjectSetDouble(0,"Price",OBJPROP_PRICE,high[1]);
51 //--- return value of prev_calculated for next call
52     return(rates_total);
53 }
54 //+-----+
55 void OnDeinit(const int reason){
56 ObjectsDeleteAll(0,-1,-1);
57 }

```



In this code, a horizontal level is created showing a minimum or maximum price of a previous bar, depending on whether this bar is bullish or bearish. And let's see an example of using the OBJPROP_ANGLE property.

```

35 ArraySetAsSeries(time, true);
36 ArraySetAsSeries(high, true);
37 ArraySetAsSeries(low, true);
38 ArraySetAsSeries(close, true);
39 ObjectDelete(0,"Line");
40 ObjectDelete(0,"Price");
41     if(!ObjectCreate(0,"Line",OBJ_VLINE,0,time[1],close[1]))
42     {
43         return(false);
44     }
45     ObjectSetInteger(0,"Line",OBJPROP_COLOR,clrBlue);
46     ObjectSetInteger(0,"Line",OBJPROP_WIDTH,1);
47     ObjectSetString(0,"Line",OBJPROP_TOOLTIP,close[1]);
48
49     if(!ObjectCreate(0,"Price",OBJ_TEXT,0,time[3],high[1]))
50     {
51         return(false);
52     }
53     ObjectSetString(0,"Price",OBJPROP_TEXT,close[1]);
54     ObjectSetInteger(0,"Price",OBJPROP_COLOR,clrBlack);
55     ObjectSetDouble(0,"Price",OBJPROP_ANGLE,90);
56     ObjectSetString(0,"Price",OBJPROP_TOOLTIP,close[1]);
57
58 //--- return value of prev_calculated for next call
59     return(rates_total);
60 }
61 //-----
62 void OnDeinit(const int reason){
63 ObjectsDeleteAll(0,-1,-1);
64 }

```



This code creates a vertical line with a label of a previous bar closing price. Using the `ObjectSetInteger` function, you can set such properties of a graphical object as color, style, size, and others.

ObjectSetInteger

The function sets the value of the corresponding object property. The object property must be of the `datetime`, `int`, `color`, `bool` or `char` type. There are 2 variants of the function.

Setting property value, without modifier

```
bool ObjectSetInteger(
    long          chart_id,      // chart identifier
    string        name,          // object name
    EDON_OBJECT_PROPERTY_INTEGER prop_id,   // property
    long          prop_value); // value
);
```

Setting a property value indicating the modifier

```
bool ObjectSetInteger(
    long          chart_id,      // chart identifier
    string        name,          // object name
    EDON_OBJECT_PROPERTY_INTEGER prop_id,   // property
    long          prop_modifier, // modifier
    long          prop_value); // value
);
```

ObjectSetString

The function sets the value of the corresponding object property. The object property must be of the `string` type. There are 2 variants of the function.

Setting property value, without modifier

```
bool ObjectSetString(
    long          chart_id,      // chart identifier
    string        name,          // object name
    EDON_OBJECT_PROPERTY_STRING prop_id,   // property
    string        prop_value); // value
);
```

Setting a property value indicating the modifier

```
bool ObjectSetString(
    long          chart_id,      // chart identifier
    string        name,          // object name
    EDON_OBJECT_PROPERTY_STRING prop_id,   // property
    long          prop_modifier, // modifier
    string        prop_value); // value
);
```

TextSetFont

The function sets the font for displaying the text using drawing methods and returns the result of that operation. Arial font with the size -120 (12 pt) is used by default.

```
bool TextSetFont(
    const string  name,          // font name or path to font file on the disk
    int          size,           // font size
    uint         flags,          // combination of flags
    int          orientation=0 // text slope angle
);
```

Using the `ObjectSetString` function, you can change the name of an object, and an object with an old name will be deleted and an object with a new name will be created.

And you can set text for objects such as text object, button, label, input field, and event.

You can set tooltip text for an object, description of a level for objects that have levels, font, a name of a BMP file for the object "Bitmap label" and "Bitmap", a symbol for the object "Graph".

The `TextSetFont` function allows you to set a font type of the text, its size, style, and an angle for objects containing text.

```

11 uint ExtImg[10000];
12 //+
13 //| Custom indicator initialization function
14 //+
15 int OnInit()
16 {
17 ObjectCreate(0,"Image",OBJ_BITMAP_LABEL,0,0);
18 ObjectSetString(0,"Image",OBJPROP_BMPFILE,"::IMG");
19 ArrayFill(ExtImg,0,10000,0xffffffff);
20 TextOut("Text",10,10,TA_LEFT|TA_TOP,ExtImg,100,100,0x000000,COLOR_FORMAT_XRGB_NOALPHA);
21 ResourceCreate("::IMG",ExtImg,100,100,0,0,COLOR_FORMAT_XRGB_NOALPHA);
22 ChartRedraw();
23
24 //---
25     return(INIT_SUCCEEDED);
26 }

```

As already mentioned, the TextOut function allows you to combine text and an image.

For example, the following code displays text in an image filled with one color.

Here, the ExtImg is a pixel array representing an image of 100x100 pixels.

The ObjectCreate function creates a bitmap label object OBJ_BITMAP_LABEL, and the ObjectSetString function sets an image file for this object with the name ::IMG. Regarding the sign "::", the MQL5 reference says the following:

To use your own resource in the code, the special sign ":" should be added before the resource name.

The ArrayFill function fills a pixel array with white pixels.

The TextOut function outputs the word "Text" to a pixel array.

The ResourceCreate function creates a resource named ::IMG from a pixel array.

As a result, the text "Text" is displayed on a white background.

```
11 #resource "\\\Images\\image.bmp"
12 uint ExtImg[10000];
13 //-----+
14 //| Custom indicator initialization function           |
15 //-----+
16 int OnInit()
17 {
18 ObjectCreate(0,"Image",OBJ_BITMAP_LABEL,0,0,0);
19 ObjectSetString(0,"Image",OBJPROP_BMPFILE,"::IMG");
20 uint width=100;
21 uint height=100;
22 ResourceReadImage("::Images\\image.bmp",ExtImg,width,height);
23 TextOut("Text",10,10,TA_LEFT|TA_TOP,ExtImg,100,100,0xffffffff,COLOR_FORMAT_ARGB_NORMALIZE);
24 ResourceCreate("::IMG",ExtImg,100,100,0,0,COLOR_FORMAT_XRGB_NOALPHA);
25 ChartRedraw();
```

You can also display text on the existing image.

Here, the ResourceReadImage function reads an existing image from the Images folder of the Navigator window of the MQL5 editor into the ::IMG pixel array associated with the Bitmap Label object, and the TextOut function outputs the word "Text" to the pixel array.

```

24 int OnCalculate(const int rates_total,
25                  const int prev_calculated,
26                  const datetime &time[],
27                  const double &open[],
28                  const double &high[],
29                  const double &low[],
30                  const double &close[],
31                  const long &tick_volume[],
32                  const long &volume[],
33                  const int &spread[])
34 {
35 //---
36 //---
37 ArraySetAsSeries(time, true);
38 ArraySetAsSeries(high, true);
39 ArraySetAsSeries(low, true);
40 ArraySetAsSeries(close, true);
41 ObjectDelete(0,"Image");
42 ObjectCreate(0,"Image",OBJ_BITMAP,0,time[1],close[1]);
43 ObjectSetString(0,"Image",OBJPROP_BMPFILE,"::IMG");
44 uint width=100;
45 uint height=100;
46 ResourceReadImage("::Images\\image.bmp",ExtImg,width,height);
47 TextOut("Text",10,10,TA_LEFT|TA_TOP,ExtImg,100,100,0xffffffff,COLOR_FORMAT_ARGB_NORMALIZE);
48 ResourceCreate("::IMG",ExtImg,100,100,0,0,COLOR_FORMAT_XRGB_NOALPHA);
49 ChartRedraw();

```

The same can be done with the object "Bitmap" - OBJ_BITMAP.

Here, in the OnCalculate function, we create a Bitmap object using the ObjectCreate function, which attaches the drawing to a Close price of the last bar.

And further, we fill the file connected with this object with the text and image.

As an example of using graphical objects, let's consider creating an indicator that displays the same chart in a small window on a symbol's chart, but with a different time period.

To do this, you can use the OBJ_CHART graphical object.

```
1 //+-----+
2 //|                                         OBJ_CHART.mq5 |
3 //|                                         Copyright 2018, NOVTS |
4 //|                                         http://novts.com |
5 //+-----+
6 #property copyright "Copyright 2018, NOVTS"
7 #property link      "http://novts.com"
8 #property version   "1.00"
9 #property indicator_chart_window
10
11 input string        InpSymbol="EURUSD";           // Symbol
12 input ENUM_TIMEFRAMES InpPeriod=PERIOD_CURRENT; // Period
13
```

As the input parameters of the indicator, we use a symbol's chart and its period.

```
17 int OnInit()
18 {
19     if(!ObjectCreate(0,"Chart",OBJ_CHART,0,0,0))
20     {
21         return(false);
22     }
23 ObjectSetInteger(0,"Chart",OBJPROP_XDISTANCE,10);
24 ObjectSetInteger(0,"Chart",OBJPROP_YDISTANCE,20);
25 ObjectSetInteger(0,"Chart",OBJPROP_XSIZE,300);
26 ObjectSetInteger(0,"Chart",OBJPROP_YSIZE,200);
27 ObjectSetString(0,"Chart",OBJPROP_SYMBOL,InpSymbol);
28 ObjectSetInteger(0,"Chart",OBJPROP_PERIOD,InpPeriod);
29 ObjectSetInteger(0,"Chart",OBJPROP_DATE_SCALE,true);
30 ObjectSetInteger(0,"Chart",OBJPROP_WIDTH,1);
31 ObjectSetInteger(0,"Chart",OBJPROP_PRICE_SCALE,true);
32 ObjectSetInteger(0,"Chart",OBJPROP_SELECTABLE,true);
33 ObjectSetInteger(0,"Chart",OBJPROP_SELECTED,true);
34 ObjectSetInteger(0,"Chart",OBJPROP_COLOR,clrBlue);
--
```

In the OnInit function, let's create the graphical object OBJ_CHART.

By default, the anchor point of this object is the upper left corner of a chart.

And we define the offset of the anchor point of an object, its size, a symbol and a period of the chart, displaying the time scale, a size of the anchor point with which you can move the object, displaying the price scale, the mouse mode moving, and a color of the chart border.

```
long chartId=ObjectGetInteger(0,"Chart",OBJPROP_CHART_ID);
```

https://www.mql5.com/en/docs/chart_operations

Chart Operations

Functions for setting chart properties (`ChartSetInteger`, `ChartSetDouble`, `ChartSetString`) are asynchronous and are used for sending update commands to a chart. If these functions are executed successfully, the command is included in the common queue of the chart events. Chart property changes are implemented along with handling of the events queue of this chart.

Thus, do not expect an immediate update of the chart after calling asynchronous functions. Use the `ChartRedraw` function to forcibly update the chart appearance and properties.

Function	Action
<code>ChartApplyTemplate</code>	Applies a specific template from a specified file to the chart
<code>ChartSaveTemplate</code>	Saves current chart settings in a template with a specified name
<code>ChartWindowFind</code>	Returns the number of a subwindow where an indicator is drawn
<code>ChartTimePriceToXY</code>	Converts the coordinates of a chart from the time/price representation to the X and Y coordinates
<code>ChartXYToTimePrice</code>	Converts the X and Y coordinates on a chart to the time and price values
<code>ChartOpen</code>	Opens a new chart with the specified symbol and period
<code>ChartClose</code>	Closes the specified chart
<code>ChartFirst</code>	Returns the ID of the first chart of the client terminal
<code>ChartNext</code>	Returns the chart ID of the chart next to the specified one
<code>ChartSymbol</code>	Returns the symbol name of the specified chart
<code>ChartPeriod</code>	Returns the period value of the specified chart
<code>ChartRedraw</code>	Calls a forced redrawing of a specified chart

https://www.mql5.com/en/docs/constants/chartconstants/enum_chart_property

Chart Properties

Identifiers of `ENUM_CHART_PROPERTY` enumerations are used as parameters of [functions for working with charts](#). The abbreviation of `r/o` in the "Property Type" column means that this property is read-only and cannot be changed. The `w/o` abbreviation in the "Property Type" column means that this property is write-only and it cannot be received. When accessing certain properties, it's necessary to specify an additional parameter-modifier (modifier), which serves to indicate the number of chart subwindows. 0 means the main window.

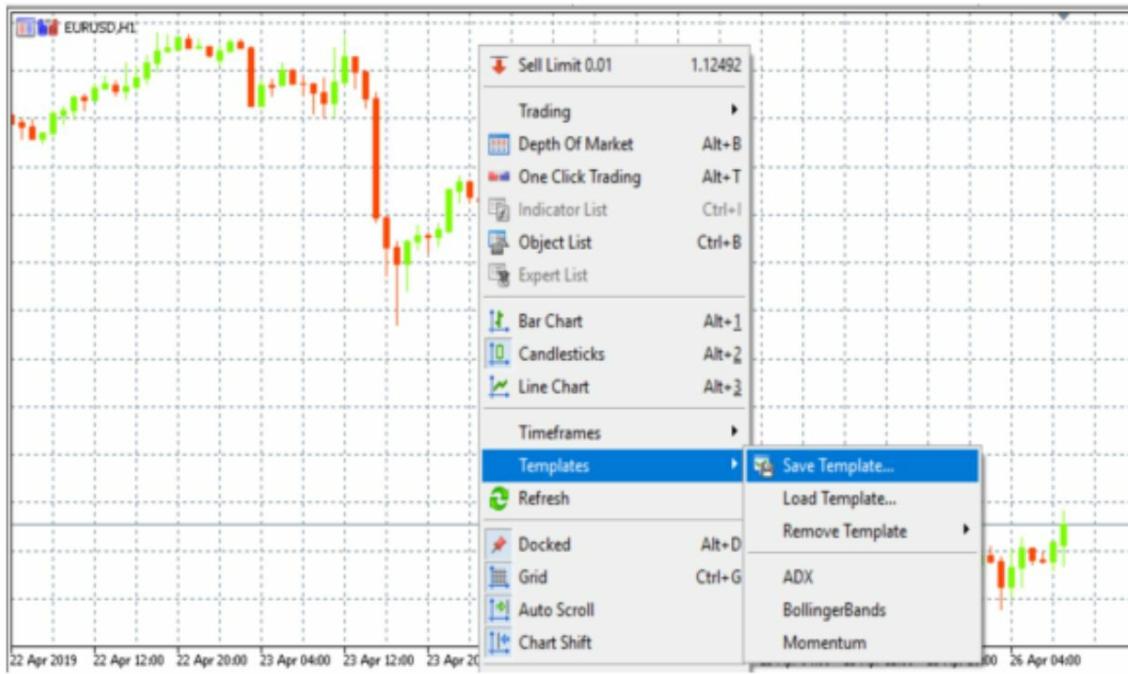
The functions defining the chart properties are actually used for sending change commands to the chart. If these functions are executed successfully, the command is included in the common queue of the chart events. The changes are implemented to the chart when handling the queue of the chart events.

Thus, do not expect an immediate visual update of the chart after calling these functions. Generally, the chart is updated automatically by the terminal following the change events - a new quote arrival, resizing the chart window, etc. Use the `ChartRedraw` function to forcibly update the chart.

For functions `ChartSetInteger()` and `ChartGetInteger()`:

ID	Description	Property Type
<code>CHART_SHOW</code>	Price chart drawing. If false, drawing any price chart attributes is disabled and all chart border indents are eliminated, including time and price scales, quick navigation bar, Calendar event labels, trade labels, indicator and bar tooltips, indicator subwindows, volume histograms, etc. Disabling the drawing is a perfect solution for creating a custom program interface using the graphical resources. The graphical objects are always drawn regardless of the <code>CHART_SHOW</code> property value.	<code>bool</code>

Using the `OBJPROP_CHART_ID` property of the `ObjectGetInteger` function, we can obtain a chart identifier, using which we can now use the functions of working with the chart and properties of the chart.



Let's open our chart of the symbol to which we want to attach an indicator, and by right-clicking the mouse, select the items Templates and Save Template in the context menu.

```

17 int OnInit()
18 {
19     if(!ObjectCreate(0,"Chart",OBJ_CHART,0,0,0))
20     {
21         return(false);
22     }
23 ObjectSetInteger(0,"Chart",OBJPROP_XDISTANCE,10);
24 ObjectSetInteger(0,"Chart",OBJPROP_YDISTANCE,20);
25 ObjectSetInteger(0,"Chart",OBJPROP_XSIZE,300);
26 ObjectSetInteger(0,"Chart",OBJPROP_YSIZE,200);
27 ObjectSetString(0,"Chart",OBJPROP_SYMBOL,InpSymbol);
28 ObjectSetInteger(0,"Chart",OBJPROP_PERIOD,InpPeriod);
29 ObjectSetInteger(0,"Chart",OBJPROP_DATE_SCALE,true);
30 ObjectSetInteger(0,"Chart",OBJPROP_WIDTH,1);
31 ObjectSetInteger(0,"Chart",OBJPROP_PRICE_SCALE,true);
32 ObjectSetInteger(0,"Chart",OBJPROP_SELECTABLE,true);
33 ObjectSetInteger(0,"Chart",OBJPROP_SELECTED,true);
34 ObjectSetInteger(0,"Chart",OBJPROP_COLOR,clrBlue);
35
36 long chartId=ObjectGetInteger(0,"Chart",OBJPROP_CHART_ID);
37 ChartApplyTemplate(chartId,"my.tpl");
38 ChartRedraw(chartId);
39
40 //---
41     return(INIT_SUCCEEDED);
42 }
```

Now we can transfer all symbol's chart settings and indicators to our graphical object.



By attaching the indicator to the symbol's chart, we can right-click on it and change its properties, including its period, size, and others.

The PlaySound function

The PlaySound function plays an audio file.

PlaySound

It plays a sound file.

```
bool PlaySound(  
    string filename      // file name  
>;
```

Parameters

filename

[in] Path to a sound file. If filename=NULL, the playback is stopped.

Return Value

true - if the file is found, otherwise - false.

Note

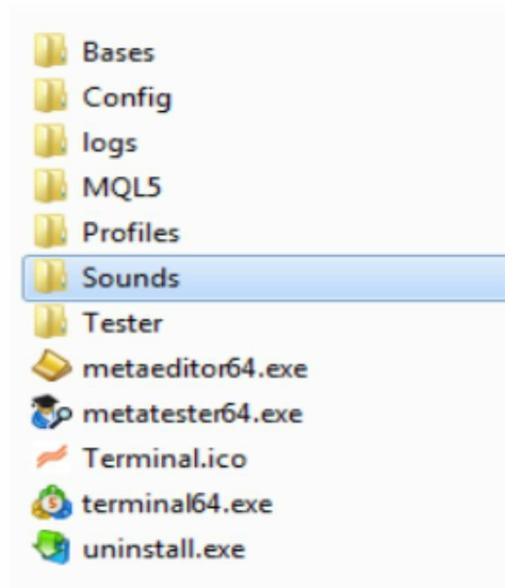
The file must be located in terminal_directory\Sounds or its sub-directory. Only WAV files are played.

Call of PlaySound() with NULL parameter stops playback.

PlaySound() function does not work in the [Strategy Tester](#).

For example, this can be done when an indicator's signal appears for a reminder.

As an example, let's add a beep to our Impulse Keeper indicator when a first buy or sell signal appears.



Let's download some WAV-signal from the Internet and place its file in the Sounds folder of the terminal.

And let's add the code to the Impulse keeper indicator.

```

25 int countBuy=0;
26 int countSell=0;

104 if (start!=1){
105
106 if(close[i-1]>open[i-1]&&close[i-1]>EMA34HBuffer[i-1]&&close[i-1]>EMA34LBuffer[i-1]
107 &&low[i-1]>EMA125Buffer[i-1]&&low[i-1]>PSARBuffer[i-1]&&EMA125Buffer[i-1]<EMA34LBuffer[i-1]
108 &&EMA125Buffer[i-1]<EMA34HBuffer[i-1]){
109
110     countBuy++;
111     if (countBuy==1) PlaySound("alert2.wav");
112     }else{
113     countBuy=0;
114     }
115
116 if(close[i-1]<open[i-1]&&close[i-1]<EMA34HBuffer[i-1]&&close[i-1]<EMA34LBuffer[i-1]
117 &&high[i-1]<EMA125Buffer[i-1]&&high[i-1]<PSARBuffer[i-1]&&EMA125Buffer[i-1]>EMA34LBuffer[i-1]
118 &&EMA125Buffer[i-1]>EMA34HBuffer[i-1]){
119
120     countSell++;
121     if (countSell==1) PlaySound("alert2.wav");
122     }else{
123     countSell=0;
124     }
125 }

```

Here, we add counters of signals for the sale and purchase - countBuy, countSell, so that a signal sounds only when a first signal appears.

And if a buy or sell signal appears, the corresponding counter increases and an alert sounds.

And with a repeated signal, the alert does not sound.

At the end of a series of signals, the counter is reset.

The OnChartEvent function

The OnChartEvent function is a callback function that is called when a user interacts with a symbol's chart and it is called with events associated with graphic objects of a symbol' chart.

OnChartEvent

The function is called in indicators and EAs when the [ChartEvent](#) event occurs. The function is meant for handling chart changes made by a user or an MQL5 program.

```
void OnChartEvent()
    const int     id,      // event ID
    const long&  lparam,   // long type event parameter
    const double& dparam,  // double type event parameter
    const string& sparam   // string type event parameter
);
```

Parameters

id
[in] Event ID from the [ENUM_CHART_EVENT](#) enumeration.

lparam
[in] [long](#) type event parameter

dparam
[in] [double](#) type event parameter

sparam
[in] [string](#) type event parameter

Return Value

No return value

As an example of using the OnChartEvent function, let's consider our Impulse Keeper indicator and let's add functionality to it that allows you to see values of the indicators used when you click on a buy or sell signal of the indicator.

```

76 ArraySetAsSeries(time, true);
77 ArraySetAsSeries(high, true);
78 ArraySetAsSeries(low, true);
79 ArraySetAsSeries(open, true);
80 ArraySetAsSeries(close, true);
81 ArraySetAsSeries(EMA34HBuffer, true);
82 ArraySetAsSeries(EMA34LBuffer, true);
83 ArraySetAsSeries(EMA125Buffer, true);
84 ArraySetAsSeries(PSARBuffer, true);
85
86 for(int i=start;i>=1;i--)
87 {
88     if(close[i]>open[i]&&close[i]>EMA34HBuffer[i]&&close[i]>EMA34LBuffer[i]
89     &&low[i]>EMA125Buffer[i]&&low[i]>PSARBuffer[i]&&EMA125Buffer[i]<EMA34LBuffer[i]
90     &&EMA125Buffer[i]<EMA34HBuffer[i]){
91
92         if(!ObjectCreate(0,"Buy"+i,OBJ_ARROW,0,time[i],high[i]))
93     {
94             return(false);
95     }

```

To do this, let's add the OnChartEvent function to the indicator code, handling the mouse click event on the indicator's graphical object.

But firstly, here, we change the order of access to all the arrays used in the OnCalculate function using the ArraySetAsSeries function and here, we do not need to use the i-1 index since in the current tick we start the cycle with the index 1.

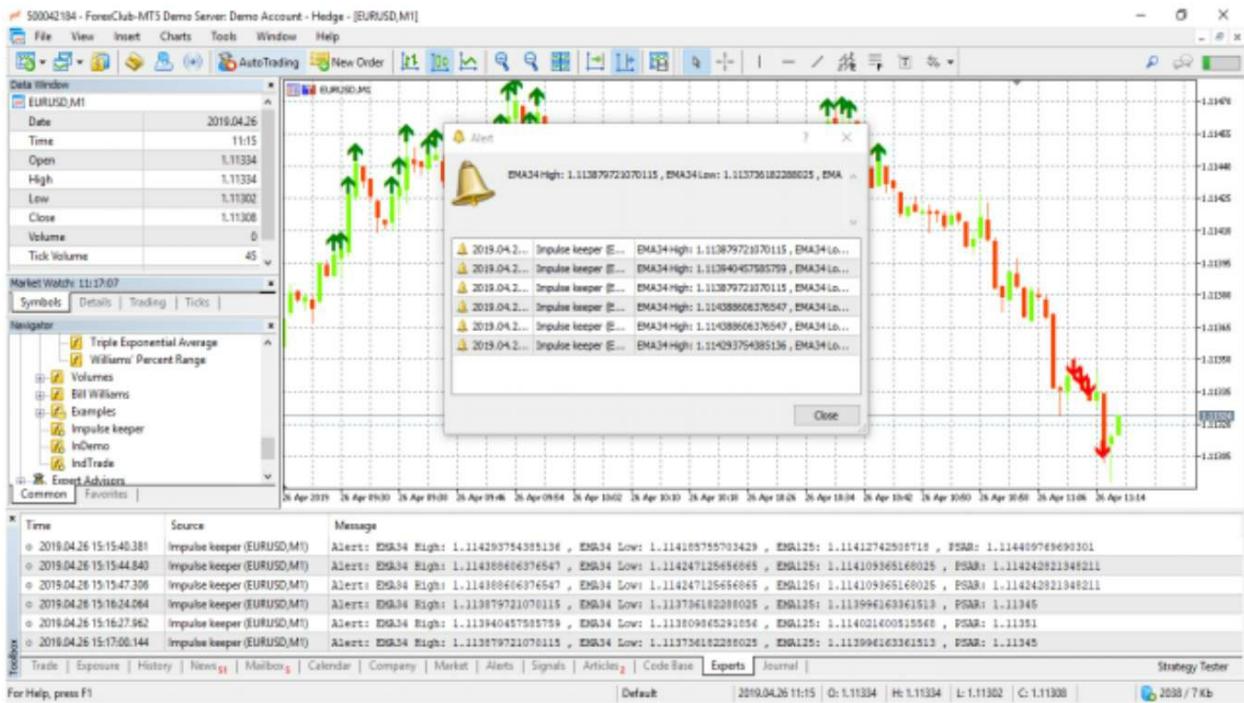
```

155 //+-----+
156 //| ChartEvent function
157 //+-----+
158 void OnChartEvent(const int id,
159                     const long &lparam,
160                     const double &dparam,
161                     const string &sparam)
162 {
163 //+---+
164     if(id==CHARTEVENT_OBJECT_CLICK){
165
166         if(StringFind(sparam,"Sell",0)!=-1){
167             int pos=StringToInteger(StringSubstr(sparam,4));
168             Alert("EMA34 High: ", EMA34HBuffer[pos], " , EMA34 Low: ",
169             EMA34LBuffer[pos], " , EMA125: ", EMA125Buffer[pos], " , PSAR: ", PSARBuffer[pos] );
170         }
171
172         if(StringFind(sparam,"Buy",0)!=-1){
173             int pos=StringToInteger(StringSubstr(sparam,3));
174             Alert("EMA34 High: ", EMA34HBuffer[pos], " , EMA34 Low: ",
175             EMA34LBuffer[pos], " , EMA125: ", EMA125Buffer[pos], " , PSAR: ", PSARBuffer[pos] );
176         }
177     }
178 }
179 //+-----+

```

Now, in the OnChartEvent function, the code first checks the event identifier and if the event is a mouse click on a graphical object, the code proceeds to check whether this object is a graphical object of the indicator.

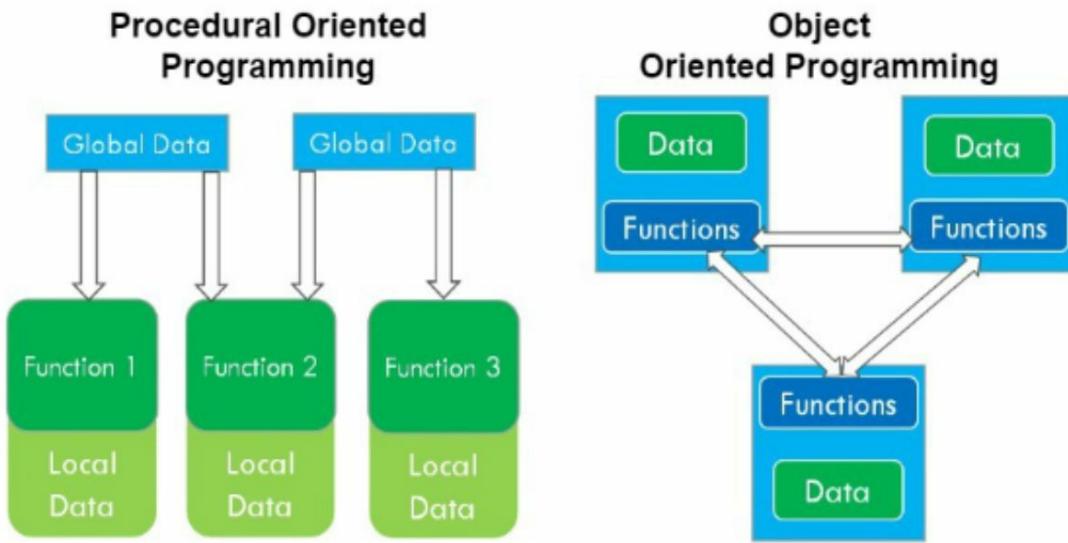
Further, the code selects a sequence number from the object name, which corresponds to the bar index, and displays buffer values of the indicators used in the Alert dialog box for displaying information to a user.



In parallel, the information is displayed in the Experts terminal window. Here, it would be convenient to display information in the MessageBox dialog box, which allows code to interact with a user, but it cannot be called from custom indicators since the indicators run in the UI thread and should not slow it down.

Object-Oriented Approach

In object-oriented languages, all are classes. Including the entry point into an application is a class that contains a specific method - the entry point into the application, or it is a class that extends a specific class of a framework or platform that contains the entry point.



With the MQL5 language, this is a bit wrong.

The development of programs with MQL5 is associated with the use of a set of callback functions that are called by the client terminal upon the occurrence of certain events.

And the MQL5 application code is not a class, but consists of a set of callback functions and supporting user code.

So, in the part of supporting user code, a developer is free to use either procedural programming or object-oriented programming.

In the case of a choice of procedural programming, user code is a set of user-defined functions.

And when choosing object-oriented programming, user code is a set of user-defined classes that can use the standard MQL5 library of classes.

Let's remember the basic concepts of object-oriented programming.

Encapsulation is when a code is represented by classes that provide public methods for accessing and modifying data, thus protecting the data.

Type extensibility is the ability to add custom data types that are based on the use of classes since each new custom class represents a new data type.

Inheritance is the ability to create new classes based on existing classes, thus reusing existing, debugged, and tested code. At the same time, there are no

multiple inheritances in MQL5.

Polymorphism is the ability for all classes of the same inheritance hierarchy to have a method with the same name but different implementation.

Overloading is the creation of class methods that have the same name but are designed to work with different data types, so the class is made universal for different data types.

As an example of using the object-oriented approach, let's consider the creation of our custom indicator Impulse Keeper using classes.

CIndicator

CIndicator is a base class for technical indicator classes of the MQL5 standard library.

Description

The **CIndicator** class provides the simplified access for all of its descendants to general MQL5 API technical indicator functions.

Declaration

```
class CIndicator: public CSeries
```

Title

```
#include <Indicators\Indicator.mqh>
```

Inheritance hierarchy

```
CObject  
CArray  
CArrayObj  
CSeries  
CIndicator
```

Direct descendants

```
CIA, CIAD, CIADX, CIADXWilder, CiAlligator, CiAMA, CIAO, CiATR, CiBands, CiBearsPower, CiBullsPower, CiBWMFI, CiCCI,  
CiChalkin, CiCustom, CiDEMA, CiDeMarker, CiEnvelopes, CiForce, CiFractals, CiFrAMA, CiGator, CiIchimoku, CiIMA, CiMACD,  
CiMFI, CiMomentum, CiOBV, CiOsMA, CiRSI, CiRV, CiSAR, CiStdDev, CiStochastic, CiTema, CiTriX, CiVidyA, CiVolumes, CiWPR
```

In this case, the use of the **CIndicator** class and its successors **CiMA** and **CiSAR**, which provide access to the MA and PSAR indicators, allows doing without buffers of the Impulse keeper indicator at all.

Since we needed them to copy the MA and PSAR indicator buffers, and the **CiMA** and **CiSAR** classes provide direct access to their buffers.

```

5 //+
6 #property copyright "Copyright 2018, NOVTS"
7 #property link      "http://novts.com"
8 #property version   "1.00"
9 #property indicator_chart_window
10
11 #include <Indicators\Trend.mqh>
12
13 CiMA           MA34H;
14 CiMA           MA34L;
15 CiMA           MA125;
16 CiSAR          SAR;
17
18 int    bars_calculated=0;
19
20 //+
21 //| Custom indicator initialization function
22 //+
23 int OnInit()
24 {
25 MA34H.Create(_Symbol,PERIOD_CURRENT,34,0,MODE_EMA,PRICE_HIGH);
26 MA34L.Create(_Symbol,PERIOD_CURRENT,34,0,MODE_EMA,PRICE_LOW);
27 MA125.Create(_Symbol,PERIOD_CURRENT,125,0,MODE_EMA,PRICE_CLOSE);
28 SAR.Create(_Symbol,PERIOD_CURRENT,0.02, 0.2);
29 //---
30     return(INIT_SUCCEEDED);
31 }

```

To use the CIndicator class and its descendants, you must include the Trend.mqh file in the code.

Next, in the code of the Impulse keeper indicator, we declare instances of the CiMA and CiSAR classes.

And in the OnInit function, we create indicators using the Create method of the CIndicator class.

```

47 int start;
48 int calculated=MA34H.BarsCalculated();
49 if(calculated<=0)
50 {
51     return(0);
52 }
53 if(prev_calculated==0 || calculated!=bars_calculated)
54 {
55     start=rates_total-1;
56 }
57 else
58 {
59     start=1;
60 }
61 ArraySetAsSeries(time, true);
62 ArraySetAsSeries(high, true);
63 ArraySetAsSeries(low, true);
64 ArraySetAsSeries(open, true);
65 ArraySetAsSeries(close, true);
66
67 //Print(MA34H.BufferSize());
68
69 MA34H.BufferResize(rates_total);
70 MA34L.BufferResize(rates_total);
71 MA125.BufferResize(rates_total);
72 SAR.BufferResize(rates_total);

```

In the OnCalculate function, after calculating the initial position of the indicator calculation, we set sizes of the buffers of the CiMA and CiSAR indicators using the BufferResize method of the CIndicator class.

If this is not done, the size of the buffers of the used indicators will have a value of 100 by default, and our indicator will be calculated only up to the 100 bar.

Next, we update the data of the used indicators and calculate and draw our indicator.

```

for(int i=start;i>=1;i--)
{
if(close[i]>open[i]&&close[i]>MA34H.Main(i)&&close[i]>MA34L.Main(i)
&&low[i]>MA125.Main(i)&&low[i]>SAR.Main(i)&&MA125.Main(i)<MA34L.Main(i)
&&MA125.Main(i)<MA34H.Main(i)){
if(!ObjectCreate(0,"Buy"+i,OBJ_ARROW,0,time[i],high[i]))
{
return(false);
}
ObjectSetInteger(0,"Buy"+i,OBJPROP_COLOR,clrGreen);
ObjectSetInteger(0,"Buy"+i,OBJPROP_ARROWCODE,233);
ObjectSetInteger(0,"Buy"+i,OBJPROP_WIDTH,2);
ObjectSetInteger(0,"Buy"+i,OBJPROP_ANCHOR,ANCHOR_UPPER);
ObjectSetInteger(0,"Buy"+i,OBJPROP_HIDDEN,true);
ObjectSetString(0,"Buy"+i,OBJPROP_TOOLTIP,close[i]);
}
if(close[i]<open[i]&&close[i]<MA34H.Main(i)&&close[i]<MA34L.Main(i)
&&high[i]<MA125.Main(i)&&high[i]<SAR.Main(i)&&MA125.Main(i)>MA34L.Main(i)
&&MA125.Main(i)>MA34H.Main(i)){
if(!ObjectCreate(0,"Sell"+i,OBJ_ARROW,0,time[i],low[i]))
{
return(false);
}
ObjectSetInteger(0,"Sell"+i,OBJPROP_COLOR,clrRed);
ObjectSetInteger(0,"Sell"+i,OBJPROP_ARROWCODE,234);
ObjectSetInteger(0,"Sell"+i,OBJPROP_WIDTH,2);
ObjectSetInteger(0,"Sell"+i,OBJPROP_ANCHOR,ANCHOR_LOWER);
ObjectSetInteger(0,"Sell"+i,OBJPROP_HIDDEN,true);
ObjectSetString(0,"Sell"+i,OBJPROP_TOOLTIP,close[i]);
}
}
ChartRedraw(0);

```

To be consistent in the object-oriented approach, all the code for calculating and drawing our indicator can be picked out into a separate user class. Fortunately, the MQL5 Editor Wizard provides the ability to create a custom class as the option "New Class" of the wizard.

```

6 #property copyright "Copyright 2018, NOVTS"
7 #property link      "http://novts.com"
8 #property version   "1.00"
9 #include <Indicators\Trend.mqh>
10 //-----
11 //|
12 //-----
13 class IKSignal
14 {
15 private:
16 int _start;
17 datetime _time[];
18 double _open[];
19 double _high[];
20 double _low[];
21 double _close[];
22
23 public:
24         IKSignal(
25             int start,
26             const datetime &time[],
27             const double &open[],
28             const double &high[],
29             const double &low[],
30             const double &close[]
31         );
32 bool draw(CiMA &MA34H, CiMA &MA34L, CiMA &MA125, CiSAR &SAR);
33         ~IKSignal();
34 };

```

Let's call this class as the IKSignal class.

And here, in the IKSignal class, we declare the private class fields, representing an initial position of the indicator calculation and a price history.

```

38 IKSignal::IKSignal(int start,
39             const datetime &time[],
40             const double &open[],
41             const double &high[],
42             const double &low[],
43             const double &close[])
44         )
45     {
46         _start=start;
47         if(ArraySize(time)>0)
48         {
49             ArrayResize(_time,ArraySize(time));
50             ArrayCopy(_time, time);
51         }
52         if(ArraySize(open)>0)
53         {
54             ArrayResize(_open,ArraySize(open));
55             ArrayCopy(_open, open);
56         }
57         if(ArraySize(high)>0)
58         {
59             ArrayResize(_high,ArraySize(high));
60             ArrayCopy(_high, high);
61         }
62         if(ArraySize(low)>0)
63         {
64             ArrayResize(_low,ArraySize(low));
65             ArrayCopy(_low, low);
66         }
67         if(ArraySize(close)>0)
68         {
69             ArrayResize(_close,ArraySize(close));
70             ArrayCopy(_close, close);
71         }
72         ArraySetAsSeries(_time, true);
73         ArraySetAsSeries(_high, true);
74         ArraySetAsSeries(_low, true);
75         ArraySetAsSeries(_open, true);
76         ArraySetAsSeries(_close, true);
77     }

```

In the class constructor, we copy its parameters into the class fields and change the order of access to the array fields.

```

85 bool IKSignal::draw(CiMA &MA34H, CiMA &MA34L, CiMA &MA125, CiSAR &SAR) {
86     for(int i=_start;i>=i;i--) {
87         if(_close[i]>_open[i] && _close[i]>MA34H.Main(i) && _close[i]>MA34L.Main(i)
88             && _low[i]>MA125.Main(i) && _low[i]>SAR.Main(i) && MA125.Main(i)<MA34L.Main(i) && MA125.Main(i)<MA34H.Main(i)) {
89             if(!ObjectCreate(0,"Buy"+i,OBJ_ARROW,0,_time[i],_high[i])) {
90                 return(false);
91             }
92             ObjectSetInteger(0,"Buy"+i,OBJPROP_COLOR,clrGreen);
93             ObjectSetInteger(0,"Buy"+i,OBJPROP_ARROWCODE,233);
94             ObjectSetInteger(0,"Buy"+i,OBJPROP_WIDTH,2);
95             ObjectSetInteger(0,"Buy"+i,OBJPROP_ANCHOR,ANCHOR_UPPER);
96             ObjectSetInteger(0,"Buy"+i,OBJPROP_HIDDEN,true);
97             ObjectSetString(0,"Buy"+i,OBJPROP_TOOLTIP,_close[i]);
98         }
99         if(_close[i]<_open[i] && _close[i]<MA34H.Main(i) && _close[i]<MA34L.Main(i)
100             && _high[i]<MA125.Main(i) && _high[i]<SAR.Main(i) && MA125.Main(i)>MA34L.Main(i) && MA125.Main(i)>MA34H.Main(i)) {
101             if(!ObjectCreate(0,"Sell"+i,OBJ_ARROW,0,_time[i],_low[i])) {
102                 return(false);
103             }
104             ObjectSetInteger(0,"Sell"+i,OBJPROP_COLOR,clrRed);
105             ObjectSetInteger(0,"Sell"+i,OBJPROP_ARROWCODE,234);
106             ObjectSetInteger(0,"Sell"+i,OBJPROP_WIDTH,2);
107             ObjectSetInteger(0,"Sell"+i,OBJPROP_ANCHOR,ANCHOR_LOWER);
108             ObjectSetInteger(0,"Sell"+i,OBJPROP_HIDDEN,true);
109             ObjectSetString(0,"Sell"+i,OBJPROP_TOOLTIP,_close[i]);
110         }
111     }
112 }
113 }
114 ChartRedraw(0);
115 return(true);
116 }
```

Also, we declare the public method draw in the class, in which the indicator will actually be calculated and drawn.

The parameters of this function are instances of the CiMA and CiSAR classes.

Here, we just transfer the code from the OnCalculate function.

Now, we don't need to include the Trend.mqh file in the code of the main file, since we have already done this in the code of the IKSignal class, instead we need to include the IKSignal class file.

```

11 #include <IKSignal.mqh>
12
13 CiMA           MA34H;
14 CiMA           MA34L;
15 CiMA           MA125;
16 CISAR          SAR;
17
18 int    bars_calculated=0;

61
62 IKSignal iks(start,time,open,high,low,close);
63
64 //Print(MA34H.BufferSize());
65
66 MA34H.BufferResize(rates_total);
67 MA34L.BufferResize(rates_total);
68 MA125.BufferResize(rates_total);
69 SAR.BufferResize(rates_total);
70
71     MA34H.Refresh();
72     MA34L.Refresh();
73     MA125.Refresh();
74     SAR.Refresh();
75
76     if(!iks.draw(MA34H, MA34L, MA125, SAR)) {
77         return(false);
78     }

```

Let's place the IKSignal class file in the Include directory and include the file in the main indicator's file.

Now, the OnCalculate function takes the following form.

Here, we create an instance of the IKSignal class with the parameters specified in the correct order and apply the method draw to the instance.

As you can see, the code of the main indicator's file is significantly simplified.



At the same time, the functionality of the indicator remained the same.

Why indicators do not work

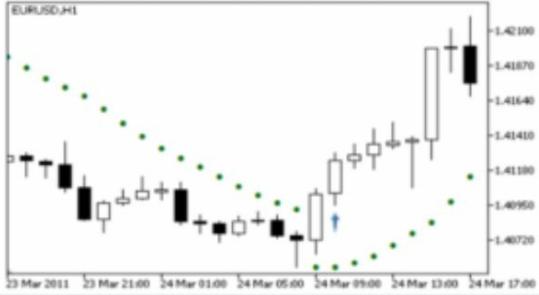
For trading on Forex, it offers a huge number of various indicators, which imposing on a price chart, you can immediately see that they all remarkably predict a price movement and give excellent signals for selling and buying.

Why, then, a novice trader immediately goes into the minus and loses money, using indicators?

The main reasons are two.

The first reason is the size of the spread. As the spread increases, the number of successful trades will rapidly decrease.

And the second reason is that, in real life, a trade does not be opened at opening the same bar on which the indicator gives a signal, but it is opened or closed later.

Signal Type	Description of Conditions
For buying	Reverse — the indicator is below the price at the analyzed bar and above the price at the previous one. 
For selling	Reverse — the indicator is above the price at the analyzed bar and below the price at the previous one. 

As an example, let's consider the Parabolic SAR indicator.

For buying - Reverse — the indicator is below the price at the analyzed bar and above the price at the previous one.

For selling - Reverse — the indicator is above the price at the analyzed bar and below the price at the previous one.

Take the indicator step as 0.02, and the spread as 0.0002 on the EURUSD pair.

Under the condition of opening and closing a position at opening the same bar where the indicator generates a signal, the following results are obtained.

For buy trades:

For the period M15 it get the number of Profit (+) 2576, Profit (-)
1659

For the period M30 it get the number of Profit (+) 2724, Profit (-)
1455

For the period H1 it get the number of Profit (+) 2797, Profit (-)
1449

For the period H4 it get the number of Profit (+) 898, Profit (-)
408

For sale trades:

For the period M15 it get the number of Profit (+) 2598, Profit (-)
1637

For the period M30 it get the number of Profit (+) 2683, Profit (-)
1496

Here, for deals to buy and sell on 15 minutes, 30 minutes, hourly and four hourly charts, we get more profitable trades than losing ones.

As you can see, the indicator works at any interval, and the number of transactions with a positive profit exceeds the number of transactions with a negative profit, and we must earn money in any case.

However, in real life, a trade is opened and closed later, and the results will be as follows for opening and closing, for example, at the next bar.

For buy trades:

For the period M15 it get the number of Profit(+) 1404, Profit(-) 2832

For the period M30 it get the number of Profit(+) 1486, Profit(-) 2693

For the period H1 it get the number of Profit(+) 1556, Profit(-) 2690

For the period H4 it get the number of Profit(+) 507, Profit(-) 799

For sale trades:

For the period M15 it get the number of Profit(+) 1379, Profit(-) 2857

For the period M30 it get the number of Profit(+) 1380, Profit(-) 2799

For the period H1 it get the number of Profit(+) 1497, Profit(-)

Here for deals to buy and sell on 15 minutes, 30 minutes, hourly and four hourly charts, we get more losing trades than profitable ones.

As we can see, the picture changes dramatically and at any interval, the indicator does not work, and the number of transactions with a negative profit exceeds the number of transactions with a positive profit, and in any case, we lose money.

Of course, this situation depends on a specific currency pair and a specific period of time.

For some period of time, such an indicator can show a profit.

But in the long term, when opening and closing a position on the next bar, the indicator will not show a total profit.

Therefore, it is impossible to use one indicator in real trading and it is necessary to develop complex trading strategies.

Adviser General Structure

An expert advisor is an MQL5 program that can automatically place and close orders for the purchase and sale of a financial instrument, thus, carrying out automatic trading in the client terminal.

Expert Advisor is an automated trading system linked to a chart.

An Expert Advisor contains event handlers to manage predefined events which activate execution of appropriate trading strategy elements.

For example, an event of program initialization and deinitialization, new ticks, timer events, changes in the Depth of Market, chart and custom events.

In addition to calculating trading signals based on the implemented rules, Expert Advisors can also automatically execute trades and send them directly to a trading server.

Expert Advisors are stored in <Terminal_Directory>\MQL5\Experts.

Like indicators, the expert's code is based on callback functions called by the client terminal when certain events occur.

For an expert, there are such functions as the OnInit, OnDeinit, OnTick, OnTimer, OnTrade, OnTradeTransaction, OnTester, OnTesterInit, OnTesterPass, OnTesterDeinit, OnBookEvent, OnChartEvent functions.

However, for the development of automated trading, two functions OnInit and OnTick are enough.

OnInit

The function is called in indicators and EAs when the [Init](#) event occurs. It is used to initialize a running MQL5 program. There are two function types.

The version that returns the result

```
int OnInit(void);
```

Return Value

[int](#) type value, zero means successful initialization.

The OnInit() call that returns the execution result is recommended for use since it not only allows for program initialization, but also returns an error code in case of an early program termination.

The version without a result return is left only for compatibility with old codes. It is not recommended for use

```
void OnInit(void);
```

OnTick

The function is called in EAs when the [NewTick](#) event occurs to handle a new quote.

```
void OnTick(void);
```

Return Value

No return value

Note

The [NewTick](#) event is generated only for EAs upon receiving a new tick for a symbol of the chart the EA is attached to. There is no point in defining the OnTick() function in a custom indicator or a script since a NewTick event is not generated for them.

Unlike indicators, experts do not specifically declare properties, with the exception of the properties link, copyright, version and description, and if an expert does not draw an indicator along with trading.

Therefore, before the callback functions, in an expert, it declares input parameters, handles of the technical indicators used, global variables, and constants.

Here, however, there is one parameter that you cannot see in an indicator, but which is present for an expert.

The Trade Request Structure (MqlTradeRequest)

Interaction between the client terminal and a trade server for executing the order placing operation is performed by using trade requests. The trade request is represented by the special predefined [structure](#) of `MqlTradeRequest` type, which contain all the fields necessary to perform trade deals. The request processing result is represented by the structure of `MqlTradeResult` type.

```
struct MqlTradeRequest
{
    ENUM_TRADE_REQUEST_ACTIONS    action;           // Trade operation type
    ulong                           magic;            // Expert Advisor ID (magic number)
    ulong                           order;            // Order ticket
    string                          symbol;           // Trade symbol
    double                          volume;           // Requested volume for a deal in lots
    double                          price;             // Price
    double                          stoplimit;         // StopLimit level of the order
    double                          sl;                // Stop Loss level of the order
    double                          tp;                // Take Profit level of the order
    ulong                           deviation;        // Maximal possible deviation from the requested price
    ENUM_ORDER_TYPE                 type;              // Order type
    ENUM_ORDER_TYPE_FILLING         type_filling;     // Order execution type
    ENUM_ORDER_TYPE_TIME            type_time;        // Order expiration type
    datetime                        expiration;       // Order expiration time (for the orders of ORDER_TIME_SPECIFIED type)
    string                          comment;          // Order comment
    ulong                           position;         // Position ticket
    ulong                           position_by;      // The ticket of an opposite position
};
```

Fields description

Field	Description
action	Trade operation type. Can be one of the ENUM TRADE REQUEST ACTIONS enumeration values.
magic	Expert Advisor ID. It allows organizing analytical processing of trade orders. Each Expert Advisor can set its own unique ID when sending a trade request.

This is a magic number or an expert's id.

With the help of the magic number, trade orders are identified when it placed by an expert.

This makes it possible to create an interconnected system of several working experts.

```
5 //+-----+
6 #property copyright "Copyright 2018, NOVTS"
7 #property link      "http://novts.com"
8 #property version   "1.00"
9 //+-----+
10 //| Expert initialization function
11 //+-----+
12 int OnInit()
13 {
14 //---
15
16 //---
17     return(INIT_SUCCEEDED);
18 }
19 //+-----+
20 //| Expert deinitialization function
21 //+-----+
22 void OnDeinit(const int reason)
23 {
24 //---
25
26 }
27 //+-----+
28 //| Expert tick function
29 //+-----+
30 void OnTick()
31 {
32 //---
33
34 }
```

The expert's OnInit function initializes handles of the technical indicators used and expert's variables.

The expert's OnDeinit function, as a rule, deletes global variables of the client terminal created by an expert during testing, optimization or debugging, as well as releasing calculated parts of the indicators used and removing indicators from a chart after the expert has finished testing using the IndicatorRelease function.

And the global variables of the client terminal are different from global variables of the MQL5 application.

GlobalVariableSet

Sets a new value for a global variable. If the variable does not exist, the system creates a new global variable.

```
datetime GlobalVariableSet(
    string name,           // Global variable name
    double value           // Value to set
);
```

Parameters

name
[in] Global variable name.
value
[in] The new numerical value.

Return Value

If successful, the function returns the last modification time, otherwise 0. For more details about the [error](#), call [GetLastError\(\)](#).

Global variables of the client terminal can be created by an MQL5 application using the GlobalVariableSet function, but they also become available to other MQL5 applications of the client terminal, unlike global variables of an MQL5 application.

Thus, the global variables of the client terminal are a means of communication between different MQL5 applications.

As a rule, global variables of the client terminal are created by experts to check for the expiration of a time limit for a previous deal.

By themselves, global variables exist in the client terminal 4 weeks after the last access, after which they are automatically destroyed. And accessing a global variable is not only setting a new value but also reading the value of a global variable.

When testing an expert, global variables of the client terminal are emulated, and they are in no way associated with real global variables of the terminal.

All operations with global variables of the terminal when testing an expert are performed in a testing agent outside the client terminal.

GlobalVariablesDeleteAll

Deletes global variables of the client terminal.

```
int GlobalVariablesDeleteAll(
    string   prefix_name=NULL,      // All global variables with names beginning with the prefix
    datetime limit_data=0          // All global variables that were changed before this date
);
```

Parameters

`prefix_name=NULL`

[in] Name prefix global variables to remove. If you specify a prefix NULL or empty string, then all variables that meet the data criterion will be deleted.

`limit_data=0`

[in] Date to select global variables by the time of their last modification. The function removes global variables, which were changed before this date. If the parameter is zero, then all variables that meet the first criterion (prefix) are deleted.

Return Value

The number of deleted variables.

To force the destruction of global variables of the client terminal you can call the `GlobalVariableDel` or `GlobalVariablesDeleteAll` functions.

OnTick

The function is called in EAs when the [NewTick](#) event occurs to handle a new quote.

```
void OnTick(void);
```

Return Value

No return value

Note

The [NewTick](#) event is generated only for EAs upon receiving a new tick for a symbol of the chart the EA is attached to. There is no point in defining the `OnTick()` function in a custom indicator or a script since a `NewTick` event is not generated for them.

The Tick event is generated only for EAs, but this does not mean that EAs have to feature the `OnTick()` function, since Timer, BookEvent and ChartEvent events are also generated for EAs in addition to `NewTick`.

All events are handled one after another in the order of their receipt. If the queue already contains the [NewTick](#) event or this event is in the processing stage, then the new `NewTick` event is not added to mql5 application queue.

The `NewTick` event is generated regardless of whether auto trading is enabled (AutoTrading button). Disabled auto trading means only a ban on sending trade requests from an EA. The EA operation is not stopped.

Disabling auto trading by pressing the AutoTrading button does not interrupt the current execution of the `OnTick()` function.

The `OnTick` function of the expert first checks the possibility of trading on the account, the sufficiency of funds on the account, and the adequacy of the loaded price history for calculating a trading strategy.

Then, temporary filters are set for trading, then the open positions are checked, then a trading strategy is calculated and, based on its signals, positions are opened or closed or pending orders are placed.

OnTimer

The function is called in EAs during the [Timer](#) event generated by the terminal at fixed time intervals.

```
void OnTimer(void);
```

Return Value

No return value

Note

The Timer event is periodically generated by the client terminal for an EA, which activated the timer using the [EventSetTimer\(\)](#) function. Usually, this function is called in the [OnInit\(\)](#) function. When the EA stops working, the timer should be eliminated using [EventKillTimer\(\)](#), which is usually called in the [OnDeinit\(\)](#) function.

Each Expert Advisor and each indicator work with its own timer receiving events solely from this timer. During mql5 application shutdown, the timer is forcibly destroyed in case it has been created but has not been disabled by [EventKillTimer\(\)](#) function.

If you need to receive timer events more frequently than once per second, use [EventSetMillisecondTimer\(\)](#) for creating a high-resolution timer.

The minimum interval of 1000 milliseconds is used in the strategy tester. In general, when the timer period is reduced, the testing time is increased, as the handler of timer events is called more often. When working in real-time mode, timer events are generated no more than 1 time in 10-16 milliseconds due to hardware limitations.

Only one timer can be launched for each program. Each mql5 application and chart have their own queue of events where all newly arrived events are placed. If the queue already contains [Timer](#) event or this event is in the processing stage, then the new Timer event is not added to mql5 application queue.

The OnTimer function allows you to create an alternative expert model that will calculate a trading strategy not when a new tick arrives in the OnTick function, but at intervals defined by the EventSetTimer function.

In the expert's OnDeinit function, you need to remove the timer by calling the EventKillTimer function.

OnTrade

The function is called in EAs when the [Trade](#) event occurs. The function is meant for processing changes in order, position and trade lists.

```
void OnTrade(void);
```

Return Value

No return value

Note

OnTrade() is called only for Expert Advisors. It is not used in indicators and scripts even if you add there a function with the same name and type.

For any trade action (placing a pending order, opening/closing a position, placing stops, activating pending orders, etc.), the history of orders and trades and/or the list of positions and current orders is changed appropriately.

When handling an order, a trade server sends the terminal a message about the incoming [Trade](#) event. To retrieve relevant data on orders and trades from history, it is necessary to perform a trading history request using [HistorySelect\(\)](#) first.

The trade events are generated by the server in case of:

- changing active orders,
- changing positions,
- changing deals,
- changing trade history.

The OnTrade function allows you to handle the completion of a trade operation.

A trading operation is not only opening or closing a position, but it is also setting, modifying or deleting a pending order, canceling a pending order if there is a shortage of funds or after the expiration date, triggering a pending order, and modifying an open position.

For example, the OnTrade function can be used to temporarily restrict trading when a stop loss is triggered.

Changing status of a trading account occurs as a result of a series of trade operations, for example, creating an order, executing an order, deleting an order from the list of open orders, adding orders to the history, adding a deal to the history, creating a new position.

OnTradeTransaction

The function is called in EAs when the [TradeTransaction](#) event occurs. The function is meant for handling trade request execution results.

```
void OnTradeTransaction()
    const MqlTradeTransactions* trans; // trade transaction structure
    const MqlTradeRequests* request; // request structure
    const MqlTradeResults* result; // response structure
};
```

OrderSendAsync

The `OrderSendAsync()` function is used for conducting asynchronous [trade operations](#) without waiting for the trade server's response to a sent [request](#). The function is designed for high-frequency trading, when under the terms of the trading algorithm it is unacceptable to waste time waiting for a response from the server.

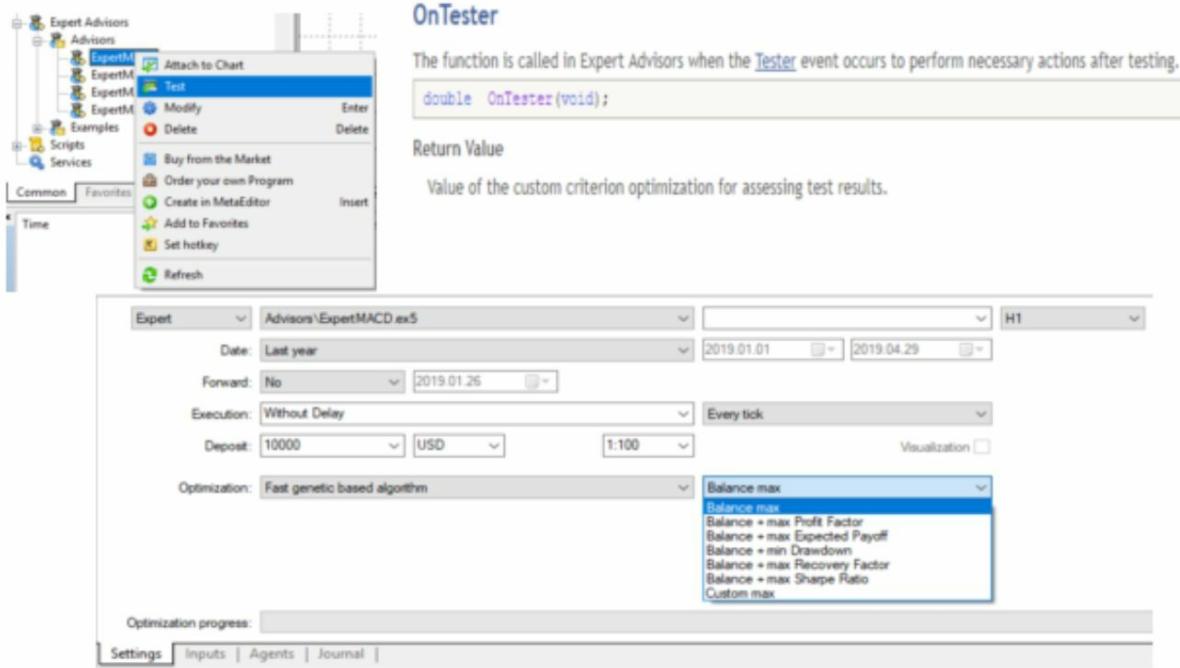
```
bool OrderSendAsync(
    MqlTradeRequests* request, // Request structure
    MqlTradeResults* result; // Response structure
);
```

Each such trade operations is followed by calling the `OnTradeTransaction` function, in which the transaction execution can be processed.

In particular, in the `OnTradeTransaction` function, you can process a result of executing a trade request on the server that is sent by the function `OrderSendAsync`.

When testing an expert in a testing mode using a genetic algorithm, the best combination of expert input parameters is selected according to an optimization criterion.

Initially, the tester offers a set of predefined optimization criteria, such as the account balance maximum, balance + maximum profit, balance + minimum drawdown, and others.



The OnTester function allows you to define a custom optimization criterion since during optimization, the tester always searches for a local maximum of a return value of the OnTester function that is automatically called after the end of the next expert testing pass at a given date interval.

To use the OnTester function, you should select the optimization criterion Custom maximum in the tester.

OnTesterInit

The function is called in EAs when the [TesterInit](#) event occurs to perform necessary actions before optimization in the strategy tester. There are two function types.

The version that returns the result

```
int OnTesterInit(void);
```

Return Value

`int` type value, zero means successful initialization of an EA launched on a chart before optimization starts.

OnTesterDeinit

The function is called in EAs when the [TesterDeinit](#) event occurs after EA optimization.

```
void OnTesterDeinit(void);
```

Return Value

No return value

OnTesterPass

The function is called in EAs when the [TesterPass](#) event occurs for handling a new data frame during EA optimization.

```
void OnTesterPass(void);
```

Return Value

No return value

The OnTesterInit, OnTesterPass, and OnTesterDeinit functions allow you to organize dynamic processing of results of expert parameters optimization in the tester during each optimization pass.

OnBookEvent

The function is called in indicators and EAs when the [BookEvent](#) event occurs. It is meant for handling Depth of Market changes.

```
void OnBookEvent(  
    const string& symbol           // symbol  
) ;
```

Parameters

symbol
[in] Name of a symbol the [BookEvent](#) has arrived for

Return Value

No return value

The OnBookEvent function allows you to develop an expert advisor or indicator that uses a trading strategy that is based on a glass of prices if of course, the dealing center provides such an opportunity.

OnChartEvent

The function is called in indicators and EAs when the [ChartEvent](#) event occurs. The function is meant for handling chart changes made by a user or an MQL5 program.

```
void OnChartEvent()
    const int      id,          // event ID
    const long     lparam,       // long type event parameter
    const double   dparam,       // double type event parameter
    const string   sparam       // string type event parameter
};
```

Parameters

id
[in] Event ID from the [ENUM_CHART_EVENT](#) enumeration.

lparam
[in] [long](#) type event parameter

dparam
[in] [double](#) type event parameter

sparam
[in] [string](#) type event parameter

The OnChartEvent function, as well as the OnTimer function, allows you to create an alternative expert model that will calculate a trading strategy not when a new tick arrives in the OnTick function, but when receiving events from indicators attached to a symbol's chart.

The OnTick function

As already mentioned, in the OnTick function, the code, as a rule, begins with various checks before calculating a trading strategy, although some checks can be performed in the OnInit function.

AccountInfoDouble

Returns the value of the corresponding account property.

```
double AccountInfoDouble(
    ENUM_ACCOUNT_INFO_DOUBLE property_id      // identifier of the property
);
```

AccountInfoInteger

Returns the value of the properties of the account.

```
long AccountInfoInteger(
    ENUM_ACCOUNT_INFO_INTEGER property_id      // Identifier of the property
);
```

AccountInfoString

Returns the value of the corresponding account property.

```
string AccountInfoString(
    ENUM_ACCOUNT_INFO_STRING property_id      // Identifier of the property
);
```

You can get client account information using the AccountInfoDouble, AccountInfoInteger, and AccountInfoString functions.

The argument of these functions is the identifier of the property whose value is to be obtained.

For the function [AccountInfoInteger\(\)](#)

ENUM_ACCOUNT_INFO_INTEGER

Identifier	Description	Type
ACCOUNT_LOGIN	Account number	long
ACCOUNT_TRADE_MODE	Account trade mode	ENUM ACCOUNT TRADE MODE
ACCOUNT_LEVERAGE	Account leverage	long
ACCOUNT_LIMIT_ORDERS	Maximum allowed number of active pending orders	int
ACCOUNT_MARGIN_SO_MODE	Mode for setting the minimal allowed margin	ENUM ACCOUNT STOPOUT MODE
ACCOUNT_TRADE_ALLOWED	Allowed trade for the current account	bool
ACCOUNT_TRADE_EXPERT	Allowed trade for an Expert Advisor	bool
ACCOUNT_MARGIN_MODE	Margin calculation mode	ENUM ACCOUNT MARGIN MODE
ACCOUNT_CURRENCY_DIGITS	The number of decimal places in the account currency, which are required for an accurate display of trading results	int

For the AccountInfoInteger function, these are the following properties:

- ACCOUNT_LOGIN - the function returns the account number.
- ACCOUNT_TRADE_MODE - the function returns a type of trading account. The function returns 0 for a demo trading account, 1 for a contest trading account, 2 for a real trading account.
- ACCOUNT_LEVERAGE - returns a size of the account leverage, for example, for the leverage 1: 100, the function returns 100.
- ACCOUNT_LIMIT_ORDERS - the function returns a maximum allowed number of pending orders. This restriction is set by a broker, and if there are no restrictions, the function returns 0.
- ACCOUNT_MARGIN_SO_MODE - it sets a type of the minimum allowable level of margin - in percentage or in money. The minimum allowable level of margin is the margin level that requires account replenishment, or the margin level upon reaching which there is a forced closing of the most unprofitable position. The minimum allowable level of margin is set by a broker and the function returns 0 if the level is set as a percentage, and returns 1 if the level is set in money.
- ACCOUNT_TRADE_ALLOWED - the function returns 0 if trading is forbidden for the account when connecting to the account in the investor mode. The function returns 0 if there is no connection to a server when trading is prohibited on the server side and if the account has been sent to the archive. The function returns 1 if trading is allowed for the account.
- ACCOUNT_TRADE_EXPERT - the function returns 0 if a broker prohibits automatic trading, and it returns 1 if automatic trading is enabled.

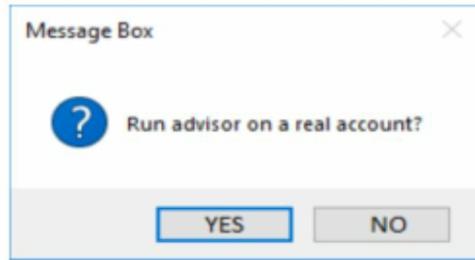
The ACCOUNT_LOGIN property can be used to protect an expert by binding it to a specific account.

```
const long _ACCOUNT=50009917;  
...  
if (AccountInfoInteger(ACCOUNT_LOGIN)!=_ACCOUNT){  
    Print("Invalid account number");  
    return(0);  
}
```

To do this, you can declare a constant that represents a valid account number, and you can compare it with a current account in the OnInit function.

A value of the ACCOUNT_TRADE_MODE property can be output as an enumeration, to do this, returned function's value should be cast to the enumeration, and then you should convert it to a string.

```
Print("ACCOUNT_TRADE_MODE ", EnumToString  
((ENUM_ACCOUNT_TRADE_MODE)AccountInfoInteger(ACCOUNT_TRADE_MODE));  
  
if((ENUM_ACCOUNT_TRADE_MODE)AccountInfoInteger(ACCOUNT_TRADE_MODE)==ACCOUNT_TRADE_MODE_REAL){  
    int mb=MessageBox("Run advisor on a real account?","Message Box",MB_YESNO|MB_ICONQUESTION);  
    if(mb==IDNO) return(0);  
}
```



The ACCOUNT_TRADE_MODE property can be used to check running an expert on a real account in the OnInit function.

Here, we compare the value of the property ACCOUNT_TRADE_MODE with the ACCOUNT_TRADE_MODE_REAL value and open the dialog box for a user.

At the same time, a user can see a dialog box that allows further code execution when selecting the Yes button.

```

bool IsNewOrderAllowed(int _max_orders)
{
int orders=OrdersTotal();
int max_allowed_orders=(int)AccountInfoInteger(ACCOUNT_LIMIT_ORDERS);

if(max_allowed_orders!=0&&_max_orders>max_allowed_orders){
_max_orders=max_allowed_orders;
}

return(orders<_max_orders);
}

input int max_orders=10; //maximum number of orders

Print(IsNewOrderAllowed(max_orders));

```

The ACCOUNT_LIMIT_ORDERS property can be used to check and set a maximum number of pending orders.

Here, we get a total number of pending orders using the OrdersTotal function.

Then, using the ACCOUNT_LIMIT_ORDERS property, we get a maximum allowed number of pending orders.

And we set the value of the maximum number of pending orders.

Next, we compare the total number of pending orders with the maximum number of pending orders.

Now, we declare the input parameter - a maximum number of orders, and we call the function defined by us.

```
//-----
if(!TerminalInfoInteger(TERMINAL_CONNECTED)){
    Alert("No connection to the trade server");
    return(0);
} else{
    if(!AccountInfoInteger(ACCOUNT_TRADE_ALLOWED)){
        Alert("Trade for this account is prohibited");
        return(0);
    }
}
if(!AccountInfoInteger(ACCOUNT_TRADE_EXPERT)){
    Alert("Trade with the help of experts for the account is prohibited");
    return(0);
}
//-----
```

Checking the ACCOUNT_TRADE_ALLOWED and ACCOUNT_TRADE_EXPERT properties can be organized in the OnInit function.

Here, we use the TERMINAL_CONNECTED property to check a connection to a broker's server.

Then, using the ACCOUNT_TRADE_ALLOWED property, we check a possibility of trading for the account.

And using the ACCOUNT_TRADE_EXPERT property, we check a possibility of automated trading.

Additionally, checking a separate connection with a server can be done in the OnTick function.

For the function [AccountInfoDouble\(\)](#)

ENUM_ACCOUNT_INFO_DOUBLE

Identifier	Description	Type
ACCOUNT_BALANCE	Account balance in the deposit currency	double
ACCOUNT_CREDIT	Account credit in the deposit currency	double
ACCOUNT_PROFIT	Current profit of an account in the deposit currency	double
ACCOUNT_EQUITY	Account equity in the deposit currency	double
ACCOUNT_MARGIN	Account margin used in the deposit currency	double
ACCOUNT_MARGIN_FREE	Free margin of an account in the deposit currency	double
ACCOUNT_MARGIN_LEVEL	Account margin level in percents	double
ACCOUNT_MARGIN_SO_CALL	Margin call level. Depending on the set ACCOUNT_MARGIN_SO_MODE is expressed in percents or in the deposit currency	double
ACCOUNT_MARGIN_SO_SO	Margin stop out level. Depending on the set ACCOUNT_MARGIN_SO_MODE is expressed in percents or in the deposit currency	double
ACCOUNT_MARGIN_INITIAL	Initial margin. The amount reserved on an account to cover the margin of all pending orders	double
ACCOUNT_MARGIN_MAINTENANCE	Maintenance margin. The minimum equity reserved on an account to cover the minimum amount of all open positions	double
ACCOUNT_ASSETS	The current assets of an account	double
ACCOUNT_LIABILITIES	The current liabilities on an account	double
ACCOUNT_COMMISSION_BLOCKED	The current blocked commission amount on an account	double

The following properties are defined for the AccountInfoDouble function.

- ACCOUNT_BALANCE - an account balance. It corresponds to a Balance value in the Trade tab of the client terminal.



ACCOUNT_CREDIT - the size of the loan provided. A typical situation is when this value is 0.

ACCOUNT_PROFIT - an amount of current profit on the account. It corresponds to the Profit column in the Trade tab of the client terminal.

ACCOUNT_EQUITY - a value of equity on the account. It corresponds to an Equity value in the Trade tab in the client terminal.

ACCOUNT_MARGIN - the size of the reserved collateral on the account. It corresponds to the Margin value in the Trade tab of the client terminal. If there are no open positions, this value is 0.

ACCOUNT_MARGIN_FREE - a volume of free funds on the account available for opening a position. It corresponds to a Free Margin value in the Trade tab of the client terminal.

ACCOUNT_MARGIN_LEVEL - a level of the margin on the account as a percentage. Corresponds to a Margin Level value in the Trade tab of the client terminal. It is calculated as Funds / Margin * 100%. If there are no open positions, this value is 0.

```
if(ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_PERCENT)
Print(AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)," %");
```

```
if(ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_MONEY)
Print(AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)," USD");
```

ACCOUNT_MARGIN_SO_CALL - a level of the margin that requires replenishment of the account (Margin Call).

Depending on the ACCOUNT_MARGIN_SO_MODE property, it is expressed as a percentage or in the deposit currency.

Margin Call is rather an informational signal for a trader that his account is near to closing and it is not accompanied by actions of a broker.

The consequences come in the Stop Out event.

For example, with the ACCOUNT_MARGIN_SO_CALL = 50%, the Margin Call event will occur when the amount of funds in the account becomes half the margin.

```
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_PERCENT)
```

```
Print(AccountInfoDouble(ACCOUNT_MARGIN_SO_SO)," %");
```

```
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_MONEY)
```

```
Print(AccountInfoDouble(ACCOUNT_MARGIN_SO_SO)," USD");
```

ACCOUNT_MARGIN_SO_SO - a level of the margin, upon reaching which there is a forced closing of the most unprofitable position (Stop Out).

Depending on the ACCOUNT_MARGIN_SO_MODE property, it is expressed as a percentage or in the deposit currency.

For example, with the ACCOUNT_MARGIN_SO_SO = 10%, the Stop Out event occurs when the amount of funds in the account is 10% of the margin, while open positions will be forcibly closed by a broker.

And let's consider other properties of the AccountInfoDouble function.

ACCOUNT_MARGIN_INITIAL - a volume of funds reserved on the account to provide a guaranteed amount for all pending orders.

As a rule, this value is 0.

ACCOUNT_MARGIN_MAINTENANCE - a volume of funds reserved on the account to ensure the minimum amount for all open positions.

As a rule, this value is 0.

ACCOUNT_ASSETS - a current amount of assets on the account.

As a rule, this value is 0.

ACCOUNT_liabilities - a current amount of obligations on the account.

As a rule, this value is 0.

`ACCOUNT_COMMISSION_BLOCKED` - a current amount of blocked account commissions.

As a rule, this value is 0.

Using the properties of the `AccountInfoDouble` function, you can organize various kinds of checks in the `OnTick` function of an expert.

Margin Call

```
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_PERCENT){  
    if(AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)!=0  
        &&AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)  
        <=AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL))  
        Alert("Margin Call!!!");  
    }  
  
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_MONEY){  
    if(AccountInfoDouble(ACCOUNT_EQUITY)  
        <=AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL))  
        Alert("Margin Call!!!");  
    }  
}
```

For example, let's consider a Margin Call event.

Here, we compare the value of the property ACCOUNT_MARGIN_SO_CALL with the volume of the account's funds.

Stop Out

```
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_PERCENT){  
    if(AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)!=0  
        &&AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)  
        <=AccountInfoDouble(ACCOUNT_MARGIN_SO_SO)){  
        Alert("Stop Out!!!");  
        return;  
    }}  
  
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_MONEY){  
    if(AccountInfoDouble(ACCOUNT_EQUITY)  
        <=AccountInfoDouble(ACCOUNT_MARGIN_SO_SO)){  
        Alert("Stop Out!!!");  
        return;  
    }}
```

Also, when checking for a Stop Out event, we compare the value of the property ACCOUNT_MARGIN_SO_SO with the volume of the account's funds.

```

double margin;
MqlTick last_tick;
ResetLastError();

if(SymbolInfoTick(Symbol(),last_tick))
{
    if(OrderCalcMargin(ORDER_TYPE_BUY, Symbol(), Lot, last_tick.ask, margin))
    {
        if(margin>AccountInfoDouble(ACCOUNT_MARGIN_FREE)){
            Alert("Not enough money in the account!");
            return;
        }
    }
}
else
{
    Print(GetLastError());
}

```

You can also organize checking a volume of free funds on the account available for opening a position.

Here, the MqlTick is a standard structure for storing prices that are populated with values using the SymbolInfoTick function.

Calling the ResetLastError function is made to reset an error before calling the function, after which the occurrence of an error is checked.

The OrderCalcMargin function calculates a volume of funds required to open a position.

And if the amount of free funds on the account (ACCOUNT_MARGIN_FREE) is less than the volume of funds required to open a position, the money on the account is not enough and trading is impossible.

For function [AccountInfoString\(\)](#)

ENUM_ACCOUNT_INFO_STRING

Identifier	Description	Type
ACCOUNT_NAME	Client name	string
ACCOUNT_SERVER	Trade server name	string
ACCOUNT_CURRENCY	Account currency	string
ACCOUNT_COMPANY	Name of a company that serves the account	string

For the AccountInfoString function, the following properties are defined: a client name ACCOUNT_NAME, a trading server name ACCOUNT_SERVER, a deposit currency ACCOUNT_CURRENCY, a name of the company servicing the account ACCOUNT_COMPANY.

Using the ACCOUNT_NAME property, as well as using the ACCOUNT_LOGIN property, you can protect an advisor.

```
if (AccountInfoString(ACCOUNT_NAME)!=_name){  
    Print("User name does not match");  
    return(0);  
}
```

Information about the client terminal can be obtained using the TerminalInfoInteger and TerminalInfoString functions.

[TerminalInfoInteger](#)

Returns the value of a corresponding property of the mql5 program environment.

```
int TerminalInfoInteger(  
    int property_id // identifier of a property  
)
```

[TerminalInfoString](#)

Returns the value of a corresponding property of the mql5 program environment. The property must be of string type.

```
string TerminalInfoString(  
    int property_id // identifier of a property  
)
```

```
#import "dll_lib.dll"  
  
TERMINAL_DLLS_ALLOWED  
  
Print((bool)TerminalInfoInteger(TERMINAL_DLLS_ALLOWED));
```

And as an argument, these functions also take properties.

We have already seen checking a terminal's connection to a server using the TERMINAL_CONNECTED property.

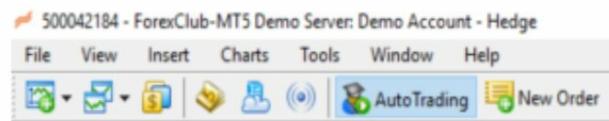
And the TERMINAL_DLLS_ALLOWED property allows you to find out if there is permission to use a DLL.

DLL files are another way to create reusable libraries - code modules for MQL5 programs.

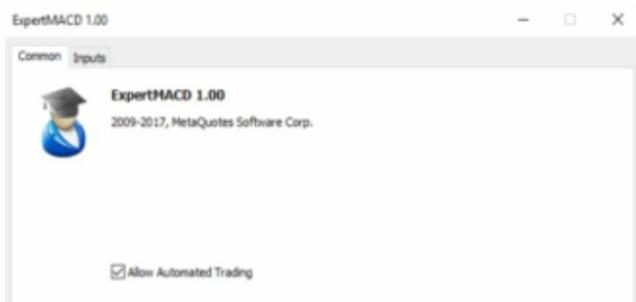
DLL libraries are located in the MQL5\Libraries folder of the trading terminal, and they are included in a code of an MQL5 program using the #import command.

At the same time, permission to use DLL libraries is set in the Expert Advisors tab of the client terminal options.

DLL libraries can also be used to protect an expert by transferring the main trading strategy code to a DLL file.



```
if (!TerminalInfoInteger(TERMINAL_TRADE_ALLOWED))
    Alert("Auto Trading Off!");
```



```
if(!MQLInfoInteger(MQL_TRADE_ALLOW ED))
    Alert("Automatic trading is prohibited in
the properties of the expert ",__FILE__);
```

The TERMINAL_TRADE_ALLOWED property indicates whether the auto-trade button is enabled in the client terminal.

To check this property, you can include code using the TERMINAL_TRADE_ALLOWED property in the OnTick function.

However, permission to trade with the help of an expert can be disabled in the common properties of the expert itself.

To check this condition, you can use the MQL_TRADE_ALLOWED property of the MQLInfoInteger function.

SymbolInfoInteger

Returns the corresponding property of a specified symbol. There are 2 variants of the function.

1. Immediately returns the property value.

```
long SymbolInfoInteger(  
    string name, // symbol  
    ENUM_SYMBOL_INFO_INTEGER prop_id // identifier of a property  
) ;
```

2. Returns true or false depending on whether a function is successfully performed. In case of success, the value of the property is placed into a recipient variable, passed by reference by the last parameter.

```
bool SymbolInfoInteger(  
    string name, // symbol  
    ENUM_SYMBOL_INFO_INTEGER prop_id, // identifier of a property  
    long& long_var // here we accept the property value  
) ;
```

```
double _spread = SymbolInfoInteger(Symbol(),SYMBOL_SPREAD)  
*MathPow(10, -SymbolInfoInteger(Symbol(),SYMBOL_DIGITS))/MathPow(10,-4);  
  
if(_spread>spreadLevel){  
    Alert("Too big spread!");  
    return;  
}
```

Using the SYMBOL_SPREAD property of the SymbolInfoInteger function, you can control a broker spread.

Here, using the SYMBOL_DIGITS property, we find out how many decimal places are in price and calculate the spread in points.

Then we compare it with a threshold value, and if the current spread is greater than the threshold value, we do not trade.

```
if((ENUM_SYMBOL_TRADE_MODE)SymbolInfoInteger(Symbol(),SYMBOL_TRADE_MODE)!=SYMBOL_TRADE_MODE_FULL){  
    Alert("Restrictions on trading operations!");  
    return;  
}
```

Using the SYMBOL_TRADE_MODE property of the SymbolInfoInteger function, you can check restrictions on trading operations for a symbol, which are set by a broker.

Here, we compare the value of the SYMBOL_TRADE_MODE property with the SYMBOL_TRADE_MODE_FULL constant.

For example, a broker may disable trading for any financial instrument.

```
input int startHour=0; //start hour
input int startMin=0; //start minute
input int stopHour=23; // stop hour
input int stopMin=0; // stop minute
...
long _seconds=TimeLocal()%86400;
long startSec=3600*startHour+60*startMin;
long stopSec=3600*stopHour+60*stopMin;
if(_seconds<startSec || _seconds>stopSec) return;
```

Also in the OnTick function, you can limit the work of an expert in time.

If you want to run an expert every day at a specified time interval, you can define starting and ending hours and minutes of the time interval and compare them with the current time.

Here, local time is taken, and if you want to compare with server time, you can use the TimeCurrent function, but not the TimeLocal function.

```
input datetime startSession=D'2015.05.05';
input datetime stopSession=D'2015.05.06';

...
datetime _session = TimeLocal();
if(_session<startSession || _session>stopSession) return;
```

If you want to run an expert simply within a specified time interval, you can define start and end dates of the time interval and compare them with the current time.

In the OnTick function of the expert, it would also be nice to check if there are enough bars in history to calculate an advisor.

```
if(Bars(Symbol(), 0)<100)
{
    Alert("In the chart little bars, Expert will not work!!");
    return;
}
```

Или

```
if(SeriesInfoInteger(Symbol(),0,SERIES_BARS_COUNT)<100)
{
    Alert("In the chart little bars, Expert will not work!!");
    return;
}
```

This can be done in two ways - using the Bars function and using the SERIES_BARS_COUNT property of the SeriesInfoInteger function.

You can limit calculations of an expert advisor in the OnTick function by the appearance of a new bar on a chart in two ways, using the SERIES_LASTBAR_DATE property of the SeriesInfoInteger function or using the CopyTime function.

```

static datetime last_time;
datetime last_bar_time=
(datetime)SeriesInfoInteger(Symbol(),Period(),
SERIES_LASTBAR_DATE);

if(last_time!=last_bar_time)
{
last_time=last_bar_time;
}
else{
return;
}

static datetime Old_Time;
datetime New_Time[1];

int copied=CopyTime(Symbol(),Period(),0,1,New_Time);
ResetLastError();
if(copied>0)
{
if(Old_Time!=New_Time[0])
{
Old_Time=New_Time[0];
}
else{
return;
}
}
else
{
Print(GetLastError());
return;
}

```

That is, here, we perform calculations in the OnTick function only when a new bar appears on a symbol's chart, skipping all intermediate ticks.

We do this by getting an open time of the last bar and using a static local variable for comparison.

If, suppose, you want to stop trading with an advisor for today after you have caught a stop loss, you should to properly handle this StopLoss event.

```

void OnTrade()
{
    static int _deals;
    ulong _ticket=0;

    if(HistorySelect(0,TimeCurrent()))
    {
        int i=HistoryDealsTotal()-1;

        if(_deals!=i) {
            _deals=i;
        } else { return; }

        if(_ticket==HistoryDealGetTicket(i))>0
        {
            string
            _comment=HistoryDealGetString(_ticket,DEAL_COMMENT);

            if(StringFind(_comment,"sl",0)!=-1) {
                flagStopLoss=true;
            }
        }
    }
}

```

As you know, the OnTrade function is called when opening or closing a position, placing, modifying or deleting a pending order, canceling a pending order if there is a shortage of funds or after the expiration date, triggering a pending order, and modifying an open position.

Therefore, in the OnTrade function, you only need to select deal events, and then select the event for StopLoss closing a position from deals.

Here, the HistorySelect function requests the history of deals and orders for the whole time, then using the HistoryDealsTotal function we get an index of the last deal and compare it with a static variable that stores an index of a previous deal. Thus, we select only an event of making a deal.

Using the HistoryDealGetTicket function, we get the last deal ticket and, using the DEAL_COMMENT property of the HistoryDealGetString function, we get a comment on the deal.

If the comment contains sl, then it was a StopLoss position closing deal.

The flagStopLoss is a global variable that we can now use in the OnTick function.

```

bool flagStopLoss=false;
...
static datetime Old_TimeD1;
datetime New_TimeD1[1];
bool IsNewBarD1=false;

int copiedD1=CopyTime(_Symbol,PERIOD_D1,0,1,New_TimeD1);
ResetLastError();
if(copiedD1>0)
{
    if(Old_TimeD1!=New_TimeD1[0])
    {
        IsNewBarD1=true;
        Old_TimeD1=New_TimeD1[0];
        flagStopLoss=false;
    }
}
else
{
    Print(GetLastError());
    return;
}
if(IsNewBarD1==false)
{
    if(flagStopLoss==true){
        return;
    }
}

```

Here, the expert stops calculating if a stop loss is received and a new daily bar has not arrived.

These flags are the FlagStopLoss and IsNewBarD1 flags, respectively.

```

bool BuyOpened=false;
bool SellOpened=false;

if(PositionSelect(_Symbol)==true)
{
    if(PositionGetInteger(POSITION_TYPE)==POSITION_TYPE_BUY)
    {
        BuyOpened=true;
    }
    else
    if(PositionGetInteger(POSITION_TYPE)==POSITION_TYPE_SELL)
    {
        SellOpened=true;
    }
}

```

Since for each financial instrument (symbol) only one open position is possible, in the OnTick function it is necessary to organize an open position check in order not to try to open it again, with its fixed volume.

Here, the PositionSelect function copies information about an open position of a symbol to the software environment. Then, using the POSITION_TYPE property of the PositionGetInteger function, it is determined whether the open position is a sell or buy position.

Positions are the presence of purchased or sold contracts for a financial instrument.

A long position (Long) is formed as a result of purchases in anticipation of a price increase, a short position (Short) is the result of the sale of an asset based on a price reduction in the future.

And there can be only one position on one account for each financial instrument.

For each symbol at any time, there can be only one open position - long or short.

Position volume may increase as a result of a new trade in the same direction. That is, a volume of a long position will be increased after a new purchase (Buy operation) and will decrease after a sale (Sell operation).

A position is considered closed if, as a result of a trade operation, the volume of obligations became equal to zero.

Such an operation is called closing a position.

After performing various checks in the OnTick function, you could perform calculations of signals of a trading system in an expert.

```
MqlRates mrate[];  
ResetLastError();  
  
if(CopyRates(Symbol(), Period(), 0, 3, mrate) < 0)  
    {  
        Print(GetLastError());  
        return;  
    }  
ArraySetAsSeries(mrate,true);
```

As a rule, historical data of a symbol is required to calculate signals of a trading system.

This can be done using the CopyRates function and the MqlRates structure, which contains historical prices of a symbol's bar.

Here, data of the last three bars are copied into an array of the MqlRates structure, and then the order of access to the array is changed.

Finally, after calculating signals of an expert's trading system, you can open or close the positions.

```
if(Lot<SymbolInfoDouble(Symbol(),SYMBOL_VOLUME_MIN)
||Lot>SymbolInfoDouble(Symbol(),SYMBOL_VOLUME_MAX))
return;
```

But before making a trade, it would be nice to check the correctness of a volume with which we are going to enter the market.

This can be done using the SYMBOL_VOLUME_MIN and SYMBOL_VOLUME_MAX properties.

OrderSend

The OrderSend() function is used for executing [trade operations](#) by sending [requests](#) to a trade server.

```
bool OrderSend(  
    MqlTradeRequest* request,      // query structure  
    MqlTradeResult* result        // structure of the answer  
);
```

Parameters

request

[in] Pointer to a structure of [MqlTradeRequest](#) type describing the trade activity of the client.

result

[in,out] Pointer to a structure of [MqlTradeResult](#) type describing the result of trade operation in case of a successful completion (if true is returned).

Return Value

In case of a successful basic check of structures (index checking) returns true. However, this is not a sign of successful execution of a trade operation. For a more detailed description of the function execution result, analyze the fields of *result* structure.

Opening and closing a position, changing a volume of an open position, changing Stop Loss and Take Profit values at an open position, setting, modifying and deleting pending orders - all this can be done using the OrderSend function.

The MqlTradeRequest structure defines a type of trading operation, which the OrderSend function will attempt to perform.

```

struct MqlTradeRequest
{
    ENUM_TRADE_REQUEST_ACTIONS    action;           // Trade operation type
    ulong                         magic;            // Expert Advisor ID (magic number)
    ulong                         order;             // Order ticket
    string                         symbol;            // Trade symbol
    double                         volume;            // Requested volume for a deal in lots
    double                         price;              // Price
    double                         stoplimit;         // StopLimit level of the order
    double                         sl;                // Stop Loss level of the order
    double                         tp;                // Take Profit level of the order
    ulong                         deviation;         // Maximal possible deviation from the requested price
    ENUM_ORDER_TYPE                type;              // Order type
    ENUM_ORDER_TYPE_FILLING        type_filling;     // Order execution type
    ENUM_ORDER_TYPE_TIME           type_time;         // Order expiration type
    datetime                       expiration;        // Order expiration time (for the orders of ORDER_TIME_SPECIFIED type)
    string                         comment;           // Order comment
    ulong                         position;          // Position ticket
    ulong                         position_by;       // The ticket of an opposite position
};

```

The first parameter `action` of the `MqlTradeRequest` structure defines a type of the trade operation of the `OrderSend` function using the `NUM_TRADE_REQUEST_ACTIONS` enumeration.

This can be an immediate execution of a buy or sell deal (`TRADE_ACTION_DEAL`),

changing Stop Loss and Take Profit values at an open position (`TRADE_ACTION_SLTP`),

setting a pending buy or sell order (`TRADE_ACTION_PENDING`),

changing pending order parameters (`TRADE_ACTION MODIFY`),

And deleting a pending order (`TRADE_ACTION_REMOVE`).

If you want to make an instant deal to buy or sell, in this case, a type of order execution for the financial instrument is defined by a broker.

This can be Instant Execution, Request Execution, Market Execution, and Exchange Execution.

You can find out the type of order execution using the `SYMBOL_TRADE_EXEMODE` property of the `SymbolInfoInteger` function.

```
Print(EnumToString((ENUM_SYMBOL_TRADE_EXEC  
UTION)  
SymbolInfoInteger(Symbol(),SYMBOL_TRADE_EXEM  
ODE)));
```

For the Instant Execution, a market order is executed at a price you offer to a broker.

If the broker cannot accept the order at the offered prices, it will offer new execution prices to the trader, which will be contained in the MqlTradeResult structure.

For the Instant Execution, the filling of the MqlTradeRequest structure for a buy order will be as follows.

```

MqlTradeRequest mrequest;
ZeroMemory(mrequest);
MqlTick latest_price;

if(!SymbolInfoTick(_Symbol,latest_price))
{
    Alert("Error getting the latest quotes - : ",GetLastError(),"!");
    return;
}
if(((ENUM_SYMBOL_TRADE_EXECUTION)SymbolInfoInteger(Symbol(),SYMBOL_TRADE_EXEMODE))==SYMBOL_TRADE_EXECUTION_INSTANT){

mrequest.action = TRADE_ACTION_DEAL;
mrequest.symbol = _Symbol;
mrequest.volume = Lot;
mrequest.price = NormalizeDouble(latest_price.ask,_Digits);
mrequest.sl = NormalizeDouble(latest_price.bid - 0.01,_Digits);
mrequest.tp = NormalizeDouble(latest_price.ask + 0.01,_Digits);
mrequest.deviation=10;
mrequest.type = ORDER_TYPE_BUY;
mrequest.type_filling = ORDER_FILLING_FOK;
}

```

Here, using the `SymbolInfoTick` function, we get current symbol prices in the `MqlTick` structure for offering to a broker.

Then, for the Instant Execution, the required fields of the structure `MqlTradeRequest` are filled: action, symbol, volume, price, sl, tp, deviation, type, and `type_filling`.

In practice, the maximum acceptable deviation from the asking price is deviation specified in points, which is accepted by a broker, and it is no more than 5 points.

With a strong market movement, upon receipt of an order to a broker, if the price has gone to a greater value, the so-called Requote will occur - the broker will return the prices at which this order can be executed.

Here, the `NormalizeDouble` function is used to round prices to the number of decimal places after the decimal point, which defines the accuracy of a symbol's price of the current chart.

```
if(((ENUM_SYMBOL_TRADE_EXECUTION)SymbolInfoInteger(Symbol(),SYMBOL_TRADE_EXEMODE))==SYMBOL_TRADE_EXECUTION_INSTANT){

mrequest.action = TRADE_ACTION_DEAL;
mrequest.symbol = _Symbol;
mrequest.volume = Lot;
mrequest.price = NormalizeDouble(latest_price.ask,_Digits);
mrequest.sl = NormalizeDouble(latest_price.ask + 0.01,_Digits);
mrequest.tp = NormalizeDouble(latest_price.bid - 0.01,_Digits);
mrequest.deviation=10;
mrequest.type = ORDER_TYPE_SELL;
mrequest.type_filling = ORDER_FILLING_FOK;
}
```

For a sell order, the Instant Execution filling in the required fields of the MqlTradeRequest structure will be as follows.

After filling in the fields of the MqlTradeRequest structure, it is recommended to check it using the OrderCheck function.

OrderCheck

The OrderCheck() function checks if there are enough money to execute a required [trade operation](#). The check results are placed to the fields of the [MqlTradeCheckResult](#) structure.

```
bool OrderCheck(
    MqlTradeRequest& request,      // request structure
    MqlTradeCheckResult& result    // result structure
);
```

```
struct MqlTradeCheckResult
{
    uint     retcode;           // Reply code
    double   balance;          // Balance after the execution of the deal
    double   equity;           // Equity after the execution of the deal
    double   profit;           // Floating profit
    double   margin;           // Margin requirements
    double   margin_free;      // Free margin
    double   margin_level;     // Margin level
    string   comment;          // Comment to the reply code (description of the error)
};
```

The results of the check will be contained in the [MqlTradeCheckResult](#) structure.

The OrderCheck function returns true if the [MqlTradeRequest](#) structure has been successfully verified, and the retcode will be 0, otherwise, the function will return false.

```
MqlTradeCheckResult check_result;
ZeroMemory(check_result);

if(!OrderCheck(mrequest,check_result))
{
    if (check_result.retcode == 10014) Alert ("Invalid volume in the request");
    if (check_result.retcode == 10015) Alert ("Incorrect price in the request");
    if (check_result.retcode == 10016) Alert ("Wrong stops in the request");
    if (check_result.retcode == 10019) Alert ("There is not enough money to execute the request");
    return;
}
```

After checking the MqlTradeRequest structure, you can send a request to perform a trade operation to a broker using the OrderSend function.

```

//-----
MqlTradeRequest mrequest;
MqlTradeCheckResult check_result;
MqlTradeResult mresult;

MqlTick latest_price;
if(!SymbolInfoTick(_Symbol,latest_price))
{
    Alert("Error getting the latest quotes - :",GetLastError(),"!!");
    return;
}
if(TradeSignalBuy==true&&BuyOpened==false){
if(((ENUM_SYMBOL_TRADE_EXECUTION)SymbolInfoInteger(Symbol(),SYMBOL_TRADE_EXEMODE))==SYMBOL_TRADE_EXECUTION_INSTANT){
ZeroMemory(mrequest);
mrequest.action = TRADE_ACTION_DEAL;
mrequest.symbol = _Symbol;
mrequest.volume = Lot;
mrequest.price = NormalizeDouble(latest_price.ask,_Digits);
mrequest.sl = NormalizeDouble(latest_price.bid - 0.01,_Digits);
mrequest.tp = NormalizeDouble(latest_price.ask + 0.01,_Digits);
mrequest.deviation=10;
mrequest.type = ORDER_TYPE_BUY;
mrequest.type_filling = ORDER_FILLING_FOK;
}
}

```

Here we get latest prices, then check the buy flag and the flag for an open position.

Then we fill in the structure of the MqlTradeRequest trade request.

The ORDER_FILLING_FOK means that the order can be executed exclusively in the specified volume.

```

ZeroMemory(check_result);
ZeroMemory(mresult);
if(!OrderCheck(mrequest,check_result))
{
    if (check_result.retcode == 10014) Alert ("Invalid volume in the request");
    if (check_result.retcode == 10015) Alert ("Incorrect price in the request");
    if (check_result.retcode == 10016) Alert ("Wrong stops in the request");
    if (check_result.retcode == 10019) Alert ("There is not enough money to execute the request");
    return;
} else{      if(OrderSend(mrequest,mresult)){
if(mresult.retcode==10009 || mresult.retcode==10008) //request executed or order placed successfully
{
Print("Price ", mresult.price);
} else{      if(mresult.retcode==10004) //Requote
{
Print("Requote bid ",mresult.bid);
Print("Requote ask ",mresult.ask);
} else{
Print("Retcode ",mresult.retcode);
} }
} else{
Print("Retcode ",mresult.retcode);
}}}}

```

Next, we check the structure of MqlTradeRequest and send a request for a trade operation to a broker using the OrderSend function.

```
if(TradeSignalSell==true&&SellOpened==false){

if(((ENUM_SYMBOL_TRADE_EXECUTION)SymbolInfoInteger(Symbol(),SYMBOL_TRADE_EXEMODE))==SYMBOL_TRADE_EXECUTION_INSTANT)

ZeroMemory(mrequest);
mrequest.action = TRADE_ACTION_DEAL;
mrequest.symbol = _Symbol;
mrequest.volume = Lot;
mrequest.price = NormalizeDouble(latest_price.ask,_Digits);
mrequest.sl = NormalizeDouble(latest_price.ask + 0.01,_Digits);
mrequest.tp = NormalizeDouble(latest_price.bid - 0.01,_Digits);
mrequest.deviation=10;
mrequest.type = ORDER_TYPE_SELL;
mrequest.type_filling = ORDER_FILLING_FOK;
```

We do the same for opening a sell position.

We check the flag for sale and the flag for an open position.

Then we fill in the structure of the MqlTradeRequest trade request.

```

ZeroMemory(check_result);
ZeroMemory(mresult);
if(!OrderCheck(mrequest,check_result))
{
    if (check_result.retcode == 10014) Alert ("Invalid volume in the request");
    if (check_result.retcode == 10015) Alert ("Incorrect price in the request");
    if (check_result.retcode == 10016) Alert ("Wrong stops in the request");
    if (check_result.retcode == 10019) Alert ("There is not enough money to execute the request");
    return;
}
else{
if(OrderSend(mrequest,mresult)){
if(mresult.retcode==10009 || mresult.retcode==10008) //request executed or order placed successfully
{
Print("Price ", mresult.price);
} else {
if(mresult.retcode==10004) //Реквота
{
Print("Requote bid ",mresult.bid);
Print("Requote ask ",mresult.ask);
}else{ Print("Retcode ",mresult.retcode);
} }
}else{ Print("Retcode ",mresult.retcode);
}} } }

```

Next, we check the structure of MqlTradeRequest and send a request for a trade operation to a broker using the OrderSend function.

Thus, a request for a trade operation is sent if there is a signal to open a position and at the same time an open position does not exist yet.

After OrderCheck checking, the MqlTradeRequest structure is re-checked as a return value of the OrderSend function.

Next, it checks a result code of the operation of the MqlTradeResult structure. And I personally did not meet the execution of an order Request Execution with brokers.

Instead of the Instant Execution, a broker can offer order execution Market Execution or Exchange Execution.

In the Market Execution mode, a deal is made at the price offered by a broker, and there are no requotes.

In the Exchange Execution mode, trading operations are allegedly transferred to an external trading system and deals are executed at current market prices, while requotes are also absent.

```
//-----
if(((ENUM_SYMBOL_TRADE_EXECUTION)SymbolInfoInteger(Symbol(),SYMBOL_TRADE_EXEMODE))==SYMBOL_TRADE_EXECUTION_EXCHANGE){
//или SYMBOL_TRADE_EXECUTION_MARKET

ZeroMemory(mrequest);
mrequest.action = TRADE_ACTION_DEAL;
mrequest.symbol = _Symbol;
mrequest.volume = Lot;
mrequest.type = ORDER_TYPE_BUY;
mrequest.type_filling = ORDER_FILLING_FOK;
```

When using Market Execution or Exchange Execution, the fields of the structure MqlTradeRequest - action, symbol, volume, type, and type_filling - are mandatory.

```

ZeroMemory(check_result);
ZeroMemory(mresult);
if(!OrderCheck(mrequest,check_result))
{
    if (check_result.retcode == 10014) Alert ("Invalid volume in the request");
    if (check_result.retcode == 10019) Alert ("There is not enough money to execute the request");
    return;
}else{
if(OrderSend(mrequest,mresult)){
if(mresult.retcode==10009 || mresult.retcode==10008) //request executed or order placed successfully
{
//-----
ZeroMemory(mrequest);
mrequest.action = TRADE_ACTION_SLTP;
mrequest.symbol = _Symbol;
mrequest.sl = NormalizeDouble(mresult.price - 0.01,_Digits);
mrequest.tp = NormalizeDouble(mresult.price + 0.01,_Digits);
ZeroMemory(check_result);
ZeroMemory(mresult);
}
}
}

```

After filling in the MqlTradeRequest structure, we check it and send a request to a broker.

Next, we form a new request in which we set the Stop Loss and Take Profit values for the open position, based on prices received from a broker.

```
if(!OrderCheck(mrequest,check_result))
{
if (check_result.retcode == 10015) Alert ("Incorrect price in the request");
if (check_result.retcode == 10016) Alert ("Wrong stops in the request");
return;
}else{
if(OrderSend(mrequest,mresult)){
if(mresult.retcode==10009 || mresult.retcode==10008) //request executed or order placed successfully
{
Print("SL ", mrequest.sl, "TP ",mrequest.tp);
}
else { Print("Retcode ",mresult.retcode);
}
}else{ Print("Retcode ",mresult.retcode);
}} }else{ Print("Retcode ",mresult.retcode);
}} }
```

After filling in the new structure MqlTradeRequest, we check it and send a new request to a broker.

```
//-----
if(((ENUM_SYMBOL_TRADE_EXECUTION)SymbolInfoInteger(Symbol(),SYMBOL_TRADE_EXEMODE)==SYMBOL_TRADE_EXECUTION_EXCHANGE){
//или SYMBOL_TRADE_EXECUTION_MARKET

ZeroMemory(mrequest);
mrequest.action = TRADE_ACTION_DEAL;
mrequest.symbol = _Symbol;
mrequest.volume = Lot;
mrequest.type = ORDER_TYPE_SELL;
mrequest.type_filling = ORDER_FILLING_FOK;
```

We do the same when opening a sell position.
We fill in the MqlTradeRequest structure.

```

ZeroMemory(check_result);
ZeroMemory(mresult);

if(!OrderCheck(mrequest,check_result))
{
    if (check_result.retcode == 10014) Alert ("Invalid volume in the request");
    if (check_result.retcode == 10019) Alert ("There is not enough money to execute the request");
    return;
}else{
if(OrderSend(mrequest,mresult)){
if(mresult.retcode==10009 || mresult.retcode==10008) //request executed or order placed
successfully
{

//-----
ZeroMemory(mrequest);
mrequest.action = TRADE_ACTION_SLTP;
mrequest.symbol = _Symbol;
mrequest.tp = NormalizeDouble(mresult.price - 0.01,_Digits);
mrequest.sl = NormalizeDouble(mresult.price + 0.01,_Digits);
ZeroMemory(check_result);
ZeroMemory(mresult);
}
}

```

And after filling in the MqlTradeRequest structure, we check it and send a request to a broker.

Next, we form a new request in which we set the Stop Loss and Take Profit values for the open position, based on prices received from a broker.

```

if(!OrderCheck(mrequest,check_result))
{
    if (check_result.retcode == 10015) Alert ("Incorrect price in the request");
    if (check_result.retcode == 10016) Alert ("Wrong stops in the request");
    return;
}
else{
    if(OrderSend(mrequest,mresult)){
        if(mresult.retcode==10009 || mresult.retcode==10008) //request executed or order placed successfully
        {
            Print("SL ", mrequest.sl, "TP ",mrequest.tp);
        }
        else{ Print("Retcode ",mresult.retcode);
        }
    }
    else{ Print("Retcode ",mresult.retcode);
    }
}
else{ Print("Retcode ",mresult.retcode);
}
}
}

```

After filling in the new structure `MqlTradeRequest`, we check it and send a new request to a broker.

Thus, a position is opened here without determining `StopLoss` and `TakeProfit`, and then, in the case of successful execution of the request, a trade order is placed to modify the `StopLoss` and `TakeProfit` levels.

ORDER_TYPE_BUY_LIMIT - pending order to buy, with the current prices above the price of the order.

ORDER_TYPE_SELL_LIMIT - pending order to sell, with the current prices below the price of the order.

ORDER_TYPE_BUY_STOP - pending order to buy, with the current prices below the price of the order.

ORDER_TYPE_SELL_STOP - pending order to buy, with the current prices above the price of the order.

ORDER_TYPE_BUY_STOP_LIMIT - deferred placing a pending order of type Buy Limit to trade on a rollback, while the current prices are lower than the price of the order.

ORDER_TYPE_SELL_STOP_LIMIT - deferred placing a pending order of type Sell Limit to trade on a rollback, while the current prices are higher than the price of the order.

To place a pending order to buy or sell (TRADE_ACTION_PENDING), 11 fields of the MqlTradeRequest structure are required: action, symbol, volume, price, stoplimit, sl, tp, type, type_filling, type_time, and expiration.

In this case, the type field can take the following values.

- ORDER_TYPE_BUY_LIMIT - it is a pending order to buy, while current prices are higher than the price of the order.
- ORDER_TYPE_SELL_LIMIT - it is a pending order to sell, while current prices are lower than the price of the order.
- ORDER_TYPE_BUY_STOP - it is a pending order to buy, while current prices are lower than the price of the order.
- ORDER_TYPE_SELL_STOP - it is a pending order to sell, with current prices higher than a price of the order.
- ORDER_TYPE_BUY_STOP_LIMIT - it is the deferred placing of a pending order of the Buy Limit type for trading on a rollback, while current prices are lower than a price of the order.
- ORDER_TYPE_SELL_STOP_LIMIT - it is deferred placing a pending order of the Sell Limit type to trade on a rollback, while current prices are higher than a price of the order.

To change parameters of a pending order (TRADE_ACTION MODIFY),

you need to specify 7 fields of the MqlTradeRequest structure: action, order, price, sl, tp, type_time, and expiration.

In this case, the value of the order field is taken from the MqlTradeResult structure of a result of placing an order.

To remove a pending order (TRADE_ACTION_REMOVE), 2 fields of the MqlTradeRequest structure are required: action and order.

Example of Creating Advisor

As the basis of an adviser, we take the following code.

```

input double    Lot=1;
input int       EA_Magic=1000;
input double   spreadLevel=5.0;
input double   StopLoss=0.01;
input double   Profit=0.01;

bool flagStopLoss=false;

int OnCheckTradeInit(){
//Check expert start on a real account
if((ENUM_ACCOUNT_TRADE_MODE)AccountInfoInteger(ACCOUNT_TRADE_MODE)==ACCOUNT_TRADE_MODE_REAL){
    int mb=MessageBox("Run the advisor on a real account?","Message Box",MB_YESNO|MB_ICONQUESTION);
    if(mb==IDNO) return(0);
}
//Checking:
//No connection to the server, prohibition of trade on the server side
//The broker prohibits automatic trading

if(!TerminalInfoInteger(TERMINAL_CONNECTED)){
Alert("No connection to the trade server");
return(0);
}
if(!AccountInfoInteger(ACCOUNT_TRADE_ALLOWED)){
Alert("Trade for this account is prohibited");
return(0);
}
}

```

In this code, firstly, we define a volume of trade in lots, a maximum allowable broker spread, in which we agree to trade, and values of the stop loss and take profit for our trade, which can be optimized later.

We also define the flagStopLoss flag in order to stop trading if we catch a stop loss.

Here, the general checks of the OnInit function are picked out in the separate OnCheckTradeInit function, and the general checks of the OnTick function are picked out in the separate OnCheckTradeTick function.

```

int OnCheckTradeInit(){
//Check expert start on a real account
if((ENUM_ACCOUNT_TRADE_MODE)AccountInfoInteger(ACCOUNT_TRADE_MODE)==ACCOUNT_TRADE_MODE_REAL){
    int mb=MessageBox("Run the advisor on a real account?","Message Box",MB_YESNO|MB_ICONQUESTION);
    if(mb==IDNO) return(0);
}
//Checking:
//No connection to the server, prohibition of trade on the server side
//The broker prohibits automatic trading

if(!TerminalInfoInteger(TERMINAL_CONNECTED)){
Alert("No connection to the trade server");
return(0);
}
if(!AccountInfoInteger(ACCOUNT_TRADE_ALLOWED)){
Alert("Trade for this account is prohibited");
return(0);
}
if(!AccountInfoInteger(ACCOUNT_TRADE_EXPERT)){
Alert("Trade with the help of experts for the account is prohibited");
return(0);
}
}
//Check the correctness of the volume with which we are going to enter the market
if(Lot<SymbolInfoDouble(Symbol(),SYMBOL_VOLUME_MIN)||Lot>SymbolInfoDouble(Symbol(),SYMBOL_VOLUME_MAX)){
Alert("Lot is not correct!!!");
return(0);
}
return(INIT_SUCCEEDED);
}

```

In the OnCheckTradeInit function, we ask a user for permission to launch an expert on a real account, and then check a connection to a server, and also, whether trading is prohibited on a server side, and whether a broker has prohibited automatic trading.

```
48 //+-----+
49 //| Expert initialization function
50 //+-----+
51 int OnInit()
52 {
53     return(OnCheckTradeInit());
54 }
55 //+-----+
56 //| Expert deinitialization function
57 //+-----+
58 void OnDeinit(const int reason)
59 {
60 }
```

Accordingly, in the OnInit function, we simply call our OnCheckTradeInit function.

```

int OnCheckTradeTick(){
//Check there is no connection to the server
if(!TerminalInfoInteger(TERMINAL_CONNECTED)){
Alert("No connection to the trade server");
return(0);
}
//Is the auto-trade button on the client terminal enabled?
if (!TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)){
Alert("Auto Trade Permission Off!");
return(0);
}
//Permission to trade with the help of an expert is disabled in the common properties of the expert itself.
if(!MQLInfoInteger(MQL_TRADE_ALLOWED)){
Alert("Automatic trading is prohibited in the properties of the expert ",__FILE__);
return(0);
}
//Margin level at which account replenishment is required
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_PERCENT){
if(AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)!=0&&AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)
<=AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)){
Alert("Margin Call!!!");
return(0);
}}
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_MONEY){
if(AccountInfoDouble(ACCOUNT_EQUIITY)<=AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)){
Alert("Margin Call!!!");
return(0);
}}

```

In the OnCheckTradeTick function that performs general checks for the OnTick function, we check a connection to a server, and, also, whether the auto-trading button is enabled in the client terminal, whether the trading permission is enabled with the help of an expert in the general properties of the expert itself, and the occurrence of a Margin Call event.

```

//Margin level upon reaching which there is a forced closing of the most unprofitable position
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_PERCENT){
if(AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)!=0&&AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)
<=AccountInfoDouble(ACCOUNT_MARGIN_SO_SO)){
Alert("Stop Out!!!");
return(0);
}
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_MONEY){
if(AccountInfoDouble(ACCOUNT_EQUITY)<=AccountInfoDouble(ACCOUNT_MARGIN_SO_SO)){
Alert("Stop Out!!!");
return(0);
}
//Checking the free funds volume on the account available for opening a position
double margin;
MqlTick last_tick;
ResetLastError();
if(SymbolInfoTick(Symbol(),last_tick))
{
if(OrderCalcMargin(ORDER_TYPE_BUY,Symbol(),Lot,last_tick.ask,margin))
{
if(margin>AccountInfoDouble(ACCOUNT_MARGIN_FREE)){
Alert("Not enough money in the account!");
return(0);
}
}
else
{
Print(GetLastError());
}
}

```

Next, we check the occurrence of a Stop Out event and whether there are enough available funds on the account to open a position.

```

//Broker spread control
double _spread=
SymbolInfoInteger(Symbol(),SYMBOL_SPREAD)*MathPow(10,-SymbolInfoInteger(Symbol(),SYMBOL_DIGITS))/MathPow(10,-4);
if(_spread>spreadLevel){
Alert("Too big spread!");
return(0);
}
//Check of restrictions on trading operations on the symbol set by the broker
if((ENUM_SYMBOL_TRADE_MODE)SymbolInfoInteger(Symbol(),SYMBOL_TRADE_MODE)!=SYMBOL_TRADE_MODE_FULL){
Alert("Restrictions on trading operations!");
return(0);
}
//Are there enough bars in the history to calculate the advisor
if(Bars(Symbol(), 0)<100)
{
Alert("In the chart little bars, Expert will not work!!!");
return(0);
}

return(1);
}

```

And then, we control a broker's spread, check restrictions on trading operations for the symbol set by a broker, and check if there are enough bars in the history to calculate the advisor.

```

//+
//| Expert tick function
//+
void OnTick()
{
    if(!OnCheckTradeTick()){
        return;
    }
    //Limit the calculations of the adviser by the appearance of a new bar on the chart
    static datetime last_time;
    datetime last_bar_time=(datetime)SeriesInfoInteger(Symbol(),Period(),SERIES_LASTBAR_DATE);
    if(last_time!=last_bar_time)
    {
        last_time=last_bar_time;
    }else{
        return;
    }
    //Limit adviser calculation on flagStopLoss
    static datetime last_time_daily;
    datetime last_bar_time_daily=(datetime)SeriesInfoInteger(Symbol(),PERIOD_D1,SERIES_LASTBAR_DATE);
    if(last_time_daily!=last_bar_time_daily)
    {
        last_time_daily=last_bar_time_daily;
        flagStopLoss=false;
    }
    if(flagStopLoss==true)return;
}

```

Accordingly, in the OnTick function, we first call the OnCheckTradeTick function to perform all the checks.

Then we limit the work of the expert on the appearance of a new bar on a symbol's chart so that the adviser does not work with every tick.

For this, we use the static variable last_time, in which we record time of the appearance of the last bar.

Next, we limit daily trading when receiving a stop loss.

This step is, of course, optional and depends on the trader's strategy.

We present it exclusively to demonstrate usage of the OnTrade function.

```

//Check for an open position so as not to try to re-open it
    bool BuyOpened=false;
    bool SellOpened=false;
    if(PositionSelect(_Symbol)==true)
    {
        if(PositionGetInteger(POSITION_TYPE)==POSITION_TYPE_BUY)
        {
            BuyOpened=true;
        }
        else if(PositionGetInteger(POSITION_TYPE)==POSITION_TYPE_SELL)
        {
            SellOpened=true;
        }
    }
    //To calculate the signals of the trading system requires symbol's historical data
    int num;
    if(numberBarOpenPosition>numberBarStopPosition)num=numberBarOpenPosition;
    if(numberBarOpenPosition<=numberBarStopPosition)num=numberBarStopPosition;
    MqlRates mrate[];
    ResetLastError();
    if(CopyRates(Symbol(),Period(),0,num,mrate)<0)
    {
        Print(GetLastError());
        return;
    }

    ArraySetAsSeries(mrate,true);

    bool TradeSignalBuy=false;
    bool TradeSignalSell=false;

```

Then we check the presence of an open position in order not to try to re-open it, using the BuyOpened and SellOpened flags.

And we get a price of a symbol to calculate signals of a trading system.

Prices are recorded in the structure MqlRates.

And the advisor will send buy and sell orders when setting the TradeSignalBuy and TradeSignalSell flags to true.

Setting values of the flags TradeSignalBuy and TradeSignalSell should be performed by a trading system.

```

MqlTradeRequest mrequest;
MqlTradeCheckResult check_result;
MqlTradeResult mresult;

MqlTick latest_price;
if(!SymbolInfoTick(_Symbol,latest_price))
{
    Alert("Error getting the latest quotes - : ",GetLastError(),"!!");
    return;
}
if(TradeSignalBuy==true&&buyOpened==false){

if((ENUM_SYMBOL_TRADE_EXECUTION)SymbolInfoInteger(Symbol(),SYMBOL_TRADE_EXEHODE)==SYMBOL_TRADE_EXECUTION_INSTANT){
ZeroMemory(mrequest);
mrequest.action = TRADE_ACTION_DEAL;
mrequest.symbol = _Symbol;
mrequest.volume = Lot;
mrequest.price = NormalizeDouble(latest_price.ask,_Digits);
mrequest.sl = NormalizeDouble(latest_price.bid - Stoploss,_Digits);
mrequest.tp = NormalizeDouble(latest_price.ask + Profit,_Digits);
mrequest.deviation=10;
mrequest.type = ORDER_TYPE_BUY;
mrequest.type_filling = ORDER_FILLING_FOK;

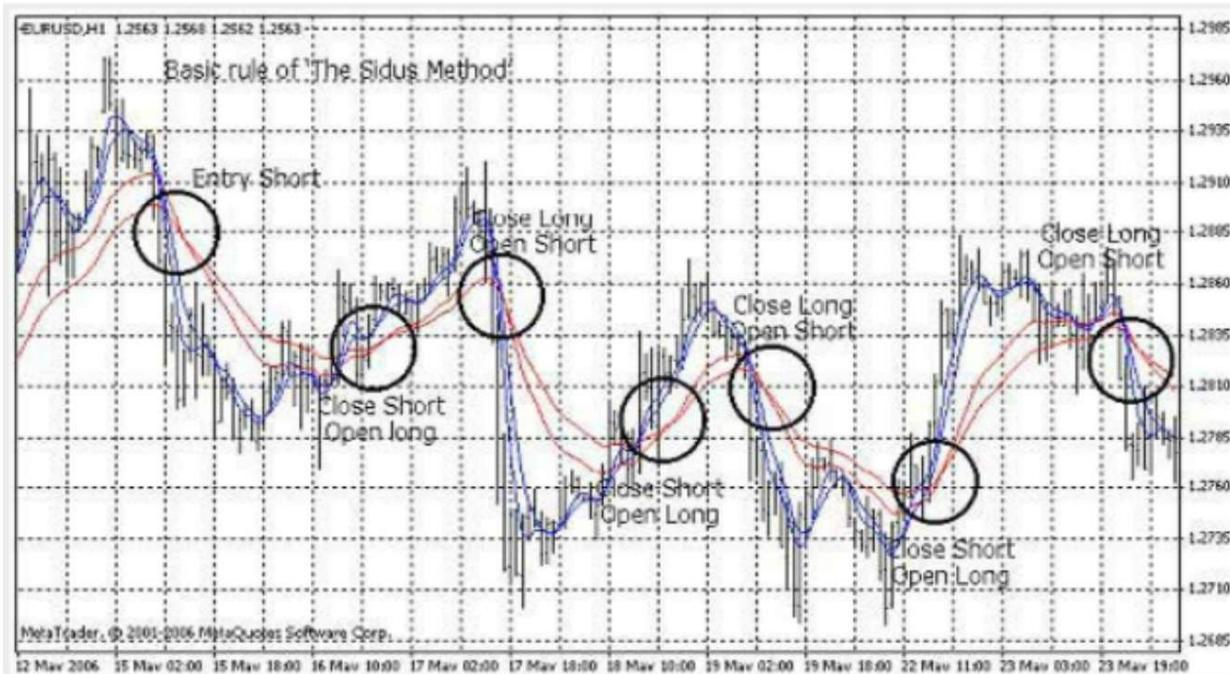
ZeroMemory(check_result);
ZeroMemory(mresult);
if(!OrderCheck(mrequest,check_result))
{
    if (check_result.retcode == 10014) Alert ("Invalid volume in the request");
    if (check_result.retcode == 10015) Alert ("Incorrect price in the request");
    if (check_result.retcode == 10016) Alert ("Wrong stops in the request");
    if (check_result.retcode == 10019) Alert ("There is not enough money to execute the request");
    return;
}
}
}

```

Further, when receiving signals from a trading system, we send orders to buy or sell.

```
423 void OnTrade()
424 {
425 static int _deals;
426 ulong _ticket=0;
427
428 if(HistorySelect(0,TimeCurrent()))
429 {
430 int i=HistoryDealsTotal()-1;
431
432 if(_deals!=i) {
433 _deals=i;
434 } else { return; }
435
436 if({_ticket=HistoryDealGetTicket(i)})>0
437 {
438 string _comment=HistoryDealGetString(_ticket,DEAL_COMMENT);
439 if(StringFind(_comment,"sl",0)!=-1) {
440 flagStopLoss=true;
441 }
442 }
443 }
444 }
```

Finally, in the OnTrade function, we process a stop loss event and set the flagStopLoss flag to true.



As a trading system, we take the "Sidus Method":

Here, the time interval is taken - H1 - an hour.

And we use the Exponential Moving Average: 18 EMA and 28 EMA;

As well as the Weighted Moving Average - 5WMA and 8 WMA.

And we consider the following trading signals to enter the market using the Sidus Method:

- to open a buy position: the 5WMA and 8WMA moving averages cross the 18EMA and 28EMA tunnel from bottom to up.
- to open a sell position: the 5WMA and 8 WMA moving averages cross the 18 EMA and 28 EMA tunnel from top to bottom.

And we consider the following trading signals to exit the market according to the Sidus Method:

- for buy: a price on a chart has reached the top and the 5 WMA moving average seems to be "diving" under the 8 WMA moving average. It should close an open trading position.
- for sale: a price on a chart has reached the bottom and the 5 WMA moving average seems to be "jumping" over the 8WMA moving average. It should close the trading position.

```

18 int      handleIMA18;
19 double    MA18Buffer[];
20 int      handleIMA28;
21 double    MA28Buffer[];
22 int      handleIWMA5;
23 double    WMA5Buffer[];
24 int      handleIWMA8;
25 double    WMA8Buffer[];

57 //-----+
58 //| Expert initialization function
59 //+-----+
60 int OnInit()
61 {
62 handleIMA18=iMA(_Symbol,PERIOD_H1,18,0,MODE_EMA,PRICE_CLOSE);
63 handleIMA28=iMA(_Symbol,PERIOD_H1,28,0,MODE_EMA,PRICE_CLOSE);
64 handleIWMA5=iMA(_Symbol,PERIOD_H1,5,0,MODE_LWMA,PRICE_CLOSE);
65 handleIWMA8=iMA(_Symbol,PERIOD_H1,8,0,MODE_LWMA,PRICE_CLOSE);
66
67 return(OnCheckTradeInit());
68 }

```

Now we add the code of the adviser with the implementation of the Sidus Method.

Let's declare handles of the used indicators and their buffers.

And in the OnInit function, we get these handles.

```
212 bool TradeSignalBuy=false;
213 bool TradeSignalSell=false;
214
215 TradeSignalBuy=OnTradeSignalBuy();
216 TradeSignalSell=OnTradeSignalSell();
217
218 bool TradeSignalBuyStop=false;
219 bool TradeSignalSellStop=false;
220
221
222 TradeSignalBuyStop=OnTradeSignalBuyStop(mrate);
223 TradeSignalSellStop=OnTradeSignalSellStop(mrate);
```

In the OnTick functions, we introduce the new OnTradeSignalBuy and OnTradeSignalSell functions to calculate buy and sell signals.

And we introduce the new OnTradeSignalBuyStop and OnTradeSignalSellStop functions for calculating signals to close buy and sell positions.

```

31 #include <Trade\PositionInfo.mqh>
32 #include <Trade\Trade.mqh>
33 CPositionInfo m_position;           // trade position object
34 CTrade      m_trade;              // trading object

453 //-----
454 if(TradeSignalSellStop==true&&SellOpened==true){
455     for(int i=PositionsTotal()-1;i>0;i--) // returns the number of current positions
456     if(m_position.SelectByIndex(i))    // selects the position by index for further access to its properties
457     {
458         ENUM_POSITION_TYPE type = m_position.PositionType();
459         if(type==POSITION_TYPE_SELL)
460             m_trade.PositionClose(m_position.Ticket()); // close a position by the specified symbol
461     }
462 }
463 //-----
464 if(TradeSignalBuyStop==true&&BuyOpened==true){
465     for(int i=PositionsTotal()-1;i>0;i--) // returns the number of current positions
466     if(m_position.SelectByIndex(i))    // selects the position by index for further access to its properties
467     {
468         ENUM_POSITION_TYPE type = m_position.PositionType();
469         if(type==POSITION_TYPE_BUY)
470             m_trade.PositionClose(m_position.Ticket()); // close a position by the specified symbol
471     }
472 }
473 }
```

Thus, we should complement the OnTick function's code by closing a buy and sell position.

And at the very beginning of the appearance of MQL5, a position is closed by sending an opposite order with the same volume, that is, closing a buy position was done by sending a sell order with the same volume, and vice versa, to close a sell position.

That is, the MetaTrader 5 platform was originally created for stock trading with netting position accounting.

When netting accounting for one financial instrument, you can have only one position, so all further operations on it lead to a change in volume, closing or reversal of the existing position.

But in order to expand the opportunities of traders, a second accounting system was added to the platform - hedging.

Now a financial instrument can have many positions, including multidirectional positions.

This allows you to implement trading strategies with the so-called locking - if a price went against a trader, it is possible to open a position in the opposite direction.

Therefore, to reliably close a position, we can use the CPositionInfo and

CTrade library classes.

Therefore, we will include the files of these classes in the code and create instances of them.

And in the OnTick function, we check all open positions in a loop and use the PositionClose method of the CTrade class to close positions of a certain type.

```

16 input int numberBarOpenPosition=5;
17 input int numberBarStopPosition=5;

503 bool OnTradeSignalBuy()
504 {
505     if(CopyBuffer(handleIMA18, 0, 0, numberBarOpenPosition, MA18Buffer)<0)
506     {
507         return false;
508     }
509     if(CopyBuffer(handleIMA28, 0, 0, numberBarOpenPosition, MA28Buffer)<0)
510     {
511         return false;
512     }
513     if(CopyBuffer(handleIWMA5, 0, 0, numberBarOpenPosition, WMA5Buffer)<0)
514     {
515         return false;
516     }
517     if(CopyBuffer(handleIWMA8, 0, 0, numberBarOpenPosition, WMA8Buffer)<0)
518     {
519         return false;
520     }
521     ArraySetAsSeries(MA18Buffer, true);
522     ArraySetAsSeries(MA28Buffer, true);
523     ArraySetAsSeries(WMA5Buffer, true);
524     ArraySetAsSeries(WMA8Buffer, true);

```

And of course, we should define the functions of trading system signals, since the `OnTick` method calls the `OnTradeSignalBuy`, `OnTradeSignalSell`, `OnTradeSignalBuyStop`, and `OnTradeSignalSellStop` functions to receive signals for sale or purchase.

But firstly, before the callback functions, we declare the input parameters - `numberBarOpenPosition` — a number of bars on which we check the intersection of 5WMA and 8 WMA with the 18 EMA and 28 EMA tunnel, and `numberBarStopPosition` — a number of bars on which we check the 5WMA and 8 WMA intersection and reaching a price top or bottom.

In the `OnTradeSignalBuy` and `OnTradeSignalSell` functions, using indicator handles, we fill dynamic arrays of indicator values.

```

526 bool flagCross1=false;
527 bool flagCross2=false;
528 bool flagCross=false;
529
530 if(WMA5Buffer[1]>MA18Buffer[1]&&WMA5Buffer[1]>MA28Buffer[1]
531 &&WMA8Buffer[1]>MA18Buffer[1]&&WMA8Buffer[1]>MA28Buffer[1]){
532 for (int i=2;i<numberBarOpenPosition;i++){
533 if(WMA5Buffer[i]<MA18Buffer[i]&&WMA5Buffer[i]<MA28Buffer[i]){
534 flagCross1=true;
535 }
536 if(WMA8Buffer[i]<MA18Buffer[i]&&WMA8Buffer[i]<MA28Buffer[i]){
537 flagCross2=true;
538 }
539 }
540 if(flagCross1==true&&flagCross2==true){
541 flagCross=true;
542 }
543 }
544 flagBuy=flagCross;
545 return flagBuy;
546 }

```

And on the number of bars `numberBarOpenPosition`, we check the intersection of the 5WMA and 8 WMA with the 18 EMA and 28 EMA tunnel.

```

548 bool OnTradeSignalBuyStop(MqlRates& mrate[]){
549     bool flagBuyStop=false;
550     if(CopyBuffer(handleIWMA5,0,0,numberBarStopPosition,WMA5Buffer)<0)
551     {
552         return false;
553     }
554     if(CopyBuffer(handleIWMA8,0,0,numberBarStopPosition,WMA8Buffer)<0)
555     {
556         return false;
557     }
558     ArraySetAsSeries(WMA5Buffer,true);
559     ArraySetAsSeries(WMA8Buffer,true);
560     bool flagCross=false;
561     if(WMA5Buffer[1]<WMA8Buffer[1]){
562         for (int i=2;i<numberBarStopPosition;i++){
563             if(WMA5Buffer[i]>WMA8Buffer[i]){
564                 flagCross=true;
565             }
566         }
567         double max=mrate[1].high;
568         for (int i=1;i<numberBarStopPosition;i++){
569             if(mrate[i].high>max)max=mrate[i].high;
570         }
571         if(flagCross==true&&mrate[1].high<=max&&mrate[numberBarStopPosition-1].high<=max){
572             flagBuyStop=true;
573         }
574     }

```

In the `OnTradeSignalBuyStop` and `OnTradeSignalSellStop` functions, with the help of indicator handles, we fill dynamic arrays of indicator values and on the number of bars `numberBarStopPosition`, we check the intersection of the 5WMA and 8 WMA and when a price reaches the top or bottom.

After compiling the advisor in the client terminal, let's right-click on the advisor and select Test.

The screenshot shows the 'Optimize Expert' interface with the following configuration:

- Expert:** ExpSidus.ex5
- Date:** Last year (2019.01.01 - 2019.05.03)
- Forward:** No (2019.01.26)
- Execution:** Without Delay (1 minute OHLC)
- Deposit:** 10000 USD (1:100)
- Visualization:** Enabled
- Optimization:** Fast genetic based algorithm (Balance max)

Optimization progress: [Progress Bar] Start

Inputs Tab (Visible):

Variable	Value	Start	Step	Stop	Steps
Lot	1	1	0.1	10.0	
EA_Magic	1000	1000	1	10000	
spreadLevel	5	5	0.5	50.0	
StopLoss	0.01	0.01	0.001	0.1	
Profit	0.01	0.01	0.001	0.1	
numberBarOpenPosition	10	3	1	15	13
numberBarStopPosition	10	3	1	15	13

Inputs Tab (Bottom): Settings | Inputs | Agents | Journal |

And let's try to optimize the numberBarOpenPosition and numberBarStopPosition parameters.

Pass	Result ▾	Profit	Total trades	numberBarOpenPosition	numberBarStopPosition
22	12214.45	2214.45	131	12	4
9	12214.45	2214.45	131	12	3
50	11039.25	1039.25	135	14	6
20	10650.09	650.09	98	10	4
7	10650.09	650.09	98	10	3
49	10558.13	558.13	134	13	6
36	10558.13	558.13	134	13	5
21	10179.53	179.53	123	11	4
8	10179.53	179.53	123	11	3
156	10152.69	152.69	12	3	15
143	10152.69	152.69	12	3	14
130	10152.69	152.69	12	3	13
117	10152.69	152.69	12	3	12
104	10152.69	152.69	12	3	11
91	10152.69	152.69	12	3	10

Settings | Inputs | Optimization Results | Optimization Graph | Agents | Journal |

As a result of optimization, we obtain the maximum profit when the value is the numberBarOpenPosition = 12, and when the value is the numberBarStopPosition = 4.

When we change the indicator price parameter on the PRICE_WEIGHTED, the profit improves.

`MathAbs(WMA5Buffer[1]-WMA5Buffer[numberBarStopPosition-1])<0.001`

And we can replace a condition for the market to reach the bottom or top by the horizontal position of the EMA line, as you can see on the slide.

And you can remove the effect of signals `TradeSignalBuyStop` and `TradeSignalSellStop` and work only on achieving Take Profit or Stop Loss.

Example of Creating Advisor using OOP

Let's take the previous example of creating an adviser and transfer the code of the functions OnCheckTradeInit, OnCheckTradeTick, OnTradeSignalBuy, OnTradeSignalBuyStop, OnTradeSignalSell, OnTradeSignalSellStop, as well as the code for opening and closing positions into separate classes.

```
12 class CheckTrade
13 {
14 private:
15
16 public:
17     CheckTrade();
18     ~CheckTrade();
19 int     OnCheckTradeInit(double    lot);
20 int     OnCheckTradeTick(double    lot, double spread);
21 };
22 //+
23 //|
24 //+
25 CheckTrade::CheckTrade()
26 {
27 }
28 //+
29 //|
30 //+
31 CheckTrade::~CheckTrade()
32 {
33 }
```

We transfer the code of the checking functions OnCheckTradeInit and OnCheckTradeTick to the CheckTrade class.

In this class, we declare two methods - OnCheckTradeInit and OnCheckTradeTick.

```

int CheckTrade::OnCheckTradeInit(double lot){
    if((ENUM_ACCOUNT_TRADE_MODE)AccountInfoInteger(ACCOUNT_TRADE_MODE)==ACCOUNT_TRADE_MODE_REAL){
        int mb=MessageBox("Run the advisor on a real account?","Message Box",MB_YESNO|MB_ICONQUESTION);
        if(mb==IDNO) return(0);
    }
    if(!TerminalInfoInteger(TERMINAL_CONNECTED)){
        Alert("No connection to the trade server");
        return(0);
    }else{
        if(!AccountInfoInteger(ACCOUNT_TRADE_ALLOWED)){
            Alert("Trade for this account is prohibited");
            return(0);
        }
        if(!AccountInfoInteger(ACCOUNT_TRADE_EXPERT)){
            Alert("Trade with the help of experts for the account is prohibited");
            return(0);
        }
        if(lot<SymbolInfoDouble(Symbol(),SYMBOL_VOLUME_MIN)||lot>SymbolInfoDouble(Symbol(),SYMBOL_VOLUME_MAX)){
            Alert("Lot is not correct!!!");
            return(0);
        }
    }
    return(INIT_SUCCEEDED);
}

```

In the class method OnCheckTradeInit, we ask a trader to run an expert on a real account.

Then we check a connection to a server, an absence of a ban on trading on a server side, and an absence of a broker ban on automated trading.

Finally, we verify the correctness of a volume with which we are going to enter the market.

```

int CheckTrade::OnCheckTradeTick(double    lot,double spread){
if(!TerminalInfoInteger(TERMINAL_CONNECTED)){
Alert("No connection to the trade server");
return(0);
}
if (!TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)){
Alert("Auto Trade Permission Off!");
return(0);
}
if(!MQLInfoInteger(MQL_TRADE_ALLOWED)){
Alert("Automatic trading is prohibited in the properties of the expert ",__FILE__);
return(0);
}
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_PERCENT){
if(AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)!=0&&AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)
<=AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)){
Alert("Margin Call!!!");
return(0);
}}
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_MONEY){
if(AccountInfoDouble(ACCOUNT_EQUITY)<=AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)){
Alert("Margin Call!!!");
return(0);
}}

```

In the OnCheckTradeTick method, we check a connection to a server, and also, whether the auto-trade button is enabled in the client terminal, whether there is permission to trade with the help of an expert in the common properties of an expert itself, and the occurrence of a Margin Call event.

```

if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_PERCENT){
if(AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)!=0&&AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)
<=AccountInfoDouble(ACCOUNT_MARGIN_SO_SO)){
Alert("Stop Out!!!");
return(0);
}}
if((ENUM_ACCOUNT_STOPOUT_MODE)AccountInfoInteger(ACCOUNT_MARGIN_SO_MODE)==ACCOUNT_STOPOUT_MODE_MONEY){
if(AccountInfoDouble(ACCOUNT_EQUITY)<=AccountInfoDouble(ACCOUNT_MARGIN_SO_SO)){
Alert("Stop Out!!!");
return(0);
}}
double margin;
MqTick last_tick;
ResetLastError();
if(SymbolInfoTick(Symbol(),last_tick))
{
    if(OrderCalcMargin(ORDER_TYPE_BUY,Symbol(),lot,last_tick.ask,margin))
    {
        if(margin>AccountInfoDouble(ACCOUNT_MARGIN_FREE)){
            Alert("Not enough money in the account!");
            return(0);
        }
    }
    else{
        Print(GetLastError());
    }
}

```

Next, we check the occurrence of a Stop Out event and check a volume of available funds on an account available for opening a position.

```

double _spread=
SymbolInfoInteger(Symbol(),SYMBOL_SPREAD)*MathPow(10,-SymbolInfoInteger(Symbol(),SYMBOL_DIGITS))/MathPow(10,-4);
if(_spread>spread){
Alert("Too big spread!");
return(0);
}
if((ENUM_SYMBOL_TRADE_MODE)SymbolInfoInteger(Symbol(),SYMBOL_TRADE_MODE)!=SYMBOL_TRADE_MODE_FULL){
Alert("Restrictions on trading operations!");
return(0);
}
if(Bars(Symbol(), 0)<100)
{
Alert("In the chart little bars, Expert will not work!!");
return(0);
}

return(1);
}

```

And then we monitor a broker's spread, check for any restrictions on trading operations for a symbol set by a broker, and check if there are enough bars in the history to calculate the advisor.

```

16 class Trade
17 {
18 private:
19 double StopLoss;
20 double Profit;
21 double Lot;
22
23 CPositionInfo m_position;           // trade position object
24 CTrade          m_trade;            // trading object
25
26 public:
27     Trade(double stopLoss, double profit, double lot);
28     ~Trade();
29 void Order(bool Buy, bool StopBuy, bool Sell, bool StopSell);
30 }
31 //+
32 //|
33 //+
34 Trade::Trade(double stopLoss, double profit, double lot)
35 {
36     StopLoss=stopLoss;
37     Profit=profit;
38     Lot=lot;
39 }
40 //+
41 //|
42 //+
43 Trade::~Trade()
44 {
45 }
46 //+

```

And we transfer the code for opening and closing a position to the class Trade.

Here, we declare the class instance variables - a stop loss, take profit and volume of trade in lots.

We also declare the objects of the library classes CPositionInfo and CTrade, which we use to close positions.

And here, we declare the Order method to open a position.

```

Trade::Order(bool Buy, bool BuyStop, bool Sell, bool Sellstop){
    bool BuyOpened=false;
    bool SellOpened=false;

    if(PositionSelect(_Symbol)==true)
    {
        if(PositionGetInteger(POSITION_TYPE)==POSITION_TYPE_BUY)
        {
            BuyOpened=true;
        }
        else if(PositionGetInteger(POSITION_TYPE)==POSITION_TYPE_SELL)
        {
            SellOpened=true;
        }
    }
    //-
    MqlTradeRequest mrequest;
    MqlTradeCheckResult check_result;
    MqlTradeResult mresult;

    MqlTick latest_price;
    if(!SymbolInfoTick(_Symbol,latest_price))
    {
        Alert("Error getting the latest quotes - :",GetLastError(),"!!");
        return;
    }
    if(Buy==true&&BuyOpened==false){

```

In the Order method, we use the input parameters of the method as flags to open or close a buy or sell position.

```

12 class Sidus
13 {
14 private:
15 int numberBarOpenPosition;
16 int numberBarStopPosition;
17 int handleIMAA8;
18 double MA18Buffer[];
19 int handleIMA28;
20 double MA28Buffer[];
21 int handleIWMA5;
22 double WMA5Buffer[];
23 int handleIWMA8;
24 double WMA8Buffer[];
25
26 public:
27         Sidus(int BarOpenPosition, int BarStopPosition);
28         ~Sidus();
29 bool OnTradeSignalBuy();
30 bool OnTradeSignalBuyStop(MqlRates& mrate[]);
31 bool OnTradeSignalSell();
32 bool OnTradeSignalSellStop(MqlRates& mrate[]);
33 };
34 //+
35 Sidus::Sidus(int BarOpenPosition, int BarStopPosition)
36 {
37     numberBarOpenPosition=BarOpenPosition;
38     numberBarStopPosition=BarStopPosition;
39     handleIMAA8=iMA(_Symbol,PERIOD_H1,18,0,MODE_EMA,PRICE_WEIGHTED);
40     handleIMA28=iMA(_Symbol,PERIOD_H1,28,0,MODE_EMA,PRICE_WEIGHTED);
41     handleIWMA5=iMA(_Symbol,PERIOD_H1,5,0,MODE_IWMA,PRICE_WEIGHTED);
42     handleIWMA8=iMA(_Symbol,PERIOD_H1,8,0,MODE_IWMA,PRICE_WEIGHTED);
43 }

```

And we transfer the code of the functions of trading system signals to the Sidus class.

Here, we declare class instance variables — handles of the indicators used and their buffers.

And we initialize them in the class constructor.

And we also declare and implement methods for calculating signals to open or close a buy or sell position.

```

10 #include "CheckTrade.mqh"
11 #include "Trade.mqh"
12 #include "Sidus.mqh"
13
14 input double    Lot=1;
15 input int       EA_Magic=1000;
16 input double   spreadLevel=10.0;
17 input double   StopLoss=0.01;
18 input double   Profit=0.01;
19 input int      numberBarOpenPosition=5;
20 input int      numberBarStopPosition=5;
21
22 CheckTrade checkTrade;
23 Trade trade(StopLoss,Profit,Lot);
24 Sidus sidus(numberBarOpenPosition,numberBarStopPosition);
25
26 bool flagStopLoss=false;
27 //+-----+
28 //| Expert initialization function
29 //+-----+
30 int OnInit()
31 {
32 return (checkTrade.OnCheckTradeInit(Lot));
33 }

```

As a result of the functions code transfer into separate classes, the code of the adviser is significantly reduced.

Here, firstly, we include the files of the classes, which we created, using the include directive.

Then we declare the input parameters of the expert advisor and create instances of the classes, passing the appropriate input parameters to the class constructors.

In the advisor's OnInit method, we now simply call the OnCheckTradeInit method of the CheckTrade class.

```

void OnTick()
{
    if(!checkTrade.OnCheckTradeTick(Lot,spreadLevel)){
        return;
    }
    static datetime last_time;
    datetime last_bar_time=(datetime)SeriesInfoInteger(Symbol(),Period(),SERIES_LASTBAR_DATE);
    if(last_time!=last_bar_time)
    {
        last_time=last_bar_time;
    }else{
        return;
    }
    static datetime last_time_daily;
    datetime last_bar_time_daily=(datetime)SeriesInfoInteger(Symbol(),PERIOD_D1,SERIES_LASTBAR_DATE);
    if(last_time_daily!=last_bar_time_daily)
    {
        last_time_daily=last_bar_time_daily;
        flagStopLoss=false;
    }

    if(flagStopLoss==true)return;
}

```

To do checks, in the OnTick advisor's method, we call the OnCheckTradeTick method of the CheckTrade class.

```

int num=5;
if(numberBarOpenPosition>numberBarStopPosition)num=numberBarOpenPosition;
if(numberBarOpenPosition<=numberBarStopPosition)num=numberBarStopPosition;
MqlRates mrate[];
ResetLastError();
if(CopyRates(Symbol(),Period(),0,num,mrate)<0)
{
Print(GetLastError());
return;
}

ArraySetAsSeries(mrate,true);
//-----
bool TradeSignalBuy=false;
bool TradeSignalSell=false;
TradeSignalBuy=sidus.OnTradeSignalBuy();
TradeSignalSell=sidus.OnTradeSignalSell();
bool TradeSignalBuyStop=false;
bool TradeSignalSellStop=false;
TradeSignalBuyStop=sidus.OnTradeSignalBuyStop(mrate);
TradeSignalSellStop=sidus.OnTradeSignalSellStop(mrate);
trade.Order(TradeSignalBuy,TradeSignalBuyStop,TradeSignalSell,TradeSignalSellStop);
}

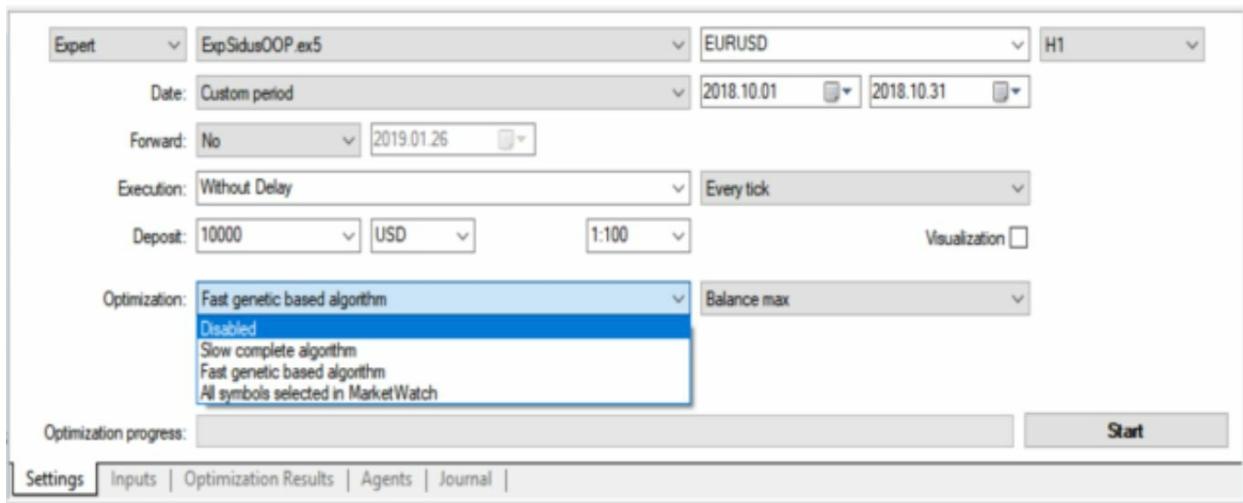
```

And after receiving the historical data of a symbol, we calculate signals of the trading system using the methods of the Sidus class.

And finally, we send a request to place an order using the class Trade.

Testing Expert Advisors

The built-in MetaTrader 5 terminal tester allows you to test and optimize input parameters of an adviser using historical data of financial instruments.



You can open the tester by right-clicking on an adviser in the terminal Navigator window and selecting the Test in the context menu.

You can also open the tester in the View menu of the terminal using the Strategy Tester command.

The testing is multithreaded and it is performed using special agent services, which are represented by three types.

Agent	Hardware	Tasks / Passed	Status
Local: 4 cores			
Core 1	Intel Pentium N3540 @ 2.16GHz, 3977 MB		ready
Core 2	Intel Pentium N3540 @ 2.16GHz, 3977 MB		ready
Core 3	Intel Pentium N3540 @ 2.16GHz, 3977 MB		ready
Core 4	Intel Pentium N3540 @ 2.16GHz, 3977 MB		ready
Local Network Farm			
MQL5 Cloud Network: 6 774 million tasks processed, 14 190 agents available			register
MQL5 Cloud Europe			not used
MQL5 Cloud USA			not used

Settings | Inputs | Optimization Results | Agents | Journal |

These are local agents installed along with the client terminal.

And the number of local agents is equal to the number of logical cores of the computer processor, which are used by agents in parallel, so all available computer resources are used for testing.

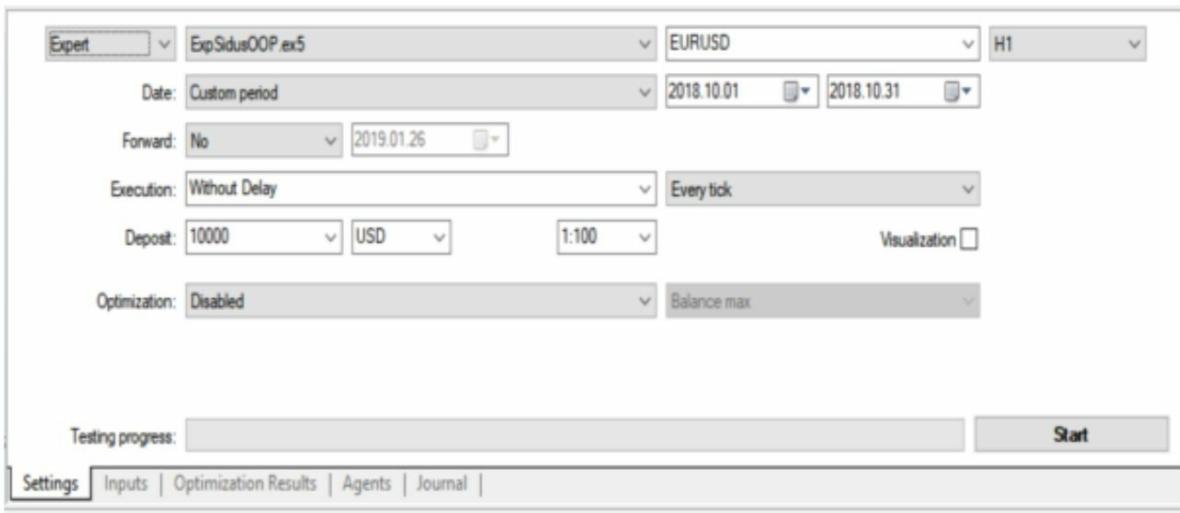
You can also use local network agents installed on other computers along with client terminals for testing.

And computers should be connected to a local network, after that, you could create an agent farm using the Agents Manager in the terminal Tool menu, and you could connect agents in the Agents tab of the terminal tester.

Local network agents can also be installed on the computer separately from the trading platform using the metatester64.exe file.

And local network agents can be installed only on 64-bit systems.

Another type of agents is cloud agents, the use of which is paid, and which you can connect adding it in the Agents tab of the terminal tester.



The tester's window allows you to select a testing interval, a financial instrument for testing, if it is not explicitly specified in an advisor, a time period of the financial instrument, and you can enable optimization of input parameters of an advisor, visual testing mode, and others.

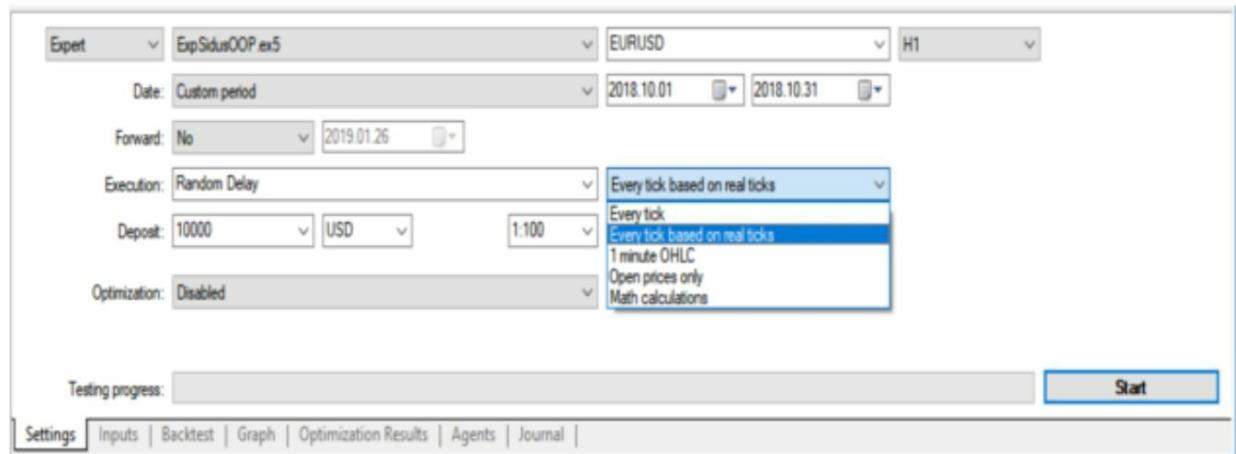


When you turn on the visual test mode using the Visualization checkbox, the indicators used by an expert are shown on a chart.

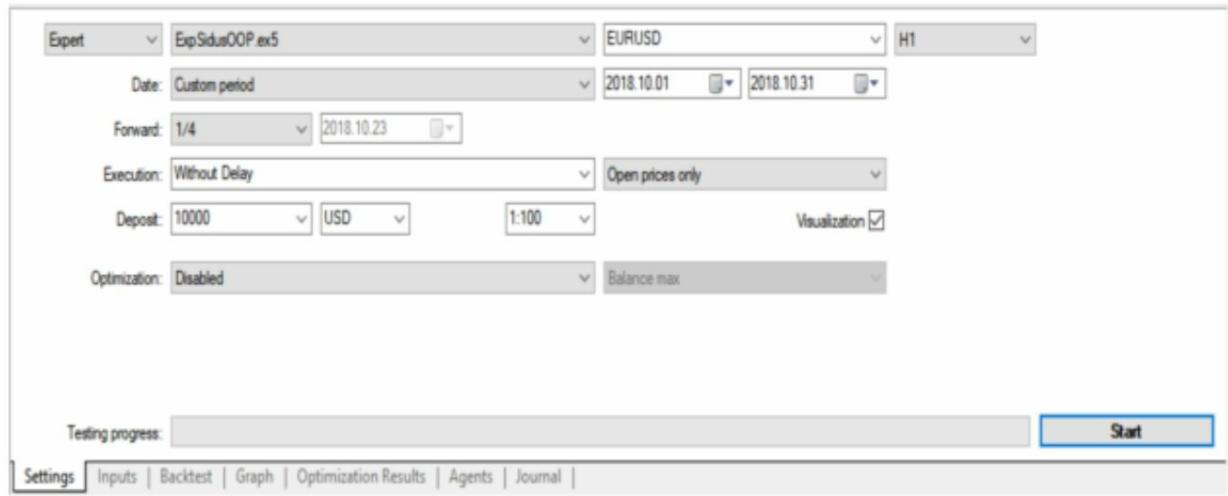
History Quality	100%			
Bars	528	Ticks	2007581	Symbols 1
Initial Deposit	10 000.00			
Total Net Profit	-406.00	Balance Drawdown Absolute	406.00	Equity Drawdown Absolute 507.00
Gross Profit	794.28	Balance Drawdown Maximal	812.67 (7.81%)	Equity Drawdown Maximal 1 157.67 (10.87%)
Gross Loss	-1 200.28	Balance Drawdown Relative	7.81% (812.67)	Equity Drawdown Relative 10.87% (1 157.67)
Profit Factor	0.66	Expected Payoff	-29.00	Margin Level 827.03%
Recovery Factor	-0.35	Sharpe Ratio	-0.15	Z-Score 2.48 (98.69%)
AHPR	0.9972 (-0.28%)	LR Correlation	-0.74	OnTester result 0
GHPR	0.9970 (-0.30%)	LR Standard Error	152.57	
Total Trades	14	Short Trades (won %)	6 (50.00%)	Long Trades (won %) 8 (25.00%)
Total Deals	28	Profit Trades (% of total)	5 (35.71%)	Loss Trades (% of total) 9 (64.29%)
	Largest profit trade		520.67	loss trade -247.00
	Average profit trade		158.86	loss trade -133.36
	Maximum consecutive wins (\$)		1 (520.67)	consecutive losses (\$) 3 (-407.00)
	Maximal consecutive profit (count)		520.67 (1)	consecutive loss (count) -407.00 (3)
	Average consecutive wins		1	consecutive losses 2

Settings | Inputs | **Backtest** | Graph | Optimization Results | Agents | Journal |

The Backtest tab of the tester's window shows results of testing an expert.



To get as close as possible to the conditions of real trading, during testing, you can choose the mode Random Delay and Every tick based on real ticks.



And, adviser's input parameters can be quickly optimized in the mode Forward: 1/4, Without Delay, and Opening prices only.

The option "Opening prices only" is mean that calculations are performed only at opening prices of bars.

The option Forward testing is mean that it re-run an advisor on a different time period.

This possibility is provided to exclude adjustment of parameters of advisers on particular intervals of historical data.

With the inclusion of this option, the history of currency quotes and stock quotes is divided into two parts.

The optimization itself is performed on the first part of the history, and the second part is used only to confirm the results obtained.

If the efficiency of a trading robot is equally high on both history parts, it means that the trading system has the best parameters and the fitting of the parameters is practically excluded.

The option Forward 1/4 uses a quarter of the specified period for forward testing.

The mode of random execution delays emulates network delays in the transmission and processing of trade requests and also simulates a delay in

execution of orders by dealers in real trading.

An important function of the Strategy Tester is to optimize a trading robot, which allows you to select the best input parameters for a particular advisor.

In the process of optimization, you could test one trading robot with different input parameters.

The number of combinations of input parameters during optimization can reach tens or hundreds of thousands.

As a result, the optimization can turn into a very long process, which can still be significantly reduced by using genetic algorithms.

This feature disables sequential enumeration of all combinations of input parameters and selects only those that best meet optimization criteria.

In subsequent stages, "optimal" combinations are crossed until test results stop improving.

Thus, the number of combinations and a total optimization time are reduced by several times.



In order to see the effectiveness of an adviser, let's create an indicator showing all potential deals for the purchase and sale of an instrument.

This indicator is based on the moving average.

As a moving average value increases, a buy position will be calculated, while a moving average value decreases, a sell position will be calculated.

```

156 if((InBuffer[i]-InBuffer[i-shift_delta])>delta){
157 ColorBuffer[i]=2;
158 if(flagBuy==false){
159 priceBuyOpen=open[i];
160 if(!ObjectCreate(0,"Buy"+i,OBJ_ARROW,0,time[i],high[i]))
161 {
162     return(false);
163 }
164
165     ObjectSetInteger(0,"Buy"+i,OBJPROP_COLOR,clrGreen);
166     ObjectSetInteger(0,"Buy"+i,OBJPROP_ARROWCODE,233);
167     ObjectSetInteger(0,"Buy"+i,OBJPROP_WIDTH,1);
168     ObjectSetInteger(0,"Buy"+i,OBJPROP_ANCHOR,ANCHOR_UPPER);
169     ObjectSetInteger(0,"Buy"+i,OBJPROP_HIDDEN,true);
170     ObjectSetString(0,"Buy"+i,OBJPROP_TOOLTIP,""+close[i]);
171
172 if((InBuffer[i-shift_delta]-InBuffer[i])>delta){
173 ColorBuffer[i]=1;
174 if(flagSell==false){
175 priceSellOpen=open[i];
176 if(!ObjectCreate(0,"Sell"+i,OBJ_ARROW,0,time[i],low[i]))
177 {
178     return(false);
179 }
180     ObjectSetInteger(0,"Sell"+i,OBJPROP_COLOR,clrRed);
181     ObjectSetInteger(0,"Sell"+i,OBJPROP_ARROWCODE,234);
182     ObjectSetInteger(0,"Sell"+i,OBJPROP_WIDTH,1);
183     ObjectSetInteger(0,"Sell"+i,OBJPROP_ANCHOR,ANCHOR_LOWER);
184     ObjectSetInteger(0,"Sell"+i,OBJPROP_HIDDEN,true);
185     ObjectSetString(0,"Sell"+i,OBJPROP_TOOLTIP,""+close[i]);
186

```

Here, if a price starts to rise, we get up to buy, if a price falls, we get up to sell, and we calculate a potential profit, which we would ideally be able to earn.

```
28 int handleProfit;
29 double ProfitBuffer[];
30 //+-----+
31 //| Expert initialization function |
32 //+-----+
33 int OnInit()
34 {
35 handleProfit=iCustom(_Symbol,PERIOD_H1,"Profit",5,0.0001,1);
36 return(checkTrade.OnCheckTradeInit(Lot));
37 }
```

Let's join this indicator to the Sidus advisor that we considered in the previous example.

```

98 //-----
99 void OnTrade()
100 {
101 static int _deals;
102 ulong _ticket=0;
103
104 if(HistorySelect(0,TimeCurrent()))
105 {
106 int i=HistoryDealsTotal()-1;
107
108 if(_deals!=i) {
109 _deals=i;
110 } else { return; }
111
112 if({_ticket=HistoryDealGetTicket(i)})>0
113 {
114 string _comment=HistoryDealGetString(_ticket,DEAL_COMMENT);
115 if(StringFind(_comment,"sl",0)!=-1) {
116 flagStopLoss=true;
117
118 if(HistoryDealGetDouble(_ticket, DEAL_PROFIT)!=0.0){
119 if(!ObjectCreate(0,"Deal"+_ticket,OBJ_TEXT,0,
120 HistoryDealGetInteger(_ticket,DEAL_TIME),HistoryDealGetDouble(_ticket,DEAL_PRICE)))
121 {
122 return;
123 }
124 ObjectSetString(0,"Deal"+_ticket,OBJPROP_TEXT,"Profit: "+HistoryDealGetDouble(_ticket, DEAL_PROFIT));
125 ObjectSetDouble(0,"Deal"+_ticket,OBJPROP_ANGLE,90.0);
126 ObjectSetInteger(0,"Deal"+_ticket,OBJPROP_COLOR,clrYellow);
127 }}}

```

In the OnTrade function of the advisor, we will show an actual profit received from a deal.

And let's make visual testing of the advisor.



From visual testing, it is clear that this adviser has problems with entering and exiting the market, as well as a lot of missed potential deals.

Money Management and Expert Evaluation

Money management is a way of deciding which part of an account you should risk in a particular trading opportunity.

And we can formulate typical money management rules.

Money Management Rules:

The ratio of possible losses and profits - 1:3

You should close unprofitable positions earlier than profitable ones

You should avoid very short positions

Do not make decisions before closing the market

The total amount of funds invested in a single trade cannot exceed 10% - 15% of the total capital

The total percentage of invested funds for all open positions cannot exceed 30%

The maximum permissible risk per trade cannot exceed 2-5% of the deposit amount

The total maximum permissible risk for all open positions cannot exceed 5-7% of the deposit amount

- The ratio of possible losses and profits 1:3.
- Close unprofitable positions earlier than profitable ones.
- Avoid very short positions.
- Do not make decisions before closing a market.
- The total funds invested in a single trade cannot exceed 10% - 15% of the total capital.
- The total percentage of invested funds for all open positions cannot exceed 30%.
- The maximum permissible risk per trade is 2-5% of the deposit amount.
- The total maximum permissible risk for all open positions is 5-7% of the deposit.

And we can formulate typical money management strategies.

Money Management Strategies:

A fixed amount or a fixed percentage of capital at risk in a next deal

Coordination of winnings and losses in trading

Intersection of price curves - this system analyzes a trading system using long and short moving average curves (for example, 3 and 8) of profits and losses of trades

Optimal f and Secure f is a definition based on testing a trading system on past trades of some optimal share of the initial capital invested in each trade in order to achieve maximum profitability of this system as the main criterion (Optimal f) or achieve maximum profitability taking into account the limit of the maximum permissible losses (Secure f)

- Fixed amount or a fixed percentage of capital at risk in the next trade.
- Coordination of profits and losses in trading.

According to this method, traders define a volume of trade after successful wins or losses.

For example, after a lost trade, they may decide to double the volume of trade after a next signal to trade, in order to recover the losses.

In this system, statistical analysis is used to establish a relationship between loss and win, when losses and winnings either alternate or after a certain amount of losses, winnings follow.

At the same time, after a win, you can increase a volume of opened positions and decrease it after a loss, or after a series of losses, you can increase a volume of a position, waiting for a quick win, or, conversely, decrease it, waiting for a loss.

- The intersection of price curves - this system analyzes a trading system using long and short moving average curves (for example, 3 and 8) of profits and losses of trades.

If the short moving average curve is above the longer moving curve, then the trading system gives better results, and if the short moving average curve is below the long sliding curve, then the system works worse and signals of the

trading system to open positions should be skipped.

At the same time, to calculate the curves, you should use all signals of the trading system, and not just those that were actually implemented.

The presence of winning and losing periods of a trading system is also defined using statistical analysis.

- Optimal f and Secure f is the definition based on testing a trading system on past trades of some optimal share of the initial capital invested in each trade in order to achieve maximum profitability of this system as the main criterion (Optimal f) or to achieve maximum profitability taking into account the limitation of maximum allowable losses (Secure f).

A relationship between losses and winnings is analyzed using a Z-score.

And at the same time, a confidence interval should exceed 94%.

$$Z = (N * (R - 0.5) - X) / ((X * (X - N)) / ((N - 1))^1/2)$$

Where:

N = total number of trades

X = 2 * (total number of profitable trades) * (total number of losing trades).

R = number of series in observation. Every time a losing trade follows a profitable trade or vice versa is considered a series.

Z-score is calculated by comparing a number of series in a set of trades relative to a number of series, which could be expected with the statistical independence of results of a current trade from past trades.

And you can see the Z-score formula on the slide.

Where:

N - total number of transactions.

X - 2 * (total number of profitable trades) * (total number of losing trades).

R - number of series in observation. Every time a losing trade follows a profitable trade or vice versa is considered a series.

A positive Z-score means that in the observed trade history, the number of series is less than would be expected with a statistically independent distribution of trade outcomes.

Consequently, there is a tendency that after winning the loss follows, and after losing the win follows.

Having a positive Z-score means that, after a losing trade, you can increase a volume of the position being opened on a next signal, this will cover losses and increase the overall profitability of a system.

A negative Z-score means that a number of series in an observable trade history are greater than one would expect given a statistically independent

distribution of trade outcomes.

Consequently, a winning trade is usually followed by another winning trade, and a losing trade is followed by another losing trade.

The presence of a negative Z-account means that you can use the intersection of the yield curves, while the trading system will work during a profitable phase and turn off after starting an unprofitable phase.

History Quality	100%				
Bars	384	Ticks	939	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	-133.61	Balance Drawdown Absolute	332.47	Equity Drawdown Absolute	416.47
Gross Profit	811.53	Balance Drawdown Maximal	739.14 (7.10%)	Equity Drawdown Maximal	1 067.14 (10.02%)
Gross Loss	-945.14	Balance Drawdown Relative	7.10% (739.14)	Equity Drawdown Relative	10.02% (1 067.14)
Profit Factor	0.86	Expected Payoff	-12.15	Margin Level	827.03%
Recovery Factor	-0.13	Sharpe Ratio	-0.05	Z-Score	1.67 (90.51%)
AHPR	0.9990 (-0.10%)	LR Correlation	-0.66	On tester result	0
GHPR	0.9988 (-0.12%)	LR Standard Error	170.20		
Total Trades	11	Short Trades (won %)	5 (60.00%)	Long Trades (won %)	6 (16.67%)
Total Deals	22	Profit Trades (% of total)	4 (36.36%)	Loss Trades (% of total)	7 (63.64%)
	Largest	profit trade	520.67	loss trade	-247.00
	Average	profit trade	202.88	loss trade	-135.02
	Maximum	consecutive wins (\$)	1 (520.67)	consecutive losses (\$)	3 (-407.00)
	Maximal	consecutive profit (count)	520.67 (1)	consecutive loss (count)	-407.00 (3)
	Average	consecutive wins	1	consecutive losses	2

A value of the Z-account of an expert can be viewed in the Backtest tab of the client terminal's tester after testing the expert.

In the considered example of an expert based on the Sidus trading system, with the values of numberBarOpenPosition = 5 and numberBarStopPosition = 5, the Z-score of the expert is 1.67 (90.51%).

However, on this basis, it is quite problematic to adjust the work of the expert, since, after a win, not a single loss can follow, but a series of losses and vice versa.

You can output sequences of wins and losses of an expert in its OnDeinit function.

```
34 //+-----+
35 //| Expert deinitialization function           |
36 //+-----+
37 void OnDeinit(const int reason){
38     if(HistorySelect(0,TimeCurrent()))
39     {
40         int n=HistoryDealsTotal();
41         for(int i=0;i<n;i++)
42             if(HistoryDealGetDouble(HistoryDealGetTicket(i), DEAL_PROFIT)!=0.0)
43                 Print("Profit ",HistoryDealGetDouble(HistoryDealGetTicket(i), DEAL_PROFIT));
44         }
45     }
46 }
47 }
```

And the overall effectiveness of an adviser is calculated as the difference between a profit realized during a trade and a potential profit during the same trade.

Overall Efficiency =
(Realized Price Difference) / (Potential Profit)

For long positions:

Overall Efficiency = (Closing Price - Opening Price) /
(Maximum Price - Minimum Price)

For short positions:

Overall Efficiency = (Opening Price - Closing Price) /
(Maximum Price - Minimum Price)

That is you could calculate overall effectiveness as
(Realized price difference) / (Potential profit)

For long positions:

Overall effectiveness = (Closing price - Opening price) /
(Maximum price - minimum price)

For short positions:

Overall Efficiency = (Opening Price - Closing Price) /
(Maximum price - minimum price)

And the effectiveness of a market entry could be defined as a maximum difference in prices relative to a price of entry as a part of an overall profitability potential during a trade.

Entry Efficiency =
(Maximum Price Difference Relative to Entry Price) / (Potential Profit)

For long positions:

Login Efficiency =
(Maximum Price - Trade Opening Price) / (Maximum Price - Minimum Price)

For short positions:

Login Efficiency =
(Trade Opening Price - Minimum Price) / (Maximum Price - Minimum Price)

The effectiveness of entry into a market shows how well a trading system opens trades - in the case of long positions, this is how much an entry signal is close to the smallest point during a trade, in the case of short positions, this is how much an entry signal is close to a maximum point during a trade.

And you could calculate an entry efficiency as (maximum price difference relative to entry price) / (Potential profit)

A maximum difference in prices relative to an entry price is the difference between a maximum price and an entry price (for short positions, a minimum price).

And for long positions:

An entry efficiency is equal to (Maximum price - Trade opening price) / (Maximum price - Minimum price)

And for short positions:

An entry efficiency is equal to (Trade Opening Price - Minimum Price) / (Maximum Price - Minimum Price)

Effectiveness of exit from a market is defined as a maximum difference in prices relative to an exit price as a part of an overall profitability potential during a trade.

Exit Efficiency =
(Maximum Price Difference Relative to Exit Price) / (Potential Profit)

For long positions:

Exit Efficiency =
(Exit Price - Minimum Price) / (Maximum Price - Minimum Price)

For short positions:

Exit Efficiency =
(Maximum Price - Exit Price) / (Maximum Price - Minimum Price)

The effectiveness of exit from a market shows how well a system closes trades - in the case of long positions, this is how much an exit signal is close to a maximum point during a trade, in the case of short positions, this is how much an exit signal is close to a minimum point during a trade.

And you could calculate an exit efficiency as (maximum price difference relative to exit price) / (Potential profit)

The maximum difference in prices relative to an exit price is a difference between an exit price and a minimum price (for short positions, a maximum price).

And for long positions:

The efficiency of an exit is equal to (The price of an exit - Minimum price) / (The maximum price - Minimum price)

For short positions:

A yield efficiency is equal to (Maximum price - Output price) / (Maximum price - Minimum price)

And an overall efficiency is a sum of an entry efficiency and exit efficiency minus 1.

An entry efficiency and exit efficiency can be positive values, but an overall efficiency can be negative.

If the sum of entry efficiency and exit efficiency is less than 100%, then the trade is unprofitable.

To analyze the work of an expert, you could calculate an average total efficiency, average entry efficiency, and an average exit efficiency.

To calculate averages, it is best to use statistical analysis to discard pop-up trades.

The trading system is statistically profitable if:

$$\begin{aligned} & (\text{Average Profit Trade}) * (\% \text{ Profit Trades}) - \text{Overhead Costs} > \\ & (\text{Average Losing Trade}) * (\% \text{ Loss Trades}) \end{aligned}$$

And a trading system is statistically profitable if:

$$\begin{aligned} & (\text{Average winning trade}) * (\% \text{ winnings}) - \text{overhead costs} > \\ & (\text{Average losing trade}) * (\% \text{ losses}) \end{aligned}$$

And the overheads are slippage, commissions, and others.

Such indicators as Average profitable trade, Profitable trades (% of all), Average unprofitable trade, Loss trades (% of all) can be viewed in the tester's Backtest tab of the client terminal after expert testing.

The indicator of the statistical profitability of an expert without taking into account overheads or the expectation of a gain can also be viewed in the tester's Backtest tab of the client terminal after testing an expert.

$$\text{Expected Payoff} = (\text{ProfitTrades} / \text{TotalTrades}) * (\text{GrossProfit} / \text{ProfitTrades}) - (\text{LossTrades} / \text{TotalTrades}) * (\text{GrossLoss} / \text{LossTrades})$$

TotalTrades - total number of trades;
ProfitTrades - number of profitable trades;
LossTrades - number of losing trades;
GrossProfit - total profit;
GrossLoss - total loss.

And you could calculate an expected payoff the formula as shown on the slide.

Where:

TotalTrades - total number of trades;
ProfitTrades - number of profitable trades;
LossTrades - number of losing trades;
GrossProfit - total profit;
GrossLoss - a total loss.

And here, ProfitTrades / TotalTrades is the % of winnings,
GrossProfit / ProfitTrades is the average winning trade,
LossTrades / TotalTrades is the % of losses,
GrossLoss / LossTrades is the average losing trade.

History Quality	100%				
Bars	528	Ticks	2081	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	404.36	Balance Drawdown Absolute	272.00	Equity Drawdown Absolute	2 200.28
Gross Profit	3 615.28	Balance Drawdown Maximal	764.64 (7.04%)	Equity Drawdown Maximal	4 136.00 (31.72%)
Gross Loss	-3 210.92	Balance Drawdown Relative	7.04% (764.64)	Equity Drawdown Relative	31.72% (4 136.00)
Profit Factor	1.13	Expected Payoff	7.09	Margin Level	84.69%
Recovery Factor	0.10	Sharpe Ratio	0.05	Z-Score	-1.33 (81.65%)
AHPR	1.0008 (0.08%)	LR Correlation	0.11	OnTester result	0
GHPR	1.0007 (0.07%)	LR Standard Error	241.29		
Total Trades	57	Short Trades (won %)	28 (50.00%)	Long Trades (won %)	29 (48.28%)
Total Deals	114	Profit Trades (% of total)	28 (49.12%)	Loss Trades (% of total)	29 (50.88%)
	Largest profit trade		704.67	loss trade	-241.00
	Average profit trade		129.12	loss trade	-110.72
	Maximum consecutive wins (\$)		7 (658.30)	consecutive losses (\$)	4 (-364.11)
	Maximal consecutive profit (count)		704.67 (1)	consecutive loss (count)	-460.14 (3)
	Average consecutive wins		2	consecutive losses	2

In the considered example of an expert based on the Sidus trading system, if the value numberBarOpenPosition = 7 and numberBarStopPosition = 2, the expected payoff is 7.

This is less than 10 points, so we can say that the profitability of the expert is unstable.

In the considered example of the expert based on the Sidus trading system, we replace signals for closing a position to use a trailing stop, that is when a price reaches the break-even level, we move a stop loss.

```
16 class Trade
17 {
18 private:
19 double StopLoss;
20 double Profit;
21 double Lot;
22 double TrailingStop;
23 double priceBuy;
24 double priceSell;
25 double slBuy;
26 double slSell;
27
28 CPositionInfo m_position;
29 CTrade m_trade;
30
31
32 public:
33         Trade(double stopLoss, double profit, double lot, double trailingStop);
34         ~Trade();
35 void         Order(bool Buy, bool StopBuy, bool Sell, bool StopSell);
36 void         Trailing();
37 };
```

To do this, let's change the class Trade.

Here, we declare the additional function Trailing.

```

86 if(BuyOpened==true) {
87     double TBS=0;
88     if(TrailingStop<SymbolInfoInteger(Symbol(),SYMBOL_TRADE_STOPS_LEVEL)*_Point){
89         TBS=SymbolInfoInteger(Symbol(),SYMBOL_TRADE_STOPS_LEVEL)*_Point;
90     }else{
91         TBS=TrailingStop;
92     }
93 if(
94 latest_price.bid-priceBuy>=TBS
95 ){
96     mrequest.action = TRADE_ACTION_SLTP;
97     mrequest.price = NormalizeDouble(latest_price.ask,_Digits);
98     mrequest.sl = NormalizeDouble(priceBuy,_Digits); // Stop Loss
99     mrequest.tp = NormalizeDouble(latest_price.ask + Profit,_Digits);
100    mrequest.symbol = _Symbol;
101    mrequest.volume = Lot;
102    mrequest.type_filling = ORDER_FILLING_FOK;
103    mrequest.type = ORDER_TYPE_BUY;
104
105    if(!OrderCheck(mrequest,check_result))

```

And in this function, with an open buy position, if the price has gone up enough, we move up a stop loss.

And we do the same for an open sale position.

```
83
84 bool TradeSignalBuy=false;
85 bool TradeSignalSell=false;
86
87
88 TradeSignalBuy=sidus.OnTradeSignalBuy();
89 TradeSignalSell=sidus.OnTradeSignalSell();
90
91 bool TradeSignalBuyStop=false;
92 bool TradeSignalSellStop=false;
93
94 //TradeSignalBuyStop=sidus.OnTradeSignalBuyStop();
95 //TradeSignalSellStop=sidus.OnTradeSignalSellStop();
96
97 trade.Order(TradeSignalBuy,TradeSignalBuyStop,TradeSignalSell,TradeSignalSellStop);
98 trade.Trailing();
```

In the adviser's OnTick function, we call the function Trailing of the class Trade.

And let's test the advisor.

History Quality	100%				
Bars	528	Ticks	2081	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	2 073.86	Balance Drawdown Absolute	1 124.00	Equity Drawdown Absolute	619.00
Gross Profit	4 966.53	Balance Drawdown Maximal	1 124.00 (11.24%)	Equity Drawdown Maximal	2 450.64 (18.00%)
Gross Loss	-2 892.67	Balance Drawdown Relative	11.24% (1 124.00)	Equity Drawdown Relative	18.00% (2 450.64)
Profit Factor	1.72	Expected Payoff	66.90	Margin Level	107.77%
Recovery Factor	0.85	Sharpe Ratio	0.21	Z-Score	-1.46 (85.57%)
AHPR	1.0066 (0.66%)	LR Correlation	0.67	OnTester result	0
GHPR	1.0061 (0.61%)	LR Standard Error	765.22		
Total Trades	31	Short Trades (won %)	18 (72.22%)	Long Trades (won %)	13 (15.38%)
Total Deals	62	Profit Trades (% of total)	15 (48.39%)	Loss Trades (% of total)	16 (51.61%)
Largest profit trade			704.67	loss trade	-1 010.00
Average profit trade			331.10	loss trade	-180.79
Maximum consecutive wins (\$)			8 (3 756.36)	consecutive losses (\$)	5 (-706.28)
Maximal consecutive profit (count)			3 756.36 (8)	consecutive loss (count)	-1 124.00 (2)
Average consecutive wins			3	consecutive losses	3

After optimizing the Stop Loss and Take Profit, the expected payoff is now 67.

The net profit during trailing has also increased, and the statistical profitability of the expert has improved.

Another indicator of the Backtest tab of the strategy tester is the Sharpe Ratio, which characterizes the efficiency and stability of an expert.

The value of the indicator is higher, that the rate of return per unit of risk is more.

And the value of the Sharpe Ratio for sustainable experts is more than 0.25.

In our case, the Sharpe Ratio is 0.21.

The growth factor of an expert or GHPR (Geometric Average Holding Period Return) is equal to (Final Deposit / Initial Deposit) to the degree 1/(Total trades).

In this case, GHPR (growth factor) is equal to 1.0061 - more than one, which means the ability to trade using reinvestment.

Other indicators of an expert, which can be seen in the Backtest tab of the strategy tester, are the following:

Profit Factor - Total Profit/Total Loss. The recommended value is at least 2.

Recovery Factor - Net Profit/Maximum Drawdown.

The higher the value of the indicator, that an advisor is less risky.

Many traders believe that an effective trading system should have a recovery factor at least 3.

And AHPR (Arithmetical Holding Return Period) is a trade's arithmetic mean.

A positive value of AHPR indicates that the trading system is profitable.

The indicator Margin Level is a minimum margin level in percent, which was recorded during the testing period.

The indicator LR Correlation allows you to estimate the deviations of the points of the account balance chart from linear regression.

The closer the LR Correlation is to zero, the more random the nature of the trade.

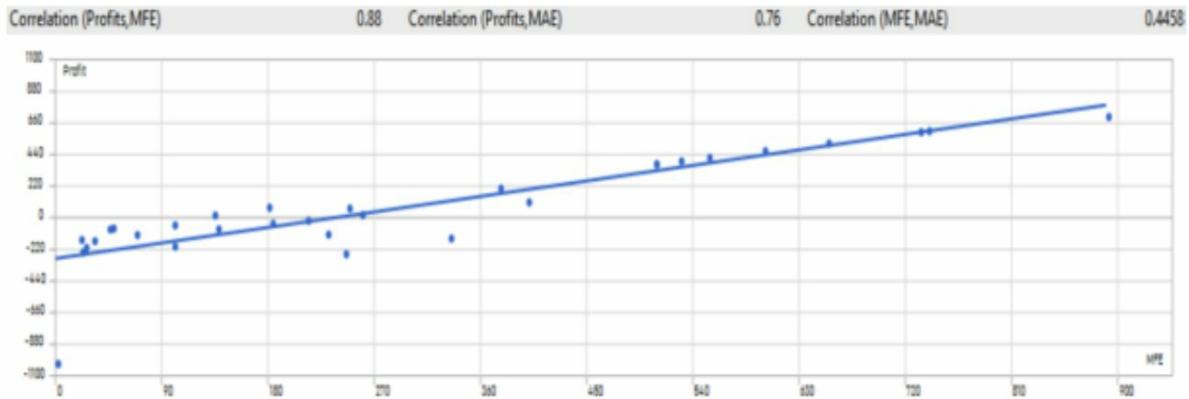
The indicator LR Standard Error is a deviation of the account balance chart from linear regression in monetary terms.

Average profit trade / Average loss trade - it is desirable that a ratio of these two indicators be more than one.

The indicator Maximal count of consecutive loss - it is desirable that this indicator was as small as possible.

Based on the depth of the maximum drawdown, you can calculate a minimum deposit required to start trading with this expert.

You could calculate it as $\text{Min deposit} = \text{Maximum drawdown} * 2$



Correlation (Profits, MFE) is a relationship between the results of positions and MFE (Maximum Favorable Excursion - the maximum amount of potential profit observed during the position holding).

MFE shows a maximum price movement in a favorable direction.

The closer the Correlation (Profits, MFE) indicator to one, the better the expert realizes the potential profit.

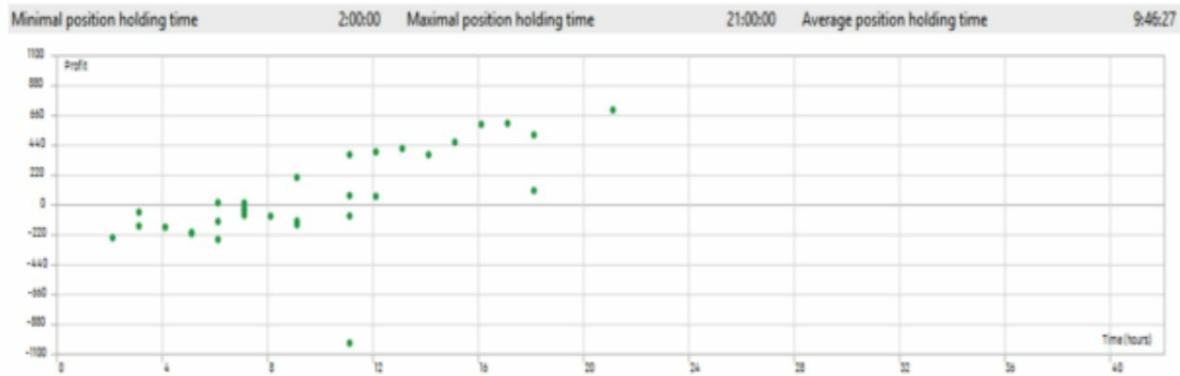
Correlation (Profits, MAE) is a relationship between position results and MAE (Maximum Adverse Excursion - the maximum potential loss observed during position holding).

MAE shows the most unfavorable price movement.

The closer the Correlation (Profits, MAE) indicator to one, the better the expert uses a protective stop loss.

Correlation (MFE, MAE) is a connection between MFE and MAE.

The closer the Correlation indicator (MFE, MAE) is to one, the better the expert realizes a maximum profit and protects a position as much as possible throughout its life.



Average position holding time - the indicator is calculated as a total holding time divided by a number of trades.

Holding a position increases the risk because a drawdown can obviously be greater with positions held for a longer period of time.

Creating Expert with MQL5 Wizard

The MQL5 Wizard, which you could open using the New button in the toolbar of the MetaEditor, allows you to generate expert code based on ready-made modules — signals, money management modules, and trailing stop modules.

The image consists of two side-by-side windows from the MQL Wizard.

Left Window (MQL Wizard: File):

Welcome to MQL5 Wizard

Please select what you would like to create:

- Expert Advisor (template)
- Expert Advisor (generate)
- Custom Indicator
- Script
- Service
- Library
- Include (*.mqh)
- New Class

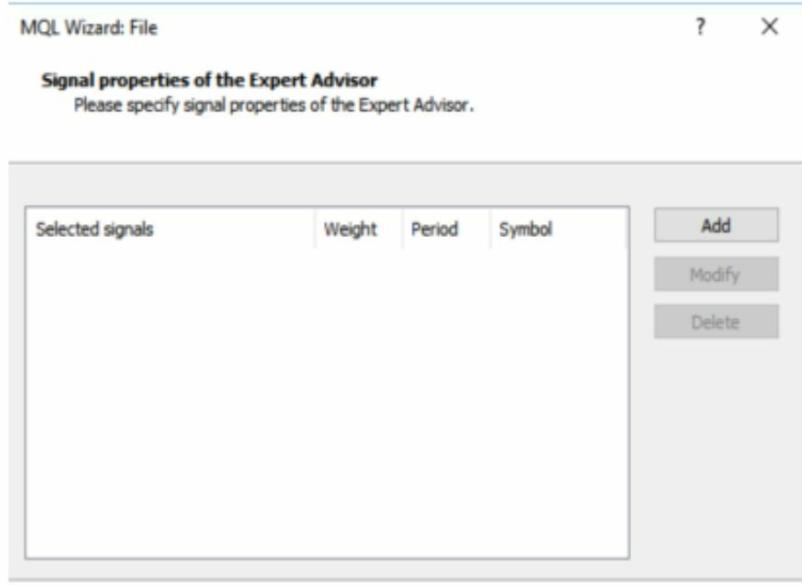
Right Window (MQL Wizard: File):

General properties of the Expert Advisor

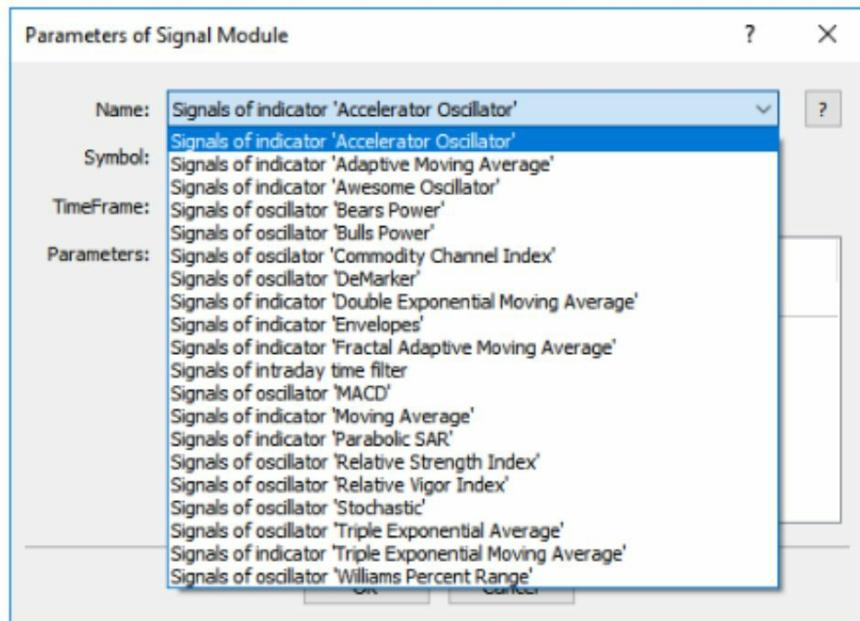
Please specify general properties of the Expert Advisor.

Name:	Experts										
Author:	Copyright 2019, NOVTS										
Link:	http://novts.com										
Parameters:	<table border="1"><thead><tr><th>Name</th><th>Type</th><th>Value</th></tr></thead><tbody><tr><td>Symbol</td><td>string</td><td>current</td></tr><tr><td>TimeFrame</td><td>Timeframe</td><td>current</td></tr></tbody></table>		Name	Type	Value	Symbol	string	current	TimeFrame	Timeframe	current
Name	Type	Value									
Symbol	string	current									
TimeFrame	Timeframe	current									
	Add	Delete									

And here, you could add a signal module using the Add button.



Signal module files are Include (*.mqh) files located in the MQL5\Include\Expert\Signal folder.



As an example, let's select the MACD signal and the PSAR signal.

Signal properties of the Expert Advisor

Please specify signal properties of the Expert Advisor.

Selected signals	Weight	Period	Symbol
MACD(12,24,9,PRICE_CLOSE)	1.0	current	current
Parabolic SAR(0.02,0.2)	1.0	current	current

Add
Modify
Delete

If you look at the SignalMACD and SignalSAR files in the MQL5\Include\Expert\Signal folder, the MACD signal has 5 models of price prediction.

```

25 //|      the 'Moving Average Convergence/Divergence' oscillator. |
26 //| Is derived from the CExpertSignal class. |
27 //+
28 class CSignal1MACD : public CExpertSignal
29 {
30 protected:
31     C1MACD          m_MACD;           // object-oscillator
32     //--- adjusted parameters
33     int              m_period_fast;   // the "period of fast EMA" parameter of the oscillator
34     int              m_period_slow;    // the "period of slow EMA" parameter of the oscillator
35     int              m_period_signal;  // the "period of averaging of difference" parameter of the oscillator
36     ENUM_APPLIED_PRICE m_applied;    // the "price series" parameter of the oscillator
37     //--- "weights" of market models (0-100)
38     int              m_pattern_0;      // model 0 "the oscillator has required direction"
39     int              m_pattern_1;      // model 1 "reverse of the oscillator to required direction"
40     int              m_pattern_2;      // model 2 "crossing of main and signal line"
41     int              m_pattern_3;      // model 3 "crossing of main line an the zero level"
42     int              m_pattern_4;      // model 4 "divergence of the oscillator and price"
43     int              m_pattern_5;      // model 5 "double divergence of the oscillator and price"
44     //--- variables
45     double           m_extr_osc[10];   // array of values of extrems of the oscillator
46     double           m_extr_pr[10];    // array of values of the corresponding extrems of price
47     int              m_extr_pos[10];   // array of shifts of extrems (in bars)
48     uint             m_extr_map;     // resulting bit-map of ratio of extrems of the oscillator and the price
49

```

Model 0 - "the oscillator has the necessary direction" with the significance 10.

The model 1 - "turning the oscillator in the right direction" with the significance 30.

The model 2 - "the intersection of the main and signal lines" with the significance 80.

Model 3 - "the intersection of the main line of the level zero" with the significance 50.

Model 4 - "oscillator divergence and price" with the significance 60.

And the model 5 - "double divergence of the oscillator and prices" with the significance of 100.

```

SignalEnvelopes.mqh
SignalFrAMA.mqh
SignalITF.mqh
SignalMA.mqh
SignalMACD.mqh
SignalRSI.mqh
SignalRVI.mqh
SignalSAR.mqh
SignalStoch.mqh

26 class CSignalSAR : public CExpertSignal
27 {
28 protected:
29     CiSAR         m_sar;           // object-indicator
30     //--- adjusted parameters
31     double        m_step;         // the "speed increment" parameter of the indicator
32     double        m_maximum;       // the "maximum rate" parameter of the indicator
33     //--- "weights" of market models (0-100)
34     int          m_pattern_0;     // model 0 "the parabolic is on the necessary side from the price"
35     int          m_pattern_1;     // model 1 "the parabolic has 'switched'"
```

The SAR signal has 2 models of price prediction:

Model 0 - "parabolic is on the right side of the price" with the significance 40.

And the model 1 - "parabolic switches to the other side of the price" with the significance 90.

If the model gives a signal for a price drop, the significance is negative; if it gives a signal for a price increase, the significance is positive.

The final forecast of the two modules will be calculated using the following formula.

$$\text{Final Forecast} = (\text{MACD Forecast} + \text{SAR Forecast}) / 2$$

Where:

$$\text{MACD Forecast} = \text{MACD Signal Weight} * \text{MACD Model Significance}$$

$$\text{SAR Forecast} = \text{SAR Signal Weight} * \text{SAR Model Significance}$$

The final forecast is equal to $(\text{MACD Forecast} + \text{SAR Forecast}) / 2$

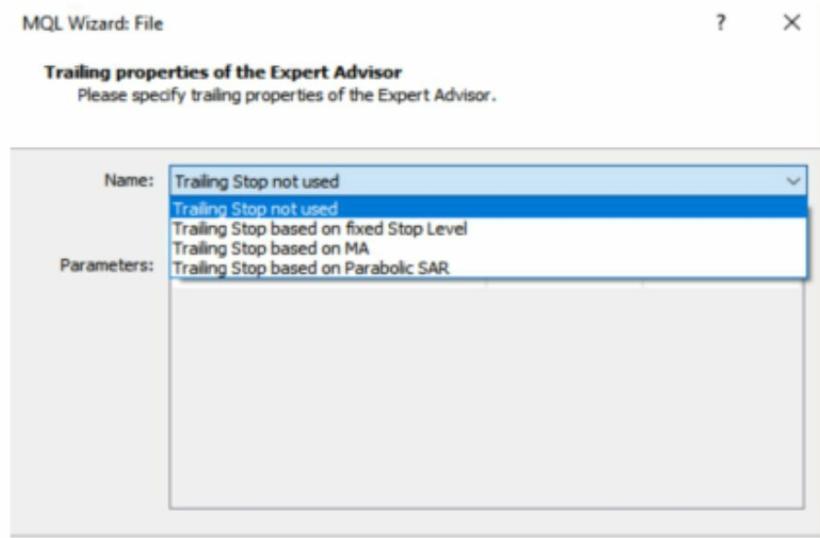
Where MACD Prediction = MACD Signal Weight * Significance of the MACD Model,

And SAR Forecast = SAR Signal Weight * SAR Model Significance

In our case, we set the signal weights to 1.

If the final forecast exceeds the threshold value, the expert will make a deal to buy or sell.

After defining the expert signals, you could define an algorithm for trailing an open position.



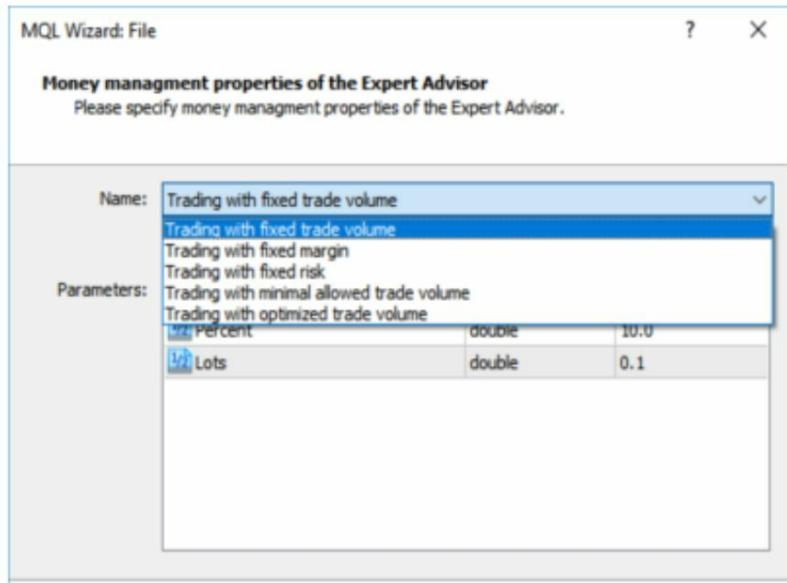
This is:

- Trailing an open position by a fixed "distance" (in points) when Stop Loss and Take Profit levels of an open position are moved at a fixed distance as a price moves in the direction of the open position.

When the price moves in the direction of an open position by a distance that exceeds the number of points corresponding to a Trailing Stop level, the expert changes the values of the Stop Loss level and Take Profit level (if the Trailing Profit level > 0).

- Trailing an open position by the moving average indicator's values on a previous bar.
- Trailing an open position by the Parabolic SAR indicator's values on a previous bar.

And the implementation files for trailing an open position are in the MQL5\Include\Expert\Trailing folder.



After defining the algorithm for trailing an open position, you could select an algorithm for managing capital and risks.

This is:

- Trading with fixed trade volume and.
- Trading with a fixed margin.

Here, a value of the lot is calculated by the `MaxLotCheck` function, which returns the maximum possible volume of a trade operation based on the free margin share (here, by default, it is 10%).

The next option is:

- Trading with fixed risk.

Here, a value of the lot is calculated as the ratio of the balance's proportion allocated for risk to the stop loss.

The next option is:

- Trading with minimal allowed trade volume.
- And Trading with optimized trade volume where a trading volume is defined by the results of previous trades.

Here, firstly, a lot of value is calculated by the `MaxLotCheck` function, which returns the maximum possible volume of a trade operation based on the free margin share (here, by default, 10%).

Then, in the event of a loss, the lot is reduced by the Decrease Factor factor (by default 3).

If a loss is equal to the specified percentage of the current capital, the unprofitable position is forcedly closed.

The files of the implementation of the capital and risk management algorithms are located in the MQL5\Include\Expert\Money folder.

After selecting a capital and risk management algorithm, an expert code is generated.

```

12 #include <Expert\Expert.mqh>
13 //---- available signals
14 #include <Expert\Signal\SignalMACD.mqh>
15 #include <Expert\Signal\SignalSAR.mqh>
16 //---- available trailing
17 #include <Expert\Trailing\TrailingFixedPips.mqh>
18 //---- available money management
19 #include <Expert\Money\MoneyFixedLot.mqh>
20 //+-----+
21 //| Inputs |+
22 //+-----+
23 //---- inputs for expert
24 input string      Expert_Title      = "ExpGen";    // Document name
25 ulong           Expert_MagicNumber = 31606;     //
26 bool            Expert_EveryTick   = false;       //
27 //---- inputs for main signal
28 input int        Signal_ThresholdOpen = 10;        // Signal threshold value to open [0...100]
29 input int        Signal_ThresholdClose = 10;        // Signal threshold value to close [0...100]
30 input double     Signal_PriceLevel  = 0.0;        // Price level to execute a deal
31 input double     Signal_StopLevel   = -50.0;      // Stop Loss level (in points)
32 input double     Signal_TakeLevel  = 50.0;       // Take Profit level (in points)
33 input int        Signal_Expiration = 4;          // Expiration of pending orders (in bars)
34 input int        Signal_MACD_PeriodFast = 12;      // MACD(12,24,9,PRICE_CLOSE) Period of fast EMA
35 input int        Signal_MACD_PeriodSlow = 24;      // MACD(12,24,9,PRICE_CLOSE) Period of slow EMA
36 input int        Signal_MACD_PeriodSignal = 9;      // MACD(12,24,9,PRICE_CLOSE) Period of averaging of difference
37 input ENUM_APPLIED_PRICE Signal_MACD_Applied = PRICE_CLOSE; // MACD(12,24,9,PRICE_CLOSE) Prices series
38 input double     Signal_MACD_Weight = 1.0;        // MACD(12,24,9,PRICE_CLOSE) Weight [0...1.0]
39 input double     Signal_SAR_Step   = 0.02;       // Parabolic SAR(0.02,0.2) Speed increment
40 input double     Signal_SAR_Maximum = 0.2;        // Parabolic SAR(0.02,0.2) Maximum rate
41 input double     Signal_SAR_Weight = 1.0;        // Parabolic SAR(0.02,0.2) Weight [0...1.0]
42 //---- inputs for trailing
43 input int        Trailing_FixedPips_StopLevel = -30; // Stop Loss trailing level (in points)
44 input int        Trailing_FixedPips_ProfitLevel = 50; // Take Profit trailing level (in points)
45 //---- inputs for money
46 input double     Money_FixLot_Percent = 10.0;     // Percent
47 input double     Money_FixLot_Lots   = 1.0;       // Fixed volume

```

The Expert Advisor code is based on using an instance of the CExpert class, the file of which is located in the MQL5\Include\Expert folder.

The threshold values Signal_ThresholdOpen and Signal_ThresholdClose of the final signal prediction are 10 by default.

Testing this expert with the MACD and PSAR signals on the EUR/USD hourly chart with different trailing and money management algorithms gives a negative expectation of winning.

Initial Deposit	10 000.00				
Total Net Profit	-1 421.26	Balance Drawdown Absolute	1 609.26	Equity Drawdown Absolute	1 699.65
Gross Profit	3 372.67	Balance Drawdown Maximal	1 948.32 (18.84%)	Equity Drawdown Maximal	2 124.60 (20.38%)
Gross Loss	-4 793.93	Balance Drawdown Relative	18.84% (1 948.32)	Equity Drawdown Relative	20.38% (2 124.60)
Profit Factor	0.70	Expected Payoff	-20.30	Margin Level	741.63%
Recovery Factor	-0.67	Sharpe Ratio	-0.12	Z-Score	-0.05 (3.99%)
AHPR	0.9980 (-0.20%)	LR Correlation	-0.88	OnTester result	0
GHPR	0.9978 (-0.22%)	LR Standard Error	274.99		

The optimization of such parameters as the `Trailing_FixedPips_StopLevel`, `Signal_MACD_Weight`, and `Signal_SAR_Weight` does nothing.

The expert remains unprofitable.

Let's try to create an expert and include all standard signals of the MQL5 Wizard in it.

When testing on the EUR/USD hourly chart, such an expert with equal weights of signals also shows a negative expectation of a win.

However, when optimizing the weights of the signals, the expert will come out in plus, and you could see combinations of signals with significant weights that give the best result.

You can also optimize the forecast threshold values along with the optimization of signal weights.

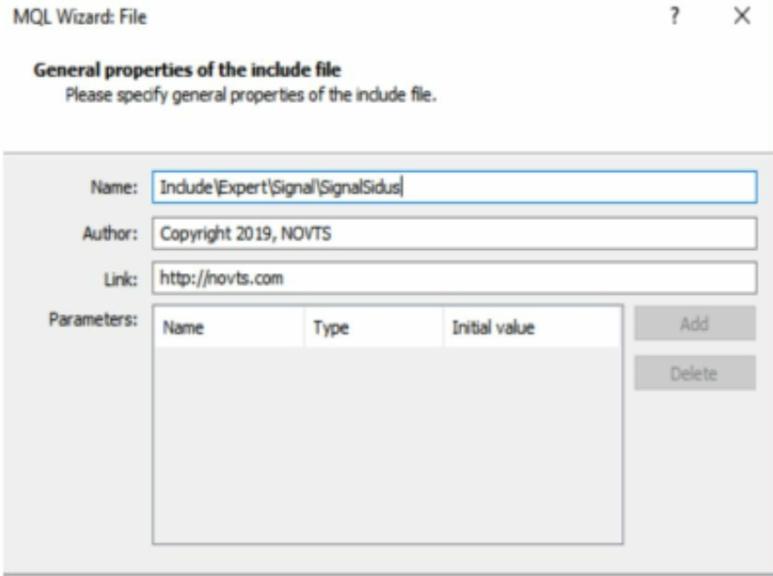
And based on the combinations of signals that bring the adviser to profit, we can build trading systems.

The modular structure of the adviser generated by the MQL5 Wizard allows you to include your own custom trading system signals module in the advisor.

As an example, let's consider the creation of a module of signals based on the Sidus trading system.



In the MetaEditor, let's click the New button and create an include file with the MQL5 Wizard.



Which will be placed in the MQL5\Include\Expert\Signal folder.
The include file should contain a class that extends the CExpertSignal class.

```
6 #property copyright "Copyright 2018, NOVTS"
7 #property link      "http://novts.com"
8
9 #include <Expert\ExpertSignal.mqh>
10
```

Therefore, the code should include the ExpertSignal file of the CExpertSignal class, using the include directive.

```
11 // wizard description start
12 //+-----
13 /// Description of the class
14 /// Title=Signals of the system 'Sidus'
15 /// Type=SignalAdvanced
16 /// Name=Sidus
17 /// ShortName=MA
18 /// Class=CSignalSidus
19 /// Page=
20 /// Parameter=NumberOpenPosition,int,5,Bars number checked to cross
21 /// Parameter=Pattern_0,int,80,Model 0
22 /// Parameter=Pattern_1,int,20,Model 1
23 /// Parameter=Pattern_2,int,80,Model 2
24 //+-----
25 // wizard description end
26 //+-----
27 /// Class CSignalSidus.
28 /// Purpose: Class of generator of trade signals based on
29 ///           the 'Sidus' system.
30 /// Is derived from the CExpertSignal class.
31 //+-----
```

Next, there should be information about the signal module intended for the MQL5 Wizard, which is used to recognize the signal module by the MQL5 Wizard when creating an expert in the add signals window, as well as when generating the expert code itself.

Without this information, the MQL5 Wizard simply will not add a signal to its interface, although the signal file will be located in the MQL5\Include\Expert\Signal folder.

After creating the signal module, the MetaEditor must be restarted in order for the signal to be added to the MQL5 Wizard.

Here, the Title line is displayed in the signal list of the MQL5 Wizard window, the Page line indicates the help topic and should be empty, and Parameter lines describe methods for setting the signal's parameters and are used when generating the expert code.

```

33 class CSignalSidus : public CExpertSignal
34 {
35 protected:
36     CiMA          m_ma18;           // object-indicator
37     CiMA          m_ma28;           // object-indicator
38     CiMA          m_ma5;            // object-indicator
39     CiMA          m_ma8;            // object-indicator
40     //--- adjusted parameters
41     int m_numberOpenPosition;
42     //--- "weights" of market models (0-100)
43     int          m_pattern_0;        // model 0 "5 WMA and 8 WMA cross the 18 EMA and the 28 EMA upward or top down" 80
44     int          m_pattern_1;        // model 1 "5 WMA crosses the 8 WMA upward or top down" 10
45     int          m_pattern_2;        // model 2 "18 EMA crosses the 28 EMA upward or top down" 80
46
47 public:
48     CSignalSidus(void);
49     ~CSignalSidus(void);
50     //--- methods of setting adjustable parameters
51     void          NumberOpenPosition(int value) { m_numberOpenPosition=value; }
52     //--- methods of adjusting "weights" of market models
53     void          Pattern_0(int value) { m_pattern_0=value; }
54     void          Pattern_1(int value) { m_pattern_1=value; }
55     void          Pattern_2(int value) { m_pattern_2=value; }
56     //--- method of verification of settings
57     virtual bool  ValidationSettings(void);
58     //--- method of creating the indicator and timeseries
59     virtual bool  InitIndicators(CIndicators *indicators);
60     //--- methods of checking if the market models are formed
61     virtual int   LongCondition(void);
62     virtual int   ShortCondition(void);
63 protected:
64     //--- method of initialization of the indicator
65     bool          InitMA(CIndicators *indicators);
66 };

```

Next, you can see the declaration of the parameters and methods of the signal module class.

Here, the `m_ma*` objects represent the moving averages used by the Sidus system, the `m_numberOpenPosition` parameter represents a number of bars at which the intersection of the moving averages will be checked.

To generate signals, we define three price forecast models - the intersection of the 5 WMA and 8 WMA moving averages the 18 EMA and 28 EMA moving averages, the intersection of the 5 WMA moving average the 8 WMA moving average, and the intersection of the 18 EMA moving average the 28 EMA moving average.

And here the `ValidationSettings` method checks the signal parameters for correctness, the `InitIndicators` method initializes the `m_ma*` objects.

The generated expert will receive the signals from the module using the methods `CheckOpenLong`, `CheckOpenShort`, `CheckCloseLong`, `CheckCloseShort`, `CheckReverseLong`, and `CheckReverseShort` of the class `CExpertSignal`.

However, these methods, in turn, form their return values based on the values that are returned by the `LongCondition` and `ShortCondition` methods of our module.

And just the `LongCondition` and `ShortCondition` methods operate on price

forecast models.

Therefore, it is sufficient to define these two methods in the signal module.

```
68 CSignalSidus::CSignalSidus(void) : m_numberOpenPosition(5),
69   m_pattern_0(80),
70   m_pattern_1(10),
71   m_pattern_2(80)
72 {
73 //--- initialization of protected data
74   m_used_series=USE_SERIES_OPEN+USE_SERIES_HIGH+USE_SERIES_LOW+USE_SERIES_CLOSE;
75 }
76 //-----+
77 //| Destructor
78 //+-----+
79 CSignalSidus::~CSignalSidus(void)
80 {
81 }
82 //+-----+
83 //| Validation settings protected data.
84 //+-----+
85 bool CSignalSidus::ValidationSettings(void)
86 {
87 //--- validation settings of additional filters
88   if(!CExpertSignal::ValidationSettings())
89     return(false);
90
91 //--- ok
92   return(true);
93 }
```

Next, we define a class constructor and class's methods.

```

173 int CSignalSidus::LongCondition(void)
174 {
175     int result=0;
176     int idx=StartIndex();
177 if(m_ma5.Main(idx)>m_ma8.Main(idx) && m_ma8.Main(idx)>m_ma18.Main(idx) && m_ma8.Main(idx)>m_ma28.Main(idx)) {
178     bool flagCross1=false;
179     bool flagCross2=false;
180     for (int i=(idx+1);i<m_numberOpenPosition;i++) {
181         if(m_ma5.Main(i)<m_ma18.Main(i) && m_ma5.Main(i)<m_ma28.Main(i)) {
182             flagCross1=true;
183         }
184         if(m_ma8.Main(i)<m_ma18.Main(i) && m_ma8.Main(i)<m_ma28.Main(i)) {
185             flagCross2=true;
186         }
187         if(flagCross1==true&&flagCross2==true) {
188             result=m_pattern_0;
189         }
190     if(m_ma5.Main(idx)>m_ma8.Main(idx)) {
191         bool flagCross=false;
192         for (int i=(idx+1);i<m_numberOpenPosition;i++) {
193             if(m_ma5.Main(i)<m_ma8.Main(i)) {
194                 flagCross=true;
195             }
196             if(flagCross==true) {
197                 result=m_pattern_1;
198             }
199             if(m_ma18.Main(idx)>m_ma28.Main(idx)) {
200                 bool flagCross=false;
201                 for (int i=(idx+1);i<m_numberOpenPosition;i++) {
202                     if(m_ma18.Main(i)<m_ma28.Main(i)) {
203                         flagCross=true;
204                     }
205                     if(flagCross==true) {
206                         result=m_pattern_2;

```

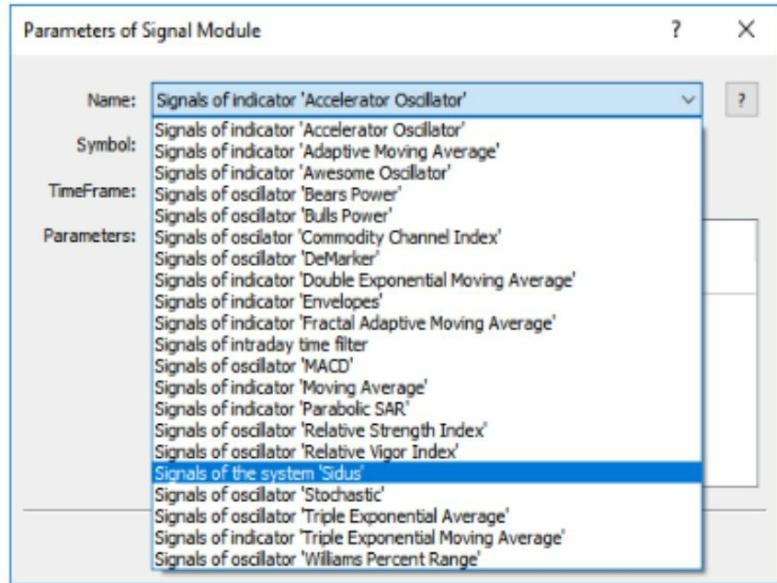
Here, in the LongCondition method, we implement three models of the price forecast to buy.

```

213 int CSignalSidas::ShortCondition(void)
214 {
215     int result=0;
216     int idx = startIndex();
217     if(m_ma5.Main(idx)<m_ma8.Main(idx) && m_ma8.Main(idx)<m_ma18.Main(idx) && m_ma8.Main(idx)<m_ma28.Main(idx)) {
218         bool flagCross1=false;
219         bool flagCross2=false;
220         for (int i=(idx+1);i<m_numberOpenPosition;i++) {
221             if(m_ma5.Main(i)>m_ma18.Main(i) && m_ma5.Main(i)>m_ma28.Main(i)) {
222                 flagCross1=true;
223             }
224             if(m_ma8.Main(i)>m_ma18.Main(i) && m_ma8.Main(i)>m_ma28.Main(i)) {
225                 flagCross2=true;
226             }
227             if(flagCross1==true&&flagCross2==true) {
228                 result=m_pattern_0;
229             }
230             if(m_ma5.Main(idx)<m_ma8.Main(idx)) {
231                 bool flagCross=false;
232                 for (int i=(idx+1);i<m_numberOpenPosition;i++) {
233                     if(m_ma5.Main(i)>m_ma8.Main(i)) {
234                         flagCross=true;
235                     }
236                     if(flagCross==true) {
237                         result=m_pattern_1;
238                     }
239                     if(m_ma18.Main(idx)<m_ma28.Main(idx)) {
240                         bool flagCross=false;
241                         for (int i=(idx+1);i<m_numberOpenPosition;i++) {
242                             if(m_ma18.Main(i)>m_ma28.Main(i)) {
243                                 flagCross=true;
244                             }
245                             if(flagCross==true) {
246                                 result=m_pattern_2;

```

And in the ShortCondition method, we implement three models of the price forecast for the sale.



Using the MQL5 Wizard, we generate the expert code based on the created module of signals.

```

26//--- inputs for main signal
27 input int  Signal_ThresholdOpen      =20;      // Signal threshold value to open [0...100]
28 input int  Signal_ThresholdClose     =20;      // Signal threshold value to close [0...100]
29 input double Signal_PriceLevel       =0.0;     // Price level to execute a deal
30 input double Signal_StopLevel        =50.0;    // Stop Loss level (in points)
31 input double Signal_TakeLevel        =50.0;    // Take Profit level (in points)
32 input int   Signal_Expiration        =4;       // Expiration of pending orders (in bars)
33 input int   Signal_MA_NumberOpenPosition =3;     // Sidus(5,80,20,80) Bars number checked to cross
34 input int   Signal_MA_Pattern_0       =60;      // Sidus(5,80,20,80) Model 0
35 input int   Signal_MA_Pattern_1       =10;      // Sidus(5,80,20,80) Model 1
36 input int   Signal_MA_Pattern_2       =100;     // Sidus(5,80,20,80) Model 2
37 input double Signal_MA_Weight        =1.0;     // Sidus(5,80,20,80) Weight [0...1.0]
38//--- inputs for trailing
39 input int   Trailing_FixedPips_StopLevel =100;   // Stop Loss trailing level (in points)
40 input int   Trailing_FixedPips_ProfitLevel=50;   // Take Profit trailing level (in points)

```

When optimizing the parameters of this expert on the EUR/USD hourly chart, we get that the expert works best with the following parameter values as shown on the slide.

Creating Indicator based on Expert Trading Signal Modules

As an example, let's create an indicator based on the two SignalMA and SignalMACD modules, which files are in the MQL5\Include\Expert\Signal directory.



In the MQL5 wizard, we create a basis of the indicator by selecting the Custom indicator option.

```

5 //+-
6 #property copyright "Copyright 2018, NOVTS"
7 #property link      "http://novts.com"
8 #property version   "1.00"
9 #property indicator_chart_window
10 //+-
11 //| Custom indicator initialization function
12 //+-
13 int OnInit()
14 {
15 //--- indicator buffers mapping
16
17 //---
18     return(INIT_SUCCEEDED);
19 }
20 //+-
21 //| Custom indicator iteration function
22 //+-
23 int OnCalculate(const int rates_total,
24                 const int prev_calculated,
25                 const datetime &time[],
26                 const double &open[],
27                 const double &high[],
28                 const double &low[],
29                 const double &close[],
30                 const long &tick_volume[],
31                 const long &volume[],
32                 const int &spread[])
33 {
34 //---
35
36 //--- return value of prev_calculated for next call
37     return(rates_total);
38 }
39 //+-

```

Now we need to specify properties of the indicator, as well as define the OnInit and OnCalculate functions.

When creating the indicator based on the modules of the adviser's trading signals, we could rely on the expert's code generated by the MQL5 Wizard.

```
159 //+-----+
160 //| "Tick" event handler function
161 //+-----+
162 void OnTick()
163 {
164     ExtExpert.OnTick();
165 }
166 //+-----+
167 //| "Trade" event handler function
168 //+-----+
169 void OnTrade()
170 {
171     ExtExpert.OnTrade();
172 }
```

During the work of such an expert, the OnTick function of the CExpert class is called in the OnTick expert's function.

```
639 //+
640 //| OnTick handler
641 //+
642 void CExpert::OnTick(void)
643 {
644 //--- check process flag
645 if(!m_on_tick_process)
646     return;
647 //--- updated quotes and indicators
648 if(!Refresh())
649     return;
650 //--- expert processing
651 Processing();
652 }
```

In this function, the Refresh function is first called, which updates prices of a symbol and values of the used indicators, and only then the Processing function is called, which receives trading signals and places orders.

Since the Refresh function is protected, and we need to update the data of the modules of trading signals in our OnCalculate function, so we create our own CExpertInd class that extends the CExpert class.

```

10 #include <Expert\Expert.mqh>
11 //+---+
12 //| |
13 //+---+
14 class CExpertInd : public CExpert
15 {
16   public:
17   virtual bool RefreshInd(void);
18 };
19 //+---+
20 //| Refreshing data for processing |
21 //+---+
22 bool CExpertInd::RefreshInd(void)
23 {
24   MqlDateTime time;
25 //--- refresh rates
26   if(!m_symbol.RefreshRates())
27     return(false);
28 //--- check need processing
29   TimeToStruct(m_symbol.Time(),time);
30   if(m_period_flags!=WRONG_VALUE && m_period_flags!=0)
31     if((m_period_flags&TimeframesFlags(time))==0)
32       return(false);
33   m_last_tick_time=time;
34 //--- refresh indicators
35   m_indicators.Refresh();
36 //--- ok
37   return(true);
38 }

```

Here, we simply transferred the code of the Refresh function from the CExpert class, making the Refresh function public.

Looking at the classes CSignalMA and CSignalMACD, we can see that the functions LongCondition and ShortCondition, which give market models, are called on the current bar.

But we need to call these functions throughout the history of the symbol.

In addition, we need the BarsCalculated function, which returns a number of calculated values in the signal module.

Therefore, we create the classes CSignalMAInd and CSignalMACDInd, extending the classes CSignalMA and CSignalMACD.

```

10 #include <Expert\Signal\SignalMA.mqh>
11 //-----+
12 //| |
13 //-----+
14 class CSignalMAInd : public CSignalMA
15 {
16     public:
17     virtual int      BarsCalculatedInd();
18     virtual int      LongConditionInd(int ind);
19     virtual int      ShortConditionInd(int ind);
20 };
21 |
22 //-----+
23 int CSignalMAInd:: BarsCalculatedInd()
24 return m_ma.BarsCalculated();
25 )
26
27 //-----+
28 //| "Voting" that price will grow. |
29 //-----+
30 int CSignalMAInd::LongConditionInd(int idx)
31 {
32
33     int result=0;
34
35 //--- analyze positional relationship of the close price and the indicator at the first analyzed bar
36     if(DiffCloseMA(idx)<0.0)
37     {

```

Here we define the BarsCalculated function, which returns a number of calculated values in the signal module.

And we changed the LongCondition and ShortCondition functions, passing an index of the bar on that the signal should be calculated as a parameter in the functions.

```

10 #include <Expert\Signal\SignalMA.mqh>
11 #include <Expert\Signal\SignalMACD.mqh>
12 //+
13 //|
14 //+
15 class CSignalMACDInd : public CSignalMACD
16 {
17     public:
18     virtual int      BarsCalculatedInd();
19     virtual int      LongConditionInd(int ind);
20     virtual int      ShortConditionInd(int ind);
21
22 };
23 //+
24 int CSignalMACDInd:: BarsCalculatedInd(){
25     return m_MACD.BarsCalculated();
26 }
27
28 //+
29 //| "Voting" that price will grow.
30 //+
31 int CSignalMACDInd::LongConditionInd(int idx)
32 {
33     int result=0;
34
35 //--- check direction of the main line
36     if(DiffMain(idx)>0.0)
37     {

```

And we do the same in the CSignalMACDInd class that extends the CSignalMACD class.

We define the BarsCalculated function that returns a number of calculated values in the signal module.

And we change the functions LongCondition and ShortCondition, passing an index of the bar on that the signal should be calculated as a parameter in the functions.

Now we can proceed to the code of our indicator.

```

9 #property indicator_chart_window
10
11 #include <Expert\ExpertInd.mqh>
12 #include <Expert\Signal\SignalMACDInd.mqh>
13 #include <Expert\Signal\SignalMAInd.mqh>
14
15 #property indicator_buffers 2
16 #property indicator_plots 1
17 #property indicator_type1 DRAW_COLOR_LINE
18 #property indicator_color1 clrBlack,clrRed,clrLawnGreen
19 #property indicator_style1 STYLE_SOLID
20 #property indicator_width1 2
21 double InBuffer[];
22 double ColorBuffer[];
23 int bars_calculated=0;
24
25 input int Signal_ThresholdOpen =20; // Signal threshold value to open [0...100]
26 input int Signal_ThresholdClose =20; // Signal threshold value to close [0...100]
27
28 input int Signal_MACD_PeriodFast =12; // MACD(12,24,9,PRICE_CLOSE) Period of fast EMA
29 input int Signal_MACD_PeriodSlow =24; // MACD(12,24,9,PRICE_CLOSE) Period of slow EMA
30 input int Signal_MACD_PeriodSignal =9; // MACD(12,24,9,PRICE_CLOSE) Period of averaging
31 input ENUM_APPLIED_PRICE Signal_MACD_Applied =PRICE_CLOSE; // MACD(12,24,9,PRICE_CLOSE) Prices series
32 input double Signal_MACD_Weight =1.0; // MACD(12,24,9,PRICE_CLOSE) Weight [0...1.0]
33 input int Signal_MA_PeriodMA =12; // Moving Average(12,0,...) Period of averaging
34 input int Signal_MA_Shift =0; // Moving Average(12,0,...) Time shift
35 input ENUM_MA_METHOD Signal_MA_Method =MODE_SMA; // Moving Average(12,0,...) Method of averaging
36 input ENUM_APPLIED_PRICE Signal_MA_Applied =PRICE_CLOSE; // Moving Average(12,0,...) Prices series
37 input double Signal_MA_Weight =1.0; // Moving Average(12,0,...) Weight [0...1.0]
38
39 CExpertInd ExtExpert;
40 CSignalMAInd *filter0 = new CSignalMAInd;
41 CSignalMACDInd *filter1 = new CSignalMACDInd;

```

We will draw the indicator in the form of a line at the opening prices, a line that will change color depending on the forecast for a rise or fall in price. And here, we took the input parameters and the initialization code from the code of the generated expert.

```

69 filter0.PeriodMA(Signal_MA_PeriodMA);
70 filter0.Shift(Signal_MA_Shift);
71 filter0.Method(Signal_MA_Method);
72 filter0.Applied(Signal_MA_Applied);
73
74 filter1.PeriodFast(Signal_MACD_PeriodFast);
75 filter1.PeriodSlow(Signal_MACD_PeriodSlow);
76 filter1.PeriodSignal(Signal_MACD_PeriodSignal);
77 filter1.Applied(Signal_MACD_Applied);
78
79 signal.AddFilter(filter0);
80 signal.AddFilter(filter1);
81 if(!ExtExpert.ValidationSettings())
82 {
83     //--- failed
84     ExtExpert.Deinit();
85     return(INIT_FAILED);
86 }
87 //--- Tuning of all necessary indicators
88 if(!ExtExpert.InitIndicators())
89 {
90     //--- failed
91     printf(__FUNCTION__+": error initializing indicators");
92     ExtExpert.Deinit();
93     return(INIT_FAILED);
94 }
95 //--- ok
96 //--- indicator buffers mapping
97 SetIndexBuffer(0,InBuffer,INDICATOR_DATA);
98 SetIndexBuffer(1,ColorBuffer,INDICATOR_COLOR_INDEX);
99
100 ArraySetAsSeries(InBuffer,true);
101 ArraySetAsSeries(ColorBuffer,true);

```

Since the signal modules work with data as with time series, let's apply the `ArraySetAsSeries` function to the buffers of our indicator.

```

146 bars_calculated=calculated;
147
148 ArraySetAsSeries(open,true);
149
150 ExtExpert.RefreshInd();
151
152 if(values_to_copy>1)
153 {
154
155 for (int i=0; i<values_to_copy; i++){
156
157 ColorBuffer[i]=0;
158 InBuffer[i]=open[i];
159
160 double result0=Signal_MA_Weight*(filter0.LongConditionInd(i)-filter0.ShortConditionInd(i));
161 double result1=Signal_MACD_Weight*(filter1.LongConditionInd(i)-filter1.ShortConditionInd(i));
162 double result=(result0+result1)/2;
163
164 if(result>=Signal_ThresholdOpen)
165 {
166 ColorBuffer[i]=2;
167 }
168
169 if(-result>=Signal_ThresholdOpen){
170 ColorBuffer[i]=-1;
171 }
172 }
173
174 //--- return value of prev_calculated for next call
175 return(rates_total);
176 }
```

In the OnCalculate indicator function, firstly, we update all the data, and then we obtain the weighted signals of the modules and compare them with the threshold value.

As a result, we obtain a forecast for an increase or decrease in price.

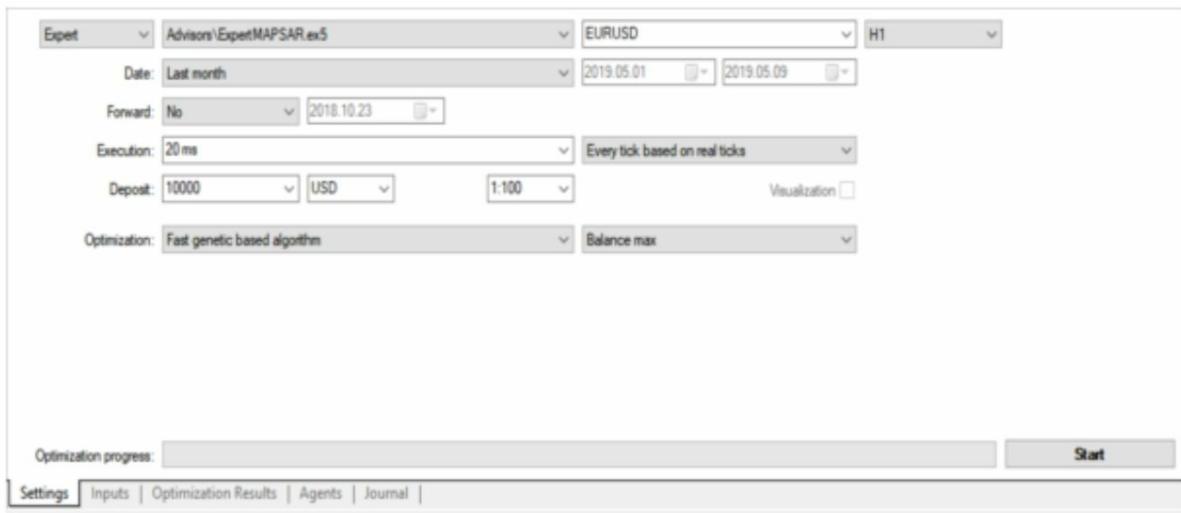


After attaching our indicator based on the modules of the adviser's signals to a symbol's chart, we could see how the indicator changes color depending on the forecast for a rise or fall in price.

Creating such an indicator is a way to visually see how the advisor works on the symbol's chart.

Genetic Algorithms

If we consider a task of creating a self-optimizing adviser,



then creating a self-optimizing advisor, at a certain stage of its work, an automatic call of the code is required, which re-optimizes parameters of the adviser on the history of the financial instrument, and then the adviser will continue its work with new parameters.

Since the advisor itself works on the current bar and does not use a symbol's history, the optimization code of the advisor's parameters should be based on the indicator's code, which in turn is based on the advisor's code.

To optimize the parameters of the adviser, or now the indicator's parameters that match the parameters of the adviser, you can use a full search, however, to reduce the optimization time, you can apply a genetic algorithm.

And firstly, we turn to the terminology.

A chromosome consists of a set of genes - a set of randomly selected parameters of the adviser in the permissible ranges.

Generation is a set of chromosomes.

Fitness function FF is the code that returns the value of the advisor that is optimized

$$p(x_i^t) = \frac{f(x_i^t)}{\sum_{j=1}^k f(x_j^t)}$$

$f(x_i)$ - fitness function value VFF

A chromosome consists of a set of genes that are a set of randomly selected parameters of an adviser in the permissible ranges.

And generation is a set of chromosomes.

The fitness function FF is a code that returns the value of an adviser by which optimization is performed, for example, a profit value.

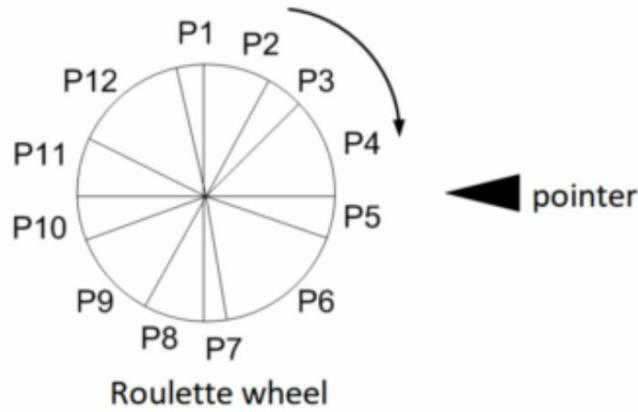
The value of the fitness function VFF is used to calculate the probability P of the chromosome, a member of the population, to be reproduced.

The initial population is formed randomly and the population size (number of individuals) is fixed and does not change during the operation of the entire algorithm.

Based on the probabilities of reproduction of the chromosomes of the initial population, a new population is formed, and so on and so forth, until the termination condition of the algorithm is fulfilled.

And each next population is formed from the previous one with the help of selection, crossing, and mutation.

For selection, you could use such algorithms as the roulette, tournament selection, and the elite selection.



$$M = P(x_i) * N$$

where N is the number of individuals in the population

When using the roulette algorithm, the roulette wheel is divided into sectors, the number of which coincides with the number of individuals of the population.

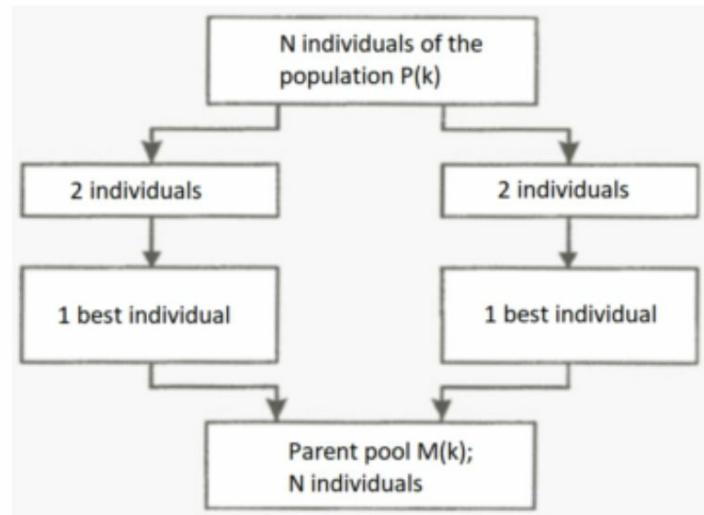
The area of each sector is proportional to the probability of reproduction of an individual of the population.

And the selection is made by rotating the roulette wheel.

Thus, the individual with the highest probability of reproduction falls into the next population the largest number of times.

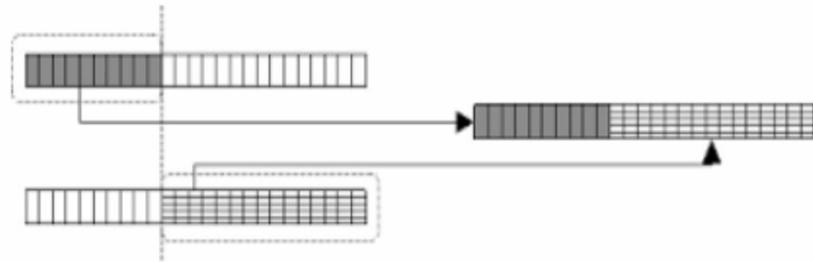
And the number of copies of an individual in the following population is determined by the formula as shown on the slide.

Where N is a number of individuals in the population.



In the tournament selection algorithm, two individuals are randomly selected from the population, of which the individual with the highest probability of reproduction remains.

And with the elite selection algorithm, some of the best individuals move to the next population unchanged, without participating in the selection and crossing.



After the selection, individuals are crossed, where two individuals are first selected, then a breakpoint is randomly determined in the range of the number of parameters of the fitness function, after which the individuals exchange chromosome segments.

And the crossing itself is performed with the probability $\frac{1}{2}$.

When a mutation is applied, firstly, a fitness function parameter is randomly selected, then it is modified.

And the mutation itself is performed with the probability 0.001.

```

1 //+-----+
2 //|          JQS UGA v1.3.1 |
3 //|          Copyright © 2010, JQS aka Joo. |
4 //|          http://www.mql4.com/users/joo |
5 //|          https://login.mql5.com/users/joo |
6 //+-----+
7 //The library of the "Universal Genetic Algorithm UGAlib"
8 //that uses the representation of a chromosome as real numbers.
9 //+-----+
10
11 //-----Global variables-----
12 double Chromosome[];           //A set of optimized function arguments - genes
13                           //for example, weight of the neural network, etc)-chromosome
14 int    ChromosomeCount     =0; //The maximum possible number of chromosomes in a colony
15 int    TotalOfChromosomesInHistory=0;//Total number of chromosomes in history
16 int    ChrCountInHistory   =0; //The number of unique chromosomes in the base of chromosomes
17 int    GeneCount          =0; //Number of genes in the chromosome
18
19 double RangeMinimum        =0.0;//Minimum of the search range
20 double RangeMaximum        =0.0;//Maximum of the search range
21 double Precision           =0.0;//Search step
22 int    optimizeMethod      =0; //1-minimum, any other - maximum
23
24 double Population  [[1000]]; //Population
25 double Colony      [[see]]; //The colony of children
26 int    PopulChromosCount =0; //the current number of chromosomes in the population
27 int    Epoch         =0; //Number of epochs without improvement
28 int    AmountStartsFF=0; //Number of runs of the fitness function
29 //-----+

```

The UGAlib MQL5 Community library of the genetic algorithm implementation uses a chromosome representation in the form of the Chromosome array, where the 0 index is a value of the fitness function, and the remaining indices are the parameters of the fitness function or chromosome genes.

The parameters of the fitness function are optimized in the same range from the RangeMinimum to the RangeMaximum.

Therefore, if the parameters being optimized have different ranges, they should be brought to the same range, as a rule, from 0.0 to 1.0.

The population of individuals is represented by a two-dimensional array Population, where rows are chromosomes.

The population is ranked by the value of the fitness function in such a way that its first member has the best value of the fitness function according to the optimization criterion.

And the population is divided into two parts.

```

//=====
// 1) Create protopopulation
ProtopopulationBuilding ();-----1)
//=====
// 2) Determine the fitness of each individual——2)
//For 1-nd colony
for (chromos=0;chromos<ChromosomeCount;chromos++)
    for (gene=1;gene<=GeneCount;gene++)
        Colony[gene][chromos]=Population[gene][chromos];

GetFitness(historyHromosomes);

for (chromos=0;chromos<ChromosomeCount;chromos++)
    Population[@][chromos]=Colony[@][chromos];

//For 2-nd colony
for (chromos=ChromosomeCount;chromos<ChromosomeCount*2;chromos++)
    for (gene=1;gene<=GeneCount;gene++)
        Colony[gene][chromos-ChromosomeCount]=Population[gene][chromos];

GetFitness(historyHromosomes);

for (chromos=ChromosomeCount;chromos<ChromosomeCount*2;chromos++)
    Population[@][chromos]=Colony[@][chromos-ChromosomeCount];

```

Initially, the entire population is populated by individuals with randomly selected genes in the range.

Then for these individuals, a value of the fitness function is calculated, which is placed in the 0 chromosome's index.

At the same time, the GetFitness function for calculating the fitness function value operates on a colony of descendants represented by the array Colony, which is two times smaller than the population size.

Thus, the population is populated by two colonies of descendants.

A colony of descendants has a size smaller than the size of the population so that after mutations and crosses to populate that part of the population that has the worst values of the fitness function.

At the same time, individuals, which are obtained as a result of mutations and crosses, just populate a colony of descendants.

```

//-----
// 3) Prepare the population for propagation -----3)
RemovalDuplicates();
//-----
// 4) Select the standard chromosome -----4)
for (gene=0; gene<GeneCount; gene++)
    Chromosome[gene]=Population[gene][@];
//-----
ServiceFunction();

//Main cycle of the genetic algorithm from 5 to 6
while (currentEpoch<=Epoch)
{
    //-----
    // 5) Operators of UGA -----5)
    CycleOfOperators
    (
        historyHromosomes,
    //---
        ReplicationPortion, //Portion of replication.
        NMutationPortion, //Portion of natural mutation.
        ArtificialMutation, //Portion of artificial mutation.
        GenoMergingPortion, //Portion of adoption of genes.
        CrossingOverPortion, //Portion of crossing over.
    //---
        ReplicationOffset, //Rate of shifting the interval borders
        NMutationProbability//Probability of mutation of each gene in %
    );
    //-----
    // 6) Compare genes of the best offspring with genes of the standard chromosome.
    // If the chromosome of offspring is betters than the standard one,
    // replace the standard one. -----6)
    //If the optimization mode is minimization
}

```

After the initial population filling, duplicates are removed using the RemovalDuplicates function, in which the population is also ranked by a value of the fitness function.

After the initial population has been prepared, a cycle of epochs is called - a cycle of birth of new populations, which continues until the number of epochs without improvement exceeds the threshold Epoch.

The criterion of improvement is the reference chromosome - the first individual of the population.

In a cycle of epochs, the population is modified by such operations as replication, natural mutation, artificial mutation, gene borrowing, and crossing, which are used for settling a new colony of descendants, which then replaces the part of the population that has the worst values of the fitness function.

In the cycle of settling a new colony of descendants, in the function CycleOfOperators, the operators of replication, natural mutations, artificial mutations, gene borrowing, and crossing are chosen randomly, depending on their share from 0 to 100.

After creating a new population, it also removes duplicates, and it is ranked by a value of the fitness function.

Here, the replication consists in choosing two chromosomes of a population

and creating on their basis a new chromosome, for which genes are randomly selected in an extended range using a shift from the gene of the first individual to the left and from the gene of the second individual to the right.

The choice of two parents from the population is carried out using the roulette algorithm mentioned above.

The natural mutation is performed by selecting one parent from a population, using the roulette algorithm, and replacing its genes with genes randomly selected in the range from the RangeMinimum to the RangeMaximum.

And the gene replacement is performed with the NMutationProbability probability from 0.0 to 100.0.

In the artificial mutation, two parents are selected from the population using the roulette algorithm, and the descendant genes are randomly selected from the range unoccupied by parents genes on the numerical line in the ranges from the RangeMinimum to the shift from the first individual gene to the right and from the shift of the second individual to the left to the RangeMaximum.

When genes are borrowed, for the first gene of the descendant, the parent is selected from the population using the roulette algorithm, and the first gene is taken from it, then the second parent is selected for the second gene, and the second gene is taken, and so on.

When crossing, two parents are selected from the population using the roulette algorithm, and the genes of the descendant are formed by exchanging segments of mom and dad chromosomes.

The complete implementation code of the genetic algorithm can be found in the UGAlib file.

To use the UGAlib library, you need to write two FitnessFunction and ServiceFunction functions.

The FitnessFunction function receives a chromosome index as an input and calculates the value for it, according to which the chromosome genes are optimized.

The ServiceFunction function can output the value of the fitness function and the rest of the genes of the reference chromosome during each optimization pass.

As an example, let's consider the optimization of parameters of the indicator created in the previous lecture.

```

187 if(result>=Signal_ThresholdOpen)
188 {
189 ColorBuffer[i]=2;
190 if(flagSell==true){
191 flagSell=false;
192 priceStopSell=InBuffer[i];
193 profitSell=(priceStopSell-priceSell-sp)*10000;
194 if(profitSell>0)(countProfitSellPlus++);else(countProfitSellMinus++);
195 profitTotalSell=profitTotalSell+profitSell;
196 }
197 if(flagBuy==false){
198 flagBuy=true;
199 priceBuy=InBuffer[i];
200 }
201 }
202
203 if(-result>=Signal_ThresholdOpen){
204 ColorBuffer[i]=1;
205 if(flagBuy==true){
206 flagBuy=false;
207 priceStopBuy=InBuffer[i];
208 profitBuy=(priceStopBuy-priceBuy-sp)*10000;
209 if(profitBuy>0)(countProfitBuyPlus++);else(countProfitBuyMinus++);
210 profitTotalBuy=profitTotalBuy+profitBuy;
211 }
212 if(flagSell==false){
213 priceSell=InBuffer[i];
214 flagSell=true;
215 }
216 }
217 }
218 //Print(" ProfitBuy ", profitTotalBuy," countProfitBuyPlus ",countProfitBuy!
219 //Print(" ProfitSell ", profitTotalSell," countProfitSellPlus ",countProfit!
220

```

We modify the indicator's code in such a way as to calculate virtual trades for the purchase and sale of a financial instrument based on the indicator's signals.

Here, we added the calculation of a virtual profit and the counter of winning trades.

And let's write a fitness function based on this indicator.

```
7 void FitnessFunction(int chromos)
8 {
9
10 double _MACD_Weight=0.0;
11 double _MA_Weight=0.0;
12 double sum=0.0;
13 int cnt=1;
14
15 while (cnt<=GeneCount)
16 {
17     _MACD_Weight=Colony[cnt][chromos];
18     cnt++;
19     _MA_Weight=Colony[cnt][chromos];
20     cnt++;
21 }
```

We optimize the weights of MA and MACD signals to get a maximum profit.

```
1 //+-----+ GenScript.mq5 |
2 //| Copyright 2018, NOVTS |
3 //| http://novts.com |
4 //|-+
5 //+-----+
6 #property copyright "Copyright 2018, NOVTS"
7 #property link      "http://novts.com"
8 #property version   "1.00"
9 #include <MAHACDFitness.mqh>
10 #include <UGAlib.mqh>
11
12 double ReplicationPortion_E = 100.0; // Replication share.
13 double NMutationPortion_E = 10.0; // The proportion of natural mutation.
14 double ArtificialMutation_E = 10.0; // The proportion of artificial mutation.
15 double GenoMergingPortion_E = 20.0; // Share of gene borrowing.
16 double CrossingOverPortion_E = 20.0; // Crossover share.
17 // ---
18 double ReplicationOffset_E = 0.5; // Interval offset factor
19 double NMutationProbability_E = 5.0; // The probability of mutation of each gene in %
```

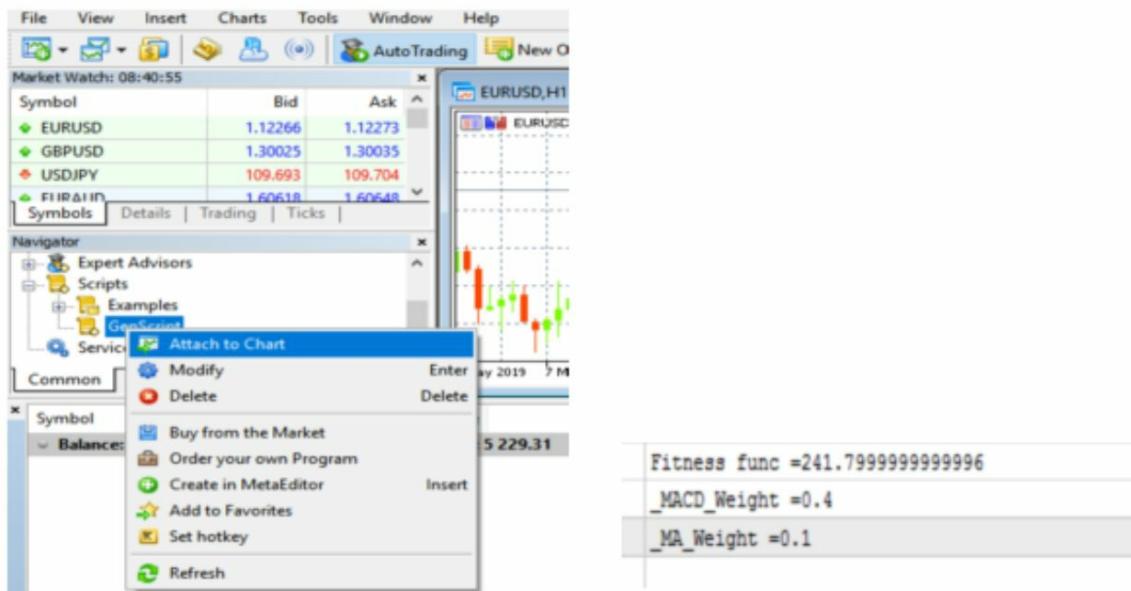
And we write a script that optimizes parameters of the indicator using a genetic algorithm.

```

42 void OnStart()
43 {
44 //...
45 ChromosomeCount = 10; // Number of chromosomes in the colony
46 GeneCount = 2; // number of genes
47 Epoch = 50; // Number of eras without improvement
48 // ...
49 RangeMinimum = 0.0; // Minimum search range
50 RangeMaximum = 1.0; // Maximum search range
51 Precision = 0.1; // Required accuracy
52 Optimizemethod = 2; //optim.:1-Min, other-Max
53 ArrayResize(Chromosome, GeneCount+1);
54 ArrayInitialize(Chromosome, 0);
55
56 // local variables
57 int time_start=(int)GetTickCount(),time_end=0;
58 //-----
59
60 // Run the main function of the UGA
61 UGA
62 (
63 ReplicationPortion_E, // Replication share
64 NMutationPortion_E, // Proportion of natural mutation
65 ArtificialMutation_E, // Share of artificial mutation
66 GenoMergingPortionor_E, // Share of gene borrowing
67 CrossingOverPortion_E, // Grossing over share
68 // ...
69 ReplicationOffset_E, // Interval offset factor
70 NMutationProbability_E // Probability of mutation of each gene in %
71 );
72 //-----
73 time_end=(int)GetTickCount();
74 //-----
75 Print(time_end-time_start," ms - Execution time");

```

Here, in the OnStart script function, we call the main UGA function of the genetic algorithm, which uses the FitnessFunction and ServiceFunction functions written by us in the included MAMACDFitness file.



By running the script, by attaching it to a symbol's chart, we get the following result that gives us the optimal signal's weights.

The indicator used is based on the use of the CiMA and CiMACD classes that have problems with the depth of history, therefore the size parameter in the fitness function cannot be large.

Therefore, we rewrite the indicator using a lower-level interface.

```
12 #include <Expert\Signal\SignalMA.mqh>
13 class CSignalMAIndLow : public CSignalMA
14 {
15     public:
16     virtual int      BarsCalculatedInd();
17     virtual int      LongConditionInd(int ind, int amount, double close, double open, double low);
18     virtual int      ShortConditionInd(int ind, int amount, double close, double open, double high);
19 };

21 //+-----+
22 #include <Expert\Signal\SignalMACD.mqh>
23
24 class CSignalMACDIndLow : public CSignalMACD
25 {
26     public:
27     virtual int      BarsCalculatedInd();
28     virtual int      LongConditionInd(int ind, int amount, double &low[], double &high[]);
29     virtual int      ShortConditionInd(int ind, int amount, double &low[], double &high[]);
30     protected:
31     int              StateMain(int ind,double &Main[]);
32     bool             ExtState(int ind, double &Main[], double &low[], double &high[]);
33 };
```

Let's create the signals classes CSignalMACDIndLow and CSignalMAIndLow.

```

29 //-----+
30 //| "Voting" that price will grow. |
31 //-----+
32 int CSignalMAIndLow::LongConditionInd(int idx, int amount, double close, double open, double low)
33 {
34     int handle=m_ma.Handle();
35     double iMABuffer[];
36     if(CopyBuffer(handle,0,0,amount,iMABuffer)<0)
37     {
38         PrintFormat("Failed to copy data from iMA indicator, error code %d",GetLastError());
39     }
40     return(-1);
41 }
42 ArraySetAsSeries(iMABuffer,true);
43
44 int result=0;
45
46 double DiffCloseMA = close - iMABuffer[idx];
47 double DiffOpenMA = open - iMABuffer[idx];
48 double DiffMA = iMABuffer[idx] - iMABuffer[idx+1];
49 double DiffLowMA = low - iMABuffer[idx];

```

And in the classes, in the LongCondition and ShortCondition methods, we do not use the CiMA and CiMACD classes but use the handles of indicators.

```
188 for (int i=0; i<(values_to_copy-2); i++) {
189
190 ColorBuffer[i]=0;
191 InBuffer[i]=open[i];
192
193 double result0=Signal_MA_Weight*(filter0.LongConditionInd(i, values_to_copy, close[i], open[i],
194 low[i])-filter0.ShortConditionInd(i, values_to_copy, close[i], open[i], high[i]));
195
196 double result1=Signal_MACD_Weight*(filter1.LongConditionInd(i, values_to_copy,
197 _low, _high)-filter1.ShortConditionInd(i, values_to_copy, _low, _high));
198
199 double result=(result0+result1)/2;
```

Accordingly, in the indicator, we call these rewritten functions.

```
22 int handleInd;
23 double BufferInd[];
24 double BufferColorInd[];
25
26 handleInd=iCustom(NULL,0,"IndSignalsLow",
27 Signal_ThresholdOpen,
28 Signal_ThresholdClose,
29 Signal_MACD_PeriodFast,
30 Signal_MACD_PeriodSlow,
31 Signal_MACD_PeriodSignal,
32 Signal_MACD_Applied,
33 _MACD_Weight,
34 Signal_MA_PeriodMA,
35 Signal_MA_Shift,
36 Signal_MA_Method,
37 Signal_MA_Applied,
38 MA_Weight
39 );
```

And let's include this rewritten indicator in the fitness function.

And run the script.

The depth of the history increases, but at the same time, the optimization time increases significantly - an order of magnitude, with the same result by 1000 bars.

Therefore, for self-optimization of the adviser, we could use the first version of the indicator with a history depth of 1000 bars.

Using the MQL5 wizard, we generate the advisor's code based on the MA and MACD signals and add self-optimization of the Signal_MA_Weight and Signal_MACD_Weight parameters using the above fitness function.

```

210 void OnTick()
211 {
212     if(AccountInfoDouble(ACCOUNT_BALANCE)>balance) balance=AccountInfoDouble(ACCOUNT_BALANCE);
213     double bd = ((balance-AccountInfoDouble(ACCOUNT_BALANCE))/balance)*100;
214     if(bd>10){
215         UGA
216     (
217         ReplicationPortion_E, // Replication share.
218         NMutationPortion_E, // Proportion of natural mutation.
219         ArtificialMutation_E, // Share of artificial mutation.
220         GenoMergingPortionor_E, // Share of gene borrowing.
221         CrossingOverPortionor_E, // Crossing over share.
222         // ---
223         ReplicationOffset_E, // nterval offset factor
224         NMutationProbability_E // Probability of mutation of each gene in %
225     );
226     double _MACD_Weight=0.0;
227     double _MA_Weight=0.0;
228     int cnt=1;
229     while (cnt<=GeneCount)
230     {
231         _MACD_Weight=Chromosome[cnt];
232         cnt++;
233         _MA_Weight=Chromosome[cnt];
234         cnt++;
235     }
236     filter0.Weight(_MA_Weight);
237     filter1.Weight(_MACD_Weight);
238
239 balance=AccountInfoDouble(ACCOUNT_BALANCE);
240 }

```

Here, in the OnTick function, when the balance drawdown threshold is exceeded, the UGA function of the genetic algorithm is called, which optimizes the signal weights.

And in the fitness function, before copying the indicator buffers, we add a Sleep call for the indicator to be calculated.

When testing the adviser on EUR/USD pair on the hour symbol's chart without self-optimization, we get the following result.

Total Net Profit:	26.00
Balance Drawdown Absolute:	4 431.00
Equity Drawdown Absolute:	4 550.00
Gross Profit:	79 532.00
Balance Drawdown Maximal:	5 862.00 (50.08%)
Equity Drawdown Maximal:	5 956.00 (50.57%)
Gross Loss:	-79 506.00
Balance Drawdown Relative:	50.08% (5 862.00)
Equity Drawdown Relative:	50.57% (5 956.00)
Profit Factor:	1.00
Expected Payoff:	0.04

With the advisor's self-optimization enabled, we get the following result when testing.

Total Net Profit:	384.00
Balance Drawdown Absolute:	504.00
Equity Drawdown Absolute:	612.00
Gross Profit:	4 004.00
Balance Drawdown Maximal:	1 276.00 (10.94%)
Equity Drawdown Maximal:	1 341.00 (11.44%)
Gross Loss:	-3 620.00
Balance Drawdown Relative:	10.94% (1 276.00)
Equity Drawdown Relative:	12.43% (1 333.00)
Profit Factor:	1.11
Expected Payoff:	24.00

As we can see, the expectation of profit has increased significantly.

Using MQL5 Library Classes and Functions

In the advisor's OnTick function, in order to trade only when a new bar appears on a symbol's chart, we used the datetime data structure, a local static variable, and the CopyTime function to track the bar opening time.

```
static datetime Old_Time;
datetime New_Time[1];
bool IsNewBar=false;

int copied=CopyTime(_Symbol,PERIOD_CURRENT,0,1,New_Time);
if(copied>0)
{
    if(Old_Time!=New_Time[0])
    {
        IsNewBar=true;
        Old_Time=New_Time[0];
    }
}
else
{
    Alert(" Time copy error, error code = : ",GetLastError());
    ResetLastError();
    return;
}

if(IsNewBar==false)
{
    return;
}
```

The same can be done with the iTime function.

```
if (!IsNewCandle())
return;

bool IsNewCandle(void)
{
    static datetime prevTime = 0;
    datetime currTime = iTime(_Symbol, PERIOD_CURRENT, 0);

    if (prevTime != currTime)
    {
        prevTime = currTime;
        return (true);
    }
    return (false);
}
```

Here, we also use the datetime data structure and a local static variable. But instead of the CopyTime function, we use the iTime function to get the bar opening time.

Now, to open a position, we used the MqlTradeRequest structure and the OrderSend function.

```

mrequest.action = TRADE_ACTION_DEAL;
mrequest.price = NormalizeDouble(latest_price.ask,_Digits);
mrequest.symbol = _Symbol;
mrequest.volume = Lot;
mrequest.magic = EA_Magic;
mrequest.type = ORDER_TYPE_BUY;
mrequest.type_filling = ORDER_FILLING_FOK;
mrequest.deviation=100;
mrequest.sl = NormalizeDouble(latest_price.bid - SLB,_Digits);
mrequest.tp = NormalizeDouble(latest_price.ask + ProfitBuy,_Digits);

if(!OrderCheck(mrequest,check_result))
{
    return;
}
else{
    if(OrderSend(mrequest,mresult)) {}

}

if(mresult.retcode==10009 || mresult.retcode==10008)
{
    priceBuy=mresult.price;
    slBuy= mrequest.sl;
}
else
{
    return;
}

```

And you can do the same with the CTrade and CPositionInfo classes.

```

CTrade trd;
CPositionInfo pos;

void OpenBuy(double volume)
{
    if (trd.Buy(volume, _Symbol, Ask(), 0, 0, ""))
    {
        ulong ticket = trd.ResultDeal();
        pos.SelectByTicket(ticket);

        double sl = 0;
        double tp = 0;

        if (inpStopLoss > 0)
            sl = ND(pos.PriceOpen()) - pos.PriceOpen() * (inpStopLoss / 100);

        if (inpTakeProfit > 0)
            tp = ND(pos.PriceOpen()) + pos.PriceOpen() * (inpTakeProfit / 100);

        if (sl == 0 && tp == 0)
            return;

        trd.PositionModify(ticket, sl, tp);
    }
}

```

Here, we use the Buy method of the CTrade class to open a long position with the specified volume, on the current symbol, at the price that is returned by the Ask function, and with zero stop loss and take profit.

Then we get the deal ticket using the ResultDeal method and fix the position CPositionInfo using the SelectByTicket method.

Then we calculate the stop loss and take profit based on the input parameters, and modify the open position using the PositionModify method of the CTrade class.

```
-----+
//| The function returns the Ask price
//+
double Ask(void)
{
    return (SymbolInfoDouble(_Symbol, SYMBOL_ASK));
}

-----+
//| The function normalizes the price
//+
double ND(double price)
{
    return (NormalizeDouble(price, _Digits));
}
```

Here, the slide shows the used functions that return a bid price from a broker and round the price to the specified accuracy.

```

void CheckTrailingStop(void)
{
    if (PositionSelect(_Symbol))
    {
        ulong ticket = pos.Identifier();
        long type = pos.PositionType();
        double op = pos.PriceOpen();
        double sl = pos.StopLoss();

        if (type == POSITION_TYPE_BUY)
            if (op + inpTrailingStop * _Point <= Bid())
            {
                if (op > sl || sl == 0)
                {
                    sl = Bid() - inpTrailingStopStep * _Point;
                    trd.PositionModify(ticket, sl, 0);
                }
                if (sl + inpTrailingStopStep * 2 * _Point <= Bid())
                {
                    sl = Bid() - inpTrailingStopStep * _Point;
                    trd.PositionModify(ticket, sl, 0);
                }
            }
        if (type == POSITION_TYPE_SELL)
            if (op - inpTrailingStop * _Point >= Ask())
            {
                if (op < sl || sl == 0)
                {
                    sl = Ask() + inpTrailingStopStep * _Point;
                    trd.PositionModify(ticket, sl, 0);
                }
                if (sl - inpTrailingStopStep * 2 * _Point >= Ask())
                {
                    sl = Ask() + inpTrailingStopStep * _Point;
                    trd.PositionModify(ticket, sl, 0);
                }
            }
    }
}

```

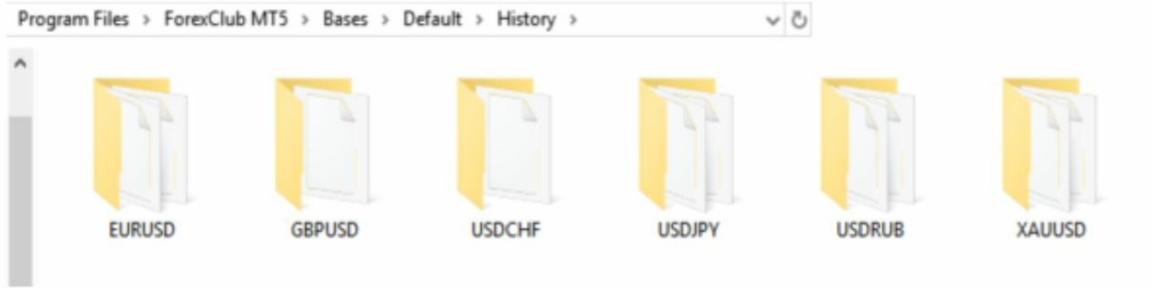
And using the methods of the CTrade and CPositionInfo classes, it is also possible to implement the trailing of buy and sell positions.

And once again about testing advisers

Any robot that you create will have input parameters for its work that will need to be optimized for a particular currency pair.

Moreover, the optimization of the adviser's parameters will have to be carried out periodically, adjusting them to the current market behavior.

It is impossible to create a universal robot that would produce the same results for any symbol and for any period of time.



To optimize the parameters of the adviser, it should be tested on the price history that the terminal downloads from the broker's server and saves it in the Bases\Default\History directory.

These price histories are downloaded from the trading server at the request of the terminal in the form of minute bars.

However, the thing is, if you install terminals from different brokers, you will find that each broker will provide their own price history.

And the same adviser will have different optimized parameters for different brokers.

You will not get universal optimized parameters of the adviser.

Moreover, the same adviser with the same parameters will trade differently at the terminals of different brokers.

If we compare the trading of the adviser in real time in the broker's terminal and then test the same adviser for the same period of time, the results will be the same.

So the broker with its history does not cheat.

With the sensitivity of the results of the adviser to the optimization of the parameters of the adviser, the question arises, how to configure the adviser to ensure profitable trading in the real market?

Anyone who finds the answer to this question will be fabulously rich.

One way to optimize an adviser is to select a section of history that you think will be similar to the price movement in the future and to optimize the adviser on this time period of the price history.

For automated trading using an adviser, you firstly do not need to code, but to create and form for yourself a market philosophy and trading on it.

Firstly, you need to understand the driving forces and mechanisms of the market.

Only after that, based on your philosophy, you could choose a trading strategy and implement it in the code of an adviser.

As a demonstration, I show the work of a trading robot created using this technology.

Trade 2 week	
Total Net Profit	1135.28
Gross Profit	2543.72
Gross Loss	-1408.44

Trade 3 week	
Total Net Profit	349.31
Gross Profit	2246.56
Gross Loss	-1897.25

Trade 4 week	
Total Net Profit	158.64
Gross Profit	1477.78
Gross Loss	-1319.14

Any adviser requires periodic adjustment of its parameters.

The values of the parameters of the adviser essentially reflect the mood in the market.

And since the mood in the market changes periodically, you need to reconfigure an adviser.

The advisor is configured using the terminal strategy tester.

Since the strategy tester can set different modes of delay and ticks, it is necessary to investigate the broker's server operation in order to understand which mode of the tester corresponds to the server operation.

You can investigate the broker's server as follows.

You can start trading with your advisor, and then after a certain period of time, test the adviser for this period of time, changing the parameters of the tester so that the testing coincides as much as possible with the real trading.

Here, in our example, we set up our adviser every week on the last week's period and start trading the thus configured advisor during the next week.

As we can see, our adviser consistently gives a profit.



However, in the period of no trend, in the period of high volatility, as shown in the 4-hour chart, a profitable adviser will issue a loss.

A sign of the approach of such a period may be a decrease in the profitability of an adviser.

In such periods it is better not to trade.