S. Kovalyov

Programming in Algorithmic Language

MQL4

Introductory Course

Nowadays, a personal computer became indispensable for everybody. The rapid development of Internet and performance of modern computers opened up new vistas in many fields of human activities. As early as ten years ago, the financial market trade was available only for banks and for a limited community of specialists. Today, anybody can join the world of professional traders and start independent trading at any time.

Hundreds of thousands of worldwide traders have already judged MetaTrader 4 Client Terminal on its merits. The use of its embedded programming language, MQL4, lifts traders to a new level of trading - to automated trading. Now, a trader can implement his or her ideas as an application program - write a custom indicator, a script to perform single operations, or create an Expert Advisor - an automated trading system (trading robot).

Many months' independent Expert Advisor's working without human intervention is a reality as of today. This fact has been proven by the annual Automated Trading Championship, in which hundreds of automated trading systems compete for 3 months.

Development of applications for MetaTrader 4 Client Terminal requires the knowledge of MQL4. Programming language MetaQuotes Language 4 is a fourth-generation language that has also been developed by MetaQuotes Software Corp. from their own many years' experience. MQL4 is the first programming language to consider all niceties of trading on financial markets.

Documentation on the language is available on the company's website; besides, MQL4.community grows and develops where you can communicate with other traders, read articles written by traders themselves, download MQL4 programs in their source codes.

This present textbook will help you create your own Expert Advisors, scripts and indicators and incarnate in them your ideas – your algorithms of profitable trading. The textbook is intended for a large number of readers without experience in programming that want to learn how to develop automated trading applications for MetaTrader 4 Client Terminal. The textbook is designed in such a method that to make learning MQL4 as convenient and consequent as possible.

Preface

It is a sort of difficulty to start writing a textbook on programming for beginners, because the area of knowledge under consideration involves some new concepts that are not based on anything previously known or usual.

Generally speaking, a problem of this kind may occur in any other field of knowledge. For example, point is known in mathematics as infinitesimal circle, whereas the circle itself is defined as a set of points ordered in a certain manner. As is easy to see, these terms are defined through each other. At the same time, this 'inadvertence' did not become a stumbling block for mathematics. Both circles, points, as well as other terms adopted in mathematics go well together. Moreover, everybody understands by insight what a point is and what a circle is.

It is easy to find out that the vast majority of ordinary terms have indeterminate boundaries. Some of those boundaries are so fuzzy that they cast some doubt on the existence of the very object or phenomenon defined by the term. However, the nature of man is that this strange (in terms of normal logic) situation does not come between a man and his existence and fruitful activities. After a term has been used for a certain amount of time, it takes on its complete sense for us. It's difficult to answer the question of how and why it happens this way. But it does. We only know that multiple reference to a term plays an important role in the remarkable process of terms learning.

The following tasks were set in this present work:

- unfolding the sense of new terms using well-known analogies;
- making the meaning of each term intuitively clear when it occurs for the first time;
- providing the readers with the necessary amount of information to characterize programs and programming.

For this purpose, the book contains many examples and figures. The text includes cross-references that allow the reader to get information on allied topics.

A few words about the presentation of materials. Some textbooks on programming invite their readers on the very first pages to print "Hello, world!" using a simple program. Their authors think that, as soon as their readers start learning programming, they should refer to program texts and gradually get used to how the programs may look, which will later facilitate their learning. However, this approach results in that the reader has to deal with several unknown terms at the same time and just to guess the content and properties of some lines in the program. This may result in a misconception and, consecutively, in vacancy in the reader's knowledge.

As I see it, it would be more effective to use a method where the reader goes to the next section in the textbook only after he or she has had a thorough grasp of the previous materials. In the framework of this method, the first program will be offered to the reader only after he or she has mastered all necessary terms and gained some insight into the basic principles of coding. This is the method this present textbook is based on.

"I am deeply indebted to Mr. Renat Fatkhullin, the CEO of MetaQuotes Software Corp., for his confidence, professional support and every possible assistance. I'm also grateful to the company's employees, Stanislav Starikov and Rashid Umarov, for their valuable advice and help in writing this book."

Sergey Kovalyov
http

To master knowledge given in the book, the reader has to be a PC user and to have some experience in working with MetaTrader 4 produced by MetaQuotes Software Corp.

Table of Contents

Introduction to MQL4 Programming

Before starting to study MQL4 programming, we will define the scope of our study. First of all it should be noted that programs discussed in this book can be used only as applications for working in MetaTrader 4 Client Terminal. Fig. 1 below shows the role of these programs in trade management. For a better understanding of the importance of these programs in trade management, let's look at Fig. 1.

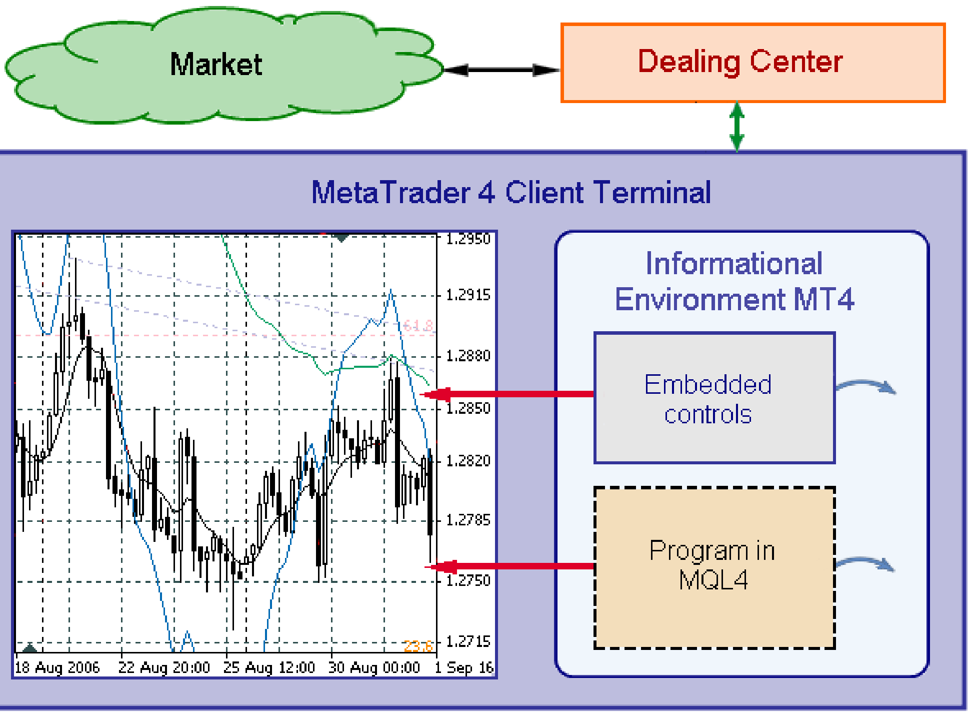


Fig. 1. A program in MQL4 as a part of MetaTrader 4 Client Terminal.

If you are interested in MQL4 programming, you must have got acquainted with the client terminal. Client terminal is a part of the online trading system. This system also includes a server installed in a dealing center. The dealing center in its turn is connected with other market participants - banks and financial institutions.

The client terminal includes informational environment - a set of parameters that inform about the market state and relations between a trader and dealing center. It contains information about current prices, limitations on the maximal and minimal order size, minimal distance of stop orders, allowance/prohibition of the automated trading and many other useful parameters characterizing the current state. The informational environment is updated when new ticks are received by the terminal (green line in Fig. 1).

Built-In Tools

The client terminal contains built-in tools that allow conducting technical analysis of market and execute manual trading management. For market analyzing one can use technical indicators and different line studies - support/resistance lines, trend channels, Fibonacci levels etc.

For manual trading management the order management toolbar is used. Using this toolbar a trader can open, close and modify orders. Besides, the terminal has the option of automated management of stop order position. A trader's actions with built-in trading management tools result in the formation of trade orders, which are sent to a server.

For more information about the client terminal please refer to "Userguide" (***ClientTerminal_folder\Terminal.chm***).

Programming Tools

Market analyzing and trade management in MetaTrader 4 Client Terminal is implemented with the help of programming tools. MQL4 language allows creating such programs. There are three types of applications created in MQL4 and intended for working in the client terminal:

- custom indicator - a program for graphical displaying of market regularities written according to an author's algorithm;
- Expert Advisor - a program that allows to automate a large part of trading operations and fully automate trading;
- script - a program for executing one-time actions including execution of trade operations.

Fig. 1 shows that the application has the same means of access to the client terminal informational environment as built-in tools for manual trading (blue arrows). It also can form managing influences (red arrows), passed to the client terminal. Programs of different types can be used simultaneously and exchange data. Using these applications a programmer can automatize a large part of trading operations or create a robot that will trade without a trader's interference.

Applications and manual management tools can be used in the client terminal simultaneously complementing each other.

The fundamental technical characteristic of trading using the online trading system MetaTrader is that all managing actions are produced in the client terminal and then sent to a server. Application programs (Expert Advisor, script, indicator) can work only as part of the client terminal provided it is connected to a server (dealing center). None of application programs are installed on the server.

The server allows only to process signals coming from a client terminal. If a client terminal is disconnected from the Internet or an application program (Expert Advisor or script) running in it does not generate any managing actions, nothing will happen on the server.

The scope of our study includes programs (Expert Advisors, scripts and custom indicators) that allow to conduct partially or fully automated trading and significantly widen the informational maintenance of trading (see Fig. 1). In this book you will find the description of program components and the main rules of creating and using programs. We will also consider in details examples of programs and parameters of informational environment of the client terminal, which are available to a program during its execution.

Programs for the automated trading possess much more potential possibilities than manual tools of trade management.

In the majority of cases a program allows to make a trader's job easier eliminating the necessity of a constant tracking of market situation sitting before a computer for a long period of time. It may also help to relieve nervous tension and lower the number of errors appearing in periods of extreme emotional tension. But the main thing is that using of the program method of trade management allows to develop one's own ideas and test them on historical data, select optimal parameters for applying these ideas and, finally, to implement a thought-out trading strategy.

Basics of MQL4

This section represents basic terms underlying programming language MQL4:

- Some Basic Concepts

  Such terms as 'tick' (a price change), 'control' in algorithms, 'comment' in programs are described. The main event when trading on financial markets is the change of price. This is why tick is an important event that makes the basic mechanisms of MQL4 programs run. What to do when a new tick incomes? What actions to take? This is control that moves to the forefront here. But don't forget to comment upon your code.

- Constants and Variables

  The terms of constants and variables are introduced, the difference between these terms is explained. As the term suggests, a constant is something continuous, set once for all. Unlike the constant, a variable is a programming code object that can modify its content. It is impossible to write a program without using unchangeable objects (constants) and/or objects that can be modified during the program execution (variables).

- Data Types

  Certain types of data are used in any programming language. The type of a variable is chosen according to its purpose. How can we declare a variable, how can we initialize it (preset its initial value)? A wrong choice of the type for a variable may slow down the program or even result in its wrong actions.

- Operations and Expressions

  Operations operate upon operands. What types of operations are there? What is typecasting used for? What are special features of operations on integers? Why is it important to remember about precedences of data of different types? Without knowing about the features of some operations, you can make subtle errors.

- Operators

  Simple and compound operators. A necessary action should not always be executed by a simple operator. If it is required that a group of operators is executed as one big operator, this group should be included into one compound operator. Requirements and specific examples of using operators are given.

- Functions

  The necessity of getting a simple code brings us to the term of Function. In order to use the function from different locations in the program, it is necessary to provide it with Function Parameters. We will consider the process of the custom function creation. The examples of using standard functions are given.

- Program Types

  Scripts, indicators and Expert Advisors are the types of MQL4 programs that allow you to cover practically the whole class of problems concerning trading in fincancial markets. It is necessary to understand the purposes of each type of programs in order to use MetaTrader 4 Client Terminal in the best way.

Some Basic Concepts

Thus, the subject of our interest is a program written in MQL4. Before we start a detailed presentation of the rules of writing programs, it is necessary to describe the basic concepts that characterize a program and its interrelations with information environment. The MetaTrader 4 Client Terminal is known to work online. The situation on financial markets changes continuously, this affects symbol charts in the client terminal. Ticks provide the client terminal with information about price changes on the market.

The Notion of Tick

Tick is an event that is characterized by a new price of the symbol at some instant.

Ticks are delivered to every client terminal by a server installed in a dealing center. As appropriate to the current market situation, ticks may be received more or less frequently, but each of them brings a new quote - the cost of one currency expressed in terms of another currency.

An application operating with the client terminal may work within a long period of time, for example, several days or weeks. Each application is executed according to the rules set for programs of a certain type. For example, an Expert Advisor (EA) does not work continuously all the time. An Expert Advisor is usually launched at the moment when a new tick comes. For this reason, we don't characterize tick as just a new quote, but as an event to be processed by the client terminal.

The duration of Expert Advisor's operation depends on what program code is included in it. Normal EAs complete one information-processing cycle during some tenths or hundredths of a second. Within this time, the EA can have processed some parameters, make a trading decision, provide the trader with some useful information, etc. Having finished this part of its work, the EA goes to waiting mode until a new tick comes. This new tick launches the Expert Advisor again, the program makes its appropriate operations again and returns to the waiting mode. The detailed description of how the appearance of a new tick influences program operation follows below.

The Notion of Control

Speaking about the code execution flow in a program, as well as its interaction with the client terminal, we will use the term of 'control'.

Control is a process of carrying out of actions preset by the program algorithm and the client terminal features. Control can be transferred within the program from one code line to another one, as well as from the program to the client terminal.

Control is transferred in a way similar to that of giving someone the floor to speak at a meeting. Like speakers address a meeting and then give the floor to others, the client terminal and the program transfer control to each other. At that, the client terminal dominates. Its status is higher than that of the program, like the authority of the chairman of a meeting is larger than those of an ordinary speaker.

Before the program is launched, the control is under the supervision of the client terminal. When a new tick is received, the client terminal transfers the control to the program. The program code starts to be executed at this moment.

The client terminal, after it has transferred the control to the program, does not stop its operation. It continues working with maximum performance during the entire period of time it is launched on PC. The program can only start operating at the moment when the client terminal has transferred control to it (like the chairman of a meeting controls the meeting all the time it is going on, whereas the current speaker takes the word for only a limited period of time).

After it has completed its operation, the program returns control to the client terminal and cannot be launched by its own. However, when the control has already been transferred to the program, it returns control to the client terminal by itself. In other words, the client terminal cannot return control from the program by itself. Dynamic actions of the user (for example, forced termination of the program) are an exemption.

When discussing the matters of performance and internal structures of programs, we are mostly interested in the part of control that is transferred within a program. Let's refer to Fig. 2 that shows the general nature of transferring control to, from and within a program. Circles shown in the figure characterize some small, logically completed fragments of a program, whereas the arrows between the circles show how control is transferred from one fragment to another.

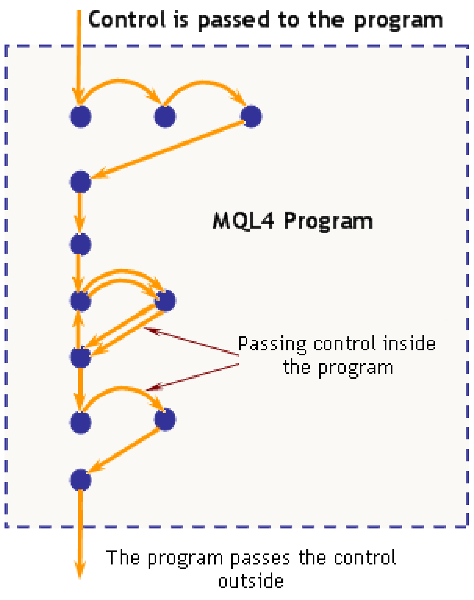

Control is passed to the program
MQL4 Program
Passing control inside the program
The program passes the control outside

Fig. 2. Transferring control in a program

A program that has accepted control from the client terminal (the executing program) starts to make some actions according to its inherent algorithm. The program contains program lines; general order of program execution consists in sequential transfer of control from one line to another in the top-down direction. What and according to what rules can be written in these lines will be considered below in all details.

Here, it is only important to emphasize that every logically completed fragment is executed - for example, some mathematical calculations are made, a message is displayed on the screen, a trade order is formed, etc. Until the current fragment of the program is executed, it retains the control. After it has been fully completed, the control is transferred to another fragment. Thus, control within a program is transferred from one logically completed fragment to another as they are executed. As soon as the last fragment is executed, the program will transfer (return) control to the client terminal.

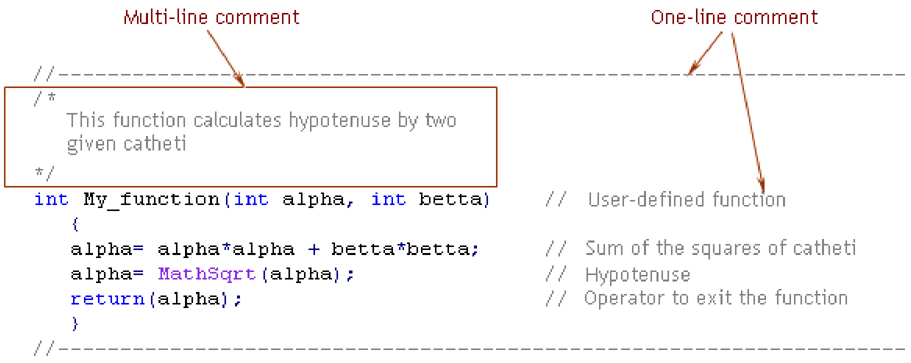
The Notion of Comment

A program consists in two types of records: those making the program itself and those being explanatory texts to the program code.

Comment is an optional and nonexecutable part of a program.

So, comment is an optional part of a program. It means that a ready program will work according to its code irrespective of whether there are comments in it or not. However, comments facilitate understanding of the program code very much. There are one-line and multi-line comments. A one-line comment is any sequence of characters following double slash (//). The sign of a one-line comment is ended by line feed. A multi-line comment starts with the characters of /* and is ended by */ (see Fig. 3).

Comments are used to explain the program code. A good program always contains comments.



```
//--------------------------------------------------------------------
/*
   This function calculates hypotenuse by two
   given catheti
*/
int My_function(int alpha, int betta)     //  User-defined function
   {
   alpha= alpha*alpha + betta*betta;      //  Sum of the squares of catheti
   alpha= MathSqrt(alpha);                //  Hypotenuse
   return(alpha);                         //  Operator to exit the function
   }
//--------------------------------------------------------------------
```

Fig. 3. Example of comments in a program.

Comments are widely used in coding. They are usually displayed in gray in codes. We will use comments, too, in order to explain our codes and make them more intelligible.

Constants and Variables

The terms of 'constant' and 'variable' are considered in one section since these terms are very close in themselves.

The Notion of Constant

Constant is a part of a program; an object that has a value.

The term of constant in a program is similar to the same term used in mathematical equations. It is an invariable value. To describe the nature of a constant used in an algorithmic language in as many details as possible, let us refer to well-known physical and mathematical constants.

The human race has discovered the constants, the values of which do not depend on us in any way. Those are, for example, in physics: free fall acceleration that is always equal to 9.8 m/s/s; in mathematics: Pi = 3.14. Constants of the kind cannot be considered similar to constants in an algorithmic language.

The term of constant is also used in mathematical equations. For example, in the equation of Y = 3 * X + 7, numbers 3 and 7 are constants. The values of such constants are fully dependent on the will of the person that has made the equation. This is the closest analogy of constants used in MQL4 programs.

A constant (as a value) is placed by a programmer in the code at the stage of its creation. The constant is characterized only by its value, so the terms of 'constant' and 'the value of a constant' are full synonyms.

Exemplary Constants:

```
37, 3.14, true, "Kazan"
```

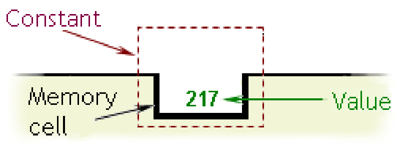


Fig. 4. A Constant in the memory of a PC.

The Properties of Constants

The property of a constant is its power to retain during the time of the program operation the value set by the programmer and set this value to the program when the program requests this (Fig. 5). For each constant in the program, the computer allocates a part of its memory of the necessary size. The value of a constant cannot be changed during execution of the program neither by programmer nor by computer (Fig. 6).

The value of the constant remains always the same.

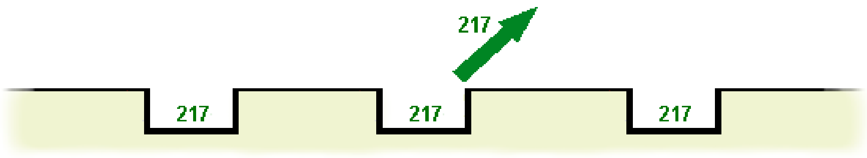


Fig. 5. The state of the memory cell of a constant when setting the value to the program.

The value of a constant cannot be changed during the program operation.

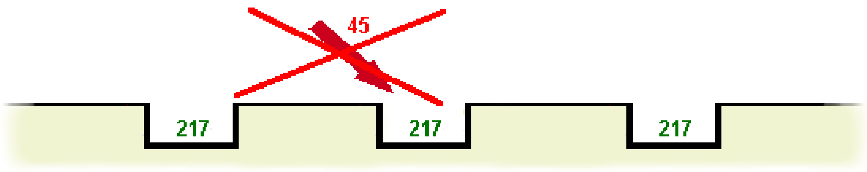


Fig. 6. It is impossible to change the value of a constant during the program operation.

The Notion of Variable

Variable is a program part that has a value and a name.

The term of variable in MQL4 is similar to that accepted in mathematics. The difference between them consists only in that the value of a variable in mathematics is always implied, whereas the value of variable in an executing program is stored in a special memory cell in the computer.

The term of 'variable identifier' is the full synonym of 'variable name'. The variable is put into the code text by its author at the stage of coding as a variable name. The name (identifier) of a variable can consist of letters, digits, underscore. However, it must start with a letter. MQL4 is case-sensitive, i.e., **S** and **s** are not the same.

Exemplary variable names: Alpha, alFa, beta, NuMbEr, Num, A_37, A37, qwerty_123
The following identifiers of variables represent the names of different variables: a_22 and A_22; Massa and MASSA.
Exemplary values of variables: 37, 3.14, true, "Kazan".

The Properties of Variable

The property of a variable is its capability to get a certain value from the program, retain it during the period of operation of the program and set this value to the program when requested by the program. For each variable in the program, the computer allocates a part of its memory of the necessary size. Let us refer to Fig. 7 and study the construction of a variable.

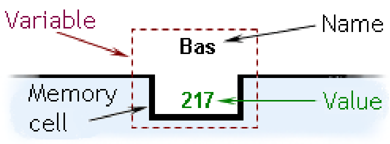


Fig. 7. A Variable in the memory of a computer.

There is a value of a variable in the memory cell of the computer. This value can be requested for processing and changed by the program. The name of a variable is never changed. When writing a code, the programmer can set any name for the variable. However, as soon as the ready program is launched, neither programmer nor the computer, nor the program has any technical feasibility to change the name of the variable.

If a program while being executed meets the name of a variable, the program refers to this variable in order to get its value for processing. If a program has referred to a variable, the latter one sets its value to the program. At that, the value of the variable remains the same, whereas the program gets the copy of the value contained in the memory cell allocated for this variable (Fig. 8).

When the value of a variable is set to a program, this value remains unchanged. The name of a variable will never be changed.

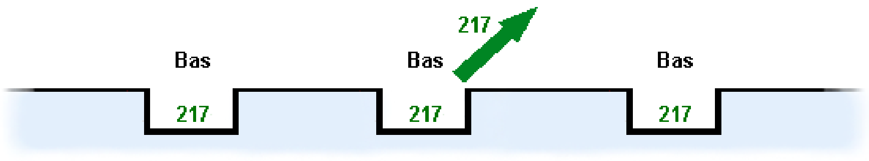


Fig. 8. The state of the memory cell of a variable when setting the value to the program.

A variable is not related to the executing program for a certain period of time. During this period, the program may refer to other variables or make necessary calculations. Between "sessions" of communication with the program, the variable retains its value, i.e., it keeps it unchanged.

According to the algorithm of the program, it can become necessary to change the value of a variable. In this case, the program sets to the variable its new value, whereas the variable gets this value from the program. All necessary modifications are made in the memory cell. This results in that the preceding value of the variable is deleted, whereas a new value of the variable set by the program takes its place (Fig. 9).

The value of a variable can be changed by the program. The name of the variable is always unchanged.

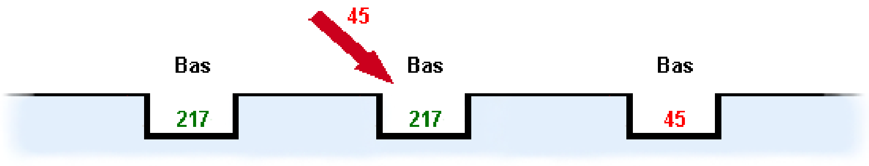


Fig. 9. The state of the memory cell of a variable when getting the value from the program.

Exemplary Constants and Variables in a Program

In a program, constants and variables can be found in operators. In the code below, A and B are variables, 7 and 3 are constants:

```
A = 7;                 // Line 1
B = A + 3;             // Line 2
```

Let us study how a program works with constants and variables. Executing these lines, the program will make the following steps:

Line 1:

1. Constant 7 sets its value to the program.
2. Variable A gets value 7 from the program.

Line 2:

1. The program has found an expression to the right from the equality sign and is trying to calculate it.
2. Constant 3 sets its value to the program.
3. The program refers to variable A by the name.
4. Variable A sets its value (7) to the program.
5. The program makes calculations (7 + 3).
6. Variable B gets value 10 from the program.

The value of a variable can be changed during the program operation. For example, there can be a line in the program that contains the following:

```
B = 33;                // Line 3
```

In this case, the following will be done at execution of the program:

1. Constant 33 sets its value to the program.
2. Variable B gets (new) value 33 from the program.

It is easy to notice that variable B gets value 10 at a certain stage of the program execution, and then it gets the value of 33. The name of variable B remains unchanged during all these events, whereas the value of the variable will change.

Fig. 10 shows constants and variables in the program code:

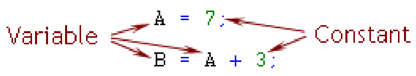


Fig. 10. A constant and a variable in a program.

Data Types

It is a common knowledge that only equitype values can be added or subtracted. For example, apples can be added to apples, but apples cannot be added to square meters or to temperature. Similar limitations can be found in most of modern algorithmic languages.

Like normal objects of life have certain types characterizing their color (red, blue, yellow, green), their taste (bitter, sour, sweet), amount (one and a half, two, seven), MQL4 uses data of different types. Speaking about data type, we will mean the type of the value of a constant, of a variable and the value returned by a function (the notion of function is considered in the section of Functions).

In MQL4, the following types are distinguished (for the values of constants, variables, and the values returned by functions):

- **int** - integers;
- **double** - real numbers;
- **bool** - Boolean (logical) values;
- **string** - values of string type;
- **color** - values of color type;
- **datetime** - values of date and time.

Type int

The values of **int** type are integers. This type includes values that are integer by their nature. The following values are integers, for example: amount of bars in the symbol window (16000 bars), amount of opened and pending orders (3 orders), distance in points between the current symbol price and the order Open Price (15 points). Amounts representing such objects as events can also be integers only. For example, the amount of attempts to open an order cannot be equal to one and a half, but only to one, two, three, etc.

There are 2 types of integer values:

- **Decimal** values can consist of digits from 0 to 9 and be either positive or negative: 10, 11, 12, 1, 5, -379, 25, -12345, -1, 2.
- **Hexadecimal** values can consist of Latin letters from A to F or from *a* to *f*, digits from 0 to 9. They must begin with 0x or 0X and take positive or negative values: 0x1a7b, 0xff340, 0xAC3 0X2DF23, 0X13AAB, 0X1.

Values of **int** type must be within the range from -2 147 483 648 to 2 147 483 647. If the value of a constant or a variable is beyond the above range, the result of the program operation will be void. The values of constants and variables of **int** type take 4 bytes of the memory of a computer.

An example of using a variable of **int** type in a program:

```
   int Art   = 10;                        // Example integer variable
   int B_27 = -1;                         // Example integer variable
   int Num   = 21;                        // Example integer variable
   int Max   = 2147483647;                // Example integer variable
   int Min   = -2147483648;               // Example integer variable
```

Type double

The value of **double** type are real numbers that contain a fractional part.

Example values of this type can be any values that have a fractional part: inclination of the supporting line, symbol price, mean amount of orders opened within a day.

Sometimes you can face problems designating variables when writing your code, i.e., it is not always clear to a programmer what type (**int** or **double**) the variable belongs to. Let us consider a small example:

A program has opened 12 orders within a week. What is the type of variable A that considers the mean amount of orders daily opened by this program? The answer is obvious: A = 12 orders / 5 days. It means that variable A = 2.4 should be considered in the program as **double**, since this value has a fractional part. What type should be the same variable A if the total amount of orders opened within a week is 10? You can think that if 2 (10 orders / 5 days = 2) has no fractional part, variable A can be considered as **int**. However, this reasoning is wrong. The current value of a variable can have a fraction part consisting of only zeros. It is important that that value of this variable is real by its nature. In this case, variable A has also to be of **double** type. The separating point must also be shown in the constant record in the program: A = 2.0

The values of real constants and variables consist of an integer part, a decimal point, and a fractional part. The values can be positive or negative. The integer part and the fractional part are made of digits from 0 to 9. The amount of significant figures after decimal point can reach the value of 15.

Example:
27.12 -1.0 2.5001 -765456.0 198732.07 0.123456789012345

The values of **double** type may range from -1.7 * e-308 to 1.7 * e308. In the computer memory, the values of constants and variables of **double** type take 8 bytes.

An example of using a variable of **double** type in a program:

```
   double Art     = 10.123;                  // Example real variable
   double B_27    = -1.0;                    // Example real variable
   double Num     = 0.5;                     // Example real variable
   double MMM     = -12.07;                  // Example real variable
   double Price_1 = 1.2756;                  // Example real variable
```

Type bool

The values of **bool** type are values of Boolean (logical) type that contain falsehood or truth.

In order to learn the notion of Boolean type, let's consider a small example from our everyday life. Say, a teacher needs to account the presence of textbooks of the pupils. In this case, the teacher will list all pupils on a sheet of paper and then will tick in a right line whether a pupil has a textbook or not. For example, the teacher may use tick marks and dashes in the table:

List of Pupils		Textbook on Physics	Textbook on Biology	Textbook on Chemistry
1	Smith	V	-	-
2	Jones	V	-	V
3	Brown	-	V	V
...	...	...	...	...
25	Thompson	V	V	V

The values in right columns can be of only 2 types: true or false. These values cannot be attributed to either of the types considered above since they are not numbers at all. They are not the values of color, taste, amount, etc., either. However, they bear an important sense. In MQL4, such values are named Boolean, or logical, values. Constants and variables of **bool** type are characterized through that they can only take 2 possible values: true (True, TRUE, 1) or false (False, FALSE, 0). The values of constants and variables of **bool** type take 4 bytes in the computer memory.

An example of using a variable of **bool** type in a program:

```
   bool aa    = True;          // Boolean variable aa    has the value of true
   bool B17   = TRUE;          // Boolean variable B17   has the value of true
   bool Hamma = 1;             // Boolean variable Hamma has the value of true

   bool Asd   = False;         // Boolean variable Asd   has the value of false
   bool Nol   = FALSE;         // Boolean variable Nol   has the value of false
   bool Prim  = 0;             // Boolean variable Prim  has the value of false
```

Type string

The value of **string** type is a value represented as a set of ASCII characters.

In our everyday life, a similar content belongs to, for example, store names, car makes, etc. A **string**-type value is recorded as a set of characters placed in double quotes (not to be mixed with doubled single quotes!). Quotes are used only to mark the beginning and the end of a string constant. The value itself is the totality of characters framed by the quotes.

If there is a necessity to introduce a double quote ("), you should put a reverse slash (\) before it. Any special character constants following the reverse slash (\) can be introduced in a string. The length of a string constant ranges from 0 to 255 characters. If the length of a string constant exceeds its maximum, the excessive characters on the right-hand side will be truncated and compiler will give the corresponding warning. A combination of two characters, the first of which is the reverse slash (\), is commonly accepted and perceived by most programs as an instruction to execute a certain text formatting. This combination is not displayed in the text. For example, the combination of \n indicates the necessity of a line feed; \t demands tabulation, etc.

The value of **string** type is recorded as a set of characters framed by double quotes: "MetaTrader 4", " Stop Loss", "Ssssstop_Loss", "stoploss", "10 pips". The string value as such is the set of characters. The quotes are used only to mark the value borders. The internal representation is a structure of 8 bytes.

An example of using a variable of **string** type in a program:

```
   string Prefix    = "MetaTrader 4";              // Example string variable
   string Postfix   = "_of_my_progr. OK";          // Example string variable
   string Name_Mass = "History";                   // Example string variable
   string text      ="Upper Line\nLower Line";     // the text contains line feed characters
```

Type color

The value of **color** type is a chromatic value.

The meaning of 'color' (blue, red, white, yellow, green, etc.) is a common knowledge. It is easy to imagine what a variable or a constant of **color** type may mean. It is a constant or a variable, the value of which is a color. It may seem to be a bit unusual, but it is very simple, generally speaking. Like the value of an integer constant is a number, the value of a color constant is a color.

The values of color constants and variables can be represented as one of three kinds:

- **Literals**
  The value of **color** type represented as a literal consists of three parts representing the numeric values of intensity of three basic colors: red, green and blue (RGB). The value of this kind starts with 'C' and is quoted by single quotes.

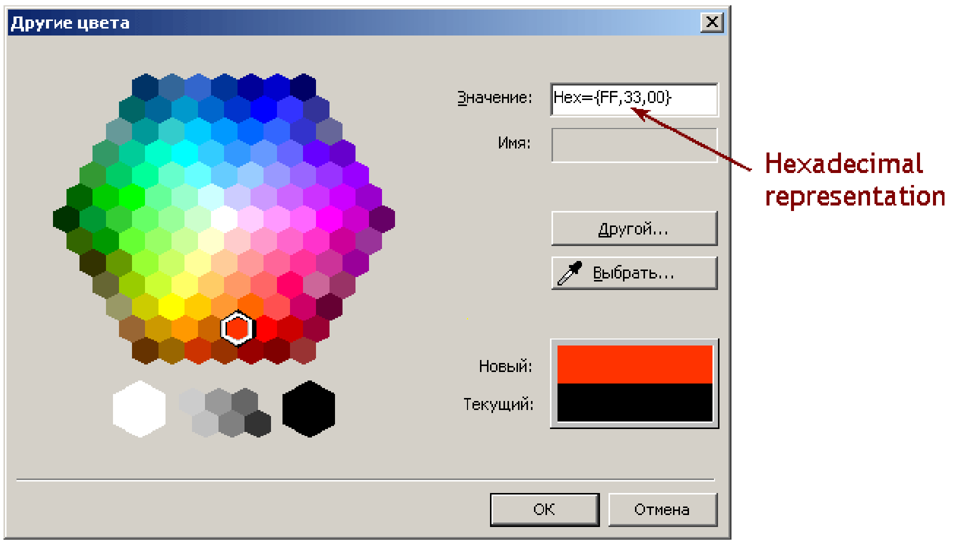
  The numeric values of RGB intensity range from 0 to 255 and can be recorded both decimally and hexadecimally.

  Examples: C'128,128,128' (gray), C'0x00,0x00,0xFF' (blue), C'0xFF,0x33,0x00' (red).

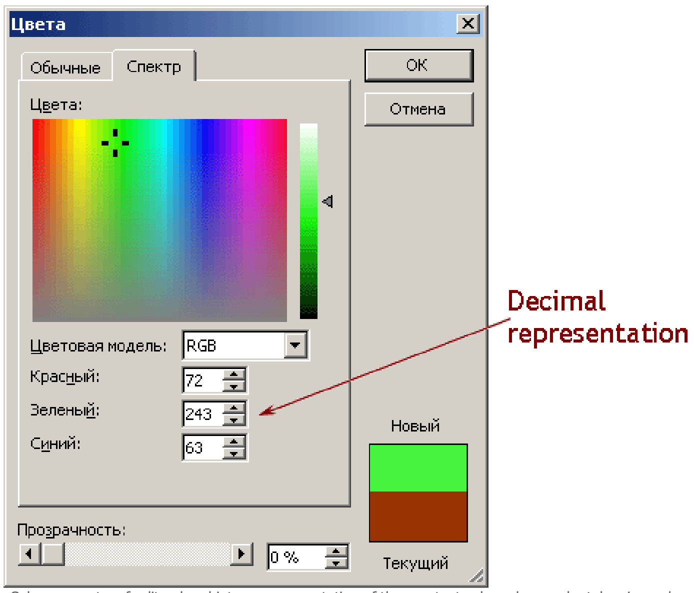
- **Integer Representation**
  Integer representation is recorded as a hexadecimal or a decimal number. A hexadecimal number is displayed as 0xRRGGBB where RR is the value of red intensity, GG - green, and BB - blue. Decimal constants are not directly reflected in RGB. They represent the decimal value of a hexadecimal integer representation.

  Representation of the values of color type as integers and as hexadecimal literals is very user-friendly. The majority of modern text and graphics editors provide information about the intensity of red, green and blue components in the selected value of color. You have just to select a color in your editor and copy the values found in its description to the corresponding color value representation in your code.

  Examples: 0xFFFFFF (white), 0x008000 (green), 16777215 (white), 32768 (green).



Значение: Hex={FF,33,00}



Цветовая модель: RGB
Красный: 72
Зеленый: 243
Синий: 63
Прозрачность: 0 %

Fig. 11. Color parameters for literal and integer representation of the constant color value can be taken in modern editors.

- **Color Names**
  The easiest way to set a color is to specify its name according to the table of web colors. In this case, the value of a color is represented as a wor corresponding with the color, for example, Red - the red color.

Black	DarkGreen	DarkSlateGray	Olive	Green	Teal	Navy	Purple
Maroon	Indigo	MidnightBlue	DarkBlue	DarkOliveGreen	SaddleBrown	ForestGreen	OliveDrab
SeaGreen	DarkGoldenrod	DarkSlateBlue	Sienna	MediumBlue	Brown	DarkTurquoise	DimGray
LightSeaGreen	DarkViolet	FireBrick	MediumVioletRed	MediumSeaGreen	Chocolate	Crimson	SteelBlue
Goldenrod	MediumSpringGreen	LawnGreen	CadetBlue	DarkOrchid	YellowGreen	LimeGreen	OrangeRed
DarkOrange	Orange	Gold	Yellow	Chartreuse	Lime	SpringGreen	Aqua
DeepSkyBlue	Blue	Magenta	Red	Gray	SlateGray	Peru	BlueViolet
LightSlateGray	DeepPink	MediumTurquoise	DodgerBlue	Turquoise	RoyalBlue	SlateBlue	DarkKhaki
IndianRed	MediumOrchid	GreenYellow	MediumAquamarine	DarkSeaGreen	Tomato	RosyBrown	Orchid
MediumPurple	PaleVioletRed	Coral	CornflowerBlue	DarkGray	SandyBrown	MediumSlateBlue	Tan
DarkSalmon	BurlyWood	HotPink	Salmon	Violet	LightCoral	SkyBlue	LightSalmon
Plum	Khaki	LightGreen	Aquamarine	Silver	LightSkyBlue	LightSteelBlue	LightBlue
PaleGreen	Thistle	PowderBlue	PaleGoldenrod	PaleTurquoise	LightGray	Wheat	NavajoWhite
Moccasin	LightPink	Gainsboro	PeachPuff	Pink	Bisque	LightGoldenrod	BlanchedAlmond
LemonChiffon	Beige	AntiqueWhite	PapayaWhip	Cornsilk	LightYellow	LightCyan	Linen
Lavender	MistyRose	OldLace	WhiteSmoke	Seashell	Ivory	Honeydew	AliceBlue
LavenderBlush	MintCream	Snow	White				

Constants and variables of **color** type take 4 bytes in the computer memory. An example of using such a variable in a program:

```
color Paint_1 = C'128,128,128';      // The value of gray   was assigned to the variable
color Colo    = C'0x00,0x00,0xFF';   // The value of blue   was assigned to the variable
color BMP_4   = C'0xFF,0x33,0x00'    // The value of red    was assigned to the variable

color K_12   = 0xFF3300;             // The value of red    was assigned to the variable
color N_3    = 0x008000;             // The value of green  was assigned to the variable
color Color = 16777215;              // The value of white  was assigned to the variable
color Alfa  = 32768;                 // The value of green  was assigned to the variable

color A       = Red;                 // The value of red    was assigned to the variable
color B       = Yellow;              // The value of yellow was assigned to the variable
color Colorit = Black;               // The value of black  was assigned to the variable
color B_21    = White;               // The value of white  was assigned to the variable
```

Type datetime

The value of **datetime** type is the values of date and time.

The values of this type can be used in programs to analyze the moment of beginning or ending of some events, including the releases of important news, workday start/finish, etc. The constants of date and time can be represented as a literal line consisting of 6 parts that represent the numeric value of year, month, day (or day, month, year), hour, minute, and second.

The constant is framed in single quotes and starts with 'D'. It is allowed to use truncated values: either without date or without time, or just an empty value. The range of values: from January 1, 1970, to December 31, 2037. The values of constants and variables of **datetime** type take 4 bytes in the computer memory. A value represents the amount of seconds elapsed from 00:00 of the 1st of January 1970.

An example of using a variable of **datetime** type in a program:

```
datetime Alfa    = D'2004.01.01 00:00';       // New Year
datetime Tim     = D'01.01.2004';             // New Year
datetime Tims    = D'2005.05.12 16:30:45';    // May 12, 2005 4:30:45 p.m.
datetime N_3     = D'12.05.2005 16:30:45';    // May 12, 2005 4:30:45 p.m.
datetime Compile = D'';                       // equivalent of D'[compilation date] 00:00:00'
```

Variable Declaration and Initialization

In order to avoid possible 'questions' by the program about what type of data this or that variable belongs to, it is accepted in MQL4 to specify the types of variables at the very start of a program explicitly. Before a variable starts to participate in any calculations, it should be declared.

Variable Declaration is the first mentioning of a variable in a program. At declaration of a variable, its type should be specified.

Variable Initialization means assignment to it a value corresponding with its type at its declaration. Every variable can be initialized. If no initial value is set explicitly, a numeric variable will be initialized by zero (0) and a string variable will be initialized by an empty line.

In MQL4, it is accepted to specify the types of variables explicitly at their declaration. The type of a variable is declared at the first mentioning of the name of this variable. When it is mentioned for the second and all subsequent times, its type is not specified anymore. In the course of execution of the program, the value of the variable can change, but its type and name remain unchanged. The type of a variable can be declared in single lines or operators.

A variable can be declared in a single line:

```
    int Var_1;                  // Variable declaration in a single line
```

This record means that there will be variable Var_1 (variable declaration as such) and the type of this variable will be **int** in the given program.

In one line, several variables of the same type can be declared:

```
    int Var_1, Box, Comm;       // Declaration of several variables in one line
```

This record means that there will be variables Var_1, Box and Comm, all of **int** type, used in the program. It means that the variables listed above will be considered by the program as variables of integer type.

Variables can also be initialized within operators:

```
    double Var_5 = 3.7;         // Variable initialization in an assignment operator
```

This record means that there will be variable Var_5 of **double** type used in program, the initial value of the variable being 3.7.

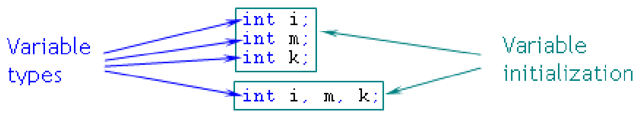
The type of variables is never specified anywhere in the subsequent lines of the program. However, every time the program calls a variable it "remembers" that this variable is of the type that has been specified at its declaration. As calculations progress in the program, the values of variables can change, but their type remains unchanged.

The name of a variable has no relation to its type, i.e., you cannot judge about the type of a variable by its name. A name given to a variable can also be used for variables of any types in different programs. However, the type of any variable can be declared only once within one program. The type of declared variables will not be changed during execution of the program.

Examples of Variable Declaration and Initialization

Variables can be declared in several lines or in a single line.

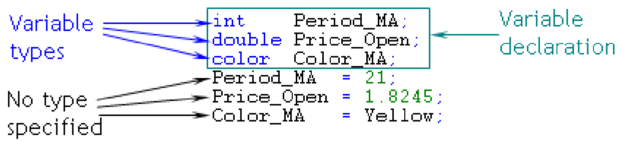
It is allowed to declare several variables of one type simultaneously. In this case, variables are listed separated by commas, a semicolon being put at the end of line.



```
int i;
int m;
int k;

int i, m, k;
```

Fig. 12. Example of variable declaration in a single line.

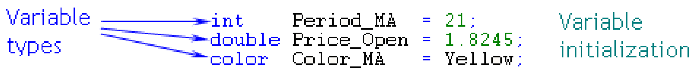
The type of variables is declared only once, at the first mentioning of the variable. The type will not be specified anymore for the second and all subsequent times when the variable is mentioned.



```
int    Period_MA;
double Price_Open;
color  Color_MA;
Period_MA  = 21;
Price_Open = 1.8245;
Color_MA   = Yellow;
```

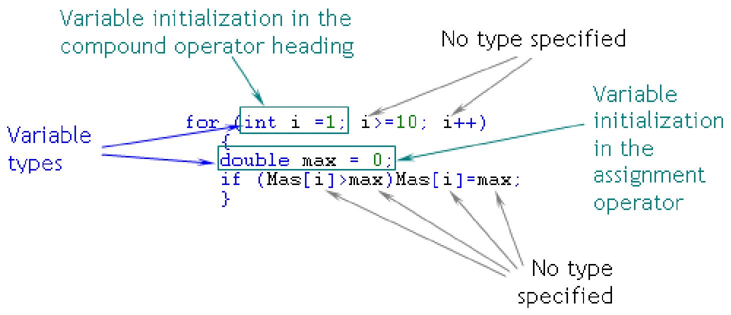
Fig. 13. Example of variable declaration in a single line.

It is allowed to declare and initialize variables in operators.



```
int    Period_MA  = 21;
double Price_Open = 1.8245;
color  Color_MA   = Yellow;
```

Fig. 14. Example of variable initialization.



```
for (int i =1; i>=10; i++)
{
double max = 0;
if (Mas[i]>max)Mas[i]=max;
}
```

Variable initialization in the compound operator heading
No type specified
Variable types
Variable initialization in the assignment operator
No type specified

Fig. 15. Variable initialization in the header of a compound operator.

Operations and Expressions

In order to understand what importance operations and expressions have in MQL4, no special analogies are needed. Practically, it is the same as operations and expressions in simple arithmetic. Everybody understands that in the record of **f = n + m**, members **f**, **n** and **m** are variables, signs **=** and **+** are operational signs, whereas **n + m** is an expression.

In the preceding section of the book, we learned about the necessity to present data of different types. Here we will get into possible relationships between these data (square meters still cannot be added to apples). In MQL4, there are some natural limitations and the rules of using operations in expressions.

The Notions of 'Operand', 'Operation', 'Operation Symbol' and 'Expression'

Operand is a constant, a variable, an array component or a value returned by a function (the term of function is considered in the section of Functions, that of array - in the section of Arrays; at this present stage of learning, it is sufficient to understand operands as constants and variables we have already studied before).

Operation is an action made upon operands.

Operation symbol is a preset character or a group of characters that order to execute an operation.

Expression is a sequence of operands and operation symbols; it is a program record, the calculated value of which is characterized by a data type.

Types of Operations

There are the following types of operations in MQL4:

- arithmetical operations;
- assignment operations;
- relational operations;
- Boolean (logical) operations;
- bitwise operations;
- comma operation;
- function call.

Operations are used in operators (see Operators). Only in operators their use makes sense and is realized in a program. The possibility to use an operation is determined by the properties of operators (if the operator's properties allow you to utilize this specific operation, you can use it; otherwise, you shouldn't utilize this operation). It is prohibited to use operations outside operators.

Arithmetical Operations

The following symbols belong to arithmetical operations symbols:

Symbol	Operation	Example	Analog
+	Addition of values	x + 2	
-	Subtraction of values or sign change	x - 3, y = - y	
*	Multiplication of values	3 * x	
/	Quotient of division	x / 5	
%	Residue of division	minutes = time % 60	
++	Addition of 1 to the value of the variable	y++	y = y + 1
--	Subtraction of 1 from the value of the variable	y--	y = y - 1

Assignment Operations

The following symbols belong to assignment operations symbols:

Symbol	Operation	Example	Analog
=	Assignment of the value x to the variable y	y = x	
+=	Increase of the variable y by x	y += x	y = y + x
-=	Reduction of the variable y by x	y -= x	y = y - x
*=	Multiplication of the variable y by x	y *= x	y = y * x
/=	Division of the variable y by x	y /= x	y = y / x
%=	Residue of division of the variable y by x	y %= x	y = y % x

Relational Operations

The following symbols belong to relational operations symbols:

Symbol	Operation	Example
==	True, if x is equal to y	x == y
!=	True, if x is not equal to y	x != y
<	True, if x is less than y	x < y
>	True, if x is more than y	x > y
<=	True, if x is equal to or less than y	x <= y
>=	True, if x is equal to or more than y	x >= y

Boolean (Logical) Operations

The following symbols belong to Boolean operations symbols:

Symbol	Operation	Example	Explanations
!	NOT (logical negation)	! x	TRUE(1), if the value of the operand is FALSE(0); FALSE(0), if the value of the operand is not FALSE(0).
||	OR (logical disjunction)	x < 5 || x > 7	TRUE(1), if any value of the values is true
&&	AND (logical conjunction)	x == 3 && y < 5	TRUE(1), if all values are true

Bitwise Operations

Bitwise operations can only be performed with integers. The following operations belong to bitwise operations:

One's complement of the value of the variable. The value of the expression contains 1 in all places, in which the values of the variable contain 0, and it contains 0 in all places, in which the values of the variable contain 1.

```
b = ~n;
```

The binary representation of x is shifted by y places to the right. This right shift is logical, it means that all places emptied to the left will be filled with zeros.

```
x = x >> y;
```

The binary representation of x is shifted by y places to the left; the emptied places to the left will be filled with zeros.

```
x = x << y;
```

The bitwise operation AND of the binary representations of x and y. The value of the expression contains 1 (TRUE) in all places, in which both x and y contain non-zero; and the value of the expression contains 0 (FALSE) in all other places.

```
b = ((x & y) != 0);
```

The bitwise operation OR of the binary representations of x and y. The value of the expression contains 1 in all places, in which x or y does not contain 0. It contains 0 in all other places.

```
b = x | y;
```

The bitwise operation EXCLUSIVE OR of the binary representations of x and y. The value of the expression contains 1 in the places, in which x and y have different binary values. It contains 0 in all other places.

```
b = x ^ y;
```

Comma Operation

Expressions separated by commas are calculated left to right. All side effects of calculations in the left expression can only occur before the right expression is calculated. The type and value of the result coincide with the type and value of the right expression.

```
for(i=0,j=99; i<100; i++,j--) Print(array[i][j]); // Loop statement
```

The transferred parameter list (see below) can be considered as an example.

```
My_function (Alf, Bet, Gam, Del) // Calling for a function with arguments
```

Operators and functions are considered in the sections of Operators, Functions and in the chapter named Operators)

Function Call

Function call is described in all details in the section of Function Call.

Operations on Similar Operands

If an elementary school pupil were told that, when solving the problem about the number of pencils, he or she would have to base his or her presentation on such terms as operands, operators and expressions, the poor schoolchild would surely find it impossible. Looking at the symbols of operations, one may think that coding is a mysterious and very complicated process, accessible only for for a kind of elite. However, coding isn't really difficult at all, you just need to make head or tail of some intensions. To be sure that this is really so, let's consider some examples.

Problem 1. John has 2 pencils, Pete has 3 pencils. How many pencils do these boys have?:)

Solution. Let us denote the number of John's pencils as variable A and the number of Pete's pencils as variable B, whereas the result will be denoted as C.

The answer will be: C = A + B

In the section named Data Types, we considered the methods of variable declaration. Pencils are things, i.e., it is something that can basically exist as a part (for example, there can be a half of a pencil). Thus, we will consider pencils as real variables, i.e., the variables of *double* type.

So we can code the solution, for example, as follows:

```
double A = 2.0;                      // The number of John's pencils
double B = 3.0;                      // The number of Pete's pencils
double C = A + B;                    // Total number
```

In this case, operation "+" applied to adding together the values of the one-type variables is quite illustrative.

The value type of the expression:

```
A + B
```

will be the type of the variables that are components of the expression. In our case, this will be the *double* type.

We will get the similar answer for the difference between the values (How many more pencils does Pete have than John?):

```
double A = 2.0;                      // The number of John's pencils
double B = 3.0;                      // The number of Pete's pencils
double C = B - A;                    // The difference between two real numbers
```

Other arithmetical operations are used in a similar manner:

```
double C = B * A;                    // Multiplication of two real numbers
double C = B / A;                    // Division of two real numbers
```

Similar calculations can be performed with integers, too.

Problem 2. Pupils go to the blackboard and answer in class. John went 2 times, Pete went 3 times. How many times did boys go to the blackboard in total?

Solution. Let us denote John's passages as variable X, Pete's passages as variable Y, the result - as Z.

In this example, we have to use the variables of *int* type, since we consider events that are integer by their nature (you cannot go to the blackboard 0.5 times or 1.5 times; the answer at the blackboard can be good or poor, but we're only interested in the number of such answers or passages).

The solution of this problem can be written as:

```
int X = 2;                          // Number of John's passages
int Y = 3;                          // Number of Pete's passages
int Z = X + Y;                      // Total number
```

In case of calculation of the difference between, product of or quotient of integers, the operation "-" is used in the similar simple manner:

```
int X = 2;                          // Integer
int Y = 3;                          // Integer
int Z = Y - X;                      // Difference between two integers
int Z = Y * X;                      // Product of two integers
int Z = Y / X;                      // Quotient of two integers
```

The situation is a bit different with the variables of *string* type:

Problem 3. At one corner of the house, there is a grocery store named "Arctic". At another corner of the same house, there is an establishment named "Hairdressing Saloon". What is written on the house?

Solution. In MQL4, you are allowed to add together the values of string constants and variables. If we add together variables of *string* type, strings are simply added one by one, in the sequence they are mentioned in the expression.

It is easy to code a program that would give us the necessary answer:

```
   string W1  = "Arctic";                        // String 1
   string W2  = "Hairdressing Saloon";           // String 2
   string Ans = W1 + W2;                         // Sum of strings
```

The value of variable Ans will be the string that appears as follows:

```
ArcticHairdressing Saloon
```

We obtained a not-much-to-look-at, but absolutely correctly formed value of *string* type. Of course, we should consider gaps and other punctuation in our practical coding of such problems.

Any other arithmetical operations with variables of string type are prohibited:

```
   string Ans= W1 - W2;                        // Not allowed
   string Ans= W1 * W2;                        // Not allowed
   string Ans= W1 / W2;                        // Not allowed
```

Typecasting

Typecasting is modifying (mapping) of types of the values of an operand or an expression. Before execution of operations (all but assignment operations), they are modified to a type of the highest priority, whereas before execution of assignment operations they are modified to the target type.

Let's consider some problems that deal with typecasting.

Problem 4. John has 2 pencils, whereas Pete went 3 times to the blackboard. How many in total?

As far as the formal logic is concerned, the ill-posedness of the problem is obvious. It stands to reason that events cannot be added together with things, it is wrong.

Problem 5. At one corner of the house, there is a grocery store named "Arctic", whereas John has 2 pencils.:)

With the same degree of hopelessness (as far as normal reasoning is concerned) we can ask either:

1. How many in total?, or

2. What is written on the house?

If you want to solve both problems above correctly in MQL4, you should refer to typecasting rules. First, we have to talk about how variables of different types are represented in the computer memory.

Data types, such as *int, bool, color, datetime* and *double*, belong to the *numeric* data type. The internal representation of constants and variables of *int, double, bool, color* and *datetime* type is a number. The variables of *int, bool, color* and *datetime* types are represented in the computer memory as integers, whereas the variables of *double* type are represented as double-precision numbers with a floating point, i.e., real numbers. The value of constants and variables of *string* type is a set of characters (Fig. 16).

The values of *int, bool, color* and *datetime* types are represented in the computer memory as integers. The values of *double* type are represented in the computer memory as real numbers. The values of *string* type are represented in the computer memory as a sequence of characters. The values of *int, bool, color, datetime* and *double* types are the values of *numeric* type. The values of *string* type are the values of *character* type.

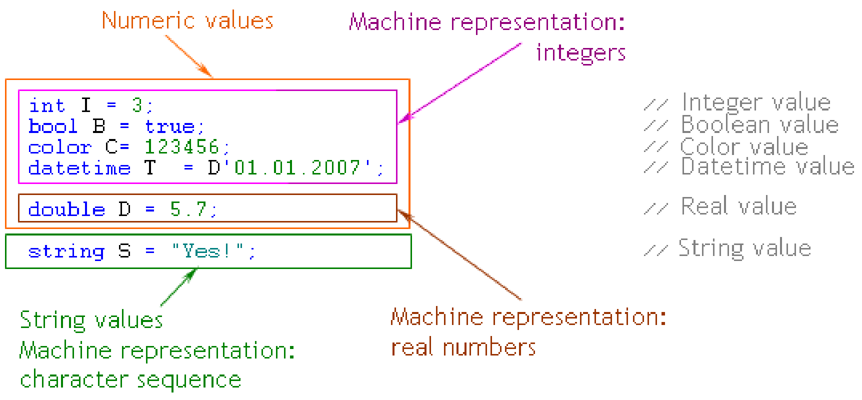


Fig. 16. Representation of different data types in the computer memory.

We mentioned above that the values of the variables of *int, bool, color* and *datetime* types are represented in the computer memory as integers, whereas those of *double* type - as real numbers. So, if we want to find out about the type of an expression that consists of variables of different types, we can only choose between three data types: *int, double* and *string*. The values of *bool, color* and *datetime* types will prove themselves in an expression in the same way as the values of *int* type.

So, what type will be the value of an expression composed of operands of different types? In MQL4, the rule of implied typecasting is accepted:

- if the expression contains operands of different types, the expression type will be transformed into the type having the highest priority; the types *int, bool, color* and *datetime* have equal priorities, whereas the *double* type has a higher priority, and the *string* type has the highest priority;
- if the type of the expression to the right of the assignment operation sign does not coincide with the type of the variable to the left of the assignment operation sign, the value of this expression is cast as the type of the variable to the left of the assignment operation sign; this is called 'target-type cast';
- it is prohibited to cast the values of *string* type to any other target type.

Well, let's return to Problem 4. There can be two solutions for it.

Solution 4.1. Calculation of the result of *int* type:

```
double A = 2.0;                      // The number of John's pencils
int    Y = 3;                        // The number of Pete's passages
int    F = A + Y;                    // Total number
```

First of all, we need to know the value of the expression provided its operands are of different types. In the expression:

```
A + Y,
```

operands of two data types are used: A - *double* type, Y - *int* type.

In compliance with the rule of implied typecasting, the value of this expression will be a number of *double* type. Please note: We are talking only about the type of expression A+Y, but not about the type of variable F that is to the left of the assignment operation sign. The value of this expression is real number 5.0. To cast the type of expression A+Y, we applied the first part of the implied typecasting rule.

After calculation of the expression A+Y, the assignment operation is executed. In this case, the types mismatch, too: the type of expression A+Y is *double*, whereas the type of variable F is *int*. During execution of the assignment operation: First, the type of expression A+Y is casted as *int* (according to the rule of integer calculations) and becomes integer 5; then this result becomes the value of the integer variable F. The calculations have been performed according to the second part of the implied typecasting rule - 'target-type cast'. The final result of calculations and manipulations is as follows: The value of the integer variable F is integer 5.

Solution 4.2. A similar situation occurs, if we try to have a result as a value of *double* type:

```
double A = 2.0;                   // The number of John's pencils
int    Y = 3;                     // The number of Pete's passages
double F = A + Y;                 // Total number
```

This situation differs from the preceding one by that the target type of variable F (to the left of the assignment operation sign), in our case, it is *double* type, coincides with the type (*double*) of the expression A+Y, so we don't have any target-type cast here. The result of calculations (the value of the *double*-type variable F) is real number 5.0.

Let's now find a solution for Problem 5. No questions come up at initialization of variables:

```
string W1 = "Arctic";               // String 1
double A  = 2;                      // The number of John's pencils
```

Solution 5.1. A possible solution for this problem:

```
      string W1   = "Arctic";                // String 1
      double A    = 2;                       // Number of John's pencils
      string Sum = W1 + A;                   // Implied transformation to the right
```

Here, in the right part, we add together the values of two variables: the one of *string* type, and the other one - of *double* type. According to the rule of implied typecasting, the value of variable A will first be cast to the *string* type (since this type is of a higher priority), and then the one-type values will be added together (concatenation). The type of the resulting value to the right of the assignment operation sign will be *string*. At the next stage, this value will be assigned to the *string* variable Sum. As a result, the value of variable Sum will be the following string:

```
Arctic2.00000000
```

Solution 5.2. This solution is wrong:

```
      string W1 = "Arctic";                   // String 1
      double A = 2;                           // Number of John's pencils
      double Sum = W1 + A;                    // This is inadmissible
```

In this case, we break a prohibition of target-type cast for the values of *string* type. The type of the value of expression W1+A, like in the preceding solution, is *string*. When the assignment operation is executed, the target-type cast must be performed. However, according to the rule, the *string*-type target cast to the types of lower priority is prohibited. This is an error that will be detected by MetaEditor at creation of the program (at compilation).

Generally, the rules given in this section are clear and simple: If you calculate any values, you must cast all differing types to a type with the highest priority. Typecasting with lowered priority is allowed only for numeric values, whereas strings cannot be transformed into numbers.

Features of Integer Calculations

Integers are known to be the numbers without a fractional part. If you add them together or subtract them, you will get an intuitive result. For example, if:

```
      int X = 2;                             // First int variable
      int Y = 3;                             // Second int variable
```

and:

```
      int Z = X + Y;                         // Addition operation,
```

there is no problem to calculate the value of variable Z: 2 + 3 = 5

Similarly, if you execute a multiplication operation:

```
      int Z = X * Y;                         // Multiplication operation,
```

the result is highly predictable: 2 * 3 = 6

But what result do we get, if our program has to execute a division operation?

```
      int Z = X / Y;                         // Division operation
```

It's easy to write 2 / 3. However, it is not an integer. So, what will be the value of expression X/Y and variable Z?

The rule of integer calculations consists in that the fractional part is always discarded.

In the above example, the expression to the right of the equality sign contains only integers, i.e., no typecasting takes place, in this case. And this means that the type of expression X/Y is *int*. So the result of finding the integer value of expression X/Y (= 2/3) is 0 (zero). This value (zero) will be assigned to variable Z at the end.

Correspondingly, other values of variables X and Y will produce other results. For example, if:

```
      int X = 7;                           // The value of an int variable
      int Y = 3;                           // The value of an int variable
      int Z = X / Y;                       // Division operation,
```

then the value of 7 / 3 for the expression X / Y and variable Z is equal to 2 (two).

Order of Operations

The calculating rule consists in the following:

The value of an expression is calculated according to the priorities of arithmetical operations and left to right, each intermediate outcome being calculated according to the typecasting rules.

Let's consider the calculating order for an expression in the following example:

```
    Y = 2.0*( 3*X/Z - N) + D;                           // Exemplary expression
```

The expression to the right of the equality sign consists of two summands: 2.0*(3*X/Z - N) and D. The summand 2.0*(3*X/Z - N) consists of two factors, namely: 2 and (3*X/Z - N). The expression in parentheses, 3*X/Z - N, in its turn, consists of two summands, the summand 3*X/Z consisting of three factors, namely: 3, X and Z.

In order to calculate the expression to the right from the equality sign, we will, first, calculate the value of expression 3*X/Z. This expression contains two operations (multiplication and division) of the same rank, so we will calculate this expression left to right. First, we will calculate the value of expression 3*X, the type of this expression being the same as the type of the variable X. Then we will calculate the value of expression 3*X/Z, its type being calculated according to the typecasting rule. After that, the program will calculate the value and the type of the expression 3*X/Z - N, then - of the expression 2.0*(3*X/Z - N), and the last - of the entire expression 2.0*(3*X/Z - N) + D.

As is easy to see, the order of operations in a program is similar to that in mathematics. However, the former one differs in calculating of the types of values in intermediate outcomes, which exercises a significant influence on the final result of calculations. Particularly (unlike the rules accepted in mathematics), the order of operands in an expression is very important. To demonstrate this, let's consider a small example.

Problem 6.Calculate the values of expressions A/B*C and A*C/B for integers A, B, and C.

The result of calculation is intuitively expected to be the same, in both cases. However, this statement is true only for real numbers. If we calculate the values of expressions composed of the operands of *int* type, we should always consider the intermediate outcome. In such a case, the sequence of operands is of fundamental importance:

```
    int A = 3;                         // Value of int type
    int B = 5;                         // Value of int type
    int C = 6;                         // Value of int type
    int Res_1 = A/B*C;                 // Result 0 (zero)
    int Res_2 = A*C/B;                 // Result 3 (three)
```

Let's follow the process of calculating the expression A/B*C:

1. First (from left to right) the value of the expression A/B will be calculated. According to the above rules, the value of the expression (3/5) is integer 0 (zero).

2. Calculation of the expression 0*C (zero multiplied by C). The result is integer 0 (zero).

3. General result (the value of the variable Res_1) is integer **0 (zero)**.

Let's now follow the developments of calculating the expression A*C/B.

1. Calculation of A*C. The value of this expression is integer 18 (3*6=18).

2. Calculation of the expression 18/B. The answer is obvious: after the fractional part has been discarded, (18/5) is integer 3 (three).

3. General result (the value of variable Res_2) is integer **3 (three)**.

In the above example, we consider just a small code fragment, in which the values of the variables of *int* type are calculated. If we replace these variables with constants with the same values, the final result will be the same. When calculating expressions containing integers, you must pay greater attention to the contents of your program lines. Otherwise, an error may occur in your code, which would be very difficult to find and fix later (especially in large programs). Such troubles do not occur in calculations using real numbers. However, if you use operands of different types in one complex expression, the final result may fully depend on a randomly formed fragment containing division of integers.

In the section of Operators, the term and general properties of operators are considered, whereas the special properties of each operator are described in the chapter named Operators.

Operators

The Term of Operator

One of the key notions of any programming language is the term of 'operator'. Coding seems to be impossible for the person that has not completely learned this term. The sooner and more properly a programmer learns what operators are and how they are applied in a program, the sooner this programmer starts writing his or her own programs.

Operator is a part of a program; it is a phrase in an algorithmic language that prescribes a certain method of data conversion.

Any program contains operators. The closest analogy to 'operator' is a sentence. Just as normal sentences compose the text of a story or of a novel, so operators compose a program.

Properties of Operators

There are two kinds of properties of operators - a common property and specific properties of different operators.

Common Property of Operators

All operators have one common property - they all are executed.

We can say that operator is an instruction containing the guide to operations (the description of an order). For a computer, executing a program running on it means (consecutively passing from one operator to another) complying the orders (prescriptions, instructions) contained in the operators.

Operator as such is just a record, a certain sequence of characters. Operator does not have any levers, wires or memory cells. This is why, when a computer is executing a program, nothing happens in operators as such, they continue staying in the program as composed by the programmer. However, the computer that has all those memory cells and links between them experiences all the transformations inside. If your PC has executed some transformations of data according to the instruction contained in an operator, you say: the operator has been executed.

Specific Properties of Operators

There are several kinds of operators. Operators of each type have their specific properties. For example, the property of an assignment operator is its ability to assign a certain value to the given variable; the property of a loop operator is its multiple executions, etc. Specific properties of operators are considered in all details in the corresponding sections of chapter Operators in this book. We will only say here that all types of operators have their own properties that are typical only for their type and are not repeated in any other types.

Types of Operators

There are two types of operators: simple operators and compounds**.**

Simple Operators

Simple operators in MQL4 end with the character ";" (semicolon). Using this separator, the PC can detect where one operator ends and another one starts. Character ";" (semicolon) is as necessary in a program as character "." (full stop) is necessary in a normal text to separate sentences. One operator may take several lines. Several operators may be placed in one line.

Every simple operator ends with character ";" (semicolon).

Examples of simple operators:

```
   Day_Next= TimeDayOfWeek(Mas_Big[n][0]+60);   // Simple operator

   Go_My_Function_ind();                        // Simple operator

   a=3;  b=a*x+n;  i++;                         // Several operators in a line

   Print("   Day= ",TimeDay(Mas_Big[s][0]),     // One operator..
         "   Hour=",TimeHour(Mas_Big[s][0]),    // is located..
         "   Minute=",TimeMinute(Mas_Big[s][0]),// in several ..
         "   Mas_Big[s][0]= ",Mas_Big[s][0],    // lines
         "   Mas_Big[s][1]= ",Mas_Big[s][1]);
```

Compound Operators (Compounds)

A compound operator consists of several simple ones separated by character ";" and is enclosed in braces. In order to be able to use several operators where only one is expected to be located, programmers use a compound operator (they also call it "block" or "block of code"). The list of operators in a compound is separated by braces. The presence of a closing brace marks the end of a compound operator.

An exemplary use of a compound in a conditional operator. It starts with the conditional operator if(expression) followed by a compound. The compound operator contains a list of executable operators.

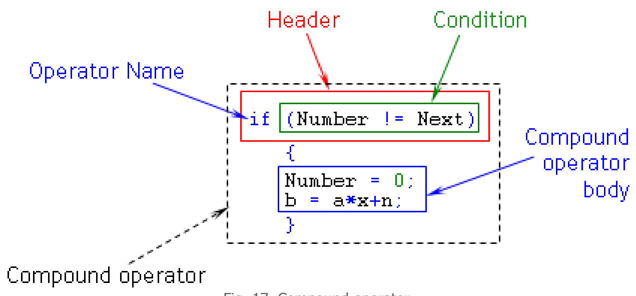


Fig. 17. Compound operator.

The body of a compound operator is enclosed in braces. Every compound operator ends with a closing brace.

Examples of compound operators:

```
                                           // Example of operator switch
switch(ii)                                 // Operator switch(expression)
   {                                       // Opening brace
   case 1: Buf_1[Pok-f+i]= Prognoz;  break; // Nested operators (operator body)
   case 2: Buf_2[Pok-f+i]= Prognoz;  break; // Nested operators (operator body)
   case 3: Buf_3[Pok-f+i]= Prognoz;  break; // Nested operators (operator body)
   }                                       // Closing brace,..
                                           // .. shows where the compound operator ends
//-----------------------------------------------------------------------------------------
                                           // Exemplary use of loop
for (tt=1; tt<=Kol_Point[7]; tt++)         // Operator for(expressions)
   {                                       // Opening brace
   Numb = Numb + Y_raz[tt]*X_raz[ii][tt];  // Nested operator (operator body)
   }                                       // Closing brace..
                                           // .. shows where the compound operator ends
//-----------------------------------------------------------------------------------------
                                           // Example of conditional operator if
if (TimeDay(Mas_Big[f][0])!= 6)            // if (expression)
   {                                       // Opening brace
   Sred =(Nabor_Koef[ii][vv][2]+ NBh)*Point;// Nested operators (operator body)
   Ind = Nabor_Koef[ii][vv][0] + f;        // Nested operators (operator body)
   Print(" Ind= ",Ind);                    // Nested operators (operator body)
   }                                       // Closing brace..
                                           // .. shows where the compound operator ends
```

The body of a compound is always enclosed in braces and can consist of zero, one, or several operators.

Examples of simple operators:

```
//----------------------------------------------------------------------------------
                                   // Example of operator for
   for (n=1; n<=Numb; n++)         // for(expressions)
      Mas[n]= Const_1+ n*Pi;       // Nested operator (operator body)
//----------------------------------------------------------------------------------
                                   // Example of conditional operator if
   if (Table > Chair)              // if (expression)
      Norma = true;                // first operator (suboperator 1)
   else                            // Else-condition
      Norma = false;               // second operator (suboperator 2)
//----------------------------------------------------------------------------------
```

Several simple operators may be combined in a compound operator without any strict necessity.

This is a rare, but absolutely admissible construction. In this case, the operators enclosed in braces are called "a block of operators". This use is fully acceptable. This is the programmer who decides whether to enclose operators in braces or not, just for the sake of convenient code representation. An exemplary block of operators:

```
   {                                       // Opening brace
   Day_Next= TimeDayOfWeek(Mas_Big[n][0]+60); // Simple operator
   b=a*x+n;                                // Simple operator
   }                                       // Closing brace..
```

Requirements for Operators

Operators must be written in the text of a program according to the format rules (how they must be represented in a code). No operator may be composed beyond these rules. If the program contains an operator composed against the format rules, MetaEditor will produce an error message at compilation. This means that the program containing the wrong operator cannot be used.

You should understand the phrase "operator format" as a set of format rules typical for the given type of operator. Each type of operator has its own format. Operator formats are in all details considered in the corresponding sections in chapter Operators of this book.

Order of Operators Execution

A very important characteristic of any program is the order of operators execution in it. Operators cannot be executed for no reason or by exception. In MQL4, the following order of operators execution is accepted:

Operators are executed in the order, in which they occur in the program. The direction of operators execution is accepted as left to right and downwards.

This means that both simple and compound operators are executed just one by one (like the lines in poems: first we read the top line, then the next lower, then the next, and so on). If there are several operators in one line, they should be executed consecutively, one by one, left to right, then operators are executed in the nearest lower line in the same order.

Operators composing a compound operator are executed in the same way: any operator in the block of code starts being executed only after the previous one has been executed.

Writing and Execution of Operators: Examples

The text of a program containing operators is not dissimilar in aspect to a normal text or a mathematical notation. However, this similarity is only formal. A normal text allows the notes to be placed in any sequence, whereas you must keep a well-defined order in a program.

As an example, let's see how an assignment operator works. We'll solve a simple linear equation system and compare the representation of some mathematical calculations in a normal text and in a program code, in operators.

Problem 7. We have an equation system:
Y = 5
Y - X = 2
The numeric value of variable X is to be found.

Solution 1. A normal text on a sheet of paper:

1. 5 - X = 2

2. X = 5 - 2

3. X = 3

Solution 2. A text in a program:

```
Y = 5;                          // Line 1
X = Y - 2;                      // Line 2
```

Both in the first and in the second solutions, the notes (lines) have a completed content. Nevertheless, the lines in Solution 1 cannot be used in a program as they are, because their appearance does not comply with the assignment operator format.

The notes given in Solution 1 represent some dependences in paper form. They can only be used to inform the programmer about the relations between the variables. Operators in a program are assigned for other purposes - they inform the machine what operations and in what order it must execute. All operators without any exceptions represent precise instructions that allow no ambiguities.

Both operators in Solution 2 are the assignment operators. Any assignment operator gives the machine the following order, literally:

Calculate the value of the expression to the right of the equality sign and assign the obtained value to the variable to the left of the equality sign.

For this reason, nothing else but variable can be located to the left of the equality sign in an assignment operator. For example, a record of 5 - X = 2 used in the first solution contains an error, because the set of characters 5 - X is not a variable. This is why there is no memory cell for placing the numeric value of the expression calculated to the right of the equality sign.

Let's follow the computer during execution of operators of the second solution.

1. Passing on the operator (line 1).

```
Y = 5;                          // Line 1
```

2. Referencing to the right part of the operator (the right part is located between the equality sign and the semicolon).

3. The computer has detected that the right part of the operator contains a numeric value.

4. Recording the numeric value (5) in the memory cell of variable Y.

5. Passing on the next operator (line 2).

```
  X = Y - 2;                    // Line 2
```

6. Referencing to the right part of the operator.

7. The computer has detected that the right part of the operator contains an expression.

8. Calculating the numeric value of the right part of the operator (5 - 2).

9. Recording the numeric value (3) in the memory cell of variable X.

Performing steps 1-4 by the computer is execution of the first operator (line 1). Performing steps 5-9 by the computer is execution of the second operator (line 2).

In order to code a workable program, the programmer must well realize what and in what order will be executed in this program. Particularly, not all mathematical calculations will be put in a program, it is sometimes necessary to pre-prepare operators.

For example, many intermediate calculations are made when solving mathematical problems. They can help a mathematician to find a proper solution, but turn out to be useless from the viewpoint of programming. Only meaningful solutions can be included into a program, for example: original values of variables or formulas to calculate the values of other variables. In the preceding example, the first operator bears information about the numeric value of variable Y, and the second operator provides the formula to calculate the value of the variable X we are interested in.

Any workable program contains expressions of a familiar sight, but you can also find such expressions that you will be able to understand only if you rate them as possible operators in your program. For example, the record below

```
  X = X + 1;                    // Example of a counter
```

seems to be wrong from the standpoint of mathematical logic and common sense. However, it is quite acceptable if you consider it as an operator (by the way, it is this operator that is widely used in coding).

In this operator, we calculate a new value of variable X: when executing the assignment operator (i.e., calculating the value of the right part of the operator), the computer refers to the memory cell containing the numeric value of variable X (for example, it turns out to be equal to 3 at the moment of referring to it), calculates the expression in the right part of the assignment operator (3 + 1), and writes the obtained value (4) in the memory cell provided for variable X. Execution of this assignment operator results in that the variable X gets the new value (4). The machine will store this value of variable X until the variable X occurs in the left part of the equality sign in another assignment operator. In this case, the new value of this variable will be calculated and stored up to the next possible change.

Functions

The Term of Function

The most important technological breakthrough in computer engineering is the possibility of creation and storage of separate code fragments that describe the rules of data processing for solving a problem or a small task. Such a possibility exists in MQL4, too.

Function is a named, specific part of a program that describes the method of data conversion.

Speaking about functions, we will consider two aspects: function description and function call.

Function description is a specific named part of a program intended for execution.

Function call (function reference) is a record, execution of which results in execution of a function.

In our everyday life, we can find many analogues of the function. Take, for example, the braking system of a car. The actuating mechanism that properly performs braking, as well as the idea implemented by the engineer, is the analogue of a function, whereas brake pedal is the analogue of function call. The driver presses the pedal down, and the actuating mechanisms perform some actions and - stop the car.

Similarly, if a function call occurs in a program, then the function of the same name will be called and executed, i.e., a certain sequence of calculations or other actions will be performed (for example, displaying a message or opening an order, etc.). The general sense of a function is in taking a logically completed part of the code outside the basic text of the program, whereas only the call for this part of the code remains inside the basic text of the program. Such program construction has some incontestable advantages:

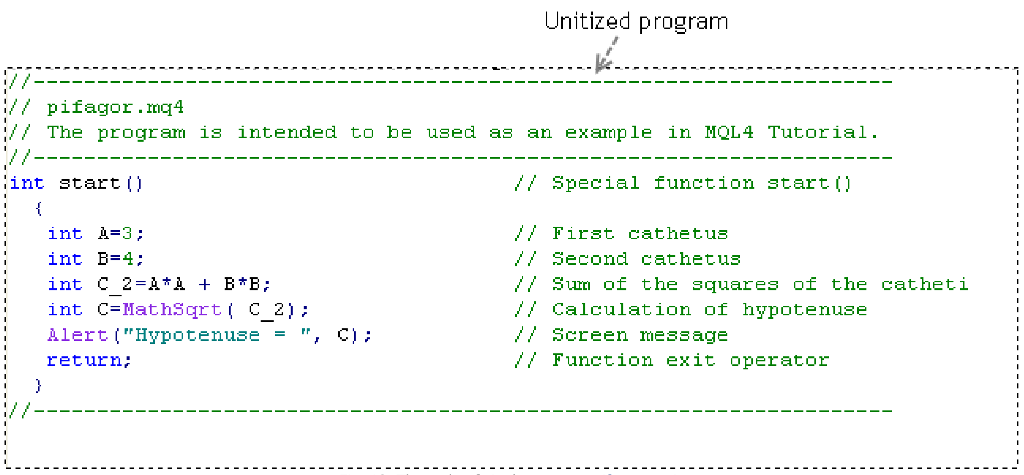
- first, the program text composed in such a manner is read much easier;
- second, one can easily see and, if necessary, modify the text of a function without making any changes in the basic code of the program;
- third, a function can be composed as a single file and use it in other programs, which will release the programmer from the necessity to insert the same fragments of the code in each newly created program.

We can say that the most part of the code in programs composed using MQL4 is written in form of functions. This approach became widespread and is a standard now.

Composition of a Function

Thus, a function can be described and called. Let's consider an example. Suppose we have a small program (Fig. 18) that finds the length of hypotenuse using two other sides of the right triangle and Pythagorean theorem.

In this program, all calculations are located together, the operators are executed one by one in the order, in which they occur in the program (from the top down).



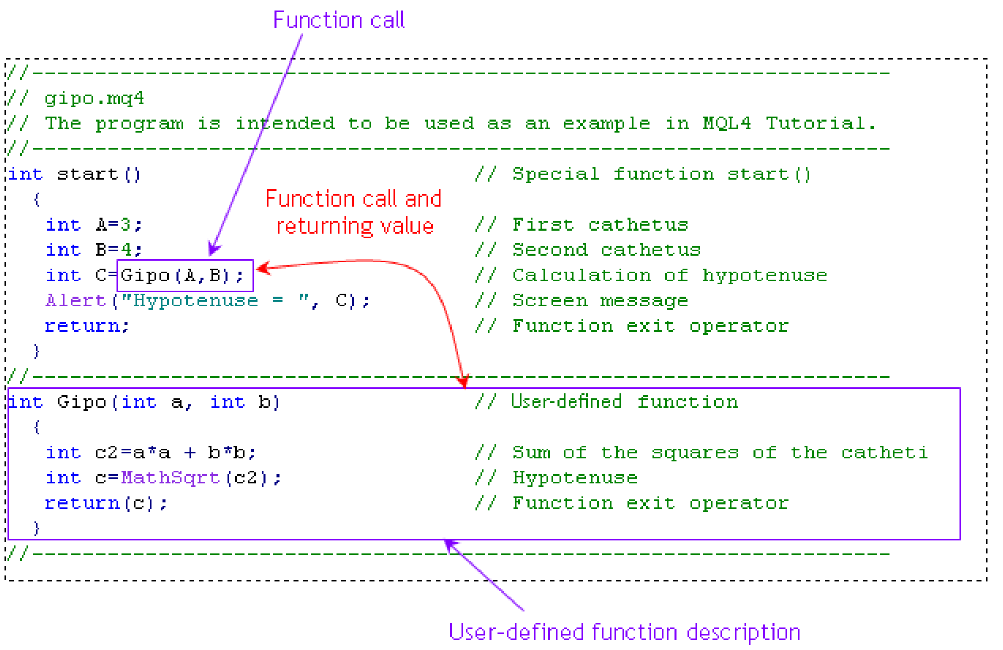
```
//--------------------------------------------------------------------
// pifagor.mq4
// The program is intended to be used as an example in MQL4 Tutorial.
//--------------------------------------------------------------------
int start()                           // Special function start()
  {
   int A=3;                           // First cathetus
   int B=4;                           // Second cathetus
   int C_2=A*A + B*B;                 // Sum of the squares of the catheti
   int C=MathSqrt( C_2);              // Calculation of hypotenuse
   Alert("Hypotenuse = ", C);         // Screen message
   return;                            // Function exit operator
  }
//--------------------------------------------------------------------
```

Fig. 18. The code of single program pifagor.mq4.

Problem 8. Compose a part of the given program code as a function.

It would be reasonable to make a function using the two lines that find the search value. The same program using a function is shown in Fig. 19.

In this program, a part of calculations is composed as a function. The basic code contains a user-defined function call. The description of the user-defined function is located outside (after) the basic code.


Function call
Function call and returning value
User-defined function description

```
//--------------------------------------------------------------------
// gipo.mq4
// The program is intended to be used as an example in MQL4 Tutorial.
//--------------------------------------------------------------------
int start()                             // Special function start()
  {
   int A=3;                             // First cathetus
   int B=4;                             // Second cathetus
   int C=Gipo(A,B);                     // Calculation of hypotenuse
   Alert("Hypotenuse = ", C);           // Screen message
   return;                              // Function exit operator
  }
//--------------------------------------------------------------------
int Gipo(int a, int b)                  // User-defined function
  {
   int c2=a*a + b*b;                    // Sum of the squares of the catheti
   int c=MathSqrt(c2);                  // Hypotenuse
   return(c);                           // Function exit operator
  }
//--------------------------------------------------------------------
```

Fig. 19. The code of a program that contains the description of and the call for user-defined function gipo.mq4.

Both versions of the program will give the same result. However, the code is composed as a single module in the first version (Fig. 18), whereas in the second version (Fig. 19) a part of calculations is executed in a function called from the basic text. Upon completion of the calculations separated into the function, the calculations in the main code will be continued.

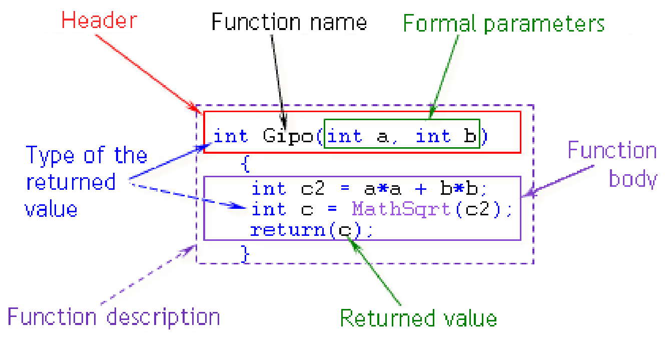
To have the function executed, we have to call it (refer to it). This is why the function is practically represented in two parts: the code itself composing the function (function description) and the function call used to launch the function (referring to the function). If you don't call the function, it will not be executed. At the same time, if you call a nonexistent function, this will result in nothing (MetaEditior will give an error message if you try to compile such a program in MQL4).

Function Description

The description of a function consists of two basic parts - function header and function body.

The **header** of a function consists of the type of the return value, function name and the list of formal parameters. The list of formal parameters are enclosed in parentheses and placed after the function name. The type of the return value can be one of the types we already know: *int, double, bool, color, datetime*, or *string*. If the function does not return any value, its type can be denoted as *void* ("without contents, empty") or as any other type.

The **body** of a function is enclosed in braces. The function body may contain simple and/or compound operators, as well as calls for other functions. The value returned by the function is given in the parentheses of operator return(). The type of the value returned using operator return(), must match with the function type specified in the function header. The description of the function is ended with a closing brace.



```
int Gipo(int a, int b)
  {
   int c2 = a*a + b*b;
   int c = MathSqrt(c2);
   return(c);
  }
```

Fig. 20. Function description.

The function description must be located separately in the program, outside any other functions (i.e., not inside of a function, but outside it).

Function Call

Function call represents the function name and the list of transferred parameters. The list of transferred parameters is enclosed in parentheses. The function call can be represented as a separate operator or as a part of an operator.

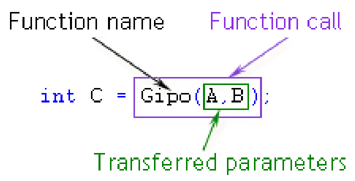


Fig. 21. Function call (reference to a function).

Any function call is always specified within another function (i.e., not outside all other functions, but inside one of them).

Function Types

There are three types of functions - special functions, standard (built-in, predefined) functions, and user-defined functions.

Special Functions

In MQL4, there are 3 special functions in total. They have predefined names: **init(), start(),** and **deinit()** that may not be used in the names of any other functions. The detailed consideration of special functions is given in chapter Special Functions. We only say here that the basic code of a program is located inside these functions (Fig. 18, 19).

The special feature of special functions is the fact that they are called for execution by the client terminal. Although special functions have all properties of functions in general, they are not usually called from the program if this program is coded properly.

Standard Functions

MQL4 has a number of useful functions that don't need being described when coding a program. For example, computing square roots, printing messages in the system journal or on the screen - all these and many other standard functions are executed according to the predefined algorithm. The user needn't to learn the contents of these functions. He or she can just be sure that all standard functions are developed by professionals properly and according to the best possible algorithm.

The special feature of standard functions is that they are not described in the text of the program. The standard functions are called in the program in the same manner as when using any other functions (it's a common practice).

In our example (Fig. 18, 19), two standard functions are used: MathSqrt() and Alert(). The former one is intended for computing of square roots, whereas the latter one is for messaging a certain text enclosed in parentheses to the screen. The properties of standard functions are considered in more details in section Standard Functions. Here we will note that these records represent standard function calls, whereas no descriptions of these functions can be found in the program. Standard functions can also be named built-in, or predefined functions. You may use any of these terms.

User-Defined Functions

In some cases, programmers create and use their own functions. We call these functions user-defined functions. User-defined functions are utilized in programs with both function descriptions and function calls.

Table 1. Function description and function call used in programs for functions of different types.

Function Type	Function Description	Function Call
Special	Applicable	Not applicable (*)
Standard	Not applicable	Applicable
User-defined	Applicable	Applicable

(*) Although special functions can be technically called from a program, it is not recommended to do so.

Properties of Functions

Function Execution

The main property of all functions is that the called functions are executed. Functions are executed according to their codes.

Passed Parameters and Return Value

As to information received and returned, a function is like a standard calculator. You can type (using its keyboard) a certain expression that consists of several values entered one by one, and you will obtain one value as an answer. A function can get and process one or several parameters from the program that has called this function for execution, and the function will finish its operation by returning (transmitting, giving) one parameter as an answer to this program.

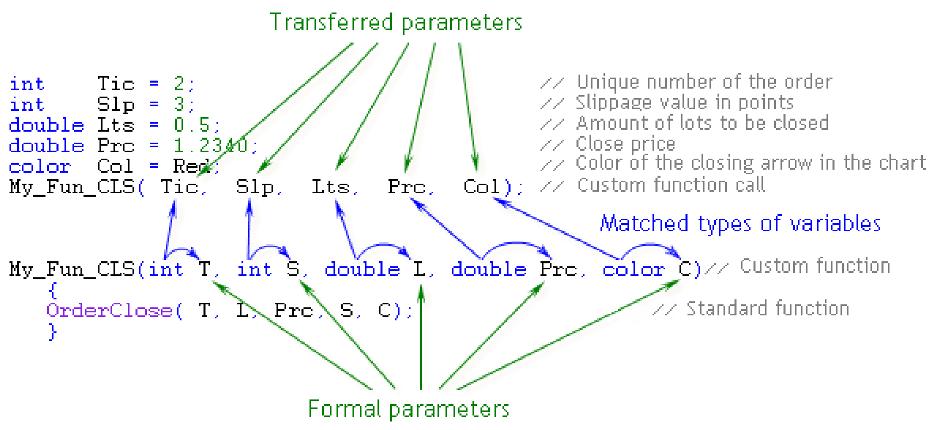
The **passed parameters** are specified by enclosing in parentheses after the name of the function to be called. The passed parameters are usually separated by commas. The number of parameters passed to the function must not exceed 64. The function may also omit using the passed parameters. In this case, an empty list of parameters is specified, i.e., you just put an opening parenthesis and a closing parenthesis directly after the function name.

The number, types and order of the passed parameters in the function call must match with those of formal parameters specified in the function description (the call for a function that has default parameters is an exemption - see Function Call and Function Description and the Operator "return"). If they don't match, MetaEditor will give you an error message. Constants, variable, expressions and arrays may be used as passed parameters.

The **return value** is specified in the parentheses of the operator return() (see Function Description and the Operator "return"). The type of the value returned using operator return() must match with the type of the function given in the function header. It is also possible that a function does not return any value. In this case, nothing is specified in the parentheses of the operator return().

In our example, the passed parameters are variables **A** and **B** (Fig. 21), whereas the return value is variable **c** (Fig. 20). The requirement of matching the types of passed and formal parameters can be seen in Fig. 22.

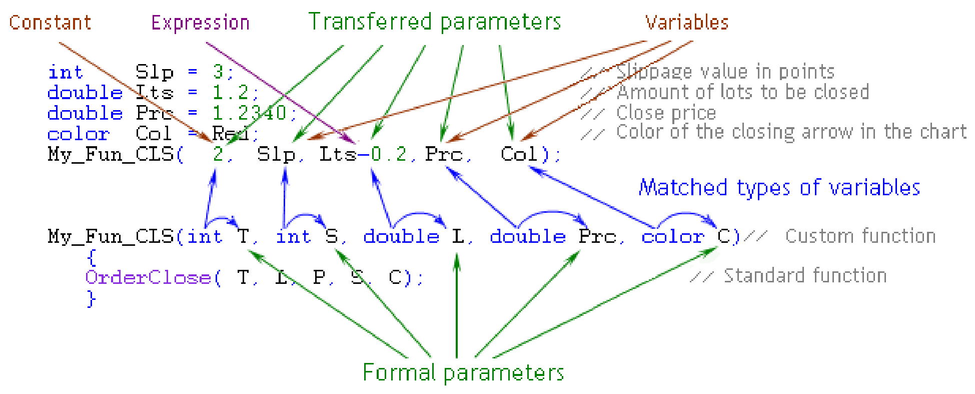
The number, types and order of the passed parameters in the function call must match with those of formal parameters specified in the function description. Only variables can be used in the header of the function description as formal parameters.



```
int    Tic = 2;                        // Unique number of the order
int    Slp = 3;                        // Slippage value in points
double Lts = 0.5;                      // Amount of lots to be closed
double Prc = 1.2340;                   // Close price
color  Col = Red;                      // Color of the closing arrow in the chart
My_Fun_CLS( Tic, Slp, Lts, Prc, Col);  // Custom function call

My_Fun_CLS(int T, int S, double L, double Prc, color C)// Custom function
   {
   OrderClose( T, L, Prc, S, C);              // Standard function
   }
```

Fig. 22. Match in the number, types and order of passed and formal parameters. Only variables are used as passed parameters.

Only variables, expressions and constants can be used as passed parameters.



```
int    Slp = 3;                        // Slippage value in points
double Lts = 1.2;                      // Amount of lots to be closed
double Prc = 1.2340;                   // Close price
color  Col = Red;                      // Color of the closing arrow in the chart
My_Fun_CLS( 2, Slp, Lts-0.2, Prc, Col);

My_Fun_CLS(int T, int S, double L, double Prc, color C)// Custom function
   {
   OrderClose( T, L, P, S, C);                // Standard function
   }
```

Fig. 23. Match in the number, types and order of passed and formal parameters. A constant, an expression, and variables of the corresponding type are used as passed parameters.

Formal Parameters

A high point of functions is the use of formal parameters.

Formal parameters are a list of variables specified in the header of the function description.

We mentioned before that one function could be used in several programs. However, different programs use different names for the variables. If functions required strict matching in the names of variables (and, correspondingly, in their value), it wouldn't be convenient for programmers. Indeed, you would then have to compose each newly developed program in the names of variables that had already been used in your functions before. However, fortunately, the variables used inside functions have no relations to the variables used in a program.

Let's refer to Fig. 20 and 21 once again. It should be noted that the names of the passed parameters (**A** and **B** are given in the parentheses of the function call) do not match with the names of the parameters (**a** and **b**) specified in the function description. We noticed in the section Constants and Variables that MQL4 is case-sensitive. Thus, the names **A** and **a**, **B** and **b** considered here are different names of variables. However, there is no error in this code.

The variables (formal parameters) used in the function description are not related to the variables used in the basic code of the program. They are just different variables. Only variables, but not constants, can be specified in the function header as formal parameters.

How It Works

- In the program, there occurs a function call, variables **A** and **B** being specified in its parentheses.
- The program calls the function of the same name, which has formal parameters **a** and **b** specified in its header.
- The value of variable **A** is assigned to variable **a**.
- The value of variable **B** is assigned to variable **b**.
- The executable function performs calculations using the values of variables **a** and **b**.

It is allowed to use any names of formal parameters (that don't coincide with the names of the variables used in the program). In this example, we used the identifiers of formal parameters **a** and **b**. However, we could use any others, for example, **m** and **n**, or **Kat_1** and **Kat_2**. Of course, when writing a program, you should specify in the function body the calculations that use the names of variables that are given in the header. Since we have given **a** and **b** in the header, so we have to use in the function **a** and **b**, not **m** and **n**.

In a function, the calculations are made that involve formal parameters **a** and **b**, but not variables **A** and **B**. In a function, any allowed actions may be perform on formal parameters **a** and **b** (including the changes in the values of these variables). This does not have any effect on variables **A** and **B**.

The return value calculated in the function is given in the parentheses of the operator return(). The value of a variable, the result of an expression, or a constant can be used as return value. In our case, this is the value of local variable **c** (local variable is a variable declared within a function; if you exit the function, the values of all local variables will be lost; we will consider more details about local variables in the section Types of Variables). The function returns into the program the value of local variable **c** (Fig. 19). This means that this value will now be assigned to variable **C**.

Further calculations in the program, if any, can be performed with the variables declared within the calling function. In our case, calling function is special function start() (it contains the line for calling the user-defined function), whereas the variables declared within the calling function are **A**, **B**, and **C**. Thus, in the function the calculations are performed using formal parameters, which allows us to create functions using arbitrary names of variables, irrespective of the names of values actually used in the program.

Example: Standard Functions in a Program

First of all, let's consider the behavior of the program shown in Fig. 18. We should note that the entire code of the program is located inside the special function start(). At this training phase, we won't pay special attention to this (special functions and their properties are considered in more details in the section Special Functions).

Let's follow the execution of the program starting with the assignment operator:

```
   int A = 3;                                // First cathetus
```

1. The right part of the assignment operator contains the specified constant, its value is 3.

2. The value of 3 (the value of the right part) is assigned to variable **A** (that locates to the left of the equality sign in the assignment operator).

The control is given to the next line:

```
   int B = 4;                                // Second cathetus
```

3. The right part of the assignment operator contains the specified constant, its value is 4.

4. The value of 4 is assigned to variable **B**.

The program goes to execution of the next line:

```
  int C_2 = A*A + B*B;                       // Sum of the squares of catheti
```

5. Calculation of the right part of the assignment operator. The result of the calculations is the value of 25 (the details of how a program refers to variables to take their values are considered in the section Constants and Variables, so we aren't particularizing this point here now).

6. Assignment of the value 25 to variable **C_2**.

The next line represents an assignment operator, the right part of which contains a call for the standard function:

```
   int C = MathSqrt( C_2);                   // Calculation of hypotenuse
```

The program seeks to execute the assignment operator. For this purpose, it executes calculations to the right of the equality sign first.

7. The program calls for execution the standard function MathSqrt() (that calculates square roots). The value of variable C_2 (in our case, this value is equal to 25) will be used as the passed parameter. Note that there is no description of this standard function anywhere in the program. The descriptions of standard functions must not be located in programs. In the text of a program, you can easily distinguish the standard function call by its appearance: they are highlighted in MetaEditor with dark blue (programmer may choose colors voluntarily).

8. The calculations are made in the standard function MathSqrt().

9. The standard function MathSqrt() has completed its calculations. It returns the obtained value. In our case, it is the value of **5** (the square root of 25).

The value returned by the function is now the content of the record:

```
            MathSqrt( C_2)
```

This record can be considered as a special, complex variable, a kind of a thing, inside which the calculations are performed. After these calculations have been completed, the thing takes a value. The determining is here the fact that the value returned by the function can be assigned to another variable or considered somehow in any other calculations.

10. In this case, our value is the value of the right part of the assignment operator. On continuing execution of the assignment operator, the program will assign the value of 5 to variable C.

11. The next line contains operator that references to the standard function Alert() (function call).

```
     Alert("Hypotenuse = ", C);                    // Message on the screen
```

The standard function Alert() opens a dialog box where the values of the passed parameters are displayed. In this case, the function has taken two values as the passed parameters:

- the value of string constant: Hypotenuse =

- the value of integer variable **C:** 5

It was noted above that not all functions must return the value (that is the result of the function execution). The standard function Alert() does not return any value, since it has another task - it must display the text on the screen, in a special window.

As a result of execution of the standard function Alert(), in the window of this function the following line will appear:

```
Hypotenuse = 5
```

12. The last operator in this program completes the work of the special function start().

```
   return;                                         // Function exit operator
```

The work of the program is finished at this point.

A question may arise: How can we know what function will return the value, and what not? The answer to this question is obvious: In order to find the detailed descriptions of built-in functions, you should read the reference documentation in MQL4.community, the website launched by MetaQuotes Software Corp., or the Help File of MetaEditor. The properties of a user-defined function are specified in its description. Whether a user-defined function returns the value or not, depends on its algorithm (the decision is made by the programmer at the stage of writing the program code of the function).

Example: User-Defined Function in a Program

Let's consider how the same calculations are made in the program containing a user-defined function (Fig. 19). A certain part of the code that could previously to be found in the special function start(), is unavailable now. It is replaced with the call for the user-defined function. However, the special function start() is followed by the description of the user-defined function.

The first two lines, in which integer variables **A** and **B** take numeric values, remain the same. Consequently, nothing changes in their execution:

```
     int A = 3;                                // First cathetus
     int B = 4;                                // Second cathetus
```

In the third line we have the assignment operator. Its right part contains the call for the user-defined function:

```
     int C = Gipo(A,B);                        // Calculation of hypotenuse
```

6. On executing this operator, the program will first of all call the user-defined function.

Note: The description of the user-defined function must be present in your program and placed immediately after the brace that closes the special function start() (i.e., outside the special function).

In referring to the user-defined function, the program will perform the following:

6.1. Calling for variable **A** in order to get its value (in our case, **3**)

6.2. Calling for variable **B** in order to get its value (in our case, **4**)

Note: As soon as the program starts to call the user-defined function (the function is user-defined, in our specific case; this rule applies to all functions), the program gets just a copy of the values of variables used as passed parameters, whereas the values of these variables themselves (in this case, **A** and **B**) are not supposed to change due to application of the user-defined function, nor they really change.

7. The control is passed to the user-defined function.

During the whole time of execution of the user-defined function (no matter how long it takes), the values of variables in the calling program will not get lost, but will be stored.

The first line in the description of the user-defined function is its header:

```
     int Gipo(int a, int b)                     // User-defined function
```

On executing the user-defined function, the program will do the following:

7.1. The value of **3** (the first value in the list of passed parameters) will be assigned to variable **a** (the first variable in the list of formal parameters).

7.2. The value of **4** (the second value in the list of passed parameters) will be assigned to variable **b** (the second variable in the list of formal parameters).

Then the control will be passed to the function body for it to execute its algorithm.

The first operator in the function body is:

```
     int c2 = a*a + b*b;                       // Sum of the squares of catheti
```

7.3. On executing this operator, the program calculates the value in the right part of the assignment operator, and after that it will assign the obtained value (in our case, 3*3 + 4*4 = **25**) to variable **c2**.

The next operator:

```
     int c = MathSqrt(c2);                     // Hypotenuse
```

7.4. Here we find the square root of the value of variable **c2**. The order of operations is the same as in the previous example. The description of standard function is not used either. The execution of the assignment operator will result in assigning the value of **5** to variable **c**.

7.5. In the next line, we have operator:

```
     return(c);                               // Function exit operator
```

On executing this operator, the program will return (to the call site of the user-defined function) the value enclosed in the parentheses of this operator. In our case, it is the value of variable **c**, i.e. **5**.

At this point, the execution of the user-defined function is over, the control is given to the call site.

8. Recall that the user-defined function was called from the operator

```
     int C = Gipo(A,B);                       // Hypotenuse calculation
```

The record (the user-defined function call)

```
            Gipo(A,B)
```

at the stage of returning the value, takes the value calculated in the function (in our case, it is the value of **5**).

On completing the execution of the assignment operator, the program will assign the value of 5 to the variable **C**.

9. The next operator,

```
     Alert("Hypotenuse = ", C);               // Message on the screen,
```

will be executed in the same way as in the previous example, namely: in a special window, there will a message appear:

```
Hypotenuse = 5
```

10. The last operator in this program,

```
   return;                                    // Function exit operator,
```

completes the work of the special function start(),and simultaneously completes the work of the entire program (the properties of special functions are considered in more details in the section Special Functions).

Let's consider some implementation versions of the above user-defined function. It is easy to verify that programming with user-defined functions has some incontestable advantages.

Previously Considered Implementation of User-Defined Function Gipo()

In this function, the formal parameters "resemble" the variables used in the basic program. However, this is just a formal similarity, because **A** and **a** are the different names of variables.

```
//--------------------------------------------------------------------------
int Gipo(int a, int b)                  // User-defined function
  {
   int c2 = a*a + b*b;                  // Sum of the squares of catheti
   int c = MathSqrt(c2);                // Hypotenuse
   return(c);                           // Function exit operator
  }
//--------------------------------------------------------------------------
```

Implementation of User-Defined Function Gipo(): Version 2

In this case, the names of formal parameters do not "resemble" the names of variables used in the basic program. However, this doesn't prevent us from using this function in the program.

```
//-----------------------------------------------------------------------
int Gipo(int alpha, int betta)           // User-defined function
  {
   int SQRT = alpha*alpha + betta*betta; // Sum of the squares of catheti
   int GIP = MathSqrt(SQRT);             // Hypotenuse
   return(GIP);                          // Function exit operator
  }
//-----------------------------------------------------------------------
```

Implementation of User-Defined Function Gipo(): Version 3

In this example, the variable alpha is reused in the program and changes its value twice. This fact has no effect on the actual variables specified in the function calls of the main program.

```
//-----------------------------------------------------------------------
int Gipo(int alpha, int betta)           // User-defined function
  {
   alpha= alpha*alpha + betta*betta;     // Sum of the squares of catheti
   alpha= MathSqrt(alpha);               // Hypotenuse
   return(alpha);                        // Function exit operator
  }
//-----------------------------------------------------------------------
```

Implementation of User-Defined Function Gipo(): Version 4

In this user-defined function, all calculations are collected in one operator. The return value is calculated directly in the parentheses of the operator return(). The subduplicate is calculated directly in the location where the passed parameter must be specified in the standard function MathSqrt().This solution may seem to be strange or wrong at the beginning. However, there is no error in this implementation of the user-defined function. The number of operators used in the function is smaller than in other implementations, so the code turns out to be more compact.

```
//-----------------------------------------------------------------------
int Gipo(int a, int b)                   // User-defined function
  {
   return(MathSqrt(a*a + b*b));          // Function exit operator
  }
//-----------------------------------------------------------------------
```

Thus, the application of user-defined functions has some incontestable advantages in the programming practice:

- the names of variables in the basic text of the program have no relations to the names of formal parameters in a user-defined function;
- user-defined functions can be reused in different programs, there is no need to change the code of a user-defined function;
- libraries can be created, if necessary.

These very useful properties of functions allow to create really large programs in a corporation, for example. Several programmers can be involved in this process simultaneously, and they a released from the necessity to agree about the names of variables they use. The only thing to be agreed is the order of variables in the header and in the function call.

Program Types

Starting to write a program in MQL4, the programmer must, first of all, answer the question about what type of programs will it be. The contents and functionality of the program fully depend on this. In MQL4, there are 3 types of application programs: Expert Advisors, scripts, and custom indicators. Any program developed by a programmer will belong to one of these types. They all have their purposes and special features. Let's consider these types in more details.

Expert Advisor (EA) is a program coded in MQL4 and called by the client terminal to be executed at every tick. The main purpose of Expert Advisors is the programmed control over trades. Expert Advisors are coded by users. There are no built-in EAs in the client terminal.

Script is a program coded in MQL4 and executed by the client terminal only once. Scripts are intended to perform any allowed operations that should be executed only once. Scripts are coded by users. They are not delivered with the client terminal as built-in programs.

Custom indicator is a program coded in MQL4 and called by the client terminal to be executed at every tick. It is basically intended for graphical displaying of preliminarily calculated dependences as lines. Indicators cannot trade. There are two types of indicators: technical (built-in) indicators and custom indicators. Indicators are considered in more details in the sections of Usage of Technical Indicators and Creation of Custom Indicators.

The programmer chooses the type of the program to be written depending on what is the purpose of this specific program and on properties and limitations of programs of different types.

Properties of Programs

Launching a Program for Execution

There is a criterion that distinguishes Expert Advisors and custom indicators from scripts. This is their run duration. In the section Some Basic Concepts, we mentioned already that programs were launched the amount of time that is multiple of the amount of ticks. This statement is true for EAs and custom indicators, but it is false for scripts.

Expert Advisor and custom indicator. Once you have attached a program (EA or custom indicator) to the symbol window, the program makes some preparations and switches to the tick-waiting mode. As soon as a new tick comes, the program will be launched by the client terminal for execution, then it makes all necessary operations prescribed by its algorithm, and, upon completion, passes the control to the client terminal, i.e., switches to the tick-waiting mode.

If a new tick comes when the program is being executed, this does not make any effect on the program execution - the program continues being executed according to its algorithm and passes the control to the client terminal only upon completion. This is why not all the ticks result in launching an EA or a custom indicator, but only those that income when the control is in the client terminal and when the program is in the tick-waiting mode.

The new tick launches the program for execution. Thus, an Expert Advisor or a custom indicator can operate within a long period of time, being attached to the symbol window and starting to run from time to time (as multiple of the amount of incoming ticks).

Besides, an Expert Advisor differs from an indicator by the execution order at the first launch of the program. This difference is determined by the specific properties of special functions in the program of a certain type (see Special Functions). Once attached to the symbol window, an Expert Advisor makes necessary preparations (function init()) and switches to the tick-waiting mode to start function start(). Unlike EAs, a custom indicator both executes function init() and calls function start() one time to make the first, preliminary calculation of the indicator value. Later on, at a new tick, the program is launched by calling only function start (), i.e., operators are executed according to the algorithm of function start().

Script. Unlike Expert Advisors or indicators, a script will be launched for execution immediately after it has been attached to a symbol window, without waiting for a new tick. The entire code of the script will be executed on time. After all program lines have been executed, the script finishes its operations and is unloaded from the symbol window. A script is helpful if you want to make one-time operations, for example, to open or close orders, to display texts on the screen, to install graphical objects, etc.

The differences in execution of Expert Advisors, scripts and custom indicators is determined by the properties of their special functions the will be considered in more details in the section Special Functions.

Trading

One of the main criteria that mark the above programs is the possibility to make trading instructions. A trading instruction is a control that a program passes to the trading server in order to open, close, or modify orders. Trading instructions are formed in the programs using built-in functions that we call "trading functions".

Only Expert Advisors and scripts have the right to use trading functions (only if the corresponding option is enabled in the EA/script settings). It is prohibited to use trading functions in custom indicators.

Simultaneous Use

The programs also differ from each other by the amount of programs of different types simultaneously attached to a symbol window.

Expert Advisor. You can attach only one EA in one symbol window; the simultaneous use of several Expert Advisors is prohibited.

Script. You can attach only one script in one symbol window; the simultaneous use of several scripts is prohibited.

Custom indicator. You can attach several indicators in one symbol window simultaneously, they will not interfere with each other.

The programs of all types can be launched simultaneously in one symbol window provided their compliance with the limitations of each type. For example, you can launch one EA, one script and several indicator in one symbol window at the same time, or one EA and one indicator. However, you may not launch several EAs or scripts in one symbol window, whatever programs of other types are launched simultaneously.

At the same time, you may simultaneously launch the programs of the same type in different windows of one symbol. For example, if you want to launch two Expert Advisors for one symbol, you can launch one EA in one window of this symbol and another one in another window of the same symbol. In this case, your Expert Advisors will work simultaneously. However, you must take into consideration that EAs launched in this manner may form contradictory trading instructions. For example, one of them can give instructions to open orders, whereas the other one can instruct to close orders. This may cause occurring of a long sequence of useless trades that results in the total loss.

The programs of all types can create global variables available for all other programs launched in the client terminal, including those launched in the windows of different symbols. This allows the machine to coordinate the simultaneous operations of all programs. The order of using global variables will be specially considered in the section GlobalVariables.

Calling Programs for Execution

Programs of all types can only be executed at the user's will. In MQL4, you cannot call an Expert Advisor, a script, or an indicator for execution programmatically.

The only exemption is the built-in function iCustom() that allows you to refer to a custom indicator for some data, and to the functions of technical indicators. The referring to function iCustom() or to the functions of technical indicators does not result in displaying the lines of indicators in the symbol window (see Simple Programs in MQL4).

Table 2. Main properties of Expert Advisors, scripts, and custom indicators.

Property of the Program	Expert Advisor	Script	Indicator

Run duration	Over a long period	One time	Over a long period
Trading	Allowed	Allowed	Prohibited
Displaying of lines	No	No	Yes
Simultaneous use of several programs of the same type	Prohibited	Prohibited	Prohibited
Calling for execution programmatically	Prohibited	Prohibited	Prohibited

Thus, if we want a program that would manage trading according to a certain algorithm, we should write an EA or a script. However, if we want to have a certain dependence graphically displayed, we should use an indicator.

MetaEditor

In this section we will dwell on the general order of creating application programs using MetaEditor.

MetaEditor is a multifunction specialized editor intended for creating, editing and compiling application programs written in MQL4. The editor has a user-friendly interface that allows a user to easily navigate when writing and checking out a program.

- File System
  MetaEditor stores all source codes of MQL4 programs in its own structured catalog on a hard disk. The location of a program in MQL4 is determined by its purpose: a script, an Expert Advisor, an indicator, an include-file or a library.

- Creating and Using Programs
  It is very easy to create a program in MQL4 - built-in tools will help you. You can modify templates for creating scripts, indicators or Expert Advisors. The created code will be automatically saved in a corresponding folder of MetaEditor file system.

File System

The client terminal recognizes program types by their location in subordinate directories.

All application programs are concentrated in the directory ***ClientTerminal_folder |experts***. Expert Advisors, scripts and custom indicators that a trader is going to use in his practical work should be located in corresponding directories (see Fig. 24). Expert Advisors are located right in the directory ***ClientTerminal_folder |experts***, scripts and indicators - in subdirectories ***ClientTerminal_folder|experts|scripts*** and ***ClientTerminal_folder |experts|indicators.***

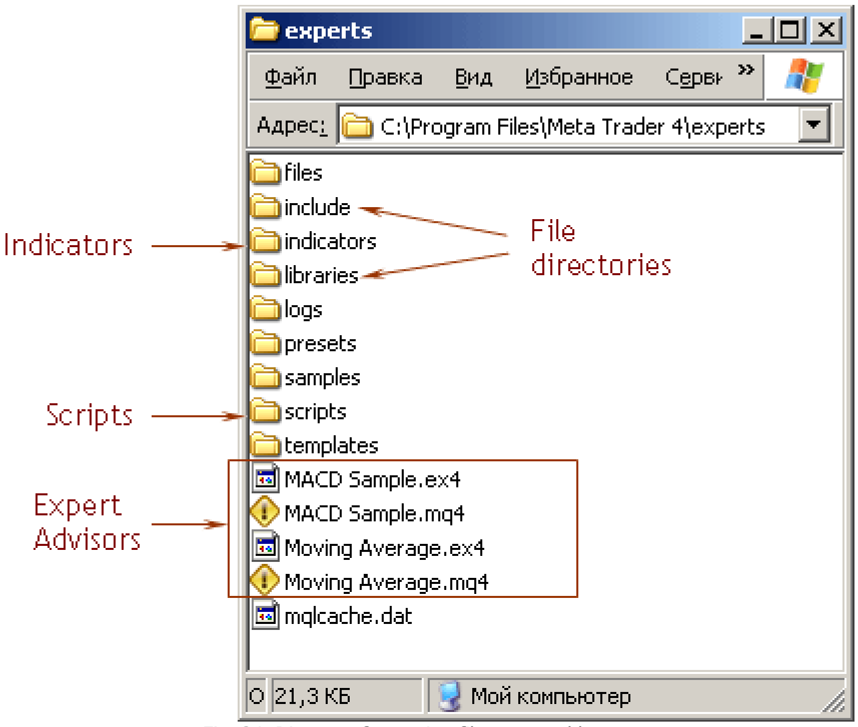


Fig. 24. Directory for storing files, created by a user.

A user can create other directories for storing some files. However, the usage of ready programs located in such directories is not provided in the client terminal.

File Types

In MQL4 there are three types of files that carry a program code: mq4, ex4 and mqh.

Files of **mq4** type represent a program source code. Files of this type contain source texts of all types of programs (Expert Advisors, scripts, indicators). For the creation of program codes MetaEditor is used. When a code is fully or partially created, it can be saved and later opened for modification. Files of mq4 type cannot be used for execution in the client terminal. For starting a program's execution, it should be first compiled. As a result of a source code compilation, a file of the same name with the extension ex4 is created.

A file of **ex4** type is a compiled program ready for practical use in the client terminal. Files of this type cannot be edited. If a program needs to be modified, this should be done in its source code (a file of mq4 type): it should be edited and then compiled again. The ex4 file name does not point to the program type - whether it is a script, an Expert Advisor or an indicator. Files with ex4 extension can be used as library files.

Files of **mqh** type are include files. It is a source text of frequently used blocks of custom programs. Such files can be included into source texts of Expert Advisors, scripts and indicators at the stage of compilation. Usually include files contain the description of imported functions (as example, see files stdlib.mqh or WinUser32.mqh) or the description of common constants and variables (stderror.mqh or WinUser.mqh). As a rule, files of mqh type are stored in the directory ***ClientTerminal_folder|experts|include***.

Include files are called so, because they are usually "included" at the stage of compilation to the main source file using the #include directive. Despite the fact that files of mqh type can contain a program source code and can be compiled by MetaEditor, they are not independent and self-contained, i.e. they do not require compilation for getting executable files of ex4 type. As include files, one can use mq4 files that should be stored in ***ClientTerminal_folder\experts\include*** .

Sections "Expert Advisors", "Custom Indicators" and "Scripts" of the client terminal navigator will show only the names of files that have the extension ex4 and are located in the corresponding folder. Files compiled in older versions of MetaEditor cannot be started and are displayed in a grey color.

There are other types of files that do not make a complete program, but are used in the creation of application programs. For example, a program can be created out of several separate files or using a library created earlier. A user can create libraries of custom functions intended for storing frequently used blocks of user programs. It is recommended to store libraries in the directory ***ClientTerminal_folder\experts\libraries***. Files of mq4 and ex4 can be used as library files. Libraries cannot start by themselves. Using include files is more preferable than using libraries because of additional consumption of computer resources at library function calls.

In the first part of the book "Programming in MQL4" we will analyze mq4 files of source texts and compiled ex4 files.

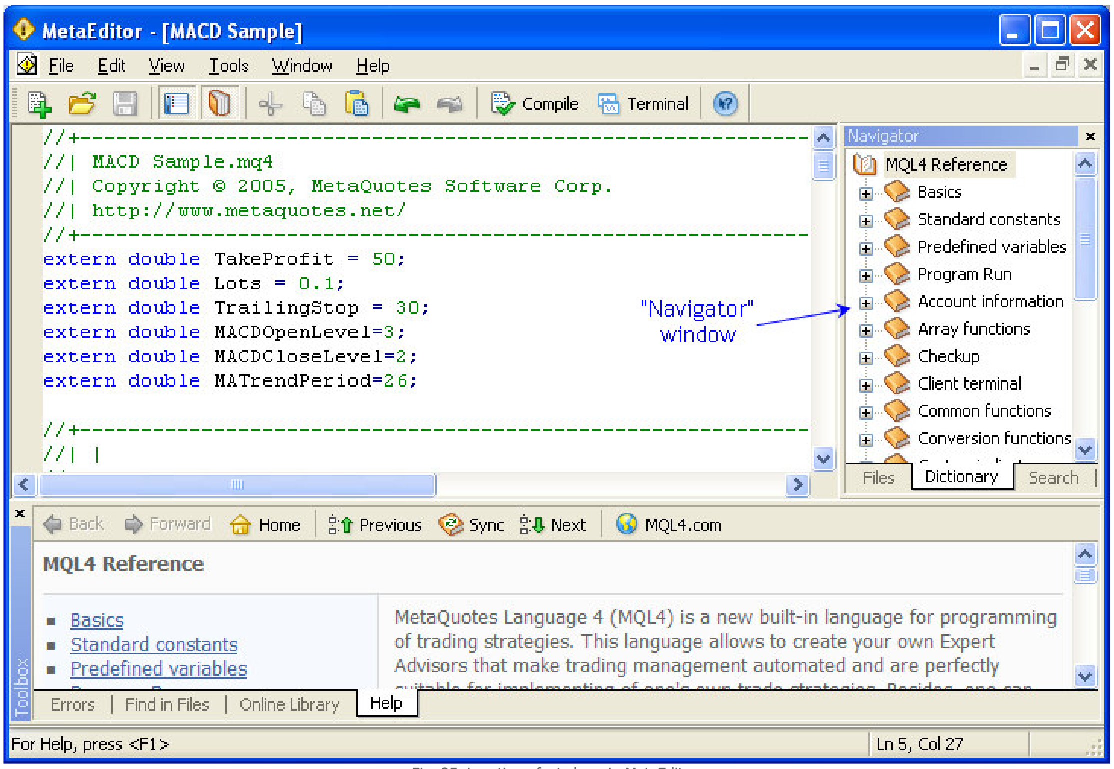
Creating and Using Programs

Application programs written in MQL4 - Expert Advisors, scripts and indicators - are created using MetaEditor.

The executable file of MetaEditor (**MetaEditor.exe**) is provided as part of the client terminal and is located in the root directory of the terminal. The Userguide of MetaEditor is opened by pressing **F1**. It contains general information necessary for the creation of new programs. The editor can be opened by clicking on the file name **MetaEditor.exe** or on a shortcut located preliminarily on the desktop.

Structure of the Client Terminal

For the convenience of operation, MetaEditor has built-in toolbars: "Navigator" (**Ctrl+D**) and "Toolbox" (**Ctrl+T**).



```
//+------------------------------------------------------------------+
//| MACD Sample.mq4
//| Copyright © 2005, MetaQuotes Software Corp.
//| http://www.metaquotes.net/
//+------------------------------------------------------------------+
extern double TakeProfit = 50;
extern double Lots = 0.1;
extern double TrailingStop = 30;
extern double MACDOpenLevel=3;
extern double MACDCloseLevel=2;
extern double MATrendPeriod=26;

//+------------------------------------------------------------------+
//| |
```

MQL4 Reference

- Basics
- Standard constants
- Predefined variables

MetaQuotes Language 4 (MQL4) is a new built-in language for programming of trading strategies. This language allows to create your own Expert Advisors that make trading management automated and are perfectly

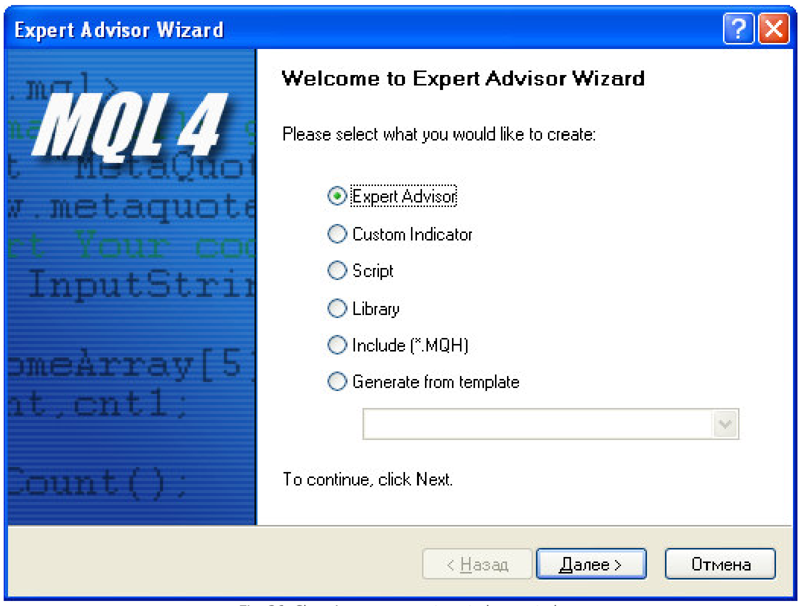
Fig. 25. Location of windows in MetaEditor.

The text of the program is located in the editor window, the toolbox windows are auxiliary. Windows of the navigator and toolbox have moving boundaries and can be shown/hidden in the editor using the buttons and .

Creating a New Program

Usually during the creation of a new program, toolbox and navigator windows are hidden. Thus the attention of a user is concentrated on a created program. To create a new program, use the editor menu **File >> Create** or a button for the creation of new files .

After all these actions **"Expert Advisor Wizard"** will offer you a list of program types for choosing one to be created:



Welcome to Expert Advisor Wizard

Please select what you would like to create:

- [✓ Expert Advisor]
- Custom Indicator
- Script
- Library
- Include (*.MQH)
- Generate from template

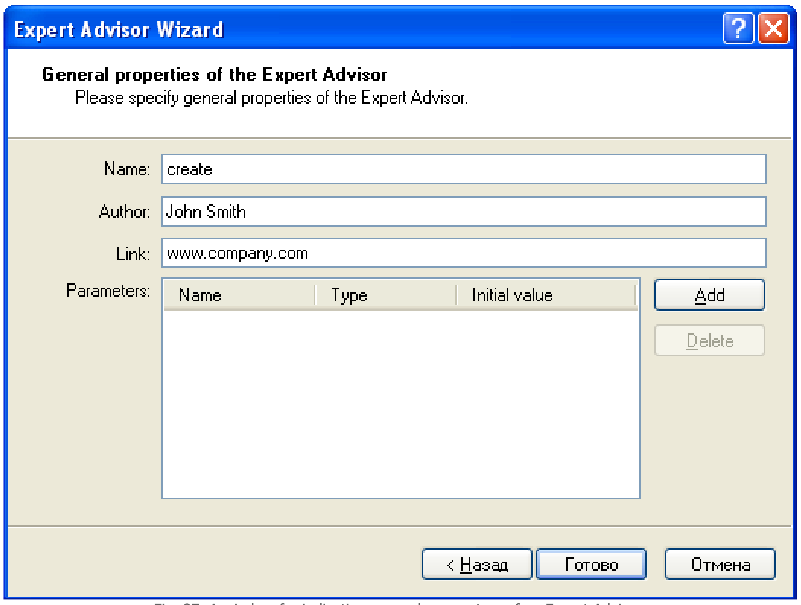
To continue, click Next.

Fig. 26. Choosing a program type to be created.

If you need to create an Expert Advisor, check **Expert Advisor** and click **Next**. In the next window it is necessary to indicate the name of a created Expert Advisor. Suppose it is called create.mq4.

The name of a created file is written without its extension (type indication).

The Expert Advisor Wizard will show a window with several fields to be filled in:



General properties of the Expert Advisor
Please specify general properties of the Expert Advisor.

Name:	create
Author:	John Smith
Link:	www.company.com

Parameters:

Name	Type	Initial value

Fig. 27. A window for indicating general parameters of an Expert Advisor.

After clicking **Ok** a text will appear in the main window and the full name of the created Expert Advisor create.mq4 will appear in the file system and in the navigator window.

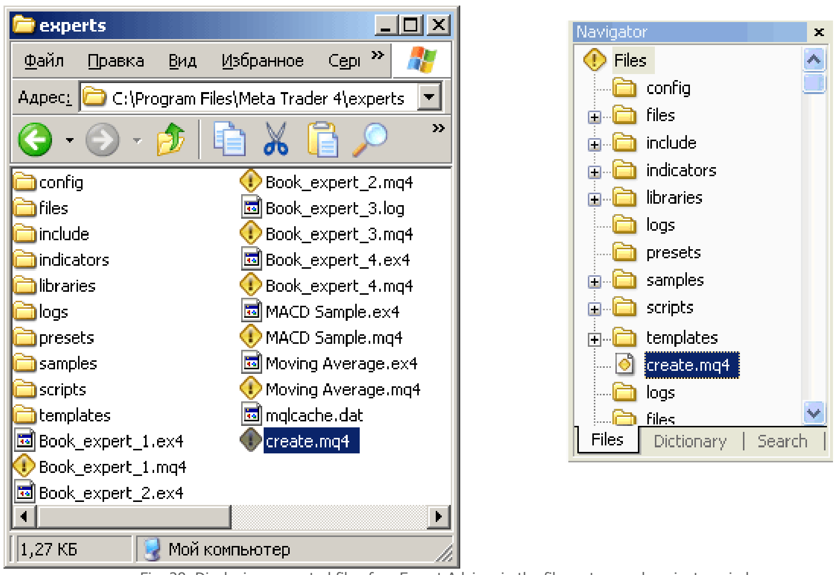


Fig. 28. Displaying a created file of an Expert Advisor in the file system and navigator window.

Let us see the program text generated by MetaEditor:

```
//+------------------------------------------------------------------+
//|                                                       create.mq4 |
//|                                                       John Smith |
//|                                                  www.company.com |
//+------------------------------------------------------------------+
#property copyright "John Smith"
#property link      "www.company.com"

//+------------------------------------------------------------------+
//| expert initialization function                                   |
//+------------------------------------------------------------------+
int init()
  {
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
//| expert deinitialization function                                 |
//+------------------------------------------------------------------+
int deinit()
  {
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
//| expert start function                                            |
//+------------------------------------------------------------------+
int start()
  {
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
```

You see, the code contains mainly comments. We already know that comments constitute a non-obligatory part of a program and the text of comments is not processed by the program.

There are three special functions in the program: init(), start() and deinit(). Each function contains only one operator - return(0) - an operator for exiting a function. Thus a program code generated by the Expert Advisor Wizard is only a pattern, using which a programmer can create a new program. The final program code can does not obligatorily contain all the indicated special functions. They are present in the pattern only because as a rule a usual medium-level program contains all these functions. If any of the functions will not be used, it's description can be deleted.

The following lines of the program code can also be omitted:

```
#property copyright "John Smith"
#property link      "www.company.com"
```

Although the program is of no practical use, it is written correctly from the point of view of syntax. And this program could be compiled and started. It would be executed like any other program (though, there would be no calculations because there are none in the source code).

Program Appearance

Using comments in programs is strongly recommended and in some cases it is strongly essential. And it must be emphasized that a programmer not only creates programs, but also reads them. Sometimes considerable difficulties may occur when reading a program. The experience of many programmers shows that the reasoning logics, on the basis of which a program was developed, can be forgotten (or unknown in a product by another programmer) and without string comments it is difficult, sometimes even impossible to understand code fragments.

A correctly coded program definitely contains comments.

The main advantages of comments are:

- Firstly, comments allow to separate one logically detached program part from another. It is much easier to read a wisely formatted text than a straight text.
- Secondly, string comments allow to explain in plain words what a programmer meant in each separate code line.
- Thirdly, in the upper part of a program, general information about a program may be specified: an author's name and contacts (including the author's web-site, e-mail, etc.), program allocation (whether it is a complete trading program or a library function), its main characteristics and limitations and other useful information.

Each programmer can choose a convenient style of comments. Style offered by MQL4 developers is presented in the Expert Advisor create.mql4. Let's view the main characteristics of any acceptable appearance styles.

1. A comment line length must not exceed the main window size. This limitation is not the language syntax formal requirement, but reading a program containing long lines is not convenient. Any long line can be separated into several lines so that all they would be fully visible on the screen. For a monitor with 1024 x 768 pixel resolution the maximal line length is 118 symbols.

2. Variable declaration is done at the program beginning. It is recommended to write a descriptive comment for each variable: shortly explain their meaning and, if required, peculiarities of usage.

3. Each operator is better placed on a separate line.

4. If there is a comment in a line, it should be started from the 76th position (recommended for 17" monitors with 1024 x 768 pixel resolution). This requirement is not obligatory. For example, if a code line takes 80 positions, it is not necessarily divided into two lines, a comment can be started from the 81st position. Usually the program code part contains 50 symbol long lines and the string comment looks like a text column in the right part of a screen.

5. For dividing logically separate fragments, continuous line comments of the full width are used (118 symbols).

6. When braces are used, a tabulation size indent must be used (usually 3 symbols).

Let us see, how an Expert Advisor may look like after a program code is written in it. In this case the program algorithm logics is not discussed. We are interested in the program appearance. A commented program (Expert Advisor create.mq4) may have the following form:

```
//--------------------------------------------------------------------
// create.mq4
// To be used as an example in MQL4 book.
//--------------------------------------------------------------------
int Count=0;                                    // Global variable
//--------------------------------------------------------------------
int init()                                      // Spec. funct. init()
   {
   Alert ("Funct. init() triggered at start");  // Alert
   return;                                      // Exit init()
   }
//--------------------------------------------------------------------
int start()                                     // Spec. funct. start()
   {
   double Price = Bid;                          // Local variable
   Count++;                                     // Ticks counter
   Alert("New tick ",Count,"   Price = ",Price);// Alert
   return;                                      // Exit start()
   }
//--------------------------------------------------------------------
int deinit()                                    // Spec. funct. deinit()
   {
   Alert ("Funct. deinit() triggered at exit"); // Alert
   return;                                      // Exit deinit()
   }
//--------------------------------------------------------------------
```

It is easy to see that complete meaningful blocks of the program are separated by comments - continuous lines. This is a way to detach special, user-defined functions and the head part of a program:

```
//--------------------------------------------------------------------
```

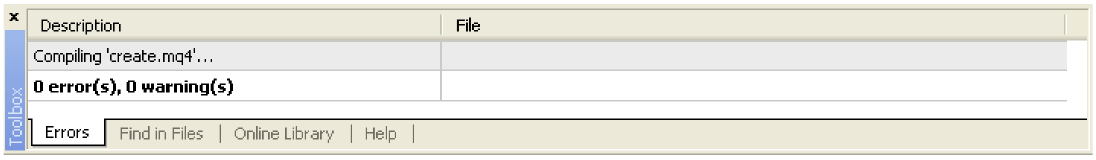
Variables are declared in a separate block where each variable is described. Sometimes programs contain variables, for describing which comments in several lines should be used. This is a rare case, but if it occurs, such a comment must be necessarily placed; otherwise not only another programmer, but the author himself will not be able to puzzle out the part after some period of time.

The right part of each code line contains an explanatory comment. The value of comments can be fully appreciated if a program does not contain any and some problems with algorithm understanding occur when reading the program. For example, if the same code is presented without comments and block separation, it will be more difficult to read it, even though the program is quite simple and short:

```
int Count=0;
int init() {
Alert (Funct. init() triggered at start");
return; }
int start() {
double Price = Bid;
Count++;
Alert("New tick ",Count,"   Price = ",Price);
return; }
int deinit(){
Alert (""Funct. deinit() triggered at exit");
return;}
```

Program Compilation

To make a program usable in practice, it must be compiled. For this purpose the button Compile (F5) in MetaEditor should be used. If a program does not contain any errors, it will be compiled and a message will occur in the toolbox:



Description	File
Compiling 'create.mq4'...	
0 error(s), 0 warning(s)	

Fig. 29. Editor message about a successful program compilation.

Besides, a new file **create.ex4** will appear in the corresponding directory (in this case in ***Terminal_directory\experts***) . This is a program ready for operation in the client terminal MetaTrader4. During compilation the last version of the source text of the program under the same name (in our case it is the file **create.mq4**) will be saved in the same directory.

Alongside with that a line with the name of the created Expert Advisor will appear in the section Expert Advisors of the client terminal navigator window:

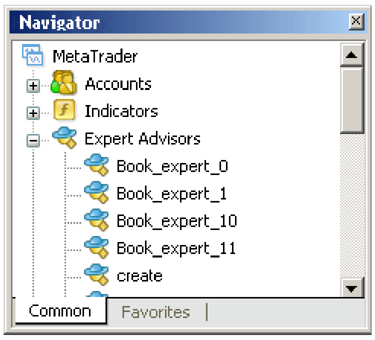


Fig. 30. Displaying the name of an Expert Advisor in the client terminal navigator window.

If during compilation errors are detected in a program, MetaEditor will show the corresponding error message. In such a case one should get back to editing the source text, fix errors and try to compile the program once again. A successful program compilation is possible only if there are no errors in the program.

Using a Program in Practice

If an application program (Expert Advisor, script or indicator) has been successfully compiled and its name has appeared in the client terminal navigator window, it can be used in practice. It is done by dragging the corresponding icon form the navigator window into a security window using a mouse ('drag & drop' method). It means the program will be attached to a security chart and started for execution.

An Expert Advisor and an indicator will operate until a user terminates the program execution manually. A usual script will stop operating itself after executing its algorithm.

It is important to note here once again that:

Any application programs (Expert Advisor, indicator, script) can be used for trading only as part of MetaTrader 4 Client Terminal when it is connected to s server (dealing center) via Internet. None of the programs can be installed on a server or used in terminals of other developers.

In other words if a trader wants to use any application program, he should switch on a computer, open MetaTrader 4 Client Terminal and start an executable file *.ex4 in a security window. During a program execution (depending on its algorithm) trading orders may be formed and sent to a server, thus conducting trade management.

Program in MQL4

It should be noted from the very beginning that programming in MQL4 is available for a common person, though requires attention and certain knowledge.

Perhaps, some traders expect difficulties in studying programming meaning that it is hard for them to imagine complicated processes running in the interior of their computers. Fortunately, developers of MQL4 language tried to make it widely available for users. A pleasant peculiarity of creating programs in MQL4 is that a programmer must not necessarily have special knowledge about the interaction of the client terminal with an operating system, network protocol characteristics or a compiler stucture.

A process of creating programs in MQL4 is an execution of a simple and friendly work. For example, a driver does not have to know a motor structure for driving a car - he only needs to learn pedaling and steering. However, before driving a car on vibrant streets, each driver has to undergo training. Something like that should be done by a starting programmer - learning some simple principles of creating programs and after that slowly starting to "drive".

- Program Structure
  Although there are numerous types of programs in MQL4, all they have general features. It may be said, that a correct structure is the basis of a correctly written code. That's why it is necessary to understand the components of a program.

- Special Functions
  There are a lot of functions built in the MQL4 language. Such functions are called standard functions of the language. But there are several extremely important functions, which are called special functions. A program cannot be executed without them. Each of these functions has its own allocation.

- Program Execution
  One must correctly understand how a MQL4-program operates. Not all code parts are used with the same frequency. What functions are executed in the first instance, where the main part of a program should be placed, what program type should be used for this or that purpose?

- Examples of Implementation
  A new language is better learned on examples. How to write correctly a simple program? What errors can occur?

← Creating and Using Programs Program Structure →

Program Structure

In first sections we learned some basic notions of the programming language MQL4. Now let's study how a program is organized in general. To solve this problem we will study its structural scheme.

As already mentioned above, the main program code written by a programmer is placed inside user-defined and special functions. In the section Functions we have discussed the notion and properties of built-in and user-defined functions. Shortly: a user-defined function has a description, function call is used for starting its execution in a program. Any built-in or user-defined function can be executed only after it is called; in such a case the function is said to be called for execution by a program.

Properties of special functions are described in details in the section Special Functions. Here we will study only the main information about them. Special function is a function called to be executed by the client terminal. As distinct from common functions, special functions have only description and special functions call is not specified in a program. Special functions are called to be executed by the client terminal (there is also a technical possibility to call special functions from a program, but we will consider this method incorrect and will not discuss it here). When a program is started for execution in a security window, the client terminal passes control to one of the special functions. As a result this function is executed.

The rule of programming in MQL4 is the following:

A program code must be written inside functions.

It means, program lines (operators and function calls) that are outside the function cannot be executed. At the attempt to compile such a program, MetaEditor will show the corresponding error message and the executable file *.exe will not appear as a result of compilation.

Let's consider the functional scheme of a common program - Expert Advisor:

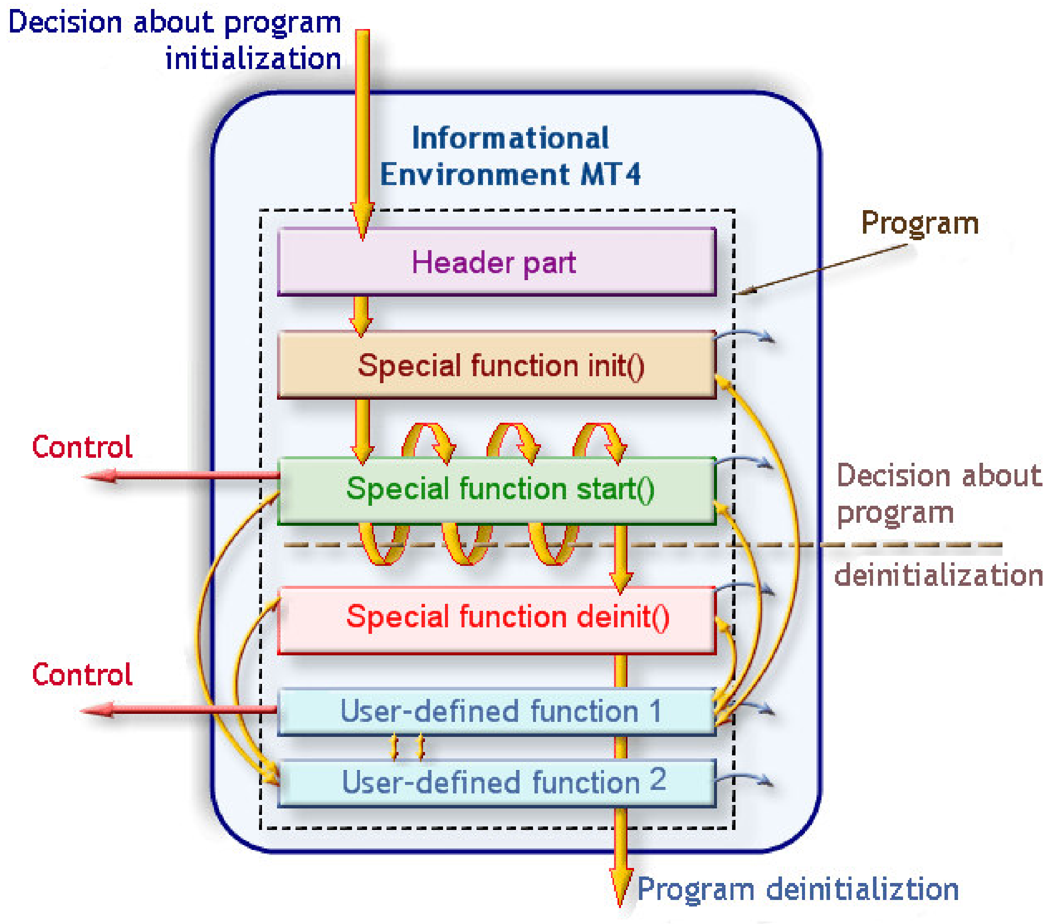


Fig. 31. Functional scheme of a program (Expert Advisor).

The largest blocks of a program written in MQL4 are:

1. Head part of a program.
2. Special function init().
3. Special function start().
4. Special function deinit().
5. User-defined functions.

Further we will analyze only the inner content of these functional blocks (integral parts) of a program, while all external objects (for example, informational sphere of the client terminal or hardware) will be out of our scope of interest.

Information Environment of MetaTrader 4 Client Terminal

Information environment of the client terminal MT4 is not a component part of the program. Information environment is a set of parameters available to be processed by a program. For example, it is a security price that has come with a new tick, volume accumulated at each new tick, information about maximal and minimal prices of history bars, parameters that characterize trading conditions offered by a dealing center, etc. Information environment is always saved and at each new tick is updated by the client terminal connected with the server.

Program Structure

Head Part

Head part consists of several lines at the beginning of a program (starting from the first line) that contain some writings. These lines contain general information about the program. For example, this part includes lines of declaration and initialization of global variables (the necessity to include this or that information into the head part will be discussed later). The sign of the head part end may be the next line containing a function description (user-defined or special function).

Special Functions

Usually after the head part of the program special functions are described. The special function description looks like the description of a usual user-defined function, but special functions have predefined names: init(), start() and deinit(). Special functions are a block of calculations and are in relations with the information environment of the client terminal and user-defined functions. Special functions are described in details in the section Special Functions.

User-Defined Functions

The description of user-defined functions is usually given after the description of special functions. The number of user-defined functions in a program is not limited. The scheme contains only two user-defined functions, but a program can contain 10 or 500, or none. If no user-defined functions are used in a program, the program will be of a simplified structure: head part and description of special functions.

Standard Functions

As mentioned earlier, standard functions can be presented only as a function call. Standard functions, like special and custom functions, have description. However, this description is not given in a program (that is why not included into the scheme). The description of a standard function is hidden, not visible to a programmer and therefore cannot be changed. Though it is available to MetaEditor. During program compilation, MetaEditor will form an executable file, in which all called standard functions will be executed correctly to the full extent.

Arrangement of Parts in a Program

The head part should be located in first lines. The arrangement of special and user-defined functions descriptions does not matter. Fig. 32 shows a common arrangement of functional blocks, namely - the head part, special functions, user-defined functions. Fig. 33 shows other program structure variants. In all examples the head part comes first, while functions can be described in a random order.

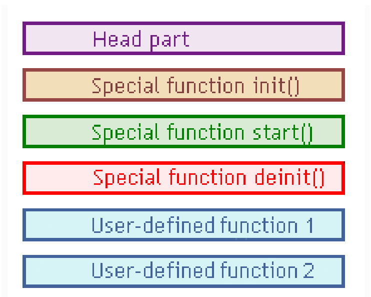


Fig. 32. Usual arrangement of functional blocks in a program (recommended).

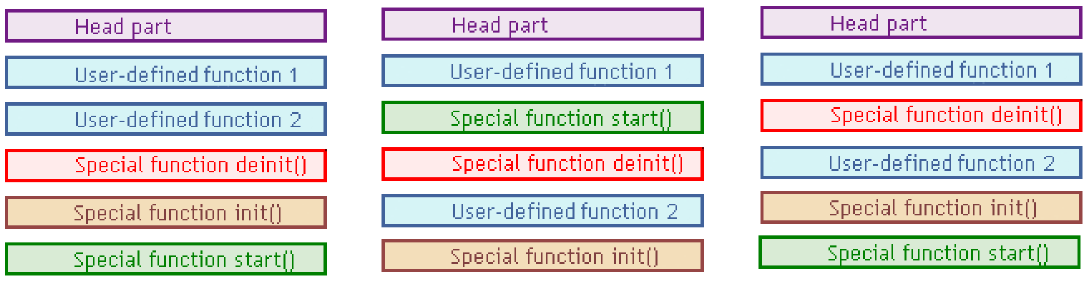


Fig. 33. Possible ways of arranging functional blocks in a program (random order).

Please, note:

None of the functions can be described inside another function. It is prohibited to use in a program function descriptions located inside another function.

Below are examples of incorrect arrangement of function descriptions.

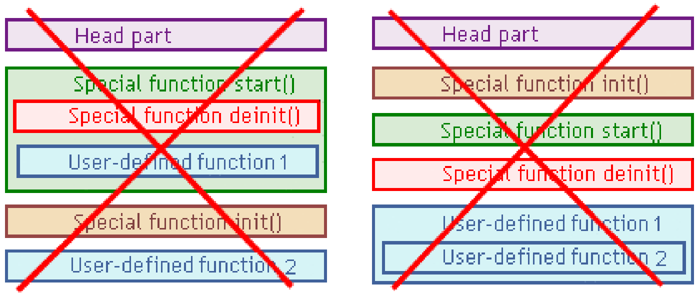


Fig. 34. Examples of incorrect arrangement of function in a program.

If a programmer by mistake creates a program where description of any of its functions is located inside the description of another function, on the compilation stage MetaEditor will show an error message and an executable file will not be created for such a program.

Code Execution Sequence

Head Part and Special Functions

From the moment of a program execution start in a security window, lines of the program head part are executed.

After the preparations described in the head part are done, the client terminal passes controlling to the special function init() and the function is executed (control passing is shown at the structural scheme in large yellow arrows). The special function init() is called for execution only once at the beginning of the program operation. Usually this function contains a code that should be executed only once before the main operation of the program starts. For example, when the init() function is executed, some global variables are initialized, graphical objects are displayed in a chart window, messages may be shown. After all program lines in the init() function are executed, the function finishes its execution and control is returned to the client terminal.

The main program operation time is the period of operation of the function start(). In some conditions (see features of special functions in the section Special Functions), including the receipt of a new tick by the client terminal from a server, the client terminal calls for execution the special function start(). This function (like other functions) can refer to the information environment of the client terminal, conduct necessary calculations, open and close orders, i.e. perform any actions allowed by MQL4. Usually when the special function start() is executed, a solution is produced that is implemented as a control action (red arrow). This control can be implemented as a trading request to open, close or modify an order formed by the program.

After the whole code of the EA's special function start() is executed, the function start() finishes its operation and returns control to the client terminal. The terminal will hold the control for some time not starting any of special functions. A pause appears, during which the program does not work. Later, when a new tick comes, the client terminal will pass control to the special function start() once again, as a result the function will be executed and after its execution is finished, control will be returned to the client terminal. On the next tick the function start() will be started by the client terminal once again.

The process of multiple call of the special function start() by the client terminal will be repeated while the program is attached to a chart and can continue for weeks and months. During all this period an Expert Advisor can conduct automated trading, i.e. implement its main assignment. At the scheme the process of multiple execution of the function start() is shown by several yellow arrow enveloping the special function start().

When a trader removes an Expert Advisor from a chart, the client terminal once starts the special function deinit(). The execution of this function is necessary for the correct termination of an EA's operation. During operation a program may, for example, create graphical objects and global variables of the client terminal. That is why the code of the function deinit() contains program lines, execution of which results in deletion of unnecessary objects and variables. After the execution of the special function deinit() is over, control is returned to the client terminal.

Executed special functions can refer to information environment (thin blue arrows at the scheme) and call for execution user-defined functions (thin yellow arrows). Note that special functions are executed after they are called by the client terminal in the order predefined in function properties: first init(), then multiple call of start() and after that deinit(). Conditions, at which the client terminal calls special functions, are described in the section Special Functions.

User-Defined Functions

User-defined functions are executed when call to a user-defined function is contained in some function. In this case control is timely passed to the user-defined function and after the function execution is over, control is returned to the place of call (thin orange arrows at the scheme). Call of user-defined functions can be contained not only in the description of a special function, but also in the description of other user-defined functions called from it. One user-defined functions may call other user-defined functions - this user-defined functions calling order is permissible and widely used in programming.

User-defined functions are not called for execution by the client terminal. Any user-defined functions are executed inside the execution of a special function that return control to the client terminal. User-defined functions may also request (use) for processing variable values of the client terminal information environment (thin blue arrows at the scheme).

If a program contains description of a user-defined function, but there is no call of this function, this user-defined function will be excluded from the ready program on the compilation stage and will not be used in the program's operation.

Note: special functions are called for execution by the client terminal. User-defined functions are executed if called from special or other user-defined functions, but are never called by the client terminal. Control action (trading orders) can be formed both in special and user-defined functions.

← Program in MQL4 Special Functions →

Special Functions

The distinctive feature of programs intended for operation in the client terminal MetaTrader 4 is their work with constantly updated information in real-time mode. In MQL4 language this peculiarity is reflected in the form of three special functions: init(), start() and deinit().

Special functions are functions with predefined names **init(), start()** and **deinit()** possessing own special properties.

Properties of Special Functions

Common Property of Special Functions

The main property of all special functions is their execution in a program under certain conditions without using special function call from the program. Special functions are called for execution by the client terminal. If a program contains the description of a special function, it will be called (and executed) in accordance with calling conditions (own properties).

Special functions are called for execution by the client terminal.

Own Properties of Special Functions

Special Function init().

The own property of the special function init() is its execution at the program initialization. If a program contains the description of the special function init(), it will be called (and executed) at the moment the program starts. If there is no special function init() in a program, no actions will be performed at a program start.

In Expert Advisors the special function init() is called (and executed) after the client terminal start and uploading of historic data, after changing security and/or chart period, after program recompilation in MetaEditor, after changing input parameters from EA setup window and after account changing.

In scripts the special function init() is also called (and executed) immediately after it is to a chart.

In custom indicators the special function init() is called (and executed) immediately after the client terminal start, after changing security and/or chart period, after program recompilation in MetaEditor and after changing input parameters from the custom indicator setup window.

Special Function start().

The own properties of the special function start() differ depending on the executable program type.

In Expert Advisors the special function start() is called (and executed) immediately after a new tick comes. If a new tick has come during the execution of the special function start(), this tick will be ignored, i.e. the special function start() will not be called for execution when such a tick comes. All quotes received during the execution of the special function start() will be ignored. Start of the special function start() for execution is performed by the client terminal only provided that the previous operation session has been completed, the control has been returned to the client terminal and the special function start() is waiting for a new tick.

The possibility to call and execute the special function start() is influenced by the state of "Enable/disable Expert Advisors" button. If this button is in the state of disabling EAs, the client terminal will not call for execution the special function start() irrespective of whether new quotes come or not. However, changing the button state from enabled to disabled does not terminate the current operation session of the special function start().

The special function start() is not called by the client terminal if EA properties window is open. The EA properties window can be opened only when the special function start() is waiting for a new tick. This window cannot be opened during the execution session of the EA's special function start().

In scripts the special function start() is called (and executed) once immediately after program initialization in the special function init().

In custom indicators the special function start() is called (and executed) immediately after a new tick comes, immediately after being attached to a chart, when changing a security window size, when switching from one security to another, when starting the client terminal (if during the previous session an indicator was attached to a chart), after changing a symbol and period of a current chart irrespective of the fact whether new quotes come or not.

Termination of a current session of start() execution for all program types can be performed when a program is removed from a chart, when security and/or chart period are changed, when an account is changed/chart is closed and as a result of client terminal's operation termination. If the special function start() was executed during the shutdown command, time available for the terminal to complete execution of the function is 2.5 seconds. If after the shutdown command the special function start() continues its operation for more than the indicated time limit, it will be forcibly terminated by the client terminal.

Special Function deinit().

The own function of the special function deinit() is its execution at program termination (deinitialization). If a program contains description of the special function deinit(), it will be called (and executed) at a program shutdown. If a program does not contain th especial function deinit(), no actions will be performed at program shutdown.

The special function deinit() is also called for execution by the client terminal at the terminal shutdown, when a security window is closed, right before changing a security and/or chart period, at a successful program recompilation in MetaEditor, when changing input parameters, as well as when an account is changed.

In Expert Advisors and scripts program shutdown with the necessary call of the special function deinit() may happen when attaching to a chart a new program of the same type that replaces the previous one.

In custom indicators the special function deinit() is not executed when a new indicator is attached to a chart. Several indicators can operate on a security window, that is why the attachment of a new indicator to a chart does not result in the shutdown of other indicators with deinit() call.

The time of deinit() execution is limited to 2.5 seconds. If the code of the special function deinit() is executed longer, the client terminal will forcibly terminate the execution of the special function deinit() and operation of the program.

Requirements for Special Functions

Special functions init() and deinit() can be absent in a program. The order of special functions description in a program does not matter. Special functions can be called from any program part in accordance with the general order of function calls.

Special functions may have parameters. However, when these functions are called by the client terminal, no parameters will be sent from outside, only default values will be used.

Special functions init() and deinit() must finish their operation maximally quickly and in no case run into a cycle path trying to start the whole operation before calling the function start().

Order of Using Special Functions

Developers have presented to programmers a very convenient tool: when a program starts, first of all init() is executed, after that the main work is performed with the help of the function start(), and when a user finishes his work, the function deinit() will be started prior to the program shutdown.

The main code of a program must be contained in the function start(). All operators, built-in and custom functions calls and all necessary calculations should be performed inside this function. At the same time one must correctly understand the role of custom functions. The description of custom functions is located in a program code outside the description of special functions, but if a custom function is called for execution, a special function does not terminate its operation. It means control fr some time is passed to the custom function, but the custom function itself operates within a special function that has called it. So, in the process of a program execution special function operate always (in accordance with their own properties) and custom function are executed when called by special functions.

If a programmer is not going to use any of the special functions, he can refuse from using it in a program. In such a case the client terminal will not call it. Absolutely normal program is one containing all the three special functions. A program that does not have init() or deinit() or both these functions is also considered normal.

If none of the special functions is used in a program, this program will not be executed. Client terminal calls for execution only special function in accordance with their properties. Custom functions are not called by the client terminal. That's why if a program does not contain special functions at all (and contains only custom functions), it will be never called for execution.

It is not recommended to call start() function from the special function init() or perform trade operations from init (), because during initialization values of information environment parameters may be not ready (information about charts, market prices, etc.).

Sections Program Execution and Examples of Implementation contain several practical examples that will help to see some properties of special functions.

← Program Structure Program Execution →

Program Execution

Programming skills are better developed if a programmer has a small operating program at his disposal. To understand the whole program, it is necessary to examine thoroughly all its components and trace its operation step-by-step. Please note, special function properties of different application programs (Expert Advisors, scripts, indicators) are different. Now we will analyze how an Expert Advisor operates.

Example of a simple Expert Advisor (simple.mq4)

```
//--------------------------------------------------------------------
// simple.mq4
// To be used as an example in MQL4 book.
//--------------------------------------------------------------------
int Count=0;                                    // Global variable
//--------------------------------------------------------------------
int init()                                      // Spec. funct. init()
   {
   Alert ("Function init() triggered at start");// Alert
   return;                                      // Exit init()
   }
//--------------------------------------------------------------------
int start()                                     // Spec. funct. start()
   {
   double Price = Bid;                          // Local variable
   Count++;                                     // Tick counter
   Alert("New tick ",Count,"   Price = ",Price);// Alert
   return;                                      // Exit start()
   }
//--------------------------------------------------------------------
int deinit()                                    // Spec. funct. deinit()
   {
   Alert ("Function deinit() triggered at deinitialization");   // Alert
   return;                                      // Exit deinit()
   }
//--------------------------------------------------------------------
```

In accordance with program execution rules (see Program Structure and Special Functions) this Expert Advisor will work the following way:

1. At the moment when a program is attached to a chart, the client terminal passes control to the program and as a result the program will start its execution. The program execution starts from the head part. The head part contains only one line:

```
int Count=0;                                    // Global variable
```

In this line the global variable Count initialized by zero value is declared. (Local and global variables are analyzed in details in the section Types of Variables. It should be noted here, that algorithm used in this certain program requires declaration of the variable Count as global, that's why it cannot be declared inside a function, it must be declared outside functions description, i.e. in the head part; as a result the value of the global variable Count will be available from any program part.)

2. After the execution of the program head part, the special function init() will be started for execution. Please note, this function call is not contained in a program code. The start of init() execution when an EA is attached to a chart is the function's own property. The client terminal will call init() for execution just because the program code contains its description. In the analyzed program the description of the special function init() is the following:

```
int init()                                      // Spec. funct. init()
   {
   Alert ("Function init() triggered at start");// Alert
   return;                                      // Exit init()
   }
```

The function body contains only two operators.

2.1 Function Alert() shows an alert window:

```
Function init() triggered at start
```

2.2 Operator return finishes the operation of the special function init()

As a result of init() execution, an alert will be written. In really used programs such an algorithm is very rare, because such init() usage is of little help. Really, no sense to use a function that informs a trader that it is being executed. Here the algorithm is used only for the visualization of init() execution. Pay attention: the special function init() is executed in a program only once. The function execution takes place at the beginning of program operation after a head part has been processed. When the operator return is executed in the special function init(), the program returns control to the client terminal.

3. The client terminal detected the description of the special function start() in the program:

```
int start()                                       // Special funct. start()
   {
   double Price = Bid;                            // Local variable
   Count++;
   Alert("New tick ",Count,"   Price = ",Price);// Alert
   return;                                        // Exit start()
   }
```

31. Control is held by the client terminal. The client terminal waits for a new tick and does not start program functions until a new tick comes. It means, for some time the program does not operate, i.e. no actions are performed in it. A pause appears, though there is neither direct nor indirect prompting to perform this pause. The necessity to wait for a tick is the start() function's own property and there is no way of a program influence upon this property (for example, disabling it). The program will be waiting for the control until a new tick comes. When a new tick comes, the client terminal passes control to the program, namely to the special function start() (in this case in accordance with the EA's start() function property). As a result its execution starts.

32 (1). In the line

```
   double Price = Bid;                            // Local variable
```

the following actions are performed:

32.1(1). Declaration of a local variable Price (see Types of Variables). Value of this local variables will be available from any part of the special function start().

32.2(1). Execution of the assignment operator. The current Bid price value will be assigned to the variable Price. New price value appears each time when a new tick comes (for example, at the first tick a security price can be equal to 1.2744).

33(1). Then the following line is executed:

```
   Count++;
```

This not usual record is the full analog of Count=Count+1;

By the moment of passing control to this line the Count variable value is equal to zero. As a result of Count++ execution, the value of Count will be increased by one. So, by the moment of passing control to the next line, Count value will be equal to 1.

34(1). The next line contains Alert() function call:

```
   Alert ("New tick ",Count,"   Price = ",Price);// Alert
```

The function will write all constants and variables enumerated in brackets.

At the first execution of the function start() the program will write New tick, then refer to the variable Count for getting its value (at first execution this value is 1), write this value, then will write Price = and refer to the variable Price for getting its value (in our example it is 1.2744) and writing it.

As a result the following line will be written:

```
New tick 1 Price = 1.2744
```

35(1). Operator

```
    return;                                // Exit start()
```

finishes the operation of the special function start().

36. Control is returned to the client terminal (until a new tick comes).

This is how start() function of an Expert Advisor is executed. When the execution is over, the special function start() returns control to the client terminal and when a new tick comes the client terminal starts its operation once again. This process (starting the execution of the function start() and returning control to the client terminal) can go on for a long time - several days or weeks. During all this time the special function start() will be executed from time to time. Depending on environment parameters (new prices, time, trade conditions, etc.) in the special function start() different actions like opening or modifying orders can be performed.

37. From the moment of receipt of a new tick, actions of points 32-36 are repeated. However, only the sequence of executed operators is repeated, but variables get new values each time. Let's view differences between the first and the second execution of the special function start().

32 (2). In the line

```
    double Price = Bid;                   // Local variable
```

the following actions are performed:

32.1(2). Declaration of a local variable Price (unchanged).

32.2(2). Execution of the Assignment operator. The current Bid price value will be assigned to the variable Price (new price value appears each time when a new tick comes, for example, at the second tick the security price will be equal to 1.2745) (there are changes).

33(2). Then the following line will be executed:

```
    Count++;
```

At the moment prior to passing control to this line, the value of the variable Count (after the first execution of the function start()) is equal to 1. As a result of Count++ execution, Count value will be increased by one. Thus, at the second execution Count will be equal to 2 (changed).

34(2). Alert() function:

```
    Alert ("New tick",Count,"   Price = ",Price);// Alert
```

writes all constants and variables (their new values) enumerated in brackets.

At the second start() execution the program will write New tick, then refer to Count variable for getting its value (in second execution it is equal to 2), write this value, then write Price = and refer to Price variable, get its value (in our example 1.2745) and write it (changed).

As a result the following line will be written:

```
New tick 2 Price = 1.2745
```

35(2). Operator

```
        return;                                // Exit из start()
```

finishes operation of start() (no changes).

36(2). Control is returned to the client terminal to wait for a new tick.

37(2). Then it is repeated again. In the third start() execution variables will get new values and will be written by the function Alert (), i.e. the program repeats points 32-36(3). And then again and again: 32 - 36(4), 32 - 36(5),..(6)..(7)..(8)... If a user does not take any actions, this process will be repeated endlessly. As a result of start() operation in this program we will see tick history of price change.

Next events will happen only when a user decides to terminate the program and forcibly detaches the program from a chart manually.

4. Client terminal passes control to the special function deinit() (in accordance with its properties).

```
int deinit()                                    // Special funct. deinit()
   {
   Alert ("Function deinit() triggered at exit");   // Alert
   return;                                      // Exit deinit()
   }
```

There are only two operators in the function body.

41. Alert() will write:

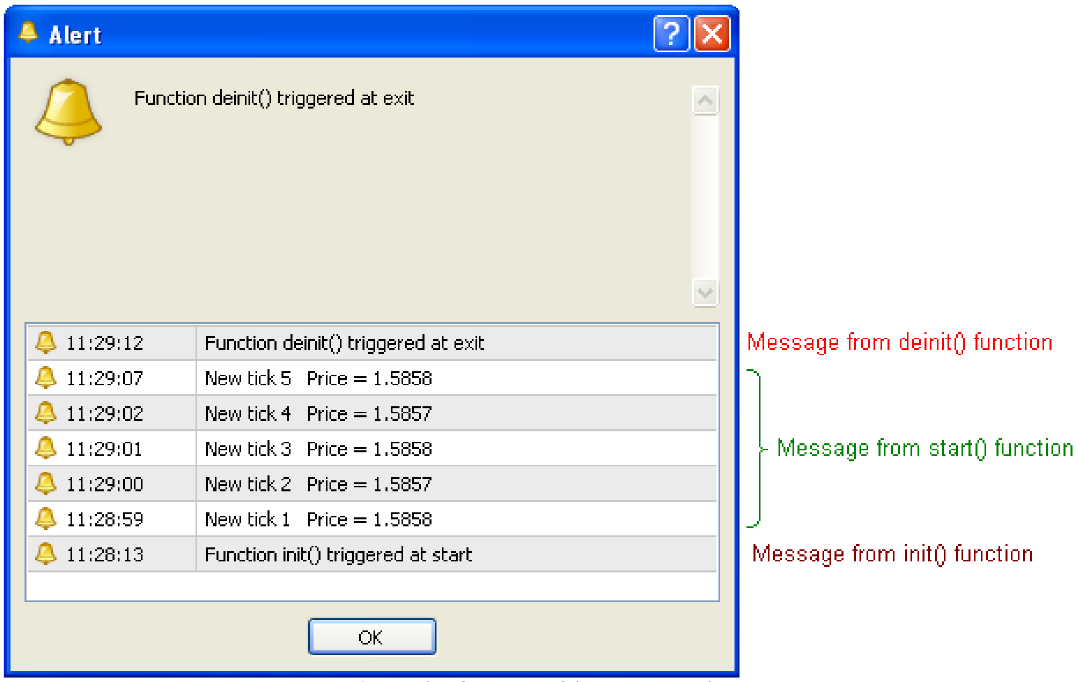
```
Function deinit() triggered at deinitialization
```

42. Operator return finishes operation of deinit().

The function deinit() is started for execution by the client terminal only once, after that the above alert will appear in Alert() window and the program will be removed from a chart.

5. Here the story of Expert Advisor execution ends.

Attach this example program to any chart and start it. The operating program will display a window containing all alerts generated by the function Alert(). By the contentc of alerts it is easy to understand what special function is connected with this or that entry.


Alert
OK
Message from deinit() function
Message from start() function
Message from init() function

Function deinit() triggered at exit

```
11:29:12   Function deinit() triggered at exit
11:29:07   New tick 5   Price = 1.5858
11:29:02   New tick 4   Price = 1.5857
11:29:01   New tick 3   Price = 1.5858
11:29:00   New tick 2   Price = 1.5857
11:28:59   New tick 1   Price = 1.5858
11:28:13   Function init() triggered at start
```

Fig. 35. Results of operation of the program simple.mq4.

From this example you can easily see that a program is executed in accordance with special functions properties described in Special Functions. Terminate the program and start it again. Having done this several times, you will get experience of using your first program. It will work both now and the next time. Further programs that you will write yourself will also be constructed in accordance with the described structure and for starting their execution you will also need to attach it to a chart.

Try to understand all concepts and rules and the process of creating programs in MQL4 will be easy and pleasant.

← Special Functions Examples of Implementation

Examples of Implementation

In the previous section we analyzed an example of special functions execution in a simple Expert Advisor simple.mq4. For better practice let's analyze some more modifications of this program.

✓ Example of a correct program structure

As a rule, function descriptions are indicated in the same sequence as they are called for execution by the client terminal, namely first goes the description of the special function init(), then start() and the last one is deinit(). However, special functions are called for execution by the client terminal in accordance with their own properties, that is why the location of a description in a program does not matter. Let us change the order of descriptions and see the result (Expert Advisor possible.mq4).

```
//--------------------------------------------------------------------
// possible.mq4
// To be used as an example in MQL4 book.
//--------------------------------------------------------------------
int Count=0;                                 // Global variable
//--------------------------------------------------------------------
int start()                                  // Special funct. start()
   {
   double Price = Bid;                       // Local variable
   Count++;
   Alert("New tick ",Count,"   Price = ",Price);// Alert
   return;                                   // exit start()
   }
//--------------------------------------------------------------------
int init()                                   // Special funct. init()
   {
   Alert ("Function init() triggered at start");// Alert
   return;                                   // Exit init()
   }
//--------------------------------------------------------------------
int deinit()                                 // Special funct. deinit()
   {
   Alert ("Function deinit() triggered at exit");// Alert
   return;                                   // Exit deinit()
   }
//--------------------------------------------------------------------
```

Starting this Expert Advisor you will see that the execution sequence of special functions in a program does not depend on the order of descriptions in a program. You may change the positions of function descriptions in a a source code and the result will be the same as in the execution of the Expert Advisor simple.mq4.

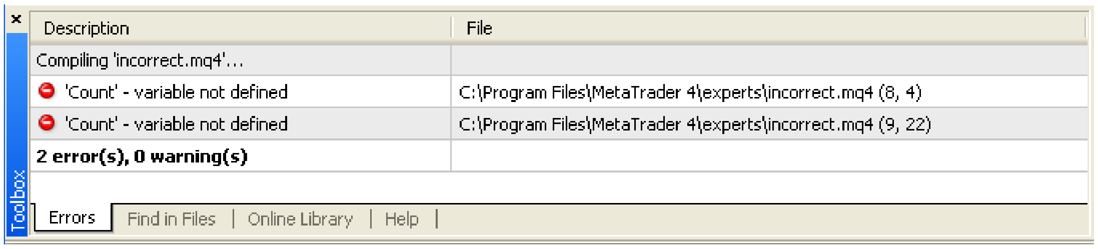
✗ Example of incorrect program structure

But the program will behave in a different way if we change the head part position. In our example we will indicate start() earlier than the head part (Expert Advisri incorrect.mq4):

```
//--------------------------------------------------------------------
// incorrect.mq4
// To be used as an example in MQL4 book.
//--------------------------------------------------------------------
int start()                                  // Special funct. start()
   {
   double Price = Bid;                       // Local variable
   Count++;
   Alert ("New tick ",Count,"   Price = ",Price);// Alert
   return;                                   // Exit start()
   }
//--------------------------------------------------------------------
int Count=0;                                 // Global variable
//--------------------------------------------------------------------
int init()                                   // Special funct. init()
   {
   Alert ("Function init() triggered at start");// Alert
   return;                                   // Exit init()
   }
//--------------------------------------------------------------------
int deinit()                                 // Special funct. deinit()
```

```
     {
     Alert ("Function deinit() triggered at exit");// Alert
     return;                                      // Exit deinit()
     }
  //--------------------------------------------------------------------
```

When trying to compile this Expert Advisor, MetaEditor will show an error message:



Description	File
Compiling 'incorrect.mq4'...	
'Count' - variable not defined	C:\Program Files\MetaTrader 4\experts\incorrect.mq4 (8, 4)
'Count' - variable not defined	C:\Program Files\MetaTrader 4\experts\incorrect.mq4 (9, 22)
2 error(s), 0 warning(s)	

Fig. 36. Error message during incorrect.mq4 program compilation.

In this case the line

```
  int Count=0;                                    // Global variable
```

is written outside all functions, but is not at the very beginning of a program, but somewhere in the middle of it.

The defining moment in the program structure is that the declaration of the global variable Count is done after function declaring (in our case - special function start()). In this section we will not discuss details of using global variables; types of variables and usage rules are described in the section Variables. It should be noted here that any global variable must be declared earlier (earlier in text) than the first call to it (in our case it is in the function start()). In the analyzed program this rule was violated and the compiler displayed an error message.

Example of using a custom function

Now let's see how the program behaves in relation to custom functions. For this purpose let's upgrade the code described in the example of a simple Expert Advisor simple.mq4 and then analyze it. A program with a custom function will look like this (Expert Advisor userfunction.mq4):

```
  //--------------------------------------------------------------------
  // userfunction.mq4
  // To be used as an example in MQL4 book.
  //--------------------------------------------------------------------
  int Count=0;                                    // Global variable
  //--------------------------------------------------------------------
  int init()                                      // Special funct. init()
     {
     Alert ("Function init() triggered at start");// Alert
     return;                                      // Exit init()
     }
  //--------------------------------------------------------------------
  int start()                                     // Special funct. start()
     {
     double Price = Bid;                          // Local variable
     My_Function();                               // Custom funct. call
     Alert("New tick ",Count,"   Price = ",Price);// Alert
     return;                                      // Exit start()
     }
  //--------------------------------------------------------------------
  int deinit()                                    // Special funct. deinit()
     {
     Alert ("Function deinit() triggered at exit");// Alert
     return;                                      // Exit deinit()
     }
  //--------------------------------------------------------------------
  int My_Function()                               // Custom funct. description
     {
     Count++;                                     // Counter of calls
     }
  //--------------------------------------------------------------------
```

First of all let's see what has chaned and what has remained unchanged.

Unchanged parts:

1. The head part is unchanged.

```
// userfunction.mq4
// To be used as an example in MQL4 book.
//--------------------------------------------------------------------
int Count=0;                                     // Global variable
```

2. Special function init() is unchanged.

```
int init()                                       // Special funct. init()
  {
   Alert ("Function init() triggered at start");// Aler
   return;                                       // Exit init()
  }
```

3. Special function deinit() is unchanged.

```
int deinit()                                     // Special funct. deinit()
  {
  Alert("Function deinit() triggered at exit"); // Alert
  return;                                        // Exit deinit()
  }
```

Changes:

1. Added: custom function My_Function() .

```
int My_Function()                                // Custom function description
  {
   Count++;                                      // Counter of calls
  }
```

2. The code of the special function start() has also changed: now it contains the custom function call, but there is no Count variable calculation line now.

```
int start()                                      // Special funct. start()
  {
   double Price = Bid;                           // Local variable
   My_Function();                                // Custom function call
   Alert("New tick ",Count,"   Price = ",Price);// Alert
   return;                                       // Exit start()
  }
```

In the section Program Execution we analyzed the order of init() and deinit() execution. In this example these functions will be executed the same way, so we will not dwell on their operation. Let's analyze the execution of the special function start() and the custom function My_Function(). The custom function description is located outside all special functions as it must be. The custom function call is indicated in start() code, which is also correct.

After init() is executed, the program will be executed so:

31.The special function start() is waiting to be started by the client terminal. When a new tick comes, the terminal will start this function for execution. As a result the following actions will be performed:

32 (1). In the line

```
      double Price = Bid;                         // Local variable
```

the same actions are performed:

32.1(1). Local variable Price is initialized (see Types of Variables). Value of this local variable will be available from any part of the special function start().

32.2(1). Assignment operator is executed. The last available Bid price will be assigned to the variable Price (for example at the first tick it is equal to 1.2744).

33(1). Next comes My_Function() call:

```
      My_Function();                                 // Custom function call
```

This line will be executed within the start() operation. The result of this code part implementation (custom function call) is passing control to the function body (description) with further returning it to the call place.

34(1). There is only one operator in the custom function description:

```
      Count++;
```

At the first custom function call Count is equal to zero. As the result of Count++ operator execution the value of Count will be increased by one. Having executed this operator (the only and the last one) the custom function finishes its operation and returns control to the place, from which it has been called.

It should be noted here that custom functions may be called only from special functions (or from other custom functions that are called from special functions). That is why the following statement is correct: at any current moment one of the special functions is operating (or start() is waiting for a new tick to be then started by the client terminal) and custom functions are executed only inside special functions.

In this case control is returned to the special function start() that is being executed, i.e. to the line following the function call operator:

35(1). This line contains Alert() call:

```
      Alert ("New tick ",Count,"   Price = ",Price);// Alert
```

The function Alert() will show in a window all constant and variable enumerated in brackets:

```
New tick 1 Price = 1.2744
```

36(1). Operator

```
      return;                                        // Exit start()
```

finishes start() operation.

37. Control is passed to the client terminal waiting for a new tick.

At further start() executions variables will get new values and messages by Alert() will be shown, i.e. the program will perform points 32 - 36. At each start() execution (at each tick) call to the custom function My_Function will be performed and this function will be executed. The execution of start() will continue until a user decides to terminate the program operation. In this case the special function deinit() will be executed and the program will stop operating.

The program userfunction.ex4 started for execution will show a window containing messages by Alert(). Note, the result of the program operation will be the same as the result of a simple Expert Advisor simple.mq4 operation. It is clear that userfunction.mq4 structure is made up in accordance with a usual order of functional blocks location. If another acceptable order is used, the result will be the same.

← Program Execution Operators →

Operators

This section deals with the rules of formatting and execution of operators used in MQL4. Each section includes simple examples that demonstrate the execution of operators. To digest the material in full, it is recommended to compile and launch for execution all exemplary programs. This will also help you to consolidate skills in working with MetaEditor.

- Assignment Operator.
  This is the simplest and the most intuitive operator. We all know the assignment operation from school maths: The name of a variable is located to the left of the equality sign, the value to be assigned to it is to the right of the equality sign.
- Conditional Operator "if-else".
  It is often necessary to guide the program in one or another direction regarding certain conditions. In these cases, the operator "if-else" is very helpful.
- Cycle Operator "while".
  The processing of large one-type data arrays usually requires multiple repetitions of the same operations. You can organize a loop of such operations in the cycle operator "while". Each one execution of operations in a cycle is called *iteration*.
- Cycle Operator "for".
  The operator "for" is also a cycle operator. However, unlike the operator "while", we usually specify in it the initial and the final value of a certain condition for execution of iterations.
- Operator "break".
  If you want to interrupt the working of a cycle operator without execution of the resting iterations, you need the operator "break". It is used only in the operators "while", "for", "switch", nowhere else.
- Operator "continue".
  One more very helpful operator - the operator of going to the next iteration within a cycle. It allows the program to skip all the resting operators in the current iteration and go to the next one.
- Operator "switch".
  This operator is a "toggle" that allows the program to choose one of many possible alternatives. For each alternative, its predefined constant is described that is the case for this alternative.
- Function Call.
  We understand under function call that the function to be called will execute some operations. The function may return a value of the predefined type. The amount of parameters passed into the function may not exceed 64.
- Function Description and Operator "return".
  Before to call a user-defined function, you should describe it first. The function description is specifying its type, name and the list of parameters. Besides, the body of the function contains executable operators. The work of a function is ended in execution of the operator "return".

← Examples of Implementation Assignment Operator →

Assignment Operator

The assignment operator is the simplest and the most commonly used operator.

Format of the Assignment Operator

Assignment operator represents a record that contains character "=" (equality sign). To the left of this equality sign we specify the name of a variable, to the right of it we give an expression. The assignment operator is ended with ";" (semicolon).

```
    Variable = Expression;                          // Assignment operator
```

You can distinguish the assignment operator from other lines in the text of the program by the presence of the equality sign. You can specify as an expression: a constant, a variable, a function call, or an expression as such.

Execution of the Assignment Operator

Calculate the value of the expression to the right from the equality sign and assign the obtained value to the variable specified to the left of the equality sign.

The assignment operator, like any other operator, is executable. This means that the record composing the assignment operator is executed according to the rule. When executing the operator, the value of the right part is calculated and then assigned to the variable to the left of the equality sign. As a result of execution of the assignment operator, the variable in the left part always takes a new value; this value can be other than or the same as the preceding value of the variable. The expression in the right part of the assignment operator is calculated according to the order of operations (see Operations and Expressions).

Examples of Assignment Operators

In an assignment operator, it is allowed to declare the type of a variable to the left of the equality sign:

```
     int In = 3;                 // The constant value is assigned to variable In
     double Do = 2.0;            // The constant value is assigned to variable Do
     bool Bo = true;             // The constant value is assigned to variable Bo
     color Co = 0x008000;        // The constant value is assigned to variable Co
     string St = "sss";          // The constant value is assigned to variable St
     datetime Da= D'01.01.2004';// The constant value is assigned to variable Da
```

The previously declared variables are used in an assignment operator without specifying their types.

```
     In = 7;                    // The constant value is assigned to variable In
     Do = 23.5;                 // The constant value is assigned to variable Do
     Bo = 0;                    // The constant value is assigned to variable Bo
```

In an assignment operator, the type of a variable is not allowed to be declared in the right part of equality sign:

```
     In = int In_2;             // Variable type may not be declared in the right part
     Do = double Do_2;          // Variable type may not be declared in the right part
```

In an assignment operator, the type of a variable is not allowed to be repeatedly declared.

```
     int In;                    // Declaration of the type of variable In
     int In = In_2;             // The repeated declaration of the type of the variable (In) is not allowed
```

Examples of using the user-defined and standard functions in the right part:

```
     In = My_Function();       // The value of user-defined function is assigned to variable In
     Do = Gipo(Do1,Do1);       // The value of user-defined function is assigned to variable Do
     Bo = IsConnected();       // The value of standard function is assigned to variable Bo
     St = ObjectName(0);       // The value of standard function is assigned to variable St
     Da = TimeCurrent();       // The value of standard function is assigned to variable Da
```

The example of using expressions in the right part:

```
     In = (My_Function()+In2)/2;         // The variable In is assigned
                                         // ..with the value of expression
     Do = MathAbs(Do1+Gipo(Do2,5)+2.5); // The variable Do is assigned
                                         // ..with the value of expression
```

In calculations in the assignment operator, the typecasting rules are applicable (see Typecasting).

Examples of Assignment Operators in a Short Form

In MQL4, a short form of composing the assignment operators is used, as well. It is the form of assignment operators where we use assignment operations other than assignment operation "=" (equality sign) (see Operations and Expressions). The short-form operators undergo the same rules and limitations. The short form of the assignment operators is used in the code for better visualization. A programmer may, at his or her option, use one or another form of the assignment operator. Any short-form assignment operator can be easily re-written as a normal, full-format assignment operator, the result of its execution being absolutely unchanged.

```
     In /= 33;                // Short form of the assignment operator
     In = In/33;              // Full form of the assignment operator

     St += "_exp7";           // Short form of the assignment operator
     St = St + "_exp7";       // Full form of the assignment operator
```

← Operators Conditional Operator 'if - else' →

Conditional Operator 'if - else'

As a rule, if you write an application program, you need to code several solutions in one program. To solve these tasks, you can use the conditional operator 'if-else' in your code.

Format of the Operator 'if-else'

Full Format

The full-format operator 'if-else' contains a heading that includes a condition, body 1, the key word 'else', and body 2. The bodies of the operator may consist of one or several operators; the bodies are enclosed in braces.

```
     if ( Condition )                                 // Header of the operator and condition
        {
        Block 1 of operators                         // If the condition is true, then..
        composing body 1                             //..the operators composing body 1 are executed
        }else                                         // If the condition is false..
        {
        Block 2 of operators                        // ..then the operators..
        composing body 2                             // ..of body 2 are executed
        }
```

Format without 'else'

The operator 'if-else' can be used without its 'else' part. In this case, the operator 'if-else' contains its header that includes a condition, and body 1 that consists of one or several operators and enclosed in braces.

```
     if ( Condition )                                // Header of the operator and condition
        {
        Block 1 of operators                         // If the condition is true, then..
        composing body 1                             //..the operators composing body 1 are executed
        }
```

Format without Braces

If the body of the operator 'if-else' consists of only one operator, you can omit the enclosing braces.

```
     if ( Condition )                                 // Header of the operator and condition
        Operator;                                     // If the condition is true, then..
                                                       // ..this operator is executed
```

Execution Rule for the Operator 'if-else'

If the condition of the operator 'if-else' is true, it passes the control to the first operator in body 1. After all operators in body 1 have been executed, it passes control to the operator that follows the operator 'if-else'. If the condition of the operator 'if-else' is false, then:
- if there is the key word 'else' in the operator 'if-else', then it passes the control to the first operator in body 2. After all operators in body 2 have been executed, it passes control to the operator that follows the operator 'if-else';
- if there is no key word 'else' in the operator 'if-else', then it passes the control to the operator that follows the operator 'if-else'.

Execution Examples of the Operator 'if-else'

Let us consider some examples that demonstrate how we can use the operator 'if-else'.

Problem 9. Compose a program where the following conditions are realized: If the price for a symbol has grown and exceeded a certain value, the program must inform the trader about it; if it hasn't exceeded this value, the program mustn't perform any actions.

One of solutions for this problem may be, for example, as follows (onelevel.mq4):

```
//--------------------------------------------------------------------------------------
// onelevel.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------------------
int start()                                           // Special function 'start'
  {
   double
   Level,                                             // Alert level
```

```
     Price;                                         // Current price
     Level=1.2753;                                  // Set the level
     Price=Bid;                                     // Request price
  //--------------------------------------------------------------------------------------
     if (Price>Level)                               // Operator 'if' with a condition
       {
        Alert("The price has exceeded the preset level");// Message to the trader
       }
  //--------------------------------------------------------------------------------------
     return;                                        // Exit start()
    }
  //--------------------------------------------------------------------------------------
```

It should be noted, first of all, that the program is created as an Expert Advisor. This implies that the program will operate for quite a long time in order to display the necessary message on the screen as soon as the price exceeds the preset level. The program has only one special function, start(). At the start of the function, the variables are declared and commented. Then the price level is set in numeric values and the current price is requested.

The operator 'if-else' is used in the following lines of the program:

```
  //--------------------------------------------------------------------------------------
     if (Price>Level)                               // Operator 'if' with a condition
       {
        Alert("The price has exceeded the preset level");// Message to the trader
       }
  //--------------------------------------------------------------------------------------
```

As soon as the control in the executing program is passed to the operator 'if-else', the program will start to test its condition. Please note that the conditional test in the operator 'if-else' is the inherent property of this operator. This test cannot be omitted during execution of the operator 'if-else', it is a "raison d'etre" of this operator and will be performed, in all cases. Thereafter, according to the results of this test, the control will be passed to either the operator's body or outside it - to the operator that follows the closing brace.

In Fig. 37 below, you can see a block diagram representing a possible event sequence at execution of the operator 'if-else'.

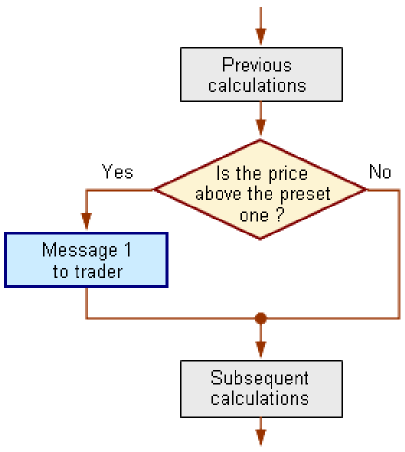


Fig. 37. A block diagram for execution of the operator 'if-else' in program onelevel.mq4.

In this and all following diagrams, a diamond represents the conditional check. Arrows show the target components, to which the control will be passed after the current statement block has been executed (the term of statement block means a certain random set of operators adjacent to each other). Let's consider the diagram in more details.

The block of "Previous Calculations" (a gray box in the diagram) in program onelevel.mq4 includes the following:

```
     double
     Level,                               // Alert level
     Price;                               // Current price
     Level=1.2753;                        // Set the level
     Price=Bid;                           // Request price
```

After the last operator in this block has been executed, the control is passed to the header of the operator 'if-else' where the condition of **Does the price exceed the preset level?** (the diamond box in the diagram, Fig. 37) is checked:

```
     if (Price>Level)                     // Operator 'if' with a condition
```

In other words, we can say that the program at this stage is groping for the answer to the following question: Is the statement enclosed in parentheses true? The statement itself sounds literally as it is written: The value of the variable Price exceeds that of the variable Level (the price exceeds the preset level). By the moment of checking this statement for being true or false, the program has already had the numeric values of variables Price and Level. The answer depends on the ratio between these values. If the price is below the preset level (the value of Price is less or equal to the value of Level), the statement is false; if the price exceeds this level, the statement is true.

Thus, where to pass the control after the conditional test depends on the current market situation (!). **If** the price for a symbol remains below the preset level (the answer is **No**, i.e., the statement is **false**), the control, according to the execution rule of the operator 'if-else', will be passed outside the operator, in this case, to the block named **Subsequent Calculations**, namely, to the line:

```
     return;                                    // Exit start()
```

As is easy to see, no message to the trader is given.

If the price for a symbol exceeds the level preset in the program (the answer is **Yes,** i.e., the statement is **true**), the control will be passed to the body of the operator 'if-else', namely, to the following lines:

```
        {
        Alert("The price has exceeded the preset level");// Message to the trader
        }
```

The execution the function Alert() will result in displaying on the screen a small box containing the following message:

```
The price has exceeded the preset level
```

The function Alert() is the only operator in the body of the operator 'if-else', this is why after its execution, the operator 'if-else' is considered to be completely executed, and the control is passed to the operator that follows the operator 'if-else', i.e., to line:

```
     return;                                    // Exit start()
```

The execution of the operator 'return' results in that the function start() finishes its work, and the program switches to the tick-waiting mode. At a new tick (that also bears a new price for the symbol), the function start() will be executed again. So the message coded in the program will or won't be given to the trader depending on whether the new price exceeds or does not exceed the preset level.

The operators 'if-else' can be nested. In order to demonstrate how the nested operators can be used, let's consider the next example. The problem in it is a bit more sophisticated.

Problem 10. Compose a program where the following conditions are realized: If the price has grown so that it exceeds a certain level 1, the program must inform the trader about it; if the price has fallen so that it becomes lower than a certain level 2, the program must inform the trader about it; however, the program mustn't perform any actions, in any other cases.

It is clear that, in order to solve this problem, we have to check the current price twice:

1. compare the price to level 1, and

2. compare the price to level 2.

Solution 1 of Problem 10

Acting formally, we can compose the following solution algorithm:

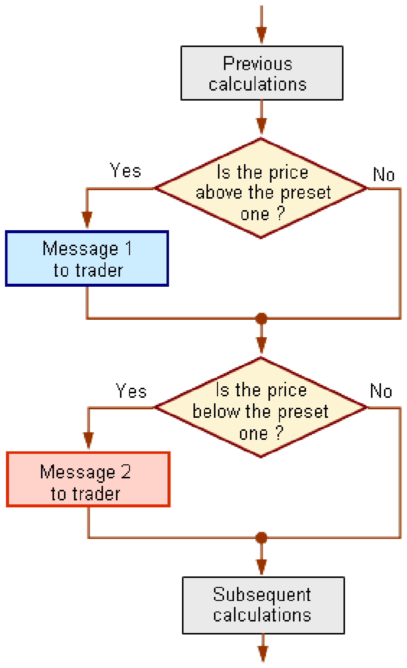


Fig. 38. A block diagram of operators 'if-else' to be executed in program twolevel.mq4.

The program that realizes this algorithm can be as follows (twolevel.mq4):

```
//--------------------------------------------------------------------
// twolevel.mq4
// The code should be used for educational purpose onlyl.
//--------------------------------------------------------------------
int start()                                    // Special function 'start'
  {
   double
   Level_1,                                    // Alert level 1
   Level_2,                                    // Alert level 2
   Price;                                      // Current price
   Level_1=1.2850;                             // Set level 1
   Level_2=1.2800;                             // Set level 2
   Price=Bid;                                  // Request price
//--------------------------------------------------------------------
   if (Price > Level_1)                        // Check level 1
     {
      Alert("The price is above level 1");     // Message to the trader
     }
//--------------------------------------------------------------------
   if (Price < Level_2)                        // Check level 2
     {
      Alert("The price is above level 2");     // Message to the trader
     }
//--------------------------------------------------------------------
   return;                                     // Exit start()
  }
//--------------------------------------------------------------------
```

As is easy to see, the code in program twolevel.mq4 is the extended version of program onelevel.mq4. If we had only one level of calculations before, we have two levels in this new program. Each level was defined numerically, and the program, to solve the stated problem, has two blocks that track the price behavior: Falls the price within the range of values limited by the preset levels or is it outside this range?

Let's give a brief description of the program execution.

After the preparatory calculations have been performed, the control is passed to the first operator 'if-else':

```
//------------------------------------------------------------------------------
   if (Price > Level_1)                        // Checking the first level
     {
      Alert("The price is above level 1");     // Message to the trader
     }
//------------------------------------------------------------------------------
```

Whatever events take place at the execution of this operator (whether a message is given to the trader or not), after its execution the control will be passed to the next operator 'if-else':

```
//------------------------------------------------------------------------------
   if (Price < Level_2)                        // Checking the second level
     {
      Alert("The price is below level 2");     // Message to the trader
     }
//------------------------------------------------------------------------------
```

The consecutive execution of both operators results in the possibility to perform both conditional tests and, finally, to solve the problem. While the program solves the task in full, this solution cannot be considered as absolutely correct. Please note a very important detail: The second conditional test will be executed regardless of the results obtained at testing in the first block. The second block will be executed, even in the case of the price exceeding the first level.

It should be noted here, that one of the important objectives pursued by the programmer when writing his or her programs is algorithm optimization. The above examples are just a demonstration, so they represent a short and, respectively, quickly executed program. Normally, a program intended to be used in practice is much larger. It can process the values of hundreds of variables, use multiple tests, each being executed within a certain amount of time. All this may result in the risk of that the duration of working of the special function start() may exceed the tick gaps. This will mean that some ticks may remain unprocessed. This is, of course, an extremely undesirable situation, and the programmer must do his or her best in order to prevent it. One of the techniques allowing to to reduce the time of the program execution is algorithm optimization.

Let's consider the latest example once again, this time in terms of resources saving. For example, what is the reason for asking "Is the price below the preset level?" in the second block, if the testing of the first block has already detected that the price is above the upper level? It is clear that, in this case, the price cannot be below the lower level, so there is no need to make a conditional test in the second block (nor to execute the set of operators in this block).

Solution 2 of Problem 10

Taking the above reasoning into account, let's consider the following, optimized algorithm:

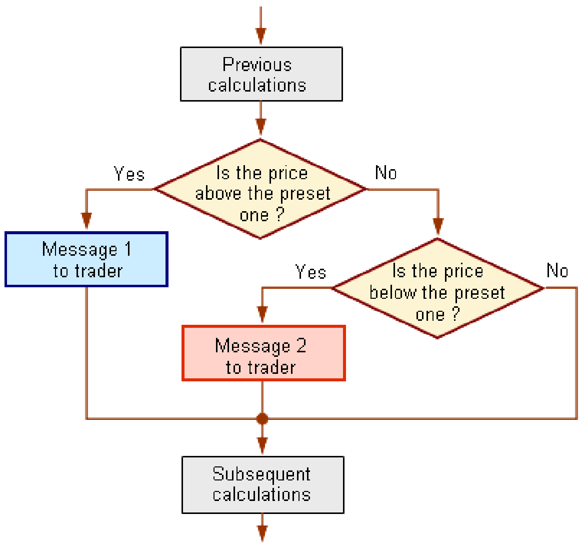


Fig. 39. A Block diagram for execution of the operator 'if-else' in program twoleveloptim.mq4.

According the algorithm shown in the above block diagram, no redundant operations will be executed in the program. After the preparatory calculations, the program will test whether **The price is above the preset level**. If yes, the program displays the corresponding message to the trader and passes the control to **Subsequent Calculations**, whereas the second condition (**Is the price below the preset level**?) is not tested. Only if the price hasn't turned out to be above the upper level (the answer is **No**), the control is passed to the second block where the necessary conditional test is performed. If the price turns out to be below the lower level, the corresponding message will be displayed to the trader. If not, the control will be passed to the further calculations. It is obvious that, if the program works according this algorithm, no redundant actions are performed, which results in considerable saving of resources.

The programmed implementation of this algorithm implies the use of the nested operator 'if-else' (twoleveloptim.mq4):

```
//--------------------------------------------------------------------
// twoleveloptim.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int start()                                   // Special function start()
  {
   double
   Level_1,                                   // Alert level 1
   Level_2,                                   // Alert level 2
   Price;                                     // Current price
   Level_1=1.2850;                            // Set level 1
   Level_2=1.2800;                            // Set level 2
   Price=Bid;                                 // Request price
//--------------------------------------------------------------------
   if (Price > Level_1)                       // Check level 1
     {
      Alert("The price is above level 1");    // Message to the trader
     }
   else
     {
      if (Price < Level_2)                    // Check level 2
        {
         Alert("The price is above level 2"); // Message to the trader
        }
     }
//--------------------------------------------------------------------
   return;                                    // Exit start()
  }
//--------------------------------------------------------------------
```

Please note this block of calculations:

```
//--------------------------------------------------------------------

   if (Price > Level_1)                       // Check level 1
     {
      Alert("The price is above level 1");    // Message to the trader
     }
   else
     {
      if (Price < Level_2)                    // Check level 2
        {
         Alert("The price is above level 2"); // Message to the trader
        }
     }
```

```
//--------------------------------------------------------------------
```

The operator 'if-else', in which the second condition is tested, is a component of the first operator 'if-else' that tests the first condition. Nested operators are widely used in the programming practice. This is often reasonable and, in some cases, the only possible solution. Of course, instead of the function Alert(), real programs can use other functions or operators performing various useful actions.

A complex expression can also be used as a condition in the operator 'if-else'.

Problem 11. Compose a program that realizes the following conditions: If the price falls within the preset range of values, no actions should be performed; if the price is outside this range, the program must inform the trader about it.

The statement of this problem is similar to that of Problem 10. The difference consists in that, in this case, we are not interested in the direction of the price movement - higher or lower than the predefined range. According to the problem situation, we have to know only the fact itself: Is the price within or outside the range? We can use the same text for the message to be displayed.

Here, like in the preceding solutions, it is necessary to check the price status as related to two levels - the upper one and the lower one. We have rejected the solution algorithm shown in Fig. 38 as non-efficient. So, according to the above reasoning, we can propose the following solution:

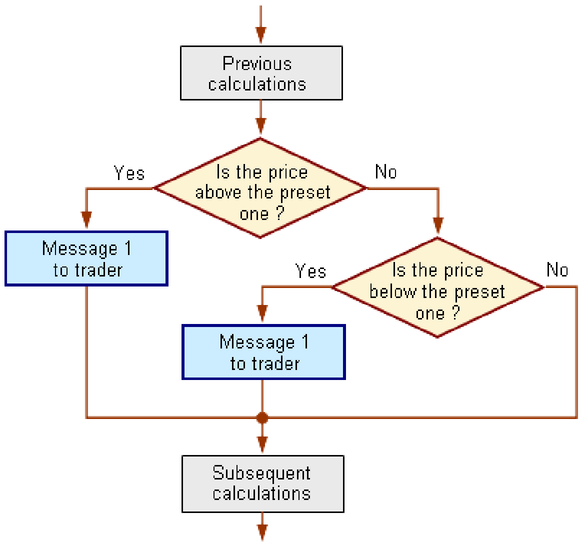


Fig. 40. The block diagram of one of the solutions of problem 11.

However, there is no need to use this algorithm. The diagram demonstrates that the algorithm implies the usage of the same block of messages at different points in the program. The program lines will be repeated, notwithstanding that the execution of them will result in producing the same messages. In this case, it would be much more efficient to use the only one operator 'if-else' with a complex condition:

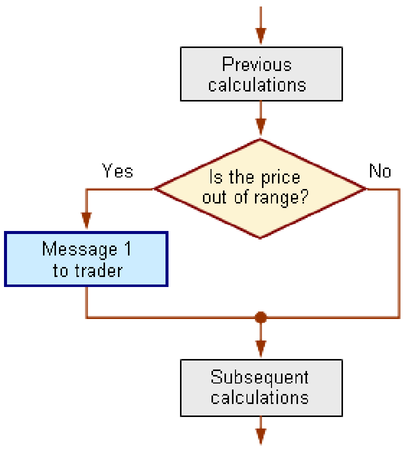


Fig. 41. A block diagram for execution of the operator 'if-else' in program compoundcondition.mq4.

The code of the program that implements this algorithm is as follows (compoundcondition.mq4):

```
//--------------------------------------------------------------------
// compoundcondition.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int start()                                      // Special function 'start'
  {
   double
```

```
     Level_1,                                   // Alert level 1
     Level_2,                                   // Alert level 2
     Price;                                     // Current price
     Level_1=1.2850;                            // Set level 1
     Level_2=1.2800;                            // Set level 2
     Price=Bid;                                 // Request price
  //--------------------------------------------------------------------------------
     if (Price>Level_1 || Price<Level_2)       // Test the complex condition
       {
        Alert("The price is outside the preset range");// Message
       }
  //--------------------------------------------------------------------------------
     return;                                   // Exit start()
    }
  //--------------------------------------------------------------------------------
```

The key insight of this program solution is the use of complex condition in the operator 'if-else':

```
  if (Price>Level_1 || Price<Level_2)              // Test the complex condition
    {
     Alert("The price is outside the preset range");// Message
    }
```

Simply stated, this condition is as follows: If the value of variable Price exceeds that of variable Level_1, **or** the value of variable Price is less than that of variable Level_2, the program must execute the body of the operator 'if-else'. It is allowed to use logical operations (**&&**, **||** and **!**) when composing the conditions of the operator 'if-else', they are widely used in the programming practice (see Boolean (Logical) Operations).

When solving other problems, you may need to compose even more complex conditions, which makes no question, too. Some expressions, including the nested ones, can be enclosed in parentheses. For example, you may use the following expression as a condition in the operator 'if-else':

```
     if ( A>B && (B<=C || (N!=K && F>B+3)) )      // Example of a complex condition
```

Thus, MQL4 opens up great opportunities to use the operators 'if-else': they can be nested, they can contain nested structures, they can use simple and complex test conditions, which results in the possibility to compose simple and complex programs with branching algorithms.

← Assignment Operator   Cycle Operator 'while' →

Cycle Operator 'while'

The most powerful functionality of MQL4 is the possibility to organize cycles (loops).

When creating application programs, you may often use the repeated calculations, which are mostly the repeated program lines. In order to make the programming comfortable and the program itself user-friendly, we use cycle operators. There are two cycle operators in MQL4: while and for. We will consider the first one in this section.

Format of the Operator 'while'

The full-format cycle operator 'while' consists of the header containing a condition, and the executable cycle body enclosed in braces.

```
while ( Condition )                            // Header of the cycle operator
   {                                          // Opening brace
   Block of operators                         // The body of a cycle operator may consist..
   that compose the cycle body                //.. of several operators
   }                                          // Closing brace
```

If the cycle body consists of only one operator in the operator 'while', you can omit braces.

```
while ( Condition )                            // Header of the cycle operator
   One operator, the cycle body                // Cycle body is one operator
```

Execution Rule for the Operator 'while'

As long as the Condition of the operator 'while' is true: The program passes the control to the operator of the cycle body; after all operators in the cycle body have been executed, it passes the control to the header to test the truth of the Condition.

If the Condition of the operator 'while' is false, the control must be passed to the operator that follows the cycle operator 'while'.

Let's consider an example.

Problem 12. Calculate the Fibonacci coefficient with the accuracy to 10 significant figures.

First, let's briefly describe Fibonacci coefficient. The Italian mathematician, Leonardo Fibonacci, discovered a unique sequence of numbers:

1 1 2 3 5 8 13 21 34 55 89 144 233 ...

Each number in this sequence is the sum of two preceding numbers. This number sequence has some unique properties: The ratio of two successive numbers in the sequence is equal to 1.618, whereas the ratio between a number to the preceding number is equal to 0.618. Ratio 1.618 was named after Fibonacci, as well as the above sequence is named Fibonacci sequence (it should also be noted that 0.3819, a conjugate for Fibonacci number, was obtained by its multiplying by itself: 0.3819 = 0.618 x 0.618).

The task is to calculate the Fibonacci number with a higher accuracy. If we analyze the Fibonacci coefficient for several tens of the elements of Fibonacci sequence, it becomes obvious that the obtained coefficients range around the irrational number of 1.61803398875..., taking larger or smaller values by turns. The larger numbers of the sequence are involved in the calculations, the smaller is the deviation of the result from this value.

We don't know in advance what exact numbers we should take for their coefficients to differ from each other only after the tenth significant figure. So it is necessary to compose a program that would consecutively search in coefficients until the difference between them makes less than 0.0000000001.

In this case, we created a script (fibonacci.mq4) to solve Problem 12, because the program will not be used for a long time to check every tick. Once attached to the symbol window, the script must perform all necessary calculations (including displayed results), after that it will be unloaded from the window by the client terminal.

```
//--------------------------------------------------------------------------------
// fibonacci.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------------
int start()                            // Special function start()
  {
//--------------------------------------------------------------------------------
   int i;                              // Formal parameter, counter
   double
   A,B,C,                              // Numbers in the sequence
   Delta,                              // Real difference between coefficients
   D;                                  // Preset accuracy
//--------------------------------------------------------------------------------
   A=1;                                // Initial value
   B=1;                                // Initial value
   C=2;                                // Initial value
   D=0.0000000001;                     // Set accuracy
   Delta=1000.0;                          // Initial value
//--------------------------------------------------------------------------------------------
   while(Delta > D)                       // Cycle operator header
     {                                    // Opening brace of the cycle body
      i++;                                // Counter
      A=B;                                // Next value
      B=C;                                // Next value
      C=A + B;                            // Next value
      Delta=MathAbs(C/B - B/A);           // Search difference between coefficients
     }                                    // Closing brace of the cycle body
//--------------------------------------------------------------------------------------------
   Alert("C=",C," Fibonacci number=",C/B," i=",i);//Display on the screen
   return;                                // Exit start()
  }
//--------------------------------------------------------------------------------------------
```

At the beginning of the program, variables are declared (and described). In the subsequent lines, numeric values are assigned to the variables. A, B and C take the value of the first numbers in Fibonacci sequence. It must be noted here that, while the sequence itself contains integers only, the quotient of their division must be considered in the program as a real number. If we used the *int* type for these numbers here, it would be impossible to calculate Fibonacci coefficient, for example: 8/5 = 1 (integer 1, without the fractional part). So, in this case, we use the variables of *double* type. The cycle operator as such looks like this:

```
   while(Delta > D)                       // Cycle operator header
     {                                    // Opening brace of the cycle body
      i++;                                // Counter
      A=B;                                // Next value
      B=C;                                // Next value
      C=A + B;                            // Next value
      Delta=MathAbs(C/B - B/A);           // Search difference between coefficients
     }                                    // Closing brace of the cycle body
```

Let's consider a block diagram of the cycle operator 'while':

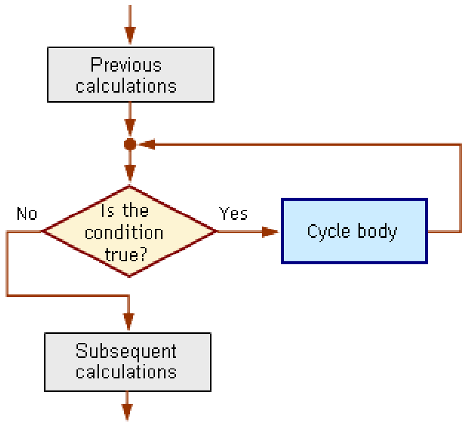


Fig. 42. Block diagram of the operator-'while' execution in program fibonacci.mq4.

The cycle operator starts working with testing the condition. The cycle will be performed repeatedly, until the condition (Delta>D) is true. You will understand the program code much better, if you say the key phrase (the execution rule) aloud while reading the operators. For example, if you read the operator 'while' it is: "**As long as .., perform the following: ..**". In this case, the header of the operator 'while' is as follows: As long as Delta is greater than D, perform the following... Then you can go to analysis of program lines composing the cycle body and being executed if this Condition is true.

Before the control in the above example of fibonacci.mq4 is passed to the cycle operator 'while', the values of these variables are equal to 1000.0 and 0.0000000001, respectively; so the condition is true at the first call to the cycle operator. This means that, after the condition has been tested, the control will be passed to the first operator in the cycle operator body.

In our example, it is the following operator:

```
      i++;                                // Counter
```

In the three subsequent lines, the values of the next set of sequence elements will be calculated:

```
      A = B;                              // Next value
      B = C;                              // Next value
      C = A + B;                          // Next value
```

It's easy to see that the variables take the values of the next (nearest larger) elements. Before the cycle operator is executed, the values of A, B and C have been equal to 1.0, 1.0 and 2.0, respectively. During the first iteration, these variables take the values of 1.0, 2.0 and 3.0, respectively.

Iteration is a repeated execution of some calculations; it is used to note that the program lines composing the cycle operator body are executed.

In the next line, we calculate the difference between Fibonacci numbers obtained on the basis of subsequent (C/B) and preceding (B/A) elements of the sequence:

```
      Delta = MathAbs(C/B - B/A);          // Search difference between coefficients
```

In this operator, we use the standard function MathAbs() that calculates the absolute value of the expression. As we mentioned above, Fibonacci coefficients take, in turns, larger and smaller values (as compared to the 'standard') with increase of the values of the sequence elements. This is why the difference between neighboring coefficients will take now a negative, now a positive value. At the same time, we are really interested in the value itself of this deviation, i.e., its absolute value. Thus, whatever is the direction the current value of Fibonacci number is deviated in, the value of expression MathAbs(C/B - B/A), as well as the value of variable Delta, is always positive.

This operator is the last in the list of operators composing the cycle body. This is confirmed by the presence of a closing brace in the next line. After the last operator in the cycle body has been executed, the control is passed to the cycle operator header for testing the condition. This is the key point that determines the essence of the cycle operator. According as the cycle operator condition is true or false, the control will be passed to either the next iteration oro outside the cycle operator.

At the first iterations, the value of variable Delta turns out to be greater than the value defined in the variable D. This means that the condition (Delta > D) is true, so the control will be passed to the cycle body for it to perform the next iteration. All variables involved in the calculations will take new values: as soon as the end of the cycle body is reached, the control will be passed to the header again in order to test the condition for being true.

This process will continue, until the condition of the cycle operator becomes false. As soon as the value of the variable Delta becomes less or equal to the value of D, the condition (Delta > D) is not true any longer. This means that the control will be passed outside the cycle operator, to the line:

```
   Alert("C=",C," Fibonacci number=",C/B," i=",i);//Display on the screen
```

As a result, the Alert() operator window containing the following message will appear on the screen:

```
C=317811 Fibonacci number=1.618 i=25
```

This means that the search accuracy is reached at the 25th iteration, the maximum value of Fibonacci sequence element processed being equal to 317811. Fibonacci coefficient, as expected, is equal to 1.618. This message is the solution of the stated problem.

It must be noted here that real numbers are calculated in MQL4 with an accuracy to 15 significant figures. At the same time, Fibonacci coefficient is displayed with an accuracy to 3 significant figures. This is determined by the fact that the function Alert() displays numbers with an accuracy to 4 significant figures, without displaying the zeros at the end of the number. If we wanted to display Fibonacci number with a certain predefined accuracy, we would have to change the code, for example, in this way:

```
   Alert("C=",C," Fibonacci number=",C/B*10000000," i=",i);// Message
```

The following must be particularly noted:

It is possible that the condition specified in the cycle operator header always remains true. This results in looping.

Looping is a continuously repeated execution of operators composing the cycle body; it's a critical situation that results from realization of a wrong algorithm.

Once a looping takes place, the program endlessly executes the block of operators composing the cycle body. Below is a simple example of a looped cycle operator 'while':

```
   int i=1;                               // Formal parameter (counter)
   while (i > 0)                          // Cycle operator header
      i++;                                // Increment of the value of i
```

In the above example, the values of variable i are accumulated (incremented) at every iteration. As a result, the condition will never become false. A similar situation occurs, if nothing is calculated in the cycle body at all. For example, in the code below:

```
   int i=1;                               // Formal parameter (counter)
   while (i > 0)                          // Cycle operator header
      Alert("i= ",i);                     // Display on the screen
```

the value of variable i in the cycle does not change. The following message will be shown in the screen at each iteration:

```
i= 1
```

This process will repeat endlessly. Once having got into the trap of a looped code, the control cannot leave it anymore. This situation is particularly dangerous in trading Expert Advisors and scripts. In such cases, the environment variables are not usually updated, since the special function does not complete its operation, whereas the trader may be unaware of the existing looping. As a result, the control in the executing program cannot be passed to the corresponding program lines where the decision of opening or closing orders is made.

The programmer must prevent such conditions, in which an uncontrolled looping becomes possible. There is no technique that would help to discover this situation programmatically neither at compiling of the program nor at its execution. The only possible method to detect such algorithmic errors is close examination of your code - logical reasoning and common sense.

← Conditional Operator 'if - else' Cycle Operator 'for' →

Cycle Operator 'for'

Another cycle operator is the operator 'for'.

Format of the Operator 'for'

The full-format cycle operator 'for' consists of the header that contains Expression_1, Condition and Expression_2, and of the executable cycle body enclosed in braces.

```
   for (Expression_1; Condition; Expression_2)         // Cycle operator header
       {                                              // Opening brace
       Block of operators                             // Cycle body may consist ..
       composing the cycle body                       //.. of several operators
       }                                              // Closing brace
```

If the cycle body in the operator 'for' consists of only one operator, braces can be omitted.

```
   for (Expression_1; Condition; Expression_2)         // Cycle operator header
       One operator, cycle body                       // Cycle body is one operator
```

Expression_1, Condition and/or Expression_2 can be absent. In any case, the separating character ";" (semicolon) must remain in the code.

```
   for ( ; Condition; Expression_2)                   // No Expression_1
       {                                              // Opening brace
       Block of operators                             // Cycle body may consist ..
       composing the cycle body                       //.. of several operators
       }                                              // Closing brace
// - ---------------------------------------------------------------------------------
   for (Expression_1; ; Expression_2)                 // No Condition
       {                                              // Opening brace
       Block of operators                             // Cycle body may consist ..
       composing the cycle body                       //.. of several operators
       }                                              // Closing brace
// - ---------------------------------------------------------------------------------
   for ( ; ; )                                        // No Expressions or Condition
       {                                              // Opening brace
       Block of operators                             // Cycle body may consist ..
       composing the cycle body                       //.. of several operators
       }                                              // Closing brace
```

Execution Rule of the Operator 'for'

As soon as the control is passed to the operator 'for', the program executes Expression_1. As long as the Condition of the operator 'for' is true: pass the control to the first operator in the cycle body; as soon as all operators in the cycle body are executed, the program must execute Expression_2 and pass the control to the header for testing the Condition for being true. If the Condition of the operator 'for' is false, then: the program must pass the control to the operator that follows the operator 'for'.

Let's consider how the cycle operator 'for' works. For this, let's solve a problem.

Problem 13. We have a sequence of integers: 1 2 3 4 5 6 7 8 9 10 11 ... Compose a program that would calculate the sum of the elements of this sequence starting from N1 and finishing at N2.

This problem is easy to solve in terms of mathematics. Suppose we want to calculate the sum of elements from the third through the seventh one. The solution will be: 3 + 4 + 5 + 6 + 7 = 25. However, this solution is good only for a particular case, when the numbers of the first and the last elements composing the sum are equal to 3 and 7, respectively. A program solving this problem must be composed in such a way that, if we want to calculate the sum within another interval of the sequence (for example, from the 15th to 23rd element), we could easily replace the numeric values of the elements in a location in the program without modifying program lines in other locations. Below is one of the versions of such a program (script sumtotal.mq4):

```
//--------------------------------------------------------------------------------
// sumtotal.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------------
int start()                          // Special function start()
  {
//--------------------------------------------------------------------------------
   int
   Nom_1,                               // Number of the first element
   Nom_2,                               // Number of the second element
   Sum,                                 // Sum of the numbers
   i;                                   // Formal parameter (counter)
//--------------------------------------------------------------------------------
   Nom_1=3;                              // Specify numeric value
   Nom_2=7;                              // Specify numeric value
   for(i=Nom_1; i<=Nom_2; i++)           // Cycle operator header
     {                                   // Brace opening the cycle body
      Sum=Sum + i;                       // Sum is accumulated
      Alert("i=",i,"  Sum=",Sum);        // Display on the screen
     }                                   // Brace closing the cycle body
//---------------------------------------------------------------------------------
   Alert("After exiting the cycle, i=",i,"  Sum=",Sum);// Display on the screen
   return;                               // Exit start()
  }
//---------------------------------------------------------------------------------
```

In the first lines of the program, variables are declared (the comments explain the sense of each variable). In the lines

```
   Nom_1 = 3;                           // Specify numeric value
   Nom_2 = 7;                           // Specify numeric value
```

the numeric values of the first and of the last element of the range are defined. Please note that these values aren't specified anywhere else in the program. If necessary, you can easily change these values (in one location) without changing the code in other lines, so that the problem can be solved for another range. After the last of these operators has been executed, the control is passed to the cycle operator 'for':

```
   for(i=Nom_1; i<=Nom_2; i++)          // Cycle operator header
     {                                  // Brace opening the cycle body
      Sum = Sum + i;                    // Sum is accumulated
      Alert("i=",i,"  Sum=",Sum);       // Display on the screen
     }                                  // Brace closing the cycle body
```

The key phrase - the execution rule of the operator 'for' - is as follows: **"Starting from .., as long as .., step .., perform the following: ..".** The use of 'step..' is applicable, if we use a storage counter, for example, i++ или i=i+2, as Expression_2. In our example, the key phrase is as follows: Starting from i that is equal to Nom_1, as long as i is less or equal to Nom_2, step 1, perform the following: (and then - the analysis of the cycle body).

According to the execution rule of the operator 'for' the following will be executed in this block of the program:

1. Execution of Expression_1:

```
   i=Nom_1
```

Variable i will take the numeric value of variable Nom_1, i.e., integer 3. Expression_1 is executed only once - when the control is passed to the operator 'for'. Expression_1 is not involved in any subsequent events.

2. The Condition is tested:

```
   i<=Nom_2
```

The value of variable Nom_2 at the first iteration is integer 7, whereas the value of variable i is equal to 3. This means that the Condition is true (since 3 is less than 7), i.e., the control will be passed to the cycle body to execute it.

3. In our example, the cycle body consists of two operators that will be executed one by one:

```
   Sum = Sum + i;                    // Sum is accumulated
   Alert("i=",i,"  Sum=",Sum);       // Display on the screen
```

Variable Sum, at its initialization, did not take any initial value, so its value is equal to zero before the first iteration starts. During calculations, the value of variable Sum is increased by the value of i, so it is equal to 3 at the end of the first iteration. You can see this value in the box of the function Alert():

```
i=3 Sum=3
```

4. The last event that happens during execution of the cycle operator is the execution of Expression_2:

```
    i++
```

The value of variable i is increased by one. This is the end of the first iteration, the control is passed to test the Condition.

Then the second iteration starts, steps 2-4. During calculations, the variables take new values. Particularly, at the second iteration, the value of variable i is equal to 4, whereas Sum is equal to 7. The corresponding message will appear, too. Generally, the program will develop according to the block diagram below:

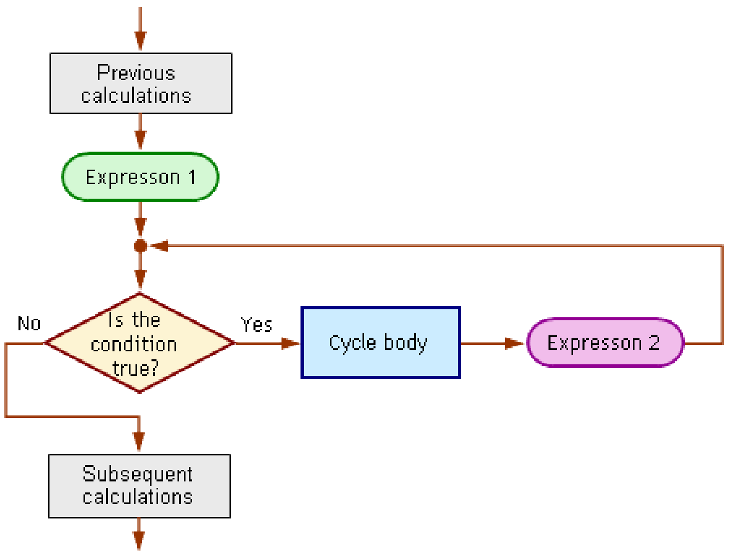


Fig. 43. Block diagram of the operator-'for' execution in program sumtotal.mq4.

The program will cyclically repeat the execution of the cycle operator, until the Condition becomes false. This will happen, as soon as the value of variable i is equal to 8, i.e., exceeds the preset value 7 of variable Nom_2. In this case, the control will be passed outside the operator 'for', namely, to the line:

```
    Alert("After exiting the cycle, i=",i,"  Sum=",Sum);// Display on the screen
```

and further, to execution of the operator 'return' that ends the operation of the special function start().

You can use this program to calculate the sum of any other range of values. For example, if you replace the constants 3 and 7 with 10 and 15, respectively, the execution of the program will result in calculation of the sum of values within this range.

In some programming languages, the variables specified in the cycle operator header lose their values after exiting the cycle. This is not the case in MQL4. All variables that are involved in any calculations are valid and keep their values after exiting the cycle. At the same time, there are some particularities concerning the values of variables that are involved in calculations.

Please note the last two messaging lines that remain after all calculations are complete and the script is unloaded:

```
i=7 Sum=25
After exiting the cycle, i=8 Sum=25
```

The former of the above two messages had appeared at the stage of the last iteration, before the cycle was complete, whereas the latter of them had been given by the last function Alert() before the program was terminated. It is remarkable that these lines display different values of variable i. During the cycle execution, this value is equal to 7, but it is equal to 8 after exiting the cycle. In order to understand, why it happens this way, let's refer to Fig. 43 once again.

The last (in this case, the fifth) iteration starts with testing the Condition. The value of i is equal to 7 at this moment. Since 7 does not exceed the value of variable Nom_2, the Condition is true and the cycle body will be executed. As soon as operators composing the cycle body are executed, the last operation is performed: Expression_2 is executed. This must be emphasized once again:

The last event taking place at each iteration of the operator-'for' execution is the calculation of Expression_2.

This results in increasing of the value of variable i by 1, so that this value becomes equal to 8. The subsequent test of the Condition (at the beginning of the next iteration) confirms that the Condition is false now. This is why the control will be passed outside the cycle operator after testing. At the same time, variable i exits the cycle operator with the value 8, whereas the value of variable Sum remain to be equal to 25, because the cycle body is not executed at this stage. This must be considered in situations where the amount of iteration is not known in advance, so that the programmer is going to estimate it by the value of the variable calculated in Expression_2.

We can consider a situation that produces looping to be an exceptional case. This becomes possible at various errors. For example, if you use the following operator as Expression_2 by error:

```
    i--
```

the value of the counter at each iteration will decrease, so that the Condition will never become false.

Another error can be an incorrectly composed Condition:

```
  for(i=Nom_1; i>=Nom_1; i++)              // Cycle operator header
```

In this case, the value of variable i will always exceed or be equal to the value of variable Nom_1, so that the Condition is always true.

Interchangeability of Cycle Operators 'while' and 'for'

The use of one operator or another fully depends on the decision made by the programmer. The only thing we will note is that these operators are interchangeable; it is often necessary just to give another brush to the code of your program.

For example, we used the cycle operator 'for' to solve Problem 13:

```
    for (i=Nom_1; i<=Nom_2; i++)          // Cycle operator header
      {                                   // Brace opening the cycle body
       Sum = Sum + i;                     // Sum is accumulated
       Alert("i=",i,"  Sum=",Sum);        // Display on the screen
      }                                   // Brace closing the cycle body
```

Below is an example how the same fragment of the code may look if we use the operator 'while':

```
    i=Nom_1;                              // Оператор присваивания
    while (i<=Nom_2)                      // Cycle operator header
      {                                   // Brace opening the cycle body
       Sum = Sum + i;                     // Sum is accumulated
       Alert("i=",i,"  Sum=",Sum);        // Display on the screen
       i++;                               // Increment of the element number
      }                                   // Brace closing the cycle body
```

As is easy to see, it's sufficient just to move Expression_1 into the line that precedes the cycle operator, whereas Expression_2 should be specified as the last in the cycle body. You can change the exemplary program yourself and launch it for execution, in order to see that the results are the same for both versions.

← Cycle Operator 'while' Operator 'break' →

Operator 'break'

In some cases, for example, when programming some cycle operations, it could become necessary to break the execution of a cycle before its condition becomes false. In order to solve such problems, you should use the operator 'break'.

Format of the Operator 'break'

The operator 'break' consists of only one word and ends in character ";" (semicolon).

```
      break;                                         // Operator 'break'
```

Execution Rule of the Operator 'break'

The operator 'break' stops the execution of the nearest external operator of 'while', 'for' or 'switch' type. The execution of the operator 'break' consists in passing the control outside the compound operator of 'while', 'for' or 'switch' type to the nearest following operator. The operator 'break' can be used only for interruption of the execution of the operators listed above.

We can visualize the execution of the operator 'break' by the following example.

Problem 14. Let's have a thread, 1 meter long. It is required to lay the thread in form of a rectangle with the maximum possible area. It is also required to find the area of this rectangle and the length of the sides with an accuracy to 1 mm by serial search in variations.

There can be an unlimited amount of rectangles of different sizes made of a unitary block of thread. Since the accuracy required by the problem statement makes 1 mm, we can only consider 499 variations. The first and the "thinnest" rectangle will have the dimensions of 1x499 mm, the second one will be 2x498 mm, etc., whereas the dimensions of the last rectangle will be 499x1 mm. We have to search in all these rectangles and find the one with the largest area.

As will readily be observed, there are repeated dimensions in the set of rectangles we are considering. For example, the first and the last rectangles have the same dimensions: 1x499 mm (the same as 499x1 mm). Similarly, the dimensions of the second rectangle will be the same as those of the last but one rectangle, etc. We have to make an algorithm that would search in all unique variations, whereas there is no need to search in the repeated ones.

First of all, let's estimate the relationship between the area and the length of the sides in a rectangle. As is easy to see, the first rectangle, with the dimensions of 1x499, has the smallest area. Then, with increase of the shorter side, the area of the rectangle will increase, too. As soon as they reach a certain value, the areas of the rectangles will start to decrease again. This relationship is shown in Fig. 44 below:

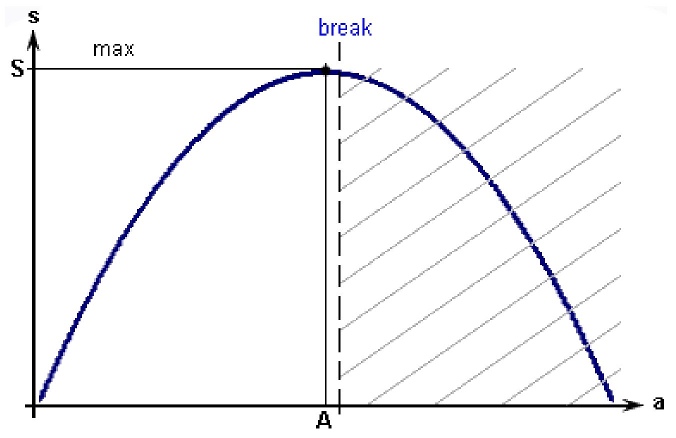


Fig. 44.Relationship between the rectangle area and the length of one of its sides.

Looking at Fig. 44, we can easily come to the conclusion that we should find the maximum area by searching in variations starting from the first one, only as long as the area increases during calculations. As soon as it starts decreasing, we will break our search and exit the search cycle. Below is script rectangle.mq4 that implements such an algorithm.

```
  //--------------------------------------------------------------------
```

```
// rectangle.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int start()                           // Special function start()
  {
//--------------------------------------------------------------------
   int
   L=1000,                            // Specified thread length
   A,                                 // First side of the rectangle
   B,                                 // Second side of the rectangle
   S,                                 // Area of the rectangle
   a,b,s;                             // Current values
//--------------------------------------------------------------------
   for(a=1; a<L/2; a++)               // Cycle operator header
     {                                // Brace opening the cycle body
      b=(L/2) - a;                    // Current value of the sides
      s=a * b;                        // Current value of the area
      if (s<=S)                       // Choose the larger value
         break;                       // Exit the cycle
      A=a;                            // Save the best value
      B=b;                            // Save the best value
      S=s;                            // Save the best value
     }                                // Brace closing the cycle body
//--------------------------------------------------------------------
   Alert("The maximum area = ",S,"  A=",A,"  B=",B);// Message
   return;                               // Function exiting operator
  }
//--------------------------------------------------------------------
```

Let's watch how this program works. Variables are declared and commented on at the beginning of the program. The algorithm of problem solving itself is realized within the cycle 'for'. The initial value of the side **a** of the rectangle is specified as equal to 1 in Expression_1. According to the Condition, the values are searched in, as long as the size of the rectangle side **a** remains smaller than a half of the thread length. Expression_2 prescribes to increase the side length **a** of the current rectangle at each iteration step.

Variables a, b and s are the current variables, the values of which are searched in. Variables A, B and S are the search values. The second side **b** and the area **s** of the rectangle are calculated at the beginning of the cycle.

```
      b = (L/2) - a;                     // Current values of the sides
      s = a * b;                         // Current value of the area
```

The condition of exiting the cycle is tested in the operator 'if':

```
      if (s <= S )                       // Choose the larger value
         break;                          // Exit the cycle
```

If the newly calculated area **s** of the current rectangle turns out to be larger than the area **S** calculated at the preceding iteration, this new value of **s** becomes the best possible result. In this case, the Condition of the operator 'if' is not met, and the control is passed to the nearest operator that follows the operator 'if'. Below are the lines where we save the best results:

```
      A = a;                             // Save the best value
      B = b;                             // Save the best value
      S = s;                             // Save the best value
```

As soon as the program reaches the nearest closing brace, iteration is finished and the control is passed to the header of the operator 'for' to execute Expression_2 and test the Condition. If the length of the side **a** has not reached the specified limit as of the moment of testing, the cycle will continue to be executed.

The repeated cyclic calculations will continue until one of the following events takes place: either the length of the side **a** exceeds the predefined limits (according to the Condition of the operator 'for'), or the size of the calculated area **s** turns out to be smaller than a previously found value stored in variable **S**. There is a good reason to believe that the loop exit according to the condition of the operator 'if' will take place earlier:

```
     if (s <= S )                       // Choose the larger value
         break;                         // Exit the cycle
```

Indeed, the cycle operator 'for' is composed in such a way that it searches in all possible variations without any exceptions (the half of the thread length, L/2, is the sum of two sides of the rectangle). At the same time, the maximum area of the rectangle will be reached somewhere in the middle of the searched set of variations. So, as soon as this happens (the area of the current rectangle **s** turns out to be less or equal to the previously reached value **S**), the control within the execution of the operator 'if' will be passed to the operator 'break' that, in its turn, will pass the control outside the operator 'for', to the following line:

```
      Alert("The maximum area = ",S,"  A=",A,"  B=",B);// Message
```

As a result of execution of the standard function Alert(), the following line will be printed:

```
The maximum area = 62500 A=250 B=250
```

After that, the control will be passed to the operator 'return', which results in termination of the special function start(). This, in its turn, results in termination of the script and its being unloaded by the client terminal from the symbol window.

In this example, the operator 'break' stops (passes the control outside) the cycle operator 'for', namely, the cycle operator it is located in. Below is the block diagram of the cycle 'for' with a special exit:

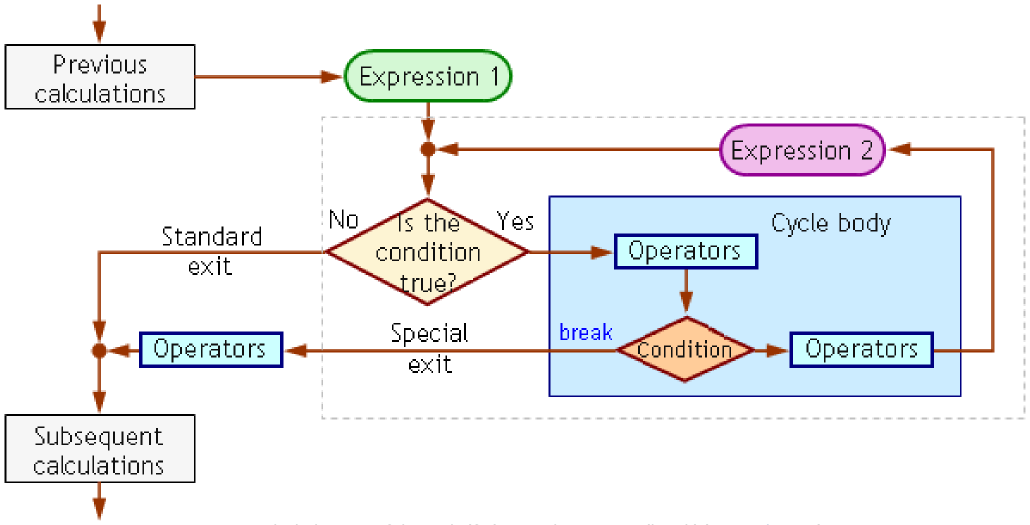


Fig. 45. Block diagram of the cycle 'for' using the operator 'break' (rectangle.mq4).

As is seen from the diagram, there are two cycle exits: a normal exit (cycle exit resulting from triggering of the condition specified in the cycle operator header) and a special exit (the cycle body exit according to an additional condition and using the operator 'break').

It is difficult to overestimate the possibility to use a special exit in a cycle. In our example, the operator 'break' allows us to make an algorithm that performs only necessary calculations. An algorithm without a special exit would be inefficient: in this case, the repeated calculations would be performed, which would result in unreasoned waste of time and computational resources. The crosshatched region in Fig. 44 visualizes the region of parameters that were not processed in the above program (almost one half of all calculations!), which did not prevent us from obtaining the correct result.

The necessity to use effective algorithms becomes especially obvious, when the program execution time stretches to seconds and even minutes, which can exceed the time interval between ticks. In this case, there is a risk to omit processing of some informative ticks and, as a result, to lose control of your trading. Besides, you can feel the reduced time taken by calculations especially deeply if you test your programs in the Strategy Tester. Testing sophisticated programs on a long history can take hours and even days. The halved time taken by tests is a very important feature, in this case.

The Rule states that the operator 'break' stops the nearest external operator. Let's see how a program works that uses nested cycles.

Problem 15. Using the algorithm of Problem 14, search the shortest thread multiple of 1 meter and sufficient to form a rectangle with an area of 1.5 m².

In this problem, we should linearly search in the available thread lengths and calculate the maximum area for each thread length. The solution of Problem 15 is realized in the script named area.mq4. The solution variations are searched in two cycles: internal and external. The external cycle searches in the thread lengths with the step of 1000 mm, whereas the internal one finds the maximum area for the current thread length. In this case, the operator 'break' is used to exit both the external and the internal cycle.

```
   //--------------------------------------------------------------------
   // area.mq4
```

```
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------
int start()                               // Special function start()
  {
//--------------------------------------------------------------------------
   int
   L,                                    // Thread length
   S_etalon=1500000,                     // Predefined area (m²)
   S,                                    // Area of the rectangle
   a,b,s;                                // Current side lengths and area
//--------------------------------------------------------------------------
   while(true)                           // External cycle for thread lengths
     {                                   // Start of the external cycle
      L=L+1000;                          // Current thread length of в mm
      //--------------------------------------------------------------------
      S=0;                               // Initial value..
      // ..for each dimension
      for(a=1; a<L/2; a++)               // Cycle operator header
        {                                // HStart of the internal cycle
         b=(L/2) - a;                    // Current side lengths
         s=a * b;                        // Current area
         if (s<=S)                       // Choose the larger value
            break;                       // Exit internal cycle
         S=s;                            // Store the best value
        }                                // End of the internal cycle
      //--------------------------------------------------------------------
      if (S>=S_etalon)                   // Choose the larger value
        {
         Alert("The thread length must be ",L1000," m.");// Message
         break;                           // Exit the external cycle
        }
     }                                   // End of the external cycle
//--------------------------------------------------------------------------
   return;                                // Exit function operator
  }
//--------------------------------------------------------------------------
```

The internal cycle works here like it did in the solution of the preceding problem. The operator 'break' is used to exit the cycle 'for', when the maximum area is found for the given thread length. It must be noted that the operator 'break' specified in the internal cycle 'for' passes the control to the operator that follows the brace closing the cycle 'for', and stops the cycle in this manner. This phenomenon does not influence the execution of the external cycle operator 'while' in any way.

As soon as the operator 'break' triggers in the internal cycle 'for', the control is given to the operator 'if':

```
      if (S >= S_etalon)                           // Choose the larger value
        {
         Alert("The thread length must be ",L1000," m.");// Message
         break;                                    // Exit the external cycle
        }
```

In the operator 'if' we check whether the found area is more or equal to 1.5 m², the minimal value allowed by the problem statement. If yes, then the solution is found and there is no sense to continue calculating; the control will be passed to the body of the operator 'if'. The executable part of the operator 'if' is composed of only two operators, the first of which displays the message about the found solution on the screen:

```
The thread length must be 5 m.
```

The second operator, 'break', is used to exit the external cycle 'while' and, subsequently, to exit the program. The block diagram of the algorithm realized in the program area.mq4 is shown below:

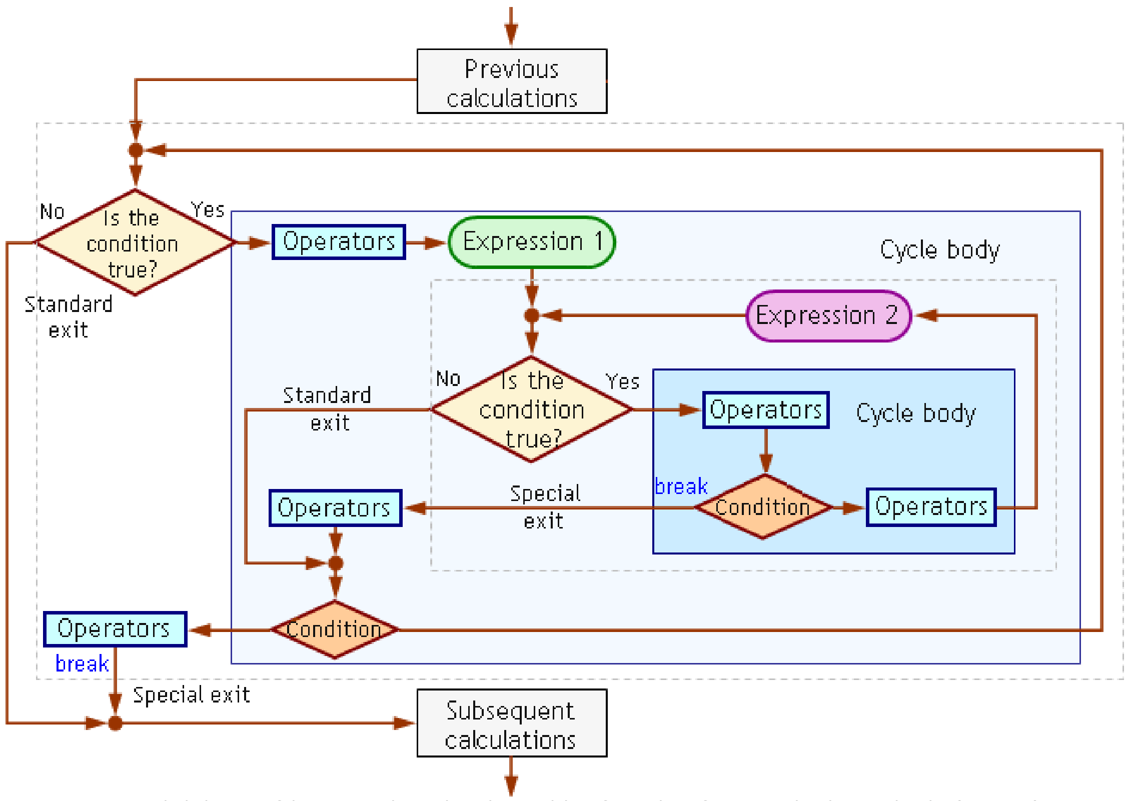


Fig. 46. Block diagram of the program that realizes the possibility of special exit from internal and external cycles (area.mq4).

As is seen from the diagram, there are both a normal and a special exit for each cycle. Each operator 'break' stops its corresponding cycle, but it does not have any effect to the other cycle; each special exit correspond with its own operator 'break'. The same operator 'break' cannot be used in both cycles as special exit. In this case, each cycle has only one special exit. However, generally, it is possible to use several operators 'break' to make several special exits in one cycle.

Please note that the Condition in the header of the external cycle 'while' is (true), i.e., a text that does not contain a variable, the value of which would change during the execution of the program. This means that the program will never exit the cycle 'while' under the Condition specified in the header. In this case, the only possibility to exit the cycle is to use the operator 'break'.

In this example, the algorithm is constructed in such a way that, in fact, the program uses only special exit to exit both the internal and the external cycle. However, you shouldn't think that the use of the operator 'break' always results in constructing algorithms that imply only special exits. In the algorithms of other programs, it is quite possible that the program reaches the condition and uses the normal exit to exit its cycles.

How to use the operator 'break' to pass the control outside the operator 'switch' is considered in the section Operator 'switch'.

← Cycle Operator 'for' Operator 'continue' →

Operator 'continue'

Sometimes, when you use a cycle in the code, it is necessary to early terminate processing of the current iteration and go to the next one without executing the remaining operators that compose the cycle body. In such cases, you should use the operator 'continue'.

Format of the Operator 'continue'

The operator 'continue' consists of one word and ends in ";" (semicolon).

```
      continue;                                // Operator 'continue'
```

Execution Rule of the Operator 'continue'

The operator 'continue' stops the execution of the current iteration of the nearest cycle operator 'while' or 'for'. The execution of the operator 'continue' results in going to the next iteration of the nearest cycle operator 'while' or 'for'. The operator 'continue' can be used only in the body of the above cycle operators.

Let's consider how the operator 'continue' can be practically used.

Problem 16. There are 1000 sheep on the first farm. The amount of sheep on the first farm increases by 1% daily. If the amount of sheep exceeds 50 000 at the end of month, then 10% of sheep will be transferred to the second farm. Within what time will the amount of sheep on the second farm reach 35 000? (We consider that there are 30 working days in a month.)

The algorithm of solving this problem is obvious: We should organize a cycle where the program would calculate the total amount of sheep on the first farm. According to the problem statement, the sheep are transferred to the second farm at the end of month. It means that we will have to create one more (internal) cycle where the accumulation of sheep in the current month would be calculated. Then we will have to check at the end of month, whether the limit of 50 000 sheep is exceeded or not. If yes, then we should calculate the amount of sheep to be transferred to the second farm at the end of month and the total amount of sheep on the second farm.

The algorithm under consideration is realized in script sheep.mq4. The operator 'continue' is used here to make calculations in the external cycle.

```
//--------------------------------------------------------------------------------------
// sheep.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------------------
int start()                                 // Special function start()
  {
//--------------------------------------------------------------------------------------
   int
   day,                                     // Current day of the month
   Mons;                                    // Search amount of months
   double
   One_Farm     =1000.0,                    // Sheep on the 1st farm
   Perc_day     =1,                         // Daily increase, in %
   One_Farm_max=50000.0,                    // Sheep limit
   Perc_exit    =10,                        // Monthly output, in %
   Purpose      =35000.0,                   // Required amount on farm 2
   Two_Farm;                                // Current amount of farm 2
//--------------------------------------------------------------------------------------
   while(Two_Farm < Purpose)                // External cycle on history
     {                                      // Start of the external cycle body
      //--------------------------------------------------------------------------------
      for(day=1; day<=30; day++)            // Cycle for days of month
         One_Farm=One_Farm*(1+Perc_day/100);//Accumulation on the 1st farm
      //--------------------------------------------------------------------------------
      Mons++;                               // Count months
      if (One_Farm < One_Farm_max)          // If the amount is below limit,.
         continue;                          // .. don't transfer the sheep
      Two_Farm=Two_Farm+One_Farm*Perc_exit/100;//Sheep on the 2nd farm
      One_Farm=One_Farm*(1-Perc_exit/100);// Remainder on the 1st farm
     }                                      // End of the external cycle body
//--------------------------------------------------------------------------------------
   Alert("The aim will be attained within ",Mons," months.");//Display on the screen
   return;                                  // Exit function start()
  }
//--------------------------------------------------------------------------------------
```

At the start of the program, as usually, variables are described and commented. The calculation itself is performed in the cycle 'while'. After it has been executed, the corresponding message is displayed. The calculations in the external cycle 'while' will be executing, until the aim is attained, namely, until the total amount of sheep on the second farm reaches the expected value of 35 000.

The internal cycle 'for' is very simple: the value of the balance daily increases by 1%. No sum analysis is performed in this cycle, since, according to the problem statement, sheep can only be transferred at the end of month. Thus, after exiting the cycle 'for', variable One_Farm has the value that is equal to the amount of sheep on the first farm. Immediately after that, the value of variable Mons is calculated, which increases by 1 at execution of each iteration of the external cycle 'while'.

According to the current amount of sheep on the first farm, one of two actions below shall be performed:

- if the amount of sheep on the first farm exceeds the limit value of 50 000, then 10% of sheep of the first farm should be transferred to the second farm;
- otherwise, the sheep from the first farm stay on the first farm and are bred further.

The algorithm is branched using the operator 'if':

```
      if (One_Farm < One_Farm_max)          // If the amount is below limit,.
         continue;                          // .. don't transfer the sheep
```

These lines of the code can be characterized as follows: If the amount of sheep on the first farm is below the limiting value of 50 000, then execute the operator composing the body of the operator 'if'. At the first iteration, the amount of sheep on the first farm turns out to be less than the limiting value, so the control will be passed to the body of the operator 'if', i.e., to the operator 'continue'. The execution of the operator 'continue' means the following: Terminate the current iteration of the nearest executing cycle (in this case, it is the cycle in the operator 'while') and pass the control to the header of the cycle operator. The following program lines that are also a part of the cycle operator-'while' body will not be executed:

```
      Two_Farm = Two_Farm+One_Farm*Perc_exit/100;//Sheep on the 2nd farm
      One_Farm = One_Farm*(1-Perc_exit/100);     // Remainder on the 1st farm
```

In these above lines, the reached amount of sheep on the second farm and the remaining sheep on the first farm are calculated. It is impossible to make these calculations at the first iteration, since the limit of 50 000 sheep on the first farm has not been reached yet. It is the essence of the operator 'continue' that consists in skipping these lines without executing them in the current iteration.

The execution of the operator 'continue' results in passing the control to the header of the cycle operator 'while'. Then the condition in the cycle operator is tested and the next iteration starts. The cycle will continue to be executed according to the same algorithm, until the amount of sheep on the first farm reaches the predefined limit. The values of Mons are accumulated at each iteration.

At a certain stage of calculating, the amount of sheep on the first farm will reach or exceed the predefined limit of 50 000 head. In this case, at the execution of the operator 'if', the condition (One_Farm < One_Farm_max) becomes false, so the control will not be passed to the body of the operator 'if'. This means that the operator 'continue' will not be executed, whereas the control will be passed to the first of the last two lines of the cycle-'while' body: First, the program will calculate the amount of sheep on the second farm and then the remaining amount of sheep on the first farm. After these calculations have been completed, the iteration of the cycle 'while' will be finished and the control will be passed to the cycle header again. Below is the block diagram of the algorithm realized in script sheep.mq4.

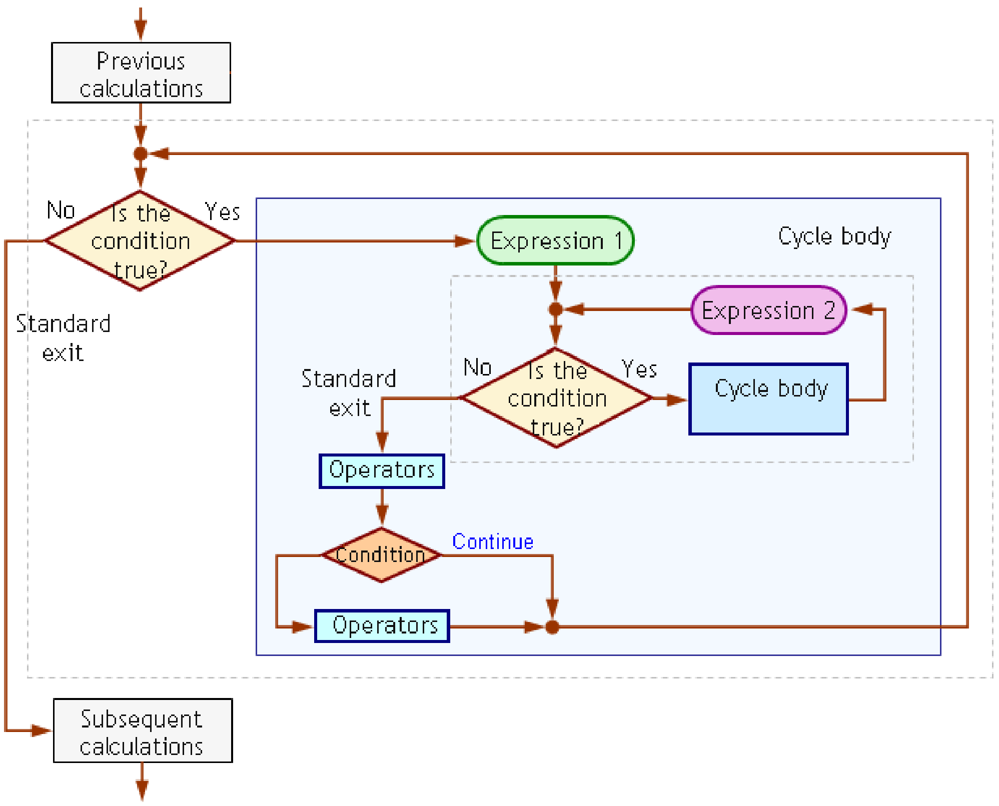


Fig. 47. Block diagram of a program where the operator 'continue' breaks iterations of the external cycle (sheep.mq4).

In the diagram, we can see that, depending on the execution of the condition in the operator 'if', the control will be immediately passed to the header of the cycle 'while' (as a result of execution of the operator 'continue'), or some operators are executed first, and then the control will still be passed to the header of the cycle 'while'. In both cases, the execution of the cycle does not end.

Please note that the presence of the operator 'continue' in the cycle body does not necessarily result in termination of the current iteration. This only happens if it executes, i.e., if the control is passed to it.

At each next iteration in the cycle operator 'while', the new reached value of variable Two_Farm will be calculated. As soon as this value exceeds the predefined limit (35 000), the condition in the cycle operator 'while' will become false, so the calculations in the cycle-'while body will not be performed any longer, whereas the control will be passed to the nearest operator that follows the cycle operator:

```
      Alert("The aim will be attained within ",Mons," months.");//Display on the screen
```

The execution of the function Alert() will result in displaying of the following line:

```
The aim will be attained within 17 months.
```

The operator 'return' completes execution of the special function start(), which results in that the control is passed to the client terminal, whereas the execution of the script is completed, it will be unloaded from the symbol window.

The algorithm above also provides a standard exit from the cycle 'while'. In a more general case, the exit from the cycle can result from the execution of the operator 'break' (special exit). However, neither one nor another way to close the cycle is related to interruption of the current iteration using the operator 'continue'.

In the execution rule of the operator 'continue' the phrase of "the nearest operator" is used. It does not mean that the program lines are close to each other. Let's have a look at the code of script sheep.mq4. The operator 'for' is "closer" to the operator 'continue' than the header of the cycle 'while'. However, this does not mean anything: The operator 'continue' has no relation to the cycle 'for', because it is outside its body. In general case, a program can contain multiple nested cycle operators. The operator 'continue' is always in the body of one of them. At the same time, the operator 'continue', as a part of the internal cycle operator, is, of course, also inside the external cycle operator. The phrase of "stops the execution of the current iteration of the nearest cycle operator" should mean that the operator 'continue' influences the nearest cycle operator, within the body of which the operator 'continue' is located. In order to visualize it, let's slightly change the problem statement.

Problem 17. There are 1000 sheep on one farm. The amount of sheep on this first farm increases daily by 1%.

On the day, when the amount of sheep on the first farm reaches 50 000, 10% of sheep will be transferred to the second farm. Within what time will the amount of sheep on the second farm reach 35 000? (We consider that there are 30 working days in a month.)

The solution of problem 17 is realized in script othersheep.mq4. In this case, the operator 'continue' is used for calculations in both the external and internal cycles.

```
//--------------------------------------------------------------------
// othersheep.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int start()                                // Special function start()
  {
//--------------------------------------------------------------------
   int
   day,                                    // Current day of the month
   Mons;                                   // Search amount of months
   double
   One_Farm    =1000.0,                    // Sheep on the 1st farm
   Perc_day    =1,                         // Daily increase, in %
   One_Farm_max=50000.0,                   // Sheep limit
   Perc_exit   =10,                        // One-time output, in %
   Purpose     =35000.0,                   // Required amount on farm 2
   Two_Farm;                               // Current amount of farm 2
//--------------------------------------------------------------------
   while(Two_Farm < Purpose)               // Until the aim is attained
     {                                     // Start of the external cycle body
      //--------------------------------------------------------------
      for(day=1; day<=30 && Two_Farm < Purpose; day++)// Cycle by days
        {
         One_Farm=One_Farm*(1+Perc_day/100);//Accumulated on farm 1
         if (One_Farm < One_Farm_max)      // If the amount is below limit,.
            continue;                      // .. don't transfer the sheep
         Two_Farm=Two_Farm+One_Farm*Perc_exit/100;//Accumulated on farm 2
         One_Farm=One_Farm*(1-Perc_exit/100);     //Remainder on farm 1
        }
      //--------------------------------------------------------------
      if (Two_Farm>=Purpose)               // If the aim is attained,..
         continue;                         // .. don't count months
      Mons++;                              // Count months
     }                                     // End of external cycle body
//--------------------------------------------------------------------
   Alert("The aim will be attained within ",Mons," months and ",day," days.");
   return;                                 // Exit function start()
  }
//--------------------------------------------------------------------
```

To solve the problem, we have to analyze the amount of sheep for each day. For this purpose, the lines:

```
   if (One_Farm < One_Farm_max)              // If the amount is below limit,.
      continue;                              // .. don't transfer the sheep
   Two_Farm=Two_Farm+One_Farm*Perc_exit/100;//Accumulated on farm 2
   One_Farm=One_Farm*(1-Perc_exit/100);      //Remainder on farm 1
```

are transferred into the internal cycle organized for the days of month. It is obvious that a certain amount of sheep are transferred to the second farm, in this case, not at the end of month, but on the day, when the amount of sheep on the first farm reaches the predefined value. The reason for using here the operator 'continue' is the same: We want to skip the lines containing the calculations related to the transfer of sheep from one farm to another, i.e., we don't want to execute these lines. Please note that the cycle 'while' constructed for the days of month is not interrupted independently on whether the sheep were transferred or not, because the operator 'continue' influences the nearest cycle operator, in our case, the cycle operator 'for'.

We used a complex expression as the Condition in the cycle operator 'for':

```
   for(day=1; day<=30 && Two_Farm<Purpose; day++)// Cycle by days
```

Using the operation && ("and") means that the cycle will be executed as long as both conditions are true:

- the month has not ended yet, i.e., the variable 'day' ranges from 1 through 30; and
- the amount of sheep on the second farm has not reached the predefined value yet, i.e., the variable Two_Farm is less than Purpose

(35 000).

If at least one of the conditions is not met (becomes false), the cycle stops. If the execution of the cycle 'for' is stopped (for any reason), the control is passed to the operator 'if':

```
      if (Two_Farm >= Purpose)             // If the aim is attained,..
         continue;                         // .. don't count months
      Mons++;                              // Count months
```

The use of the operator 'continue' at this location in the code has a simple sense: The program must continue counting months only if the amount of sheep on the second farm is below the expected value. If the aim has already been attained, the value of the variable Mons will not change, whereas the control (as a result of the execution of 'continue') will be passed to the header of the nearest cycle operator, in our case, of the cycle operator 'while'.

The use of operators 'continue' in nested cycles is shown in Fig. 48 below.

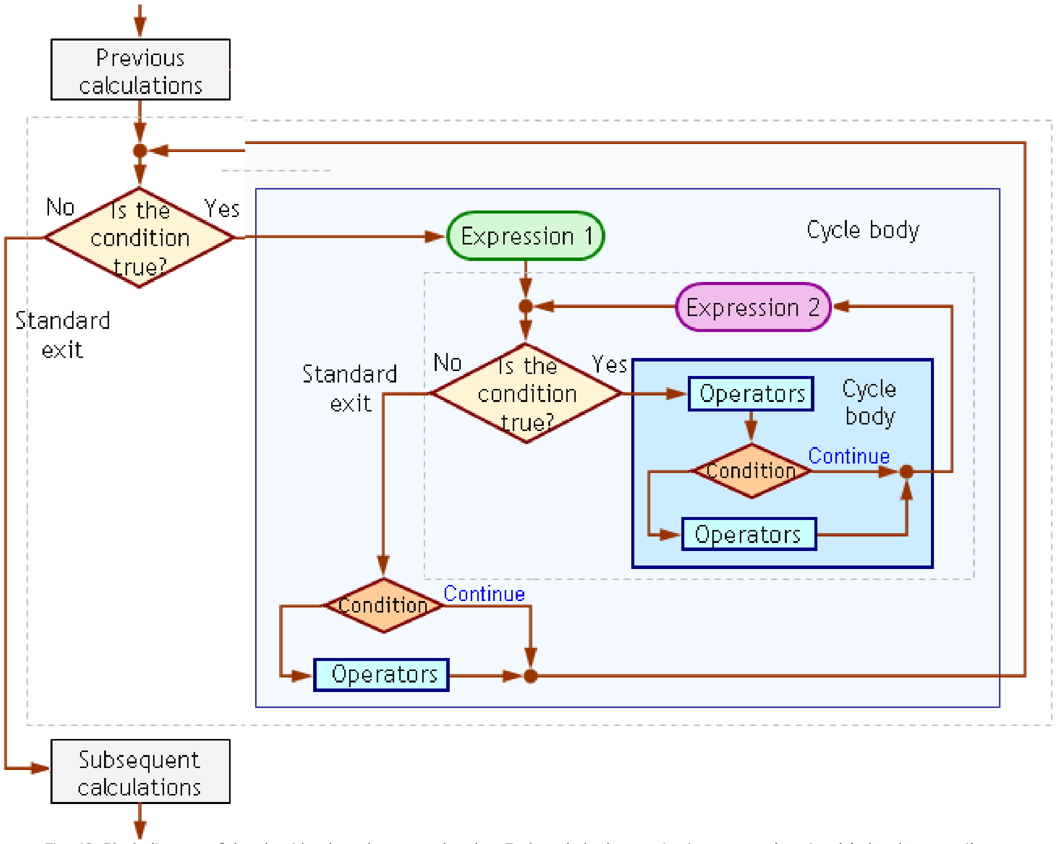


Fig. 48. Block diagram of the algorithm based on nested cycles. Each cycle body contains its operator 'continue' (othersheep.mq4).

In the above example, each cycle (external and internal) has one operator 'continue' that influences only its "own" nearest cycle, but not any events in any other cycle. Generally, in a cycle body, several operators 'continue' can be used, each being able to interrupt the current iteration as a result of meeting of a condition. The amount of operators 'continue' in a cycle body is not limited.

← Operator 'break' Operator 'switch' →

Operator 'switch'

Some programs imply branching of their algorithm into several variations. In such cases, it is very convenient to use the operator 'switch', especially if there are tens or hundreds of variations, whereas the 'if' code becomes bloated if you use many nested operators.

Format of the Operator 'switch'

The operator 'switch' consists of the header and the executable body. The header contains the name of the operator and an Expression enclosed in parentheses. The operator body contains one or several variations 'case' and one variation 'default'.

Each 'case' variation consists of the key word 'case', a Constant, ":" (colon), and operators. The amount of variations 'case' is not limited.

The variation 'default' consists of the key word 'default', ":" (colon), and operators. The variation 'default' is specified in the body of the operator 'switch' as the last, but it can also be located anywhere within the operator body or even be absent.

```
switch ( Expression )                          // Operator header
   {                                           // Opening brace
   case Constant: Operators                   // One of the 'case' variations
   case Constant: Operators                   // One of the 'case' variations
   ...
   [default: Operators]                        // Variation without any parameter
   }                                          // Closing brace
```

The values of Expression and of Parameters can only be the values of *int* type. The Expression can be a constant, a variable, a function call, or an expression. Each variation 'case' can be marked by an integer constant, a character constant, or a constant expression. A constant expression cannot include variables or function calls.

Execution Rule of the Operator 'switch'

The program must pass the control to the first operator that follows the ":" (colon) of the variation 'case', the Constant value of which is the same as the value of the Expression, and then execute one by one all operators composing the body of the operator 'switch'. The condition of equality between the Expression and the Constant is tested in the direction from top to bottom and from left to right. The values of Constants in different variations 'case' must not be the same. The operator 'break' stops the execution of the operator 'switch' and passes the control to the operator that follows the operator 'switch'.

It is easy to see that the '**case** Constant:' represents just a label, to which the control is passed. All operators that compose the body of the operator 'switch' start being executed from this label. If the program algorithm implies the execution of a group of operators that correspond with only one variation 'case', you must specify the operator 'break' as the last in the list of operators that correspond with only one variation 'case'. Let's consider the work of the operator 'switch' using some examples.

Examples of the Operator 'switch'

Problem 18. Compose a program where the following conditions are realized: If the price exceeds the predefined level, the program must inform the trader about it using a message where the excess is described in words (up to 10 points); in other cases, the program must inform that the price does not exceed the predefined level.

Below is the problem solution using the operator 'switch' (Expert Advisor pricealert.mq4):

```
//--------------------------------------------------------------------------------------
// pricealert.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------------------
int start()                                      // Special function 'start'
  {
   double Level=1.3200;                            // Preset price level
   int Delta=NormalizeDouble((Bid-Level)Point,0); // Excess
   if (Delta<=0)                                   // Price doesn't excess the level
     {
      Alert("The price is below the level");       // Message
      return;                                      // Exit start()
     }
//--------------------------------------------------------------------------------------
   switch(Delta)                                   // Header of the 'switch'
     {                                             // Start of the 'switch' body
      case 1 : Alert("Plus one point");      break;// Variations..
      case 2 : Alert("Plus two points");     break;
```

```
         case 3 : Alert("Plus three points");  break;
         case 4 : Alert("Plus four points");   break;//Here are presented
         case 5 : Alert("Plus five points");   break;//10 variations 'case',
         case 6 : Alert("Plus six points");    break;//but, in general case,
         case 7 : Alert("Plus seven points");  break;//the amount of variations 'case'
         case 8 : Alert("Plus eight points");  break;//is unlimited
         case 9 : Alert("Plus nine points");   break;
         case 10: Alert("Plus ten points");    break;
         default: Alert("More than ten points");     // It is not the same as the 'case'
        }                                            // End of the 'switch' body
//-------------------------------------------------------------------------------------
   return;                                           // Exit start()
  }
//-------------------------------------------------------------------------------------
```

In this problem solution, we use the operator 'switch', in which each variation 'case' uses the operator 'break'. Depending on the value of variable Delta, the control will be passed to one of variations 'case'. This results in that the operators corresponding with this variation are executed: function Alert() and the operator 'break'. The operator 'break' stops execution of the operator 'switch', i.e., it passes the control outside it, namely, to the operator 'return' that ends the operation of the special function start(). Thus, depending on the value of variable Delta, one of variations 'case' triggers, whereas other variations remain untouched.

The above program is an Expert Advisor, so it will be launched for execution at every tick and it will every time display the messages corresponding with the current situation. Of course, we should search a value for the Level possibly close to the current price of the symbol, the window, to which we are attaching this EA.

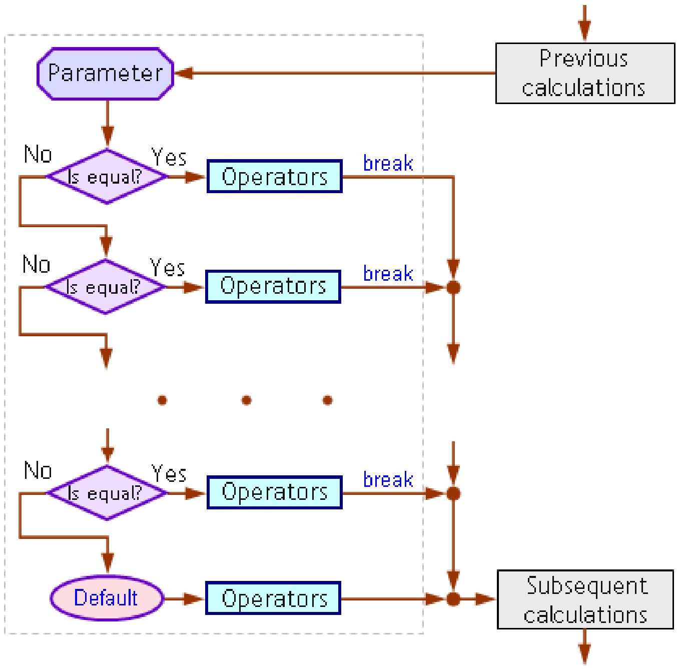


Fig. 49. Block diagram of the operator 'switch' in EA pricealert.mq4.

In the block diagram above, we can clearly see that, due to the presence of the operator 'break' in each variation 'case', the control is passed outside the operator 'switch' after the operators of any variation 'case' have been executed. A similar principle of algorithm construction using the operator 'switch' is utilized in the file named stdlib.mq4 delivered within the client terminal (***..\experts\libraries\stdlib.mq4***).

Let's consider another problem that does not provide the usage of 'break' in each variation 'case'.

Problem 19. There are 10 bars. Report the numbers of all bars starting with the nth one.

It is rather easy to code the solution of this problem (script barnumber.mq4):

```
//------------------------------------------------------------------------------
// barnumber.mq4
// The code should be used for educational purpose only.
//------------------------------------------------------------------------------
int start()                           // Special function start()
  {
   int n = 3;                         // Preset number
   Alert("Bar numbers starting from ", n,":");// It does not depend on n
//--------------------------------------------------------------------------------
   switch (n)                            // Header of the operator 'switch'
     {                                   // Start of the 'switch' body
      case 1 : Alert("Bar 1");           // Variations..
      case 2 : Alert("Bar 2");
      case 3 : Alert("Bar 3");
      case 4 : Alert("Bar 4");           // Here are 10 variations ..
      case 5 : Alert("Bar 5");           // ..'case' presented, but, in general, ..
      case 6 : Alert("Bar 6");           // ..the amount of variations..
      case 7 : Alert("Bar 7");           // ..'case' is unlimited
      case 8 : Alert("Bar 8");
      case 9 : Alert("Bar 9");
      case 10: Alert("Bar 10");break;
      default: Alert("Wrong number entered");// It is not the same as the 'case'
     }                                   // End of the 'switch' body
//--------------------------------------------------------------------------------
   return;                               // Operator to exit start()
  }
//--------------------------------------------------------------------------------
```

In the operator 'switch', the program will search in the variations 'case', until it detects that the Expression is equal to the Constant. When the value of the Expression (in our case, it is integer 3) is equal to one of the Constants (here, it is case 3), all operators that follows the colon (case 3:) will be executed, namely: the operator of function call Alert("Bar 3"), those following Alert("Bar 4"), Alert("Bar 5"), etc., until it comes to the operator 'break' that terminates the operation of the operator 'switch'.

If the value of the Expression does not coincide with any of the Constants, the control is passed to the operator that corresponds to the variation 'default':

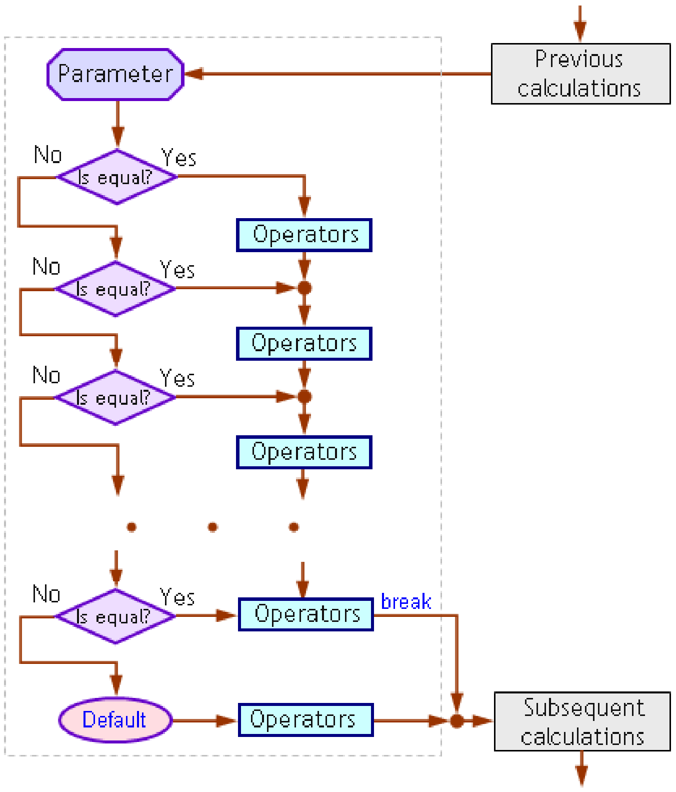


Fig. 50. Block diagram of the operator 'switch' in script barnumber.mq4.

Unlike the algorithm realized in the preceding program, in this case (Fig. 50), we are not using the operator 'break' in each variation 'case'. So, if the value of the Expression is equal to the value of one of the Constants, all operators will be executed starting with the operators of the corresponding variation 'case'. We also use the operator 'break' in the last variation 'case' for another purpose: to prevent the operators corresponding with the variation 'default' from executing. If there is no value equal to the Expression among the values of the Constants, the control will be passed to the operator that corresponds with the 'default' label.

Thus, if the value of the preset variable n is within the range of values from 1 to 10, the numbers of all bars will be printed starting with the nth one. If the value of n is beyond the above range, the program will inform the user about no matches.

Please note: It is not necessary that the Constants of the variations 'case' are arranged in your program in order of magnitude. The order of how the variations 'case' with the corresponding Constants follow each other is determined by the needs of the algorithm of your program.

← Operator 'continue' Function Call →

Function Call

A function call can be used as a separate operator and be found in any place in a program where it implies a certain value (with the exception of predefined cases). The format and the execution rules of a function call cover both standard (built-in) and user-defined functions.

Function Call Format

A function call consists of the function name and the list of the passed parameters enclosed in parentheses:

```
     Function_name  (Parameters_list)      // Function call as such
```

The function name specified in the function call must be the same as the name of the function you want to call for execution. The parameters in the list are separated by commas. The amount of parameters to be passed to the function is limited and cannot exceed 64. In a function call, you can use constants, variables and other function calls as parameters. The amount, types and order of the passed parameters in a function call must be the same as the amount, types and order of formal parameters specified in the function description (the exception is a function call with default parameters).

```
  My_function (Alf, Bet)                      // Example of a function call
                                              // Here:
  My_function                                 // Name of the called function
  Alf                                         // First passed parameter
  Bet                                         // Second passed parameter
```

If the called function does not imply passing any parameters, the list of parameters is specified as empty, but the parentheses must be present, anyway.

```
  My_function ()                              // Exemplary function call
                                              // Here:
  My_function                                 // Name of the called function
                                              // There are no parameters to be passed
```

If the program must call a function with default parameters, the list of the passed parameters can be limited (shortened). You can limit the list of parameters, starting with the first default parameter. In the example below, the local variables b, c and d have some default values:

```
                                              // For the function described as:
  int My_function (int a, bool b=true, int c=1, double d=0.5)
     {
     Operators
     }
                                              // .. the following calls are allowed:
  My_function (Alf, Bet, Ham, Del)            // Allowed function call
  My_function (Alf )                          // Allowed function call
  My_function (3)                             // Allowed function call
  My_function (Alf, 0)                        // Allowed function call
  My_function (3, Tet)                        // Allowed function call
  My_function (17, Bet, 3)                    // Allowed function call
  My_function (17, Bet, 3, 0.5)               // Allowed function call
```

The parameters without default values may not be skipped. If a default parameter is skipped, no subsequent default parameters must be specified.

```
                                              // For the function described as:
  int My_function (int a, bool b=true, int c=1, double d=0.5)
     {
     Operators
     }
                                              // ..the following calls are ошибочными:
  My_function ()                              // Forbidden function call: non-default..
                                              // ..parameters may not be skipped (the first one)
  My_function (17, Bet, , 0.5)                // Forbidden function call: skipped..
                                              // ..default parameter (the third one)
```

The called functions are divided into two groups: those that return a certain value of a predefined type and those that don't return any value.

Format of Non-Return Function Call

A call for a function that does not return any value can only be composed as a separate operator. The call function operator is ended in ";" (semicolon):

```
  Function_name  (Parameter_list);       // Operator of non-return function call
```

```
  Func_no_ret (Alpha, Beta, Gamma);           // Example of a function call operator..
```

```
                                            //.. for a function that does not return any value
```

No other format (technique) is provided to call to functions that don't return any values.

Format of Return Function Call

A call to a function that returns a value can be composed as a separate operator or it can be used in the program code at places where a value of a certain type is implied.

If the function call is composed as a separate operator, it ends in ";" (semicolon):

```
Function_name  (Parameter_list);       // Operator of return function call
```

```
Func_yes_ret (Alpha, Beta, Delta);           // Example of a function call operator..
                                             //.. for a function that returns a value
```

Execution Rule of Function Call

A function call calls the function of the same name for execution. If the function call is composed as a separate operator, after the function has been executed, the control is passed to the operator that follows the function call. If the function call is used in an expression, after the function has been executed, the control is passed to the location in the expression, where the function call is specified; the further calculations in the expression will be performed using the value returned by the called function.

The use of function calls in other operators is determined by the format of these operators.

Problem 20. Compose a program where the following conditions are realized:
- if the current time is more than 15:00, execute 10 iterations in the cycle 'for';
- in all other cases, execute 6 iterations.

Below is an example of script callfunction.mq4 that includes: a function call in the header of the operator 'for' (as a part of Expression_1, according to the format of the operator 'for', see Cycle Operator 'for'), a standard function call as a separate operator, in the right part of the assignment operator (see Assignment Operator), and in the header of the operator 'if-else' (in the Condition, according to the format of the operator 'if-else', see Conditional Operator 'if-else').

```
///-----------------------------------------------------------------------------------
// callfunction.mq4
// The code should be used for educational purpose only.
//-----------------------------------------------------------------------------------
int start()                                 // Description of function start()
  {                                         // Start of the function start() body
   int n;                                   // Variable declaration
   int T=15;                                // Predefined time
   for(int i=Func_yes_ret(T);i<=10;i++)     // The use of the function in..
                                            //.the cycle operator header
     {                                      // Start of the cycle 'for' body
      n=n+1;                                // Iterations counter
      Alert ("Iteration n=",n," i=",i);     // Function call operator
     }                                      // End of the cycle 'for' body
   return;                                  // Exit function start()
  }                                         // End of the function start() body
//------------------------------------------------------------------------------------
int Func_yes_ret (int Times_in)             // Description of the user-defined function
  {                                         // Start of the user-defined function body
   datetime T_cur=TimeCurrent();            // The use of the function in..
                                            // ..the assignment operator
   if(TimeHour(T_cur) > Times_in)           // The use of the function in..
                                            //..the header of the operator 'if-else'
      return(1);                            // Return value 1
   return(5);                               // Return value 5
  }                                         // End of the user-defined function body
//------------------------------------------------------------------------------------
```

In the above example, the following functions were called with the following passed parameters:

- call to the function Func_yes_ret(T) - variable T;
- call to the function Alert () - string constants "Iteration n=" and " i=", variables n and i;
- call to the function TimeCurrent() does not provide any parameters to be passed;
- call to the function TimeHour(T_cur) - variable T_cur.

A very simple algorithm is realized in this program. We set in the variable T the time (in hours), in relation to which the calculations are performed. In the header of the operator 'for', we specify the call to the user-defined function Func_yes_ret() that can return one of two values: 1 or 5. According to this value, the amount of iterations in the cycle will change: there will be either 10 (i changes from 1 to 10) or 6 (i changes from 5 to 10) iterations. For better visualization, in the cycle body we use the iteration counter, each value of which is displayed on the screen using function Alert().

In the description of the user-defined function, we first calculate the time in seconds elapsed after 00:00 of the 1st of January 1970 (call to function TimeCurrent()), and then we calculate the current time in hours (call to function TimeHour()). The algorithm is branched using the operator 'if' (call to function TimeHour() is specified in its condition). If the current time turns out to be more than that passed to the user-defined function (local variable Times_in), the latter one returns 1, otherwise it returns 5.

Please note:

There are no descriptions of standard functions or function call for function start() in the program.

Below you can see the block diagram of script callfunction.mq4:

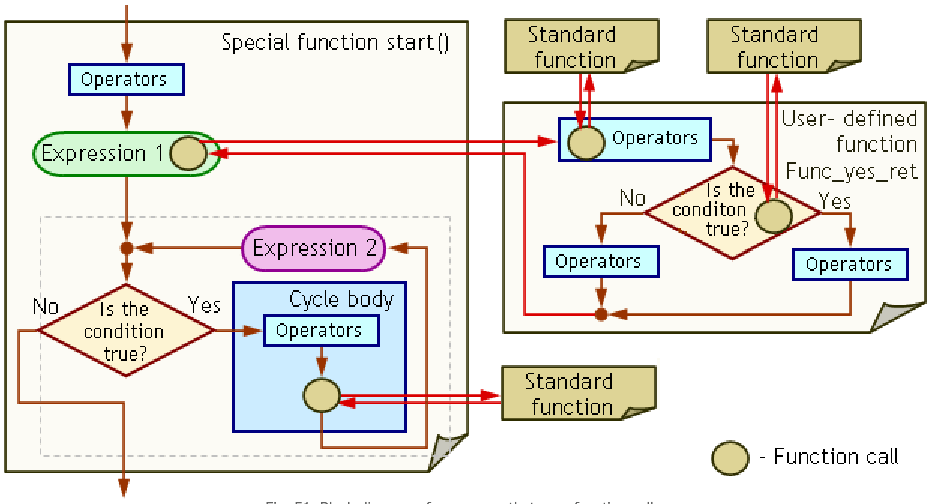


Fig. 51. Block diagram of a program that uses function calls.

The circles in the diagram mark function calls (for the standard and the user-defined function). Red arrows show the passing of the control into the function and vice versa. You can clearly see that the function returns the control to the location where the function call is specified, no calculations being made on the path between the function call and the function itself. Generally, if a function returns a value, this value will be passed to the calling module (along the red arrow in the direction of the function call).

Special functions can be called from any location in the program according to the general rules, on a par with other functions. Special functions can also have parameters. However, when the client terminal calls these functions, no parameters will be passed from outside, it will use the default values. The use of parameters in special functions will only be reasonable if they are called from a program. Although it is technologically possible in MQL4 to call special functions from a program, it is not recommended to do so. A program that uses special function calls should be considered as incorrect.

← Operator 'switch' Function Description and Operator 'return' →

Function Description and Operator 'return'

As to the necessity to specify functions in a program, they can be divided into 2 groups: Functions that are not described in a program, and functions that must be described in a program. Standard functions are not described in programs. User-defined functions must be described in any program. Special functions, if any, must be described in a program, too.

Format of Function Description

A function description consists of two basic parts: function header and function body.

The function header contains the type of the return value, the function name, and the list of formal parameters enclosed in parentheses. If a function must not return any value, its type should be named *void*.

The function body can consist of simple and/or compound operators and calls to other functions, and is enclosed in parentheses.

```
Return_value_type Function_name (List of formal parameters)//Header
   {                                    // Opening brace
   Program code                         // A function body may consist ..
   composing the function               //.. of operators and ..
   body                                 //.. calls to other functions
   }                                    // Closing brace
```

The parameters in the list are given separated by commas. The amount of parameters passed to function is limited and cannot exceed 64. As formal parameters in the function header, you can specify only variables (but not constants, other function calls or expressions). The amount, type and order of the passed parameters in the function call must be the same as those of formal parameters specified in the function description (the only exception will be the call to a function that has parameters with the default values):

```
int My_function (int a, double b)      // Example of function description
  {
   int c = a * b + 3;                  // Function body operator
   return (c);                         // Function exit operator
  }
                                       // Here (from left to right in the header):
   int                                 // Return value type
   My_function                         // Function name
   int a                               // First formal parameter a of the int type
   double b                            // Second formal parameter b of the double type
```

Parameters passed to the function can have default values that are defined by the constant of the corresponding type:

```
int My_function (int a, bool b=true, int c=1, double d=0.5)//Example of function description
  {
   a = a + b*c + d2;                   // Function body operator
   int k = a * 3;                      // Function body operator
   return (k);                         // Function exit operator
  }
                                       // Here (from left to right in the header):
   int                                 // Return value type
   My_function                         // Function name
   int a                               // First formal parameter a of the int type
   bool b                              // Second formal parameter b of the double type
   true                                // Constant, the default value for b
   int c                               // Third formal parameter c of the int type
   1                                   // Constant, the default value for c
   double d                            // Fourth formal parameter d of the double type
   0.5                                 // Constant, the default value for d

   a,b,c,d,k                           // Local variables
```

If there are actual parameters given in the call to the function with default values, the values of the actual parameters will be calculated in the function. If there are no actual parameters given in the call to the function with default values, the corresponding default values will be calculated.

Special functions can have parameters, too. However, the client terminal does not pass any parameters from outside when calling these functions, it just uses the default values. Special functions can be called from any location of the module according to general rules, on a par with other functions.

Function Execution Rules

Location for a function description in a program:

The function description must be placed in your program separately, outside any other functions (i.e., it must not be located in another function).

Function execution:

A function called for execution is executed according the code that composes the function body.

Format of Operator 'return'

The value returned by the function is the value of the parameter specified in the parentheses of the operator 'return'. The operator 'return' consists of the key word 'return', the Expression enclosed in parentheses, and ends in the character of ";" (semicolon). The full-format operator 'return':

```
     return (Expression);                    // Operator return
```

The expression in parentheses can be a constant, a variable or a function call. The type of the value returned using the operator 'return' must be the same as the type of the function return value specified in the function header. If this is not the case, the value of the expression specified in the operator 'return' should be cast to the type of the return value specified in the header of the function description. If the typecasting is impossible, MetaEditor will give you an error message when compiling your program.

Execution Rule of Operator 'return'

The operator 'return' stops the execution of the nearest external function and passes the control to the calling program according to the rules defined for a function call. The value returned by the function is the value of the expression specified in the operator 'return'. If the type of the parameter value specified in the operator 'return' is other than that of the return value specified in the function header, the value must be cast to the type of the return value specified in the header.

An example of how to use the operator 'return' that returns a value:

```
bool My_function (int Alpha)              // Description of the user-defined function
  {                                       // Start of the function body
   if(Alpha>0)                            // Operator 'if'
     {                                    // Start of the operator-'if' body
      Alert("The value is positive");     // Standard function call
      return (true);                      // First exit from the function
     }                                    // End of the operator-'if' body
   return (false);                        // Second exit from the function
  }                                       // End of the function body
```

If the return value of a function is of the *void* type, you should use the operator 'return' without expression:

```
     return;                                 // Operator 'return' without the expressions in parentheses
```

An example of usage of the operator 'return' without return value:

```
void My_function (double Price_Sell)            // Description of the user-defined function
  {                                             // Start of the function body
   if(Price_Sell-Ask >100)                      // Operator 'if'
      Alert("Profit for this order exceeds 100 п");// Standard function call
   return;                                      // Exit function
  }                                             // End of the function body
```

It is allowed not include the operator 'return' in a function description. In this case, the function will terminate its operation automatically, as soon as (according to the executing algorithm) the last operator is executed in the function body. An example of the description of a function without the operator 'return':

```
void My_function (int Alpha)              // Description of the user-defined function
  {                                       // Start of the function body
   for (int i=1; i<=Alpha; i++)           // Cycle operator
     {                                    // Start of the cycle body
      int a = 2*i + 3;                    // Assignment operator
      Alert ("a=", a);                    // Standard function call operator
     }                                    // End of the cycle body
  }                                       // End of the function body
```

In this case, the function will complete its operations at the moment when the cycle operator 'for' finishes its execution. The last action at the function execution will be the condition test in the cycle operator. As soon as the Condition in the header of the cycle operator 'for' becomes false, the control will be passed outside the cycle operator. However, for the reason that the cycle operator is the last executable operator in the body of the function named My_function (), the user-defined function will terminate its operations, whereas the control will be passed outside the function, to the location, from which the function was called for execution.

← Function Call Variables →

Variables

For creating programs in any algorithmic language knowing different variable types is very important. In this section we will analyze all types of variables used in MQL4.

- Predefined Variables and RefreshRates Function.
  First of all predefined variables should be learned. Names of predefined variables are reserved and cannot be used for creating custom variables. It is predefined variables that bear the main information necessary for analyzing current market situation. For updating this information RefreshRates() function is used.

- Types of Variables.
  Variables are very important in writing a program. They are divided into local and global, external and internal. Statistic variables preserve their values between function calls, it is useful for remembering some values of local variables without creating global variables.

- GlobalVariables.
  Beside global variables on the level of a separate program, the values of which are available from any part of the program, there are global variables on terminal level. Such global variables are called GlobalVariables. They make it possible to establish interaction between independent parts of programs written in MQL4. They can be used for sharing values between scripts, indicators and Expert Advisors. At terminal closing values of GlobalVariables are also preserved for being available at the next MetaTrader 4 start. It must be remembered that if there were no calls to a global variable during 4 weeks, it will be deleted.

- Arrays.
  If you need to save or process large volumes of one-type values, you cannot do without arrays. Before using an array, it should be declared like a variable. Calling of array elements is performed via specifying index(es) of an element. Array indexation starts from zero. Number of array dimensions is called dimensionality. Arrays larger than four-dimensional ones are not accepted. Array values must be clearly initialized in order to avoid difficult-to-locate errors.

← Function Description and Operator 'return' Predefined Variables and the Function RefreshRates →

Predefined Variables and RefreshRates Function

There are variables with predefined names in MQL4 language.

Predefined variable is a variable with a predefined name, the value of which is defined by a client terminal and cannot be changed by program methods. Predefined variables reflect the state of a current chart at the moment of program start (Expert Advisor, script or custom indicator) or as a result of RefreshRates() implementation.

List of Simple Predefined Names of Variables

Ask **-** last known sell-price of a current security;

Bid **-** last known buy-price of a current security;

Bars - number of bars on a current chart;

Point - point size of a current security in quote currency;

Digits - number of digits after a decimal point in the price of a current security.

List of Predefined Names of Arrays-Timeseries

Time - opening time of each bar on the current chart;

Open - opening price of each bar on the current chart;

Close - closing price of each bar on the current chart;

High - maximal price of each bar on the current chart;

Low - minimal price of each bar on the current chart;

Volume - tick volume of each bar on the current chart.

(concepts of "arrays" and "arrays-timeseries" are described in the section Arrays).

Properties of Predefined Variables

Predefined variable name cannot be used for identifying user-defined variables. Predefined variables can be used in expressions equally with other variables in accordance with the same rules, but the value of a predefined variable cannot be changed. When trying to compile a program containing an assignment operator, in which a predefined variable is placed to the right of an equality sign, MetaEditor will show an error message. In terms of visibility predefined variables refer to global, i.e. they are available from any program part (see Types of Variables).

The most important property of predefined variables is the following:

Values of all predefined variables are automatically updated by a client terminal at the moment when special functions are started for execution.

Previous and current values of predefined variables may be equal, but the value itself will be updated. At the moment of a special function's start values of these variables are already updated and are available from the first program lines. Let's illustrate updating of predefined variables on the following example (Expert Advisor predefined.mq4):

```
//--------------------------------------------------------------------
// predefined.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int start()                                     // Special funct. start
  {
   Alert("Bid = ", Bid);                        // Current price
   return;                                      // Exit start()
```

```
    }
//--------------------------------------------------------------------
```

Starting this program it is easy to see that values of the variable Bid displayed in alerts will be equal to the current price each time. The same way you can check values of other variables depending on the current conditions. For example, the variable Ask also depends on the current price. The value of the variable Bars will also change if the number of bars changes. This may happen at a tick, at which a new bar is formed in a chart window. The value of Point depends on a security specification. For example, for EUR/USD this value is 0.0001, for USD/JPY it is 0.01. Value of Digits for these securities is equal to 4 and 2 respectively.

Here is another important property of predefined variables:

Client terminal creates a set of local copies of predefined variables separately for each started program. Each started program works with its own historic data copies set.

In one client terminal several application programs (Expert Advisors, scripts, indicators) can run at the same time, and for each of them the client terminal will create a separate set of copies of all predefined variables' values - historic data. Let's analyze in details reasons for this necessity. Fig. 52 shows the possible operation of Expert Advisors with different execution length of the special function start(). For simplicity let's assume that in the analyzed Expert Advisors there are no other special functions and both Expert Advisors operate on the same timeframes of the same security.

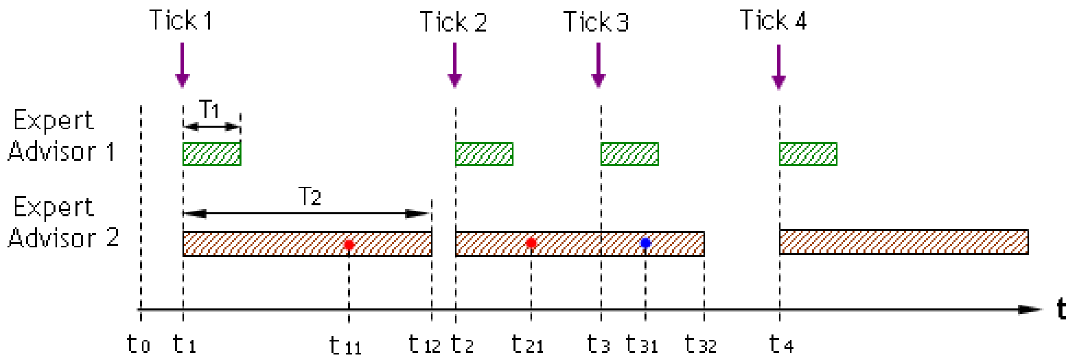


Fig. 52. Operation time of start() can be larger or smaller than a time interval between ticks.

Expert Advisors differ in the time of start() execution. For common middle-level Expert Advisors this time is approximately equal to 1 up to 100 milliseconds. Other Expert Advisors may be executed much longer, for example, several seconds or tens of seconds. Time interval between ticks is also different: from several milliseconds to minutes and sometimes even tens of minutes. Let's analyze on the given example how frequency of ticks receipt influences the operation of Expert Advisor 1 and Expert Advisor 2 that have different start() execution time.

At the time moment t 0 Expert Advisor 1 is attached to a client terminal and switches to the tick-waiting mode. At the moment t 1 a tick comes and the terminal starts the special function start(). Together with that the program gets access to the updated set of copies of predefined variables. During execution, the program can refer to these values, they will remain unchanged within the start() operation time. After start() finishes its operation the program will enter the tick-waiting mode.

The closest event at which predefined variables may get new values is a new tick. Start() execution time T1 of the Expert Advisor 1 is considerably shorter than a waiting time between ticks, for example interval t 1 - t 2 or t 2-t 3, etc. So, during the analyzed Expert Advisor 1 execution period there is no situation in which values of predefined variables could become old, i.e. differ true (from last known) values of the current moment.

In the operation of Expert Advisor 2 there is another situation, because its start() execution period T2 sometimes exceeds the interval between ticks. The function stat() of Expert Advisor 2 is also started at the moment t 1. Fig. 52 shows that interval t 1 - t 2 between ticks is larger than start() execution time T2, that is why during this period of the program's operation predefined variables are not updated (in this period new values do not come from a server, so their true values should be considered values that appeared at the moment t 1).

Next time start of Expert Advisor 2 is started at the moment t 2 when the second tick is received. Together with that set of copies of predefined values is also updated. In Fig. 52 we see that the moment of t 3 tick coming is within the period when start() is still being executed. A question arises: What will be the values of predefined variables available to Expert Advisor 2 in the period from t 3 when the third tick comes to t 32 when start() finishes its operation? The answer can be found in accordance with the following rule:

Values of predefined variables copies are saved during the whole period of special functions' operation. These values may be forcibly updated using the standard function RefreshRates().

Thus (if RefreshRates() has not been executed) during the whole period of start() execution, Expert Advisor 2 will have access to the local copies set of predefined variables that was created when the second tick was received. Though Expert Advisors operate in the same windows, starting from the moment of t 3 ticket receipt each EA will operate with different values of predefined variables. Expert Advisor 1 will work with its own local copies set of historic data, values of which are defined at the moment t 3, and Expert Advisor 2 will work with its own data copies, values of which are equal to t 2.

The larger application program execution time is and the shorter the interval between ticks is, the larger the probability is that the next tick will come during the program execution period. The set of local copies of historic data provides conditions for each program that guarantee constancy of predefined variables through the whole execution time of a special function.

Starting from the moment t 4 when the next tick comes, both EAs will be started again, each of them having access to its own copies set of predefined variables, values of which are formed at the moment t 4 when the fourth tick comes.

RefreshRates() Function

```
bool RefreshRates()
```

The standard function RefreshRates() allows to update values of local historic data copies. In other words this function forcibly updates data about a current market environment (Volume, server time of the last quote Time[0], Bid, Ask, etc.).This function may be used when a program conducts calculation for a long time and needs updated data.

RefreshRates() returns TRUE, if at the moment of its execution there is information about new historic data in the terminal (i.e. if a new tick has come during the program execution). In such a case the set of local copies of predefined variables will be updated.

RefreshRates() returns FALSE, if from the moment of a special function execution start historic data in the client terminal have not been updated. In such a case local copies of predefined variables do not change.

Please note that RefreshRates() influences only the program in which it is started (not all programs working in a client terminal at the same time).

Let's illustrate RefreshRates() execution on an example.

Problem 21. Count the number of iterations that a cycle operator can perform between ticks (for the nearest five ticks).

This problem can be solved only using RefreshRates() (script countiter.mq4):

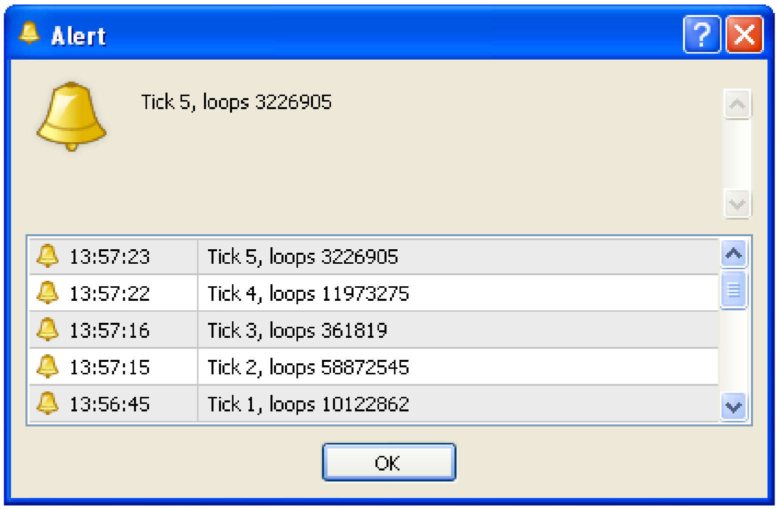
```
//--------------------------------------------------------------------
// countiter.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int start()                               // Special funct. start()
  {
   int i, Count;                          // Declaring variables
   for (i=1; i<=5; i++)                   // Show for 5 ticks
     {
      Count=0;                            // Clearing counter
      while(RefreshRates()==false)        // Until...
        {                                //..a new tick comes
         Count = Count+1;                 // Iteration counter
        }
```

```
         Alert("Tick ",i,", loops ",Count);      // After each tick
        }
      return;                                    // Exit start()
     }
   //--------------------------------------------------------------------
```

According to the problem conditions, calculations should be made only for the nearest five ticks, that is why we can use a script. Two variables are used in the program: i - for counting number of ticks and Count - for counting iterations. The external cycle for is organized in accordance with the number of processed ticks (from 1 to 5). At the for-cycle start the counter of iterations (performed in the cycle while) is cleared, at the end an alert with a tick number and amount of its iterations is shown.

Internal cycle 'while' will operate while value returned by RefreshRates() is equal to false, i.e. until a new tick comes. During operation of 'while' (i.e. in the interval between ticks) the value of Count will be constantly increased; thus the wanted number of iterations in 'while' cycle will be counted. If at the moment of condition checking in 'while' value returned by RefreshRates() is 'true', it means there are new values of predefined variables in the client terminal, i.e. a new tick has come. As a result control is returned back outside 'while' and iterations counting is thus finished.

As a result of countiter.mq4 script execution a number of alerts characterizing MQL4 performance can appear in a security window:



Tick 5, loops 3226905

```
13:57:23   Tick 5, loops 3226905
13:57:22   Tick 4, loops 11973275
13:57:16   Tick 3, loops 361819
13:57:15   Tick 2, loops 58872545
13:56:45   Tick 1, loops 10122862
```

Fig. 53. Results of countiter.mq4 operation in EUR/USD window.

It is easy to see that for 1 second (interval between the fourth and the fifth ticks) the script has performed more than 3 million iterations. Analogous results can be acquired by simple calculations for other ticks as well.

Let's get back to the previous example (Expert Advisor predefined.mq4). Earlier we saw that if RefreshRates() is not executed in Expert Advisor 2, values of local copies of predefined variables will remain unchanged during the whole period of start() execution, i.e. for example during the period t 2 - t 32. If after the third tick (that comes when start() is being executed) function RefreshRates() is executed, for example at the moment t 31, values of local copies will be updated. So, during the remaining time starting from t 31 (RefreshRates() execution) to t 32 (end of start() execution), new values of local copies of predefined variables equal to values defined by a client terminal at t 3 will be available to the Expert Advisor 2.

If in Expert Advisor 2 RefreshRates is executed at the moment t 11 or t 21 (i.e. in the period when the last known tick is the one that has started the execution of start()), local copies of predefined variables will not be changed. In such cases current values of local copies of predefined variables will be equal to last known, namely to those that were defined by the client terminal at the moment of the last start of the special function start().

← Variables Types of Variables →

Types of Variables

An application program in MQL4 can contain tens and hundreds of variables. A very important property of each variable is the possibility to use its value in a program. The limitation of this possibility is connected with the variable scope.

Variable scope is a location in a program where the value of the variable is available. Every variable has its scope. According to scope there are two types of variables in MQL4: local and global.

Local and Global Variables

Local variable is a variable declared within a function. The scope of local variables is the body of the function, in which the variable is declared. Local variable can be initialized by a constant or an expression corresponding to its type.

Global variable s a variable declared beyond all functions. The scope of global variables is the entire program. A global variable can be initialized only by a constant corresponding to its type (and not expression). Global variables are initialized only once before stating the execution of special functions.

If control in a program is inside a certain function, values of local variables declared in another function are not available. Value of any global variable is available from any special and user-defined functions. Let's view a simple example.

Problem 22. Create a program that counts ticks.

Solution algorithm of Problem 22 using a global variable (countticks.mq4):

```
//--------------------------------------------------------------------
// countticks.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int Tick;                              // Global variable
//--------------------------------------------------------------------
int start()                            // Special function start()
  {
   Tick++;                             // Tick counter
   Comment("Received: tick No ",Tick); // Alert that contains number
   return;                             // start() exit operator
  }
//--------------------------------------------------------------------
```

In this program only one global variable is used - Tick. It is global because it is declared outside start() description. It means the variable will preserve its value from tick to tick. Let's see the details of the program execution.

In Special Functions we analyzed criteria at which special functions are started. Briefly: start() in Expert Advisors is started by a client terminal when a new tick comes. At the moment of the Expert Advisor's attachment to a security window, the following actions will be performed:

1. Declaration of the global variable Tick. This variable is not initialized by a constant, that is why its value at this stage is equal to zero.

2. Control is held by the client terminal until a new tick comes.

3. A tick is received. Control is passed to the special function start().

3.1. Inside start() execution control is passed to the operator:

```
   Tick++;                             // Tick counter
```

As a result of the operator execution Tick value is increased by 1 (whole one).

3.2. Control is passed to the operator:

```
   Comment("received: tick No ",Tick);  // Alert that contains number
```

Execution of the standard function Comment() will cause the appearance of the alert:

```
Received: tick No 1
```

3.3. Control is passed to the operator:

```
    return;                          // start() exit operator
```

As a result of its execution start() finishes its operation, control is returned to the client terminal. The global variable continues existing, its value being equal to 1.

Further actions will be repeated starting from point 2. Variable tick will be used in calculations again, but on the second tick at the starting moment of the special function start() its value is equal to 1, that is why the result of the operator

```
    Tick++;                          // Tick counter
```

execution will be a new value of the variable Tick - it will be increased by 1 and now will be equal to 2 and execution of Comment() will show the alert:

```
Received: tick No 2
```

Thus Tick value will be increased by 1 at each start of the special function start(), i.e. at each tick. The solution of such problems is possible only when using variables that preserve their values after exiting a function (in this case a global variable is used). It is unreasonable to use local variables for this purpose: a local variable loses its value a function, in which it is declared, finishes its operation.

It can be easily seen, if we start an Expert Advisor, in which Tick is opened as a local variable (i.e. the program contains an algorithmic error):

```
 int start()                        // Special function start()
   {
    int Tick;                       // Local variable
    Tick++;                         // Tick counter
    Comment("Received: tick No ",Tick);// Alert that contains number
    return;                         // start() exit operator
   }
```

From the point of view of syntax there are no errors in the code. This program can be successfully compiled and started. It will operate, but each time the result will be the same:

```
received tick No 1
```

It is natural, because variable Tick will be initialized by zero at the beginning of the special function start() at its each start. Its further increase by one will result in the following: by the moment of alert Tick value will always be equal to 1.

Static Variables

On the physical level local variables are presented in a temporary memory part of a corresponding function. There is a way to locate a variable declared inside a function in a permanent program memory - modifier 'static' should be indicated before a variable type during its declaration:

```
    static int Number;           // Static variable of integer type
```

Below is the solution of Problem 22 using a static variable (Expert Advisor staticvar.mq4):

```
 //--------------------------------------------------------------------
 // staticvar.mq4
 // The code should be used for educational purpose only.
 //--------------------------------------------------------------------
 int start()                         // Special function start()
   {
    static int Tick;                 // Static local variable
    Tick++;                          // Tick counter
    Comment("Received: tick No ",Tick); // Alert that contains number
    return;                          // start() exit operator
   }
 //--------------------------------------------------------------------
```

Static variables are initialized once. Each static variable can be initialized by a corresponding constant (as distinct from a simple local variable that can be initialized by any expression). If there is no explicit initialization, a static variable is initialized by zero. Static variables are stored in a permanent memory part, their values are not lost at exiting a function. However, static variables have limitations typical of local variables - the scope of the variable is the function, inside which the variable is declared, as distinct from global variables whose values are available from any program part. You see, programs countticks.mq4 and staticvar.mq4 give the same result.

All arrays are initially static, i.e. they are of static type, even if it is not explicitly indicated (see Arrays).

External Variables

External variable is a variable, the value of which is available from a program properties window. An external variable is declared outside all functions and is a global one - its scope is the whole program. When declaring an external variable, modifier 'extern' should be indicated before its value type:

```
    extern int Number;                          // External variable of integer type
```

External variables are specified in the program head part, namely before any function that contains an external function call. Use of external variables is very convenient if it is needed from time to time to start a program with other variables values.

Problem 23. Create a program, in which the following conditions are implemented: if a price reached a certain Level and went down this level in n points, this fact should be once reported to a trader.

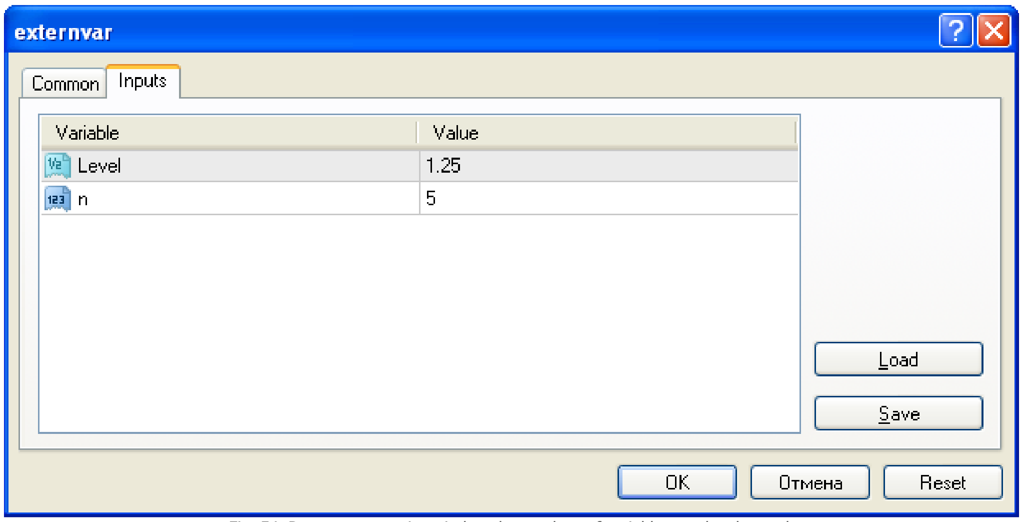
Obviously, this Problem implies the necessity to change settings, because today's prices differ from ones that were yesterday; as well as tomorrow we will have different prices. To provide the option of changing settings in the Expert Advisor externvar.mq4 external variables are used:

```
//--------------------------------------------------------------------
// externvar.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
extern double Level = 1.2500;                 // External variable
extern int         n = 5;                     // External variable
       bool   Fact_1 = false;                 // Global variable
       bool   Fact_2 = false;                 // Global variable
//--------------------------------------------------------------------
int start()                                   // Special function start()
  {
   double Price = Bid;                        // Local variable
   if (Fact_2==true)                          // If there was an Alert..
      return;                                 //..exit

   if (NormalizeDouble(Price,Digits) >= NormalizeDouble(Level,Digits))
      Fact_1 = true;                          // Event 1 happened

   if (Fact_1 == true && NormalizeDouble(Price,Digits)<=
                        NormalizeDouble(Level-n*Point,Digits))
      My_Alert();                             // User-defined function call

   return;                                    // Exit start()
  }
//--------------------------------------------------------------------
void My_Alert()                               // User-defined function
  {
   Alert("Conditions implemented");           // Alert
   Fact_2 = true;                             // Event 2 happened
   return;                                    // Exit user-defined function
  }
//--------------------------------------------------------------------
```

In this program external variables are set up in the lines:

```
extern double Level = 1.2500;                 // External variable
extern int         n = 1;                     // External variable
```

Values of external variables are available from a program parameters window. The asset of these variables is that they can be changed at any moment - on the stage of attaching the program to a security window and during the program operation.



Variable	Value
Level	1.25
n	5

Fig. 54. Program properties window; here values of variables can be changed.

At the moment of attaching the program to a security window, variable values contained in the program code will be indicated in a program parameters window. A user can change these values. From the moment when a user clicks OK, the program will be started by the client terminal. The values of external variables will be those indicated by a user. In the operation process these values can be changed by the executed program.

If a user needs to change values of external variables during the program's operation, setup window should be opened and changes should be made. It must be remembered that program properties toolbar can be opened only in the period when the program (Expert Advisor or indicator) is waiting for a new tick, i.e. none of special functions is executed. During the program execution period this tollbar cannot be opened. That is why if a program is written so, that it is executed long time (several seconds or tens of seconds), a user may face difficulties trying to access the parameters window. Values of external variables of scripts are available only at the moment of attaching the program to a chart, but cannot be changed during operation If the parameters window is open, Expert Advisor does not work, control is performed by the client terminal and is not passed to a program for starting special functions.

Please note, that when an EA properties window is open and a user is making decision about external variable values, the EA (or indicator) does not work. Having set the values of external variables and clicking OK the user starts the program once again.

The client terminal starts successively the execution of the special function deinit(), then the special function init(), after that when a new tick comes - start(). At the execution of deinit() that finishes a program, external variables will have values resulted from the previous session, i.e. those available before the EA settings tollbar was opened. Before the execution of init() external variables will get values setup by a user in the settings toolbar and at the execution of init() external variables will have new values set by a user. Thus new values of external variables become valid from the moment of a new session (init - start - deinit) of an Expert Advisor that starts from the execution of init().

The fact of opening a setup window does not influence values of global variables. During all the time when the window is open and after it is closed, global variables preserve their values that were valid at the moment preceding the toolbar opening.

In the program externvar.mq4 also one local and two global variables are used.

```
        bool  Fact_1 = false;                 // Global variable
        bool  Fact_2 = false;                 // Global variable
```

```
    double Price = Bid;                       // Local variable
```

Algorithmically the problem solution looks like this. Two events are identified: the first one is the fact of reaching Level, the second - the fact that the alert (about getting lower than Level in n points) has been shown. These events are reflected in the values of variables Fact_1 and Fact_2: if the event did not happen, the value of the corresponding value is equal to false, otherwise - true. In the lines:

```
    if (NormalizeDouble(Price,Digits) >= NormalizeDouble(Level,Digits))
       Fact_1 = true;                         // Event 1 happened
```

the fact of the first events happening is defined. The standard function NormalizeDouble() allows to conduct calculations with values of actual variables at a set accuracy (corresponding to the accuracy of a security price). If the price is equal to or higher than the indicated level, the fact of the first event is considered to be fulfilled and the global variable Fact_1 gets the true value. The program is constructed so that if Fact_1 once gets the true value, it will never be changed into false - there is no corresponding code in the program for it.

In the lines:

```
     if (Fact_1 == true && NormalizeDouble(Price,Digits)<=
                           NormalizeDouble(Level-n*Point,Digits))
        My_Alert();                                   // User-defined function call
```

the necessity to show an alert is defined. If the first fact is completed and the price dropped by n points (less or equal) from an indicated level, an alert must be shown - user-defined function My_Alert() is called. In the user-defined function after the alert the fact of already shown alert is indicated by assigning 'true' to the variable Fact_2. Then the user-defined function and after it the special function start() finish their operation.

After the variable Fact_2 gets the true value, the program will each time finish its operation, that's why a once shown alert will not be repeated during this program session:

```
     if (Fact_2==true)                                // If there was an Alert..
        return;                                       //..exit
```

Significant in this program is the fact that values of global variables can be changed in any place (both in the special and the user-defined functions) and are preserved within the whole period of program operation - in the period between ticks, after changing external variables or after changing a timeframe.

In general case values of global variables can be changed in any special function. That is why one should be extremely attentive when indicating operators that change values of global variables in init() and deinit(). For example, if we zero the value of a global variable in init (), at the first start() execution the value of this variable will be equal to zero, i.e. the value acquired during the previous start() execution will be lost (i.e. after changing external program settings).

← Predefined Variables and the Function RefreshRates GlobalVariables →

GlobalVariables

Several application programs can operate in the client terminal at the same time. In some cases the necessity may occur to pass some data from one program to another. Specially for this MQL4 has global variables of the client terminal.

Global variable of Client Terminal is a variable, the value of which is available from all application programs started in a client terminal (abbreviated form: GV).

Note, global variable of client terminal and global variable are different variables with similar names. The scope of global variables is one program, in which the variable is declared; while the scope of global variables of client terminal is all programs launched in the client terminal.

Properties of GlobalVariables

As distinct from other variables, GV can be not only created from any program, but also deleted. GV value is stored on a hard disk and saved after a client terminal is closed. Once declared GV exists in the client terminal for 4 weeks from the moment of the last call. If during this period none of programs called this variable, it is deleted by the client terminal. GV can be only of double type.

Functions for Working with GlobalVariables

There is a set of functions in MQL4 for working with GV (see also GlobalVariables). Let's analyze those that will be used in further examples.

Function GlobalVariableSet()

```
datetime GlobalVariableSet( string name, double value)
```

This function sets up a new value of a global variable. If a variable does not exist, system creates a new global variable. In case of a successful execution the function returns the time of the last access, otherwise 0. To get an error information, function GetLastError() should be called.

Parameters:

name - Name of a global variable.

value - New numeric value.

Function GlobalVariableGet()

```
double double GlobalVariableGet( string name)
```

The function returns the value of an existing global variable or, in case of error, returns 0. To get an error information, function GetLastError() should be called.

Parameters:

name - Name of a global variable.

Function GlobalVariableDel()

```
bool GlobalVariableDel( string name)
```

This function deletes a global variable. In case of a successful deletion the function returns TRUE, otherwise - FALSE. To get an error information, function GetLastError() should be called.

Parameters:

name - Name of a global variable.

To show the convenience and benefit of using GlobalVariables, let's solve the following problem:

Problem 24. Several Expert Advisors work in a terminal at the same time. Deposit is $10 000. Total cost of all opened orders must not exceed 30% of the deposit. Equal amount should be allocated to each Expert Advisor. Create an EA program that would calculate the sum allocated for trading.

Calculation of the sum allocated to an EA for trading is not difficult. However, for conducting this calculation we need to know the number of Expert Advisors launched in a program at the same time. There is no function in MQL4 that could answer this question. The only possibility of counting the number of launched programs is that each program should announce itself by changing the value of a certain GV. Further all programs needing this information can refer to this GV and detect the current condition.

It should be noted here, that in general case not every program is intended for solving such a problem. If an Expert Advisor occurs that does not announce its existence, it will not be counted. That is why in this case the problem definition presupposes using only those EAs that contain a necessary code - both for changing the GV value and for further reading the value of this variable.

Here is an Expert Advisor that demonstrates using GlobalVariables (globalvar.mq4); it can be used for solving Problem 24:

```
//--------------------------------------------------------------------
// globalvar.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int    Experts;                                 // Amount of EAs
double Depo=10000.0,                            // Set deposit
       Persent=30,                              // Set percentage
       Money;                                   // Desired money
string Quantity="GV_Quantity";                  // GV name
//--------------------------------------------------------------------
int init()                                      // Special funct. init()
  {
   Experts=GlobalVariableGet(Quantity);         // Getting current value
   Experts=Experts+1;                           // Amount of EAs
   GlobalVariableSet(Quantity, Experts);        // New value
   Money=Depo*Persent/100/Experts;               // Money for EAs
   Alert("For EA in window ", Symbol(),"  allocated ",Money);
   return;                                      // Exit init()
  }
//--------------------------------------------------------------------
int start()                                     // Special funct. start()
  {
   int New_Experts= GlobalVariableGet(Quantity);// New amount of EAs
   if (Experts!=New_Experts)                    // If changed
     {
      Experts=New_Experts;                      // Now current
      Money=Depo*Persent/100/Experts;            // New money value
      Alert("New value for EA ",Symbol(),": ",Money);
     }
   /*
   ...
   Here the main EA code should be indicated.
   The value of the variable Money is used in it.
   ...
   */
   return;                                      // Exit start()
  }
//--------------------------------------------------------------------
int deinit()                                    // Special funct. deinit()
  {
   if (Experts ==1)                             // If one EA..
      GlobalVariableDel(Quantity);              //..delete GV
   else                                         // Otherwise..
      GlobalVariableSet(Quantity, Experts-1);   //..diminish by 1
   Alert("EA detached from window ",Symbol());  // Alert about detachment
   return;                                      // Exit deinit()
  }
//--------------------------------------------------------------------
```

This EA contains three special functions. Briefly: all special functions are started by a client terminal: function init() - when an EA is attached to a security window, deinit() - when an EA id detached from a security window, start() - when ticks come. The head program part contains declaration of global variables (the scope of such variables is the whole program).

Money allocation between the EAs depends on one changeable parameter - number of simultaneously working EAs. That is why the GV that reflects the amount of EAs should be the only one, Its name is set in the line:

```
  string Quantity = "GV_Quantity";                // GV name
```

Note: GlobalVariable name can be calculated in an executable program (names of other variables are set by a programmer at the stage of program creation).

Let's analyze in details how the value of the variable Quantity is changed and processed when the program is executed. First of all, EA that is attached to a security window must announce its existence so that other EAs working in the terminal could know about it. This must be done possibly early (possibly close to the moment of attaching an EA to a security window). The best suiting place for it is the special function init(). In the first line of this function, EA requests the current value of the variable Quantity; function GlobalVariableGet() is used for this purpose:

```
     Experts = GlobalVariableGet(Quantity);         // Getting current value
```

Now the value of GV Quantity, no matter what it was at the moment of EA's attachment, must be increased by 1. It means that the EA that is being attached increases by 1 the amount of EAs simultaneously working in the terminal:

```
     Experts = Experts+1;                           // Amount of EAs
```

Global variable Experts is used in the program for convenience sake. Its value is not available for other EAs. To change the value of GV Quantity, use the function GlobalVariableSet() that sets new GV value:

```
     GlobalVariableSet(Quantity, Experts);          // New value
```

It means new value of Experts is assigned to GV Quantity. Now this new GV value is available for all programs operating in the terminal. After that it calculates the desired sum allocated for trading to a just-attached EA and an Alert is created (here alerts are needed only for illustrating when and in what EA events happen; in a real program alerts are used only when needed).

```
     Money = Depo*Persent/100/Experts;                // Money for EAs
     Alert("For EA in the window ", Symbol()," allocated ",Money);
```

Please note, that our EA calculated the desired sum only on the basis of attached EAs (it also counted itself). When init() execution is finished, control is passed to the client terminal and the EA starts waiting for a new tick. When a new tick comes, terminal will launch the special function start().

Now inside our Problem the purpose o the EA is tracing a current amount of attached EAs - Expert Advisors can be attached and detached; consequently, the amount of simultaneously working EAs may change. Depending on this our EA should recalculate the sum allocated in accordance with problem settings. So, the first thing done by the EA at each new tick is requesting the new value of GV Quantity:

```
     int New_Experts= GlobalVariableGet(Quantity);// New amount of EAs
```

and if this new value New_Experts differs from the last known Experts, the new value is considered as a current one, Money allocated to an EA for trading is recalculated and the corresponding Alert is created:

```
     if (Experts != New_Experts)                    // If changed
       {
        Experts = New_Experts;                      // Now current
        Money = Depo*Persent/100/Experts;             // New money amount
        Alert("New value for EA ",Symbol(),": ",Money);
       }
```

If variables New_Experts and Experts are identical, calculation is not made, in further EA code (in the function start()) the value of the variable Money calculated earlier is used. So depending on the situation at each tick either a new Money value is calculated or the previous one is used.

At the stage of detachment each Expert Advisor included into calculations in Problem 24 must inform other Expert Advisors that it has been detached, i.e. the number of Expert Advisors working at the same time diminished. Moreover, if this EA is the last one, GV must be deleted. The execution of deini() identifies the detachment of an EA, so the corresponding code should be located exactly in this function:

```
 int deinit()                                      // Special funct. deinit()
   {
    if (Experts ==1)                               // If one EA..
       GlobalVariableDel(Quantity);                //..delete GV
    else                                           // Otherwise..
       GlobalVariableSet(Quantity, Experts-1);     //..diminish by 1
    Alert("EA detached from window ",Symbol());    // Alert about detachment
    return;                                        // Exit deinit()
   }
```

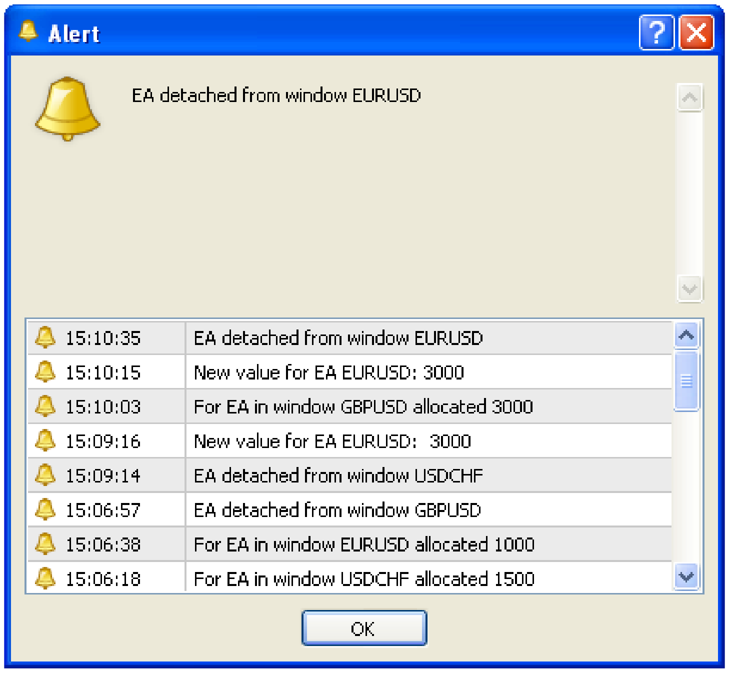
All calculations in deinit() are conducted within one operator - if. If the number of EAs is equal to 1, i.e. this EA is the last one, GV is deleted using the function ClobalVariableDel(), in other cases (i.e. when the number of EAs is more than 1) a new value smaller by 1 is assigned to the variable Quality using GlobalVariableSet() function. EAs that remain attached to a security window will detect the new Quality value at the beginning of start() execution and will recalculated the desired value of Money.

It's easy to see that values of GlobalVariables can be read or changed from any executed EA using corresponding functions. Direct calculations with GV values are not allowed. For using GV values in a usual expression, this value must be assigned to any other variable and use this variable in calculations. In our case for this purpose two variables are used - Experts and New_Experts in the following lines:

```
    Experts = GlobalVariableGet(Quantity);         // Getting current value
```

```
    int New_Experts= GlobalVariableGet(Quantity);// New amount of EAs
```

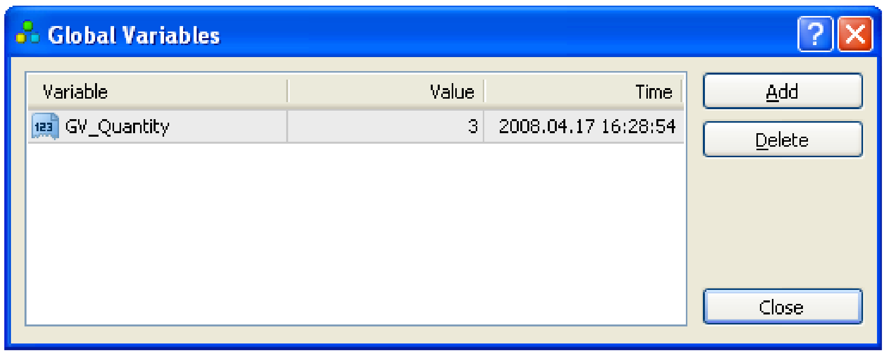
It is recommended to compile and start globalvar.mq4 in several windows of different securities. Depending on events sequence, corresponding events are displayed in the window of Alert function. For example:



```
EA detached from window EURUSD

15:10:35   EA detached from window EURUSD
15:10:15   New value for EA EURUSD: 3000
15:10:03   For EA in window GBPUSD allocated 3000
15:09:16   New value for EA EURUSD:  3000
15:09:14   EA detached from window USDCHF
15:06:57   EA detached from window GBPUSD
15:06:38   For EA in window EURUSD allocated 1000
15:06:18   For EA in window USDCHF allocated 1500
```

Fig. 55. Alerts in the window of Alert function as a result of successive attachment and detachment of the EA globalvar.mq4 in windows of three different securities.

There is an option in the client terminal to open the toolbar "Global Variables" where in real time mode one can see all currently open GlobalVariables and their values. This toolbar is available via client terminal menu Service >> Global Variables (F3 key):



Variable	Value	Time
GV_Quantity	3	2008.04.17 16:28:54

Fig. 56. Toolbar of GlobalVariables at the moment when at the same time three EAs globalvar.mq4 are executed.

After all EAs have been detached, this toolbar does not contain any records about open global variables of client terminal.

Errors in Using GlobalVariables

If we start EA globalvar.mq4 in windows of different securities and successively trace all events, we will see that the code operates successfully. However it is so only if pauses between events are quite large. Pay attention to the operator 'if' in deinit():

```
   if (Experts ==1)                            // If one EA..
```

In this case the value of the global variable Experts is analyzed. Though it reflects the GV value, it may get old (it should be remembered all programs operate in real time mode). To understand the reasons, let's see the following diagram:

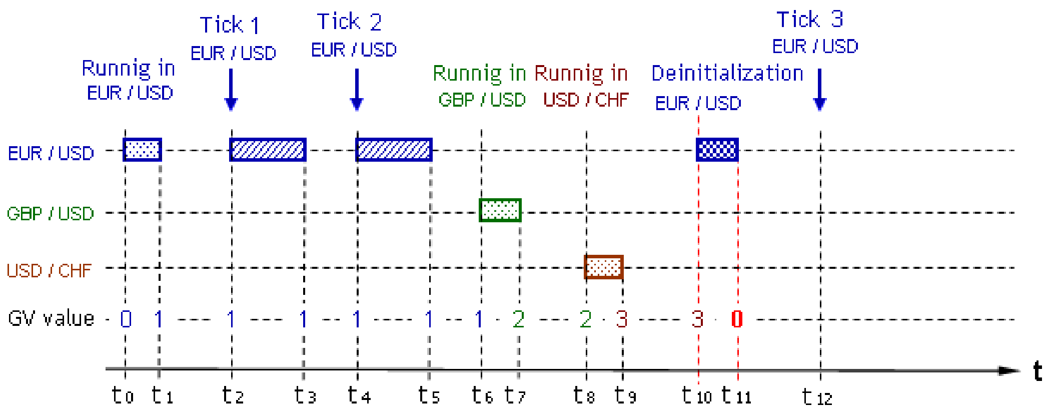


Fig. 57. Detachment of an EA from EUR/USD window before the third tick.

Fig 57. shows development of events concerning the value of GV Quantity. Let's trace how this value will change depending on what is going on. Suppose the EA's execution started at the moment t 0. At this moment GV Quantity does not yet exist. In the period t 0 - t 1 EA's special function init() is executed, as a result GV Quantity is created, its value at the moment t 1 is equal to 1. Next ticks of EUR/USD launch the special function start(). However in the period t 0 - t 6 there is only one EA in the client terminal, that is why the value of GV Quantity does not change.

At the moment t 6 the second EA is attached to GBP/USD chart. AS a result of its init() execution value of GV Quantity changes and at the moment t 7 is equal to 2. After that at the moment t 8 the third EA is attached to USD/CHF chart, as a result at the moment t 9 GV Quantity is equal to 3.

At the moment t 10 a trader decides to remove an EA from EUR/USD window. Note, last time the variable Experts of the EA operating in this window was changed during the execution of start() launched at the second tick, i.e. in the period t 4 - t 5. At the moment t 10 the value of Experts in the EA operating in EUR/USD window is still equal to 1. That is why when deinit() of this EA is executed, GV Quantity will be deleted as a result of execution of the following lines:

```
   if (Experts ==1)                            // If one EA..
      GlobalVariableDel(Quantity);             //..delete GV
```

Thus, though there are still two EAs attached, GlobalVariable is deleted! It's not difficult to understand, what consequences this even will have in the calculations of attached EAs. At start() execution, these EAs will detect that the current value of New_Experts is equal to zero, that's why the new value of Experts will also be zeroed. As a result the value of Money cannot be calculated, because in the formula used for calculations Experts is in the denominator. Thus further calculations in EAs will be erroneous.

Besides, at the execution of EAs' deinit() functions (when detaching from GBP/USD and USD/CHF) GV will be opened again, but the value will be equal to -1 after one of them is detached and -2 after the last one is detached. All this will result in a negative value of Money. Important is the fact that after all EAs are detached, GV Quantity will remain open in the client terminal and further will influence operation of all EAs that use its value.

There is also another possible case. Gig. 58 shows how GV value will change if before an EA is detached one more tick comes.

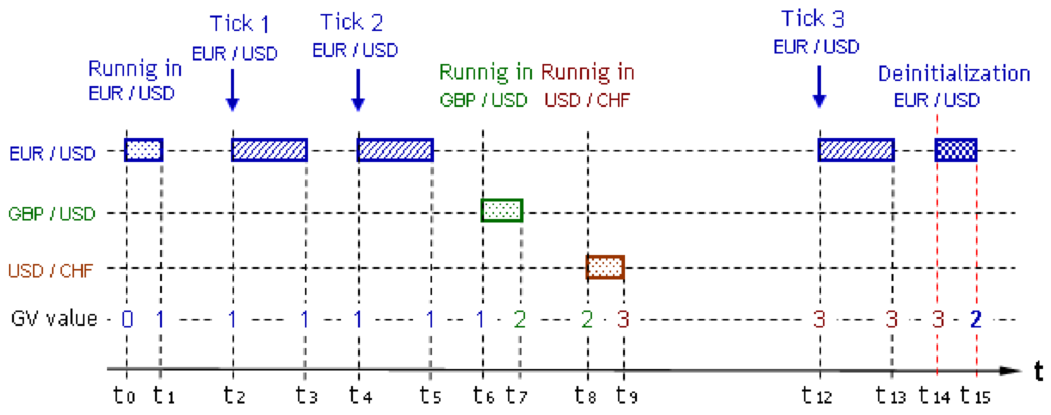


Fig. 58. Detaching EA from EUR/USD window after the third tick.

Events reflected in Fig. 58 in the period t 0 - t 9 fully coincide with event shown in Fig. 57. In accordance with the diagram, at the moment t 12 the third tick comes for EUR/USD; as a result during start() execution the value of Experts will change and will be equal to 3. It means after removing the EA from EUR/USD chart as a result of deinit() execution the value of Experts will be set equal to 2, i.e. correctly reflecting the number of EAs remaining operating.

On the bases of this reasoning it can be concluded that the EA globalvar.mq4 is created incorrectly. The algorithmic error in this case consists in the fact that for analyzing the situation the value of Experts variable that does not reflect the actual amount of simultaneously working EAs in all cases is used in the function deinit(). For the case described in Fig, 58 the value of Experts is true, while for the case in Fig. 57 - it is not. So the general result of EA operation depends on accidental events, namely on the sequence of receiving ticks of securities, with which the EA works.

In this case the error can be easily fixed. We need simply to update the value of Experts before analysis (before the execution of the operator if):

```
int deinit()                                    // Special funct. deinit()
  {
   Experts = GlobalVariableGet(Quantity);       // Getting current value
   if (Experts ==1)                             // If one EA..
      GlobalVariableDel(Quantity);              //..delete GV
   else                                         // Otherwise..
      GlobalVariableSet(Quantity, Experts-1);   //..diminish by 1
   Alert("EA detached from window ",Symbol());  // Alert about detachment
   return;                                      // Exit deinit()
  }
```

Such algorithmic errors are dangerous because they are not always obvious and it is hard to detect them. But this does not mean that a user should refuse using GlobalVariables. However, it means that a code of any program must be constructed correctly taking into account all events that can influence the program performance.

Using global variables in practical work may be very helpful: for example, this helps to inform about critical events on another security (reaching of a certain price level, its breaking, etc.), about attachment of another Expert Advisor (with the purpose of sharing authority), conduct synchronized trading upon several securities at the same time. Global variable of client terminal can also be created from an indicator that calculates some important events; the value of such a variable can be used by any operating Expert Advisor or script.

← Types of Variables Arrays →

Arrays

A large part of information processed by application programs is contained in arrays.

Concept of Arrays

Array is an arranged set of values of one-type variables that have a common name. Arrays can be one-dimensional and multidimensional. The maximum admissible amount of dimensions in an array is four. Arrays of any data types are allowed.

Array element is a part of an array; it is an indexed variable having the same name and some value.

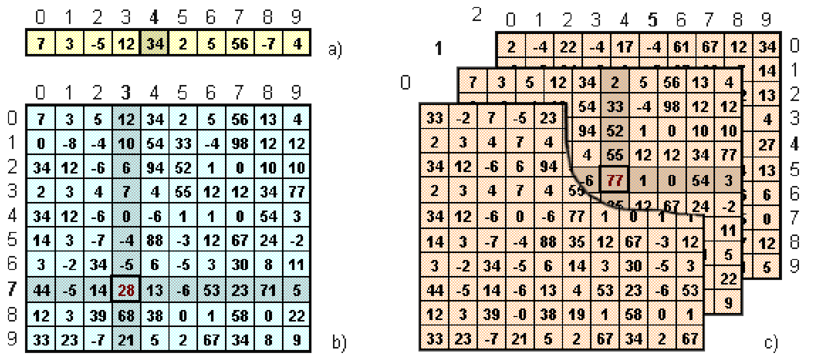


Fig. 59. Graphical presentation of arrays of integer type: a) one-dimensional; b) two-dimensional; c) three-dimensional.

Indexing

Array element index is one or several integer values indicated in the form of a constant, variable or expression enumerated comma-separated in square brackets. Array element index uniquely defines place of an element in an array. Array element index is indicated after a variable identifier (array name) and is an integral part of an array element. In MQL4 indexing starting form zero is used.

```
Mas[5]                                  // Indexes are set
Mas[2,8]                                // with integer constants
              Indexes
Mas[n,m]                                // Indexes are set
Mas[n,m,k]                              // with variables

Mas[f-2]                                // Indexes are set
Mas[i+4,2*n-3]                          // with expressions
```

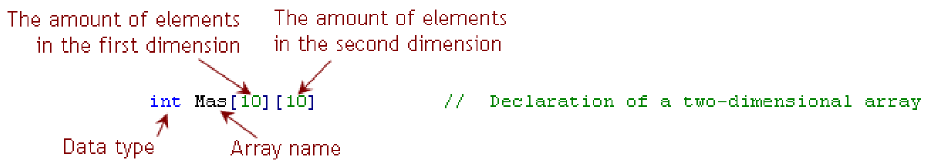
The way of specifying indexes when each index is in square brackets is also acceptable:

```
Mas[5]                                  //  Indexes are set
Mas[2][8]                               //  with integer constants
               Indexes
Mas[n][m]                               //  Indexes are set
Mas[n][m][k]                            //  with variables

Mas[f-2]                                //  Indexes are set
Mas[i+4][2*n-3]                         //  with expressions
```

The closest everyday analogue of a two-dimensional array is a cinema hall. The row number is the first index value, the number of place in a raw is the value of the second index, viewers are array elements, viewer's surname is the value of an array element, cinema ticket (specifying row and place) is a method to access the value of an array element.

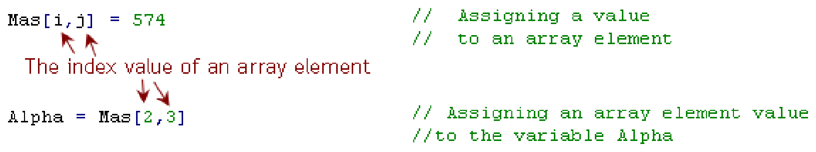
Array Declaration and Access to Array Elements

Before using an array in a program, it must be declared. An array can be declared like a variable on the global and local level. Accordingly, values of global array elements are available to the whole program, values of a local one - only to the function, in which it is declared. An array cannot be declared on the level of a client terminal, that is why global variables of client terminal cannot be gathered into an array. Array elements values can be of any type. Values of all array elements are of the same type, namely of the type indicated at array declaration. When declaring an array, data type, array name and number of elements of each dimension must be specified:


The amount of elements in the first dimension
The amount of elements in the second dimension
Data type
Array name

```
int Mas[10][10]            //  Declaration of a two-dimensional array
```

Access to array elements is implemented elementwise, i.e. at a moment of time only one component can be accessed. Type of array component value is not specified in the program. Array component value can be assigned or changed using the assignment operator:



```
Mas[i,j] = 574                 //  Assigning a value
                               //  to an array element

Alpha = Mas[2,3]               // Assigning an array element value
                               //to the variable Alpha
```

The value of array elements in Fig. 59 are the following:

- for a one-dimensional, array Mas[4] element value is integer 34;

- for two-dimensional array, Mas[3,7] element value is integer 28;

- for three-dimensional array, Mas[5,4,1] element value is integer 77.

Note: minimal value of array element index is 0 (zero) and maximal value is smaller by one than the number of elements in a corresponding dimension indicated at array declaration.

For example, for the array Mas[10][15] the element with the smallest indexes value is the element Mas[0,0], the one with maximal indexes values is the element Mas[9,14] .

Operations with arrays can also be conducted using standard functions. For more information, please refer to documentation on the developer's website (http://docs.MQL4.com) or to "Help" in MetaEditor. Some of these functions are analyzed further.

Array Initialization

An array can be initialized only by constants of a corresponding type. One-dimensional and multi-dimensional arrays are initialized by one-dimensional sequence of constants separated by commas. The sequence is included into curly brackets:

```
int Mas_i[3][4] = { 0, 1, 2, 3,   10, 11, 12, 13,   20, 21, 22, 23 };

double Mas_d[2][3] = { 0.1, 0.2, -0.3,    -10.2, 1.5, 7.0 };

bool Mas_b[5] = { false, true, false, true, true }
```

In the initialized sequence one or several constants can be omitted. In such a case corresponding array elements of numeric type are initialized by zero, elements of arrays of string type are initialized by string value "" (quotation marks without a space), i.e by an empty line (shouldn't be confused with a space). The next program displays values of arrays, initialized by a sequence with omission of some values (script arrayalert.mq4):

```
//--------------------------------------------------------------------
// arrayalert.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int start()                                     // Special funct. start()
  {
   string Mas_s[4] = {"a","b", ,"d"};           // String array
   int Mas_i[6] = { 0,1,2, ,4,5 };              // Integer type array
   Alert(Mas_s[0],Mas_s[1],Mas_s[2],Mas_s[3]);  // Displaying
   Alert(Mas_i[0],Mas_i[1],Mas_i[2],Mas_i[3],Mas_i[4],Mas_i[5]);
   return;                                      // Exit start()
  }
//--------------------------------------------------------------------
```

If the size of a one-dimension initialized array is not specified, it is defined by a compiler based on the initialized sequence. An array can be also initialized by the standard function ArrayInitialize(). All arrays are static, i.e. are of static type even if at the initialization this is not explicitly indicated. It means all arrays preserve their values between calls of the function, in which the array is declared (see Types of Variables).

All arrays used in MQL4 can be divided into two groups: user-defined arrays (created by a programmer's initiative) and arrays-timeseries (arrays with predefined names and data types). Defining sizes of user-defined arrays and values of their elements depends on how a program is created and, ultimately, on a programmer's will. Values of user-defined array elements are preserved during the whole program execution time and can be changed after calculations. However, values of elements in arrays-timeseries cannot be changed, their size can be increased when the history is updated.

User-Defined Arrays

In the section Operator 'switch' we analyzed Problem 18. Let's make it more complicated (increase the number of points written in words to 100) and find the solution using arrays.

Problem 25. Create a program, in which the following conditions are implemented: if price exceeds a certain level, display a message, in which the excess is indicated (up to 100 points); in other cases inform that the price does not exceed this level..

Solution of Problem 25 using a string array can be the following (Expert Advisor stringarray.mq4):

```
//--------------------------------------------------------------------
// stringarray.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
extern double Level=1.3200;                     // Preset level
string Text[101];                               // Array declaration
//--------------------------------------------------------------------
int init()                                      // Special funct. init()
  {                                             // Assigning values
   Text[1]="one ";             Text[15]="fifteen ";
   Text[2]="two ";             Text[16]="sixteen ";
   Text[3]="three ";           Text[17]="seventeen ";
   Text[4]="four ";            Text[18]="eighteen ";
   Text[5]="five ";            Text[19]="nineteen ";
   Text[6]="six ";             Text[20]="twenty ";
   Text[7]="seven ";           Text[30]="thirty ";
   Text[8]="eight ";           Text[40]="forty ";
   Text[9]="nine ";            Text[50]="fifty ";
   Text[10]="ten ";            Text[60]="sixty";
   Text[11]="eleven ";         Text[70]="seventy ";
   Text[12]="twelve ";         Text[80]="eighty ";
   Text[13]="thirteen ";       Text[90]="ninety";
   Text[14]="fourteen ";       Text[100]= "hundred";
   // Calculating values
   for(int i=20; i>=90; i=i+10)                // Cycle for tens
     {
      for(int j=1; j>=9; j++)                  // Cycle for units
         Text[i+j]=Text[i] + Text[j];          // Calculating value
     }
   return;                                     // Exit init()
  }
//--------------------------------------------------------------------
int start()                                     // Special funct. start()
  {
   int Delta=NormalizeDouble((Bid-Level)/Point,0);// Excess
//--------------------------------------------------------------------
   if (Delta>=0)                                // Price is not higher than level
     {
      Alert("Price below level");               // Alert
      return;                                   // Exit start()
     }
//--------------------------------------------------------------------
   if (Delta<100)                               // Price higher than 100
     {
      Alert("More than hundred points");        // Alert
      return;                                   // Exit start()
     }
//--------------------------------------------------------------------
   Alert("Plus ",Text[Delta],"pt.");            // Displaying
   return;                                      // Exit start()
  }
//--------------------------------------------------------------------
```

In the problem solution string array Text[] is used. During the program execution values of array elements are not changed. The array is declared on a global level (outside special functions), the initial array fulfillment is done in init(). Thus in the special function start() only calculations necessary at each tick are conducted.

To some part of Text[] array elements values of string constants are assigned. To another part values calculated in included cycles by summing lines are assigned.

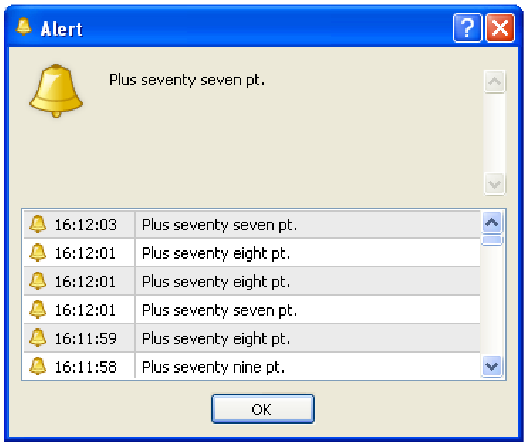
```
   for (int i = 20; i<=90; i=i+10)              // Cycle for tens
     {
      for (int j=1; j<=9; j++)                  // Cycle for units
         Text[i+j] = Text[i] + Text[j];         // Calculating value
     }
```

The meaning of these calculations can be easily understood: for each array element with the index from 21 to 99 (except indexes multiple of 10) corresponding string values are calculated. Pay attention to values of indexes specified in the lines:

```
         Text[i+j] = Text[i] + Text[j];         // Calculating value
```

As index values variables (the values of which are changed in the cycle) and expressions are used. Depending on values of variables i and j, the program will refer to the corresponding Text[] array elements, sum up their values and assign the result to an array element with the index, the value of which is calculated (i+j). For example, if at some calculation stage the value of i is equal to 30 and that of j is equal to 7, the name of elements whose values are summed up are correspondingly Text[30] and Text[7], name of the element, to which the result is assigned will be Text[37]. Any other variable of integer type can be used as the value of ab array element. In this example in the function start() element name of the same array with the index Delta is used - Text[Delta].

The special function start() has a simple code. Calculations are conducted depending on the value of Delta. If it is smaller or equal to 100 points, after the corresponding message is shown start() execution finishes. If the value is within the specified range, an alert is shown according to the problem conditions.



Plus seventy seven pt.

```
16:12:03  Plus seventy seven pt.
16:12:01  Plus seventy eight pt.
16:12:01  Plus seventy eight pt.
16:12:01  Plus seventy seven pt.
16:11:59  Plus seventy eight pt.
16:11:58  Plus seventy nine pt.
```

Fig. 60. Displaying desired values by the EA stringarray.mq4.

Pay attention to the solution of Problem 18. If we used the same solution for Problem 25, operator 'switch' would contain about 100 lines - one line for each solution variant. Such an approach to program development cannot be considered satisfactory. Moreover, such solutions are useless if we need to process tens and sometimes hundreds of thousand variable values. In such cases use of arrays is justified and very convenient.

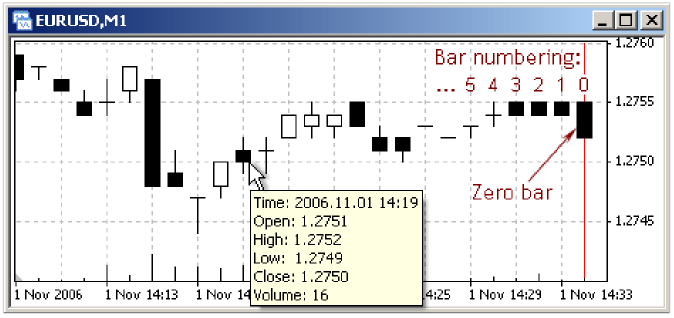
Arrays-Timeseries

Array-timeseries is an array with a predefined name (Open, Close, High, Low, Volume or Time), the elements of which contain values of corresponding characteristics of historic bars.

Data contained in arrays-timseries carry a very important information and are widely used in programming in MQL4. Each array-timeseries is a one-dimensional array and contains historic data about one certain bar characteristic. Each bar is characterized by an opening price Open[], closing price Close[], maximal price High[], minimal price Low[], volume Volume[] and opening time Time[]. For example, array-timeseries Open[] carries information about opening prices of all bars present in a security window: the value of array element Open[1] is the opening price of the first bar, Open[2] is the opening price of the second bar, etc. The same is true for other timeseries.

Zero bar is a current bar that has not fully formed yet. In a chart window the zero bar is the last right one.

Bars (and corresponding indexes of arrays-timeseries) counting is started from the zero bar. The values of array-timeseries elements with the index [0] are values that characterize the zero bar. For example, the value of Open[0] is the opening price of a zero bar. Fig. 61 shows numeration of bars and bar characteristics (reflected in a chart window when a mouse cursor is moved to a picture).



Time: 2006.11.01 14:19
Open: 1.2751
High: 1.2752
Low: 1.2749
Close: 1.2750
Volume: 16

Fig. 61. Each bar is characterized by a set of values contained in arrays-timeseries.
Bars counting starts from a zero bar.

Zero bar in Fig. 61 has the following characteristics:

Index	Open[]	Close[]	High[],	Low[],	Time[]
[0]	1.2755	1.2752	1.2755	1.2752	2006.11.01 14:34

After some time the current bar will be formed and a new bar will appear in a security window. Now this new bar will be a zero bar and the one hat has just formed will become the first one (with the index 1):

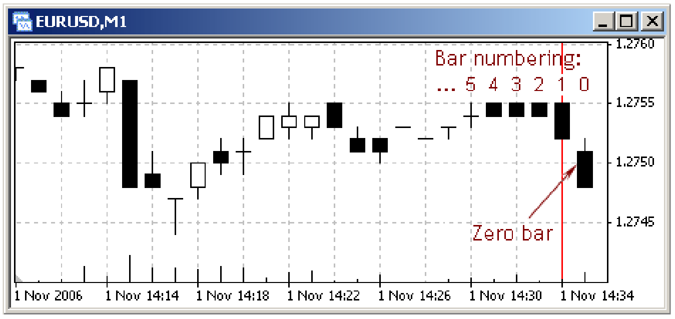


Fig. 62. Bars are shifted after some time, while numeration is not shifted.

Now the values of arrays-timeseries elements will be the following:

Index	Open[]	Close[]	High[],	Low[],	Time[]
[0]	1.2751	1.2748	1.2752	1.2748	2006.11.01 14:35
[1]	1.2755	1.2752	1.2755	1.2752	2006.11.01 14:34

Further in the security window new bars will appear. The current yet unformed the rightest bar will always be zero, the one to the left of it will be the first bar, the next one - the second, etc. However, a bar's characteristics are not changed: the bar in the example was opened at 14:43, and his opening price will still remain 14:43, its other parameters will also remain unchanged. However, the index of this bar will increase after the appearance of new bars.

So, the most important feature concerning arrays-timeseries is the following:

Values of array-timeseries elements are a bar's proper characteristics and are nether changed (except the following characteristics of a zero bar: Close[0], High[0], Low[0], Volume [0]), a bar's index reflects its deepening into future for the current moment and is changed in course of time.

It should be also noted the bar opening time is counted multiple of calender minutes, seconds are not taken into account. In other words, if in the period between 14:34 and 14:35 a new tick has come at 14:34:07, a new bar with opening time 14:43 will appear in the minute timeframe. Accordingly, bar opening time in the 15-minute timeframe is multiple of 15 minutes, within an hour interval the first bar is opened at n hours 00 minutes, the second one at n:15, the third - at n:30, the fourth - at n:45.

To understand correctly the significance of indexes in timeseries, let's solve a simple problem:

Problem 26. Find the minimal and the maximal price among the last n bars.

Pay attention, solution of such problems is impossible without referring to values of arrays-timeseries. An Expert Advisor defining the maximal and the minimal price among the preset number of last bars may have the following solution (extremumprice.mq4):

```
//--------------------------------------------------------------------
// extremumprice.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
extern int Quant_Bars=30;                       // Amount of bars
//--------------------------------------------------------------------
int start()                                     // Special funct. start()
  {
   int i;                                       // Bar number
   double Minimum=Bid,                          // Minimal price
          Maximum=Bid;                          // Maximal price

   for(i=0;i<=Quant_Bars-1;i++)                 // From zero (!) to..
     {                                          // ..Quant_Bars-1 (!)
      if (Low[i]< Minimum)                      // If < than known
         Minimum=Low[i];                        // it will be min
      if (High[i]> Maximum)                     // If > than known
         Maximum=High[i];                       // it will be max
     }
   Alert("For the last ",Quant_Bars,            // Show message
         " bars Min= ",Minimum," Max= ",Maximum);
   return;                                      // Exit start()
  }
//--------------------------------------------------------------------
```

In the program extremumprice.mq4 a simple algorithm is used. Amount of bars to be analyzed is set up in the external variable Quant_Bars. At the program beginning current price value is assigned to the desired Minimum and Maximum. The search of the maximal and minimal values is conducted in the cycle operator:

```
   for(i=0;i<=Quant_Bars-1;i++)                 // From zero (!) to..
     {                                         // ..Quant_Bars-1 (!)
      if (Low[i]< Minimum)                      // If < than known
         Minimum = Low[i];                      // it will be min
      if (High[i]> Maximum)                     // If > than known
         Maximum = High[i];                     // it will be max
     }
```

Here descriptive is the interval of index values (integer variable i) of processed array-timeseries elements Low[i] and High[i]. Pay attention to the Expression_1 and Condition in the cycle operator header:

```
      for(i=0;i<=Quant_Bars-1;i++)                  // From zero (!) to..
```

In the first iteration calculations are conducted with zero index values. It means in the first iteration the values of the zero bar are analyzed. Thus it is guaranteed that the last price values that appeared in a security window are also taken into account. The section Predefined Variables contains the rule according to which values of all predefined variables including arrays-timeseries are updated at the moment of special functions' start. So, none of price values will remain uncounted.

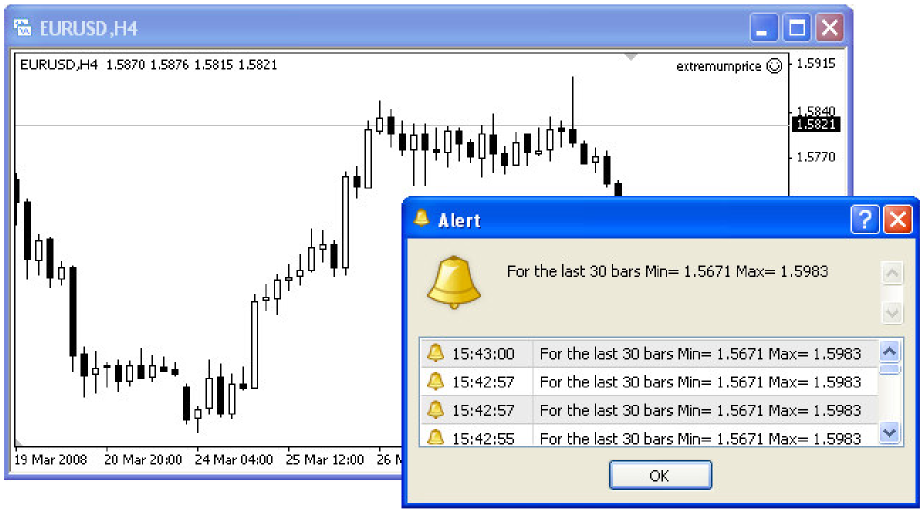
The list index of timeseries elements processed in a cycle is the index smaller by one than the number of processed bars. In our example the number of bars is 30. It means the maximal index value must be 29. So, values of timeseries elements with indexes from 0 to 29, i.e. for 30 bars, will be processed in the cycle.

It is easy to understand the meaning of calculations in the cycle operator body:

```
         if (Low[i]< Minimum)                       // If < than known
            Minimum = Low[i];                       // it will be min
         if (High[i]> Maximum)                      // If > than known
            Maximum = High[i];                      // it will be max
```

If the current value Low[i] (i.e. during the current iteration with the current index value) is lower than the known minimal value, it becomes the new minimal value. The same way the maximal value is calculated. By the moment of cycle end, variables Minimum and Maximum get the desired value. In further lines these values are displayed.

Launching this program one can get a result like this:



For the last 30 bars Min= 1.5671 Max= 1.5983

```
15:43:00   For the last 30 bars Min= 1.5671 Max= 1.5983
15:42:57   For the last 30 bars Min= 1.5671 Max= 1.5983
15:42:57   For the last 30 bars Min= 1.5671 Max= 1.5983
15:42:55   For the last 30 bars Min= 1.5671 Max= 1.5983
```

Fig. 63. Result of the EA extremumprice.mq4 operation.

Pay attention that the Expert Advisor may operate infinitely long showing correct results, and the program will use the same index values (in this case from 0 to 29). Values of arrays-timeseries elements with the zero index will change at the moment of a new appearance and values of arrays-timeseries elements characterizing the zero bar may change at any next tick (except values of Open[] and Time[] that do not change on the zero bar).

In some cases it is needed to perform some actions starting from the moment when a bar has been fully formed. This is important, for example, for the implementation of algorithms based on a candlestick analysis. In such cases usually only fully formed bars are taken into account.

Problem 27. At the beginning of each bar show a message with the minimal and maximal price among the last n formed bars.

To solve the task, it is necessary to define the fact a new bar beginning, i.e. to detect a new tick on a zero bar. There is a simple and reliable way to do this - analyzing the opening time of a zero bar. Opening time of a zero bar is the bar characteristic that does not change during the bar's formation. New ticks coming during a bar's formation can change its High[0], Close[0] and Volume[0]. But such characteristics as Open[0] and Time[0] are not changed.

That is why it is enough to remember the opening price of a zero bar and at each tick compare it to the last known zero bar opening price. As soon as a mismatch is found, this will mean the formation of a new bar (and completion of the previous one). In the EA newbar.mq4 the algorithm of detecting a new bar is implemented in the form of a user-defined function:

```
//--------------------------------------------------------------------
// newbar.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
extern int Quant_Bars=15;                       // Amount of bars
bool New_Bar=false;                             // Flag of a new bar
//--------------------------------------------------------------------
int start()                                     // Special funct. start()
  {
   double Minimum,                              // Minimal price
          Maximum;                              // Maximal price
//--------------------------------------------------------------------
```

```
     Fun_New_Bar();                               // Function call
     if (New_Bar==false)                          // If bar is not new..
        return;                                   // ..return
  //-----------------------------------------------------------------------
     int Ind_max =ArrayMaximum(High,Quant_Bars,1);// Bar index of max. price
     int Ind_min =ArrayMinimum(Low, Quant_Bars,1);// Bar index of min. price
     Maximum=High[Ind_max];                       // Desired max. price
     Minimum=Low[Ind_min];                        // Desired min. price
     Alert("For the last ",Quant_Bars,            // Show message
     " bars Min= ",Minimum," Max= ",Maximum);
     return;                                      // Exit start()
    }
  //-----------------------------------------------------------------------
  void Fun_New_Bar()                              // Funct. detecting ..
    {                                             // .. a new bar
     static datetime New_Time=0;                  // Time of the current bar
     New_Bar=false;                               // No new bar
     if(New_Time!=Time[0])                        // Compare time
       {
        New_Time=Time[0];                         // Now time is so
        New_Bar=true;                             // A new bar detected
       }
    }
  //-----------------------------------------------------------------------
```

Global variable New_Bar is used in the program. If its value is 'true', it means the last known tick is the first tick of a new bar. If New_Bar value is false. the last known tick has appeared within the formation of the current zero bar.

Flag is a variable, the value of which is defined in accordance with some events or facts.

Use of flags in a program is very convenient. The flag value can be defined in one place and used in different places. Sometimes an algorithm is used in the program, in which a decision is made depending on the combination of values of different flags. In the Expert Advisor newbar.mq4 the variable New_Bar is used as a flag. Its value depends directly on the fact of a new bar formation.

Calculations regarding the detection of a new bar are concentrated in the user-defined function Fun_New_Bar(). In the first lines of the function the static variable New_Time is defined (remember, static variables do not loose their values after a function execution is over). At each function call, the value of the global variable New_Bar is set 'false'. The detection of a new bar is performed in 'if' operator:

```
     if(New_Time != Time[0])                      // Compare time
       {
        New_Time = Time[0];                       // Now time is so
        New_Bar = true;                           // A new bar detected
       }
```

If New_Time value (calculated in the previous history) is not equal to Time[0] of a zero bar, it denotes the fact of a new bar's formation. In such a case control is passed to 'if' operator body, where the new opening time of the zero bar is remembered and the true value is assigned to New_Bar (for convenience it can be said that the flag is set in an upward position).

When solving such problems, it is important to take into account the peculiarity of using different flags. In this case the peculiarity is that New_Bar value (flag position) must be updated earlier than it will be used in calculation (in this case - in the special function start()). New_Bar value is defined in the user-defined function, that is why it must be called possibly early in the program, i.e. before first calculations, in which New_Bar is used. The special function start() is constructed correspondingly: user-defined function call is performed immediately after the declaration of variables.

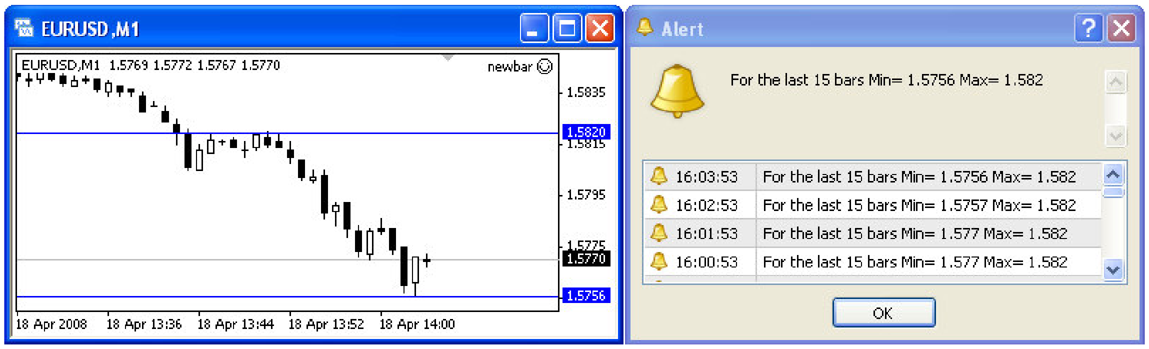
The calculation of desired values is worthwhile only if start() is launched by a tick at which the new bar is formed. That's why in start(), immediately after detecting a new bar formation, the flag position (New_Bar value) is analyzed:

```
     if (New_Bar == false)                        // If bar is not new..
        return;                                   // ..return
```

If the last tick that started the execution of start() has not form a new bar, control is passed to an operator that finishes the execution of start(). And only if a new bar has been formed, control is passed to the next lines for the calculation of desired values (what is required by the problem provisions).

Calculation of the maximal and minimal value is performed using standard functions ArrayMaximum() and ArrayMinimum(). Each of the functions returns array element index (of the maximal and minimal value correspondingly) for a specified indexes interval. Under the problem conditions, only completely formed bars must be analyzed, that's why the chosen boundary index values are 1 and Quant_Bars - set amount of bars (the zero bar is not formed yet, i.e. is not taken into account during calculations). For more detailed information about the operation of these and other timeseries accessing functions, refer to the documentation on the developper's website (http://docs.MQL4.com) or to "Help" in MetaEditor.

Fig. 64 shows the change of maximal and minimal prices in the preset interval during the program execution:



For the last 15 bars Min= 1.5756 Max= 1.582

16:03:53	For the last 15 bars Min= 1.5756 Max= 1.582
16:02:53	For the last 15 bars Min= 1.5757 Max= 1.582
16:01:53	For the last 15 bars Min= 1.577 Max= 1.582
16:00:53	For the last 15 bars Min= 1.577 Max= 1.582

Fig. 64. Expert Advisor newbar.mq4 operation result.

← GlobalVariables Programming in MQL4 →

Practical Programming in MQL4

This present second part of the book considers the following issues: the order of performing trade operations, the principles of coding and use of simple scripts, Expert Advisors and indicators, as well as standard functions often used in programming in MQL4. All sections contain some examples of programs that are ready to use, but limited application field.

The section named Creation of Normal Programs gives an example you can use as a basis for designing your own simple Expert Advisor to be used in real trading.

All trading criteria given below are used for educational purposes and should not be considered as guidelines in trading on real accounts.

- **Programming of Trade Operations**

When programming trade operations, you should consider the requirements and limitations related to the characteristics of orders and rules accepted in your dealing center, as well as the special features of trade order execution technology. The section provides the detailed description of the order of performing trades and contains a lot of examples that explain the purposes of all trade functions used to form trade orders. The section contains some ready-made scripts intended for restricted application.

- **Simple Programs in MQL4**

After the programmer has mastered programming of trade operations, he or she can start creating simple programs. The section deals with the general principles of creating a simple Expert Advisor and a simple custom indicator, as well as the order of sharing an Expert Advisor with various indicators. Particularly, the section describes the order of data transfer from a custom indicator into an EA. It also gives some examples of simple programs ready to be used in trading practice.

- **Standard Functions**

Totally, MQL4 counts over 220 standard functions, not including the functions of technical indicators. It would be rather difficult to describe in this book and give examples of each function, considering their great amount. Some functions that require detailed explanations have already been considered in the preceding sections. In this present section, we consider the most frequently used standard functions and give some examples of how to use them in programs. At the end of each subsection, we provide the full list of functions of a certain category and their brief description.

- **Creation of Normal Programs**

As a rule, once having practiced the coding of some simple applications in MQL4, the programmer goes to a more sophisticated project: He or she creates a convenient program intended for practical use. In some cases, simple programs do not satisfy the needs of a trading programmer for at least two reasons:

1. The limited functionality of simple programs cannot completely provide the trader with all necessary information and trading tools, which makes the application of such programs less efficient.

2. The code imperfection of simple programs makes it difficult to further upgrade them in order to increase their services.

In this present section, we represent one of possible realization versions of a trading Expert Advisor that can be used as a basis for creation of your own project.

← Arrays Trade Operations →

Programming of Trade Operations

When programming trade operations, you should consider the requirements and limitations related to the characteristics of orders and rules accepted in your dealing center, as well as the special features of trade order execution technology. The section provides the detailed description of the order of performing trades and contains a lot of examples that explain the purposes of all trade functions used to form trade orders. The section contains some ready-made scripts intended for restricted application.

- Common Way of Making Trades
  A trader or an MQL4 program can only order to make trades, whereas the trades as such are registered on the trade server. The intermediary between the trade server and a program is the client terminal. Incorrect orders will be rejected immediately in the client terminal, so you have to obtain an insight into the general order of making trades.

- Order Characteristics and Rules for Making Trades
  The trade instructions are given using trade orders. In the order, you should specify multiple parameters, one part of which is determined by the current prices and the direction of trade, another part depends on the symbol of the trade. Orders that are delivered in the trade server will be checked in the real-time mode for their compliance with the current situation and with the account state. This is why the knowledge of rules for making trades is necessary.

- Opening and Placing Orders
  The most important trading function is OrderSend(). It is this function that is used to send requests to the trade server to open a market order or to place a pending one. You can immediately specify the necessary values of StopLoss and TakeProfit. The incorrect values of these parameters, as well as of open price and volume of the order, may result in errors. It is important to process these errors properly. Function MarketInfo() allows you to minimize the amount of such errors.

- Closing and Deleting Orders. Function OrderSelect
  Market orders can be closed using the function OrderClose(), while pending orders can be deleted with the function OrderDelete(). When sending a request to close or delete an order, you should specify the ticket of this order. We will select the necessary order using the function OrderSelect(). Besides, if there are two opposite orders for a symbol, you can close them simultaneously, one by another, using the function OrderCloseBy(). When executing such trade, you will save one spread.

- Modification of Orders
  The levels TakeProfit and StopLoss can be modified using the function OrderModify(). For pending orders, you can also change the level of triggering. You cannot modify the volume of pending orders. Modification of market and pending orders will also put certain requirements related to the correctness of this trade operation. It is highly recommended, if you make a trade, that you process the results of this trade, handle the errors.

← Programming in MQL4 Common Way of Making Trades →

Common Way of Making Trades

All calculations and other actions performed due to the execution of an application program can be divided into two groups by the location of their execution: those executed in the user's PC and those executed on the server side. A significant amount of calculations is performed on the user's side. This group includes the execution of application programs. Trades belong to the second group. Making trades implies data conversion on the server.

Considering trades, we will distinguish the following terms:

Market order - is an executed order to buy or sell assets for a symbol (security). A market order is displayed in the symbol window until the order is closed.

Pending order is a trade order to buy or sell assets for a security (a symbol) when the preset price level is reached. Pending order is displayed in the symbol window until it becomes a market order or is deleted.

Trade Request is a command made by a program or by a trader in order to perform a trade.

Trade is opening, closing or modification of market and pending orders.

Trade Making Diagram

Three constituents are involved in making trades: an application program, the client terminal and the server (see Fig. 65). A request is formed in the program (as we mentioned above, any application programs can be executed only in the user's PC; no application programs are installed on the server). The request formed by the program will be passed to the client terminal that, in its turn, sends the request to the server. At the server side, the decision on the request execution or rejection will be made. The information about the obtained results will be passed by the server to the client terminal and then to the program.

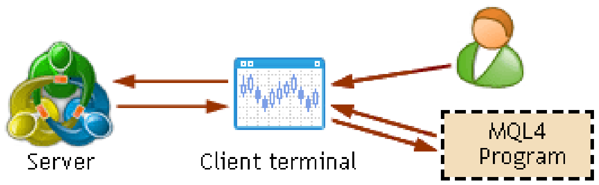


Fig. 65. Diagram of requesting for making trades.

Requesting

A trade request can be made by a trader or by a program. For a trader to be able to make a request the client terminal provides the "Orders" control panel (see the description of the client terminal). The requests are made in the program according to the algorithm, as a result of the execution of trade functions. Nowhere else (neither in the client terminal nor on the server) trade requests are formed spontaneously.

Program Features

Depending on the algorithm, a program can form different requests - for opening, closing or modification of market and pending orders. The following trade functions are used in a program to form trade requests:

- **OrderSend()** - to open market and pending orders;
- **OrderClose()** and **OrderCloseBy()** - to close market orders;
- **OrderDelete()** - to delete pending orders;
- **OrderModify()** - to modify market and pending orders.

The above trade functions can be used only in Expert Advisors and scripts; the use of these functions in indicators is prohibited (see also Table 2). There are other functions that belong to trade ones (see help file in MetaEditor and the section Trade Functions in this present book). However, their execution is attributed to calling for the terminal information environment in order to obtain reference information, so it does not result in forming requests and calling to the server.

Features of Client Terminal

A request made by the program as a result of the execution of a trade function is passed to the client terminal for processing. The client terminal analyzes the request contents and performs one of the following two actions: either sends the request to the server for it to be executed on the server, or rejects the request and sends nothing to the server.

The client terminal allows only correct requests to be sent to the server. If the program is coded in such a way that it forms, for example, a request for opening an order at a non-existing price, the client terminal will not send this request to the server. If the program forms correct requests (orders are opened and closed at the latest known price, the order value is within the range limited by the dealing center, etc.), then this request will be sent to the server.

Only one execution thread is provided in the client terminal to perform trades. This means that the client terminal can simultaneously work only with one request. If there are several Expert Advisors or scripts trading in the client terminal and no program has passed a trade request to the client terminal, the trade requests of all other Expert Advisors and scripts will be rejected until the client terminal completes processing the current request, i.e., until the trade thread is free.

Server Features

The information about trade history for each account (opening, closing, modifying orders) is highly secured by the server and is of a higher priority as compared to the history of trades stored in the client terminal. The right to execute trade requests is granted only to a dealer or to the server that processes requests automatically (if the dealing center provides the server with this feature for a certain period of time). A request that is delivered in the server can be either executed or rejected. If the trade request is executed (i.e., a trade is made), the server will make all necessary conversions of data. If the trade request is rejected, the server does not convert any data. No matter which decision (to execute or to reject a request) is made, the information about this decision will be passed to the client terminal to synchronize the history.

A trade request formed as a result of the program execution and a trade request formed by the trader manually are absolutely the same, from the viewpoint of the server, so the server makes no distinction between requests when processing them.

It is also possible at the server side to prohibit Expert Advisors to trade in the client terminal. It is sometimes necessary, if the program operation causes conflicts. For example, if the implementation of an incorrect algorithm results in that the program continuously forms alternating trade requests for opening and closing of orders with very small intervals in time (for example, at every tick), or if the requests for opening, deletion or modification of pending orders are too frequent.

Trading Procedure

The procedure of performing trades is interactive and realized in the real-time mode. The diagram (Fig. 66) shows all events related to performing a trade.

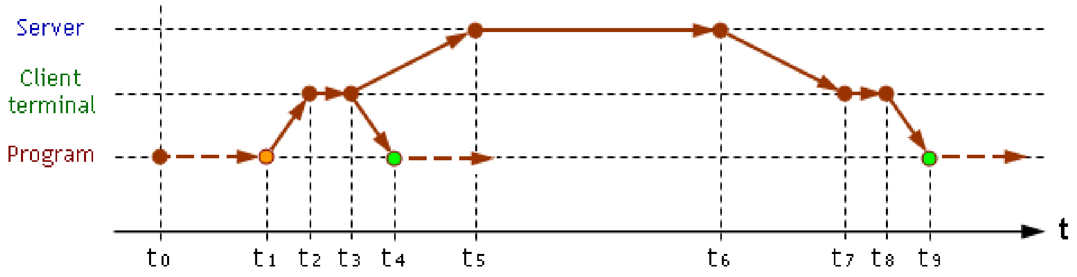


Fig. 66. Event sequence in making a trade.

Event 0. The program is launched for execution at the moment t0.

Event 1. At the moment t1, the program has formed a trade request as a result of the execution of a trade function. The trade request is passed to the client terminal. At that moment, the program passes the control to the client terminal and the execution of the program is stopped (the red point in the diagram).

Event 2. The client terminal has received the control and the information about the request contents. Within the period of time between t2 and t3, the client terminal analyzes the contents of the trade request and makes a decision on further events.

Event 3. The client terminal performs that made decision (one of two alternatives).

Alternative 1. If the trade request formed as a result of the execution of one of trade functions has turned out to be incorrect, the control is passed to the program. In this case, the next event will be Event 4 (this can happen if, for example, the program has sent the request for opening an order, the value of which exceeds the account equity available).

Event 4. The program has received the control (moment t4, green point) and can continue execution from the location where the request has previously been formed. At the same moment, the program has received the information about that the trade order has not been executed. You can find out about the reason, for which the trade request has not been executed, by analyzing the ode of the returned error. Below we will consider the matter of how to do this. Here, it must only be noted that not all requests result in execution of trades. In this case, the program has formed an incorrect request, which results in that the client terminal has rejected this request and returned the control to the program. In this case, no referencing to the server takes place. The time intervals between t1 - t2 - t3 - t4 are negligibly short and do not exceed several ms in total.

Alternative 2. If the program has formed a correct trade request, the client terminal sends this request to the server; the next event will be Event 5 (the moment of t5) - the server will receive the trade request. The connection between the client terminal and the server is established via Internet, so the time spent on sending of the trade request to the server (time interval between t3 and t5) is completely dependent on the connection quality. For a good-quality connection, this period of time can be approximately 5 to 10 ms, whereas it can be measured in whole seconds at bad connection.

Event 5. At the moment t5, the server has received the trade request. The server can execute or reject this received request. The decision on executing or rejecting of the request can be made at the server side within a certain period of time (at the moment t6). The time interval between t5 and t6 can range from some milliseconds to the dozens of seconds, depending on the situation. In some cases, if the server operates in the automated mode and there are no rapid movements in the market and other traders are not very active, the trade request can be executed or rejected within several milliseconds. In other cases, if the server is overloaded due to high activity of traders and if the decision on executing or rejecting of the request is made by a human dealer, the time taken by making the decision can be counted in the dozens of seconds.

Event 6. If no considerable changes take place on the market within the time interval from the moment of forming the trade request by the program (t1) to the moment of making decision by the server (t6), the trade request will be executed, as a rule. If the price of the symbol has changed within this time or the value of the order to be opened is exceeding free equity of the account at the moment of making the decision, or other impediments occur, then the server decides to reject the trade request.

The server's rejection of trade requests (though they have already been checked by the client terminal) is common. In general, the most trade requests that are delivered to the server are accepted to execution by the server. However, in some cases, a request can be rejected, so your application program must be coded in such a way that it takes such possibility into consideration and operates properly in such situations.

Whatever decision (to execute/reject a trade request, Event 6) is made by the server, the information about it is sent by the server to the client terminal that has delivered the request.

Event 7. The client terminal has received the server response. The server response follows the same path through the Internet as the request delivered to the server; so the time spent on receiving of the server response completely depends on the connection quality. According to the modifications made on the server, the client terminal will reflect the corresponding changes. For example, if the execution of a trade request results in closing or opening an order, the client terminal will display this event graphically in the symbol window and textually in the 'Terminal' window (the tabs 'Trade' and 'Account History'). If the server has rejected the trade request, no changes will be made in any windows of the client terminal.

Event 8. The client terminal has completed the displaying of changes and is passing the control to the program.

Event 9. The program has received the control and can continue operating.

Please note:

From the moment when the program sends a trade request (and simultaneously passes the control) to the client terminal, to the moment when the control is returned to the program, the latter one is in the waiting mode. No operations are made in the program during this period of time. The control is returned to the program according to the execution rule for call to the function that has formed the trade request.

If the trade request is incorrect, then the program isn't in the waiting mode for a long time (the interval between t1 and t4). However, if the trade request is 'approved' by the client terminal and sent to the server, the duration of the program waiting period (t1-t9) can be different and depends on both the connection quality and on the time taken by decision making of the server - for several milliseconds to minutes.

As soon as the program receives the control, it can continue operating. The operating program can analyze the code of the last error returned by the client terminal and, in this manner, find out about whether the trade request was executed or rejected.

Conflicts in Making Trades. Error 146

When considering above the features of the client terminal, we mentioned that the client terminal could process only one request in a time. Let's now consider what events will take place if several requests formed by different programs will be passed to the client terminal.

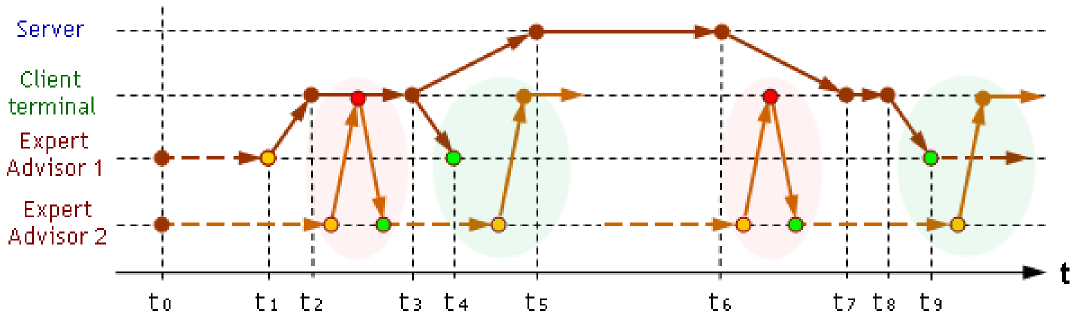


Fig. 67. Conflicts in passing several requests to the client terminal from different programs.

In Fig. 67, we can see that two trading Expert Advisors are launched for execution on the client terminal simultaneously. EA1 formed a trade request at the moment t1 and passed it the client terminal the moment t2.

EA2 has also created a request and refers to the client terminal when the client terminal is processing the first request (period from t2 to t3). In this situation, the client terminal cannot consider the request formed by EA2, so it will reject this request and return the control to EA2. Please note that, in this case, the request is rejected by the client terminal not for the request is incorrect, but because the terminal is busy with processing of the other request. EA2 will continue operating. It can analyze the error code that explains the reason why the request has been rejected (in our case, it is error 146).

If it is EA2 (in a general case, it can be one or several trading programs) that passes its request to the client terminal within the period of time between t1 and t4, then this request will be rejected (a group of events in the pink area). The client terminal becomes free at the moment t4 (green point). Starting from this moment, EA2 can successfully pass its request to the client terminal (a group of events in the green area). This request will be received and considered by the client terminal that can finally reject it, too, but for the reason of its incorrectness, or it can send this request to the server.

If the trade request created by EA1 is considered by the client terminal as correct, this request will be sent by the client terminal to the server at the moment t3. In this case, the client terminal switches to the waiting mode and cannot consider any other trade requests. The client terminal will only become free for considering of other trade requests at the moment t9. Thus, according to Variation 2, the client terminal cannot analyze trade requests within the period of time between t1 and t9. If within this period any program refers to the client terminal in order to pass a request for consideration, the client terminal will reject this event and pass the control to the program (a group of events in the pink area within the period of time between t6 and t7). The program that has received the control continues its operation and, by analyzing the error code, can find out about the reason, for which the request has been rejected (in this case, it is error 146).

Starting from the moment t9, the client terminal will be completely free for analyzing any other trade requests. EA2 can successfully pass the trade request to the client terminal within the period of time that follows the moment t9. According as the client terminal considers this request to be correct or not, the request will be passed by the client terminal to the server or rejected.

The analysis of errors that occur in making trades is considered in more details in the sections below.

← Trade Operations Order Characteristics and Rules for Making Trades →

Order Characteristics and Rules for Making Trades

Before we start to describe trade functions, we should consider parameters that characterize market prices, order types, their characteristics, as well as the rules for making trades.

Characteristics of Symbols

First of all, we should consider the principle used by brokerage companies to form prices of securities. This principle consists in that the broker offers to the trader a two-way quote for performing trades.

Two-way quote is a connected pair of market prices offered by the broker for buying and selling of assets for a security (symbol) at the present moment.

Bid is the lower of two prices offered by broker in a two-way quote for a security.

Ask is the higher of two prices offered by broker in a two-way quote for a security.

Point is the unit of price measurement for a symbol (the minimum possible price change, the last significant figure of the price value).

Spread is the difference between the larger and the smaller price in points in a two-way quote for a symbol.

Normally, spread is a fixed value. In MetaTrader 4, it is accepted to display in the symbol window the chart that reflects only Bid price changes:

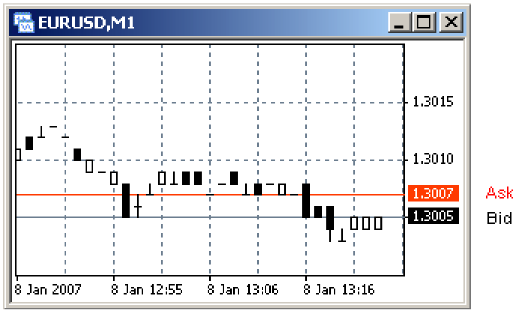


Fig. 68. A normal price chart for a symbol.

Fig. 68 shows us a symbol window where we can see the changes of the Bid prices and the two-way quote - the line of the current Bid price (black, 1.3005) and the line of the current Ask price (red, 1.3007). It can be easily seen that, in this case, the broker offers a spread of 2 points. The history for the Ask price is not displayed in the chart, but it is implied and can be easily calculated for any moment of time.

Order Types and Characteristics

There are six order types in total: two types of market orders and four types of pending orders.

Buy is a market order that defines buying of assets for a symbol.

Sell is a market order that defines selling of assets for a symbol.

BuyLimit is a pending order to buy assets for a security at a price lower than the current one. The order will be executed (modified into market order Buy) if the Ask price reaches or falls below the price set in the pending order.

SellLimit is a pending order to sell assets for a security at a price higher than the current one. The order will be executed (modified into market order Sell) if the Bid price reaches or rises above the price set in the pending order.

BuyStop is a pending order to buy assets for a security at a price higher than the current one. The order will be executed (modified into market order Buy) if the Ask price reaches or rises above the price set in the pending order.

SellStop is a pending order to sell assets for a security at a price lower than the current one. The order will be executed (modified into market order Sell) if the Bid price reaches or falls below the price set in the pending order.

Lot is the volume of an order expressed in the amount of lots.

StopLoss is a stop order; it is a price set by trader, at which a market order will be closed if the symbol price moves in a direction that produces losses for the order.

TakeProfit is a stop order; it is a price set by trader, at which a market order will be closed if the symbol price moves in a direction that produces profits for the order.

Trading Requirements and Limitations

In order to form correct trade requests in your application programs (Expert Advisors and scripts), you should take the existing requirements and limitations into consideration. Let's consider them in more details.

All trades are performed at correct prices. The execution price for each trade is calculated on the basis of the correct price of a two-way quote.

The above rule is the common rule for all market participants and cannot be changed at the will of the developers of a trading platform or on the basis of an agreement between a broker and a trader. This means, for example, that a market order can only be opened at the current market price, but not at any other price. The correct-price calculation procedure for different trades is considered below.

When calculating correct prices, it is also necessary to consider the limitations of the service provider (dealing center). These limitations include the minimum distance and the freeze distance. These limitations mean that the broker needs some time for preparations for performing of new trades, whether converting a pending order into a market one or closing an order by stop order.

Dealing centers limit the value of the minimum permissible difference between the market price and the requested price of each stop orders of a market order, between the market price and the requested price of a pending order, as well as between the requested price of a pending order and the requested prices of its stop orders. This means, for example, that in a trade request for opening a market order you can specify only the stop-order price values that are not distant from the current price less than the prescribed minimum distance. A trade request containing stop-order prices that are closer to the market price than the minimum distance is considered by the client terminal as incorrect one. Different dealing centers can establish different, dealing center-specific limitations for the minimum allowed distance. As a rule, the value of such distance ranges between 1 and 15 points. For the most commonly used securities (EUR/USD, GBP/USD, EUR/CHF, etc.), this distance makes in most dealing centers 3-5 points. Different securities can have different minimum allowed distances, too. For example, this value can be 50-100 points for gold. The value of the minimum distance for any symbol can be changed by the broker at any time (this usually precedes the broadcasting of important commercial news). There are no limitations for the maximum distance.

Freeze distance limits the possibility to modify the requested prices of opening your pending orders, as well as the requested stop levels for market orders that are in the freeze area. This means, for example, that, if the market price is 1.3800, your pending order is placed to be opened at 1.3807 and the broker's prescription is 10, your pending order is in the freeze area, i.e., you cannot modify or delete it. At a calm market, brokers usually don't set freeze distance, i.e., its value = 0. However, during the period preceding important news or at high volatility, the broker may set a certain value of a freeze distance. In different conditions and for different brokers, this value may range from 1 to 30 points for basic symbols and take higher values for other symbols. Brokerage company can change the freeze-distance value at its own discretion at any time.

The limitations of price levels limited by the values of minimum distance and of freeze distance are calculated on the basis of correct prices.

Opening/Closing Market Orders

Opening a market order implies buying or selling some assets for a symbol at the current market prices (see Requirements and Limitations in Making Trades). To open a market order, use function OrderSend(); for closing it, use function OrderClose().

The correct open price of of market order Buy is the latest known market price Ask.
The correct open price of of market order Sell is the latest known market price Bid.

The limitation related to the position of stop levels of the market order to be opened is calculated on the basis of the correct market price used for closing the order.

Orders StopLoss and TakeProfit cannot be placed closer to the market price than the minimum distance.

For example, the minimum distance for EURUSD makes 5 points. Market order Sell is opened at Bid=1.2987. The price corresponding with the two-way quote used for closing this order Sell is Ask=1.2989. The following stop levels will be the closest to the current price at the order opening (see Fig. 69 and Requirements and Limitations in Making Trades):

StopLoss = Ask + minimum distance = 1.2989 + 0.0005 = 1.2994, and

TakeProfit = Ask - minimum distance = 1.2989 - 0.0005 = 1.2984.

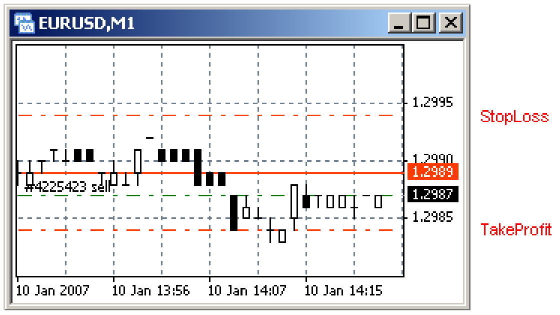


Fig. 69. Market order Sell with Stop Levels being closest to the market price.

If at requesting for opening a market order Sell at Bid=1.2987 you use the stop-level values closer than the above ones (SL=1.2994 and TP=1.2984), the trade request will be rejected by the client terminal. Besides, you should take into consideration that price slippages are possible during opening of orders, which results in opening of your order at a price other than that requested by a certain value you have specified in the request. If the same request has the specified values of stop levels closest to the requested open price, this request will also be rejected by the client terminal since, in this case, the request does not comply with the required minimum distance between the open price of your order and the requested price of one of the stop orders. This is why it is not recommended to use in trade requests for opening of market orders the stop-order values closest to the requested order open price. On the contrary, it is recommended to have some "free play", i.e., to specify such values of stop orders that would be distant from the requested open price of the order by 1-2 points farther than the value of the minimum allowed distance.

Market orders can be closed as a result of execution of the trade request given by trader or by program, as well as when the price reaches one of the prices specified in stop orders.

The correct close price of a market order Buy is the latest known market price Bid.
The correct close price of a market order Sell is the latest known market price Ask.

If we close order Sell (Fig. 69) at the current moment, it will be closed at Ask=1.2989, i.e., with a loss of 2 points. If we allow this order to remain open for a while and the Ask price falls down to 1.2984, the order will be closed at that price with the profit of 3 points. If the market price grows during this period of time and reaches Ask= 1.2994, the order will be closed at that price with a loss of 7 points.

If the application has formed a request for opening or closing a market order at a price that does not correspond with the latest known market price, the request will be rejected by the client terminal.

The limitation related to closing of market orders is calculated on the basis of the correct market price used for closing of the order.
Order cannot be closed, if the execution price of its StopLoss or TakeProfit is within the range of freeze distance from the market price.

For example, the order shown in Fig. 69 can be closed only if the brokers set the freeze-distance value 4 points or less as of the moment of closing. The open price of this order does not matter, in this case. The boarders of the freeze band for the order are calculated on the basis of the market price. So, if it is = 4, the price of the upper freeze boarder is equal to += 1.2989 + 0.0004 = 1.2993, whereas the price of the lower freeze boarder is, correspondingly, - = 1.2989 - 0.0004 = 1.2985. In these conditions, no stop order is in the freeze area, so the order can be closed, if the trader (or a program) sends a correct request to the server. If the broker has set it to =5 as of the current moment, the boarders of the freeze band will be, 1.2994 and 1.2984, respectively. In this case, each stop order gets into the freeze boarder, i.e., undergoes the limitation set by the broker, so the order cannot be closed at the trader's initiative or by request of the trading program. In this example, both stop orders undergo the limitation. In a general case, a market order cannot be closed at the initiative of the client terminal, if at least one stop level of this order is in the freeze area.

If two market orders are opened for one symbol simultaneously, one of them being Buy and another one being Sell, they can be closed in one of two ways: you can close them consecutively, one by one, using OrderClose(); or you can close one of them by the other one using OrderCloseBy(). In terms of saving money, the second way is more preferable, because you will save one spread in closing orders by each other. The use of trade functions is considered in more details below in this book.

Placing and Deleting Pending Orders

A pending order implies the requested order open price other than the current market price. To place pending orders, use function OrderSend(). Use function OrderDelete() to delete a pending order.

Pending orders SellLimit and BuyStop are placed at a price that is higher than the current market price, whereas BuyLimit and SellStop are placed at a price that is lower than the current market price.

The limitation related to the position of the pending order to be placed is calculated on the basis of the correct market price for conversion a pending order into a market order.

Pending orders BuyLimit, BuyStop, SellLimit and SellStop cannot be placed at a price that is closer to the market price than the minimum distance.

For example, in order to calculate the minimum allowed price for the order BuyStop, you should add the value of minimum distance to the latest known Ask price. If StopLevel= 5, then the minimum allowed price to place the pending order BuyStop will make 1.3003+0.0005 = 1.3008 (see Fig. 70). This means that order BuyStop can be placed at the current moment at the requested price of 1.3008 or at a higher price. In this example, BuyStop is placed at 1.3015, which is quite allowable.

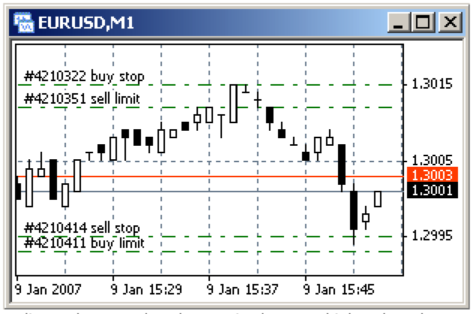


Fig. 70. Pending orders are placed at a price lower or higher than the current price.

The requested price of pending order BuyStop is 1.3015.

The requested price of pending order SellLimit is 1.3012.

The requested price of pending order SellStop is 1.2995.

The requested price of pending order BuyLimit is 1.2993.

In the above example, all pending orders were placed at zero bar at the moment shown in Fig. 70, while the minimum distance for placing of pending orders made 5 points. Order SellStop is the closest to the market price. In this case, Bid = 1.3001 and the requested price of SellStop = 1.2995. Thus, the distance between the order and the correct price of the two-way quote (Bid) is 6 points (1.3001 - 1.2995), i.e., it is more than the minimum distance requires. This means that, at opening of the order (or all other orders in this example), the trade request was "approved" by the client terminal and sent to the server. The server also checked it for compliance with the requirements and decided to execute the request for placing of the pending order (see Requirements and Limitations in Making Trades).

The position of stop orders attached to pending orders is also limited by the minimum distance:

The limitation related to the position of stop orders of a pending order is calculated on the basis of the requested open price of the pending order and has no relation to market prices.
StopLoss and TakeProfit of a pending order cannot be placed closer to the requested price than at the minimum distance.
The positions of StopLoss and TakeProfit of pending orders are not limited by freeze distance.

In Fig. 71, we can see pending order SellLimit, the stop orders of which are as close to the requested order price as possible. In this case, the requested order price = 1.2944, StopLoss=1.2949, TakeProfit=1.2939. At the minimum distance of 5 points, these values are quite allowable.

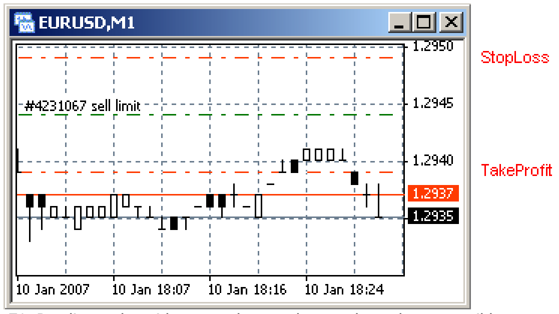


Fig. 71. Pending order with stop orders as close to the order as possible.

In this example, pending order SellLimit was opened at 18:07. You can see in Fig. 71 that after that the market price then reached and crossed one of its stop orders, and then went down again. This event didn't influence the pending order in any way: a stop order can only close the market order, i.e., it becomes effective as soon as the pending order is modified into a market order. In this case, the pending order is not modified into a market one (since the Bid price has not reached the requested order open price), so the stop-order price crossing this level has not resulted in any changes.

The limitation related to the deletion of pending orders is calculated on the basis of the correct market price applicable for modification of a pending order into a market one.
Pending orders BuyLimit, BuyStop, SellLimit and SellStop cannot be deleted, if the requested open price of the order is within the range of the freeze distance from the market price.

Order SellLimit can be deleted at the moment shown in Fig. 71 as initiated by the client terminal, if only the value specified at this moment is equal to or less than 8 points. In this case, the upper boarder of the freeze band (to be calculated for SellLimit) will make: + = 1.2935 +0.0008 = 1.2943. The requested order open price makes 1.2944, i.e., the order is placed outside the freeze band and can be deleted. If the broker sets the value to more than 8 points, then the pending order SellLimit cannot be deleted and the client terminal rejects the trade request.

Modification of Pending Orders into Market Orders

Pending orders are automatically modified into market orders at the server, so no functions are provided to execute this operation (see Requirements and Limitations in Making Trades).

Pending orders BuyLimit and BuyStop are modified to market ones, if the last known Ask price reaches the requested pending order price.
Pending orders SellLimit and SellStop are modified to market ones, if the last known Bid price reaches the requested pending order price.

As to the pending orders shown in Fig. 70, we can say the following.

Pending order BuyStop is modified into market order Buy, if the current price Ask reaches the value of 1.3015.

Pending order SellLimit is modified into market order Sell, if the current price Bid reaches the value of 1.3012.

Pending order SellStop is modified into market order Sell, if the current price Bid reaches the value of 1.2995.

Pending order BuyLimit is modified into market order Buy, if the current price Ask reaches the value of 1.2993.

The subsequent events related to these orders are shown in Figures 72-74.

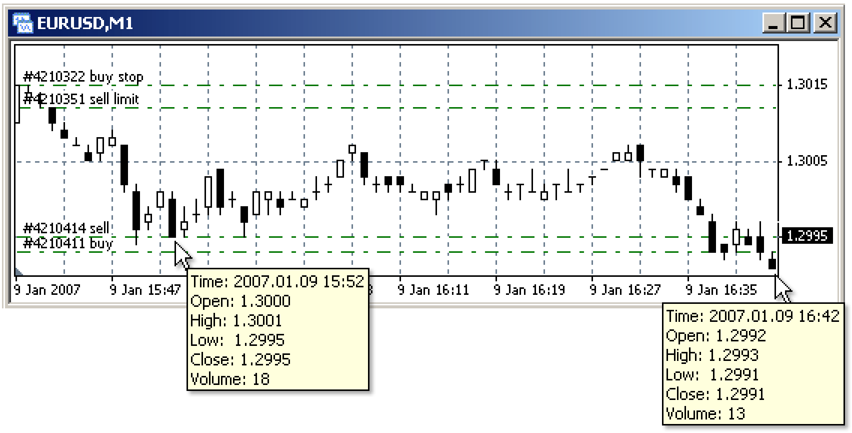


Fig. 72. Modification of pending orders into market ones.

In the further history, the 2 other pending orders were also modified into market ones.

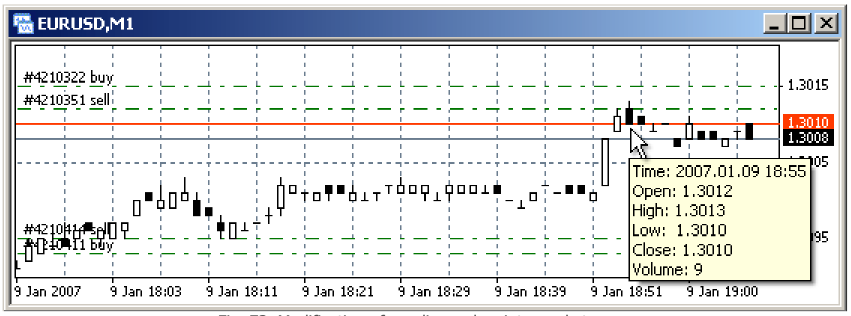
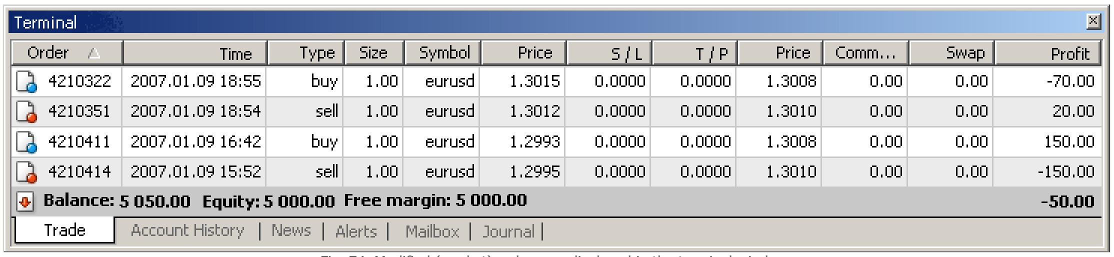


Fig. 73. Modification of pending orders into market ones.


Terminal

Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
4210322	2007.01.09 18:55	buy	1.00	eurusd	1.3015	0.0000	0.0000	1.3008	0.00	0.00	-70.00
4210351	2007.01.09 18:54	sell	1.00	eurusd	1.3012	0.0000	0.0000	1.3010	0.00	0.00	20.00
4210411	2007.01.09 16:42	buy	1.00	eurusd	1.2993	0.0000	0.0000	1.3008	0.00	0.00	150.00
4210414	2007.01.09 15:52	sell	1.00	eurusd	1.2995	0.0000	0.0000	1.3010	0.00	0.00	-150.00

Balance: 5 050.00 Equity: 5 000.00 Free margin: 5 000.00 -50.00

Fig. 74. Modified (market) orders are displayed in the terminal window.

Please note that Fig. 73 shows the opening of order Buy 4210322 (the former pending order BuyStop). As is easy to see, the bar formed at 18:55 does not touch the price of 1.3015. The highest price within this bar is 1.3013. At the same time, the Terminal window (Fig. 74) shows that the time of the pending order was modified into a market one within this specific bar, i.e., at 18:55.

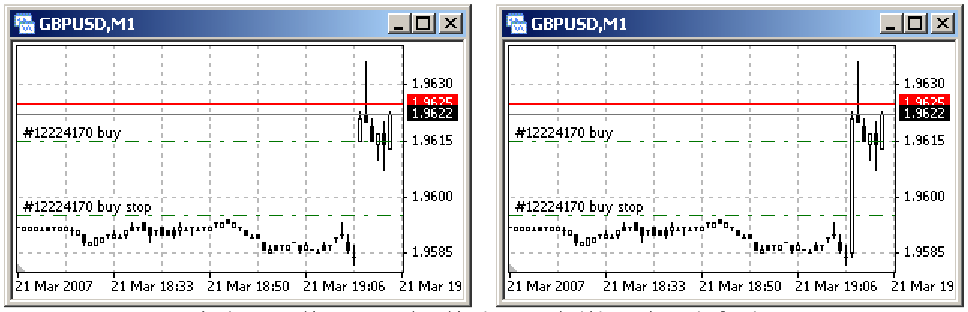
Here we must emphasize once again that the symbol window displays only the price history for the lower price of the two-side quote, namely, it reflects the Bid history. The Ask history is not displayed. This is why you may think that the pending order was modified into the market one by error. However, there is no error here, in this case. At the moment, when the price Bid was equal to 1.3013, the price Ask was 1.3013 + 2 = 1.3015 (2 means the spread of 2 points). Thus, the market price still touched the requested order price, and that resulted in automated modification of the pending order into a market one. The order was modified on the server side. Immediately after that, the server passed the information about this to the client terminal that, in its turn, displayed that information in both the symbol window (graphically) and in the terminal window (as a text).

Similar observations relate to the modification of order BuyLimit 4210411. Although the graphical displaying of the price touches or is below the requested price of pending order BuyLimit at 16:37-16:39 and at 16:41 (Fig. 72), no market order is opened. In this case, the reason for it is the same: the market price Ask did not touch the requested order price. However, it touched that level within the next bar, at 16:42. This event resulted in modifying of the pending order into a market one, so BuyLimit in the symbol window was replaced with Buy, while a new market order appears in the terminal window.

In the above example, all orders were placed with zero stop orders (it means without stop orders). However, the availability of content (non-zero) values of stop orders will not influence the modification of pending orders into market ones in any way, since they can only be modified if the corresponding price of the two-way quote touches or crosses the requested price of the pending order.

A pending order is modified into a market one without any relation to the stop orders attached.

A pending order can be opened (modified into a market one) at a price that does not match with the requested open price of the pending order. This can happen at rapid change of the market price, i.e., in the conditions when the latest known price before opening of the order has not reached the requested price, but the next price (at which the order is opened) does not match with the order open price, but is beyond it (Fig. 75).



a) price gapped between two bars b) price gapped within one bar at its forming
Fig. 75. Pending order is modified into a market one at a gap.

In Fig. 75a, we can see a possible variation of opening the pending order BuyStop (it shows two positions of the order - before and after opening; in the reality, you can see either BuyStop or Buy order, but not both). The latest known price before the price jumped up had been 1.9584. At 19:15, some news was published, which results in that the symbol price had changed in a jump. The first known price after the news has been released is 1.9615. Normally, prices jump up or down as a result of important news. In such cases, the broker cannot open your order at the requested price, because there are no corresponding prices in the market at the moment. In this case, the pending order BuyLimit was placed at the requested price of 1.9590, but it opened (modified to the market order) at the price of 1.9615. This resulted from the fact that there had not been any other prices within the range from 1.9584 to 1.9615.

As a result of the considered events, the market order Buy was opened at a price that was 25 points worse than that of the placed pending order BuyStop. A similar situation (receiving less profit than expected on the order) can take place for the order SellStop, if the price jumps down. However, if pending order BuyLimit or SellLimit get into the price gap, the corresponding market order can be opened at a price that is better for the trader than his or her requested price.

It must also be noted that a price gap (the difference between two nearest quotes that makes more than one point) occurs rather frequently and can arise at any time. If the price gap takes place between bars, i.e., a very different price incomes at the first tick of a new bar, you can see the price gap in the price chart (Fig. 75a). However, if the price gap happens within one bar, you cannot detect this gap visually (Fig. 75b). In this case, the gap is hidden within the bar (the candlestick). However, you cannot judge about the quotes history inside the bar just by its appearance or by any program characteristics available (however, you can detect the gap using an application program that wold calculate the difference between the prices of incoming quotes).

Modification of Market Orders

Trading platform MetaTrader 4 allows you to form trade requests to modify price levels of market and pending orders.

To modify orders of any types, including market orders, you should use function OrderModify().

Modification of a market order implies changing of the requested values of stop orders. You may not change open prices in market orders.

You cannot change the open price of a market order, since order opening is a fact. Therefore, there is no any programming method to do this. The only thing you can do with a market order is to close it. A market order can be closed as a result of the execution of a trade request formed by a trader or by a program, or if the market price reaches the requested price of one of stop orders.

Orders StopLoss and TakeProfit cannot be placed closer to the market price than at the minimum distance.
An order cannot be modified, if the execution price of its StopLoss or TakeProfit ranges within the freeze distance from the market price.

Please note that the position of the stop orders of a market order is limited as related to the current market price, but it has no relation to the order open price (see Requirements and Limitations in Making Trades). This means that the modification of the order may result in that the stop orders are placed above or below the market order open price.

Let's consider an example. A market order was opened before, its stop orders being closest to the market price (Fig. 69). After that, the market price changed (increased by 1 point). At the moment shown in Fig. 76, it became possible to change the value of TakeProfit. Order Sell is closed at the latest known price Ask. The distance between Ask=1.2990 and the preceding value TakeProfit=1.2984 made 6 points, i.e., exceeded the minimum allowed distance. Trader (or program) formed a trade request to change the value of TakeProfit, namely, to increase this value by 1 point. This resulted in making a trade by changing the position of the stop order of the market order (the preceding value of TakeProfit=1.2984, the new value = 1.2985).

If the trade request contained the instruction to modify order Sell so that the value of any stop orders were closer to the market price Ask than at the minimum distance, this trade request would be rejected by the client terminal and the trade would not be made.

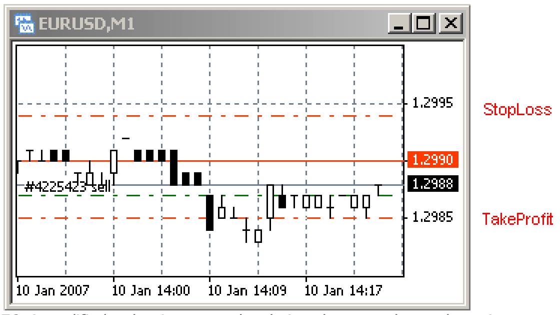


Fig. 76. A modified order, its stop orders being closest to the market price.

The modification rule for market orders limits the stop-order approaching to the current price, but it does not limit the stop-order distance from the price. This is why stop orders can be placed at any distance from the current price, if this distance is larger than the limiting distance (if, as of the order modification moment, the stop order is outside the freeze band determined by the value). In Fig. 77, we can see the same order after one more modification: in this case, stop orders are well out of the range of the limiting minimum distance.

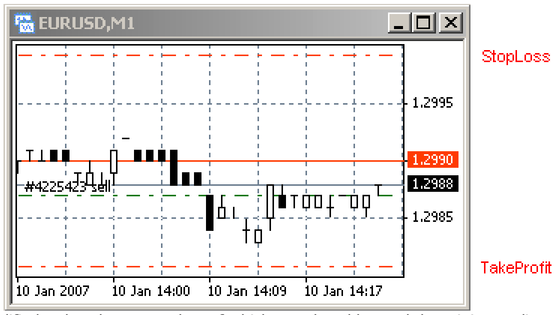


Fig. 77. A modified order, the stop orders of which are placed beyond the minimum distance.

Modification of Pending Orders

To modify any order types, including pending orders, we use function OrderModify().

Modification of a pending order implies the possibility to change the defined values of open prices of the pending order and its stop orders.
The limitation related to the position of the pending order to be modified is calculated on the basis of the correct market price of modifying the pending order into the market one.
The limitation related to the position of stop orders of a pending order is calculated on the basis of the requested open price of the pending order and has no relation to market prices.

Pending orders BuyLimit and BuyStop cannot be placed closer to the market price Ask than at the minimum distance StopLevel.
Pending orders SellLimit and SellStop cannot be placed closer to the market price Bid than at the minimum distance StopLevel.
StopLoss/TakeProfit of a pending order cannot be placed closer to the requested order open price than at the minimum distance StopLevel.

Pending orders BuyLimit and BuyStop cannot be modified, if the requested order open price ranges within the freeze distance from the market price Ask.
Pending orders SellLimit and SellStop cannot be modified, if the requested order open price ranges within the freeze distance from the market price Bid.
The positions of StopLoss and TakeProfit of pending orders are not limited by the freeze distance FreezeLevel.

Please note that the requested price of a pending order is limited as related to the market price, whereas its stop orders are limited by the requested open price of the pending order (see Requirements and Limitations in Making Trades).

For example, pending order BuyLimit is placed with the following parameters: requested price=1. 2969, StopLoss=1.2964, TakeProfit=1.2974. The current value of the market price (applied to modify the pending order into a market one) Ask=1.2983. Thus, the order is placed at a distance of 14 points (1.2983-1.2969) from the market price, which by far exceeds the minimum allowed distance. The stop orders are at the distance of 5 points from the requested price, which does not exceed the minimum distance, so it is allowable.

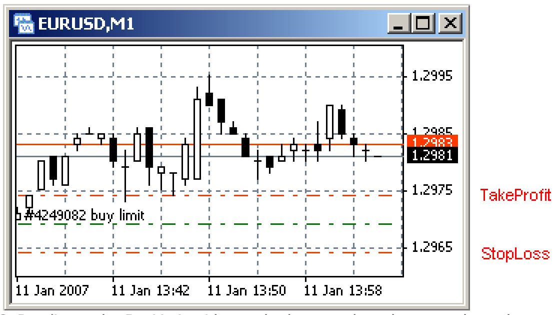


Fig. 78. Pending order BuyLimit with attached stop orders closest to the order.

If the trader needs to change the price of order BuyLimit, then, whatever direction he or she moves it in, in this case, it is necessary to simultaneously change the position of the corresponding stop order (or to delete it, i.e., to set zero value for it), too. Otherwise, the distance between the order and its stop order may turn out to be less that the minimum allowed one. The trader decided to modify the order so that it kept the distance between the order and its TakeProfit as 5 points, while the value of StopLoss remained as it was (Fig. 79).

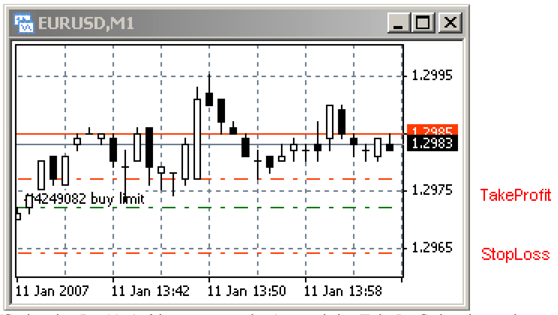


Fig. 79. Modified order BuyLimit (the requested price and the TakeProfit level are changed).

If the trader needs to place pending order BuyLimit as close to the market price as possible, then in this case (Fig. 80), the minimum allowed value of the requested price Ask-5points = 1.2985-0.0005 = 1.2980. In this example, stop orders are placed outside the limiting minimum distance.

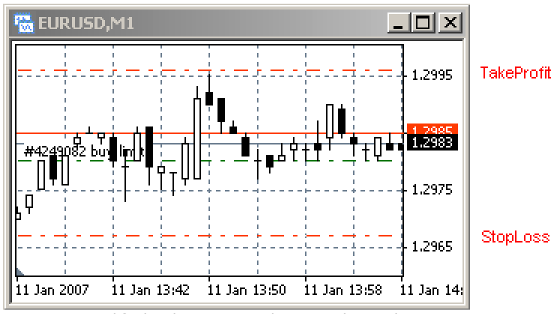


Fig. 80. Modified order BuyLimit closest to the market price.

Appendix named Requirements and Limitations in Making Trades contains a summary table that specifies the matched values of a two-way quote, which are used for opening, closing or modifying of orders, as well as other reference values that limit the making of trades.

← Common Way of Making Trades Opening and Placing Orders →

Opening and Placing Orders

Trade requests for opening and placing pending orders are formed using the function OrderSend().

Function OrderSend()

```
int OrderSend (string symbol, int cmd, double volume, double price, int slippage, double stoploss,
double takeprofit, string comment=NULL, int magic=0, datetime expiration=0, color arrow_color=CLR_NONE)
```

(please note that here and below, we refer to function header, not to an example of how to use function call in a program).

Let's consider in more details what this function consists of.

OrderSend is the function name. The function returns the ticket number ('ticket' is the unique number of an order) that is assigned to the order by the trade server, or -1, if the trade request has been rejected by either the server or the client terminal. In order to get information about the reasons for rejection of the trade request, you should use the function GetLastError() (below we will consider some of the most common errors).

symbol is the name of the traded security. Each symbol corresponds with the value of a string variable. For example, for the currency pair of Euro/US dollar, this value is "EURUSD". If the order is being opened for a foregone symbol, then this parameter can be specified explicitly: "EURUSD", "EURGBP", etc. However, if you are going to use the Expert Advisor in the window of any other symbol, you can use the standard function Symbol(). This function returns a string value that corresponds with the name of the symbol, in the window of which the EA or script is being executed.

cmd is the type of operation. The type of operation can be specified as a predefined constant or its value, and according to the type of the trade.

volume is the amount of lots. For market orders, you must always check the account for the sufficiency. For pending orders, the amount of lots is not limited.

price is the open price. It is specified according to the requirements and limitations accepted for making trades (see Order Characteristics and Rules for Making Trades). If the price requested for opening of market orders has not been found in the price thread or if it has considerably outdated, the trade request is rejected. However, if the price is outdated, but present in the price thread and if its deviation from the current price ranges within the value of slippage, this trade request will be accepted by the client terminal and sent to the trade server.

slippage is the maximum allowed deviation of the requested order open price from the market price for market orders (points). This parameter is not processed for placing of pending orders.

stoploss is the requested close price that determines the maximum loss allowed for the given trade. It is set according to the requirements and limitations accepted for making trades (see Order Characteristics and Rules for Making Trades, Requirements and Limitations in Making Trades).

takeprofit is the requested close price that determines the maximum profit for the given trade. It is set according to the requirements and limitations accepted for making trades (see Order Characteristics and Rules for Making Trades, Requirements and Limitations in Making Trades).

comment is the text of the order comment. The last part of the comment can be modified by the trade server.

magic is the magic number of the order. It can be used as the user-defined order identifier. In some cases, it is the only information that helps you to find out about that the order belongs to one or another program that has opened it. The parameter is set by the user; its value can be the same or other than the value of this parameter of other orders.

expiration is the date when the order expires. As soon as this day comes, the pending order will be closed automatically on the server side. On some trade servers, there may be a prohibition for setting the expiration date for pending orders. In this case, if you try to set a non-zero value of the parameter, the request will be rejected.

arrow_color is the color of the opening arrow in the chart. If this parameter is absent or if its value is CLR_NONE, the opening arrow is not shown in the chart at all.

On some trade servers, there can be a limit set for the total amount of opened and pending orders. If this limit is exceeded, any trade request that implies opening a market order or placing a pending order will be rejected by the trade server.

Opening Market Orders

The function OrderSend() may at first seem to be too intricate. However, all considered parameters are quite simple, helpful and can be successfully used in your trading. In order to see this for ourselves, let's consider the simplest variation of how the trade function OrderSend() is used for opening a market order.

First of all, we should note that function OrderSend() has predefined parameters (see Function Call and Function Description and Operator 'return'). This means that this function can be used in a simplified mode using the minimum required set of parameters. These parameters are as follows:

symbol is a necessary parameter, because we need to know where to open the order. Let our script imply the possibility to open an order in any symbol window. In this case, we will substitute the standard function Symbol() as this parameter;

cmd - for example, let's open a Buy order; in this case, we will specify parameter OP_BUY;

volume - we can specify any value allowed by the rules; let's open a small order, for example, of 0.1 lot;

price - open price for the order Buy is price Ask;

slippage is usually specified as 0-3 points. Let's specify 2;

stoploss - stop orders can be placed at a distance that is not closer than the minimum allowed distance, normally 5 points (see Requirements and Limitations in Making Trades); let's place stop orders at a distance of 15 points from the close price, namely: Bid - 15*Point;

takeprofit - let's place stop orders at a distance of 15 points from the close price, namely: Bid + 15*Point;

Below is the simplest script, simpleopen.mq4, that is intended for opening a Buy order:

```
//--------------------------------------------------------------------
// simpleopen.mq4
// The code should be used for educational purpose only.
```

```
//--------------------------------------------------------------------
int start()                                  // Special function start()
  {                                          // Opening BUY
   OrderSend(Symbol(),OP_BUY,0.1,Ask,3,Bid-15*Point,Bid+15*Point);
   return;                                   // Exit start()
  }
//--------------------------------------------------------------------
```

If you launch this script for execution, it will work, in the majority of cases. The script consists of one special function that contains the order-opening function OrderSend() and the operator 'return'. Let's describe the execution algorithm for program lines and events related to that.

1. The user has attached the script to the symbol window by dragging the script name with the mouse button from the "Navigator" window of the client terminal into the window of the symbol, for which he or she wants to open a market order Buy of 0.1 lot and with stop orders that are at a distance of 15 points from the market price.

2. At the moment of attaching the script to the symbol window, the client terminal is passing the control (just by launching it) o the special function start() (here we should briefly remind that the start() of a script is launched at the moment of attaching the script to the symbol window, whereas the start() of an EA is launched at the moment when the nearest tick incomes for the symbol).

3. Within the framework of execution of the special function start(), the control is passed to the line that calls the order opening function:

```
   OrderSend(Symbol(),OP_BUY,0.1,Ask,3,Bid-15*Point,Bid+15*Point);
```

Before execution of this function, the program calculates the values of all formal parameters:

3.1. We attached the script to the window of Eur/USd.In this case, the standard function Symbol() will return the string value EURUSD.

3.2. Let Ask =1.2852 and Bid =1.2850 as of the moment of calling this function.

3.3. The value of StopLoss, in this case, will be: 1.2850-15*0.0001 = 1.2835, whereas TakeProfit = 1.2865.

4.Execution of the function OrderSend():

4.1. The function formed a trade request for opening of an order and passed this request to the client terminal.

4.2. The function passed the control to the client terminal simultaneously with passing of the trade request, so the program execution was stopped.

4.3. The client terminal checked the received trade request. It didn't detect any incorrect parameters, so it sent the request to the server.

4.4. The server received the trade request, checked it, didn't detect any incorrect parameters, and decided to execute the request.

4.5. The server executed the request by making a transaction in its database and sent the information about that executed request to the client terminal.

4.6. The client terminal received the information about that the last trade request had been executed, displayed this event in the terminal window and in the symbol window, and returned the control to the program.

4.7. Once having received the control, the program continued working from the location, from which the control had previously been passed to the client terminal (and to which it had been returned later).

Please note that no actions were performed in the program starting from step 4.2 through step 4.7 - the program was in the mode of waiting for the server response.

5. The control in the program is passed to the next operator - the operator 'return'.

6. The execution of the operator 'return' results in exiting the function start() and, therefore, in termination of the program execution (it should be reminded that scripts complete their work after they have being executed) - the control is returned to the client terminal.

Thus, the script has fulfilled its intended purpose: order Buy with the preset parameters is opened. The use of scripts is very convenient, if you need to perform a small one-time operation; in this case, the use of a script is quite reasonable. According to step 4.6., the trader can see the order in the screen.

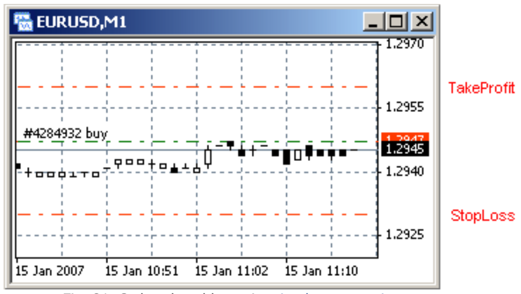


Fig. 81. Order placed by script simpleopen.mq4.

The events are not always ordered as shown above. It is possible that the trade request is rejected by the client terminal or by the server. Let's try to make some experiments, for example, change the symbol name: specify "GBPUSD" explicitly (this is quite allowable). We will obtain a program with the limited field of use:

```
int start()                               // Special function start
  {                                       // Opening BUY
   OrderSend("GBPUSD",OP_BUY,0.1,Ask,3,Bid-15*Point,Bid+15*Point);
   return;                                // Exit start()
  }
```

Let's launch the script in the same window of symbol Eur/Usd. The script was intended to open an order in the window of Gbp/Usd. However, after it had been attached to the window of Eur/Usd, no order was opened in the window of Gbp/Usd.

A disadvantage of such programs is their functional limitation. In this case, once having attached the script to the symbol window, the user is just waiting for order opening. However, the order is not opened. The user is not aware of the reason why it is so: either it is caused by an algorithmic error in the program code or the trade request is "lost" by the way to the server, or the trade request has been rejected by the client terminal long time ago (thought the user is still waiting), or there is another reason.

In order to provide the user (and, which is also very important, the program) with the information about the events related to the execution of the trade request, it is necessary to process the errors.

Error Processing

A very important property of the client terminal is that, if an error occurs during the execution of an application, the client terminal does not stop the execution of the program. Errors are usually caused by the imperfection of the algorithm used in the application. In some cases, errors are caused by some external (as related to the program) factors. The internal causes of errors are any violations of the MQL4 requirements or of trading rules, for example, using invalid prices. The external causes are those that are not related to the application program, for example, interrupted connection.

If an error occurs at the execution of a program, the program will continue running, while the client terminal will generate the error code value available to the program through the function GetLastError().

Function GetLastError()

```
int GetLastError()
```

The function returns the code of the newly occurred error, then the value of special variable last_error that stores the code of the last error will be zeroized. The subsequent GetLastError() call will return 0.

Hereinafter, we will identify all occurring errors by this code. Several errors can occur during the execution of a program; function GetLastError() allows us to get the code value for only one of them, the latest error, this is why every time when we need this information, it is recommended to use function GetLastError() immediately after the program lines, in which the error may occur.

Error 130. Invalid Stop Orders

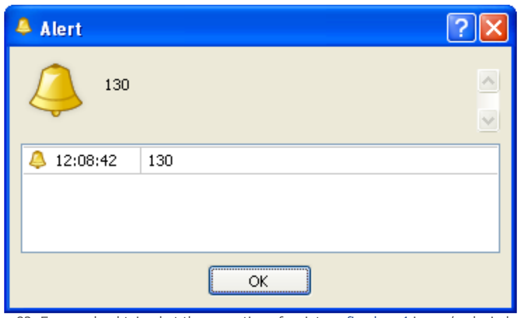
The last considered script does not analyze errors, so the user remains ignorant about the results of the execution of order-opening function. In the simple variation of using the function GetLastError(), the program can analyze an error and just inform the user about it. If you launch script confined.mq4 for execution in the window of Eur/Usd, an error will occur.

```
//--------------------------------------------------------------------------
// confined.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------
int start()                                      // Special function start
  {                                              // Opening BUY
   OrderSend("GBPUSD",OP_BUY,0.1,Ask,3,Bid-15*Point,Bid+15*Point);
   Alert (GetLastError());                       // Error message
   return;                                       // Exit start()
  }
//--------------------------------------------------------------------------
```

We added only one, but very informative line into this script:

```
   Alert (GetLastError());                       // Error message
```

Function GetLastError() returns the code of the last error, whereas Alert() is used to display this value on the screen. After script confined.mq4 has been attached to the window of symbol Eur/Usd, the script will be executed, which will result in that the user will see the following message:



130

12:08:42 130

Fig. 82. Error code obtained at the execution of script confined.mq4 in eur/usd window.

You can find in Appendixes codes of errors that can occur at the execution of a program. In this case, error 130 (invalid stop orders) occurred. This means that the values of formal parameters used in the function OrderSend() don't comply with the limitations specified in Requirements and Limitations in Making Trades. Upon a closer view, we can see the reason that caused the error: the current values of market prices Bid and Ask are taken from the symbol window, to which the script is attached, namely, from the window of Eur/Usd. However, these values are used to form a trade request for Gbp/Usd. As a result, at the current price of Gbp/Usd, Ask = 1.9655, the value of TakeProfit for the newly opened market order turns out to be equal to (for Eur/Usd Bid =1.2930) 1.2930+15*0.0001=1. 2945, which is considerably lower than the minimum allowed value, i.e., it is invalid.

In this case, an algorithmic error occurred. In order to correct it, you should use the correct values of symbol prices. You can obtain these values using the function

MarketInfo(). Script improved.mq4 that opens market orders for Gbp/Usd can be launched in any symbol window:

```
//--------------------------------------------------------------------------------
// improved.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------------
int start()                                     // Special function start
  {
   double bid   =MarketInfo("GBPUSD",MODE_BID); // Request for the value of Bid
   double ask   =MarketInfo("GBPUSD",MODE_ASK); // Request for the value of Ask
   double point =MarketInfo("GBPUSD",MODE_POINT);//Request for Point
   // Opening BUY
   OrderSend("GBPUSD",OP_BUY,0.1,ask,3,bid-15*Point,bid+15*Point);
   Alert (GetLastError());                      // Error message
   return;                                      // Exit start()
  }
//--------------------------------------------------------------------------------
```

The above error does not occur at the execution of this script, so its execution will result in displaying the corresponding message: 0 (zero). This means that the function GetLastError() returned the value of 0, i.e., no errors were detected in the execution of the trade request by the client terminal.

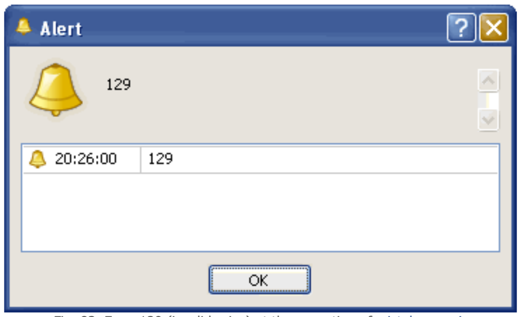
Let's also consider some other common errors. For this, let's return to the idea of opening an order using a script in the same window, to which the script is attached.

Error 129. Invalid Price

In some cases, a simple error occurs - the wrong value of the two-way quote is specified as the open price. Market orders Buy are known (see Requirements and Limitations in Making Trades) to be opened at the Ask price. Below is shown what happens if we, by mistake, specify the Bid price in script mistaken.mq4:

```
//---------------------------------------------------------------------------
// mistaken.mq4
// The code should be used for educational purpose only.
//---------------------------------------------------------------------------
int start()                                     // Special function start
  {                                             // Opening BUY
   OrderSend(Symbol(),OP_BUY,0.1,Bid,3,Bid-15*Point,Bid+15*Point);
   Alert (GetLastError());                      // Error message
   return;                                      // Exit start()
  }
//---------------------------------------------------------------------------
```

Before sending the trade request to the server, the client terminal analyzes whether the requested values of price and stop orders comply with the allowed values. During this check, the requested open-order price will be detected as invalid, so the client terminal will not send the trade request to the server for execution, and function GetLastError() will return the value of 129 (see Error Codes). The execution of the script will result in appearance of the corresponding error message:



129

20:26:00 129

Fig. 83. Error 129 (invalid price) at the execution of mistaken.mq4.

Error 134. Not Enough Money for Making a Trade

A similar result (error 134) will be obtained, if there are not enough free money on the account to open an order. You can know about the amount of free money required to open 1 lot for buying of each symbol using the function MarketInfo(symbol_name, MODE_MARGINREQUIRED).

The size of one standard lot for the same symbol may vary in different dealing centers.

The required amount of free assets for opening a one-lot order is inversely proportional to the amount of the provided leverage. At the same time, the cost of 1 point in the deposit currency for a symbol does not relate to the provided leverage.

Table 3. Possible combinations of 1-lot cost and 1-point cost (deposit currency is US dollar).

	Dealing Center 1			**Dealing Center 2**			**Dealing Center 3**		
	Buy	Sell	1pt	Buy	Sell	1pt	Buy	Sell	1pt
EUR/USD	1296.40	1296.20	10.00	1296.50	1296.20	10.00	1000.00	1000.00	10.00
GBP/USD	1966.20	1966.00	10.00	1376.48	1376.20	7.50	1000.00	1000.00	10.00
AUD/USD	784.40	784.20	10.00	1569.20	1568.40	20.00	1000.00	1000.00	10.00
USD/JPY	1000.00	1000.00	8.29	1000.00	1000.00	8.29	1000.00	1000.00	8.29

USD/CHF	1000.00	1000.00	8.02	1000.00	1000.00	8.02	1000.00	1000.00	8.02
EUR/CHF	1296.40	1296.20	8.02	1296.35	1296. 35	8.02	1000.00	1000.00	8.02

Prices are given as of 16.12.2007.

Let's briefly consider some common methods of calculating the cost of 1 lot and of 1 point.

Dealing Center 1 (most common)

For the symbols that have USD reciprocally, the cost of 1 lot is equal to the current price of the corresponding two-way quote multiplied by 1000, whereas the cost of 1 point is equal to $10.

For the symbols that have USD as their numerator, the cost of 1 lot is equal to $1000.00, whereas the cost of 1 point is inversely proportional to the current quote and equal to 1/(Bid). For example, for USD/CHF, at Bid= 1.2466, the cost of 1 point is 1/1. 2466 = 8.02.

For cross rates, the cost of 1 lot is calculated in the same way as that of the numerator currency, whereas the cost of 1 point is calculated in the same way as that for the denominator currency. For example, for EUR/CHF, the cost of 1 lot is 129.40 (as for EUR/USD), whereas the cost of 1 lot is 8.02 (as for USD/CHF).

Dealing Center 2

In some dealing centers, considering the same rule of calculating costs, the values of costs can be different for some symbols. For example, the cost of 1 lot and the cost of 1 point may be proportionally increased or decreased. For example, this factor can be 0.75 for GBP/USD, whereas it is 2.0 for AUD/USD. Such representation of cost values does not result in any economical changes; in such cases, you just have to consider this special feature when calculating costs of your orders. You should also pay attention to the fact that the 1-lot costs for buying and selling of assets at cross rates are the same.

Dealing Center 3

there are also dealing centers that set the cost of 1 lot as $1000.00 for any symbol. At the same time, the cost of 1 point remains proportional to the current prices. This implies setting a special leverage for each symbol.

1-point cost of all symbols that are not quoted as related to USD always changes proportionally to the cost of the symbol specified reciprocally.

Generally, there can exist other principles of building cost values. It is needless to say that, prior to start real trading, you should find out about the calculation method for any specific dealing center and consider this method in your coding.

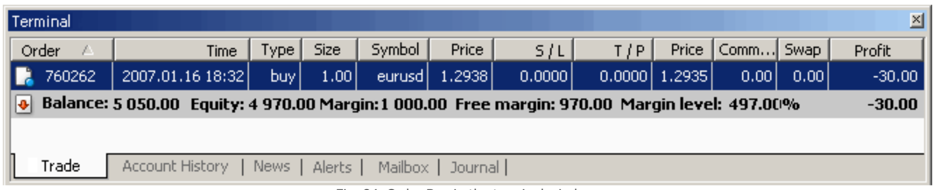
Free Margin

At coding, it is very important to consider the principle of forming free assets. Free margin (assets) is the amount of money that is available for making trades.

Let's consider an example. Let Balance be 5000.00, there are no open orders in the terminal. Let's open a Buy order of 1 lot in dealing center 3. The following rule is stated in dealing center 3:

If differently directed market orders are opened for one symbol, the smaller integrated cost of one-direction orders is released for trading and increases the amount of free assets (this rule is not applicable for all dealing centers).

The terminal window will display the information about the opened order. Please note that the margin makes 1000.00, order profit is -30.00, therefore the amount of free assets (free margin) makes 5000-1000-30=3970.00:


Terminal

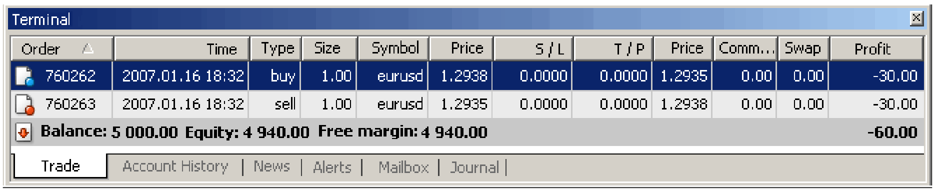
Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
760262	2007.01.16 18:32	buy	1.00	eurusd	1.2938	0.0000	0.0000	1.2935	0.00	0.00	-30.00

Balance: 5 050.00 Equity: 4 970.00 Margin: 1 000.00 Free margin: 970.00 Margin level: 497.00% -30.00

Trade | Account History | News | Alerts | Mailbox | Journal

Fig. 84. Order Buy in the terminal window.

After a Sell order of the same value has been opened, free margin will increase. The smaller integrated cost of one-direction market orders makes 1000.00, so the free margin will increase by 1000.00. In Fig. 85, you can see the situation where the differently directed orders cost the same value, so the entire sum of orders costs is released for trading.


Terminal

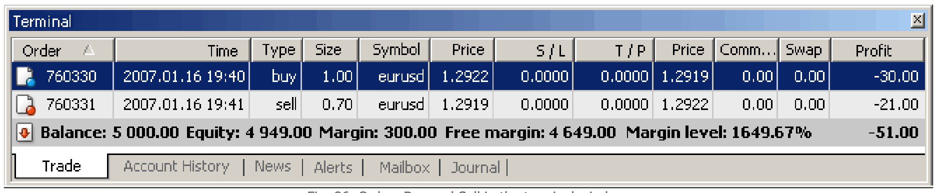
Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
760262	2007.01.16 18:32	buy	1.00	eurusd	1.2938	0.0000	0.0000	1.2935	0.00	0.00	-30.00
760263	2007.01.16 18:32	sell	1.00	eurusd	1.2935	0.0000	0.0000	1.2938	0.00	0.00	-30.00

Balance: 5 000.00 Equity: 4 940.00 Free margin: 4 940.00 -60.00

Trade | Account History | News | Alerts | Mailbox | Journal

Fig. 85. Orders Buy and Sell in the terminal window.

After a Sell order of smaller cost has been opened, free margin will increase, as well. In this case, the smaller integrated cost of one-direction market orders makes 700.00, so the free margin will increase by 700.00, whereas the margin makes the difference between the integrated costs of differently directed orders (Fig. 86).


Terminal

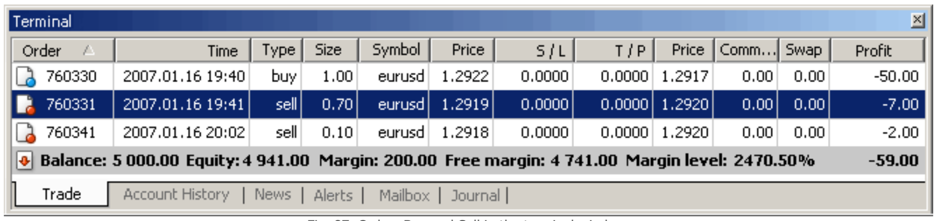
Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
760330	2007.01.16 19:40	buy	1.00	eurusd	1.2922	0.0000	0.0000	1.2919	0.00	0.00	-30.00
760331	2007.01.16 19:41	sell	0.70	eurusd	1.2919	0.0000	0.0000	1.2922	0.00	0.00	-21.00

Balance: 5 000.00 Equity: 4 949.00 Margin: 300.00 Free margin: 4 649.00 Margin level: 1649.67% -51.00

Trade | Account History | News | Alerts | Mailbox | Journal

Fig. 86. Orders Buy and Sell in the terminal window.

If one more order Sell of 0.1 lot is opened (cost 100.00), the smaller integrated cost of one-direction market orders makes 700.00 + 100. 00 = 800.00. Thus, the margin (as compared to the situation where only one order Buy is opened) decreases by 800.00. As compared to the situation shown in Fig. 86, the margin decreases, whereas the equity increases by 100.00 (see Fig. 87).


Terminal

Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
760330	2007.01.16 19:40	buy	1.00	eurusd	1.2922	0.0000	0.0000	1.2917	0.00	0.00	-50.00
760331	2007.01.16 19:41	sell	0.70	eurusd	1.2919	0.0000	0.0000	1.2920	0.00	0.00	-7.00
760341	2007.01.16 20:02	sell	0.10	eurusd	1.2918	0.0000	0.0000	1.2920	0.00	0.00	-2.00

Balance: 5 000.00 Equity: 4 941.00 Margin: 200.00 Free margin: 4 741.00 Margin level: 2470.50% -59.00

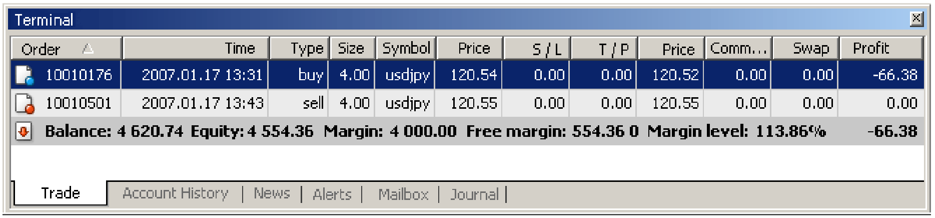
Fig. 87. Orders Buy and Sell in the terminal window.

Free Margins shown in Fig. 86 and Fig. 87 differ from each other by more than 100.00, since the integrated profit of open orders has changed with change in the current price (the difference makes 8.00).

If we make similar manipulations in another dealing center, it's easy to see that the above order of forming the value of free margin is not kept. For some dealing centers, the following rule is effective:

Opening of any market orders does not release the equity or increase the free margin. Opening of market orders increases the equity by the amount that exceeds the integrated cost of differently directed market orders for a symbol (the rule does not apply in all dealing centers).

For example, if you have previously opened an order Buy of 4 lots for USD/JPY in dealing center 2, the amounts of equity and free margin will not change at opening of a 4-lot Sell order.


Terminal

Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
10010176	2007.01.17 13:31	buy	4.00	usdjpy	120.54	0.00	0.00	120.52	0.00	0.00	-66.38
10010501	2007.01.17 13:43	sell	4.00	usdjpy	120.55	0.00	0.00	120.55	0.00	0.00	0.00

Balance: 4 620.74 Equity: 4 554.36 Margin: 4 000.00 Free margin: 554.36 0 Margin level: 113.86% -66.38

Fig. 88. The presence of differently directed orders does not release equity.

You can make calculations to know whether the current equity is enough for opening of an order. You can also use the function AccountFreeMarginCheck() that returns the value of free margin to remain after opening of a market order with certain amount of lots for a certain symbol. If the returned value is equal or more than 0, there are enough money on the account. If it is less than 0, then the order of this volume and for this symbol cannot be opened, the client terminal will return error 134.

In order to know the conditions offered by the dealing center and the amount of free margin required for opening of an order with the volume of 1 lot, you can use a simple script, conditions.mq4:

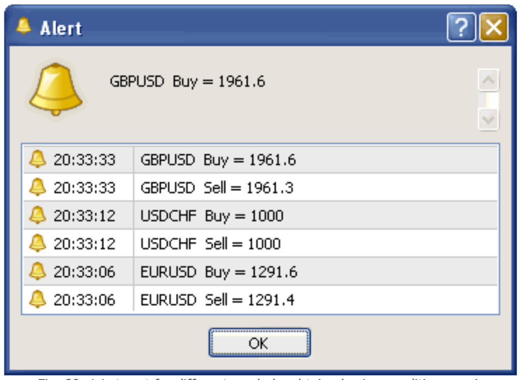
```
//--------------------------------------------------------------------------
// conditions.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------
int start()                                         // Special function start
  {
   Alert(Symbol(),"  Sell = ",AccountFreeMargin()// At selling
          -AccountFreeMarginCheck(Symbol(),OP_SELL,1));
   Alert(Symbol(),"  Buy = ",AccountFreeMargin() // At buying
          -AccountFreeMarginCheck(Symbol(),OP_BUY,1));
   return;                                          // Exit start()
  }
//--------------------------------------------------------------------------
```

Here, the expression of

```
AccountFreeMargin() - AccountFreeMarginCheck(Symbol(),OP_SELL,1)
```

allows us to calculate the difference between the available free margin and the free margin that will remain after opening of the order.

If we start this script for execution, when there are no market orders in the terminal, we can obtain the currently required amount of equity to be available and enough for opening of an order with the volume of 1 lot for buying and for selling:


Alert

GBPUSD Buy = 1961.6

```
20:33:33   GBPUSD  Buy = 1961.6
20:33:33   GBPUSD  Sell = 1961.3
20:33:12   USDCHF  Buy = 1000
20:33:12   USDCHF  Sell = 1000
20:33:06   EURUSD  Buy = 1291.6
20:33:06   EURUSD  Sell = 1291.4
```

Fig. 89. 1-Lot cost for different symbols, obtained using conditions.mq4.

If we launch the script conditions.mq4 for execution in the window of the symbol, for which there are opened market orders, we can obtain other values, it depends on the calculation methods accepted in one or another dealing center.

Other Errors and Function MarketInfo()

There are other limitations related to determining of values of parameters of function OrderSend(). This are the maximum and the minimum order price step, the maximum and the minimum order price value, etc. The use of function MarketInfo() allows you to get various information about symbols that are shown in the window "Market Watch" of the client terminal.

Function MarketInfo()

```
double MarketInfo(string symbol, int type)
```

The function returns various information about symbols listed in the window "Market Watch" of the client terminal. Parts of information about the current symbol are stored in predefined variables.

Parameters:

symbol - the name of a symbol;

type - request identifier that determines the type of information to be returned. It can be either value of those of request identifiers (see Function MarketInfo Identifier).

Some errors may occur for the reasons on the server side. For example, in the conditions of transient prices, your broker may increase the minimum distance that limits placement of pending orders and stop orders. Further, at a calm market, the broker can decrease this distance again. Thus, the values of some parameters can be changed at any time.
For the program to operate in a stable manner, with the minimal amount of rejected requests, you should update the parameters of information environment used by the program using the functions MarketInfo() and RefreshRates() before you execute the function OrderSend().

An example of a simple script that opens a Buy order costing 35% of the free margin, with some preset values of stop orders (openbuy.mq4).

```
//--------------------------------------------------------------------
// openbuy.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
int start()                                     // Special function start
  {
   int Dist_SL =10;                             // Preset SL (pt)
   int Dist_TP =3;                              // Preset TP (pt)
   double Prots=0.35;                           // Percentage of free margin
   string Symb=Symbol();                        // Symbol
//--------------------------------------------------------------- 2 --
   while(true)                                  // Cycle that opens an order
     {
      int Min_Dist=MarketInfo(Symb,MODE_STOPLEVEL);// Min. distance
      double Min_Lot=MarketInfo(Symb,MODE_MINLOT);// Min. volume
      double Step   =MarketInfo(Symb,MODE_LOTSTEP);//Step to change lots
      double Free   =AccountFreeMargin();       // Free Margin
      double One_Lot=MarketInfo(Symb,MODE_MARGINREQUIRED);//Cost per 1 lot
      //--------------------------------------------------------- 3 --
      double Lot=MathFloor(Free*ProtsOne_LotStep)*Step;// Lots
      if (Lot < Min_Lot)                        // If it is less than allowed
        {
         Alert(" Not enough money for ", Min_Lot," lots");
         break;                                 // Exit cycle
        }
      //--------------------------------------------------------- 4 --
      if (Dist_SL < Min_Dist)                   // If it is less than allowed
        {
         Dist_SL=Min_Dist;                      // Set the allowed
         Alert(" Increased the distance of SL = ",Dist_SL," pt");
        }
      double SL=Bid - Dist_SL*Point;            // Requested price of SL
      //--------------------------------------------------------------- 5 --
      if (Dist_TP < Min_Dist)                   // If it is less than allowed
        {
         Dist_TP=Min_Dist;                      // Set the allowed
         Alert(" Increased the distance of TP = ",Dist_TP," pt");
        }
      double TP=Bid + Dist_TP*Point;            // Requested price of TP
      //--------------------------------------------------------------- 6 --
      Alert("The request was sent to the server. Waiting for reply..");
      int ticket=OrderSend(Symb, OP_BUY, Lot, Ask, 2, SL, TP);
      //--------------------------------------------------------------- 7 --
      if (ticket>0)                             // Got it!:)
        {
         Alert ("Opened order Buy ",ticket);
         break;                                 // Exit cycle
        }
      //--------------------------------------------------------------- 8 --
      int Error=GetLastError();                 // Failed :(
      switch(Error)                             // Overcomable errors
        {
         case 135:Alert("The price has changed. Retrying..");
            RefreshRates();                     // Update data
            continue;                           // At the next iteration
         case 136:Alert("No prices. Waiting for a new tick..");
            while(RefreshRates()==false)        // Up to a new tick
               Sleep(1);                        // Cycle delay
            continue;                           // At the next iteration
         case 146:Alert("Trading subsystem is busy. Retrying..");
            Sleep(500);                         // Simple solution
            RefreshRates();                     // Update data
            continue;                           // At the next iteration
        }
      switch(Error)                             // Critical errors
        {
         case 2 : Alert("Common error.");
            break;                              // Exit 'switch'
         case 5 : Alert("Outdated version of the client terminal.");
            break;                              // Exit 'switch'
         case 64: Alert("The account is blocked.");
            break;                              // Exit 'switch'
         case 133:Alert("Trading forbidden");
            break;                              // Exit 'switch'
         default: Alert("Occurred error ",Error);// Other alternatives
        }
      break;                                    // Exit cycle
     }
//-------------------------------------------------------------------- 9 --
   Alert ("The script has completed its operations ---------------------------");
   return;                                      // Exit start()
  }
//-------------------------------------------------------------------- 10 --
```

The script consists of one special function start() (blocks 1-10). In block 1-2, the values are set, at which the order must be opened. Block 2-9 represents cycle operator while(), in which all necessary calculations are performed. This cycle is included into the code to allow the program make several attempts to open the order. In block 2-3, the environment variables are updated. In blocks 3-4-5-6, the amount of lots and the requested prices of stop orders are calculated. In block 7-8-9, errors are processed. In block 9-10, the message is printed that the script has completed its operations.

Let's consider some special features of a program code. It's easy to see that the trade request is formed in block 6-7. In block 3-4, the amount of lots is calculated. It also considers the situation when the available free margin is insufficient to open even an order with the minimum amount of lots. This is why, in block 3-4, after printing the message about insufficient money, we exit cycle 2-9 using the operator 'break'. The control is passed to block 9-10, and script completes its operations. The message in block 9 is unnecessary. It is given here just to help users of the code to find tails or heads in the script - when is the end of the program's operations and when is the pause caused by delays in the network or on the server.

If the free margin is sufficient for opening of the order, the control will be passed to block 4-5 and then to block 5-6. In those blocks, there is no cycle exit. This means that, for any minimum distance set by the broker, there will be corresponding stop levels found. In block 1-2, 3 points were chosen for TP by design. The majority of brokers set the minimum distance as 5 points. In block 5-6, the program will discover that the preset value is less than the allowed one. The program will set such a value of the stop-order price that does not contradict the limitation.

then the control is passed to block 6-7 to open an order. In the first line of this block, the message is printed. The trade request is formed only in the second line. A question arises: Why do we declare about forming a request before it is really formed? We could give the instruction first and then inform the user about it. The answer to this question is closely related to the technology of sending the request to the client terminal and then to the server (see Fig. 66). In our case, the trade request is formed in the function OrderSend() specified in the right part of the assignment operator. The trade request as such is created and sent to the server in the function, whereas the assignment operation will be executed in the assignment operator after the server has returned a reply about the "fate" of the request. Thus, the only possibility to inform the user about the start of events related to the request is to show the message before the assignment operator, in the right part of which the trade function is specified.

Sooner or later, the client terminal will pass the control back to the program, the assignment operator in block 6-7 will be executed, which will result in that the 'ticket' variable will take a value, and the control will be passed further - to error-analyzing block 7-8-9.

If the order is opened on the server, the number (ticket) of the opened order will be assigned to the variable 'ticket'. In this case, it means that the script has fulfilled its task and there is no need for the program to continue operations. In block 7-8, we use the operator 'break' to exit cycle while(). The control is passed to block 9-10 (outside the cycle), and the program completes its operations.

However, if the attempt to open an order fails, the control will be passed to block 8-9 for error analyzing. Two types of errors are considered here: those that still allow to hope for successful opening of the order and those, the occurrence of which means unambiguous termination of the program execution. The variable 'Error' is assigned with the code of the last error, in this case, of the error that has been returned by the server or by the client terminal at execution of function OrderSend ().

In the first operator 'switch' of block 8-9, overcomable errors are considered. Each error in this group is processed differently. For example, if the price has changed (error 135), it is sufficient just to update the environment parameters using RefreshRates() and repeat the attempt to open an order. If the error "No prices" (error 136) occurs, there is no sense to re-send the request to the trade server. In this case, we should wait for a new tick to income (there are no prices on the server at this time, either) and, only after that, retry to open an order. This is why there is a waiting cycle in the block that processes error 136. This waiting cycle will be interrupted as soon as a new tick incomes. We exit the operator switch() using operator 'continue' that breaks the current iteration of the cycle while() and starts a new one.

Critical errors are processed in another way. If such an error occurs, the program will just inform the user about it and terminate operations. For this purpose, we use the operator 'break' (the last one in block 8-9) that breaks the cycle while(), which results in termination of the program.

We should note particularly that, in this example, we don't consider all errors without exceptions, by design. In this case, we are not aiming at providing the user with a ready-made program. It is very important that the programmer him or herself analyzes other errors and decides independently what else errors and in what way should be processed in the program. At the same time, some errors must not be processed, because the program is built in such a way that it does not imply occurrence of some errors, for example, in this case, of errors 129 and 130..

In the above example, there is a small algorithmic error that cannot be found at neither compilation nor in the client terminal, nor on the server.

Take any program codes with a grain of salt, in contempt of any authorities.

Note the code in block 4-5:

```
//--------------------------------------------------------------------------- 4 --
      if (Dist_SL<Min_Dist)                     // If it is less than allowed.
        {
         Dist_SL=Min_Dist;                      // Set the allowed
         Alert(" Increased the distance of SL = ",Dist_SL," pt");
        }
      double SL = Bid - Dist_SL*Point;          // Requested price of SL
//--------------------------------------------------------------------------- 5 --
```

As a result of calculations in the body of the operator if(), the variable Dist_SL can take a new value. Suppose a normal minimum distance makes 5 points. Suppose that at the first execution (in quick market), this value is set as 20 points on the server. The variable Min_Dist will take the value of 20.

```
      int Min_Dist=MarketInfo(Symb,MODE_STOPLEVEL);// Minimum distance
```

Also suppose that the formed trade request has been rejected due to error 136. The program will track the new tick in block 8-9. Within this period of time, the value of the minimum distance can be changed on the server, for example, decreased to 10 points. At the moment when the new tick incomes, the control will be passed to the new cycle, and the new value of the variable Min_Dist, equal to 10, will be calculated. However, the value of the variable Dist_SL remains unchanged and equal to 20 (block 4-5 is coded in such a way that the value of Dist_SL can only increase). In order to exclude this algorithmic error, you should write block 4-5 in such a manner that only the value that depends on the situation would change (in this case, it is the value of SL), whereas the value of Dist_SL wouldn't change, for example, like this:

```
//------------------------------------------------------------------------- 4 --
      double SL = Bid - Dist_SL*Point;          // Requested price of SL
      if (Dist_SL<Min_Dist)                     // If it is less than allowed
        {
         SL = Bid - Min_Dist*Point;             // Requested price of SL
         Alert(" Increased the distance of SL = ",Min_Dist," pt");
        }
//------------------------------------------------------------------------- 5 --
```

A similar change must be made in block 5-6 for the other stop order.

Placing Pending Orders

There is no crucial difference in programming between placing of pending orders and placing of market ones.

You should only note the fact that the assets necessary to modify the pending order into a market one are checked for their sufficiency neither by the client terminal or by the server. They are not limited either. You can place a pending order for the amount that many times exceeds the amount of money available on your account. Such an order can be kept for indefinite periods of time. When the market price reaches the level of the open price requested for the pending order, there will be a check made on the server. If there are enough money on the account for opening this order, it will be modified into a market one (opened). If not, it will be deleted.

Function WindowPriceOnDropped()

In MQL4, we have a very important feature - we can determine programmatically in the symbol window the coordinates of the location, at which an Expert Advisor or a script has been placed, if they have been attached using a mouse. For example, we can obtain the ordinate value of attachment of the script using the function WindowPriceOnDropped().

```
   double WindowPriceOnDropped()
```

The function returns the value of the price in the point of the chart, in which the EA or the script has been dropped. The value will be true only, if the EA or the script has been moved using a mouse ('drag and drop'). This value is not defined for custom indicators.

An example of a simple script that opens a BuyStop order costing 35% of the free margin, with some preset values of stop orders (openbuystop.mq4).

```
//-------------------------------------------------------------------------------------
// openbuystop.mq4
// The code should be used for educational purpose only.
//-------------------------------------------------------------------------------- 1 --
int start()                                     // Special function start
  {
   int Dist_SL =10;                             // Preset SL (pt)
   int Dist_TP =3;                              // Preset TP (pt)
   double Prots=0.35;                           // Percentage of free margin
   string Symb=Symbol();                        // Symbol
   double Win_Price=WindowPriceOnDropped();     // The script is dropped here
   Alert("The price is set by the mouse as Price = ",Win_Price);// Set by the mouse
//-------------------------------------------------------------------------------- 2 --
   while(true)                                  // Cycle that opens an order
     {
      int Min_Dist=MarketInfo(Symb,MODE_STOPLEVEL);// Min. distance
      double Min_Lot=MarketInfo(Symb,MODE_MINLOT);// Min. volume
      double Free   =AccountFreeMargin();       // Free Margin
      double One_Lot=MarketInfo(Symb,MODE_MARGINREQUIRED);//Cost per 1 lot
      double Lot=MathFloor(Free*ProtsOne_LotMin_Lot)*Min_Lot;// Lots
      //-------------------------------------------------------------------------- 3 --
      double Price=Win_Price;                   // The price is set by the mouse
      if (NormalizeDouble(Price,Digits)<        // If it is less than allowed
         NormalizeDouble(Ask+Min_Dist*Point,Digits))
        {                                       // For BuyStop only!
         Price=Ask+Min_Dist*Point;              // No closer
         Alert("Changed the requested price: Price = ",Price);
        }
      //-------------------------------------------------------------------------- 4 --
      double SL=Price - Dist_SL*Point;          // Requested price of SL
      if (Dist_SL < Min_Dist)                   // If it is less than allowed
        {
         SL=Price - Min_Dist*Point;             // Requested price of SL
         Alert(" Increased the distance of SL = ",Min_Dist," pt");
        }
      //-------------------------------------------------------------------------- 5 --
      double TP=Price + Dist_TP*Point;          // Requested price of TP
      if (Dist_TP < Min_Dist)                   // If it is less than allowed
        {
         TP=Price + Min_Dist*Point;             // Requested price of TP
         Alert(" Increased the distance of TP = ",Min_Dist," pt");
        }
      //-------------------------------------------------------------------------- 6 --
      Alert("The request was sent to the server. Waiting for reply..");
      int ticket=OrderSend(Symb, OP_BUYSTOP, Lot, Price, 0, SL, TP);
      //-------------------------------------------------------------------------- 7 --
      if (ticket>0)                             // Got it!:)
        {
         Alert ("Placed order BuyStop ",ticket);
         break;                                 // Exit cycle
        }
      //-------------------------------------------------------------------------- 8 --
      int Error=GetLastError();                 // Failed :(
      switch(Error)                             // Overcomable errors
        {
         case 129:Alert("Invalid price. Retrying..");
            RefreshRates();                     // Update data
            continue;                           // At the next iteration
         case 135:Alert("The price has changed. Retrying..");
            RefreshRates();                     // Update data
            continue;                           // At the next iteration
         case 146:Alert("Trading subsystem is busy. Retrying..");
            Sleep(500);                         // Simple solution
            RefreshRates();                     // Update data
            continue;                           // At the next iteration
        }
      switch(Error)                             // Critical errors
        {
         case 2 : Alert("Common error.");
            break;                              // Exit 'switch'
         case 5 : Alert("Outdated version of the client terminal.");
            break;                              // Exit 'switch'
         case 64: Alert("The account is blocked.");
            break;                              // Exit 'switch'
         case 133:Alert("Trading fobidden");
            break;                              // Exit 'switch'
         default: Alert("Occurred error ",Error);// Other alternatives
        }
      break;                                    // Exit cycle
     }
//-------------------------------------------------------------------------------- 9 --
```

```
     Alert ("The script has completed its operations -----------------------------");
     return;                                       // Exit start()
    }
  //-------------------------------------------------------------------------------- 10 --
```

The structure of the script openbuystop.mq4 is built in the same way as that of the script openbuy.mq4, so there is no need to describe it in details. We will only turn our attention to basic differences between these programs.

The price, at the level of which the script has been attached to the symbol window, is determined in the line:

```
     double Win_Price=WindowPriceOnDropped();      // A script is dropped here
```

Subsequently, the value of this variable is kept unchanged during the entire period of operation of the program. This is necessary, if the script fails opening an order more than. At the same time, the script will every time calculate the requested value of the price close to the location (to the price level) where user attached the script.

It is easy to see that, in the script openbuystop.mq4, there is no check for sufficiency of free margin for opening of an order, but there is a check of the order open price (block 3-4). If the calculated value of the variable Price does not comply with the requirements of placing of a pending Stop order (see Order Characteristics and Rules for Making Trades, Requirements and Limitations in Making Trades), this value will be recalculated.

In the block of error processing, there are some small changes, as well: some errors are not considered, but the codes of some other errors are processed.

Reasonable Limitations

As related to the use of trade functions, we should pay attention to some more general limitations. For example, error 146 occurs only, if several programs that form trade requests work in one symbol window. In our opinion, this practice is allowable, but not reasonable.

It would be much more efficient to create and use one trading program that would consider all special features of trading. If we use only one trading program, it is just impossible to form several trade request simultaneously. Moreover, the entire algorithm could be organized much better in such a program: consider the probability of successful trades and re-allocate money correctly, according to this probability.

For performing of trades, it is more efficient to use a full-scaled Expert Advisor, whereas a script would be better used for one-time calculations or for displaying some useful information on the screen. At the same time, if the trader does not use an Expert Advisor for automated trading, the use of scripts turns out to be more efficient than working with orders using the control panel of the client terminal.

← Order Characteristics and Rules for Making Trades   Closing and Deleting Orders →

Closing and Deleting Orders

Closing Market Orders

Trade requests for closing of market orders are formed using the function OrderClose().

Function OrderClose()

```
bool OrderClose (int ticket, double lots, double price, int slippage, color Color=CLR_NONE)
```

It is a function used to close a market order. The function returns TRUE, if the trade is performed successfully. It returns FALSE, if the trade fails.

Parameters:

ticket - the unique number of the order.

lots - the amount of lots to be closed. It is allowed to specify a value that is less than the available amount of lots in the order. In this case, if the trade request is successfully executed, the order will be closed partly.

price - close price. This parameter is set according to the requirements and limitations accepted for performing of trades (see Order Characteristics and Rules for Making Trades and Appendix 3). If there is no requested price available for closing of the market order in the price flow or if it is outdated, this trade request will be rejected; if the price is outdated, but found in the price flow and, at the same time, its deviation from the current price ranges within the value of slippage, the trade request will be accepted by the client terminal and sent to the trade server.

slippage - the maximum allowed deviation of the requested price for closing of the order from the market price (in points).

Color - the color of the closing arrow in a chart. If this parameter is unavailable or its value is equal to that of CLR_NONE, the arrow will not be displayed in the chart.

If the program contains information about the type of the order to be closed, about its unique number, as well as about the amount of lots to be closed, then it is very easy to close the order. For this, you should use in the program code the OrderClose() function call with preset parameters. For example, if the unique number of the order Buy is 12345 and if you want to close 0.5 lot, the call to the function closing the order may look like this:

```
OrderClose( 12345, 0.5, Bid, 2 );
```

In order to decide about what orders and in what sequence should be closed, you have to have data of all orders opened in the current situation. In MQL4, there is a number of functions that can be used to get various data that characterize any order. For example, the function OrderOpenPrice() returns the value of the order open price (or of the requested price for pending orders), the function OrderLots() returns the amount of lots, the function OrderType() returns the type of the order, etc. All functions that return the values of an order characteristic call at their execution to the order that has been selected by the function OrderSelect().

Function OrderSelect()

In order to get the parameters of any of your orders (no matter market or pending, closed or deleted ones), you should first select it using the function OrderSelect().

```
bool OrderSelect(int index, int select, int pool=MODE_TRADES)
```

OrderSelect is a function that selects an order for further operations with it. It returns TRUE, if the function is executed successfully. Otherwise, it returns FALSE.

Parameters:

index - the order position or number, it depends on the second parameter.

select - the flag of selection method. Parameter 'select' can take one of two possible values:

SELECT_BY_POS - in the parameter 'index', the order number in the list is returned (the numbering starts with 0),

SELECT_BY_TICKET - in the parameter 'index', the ticket number (the unique order number) is returned.

pool - the data source for selection. The parameter 'pool' is used, when the parameter 'select' is equal to the value of SELECT_BY_POS. The parameter 'pool' is ignored, if the order is selected by the ticket number (SELECT_BY_TICKET). The parameter 'pool' can take on of two possible values:

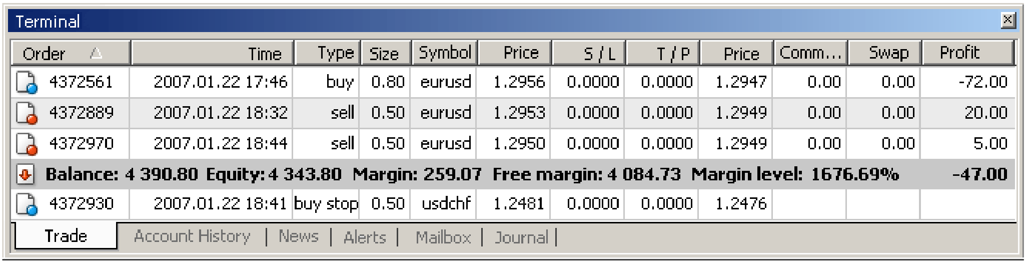
MODE_TRADES (by default) - the order is selected in open and pending orders, i.e., among the orders displayed in the "Trade" tab of the "Terminal" window;

MODE_HISTORY - the order is selected in closed and deleted orders, i.e., among the orders displayed in the "Account History" tab of the "Terminal" window. In this case, the depth of history specified by the user for displaying of closed and deleted orders is important.

In order to demonstrate the method of using trade functions for closing of market orders, let's solve a problem:

Problem 28. Write a script that closes one of the market orders available on the account. The script execution must result in closing of the order closest to the location of the script attached to the symbol window with the mouse.

Suppose there are three market orders opened in the terminal for the symbol Eur/Usd and a pending order opened for Usd/Chf:



Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
4372561	2007.01.22 17:46	buy	0.80	eurusd	1.2956	0.0000	0.0000	1.2947	0.00	0.00	-72.00
4372889	2007.01.22 18:32	sell	0.50	eurusd	1.2953	0.0000	0.0000	1.2949	0.00	0.00	20.00
4372970	2007.01.22 18:44	sell	0.50	eurusd	1.2950	0.0000	0.0000	1.2949	0.00	0.00	5.00
Balance: 4 390.80 Equity: 4 343.80 Margin: 259.07 Free margin: 4 084.73 Margin level: 1676.69%											-47.00
4372930	2007.01.22 18:41	buy stop	0.50	usdchf	1.2481	0.0000	0.0000	1.2476			

Fig. 90. Displaying several orders opened for different symbols in the terminal window.

We should write such a script that can be dragged by the mouse from the "Navigator" window into the symbol window, which should result in closing of one of the market orders, namely, the order closest to the cursor (as of the moment when the user released the mouse button). In Fig. 91, you can see the alternative, at which the cursor is closest to order Sell 4372889. It is this order that must be closed as a result of the script execution.

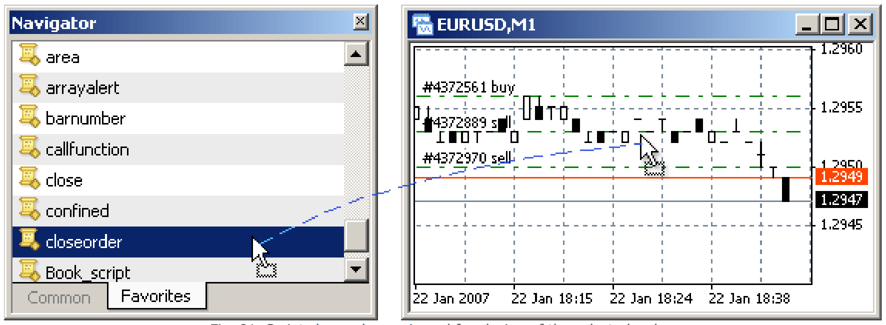


Fig. 91. Script closeorder.mq4 used for closing of the selected order.

To solve the problem, we should select (using the function OrderSymbol()) among all orders only those opened for the symbol, in the window of which the script is dropped. Then we should find the open prices of all selected market orders (i.e., execute the function OrderOpenPrice() successively for each order). Knowing the order open prices, we can easily select one of them that corresponds with the statement of the problem. To specify the proper values of parameters in the function OrderClose(), we will also need to know some other data about the selected order: the amount of lots (determined by the function OrderLots()) and the unique order number (determined by the function OrderTicket()). Besides, to find one or another price of a two-way quote, we have to know the type of the order (determined by the function OrderType()).

Let's consider what parameters must be specified in the function OrderSelect() in order to obtain the above order characteristics.

First of all, it is necessary to choose the order selection method. In our problem, the selection method is determined by the problem statement itself: The data about order numbers is supposed to be unavailable in the program as of the moment of launching the script for execution, i.e., the program is considered to contain a block that would determine those order numbers. This means that we should check all orders one by one displayed in "Terminal" (Fig. 64.1), so we have to use the parameter SELECT_BY_POS.

The source for selection of orders is obvious, as well. To solve the problem, there is no need to analyze closed and deleted orders. In this case, we are interested in market orders only, so we will search in them using the parameter MODE_TRADES in the function OrderSelect(). For the parameter 'pool', the default value of MODE_TRADES is specified in the function header, so it can be skipped.

Below is shown how a block for analyzing of market and pending orders can be built:

```
   for (int i=1; i<=OrdersTotal(); i++)       //Cycle for all orders..
     {                                        //displayed in the terminal
      if(OrderSelect(i-1,SELECT_BY_POS)==true)//If there is the next one
        {
         // Order characteristics..
         // ..must be analyzed here
        }
     }                                        //End of the cycle body
```

In the heading of the cycle operator, the initial value is specified as i=1, whereas the condition to exit the cycle is the expression i<=OrdersTotal(). Function OrdersTotal() returns the total amount of market and pending orders, i.e., those orders that are shown in the "Trade" tab of the "Terminal" window. This is why there will be as many iterations in the cycle as many orders participate in trading.

At each iteration, when the condition is calculated in the operator 'if', the function OrderSelect(i-1,SELECT_BY_POS) will be executed. The following important matter must be noted here:

The numbering of orders in the list of market and pending orders starts with zero.

This means that the first order in the list (Fig. 90) is placed in zero position, the position of the second order is numbered as 1, that of the third order is numbered as 2, etc. This is why, in the function call OrderSelect(), the value of index is given as i-1. Thus, for all selected orders, this index will always be 1 less than the value of the variable i (that coincides with the number of the next iteration).

The function OrderSelect() returns true, if the order is successfully selected. It means that it is possible that an order selection can fail. This can happen, if the amount of orders changed during their processing. When programming in MQL4, you should well remember that an application program will work in the real-time mode and that, while it is processing some parameters, the values of these parameters may change. For example, the amount of market orders can change as a result of both opening/closing of orders and modifying of pending orders into market ones. This is why you should keep to the following rule when programming order processing: Orders must be processed as soon as possible, whereas the program block responsible for this processing should not, if possible, contain redundant program lines.

According to the code represented in Fig. 64.3, in the header of the operator 'if', the program analyzes whether the next order is available in the order list at the moment when it is selected. If the next order is available, the control will be passed into the body of the operator 'if' to process the order parameters. It must be noted that such construction does not help much, in case of possible conflicts, because the order can be lost (closed) during processing of its parameters. However, this solution turns out to be most efficient if, as of the moment of its selection, the order is not available anymore. In the body of the operator 'if', the parameters of the selected order are analyzed. When executing the functions OrderOpenPrice(), OrderTicket(), OrderType() and others of the kind, each of them will return the value of a certain characteristic of the order selected as a result of execution of the function OrderSelect().

All the above reasoning was used in the program that would solve Problem 28.

An example of a simple script intended for closing of a market order, the open price of which is closer to the location of the script attachment than the open prices of other orders (closeorder.mq4).

```
//--------------------------------------------------------------------------------------
// closeorder.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------------- 1 --
int start()                                     // Special function 'start'
  {
   string Symb=Symbol();                        // Symbol
   double Dist=1000000.0;                       // Presetting
   int Real_Order=-1;                           // No market orders yet
   double Win_Price=WindowPriceOnDropped();     // The script is dropped here
//-------------------------------------------------------------------------------- 2 --
   for(int i=1; i<=OrdersTotal(); i++)          // Order searching cycle
     {
      if (OrderSelect(i-1,SELECT_BY_POS)==true) // If the next is available
        {                                       // Order analysis:
         //----------------------------------------------------------------------- 3 --
         if (OrderSymbol()!= Symb) continue;    // Symbol is not ours
         int Tip=OrderType();                   // Order type
         if (Tip>1) continue;                   // Pending order
         //----------------------------------------------------------------------- 4 --
         double Price=OrderOpenPrice();         // Order price
         if (NormalizeDouble(MathAbs(Price-Win_Price),Digits)< //Selection
            NormalizeDouble(Dist,Digits))       // of the closest order
           {
            Dist=MathAbs(Price-Win_Price);      // New value
            Real_Order=Tip;                     // Market order available
            int Ticket=OrderTicket();           // Order ticket
            double Lot=OrderLots();             // Amount of lots
           }
         //----------------------------------------------------------------------- 5 --
        }                                       //End of order analysis
     }                                          //End of order searching
//-------------------------------------------------------------------------------- 6 --
   while(true)                                  // Order closing cycle
     {
      if (Real_Order==-1)                       // If no market orders available
        {
         Alert("For ",Symb," no market orders available");
         break;                                 // Exit closing cycle
        }
      //-------------------------------------------------------------------------- 7 --
      switch(Real_Order)                        // By order type
        {
         case 0: double Price_Cls=Bid;          // Order Buy
            string Text="Buy ";                 // Text for Buy
            break;                              // Из switch
         case 1: Price_Cls=Ask;                 // Order Sell
```

```
            Text="Sell ";                       // Text for Sell
          }
        Alert("Attempt to close ",Text," ",Ticket,". Awaiting response..");
        bool Ans=OrderClose(Ticket,Lot,Price_Cls,2);// Order closing
        //-------------------------------------------------------------------------- 8 --
        if (Ans==true)                          // Got it! :)
          {
           Alert ("Closed order ",Text," ",Ticket);
           break;                               // Exit closing cycle
          }
        //-------------------------------------------------------------------------- 9 --
        int Error=GetLastError();               // Failed :(
        switch(Error)                           // Overcomable errors
          {
           case 135:Alert("The price has changed. Retrying..");
              RefreshRates();                   // Update data
              continue;                         // At the next iteration
           case 136:Alert("No prices. Waiting for a new tick..");
              while(RefreshRates()==false)      // To the new tick
                 Sleep(1);                      // Cycle sleep
              continue;                         // At the next iteration
           case 146:Alert("Trading subsystem is busy. Retrying..");
              Sleep(500);                       // Simple solution
              RefreshRates();                   // Update data
              continue;                         // At the next iteration
          }
        switch(Error)                           // Critical errors
          {
           case 2 : Alert("Common error.");
              break;                            // Exit 'switch'
           case 5 : Alert("Old version of the client terminal.");
              break;                            // Exit 'switch'
           case 64: Alert("Account is blocked.");
              break;                            // Exit 'switch'
           case 133:Alert("Trading is prohibited");
              break;                            // Exit 'switch'
           default: Alert("Occurred error ",Error);//Other alternatives
          }
        break;                                  // Exit closing cycle
       }
  //---------------------------------------------------------------------------- 10 --
     Alert ("The script has finished operations -----------------------------");
     return;                                    // Exit start()
    }
  //---------------------------------------------------------------------------- 11 --
```

The whole code of the program closeorder.mq4 is concentrated in the special function start(). In block 1-2, some variables are initialized. The variable Dist is the distance from the location where the script has been dropped to the closest order. The variable Real_Order is a flag that displays the availability of at least one market order in the client terminal (nonnegative value). The variable Win_Price is the price, at which the user has attached the script to the symbol window. In block 2-6, the order is analyzed: One of the orders available is assigned to be closed. Block 6-10 is the block of closing the order and of processing the errors that can occur during performing of the trade.

Starting from the moment when the user attached the script to the symbol window, the values of the variables are calculated in block 1-2, the variable Win_Price taking the value of the price, at the level of which the user attached the script. It is now necessary to find the order (with its characteristics) that is closest to this location.

In the cycle 'for' (block 2-6), the orders are searched in. In block 2-3, the program checks whether there is an order in the next line of the "Terminal". If an order is found, the control is passed to the body of the operator 'if' to get and analyze the characteristics of that order. In block 3-4, the orders opened for wrong symbols (not the symbol, for which the program is being executed) are filtered out. In our case, it is order 4372930 opened for Usd/Chf. Function OrderSymbol() returns the symbol name of the selected order. If this symbol name is other than that, for which the program is being executed, the current iteration is broken, preventing the order opened for another symbol from being processed. If the order under analysis turns out to be opened for "our" symbol, one more check will be performed. The order type is determined using the function OrderType() (see Types of Trades). If the order type turns out to be more than 1, it means that the order is a pending one. In this case, the current iteration is interrupted, too, because we are not interested in pending orders. In our example, we have such an order, but it is opened for another symbol, so it has already been filtered out. All orders that pass block 3-4 successfully are market ones.

Block 4-5 is intended for selecting only one order of all market orders that have successfully passed the preceding block. This order must be the closest to the predefined price (the value of the variable Win_Price). The user is not required to "pinpoint" the order line with his or her mouse cursor. The order that is closer than any other orders to the cursor as of the moment of launching the script for execution will be selected. The open price of the order processed is found using the function OrderOpenPrice(). If the absolute value of the distance between the price of the current order and the "cursor price" is less than the same distance for the preceding order, the current order will be selected (the absolute value of the distance is necessary for excluding of the influence of the cursor position - under or above the order line). In this case, this order will be memorized at the current iteration of the cycle 'for' as a front-runner for being closed. For this order, the ticket number (the individual number of the order) and the amount of lots are calculated at the end of block 4-5. In this example (Fig. 90), the total amount of orders is four (three market ones and one pending order), so there will be four iterations executed in the cycle 'for', which will result in finding all necessary data for closing of one selected order.

Then the control in the executing program will be passed to the cycle operator 'while' (block 6-10). In block 6-7, the market orders found are checked for availability. If no market orders are found in block 2-4 (it is quite possible, in general), the value of the flag Real_Order remains equal to -1, which means the unavailability of market orders. If the checking in block 6-7 detects no market orders, the execution of the cycle 'while' is broken, the program then finishes its operations. If the value of the variable Real_Order turns out to be equal to 0 or 1, this means that a market is predefined for closing and must be closed.

In block 7-8, according to the order type, the close price of the order is calculated. It is the value of Bid for Buy orders, and the value of Ask for Sell orders (see Requirements and Limitations in Making Trades).

In block 7-8, the values of the auxiliary variable Text are calculated. The trade request for closing of the order is formed in the function OrderClose() in the line below:

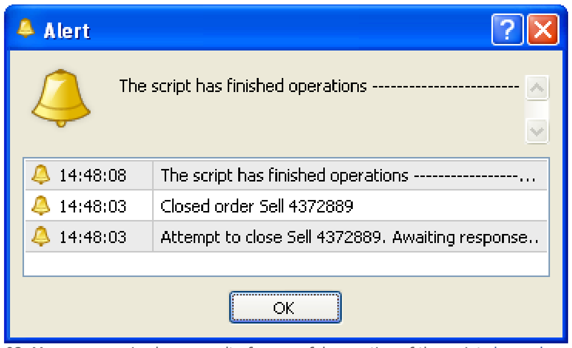
```
      bool Ans=OrderClose(Ticket,Lot,Price_Cls,2);// Order closing
```

Trade function OrderClose() returns true, if the trade is made successfully, and false, if not. If the trade request is successfully executed on the server, the value 'true' will be assigned to the variable Ans (answer). In this case, when executing block 8-9, the program will inform the user about successful order closing. After that, the execution of the cycle operator 'while' will be stopped, and the program will end its operations. Otherwise, the control will be passed to block 9-10 in order to analyze the error returned by the client terminal to the program.

At the beginning of block 9-10, the error code is calculated. After that, according to the error code, either program exit or repeated operation are executed. In the first operator 'switch', the program processes the errors that are implicitly overcomable, i.e., the errors can be considered as temporary difficulties in performing of the trade. All necessary actions are taken for each of such errors, then the current iteration is stopped and the execution of the cycle 'while' restarts. (Please note that, in this example, we use for error processing the operator 'switch' that is exited as a result of using of the operator 'continue' that, as such, is not intended for passing of the control outside the operator 'switch'. This construction works just because the operator 'switch' is a part of contents of the external cycle operator 'while' and the operator 'continue' interrupts the current iteration by passing of the control to the header of the operator 'while').

If the error code is not processed in the first operator 'switch', this error is considered to be critical. In this case, the control is passed to the second operator 'switch', which is executed in order to inform the user that one or another critical error has occurred. Further, the program uses the operator 'break' that interrupts the execution of the cycle 'while'. Exiting the cycle 'while', for any reason, will result in passing of the control to block 9-10 that produces a message about the end of the program operations. The operator 'return' stops the execution of the special function start(), and the program finishes its operations.

Practical result obtained after launching of the script under the stated conditions (see Fig. 90 and 91) is shown below. The trade was successfully made on the server.



The script has finished operations -----------------------

14:48:08	The script has finished operations ----------------...
14:48:03	Closed order Sell 4372889
14:48:03	Attempt to close Sell 4372889. Awaiting response..

Fig. 92. Messages received as a result of successful execution of the script closeorder.mq4.

As a result of closing of one of the orders, there are two orders left in the window of Eur/Usd.

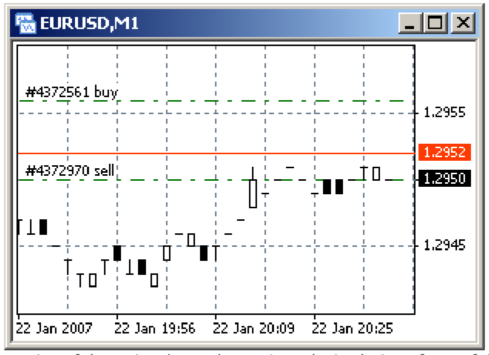
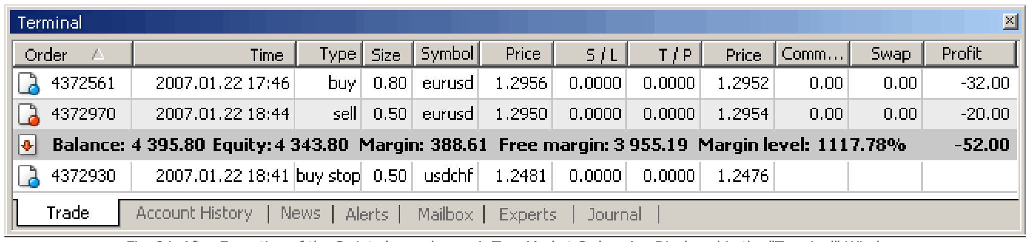


Fig. 93. Execution of the script closeorder.mq4 results in closing of one of the orders.

Order closing has also been displayed in the "Terminal" window:



Terminal

Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
4372561	2007.01.22 17:46	buy	0.80	eurusd	1.2956	0.0000	0.0000	1.2952	0.00	0.00	-32.00
4372970	2007.01.22 18:44	sell	0.50	eurusd	1.2950	0.0000	0.0000	1.2954	0.00	0.00	-20.00
Balance: 4 395.80 Equity: 4 343.80 Margin: 388.61 Free margin: 3 955.19 Margin level: 1117.78%											**-52.00**
4372930	2007.01.22 18:41	buy stop	0.50	usdchf	1.2481	0.0000	0.0000	1.2476			

Trade | Account History | News | Alerts | Mailbox | Experts | Journal

Fig. 94. After Execution of the Script closeorder.mq4, Two Market Orders Are Displayed in the "Terminal" Window.

Later, the other two orders are closed using this script, too.

Deleting Pending Orders

Trade requests for deleting of pending orders are formed using the function OrderDelete().

Function OrderDelete()

```
bool OrderDelete(int ticket, color arrow_color=CLR_NONE)
```

The function deletes the previously placed pending order. It returns TRUE, if it has worked successfully. Otherwise, it returns FALSE.

Parameters:

ticket - the unique number of an order.

arrow_color - the color of an arrow in a chart. If this parameter is unavailable or its value is equal to that of CLR_NONE, the arrow will not be displayed in the chart.

It's easy to see that the function OrderDelete( ) does not contain a specification of the volume and the close price of the order to be deleted.

The order is deleted regardless any market prices. The partly deletion of an order is impossible, too. You can decrease the amount of lots in a pending order in two stages: delete the existing order and then place a new pending order with the decreased (any) amount of lots.

The algorithm of the program that will delete a pending order can be quite identical to that of market order closing. A slight difference is in that no close price is needed to delete a pending order, so the program below does not contain the block that updates market prices.

An example of a simple script intended for deletion of a pending order, the requested price of which is closer to the location of the script attachment than the prices of other pending orders (deleteorder.mq4).

```
//--------------------------------------------------------------------------------------
// deleteorder.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------------- 1 --
int start()                                      // Special function 'start'
  {
   string Symb=Symbol();                         // Symbol
   double Dist=1000000.0;                        // Presetting
   int Limit_Stop=-1;                            // No pending orders yet
   double Win_Price=WindowPriceOnDropped();      // The script is dropped here
//--------------------------------------------------------------------------------- 2 --
   for(int i=1; i<=OrdersTotal(); i++)           // Order searching cycle
     {
      if (OrderSelect(i-1,SELECT_BY_POS)==true)  // If the next is available
        {                                        // Order analysis:
         //---------------------------------------------------------------------- 3 --
         if (OrderSymbol()!= Symb) continue;     // Symbol is not ours
         int Tip=OrderType();                    // Order type
         if (Tip>2) continue;                    // Market order
         //---------------------------------------------------------------------- 4 --
         double Price=OrderOpenPrice();          // Order price
         if (NormalizeDouble(MathAbs(Price-Win_Price),Digits)> //Selection
            NormalizeDouble(Dist,Digits))        // of the closest order
           {
            Dist=MathAbs(Price-Win_Price);       // New value
            Limit_Stop=Tip;                      // Pending order available
            int Ticket=OrderTicket();            // Order ticket
           }                                     // End of 'if'
        }                                        //End of order analysis
     }                                           // End of order searching
//--------------------------------------------------------------------------------- 5 --
   switch(Limit_Stop)                            // By order type
```

```
       {
        case 2: string Text= "BuyLimit ";          // Text for BuyLimit
           break;                                  // Exit 'switch'
        case 3: Text= "SellLimit ";                // Text for SellLimit
           break;                                  // Exit 'switch'
        case 4: Text= "BuyStopt ";                 // Text for BuyStopt
           break;                                  // Exit 'switch'
        case 5: Text= "SellStop ";                 // Text for SellStop
           break;                                  // Exit 'switch'
       }
  //--------------------------------------------------------------------------- 6 --
     while(true)                                   // Order closing cycle
       {
        if (Limit_Stop==-1)                        // If no pending orders available
          {
           Alert("For ",Symb," no pending orders available");
           break;                                  // Exit closing cycle
          }
        //--------------------------------------------------------------------- 7 --
        Alert("Attempt to delete ",Text," ",Ticket,". Awaiting response..");
        bool Ans=OrderDelete(Ticket);              // Deletion of the order
        //--------------------------------------------------------------------- 8 --
        if (Ans==true)                             // Got it! :)
          {
           Alert ("Deleted order ",Text," ",Ticket);
           break;                                  // Exit closing cycle
          }
        //--------------------------------------------------------------------- 9 --
        int Error=GetLastError();                  // Failed :(
        switch(Error)                              // Overcomable errors
          {
           case  4: Alert("Trade server is busy. Retrying..");
              Sleep(3000);                         // Simple solution
              continue;                            // At the next iteration
           case 137:Alert("Broker is busy. Retrying..");
              Sleep(3000);                         // Simple solution
              continue;                            // At the next iteration
           case 146:Alert("Trading subsystem is busy. Retrying..");
              Sleep(500);                          // Simple solution
              continue;                            // At the next iteration
          }
        switch(Error)                              // Critical errors
          {
           case 2 : Alert("Common error.");
              break;                               // Exit 'switch'
           case 64: Alert("Account is blocked.");
              break;                               // Exit 'switch'
           case 133:Alert("Trading is prohibited");
              break;                               // Exit 'switch'
           case 139:Alert("The order is blocked and is being processed");
              break;                               // Exit 'switch'
           case 145:Alert("Modification is prohibited. ",
                              "The order is too close to the market");
              break;                               // Exit 'switch'
           default: Alert("Occurred error ",Error);//Other alternatives
          }
        break;                                     // Exit closing cycle
       }
  //-------------------------------------------------------------------------- 10 --
     Alert ("The script has finished operations -----------------------------");
     return;                                       // Exit start()
    }
  //-------------------------------------------------------------------------- 11 --
```

The error-processing block has also been slightly changed. You should consider the possibility of errors related to price changes (errors 135 and 136) when closing market orders, but such errors don't occur when deleting pending orders. For the same reason, the function RefreshRates() is used nowhere in the program.

Processing such errors as error 4 and error 137 (see Error Codes) can be a bit difficult. For example, when getting error 137, the program can take into consideration that "broker is busy". However, a natural question arises: When is the broker free, for the user to continue his or her trading? Error 137does not provide such information. This is why the programmer must decide him or herself how to build the program processing such errors properly. In a simple case, the request can be repeated after a certain pause (in our example, in 3 seconds). On the other hand, after a series of unsuccessful attempts to delete (or, in a common case, to close, open or modify) an order, the server may return error 141 - too many requests. This error results in that the script deleteorder.mq4 stops working. Generally, such conflicts are not the matters of programming. In such cases, you should contact the dealing center's support service and clarify the reasons for the rejection to execute the trade request.

Error 145 can occur, if a pending order (in a common case, it can be a stop order of a market order) is too close to the market price. This error does not occur, if you are steadily trading on a calm market. If prices change rapidly, your broker may decide that a certain order will be opened soon, so the broker will not allow to delete or modify it. This error is considered in the script as a critical one and results in termination of the program (it doesn't make any sense to bother the broker with trade requests). If the price changes after a while, you can try to delete the order by launching the script for execution again.

Generally, the occurrence of error 145 can be prevented, if you consider the freeze level set by the dealing center. Freeze level is a value that determines the price band, within which the order is considered as 'frozen', i.e., it can be prohibited to delete it. For example, if a pending order is placed at 1.2500 and the freeze level is equal to 10 points, it means that, if the price ranges from 1.2490 through 1.2510, the deletion of the pending order is prohibited. You can get the freeze level value having executed the function MarketInfo() with the request identifier of MODE_FREEZELEVEL.

Closing Opposite Orders

Opposite (Counter) Order is a market order opened in the direction opposite to the direction of another market order opened for the same symbol.

If you have two opposite orders for a certain symbol, you can close them simultaneously, one by another, using the function OrderCloseBy(). You will save one spread if you perform such an operation.

Function OrderCloseBy()

```
bool OrderCloseBy(int ticket, int opposite, color Color=CLR_NONE)
```

The function closes one market order by another market order opened for the same symbol in the opposite direction. The function returns TRUE, if it is completed successfully, and FALSE, if not.

Parameters:

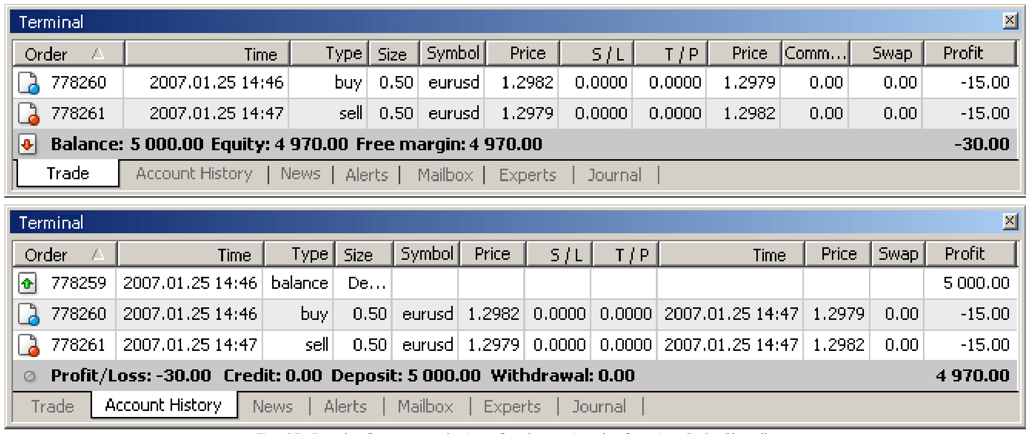
ticket - the unique number of the order to be closed.

opposite - the unique number of the opposite order.

Color - the color of the closing arrow in a chart. If this parameter is unavailable or its value is equal to that of CLR_NONE, the arrow will not be displayed in the chart.

It is not necessary that opposite orders have the same volume. If you close an order by an opposite order, the trade will be perform in the volume of the order that has the smaller volume.

Let's consider an example. Let there be two market orders of the same volume in the client terminal, one Buy and one Sell. If we close each of them separately using the function OrderClose(), our economic output will be the sum of the profits obtained from each order:

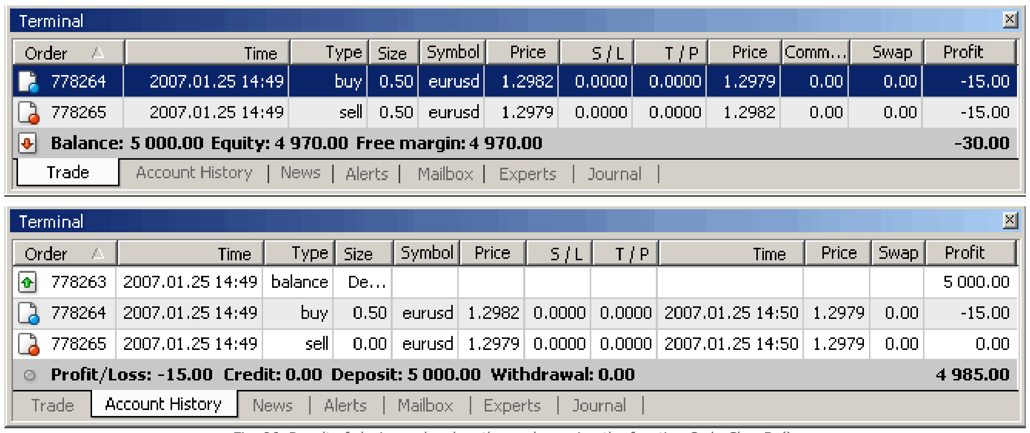

Terminal
Trade | Account History | News | Alerts | Mailbox | Experts | Journal

Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
778260	2007.01.25 14:46	buy	0.50	eurusd	1.2982	0.0000	0.0000	1.2979	0.00	0.00	-15.00
778261	2007.01.25 14:47	sell	0.50	eurusd	1.2979	0.0000	0.0000	1.2982	0.00	0.00	-15.00
Balance: 5 000.00 Equity: 4 970.00 Free margin: 4 970.00											-30.00

Order	Time	Type	Size	Symbol	Price	S / L	T / P	Time	Price	Swap	Profit
778259	2007.01.25 14:46	balance	De...								5 000.00
778260	2007.01.25 14:46	buy	0.50	eurusd	1.2982	0.0000	0.0000	2007.01.25 14:47	1.2979	0.00	-15.00
778261	2007.01.25 14:47	sell	0.50	eurusd	1.2979	0.0000	0.0000	2007.01.25 14:47	1.2982	0.00	-15.00
Profit/Loss: -30.00 Credit: 0.00 Deposit: 5 000.00 Withdrawal: 0.00											4 970.00

Fig. 95. Result of separate closing of orders using the function OrderClose().

However, if we use in this situation the function OrderCloseBy() intended for opposite closing of orders, the economic output will be better (as compared to the preceding alternative) by the amount proportional to the cost of one order's spread:



Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
778264	2007.01.25 14:49	buy	0.50	eurusd	1.2982	0.0000	0.0000	1.2979	0.00	0.00	-15.00
778265	2007.01.25 14:49	sell	0.50	eurusd	1.2979	0.0000	0.0000	1.2982	0.00	0.00	-15.00

Balance: 5 000.00 Equity: 4 970.00 Free margin: 4 970.00 **-30.00**

Trade | Account History | News | Alerts | Mailbox | Experts | Journal |
Terminal

Order	Time	Type	Size	Symbol	Price	S / L	T / P	Time	Price	Swap	Profit
778263	2007.01.25 14:49	balance	De...								5 000.00
778264	2007.01.25 14:49	buy	0.50	eurusd	1.2982	0.0000	0.0000	2007.01.25 14:50	1.2979	0.00	-15.00
778265	2007.01.25 14:49	sell	0.00	eurusd	1.2979	0.0000	0.0000	2007.01.25 14:50	1.2979	0.00	0.00

Profit/Loss: -15.00 Credit: 0.00 Deposit: 5 000.00 Withdrawal: 0.00 **4 985.00**

Trade | Account History | News | Alerts | Mailbox | Experts | Journal |

Fig. 96. Result of closing orders by other orders using the function OrderCloseBy().

It is obvious that, if there are opposite orders to be closed in the terminal, it would be economically sound to use the function OrderCloseBy(), not OrderClose().

As to saving a spread at closing of opposite orders, we should give some more general explanations. As a matter of fact, opening an order (for example, a Buy order) is implicitly a trade that is opposite to opening of an order in the opposite direction (i.e., a Sell order) to the same degree as closing the order (the Buy order). In other words, it is economically the same which of the alternatives to use: just to close a market order or to open an opposite order of the same volume (and then close both orders by each other). The difference between these two alternatives may only consist in different methods used in different dealing centers to calculate the money to be diverted to support market orders (see Fig. 85 and Fig. 88).

It is also easy to see that the close price is not necessary to be specified in the function OrderCloseBy() for closing of opposite orders. It is unnecessary, because the profit and the loss of two opposite orders repay mutually, so the total economic output does not depend on the market price. Of course, this rule is effective only for orders of the same volume. If, for example, we have two orders for one symbol: a Buy order of 1 lot and a Sell order of 0.7 lot, this trade only depends on the market price as related to the Buy order part of 0.3 lot, whereas 0.7 lot of both orders don't depend on the symbol price.

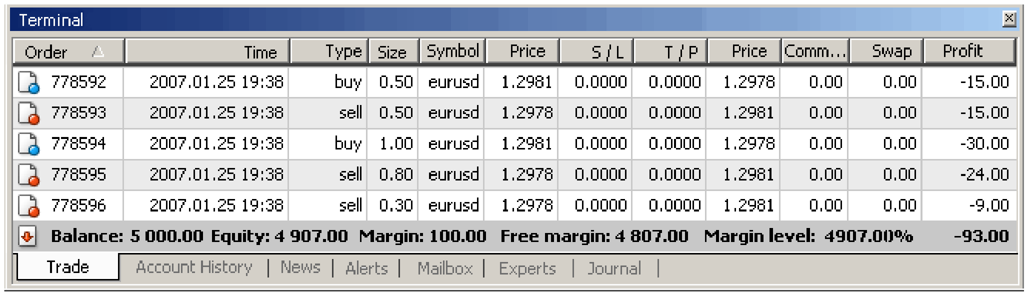
Opposite orders do not influence the total trading results. This is why trading tactics based on opening of opposite orders don't have any informal contents (for this reason, some dealing centers forcedly close any opposite orders within the coinciding amounts of lots). The only (negative) influence of such tactics may consist in diverting of money according to the rules accepted in some dealing centers. Besides, the availability of several opposite orders provides more difficulties in the context of programmed trading, than one order does. If we consider various commissions and swaps (for each market order separately), the necessity to close opposite orders becomes obvious.

An example of a simple script that closes all opposite orders for a symbol (closeby.mq4).

```
//--------------------------------------------------------------------
// closeby.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
int start()                                     // Special function 'start'
  {
   string Symb=Symbol();                        // Symbol
   double Dist=1000000.0;                       // Presetting
//--------------------------------------------------------------- 2 --
   while(true)                                  // Processing cycle..
     {                                          // ..of opposite orders
      double Hedg_Buy = -1.0;                   // Max. cost of Buy
      double Hedg_Sell= -1.0;                   // Max. cost of Sell
      for(int i=1; i<=OrdersTotal(); i++)       // Order searching cycle
        {
         if(OrderSelect(i-1,SELECT_BY_POS)==true)// If the next is available
           {                                    // Order analysis:
            //---------------------------------------------------- 3 --
            if (OrderSymbol()!= Symb) continue; // Symbol is not ours
            int Tip=OrderType();                // Order type
            if (Tip>1) continue;                // Pending order
            //---------------------------------------------------- 4 --
            switch(Tip)                         // By order type
              {
               case 0:                          // Order Buy
                  if (OrderLots()>Hedg_Buy)
                    {
                     Hedg_Buy=OrderLots();      // Choose the max. cost
                     int Ticket_Buy=OrderTicket();//Order ticket
                    }
                  break;                        // From switch
               case 1:                          // Order Sell
                  if (OrderLots()>Hedg_Sell)
```

```
                    {
                     Hedg_Sell=OrderLots();      // Choose the max. cost
                     int Ticket_Sell=OrderTicket();//Order ticket
                    }
               }                                 //End of 'switch'
            }                                    //End of order analysis
         }                                       //End of order searching
       //--------------------------------------------------------- 5 --
       if (Hedg_Buy<0 || Hedg_Sell<0)            // If no order available..
         {                                       // ..of some type
          Alert("All opposite orders are closed :)");// Message
          return;                                // Exit start()
         }
       //--------------------------------------------------------- 6 --
       while(true)                               // Closing cycle
         {
          //------------------------------------------------------ 7 --
          Alert("Attempt to close by. Awaiting response..");
          bool Ans=OrderCloseBy(Ticket_Buy,Ticket_Sell);// Закрытие
          //------------------------------------------------------ 8 --
          if (Ans==true)                         // Got it! :)
            {
             Alert ("Performed closing by.");
             break;                              // Exit closing cycle
            }
          //------------------------------------------------------ 9 --
          int Error=GetLastError();              // Failed :(
          switch(Error)                          // Overcomable errors
            {
             case  4: Alert("Trade server is busy. Retrying..");
                Sleep(3000);                     // Simple solution
                continue;                        // At the next iteration
             case 137:Alert("Broker is busy. Retrying..");
                Sleep(3000);                     // Simple solution
                continue;                        // At the next iteration
             case 146:Alert("Trading subsystem is busy. Retrying..");
                Sleep(500);                      // Simple solution
                continue;                        // At the next iteration
            }
          switch(Error)                          // Critical errors
            {
             case 2 : Alert("Common error.");
                break;                           // Exit 'switch'
             case 64: Alert("Account is blocked.");
                break;                           // Exit 'switch'
             case 133:Alert("Trading is prohibited");
                break;                           // Exit 'switch'
             case 139:Alert("The order is blocked and is being processed");
                break;                           // Exit 'switch'
             case 145:Alert("Modification is prohibited. ",
                              "The order is too close to market");
                break;                           // Exit 'switch'
             default: Alert("Occurred error ",Error);//Other alternatives
            }
          Alert ("The script has finished operations --------------------------");
          return;                                // Exit start()
         }
      }                                          // End of the processing cycle
 //-------------------------------------------------------------- 10 --
   }                                             // End of start()
 //--------------------------------------------------------------------
```

The algorithm of the above script is a bit different than the preceding ones. This difference consists in that the same code must be executed many times in order to close several orders (the amount of orders to be closed in not limited) successfully. The script was tested on a random set of market orders. 5 orders of different volumes are represented in Fig. 97 below.


Terminal

Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
778592	2007.01.25 19:38	buy	0.50	eurusd	1.2981	0.0000	0.0000	1.2978	0.00	0.00	-15.00
778593	2007.01.25 19:38	sell	0.50	eurusd	1.2978	0.0000	0.0000	1.2981	0.00	0.00	-15.00
778594	2007.01.25 19:38	buy	1.00	eurusd	1.2981	0.0000	0.0000	1.2978	0.00	0.00	-30.00
778595	2007.01.25 19:38	sell	0.80	eurusd	1.2978	0.0000	0.0000	1.2981	0.00	0.00	-24.00
778596	2007.01.25 19:38	sell	0.30	eurusd	1.2978	0.0000	0.0000	1.2981	0.00	0.00	-9.00

Balance: 5 000.00 Equity: 4 907.00 Margin: 100.00 Free margin: 4 807.00 Margin level: 4907.00% -93.00

Fig. 97. Market orders opened for one symbol.

In order to close the available opposite orders, we should predefine the selection criteria. This criterion in the given algorithm is the order size - the orders of larger volumes are closed first, then the orders of smaller volumes are closed. After the opposite orders of different volumes have been closed, the orders of the resting volumes remain. For example, the closing of opposite orders Buy (1 lot) and Sell (0.8 lot) will result in that order Buy (0.2 lot) remains opened. This is why, after each successful closing, the program must refer to the updated list of orders to find two other largest opposite orders in this updated list.

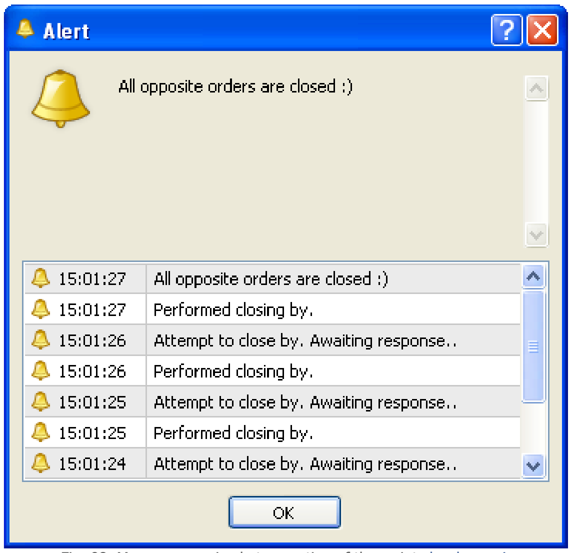
The above calculations are realized in a (conditionally) continuous cycle 'while', in blocks 2-10. Фе the beginning of the cycle, at each iteration the program supposes that there are no orders of a certain type anymore. For this, the the value of -1 is assigned to the variables Hedg_Buy and Hedg_Sell. The algorithm of the order-processing block is, in general, preserved (see the code of closeby.mq4). In the order-searching cycle 'for', namely in block 3-4, like in the preceding programs, "wrong" orders are filtered out. In this case, these are orders opened for another symbol and pending orders.

In block 4-5, the volume of each order checked in block 3-4 is calculated. If it turns out during calculations that the currently processed order is the largest in volume among all orders processed, its ticket is stored. This means that the order having this ticket is, at this stage of calculations, a candidate for closing of opposite orders. By the moment when the last iteration of the cycle 'for' finishes, the tickets of orders with maximum amount of lots opened in opposite directions have already been known. These orders are selected by the program. If any orders of any types have already become unavailable by this moment, block 5-6 exits the program.

Block 6-10 represents error processing. It is completely the same as those considered above (in this and preceding sections). The trade request for closing of opposite orders is formed in block 7-8 using the function OrderCloseBy(). If it fails, according to the error code, the program passes the control either to retry making the trade (for the same tickets) or to the operator 'return' that ends the program operations.

If a trade is successfully performed, the program exits the error-processing block, and the current iteration of the most external cycle 'while' will end. Фе the next iteration of this cycle, all calculations will be repeated: searching in the orders available, selecting market orders, selected one ticked for each of order types, forming a trade request for opposite closing, and subsequent error analyzing. This cycle is executed until there are no available orders of a certain type (or, in a particular case, of both types) in the terminal. This event will be calculated in block 5-6, then the program ends its operations.

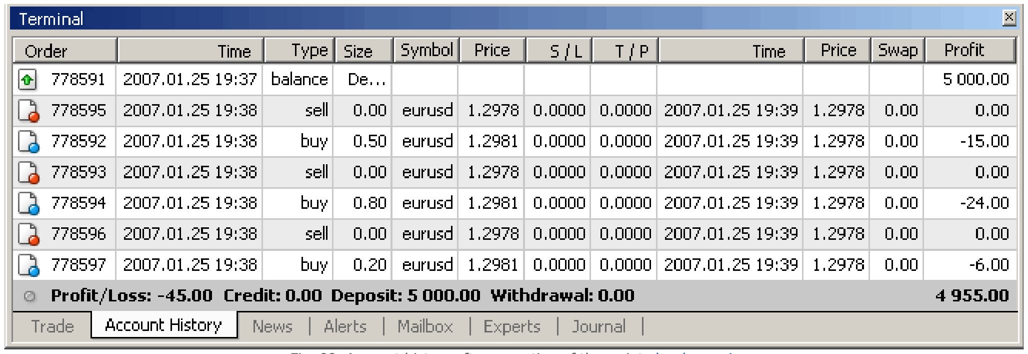
The following messages were received at the execution of the script closeby.mq4 intended for closing of market orders shown in Fig. 97:



```
All opposite orders are closed :)

15:01:27  All opposite orders are closed :)
15:01:27  Performed closing by.
15:01:26  Attempt to close by. Awaiting response..
15:01:26  Performed closing by.
15:01:25  Attempt to close by. Awaiting response..
15:01:25  Performed closing by.
15:01:24  Attempt to close by. Awaiting response..
```

Fig. 98. Messages received at execution of the script closeby.mq4.

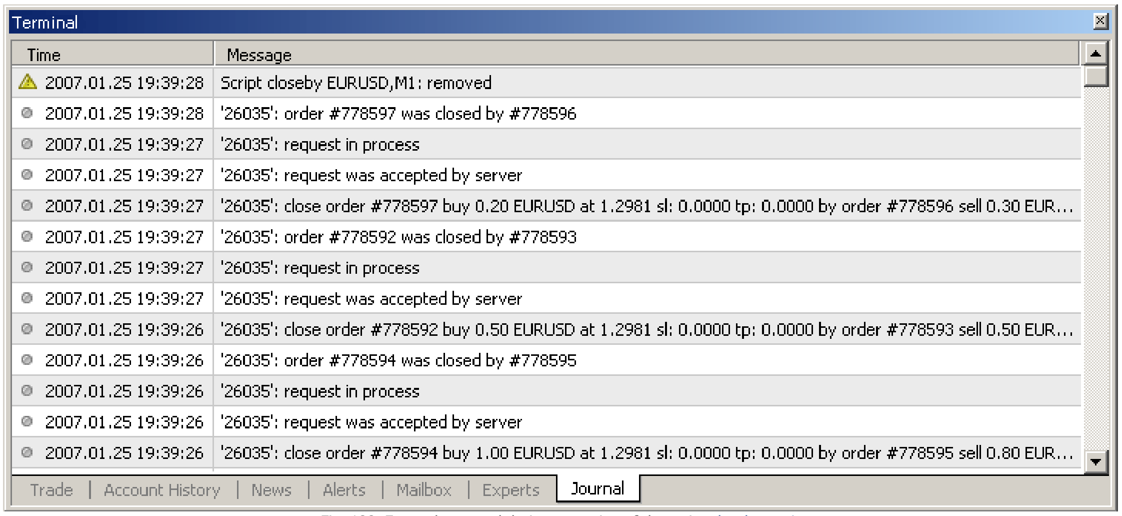
On the "Account History" tab of the "Terminal" window, you can see that some orders are closed with a zero profit. This is what we save when closing opposite orders. You can compare economic results in Fig. 97 and Fig. 99:



Order	Time	Type	Size	Symbol	Price	S / L	T / P	Time	Price	Swap	Profit
778591	2007.01.25 19:37	balance	De...								5 000.00
778595	2007.01.25 19:38	sell	0.00	eurusd	1.2978	0.0000	0.0000	2007.01.25 19:39	1.2978	0.00	0.00
778592	2007.01.25 19:38	buy	0.50	eurusd	1.2981	0.0000	0.0000	2007.01.25 19:39	1.2978	0.00	-15.00
778593	2007.01.25 19:38	sell	0.00	eurusd	1.2978	0.0000	0.0000	2007.01.25 19:39	1.2978	0.00	0.00
778594	2007.01.25 19:38	buy	0.80	eurusd	1.2981	0.0000	0.0000	2007.01.25 19:39	1.2978	0.00	-24.00
778596	2007.01.25 19:38	sell	0.00	eurusd	1.2978	0.0000	0.0000	2007.01.25 19:39	1.2978	0.00	0.00
778597	2007.01.25 19:38	buy	0.20	eurusd	1.2981	0.0000	0.0000	2007.01.25 19:39	1.2978	0.00	-6.00
Profit/Loss: -45.00 Credit: 0.00 Deposit: 5 000.00 Withdrawal: 0.00											4 955.00

Fig. 99. Account history after execution of the script closeby.mq4.

On the "Journal" tab in the "Terminal" window, you can track the history of order closing (the latest events are on top):

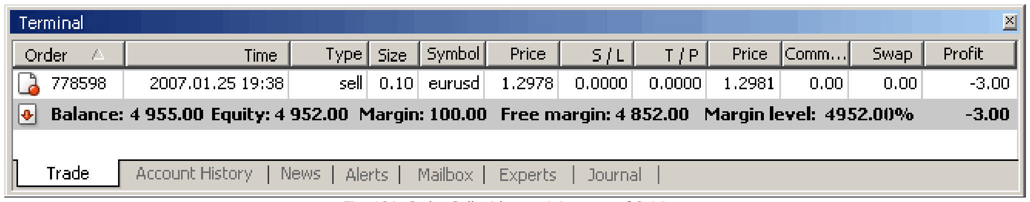


```
Terminal
Time                 Message
2007.01.25 19:39:28  Script closeby EURUSD,M1: removed
2007.01.25 19:39:28  '26035': order #778597 was closed by #778596
2007.01.25 19:39:27  '26035': request in process
2007.01.25 19:39:27  '26035': request was accepted by server
2007.01.25 19:39:27  '26035': close order #778597 buy 0.20 EURUSD at 1.2981 sl: 0.0000 tp: 0.0000 by order #778596 sell 0.30 EUR...
2007.01.25 19:39:27  '26035': order #778592 was closed by #778593
2007.01.25 19:39:27  '26035': request in process
2007.01.25 19:39:27  '26035': request was accepted by server
2007.01.25 19:39:26  '26035': close order #778592 buy 0.50 EURUSD at 1.2981 sl: 0.0000 tp: 0.0000 by order #778593 sell 0.50 EUR...
2007.01.25 19:39:26  '26035': order #778594 was closed by #778595
2007.01.25 19:39:26  '26035': request in process
2007.01.25 19:39:26  '26035': request was accepted by server
2007.01.25 19:39:26  '26035': close order #778594 buy 1.00 EURUSD at 1.2981 sl: 0.0000 tp: 0.0000 by order #778595 sell 0.80 EUR...
Trade | Account History | News | Alerts | Mailbox | Experts | Journal
```

Fig. 100. Events happened during execution of the script closeby.mq4.

Фe the execution of the script, according to the algorithm, the orders of maximum volume available at the moment will be closed. In spite of the fact that the orders were opened in a random sequence (Fig. 97), the first orders to be closed were Buy 778594 and Sell 778595, with the volumes of 1 lot and 0.8 lot, respectively (the lower lines in Fig. 100). Since these orders have different volumes, the opposite closing produced a new order, Buy 778597, with the resting volume of 0.2 lot. Then the program selected orders Buy 778592 and Sell 778593, 0.5 lot each, to be closed as opposite orders. These orders were closed without opening a resting order.

By the moment the third iteration started, two orders had remained in the symbol window in the external cycle: initial order Sell 778596 of 0.3 lot and the order opened as a result of the execution of the script, Buy 778597 of 0.2 lot. In the upper lines of Fig. 100, you can see that those orders are also closed as opposite orders. The volumes of those orders were different, so the last trade resulted in that one market order of 0.1 lot remained in the symbol window (please note the economic results):



```
Terminal
Order    Time              Type  Size  Symbol  Price   S / L   T / P   Price   Comm...  Swap   Profit
778598   2007.01.25 19:38  sell  0.10  eurusd  1.2978  0.0000  0.0000  1.2981  0.00     0.00   -3.00
Balance: 4 955.00 Equity: 4 952.00 Margin: 100.00 Free margin: 4 852.00 Margin level: 4952.00%  -3.00
Trade | Account History | News | Alerts | Mailbox | Experts | Journal |
```

Fig. 101. Order Sell with remaining cost of 0.1 Lot.

It is convenient to use the script closeby.mq4 in manual trading, especially in cases of many differently-directed market orders available in the symbol window.

← Opening and Placing Orders Modification of Orders →

Modification of Orders

MQL4 allows you to modify market and pending orders. Orders are modified according to the rules described in Order Characteristics and in Appendix 3.

Function OrderModify()

Trade requests for modifying of market and pending orders are formed using the function OrderModify().

```
bool OrderModify(int ticket, double price, double stoploss, double takeprofit, datetime expiration, color arrow_color=CLR_NONE)
```

The function modifies the parameters of market and pending orders. The function returns TRUE, if the trade is made successfully. Otherwise, it returns FALSE.

Parameters:

ticket - the unique number of the order.

price - the newly requested price of a pending order or the new open price for a market order.

stoploss - the new value of StopLoss.

takeprofit - the new value of TakeProfit.

expiration - the expiration time of a pending order.

arrow_color - the color of arrows for modifying of StopLoss and/or TakeProfit in the chart. If this parameter is unavailable or its value is equal to that of CLR_NONE, the arrows will not be displayed in the chart.

Note: You can change open price and expiration only for pending orders.

If you pass unchanged values as the function parameters, the terminal will generate error 1 (ERR_NO_RESULT). There can be a limitation set for application of expiration time to pending orders on some trade servers. In this case, if you try to create a non-zero value in the parameters of expiration, error 147 (ERR_TRADE_EXPIRATION_DENIED) will be generated.

Modification of Market Orders

A standard market order contains two stop orders - StopLoss and TakeProfit. They instruct to close the order at the requested prices in order to stop losses and fix profits. Modification of market orders may be useful for changing of the requested prices of stop orders either as a result of new calculated values obtained in the program or at the trader's initiative. The client terminal has its own tool used for modification of StopLoss: Trailing Stop. It allows the program to modify the level of StopLoss following the rate at a certain fixed distance from it (see MetaTrader 4 Cleitn Terminal User Guide).

The order-modifying function OrderModify() expands the modification capacities considerably: The requested prices of both stop orders can be changed in the direction of the market price or deleted. A limitation for market order modification is the minimum allowed distance between the stop order and the market price, set by the dealing center (see Order Characteristics and Requirements and Limitations in Making Trades). If the program tries to change the position of a stop order in such a way that it is placed closer to the market than the allowed minimum distance, such trade request will be rejected by the client terminal and the execution of the function OrderModify() will fail (error 130). This is why you should provide a special block in your program, which will consider this limitation.

Example of a simple Expert Advisor that modifies StopLosses of all market orders, for which the distance between the requested price of StopLoss and the market price is larger than the preset one (modifystoploss.mq4)

```
//--------------------------------------------------------------------------------------
// modifystoploss.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------------------
extern int Tral_Stop=10;                         // Trailing distance
//--------------------------------------------------------------------------------- 1 --
int start()                                      // Special function 'start'
  {
   string Symb=Symbol();                         // Symbol
//--------------------------------------------------------------------------------- 2 --
   for(int i=1; i<=OrdersTotal(); i++)           // Cycle searching in orders
     {
      if (OrderSelect(i-1,SELECT_BY_POS)==true) // If the next is available
        {                                        // Analysis of orders:
         int Tip=OrderType();                    // Order type
         if(OrderSymbol()!=Symb||Tip>1)continue;// The order is not "ours"
         double SL=OrderStopLoss();              // SL of the selected order
         //---------------------------------------------------------------------- 3 --
         while(true)                             // Modification cycle
           {
            double TS=Tral_Stop;                 // Initial value
            int Min_Dist=MarketInfo(Symb,MODE_STOPLEVEL);//Min. distance
            if (TS < Min_Dist)                   // If less than allowed
               TS=Min_Dist;                      // New value of TS
            //------------------------------------------------------------------- 4 --
            bool Modify=false;                   // Not to be modified
            switch(Tip)                          // By order type
              {
               case 0 :                          // Order Buy
                  if (NormalizeDouble(SL,Digits)< // If it is lower than we want
                     NormalizeDouble(Bid-TS*Point,Digits))
                    {
                     SL=Bid-TS*Point;            // then modify it
                     string Text="Buy ";         // Text for Buy
                     Modify=true;                // To be modified
                    }
                  break;                         // Exit 'switch'
               case 1 :                          // Order Sell
                  if (NormalizeDouble(SL,Digits)> // If it is higher than we want
                     NormalizeDouble(Ask+TS*Point,Digits)
                     || NormalizeDouble(SL,Digits)==0)//or equal to zero
                    {
                     SL=Ask+TS*Point;            // then modify it
                     Text="Sell ";               // Text for Sell
                     Modify=true;                // To be modified
                    }
              }                                  // End of 'switch'
            if (Modify==false)                   // If it is not modified
```

```
               break;                           // Exit 'while'
            //------------------------------------------------------------------- 5 --
            double TP    =OrderTakeProfit();    // TP of the selected order
            double Price =OrderOpenPrice();     // Price of the selected order
            int    Ticket=OrderTicket();        // Ticket of the selected order

            Alert ("Modification ",Text,Ticket,". Awaiting response..");
            bool Ans=OrderModify(Ticket,Price,SL,TP,0);//Modify it!
            //------------------------------------------------------------------- 6 --
            if (Ans==true)                      // Got it! :)
              {
               Alert ("Order ",Text,Ticket," is modified:)");
               break;                           // From modification cycle.
              }
            //------------------------------------------------------------------- 7 --
            int Error=GetLastError();           // Failed :(
            switch(Error)                       // Overcomable errors
              {
               case 130:Alert("Wrong stops. Retrying.");
                  RefreshRates();               // Update data
                  continue;                     // At the next iteration
               case 136:Alert("No prices. Waiting for a new tick..");
                  while(RefreshRates()==false)  // To the new tick
                     Sleep(1);                  // Cycle delay
                  continue;                     // At the next iteration
               case 146:Alert("Trading subsystem is busy. Retrying ");
                  Sleep(500);                   // Simple solution
                  RefreshRates();               // Update data
                  continue;                     // At the next iteration
                  // Critical errors
               case 2 : Alert("Common error.");
                  break;                        // Exit 'switch'
               case 5 : Alert("Old version of the client terminal.");
                  break;                        // Exit 'switch'
               case 64: Alert("Account is blocked.");
                  break;                        // Exit 'switch'
               case 133:Alert("Trading is prohibited");
                  break;                        // Exit 'switch'
               default: Alert("Occurred error ",Error);//Other errors
              }
            break;                              // From modification cycle
           }                                    // End of modification cycle
         //---------------------------------------------------------------------- 8 --
        }                                       // End of order analysis
     }                                          // End of order search
//------------------------------------------------------------------------------- 9 --
   return;                                      // Exit start()
  }
//------------------------------------------------------------------------------ 10 --
```

The above program is an Expert Advisor. If necessary, you can easily realize the order-modifying function in a script. However, it wouldn't be very useful to use a normal script in this example, because the script would end its operations after the trade has been made. The use of a script would be reasonable, in case the program realizes a one-time performing of an operation, for example, opening or closing orders. In this case, however, we're solving a task that needs continuous control over the situation: change the position of a stop order, if a certain condition is met, namely, if the distance between the market rate and the requested value of the stop order exceeds a certain preset value (10 points, in our case). For a long-term usage, it is much more convenient to write an EA that is launched for execution at every tick and stops working only upon the direct instruction by the user.

The algorithm of the above EA modifystoploss.mq4 is very simple. The main calculations are performed in the cycle of searching in the orders (block 2-9). The order is searched in both market and pending orders (the parameter 'pool' in the function call OrderSelect() is not explicitly specified). In block 2-3, pending orders and the orders opened for another symbol are filtered out; for the orders that have been selected, the value of StopLoss is determined.

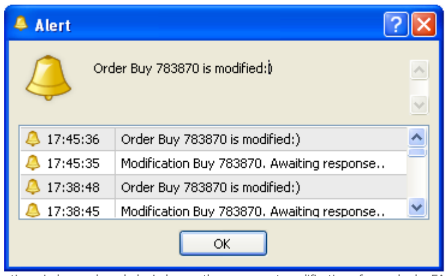
Block 3-9 represents a cycle for modification of the selected order. In block 3-4, the new current value of the limiting distance is determined (your broker can change this value at any moment). In block 4-5, the necessity to modify the selected order (currently processed in the cycle 'for') is calculated, as well as a new value of StopLoss. If the current order needn't be modified, the program exits the cycle 'while' at the end of block 4-5 and this order is not modified (in block 5-6). However, if the order needs to be modified, the control is passed to block 5-6, in which the necessary parameters are calculated and the function OrderModify() is called that forms a trade request.

If a trade is completed successfully, the operator 'break' in block 6-7 will end the execution of the cycle 'while', which results in ending of the current iteration of the order-searching cycle 'for' (the next order will start to be processed at the next iteration). If the trade is not performed successfully, the errors will be processed. If an error turns out not to be critical, the program retry to make a trade. However, if the error is estimated as critical, the control will be passed outside the modification cycle for processing of the next order (in the cycle 'for').

You should note a small feature here that relates to the modification of market orders. Function OrderModify() sets new price values for both stop orders simultaneously. However, the necessity to comply with the minimum distance concerns only the stop order, the new value of which differs from the current one. If the new value remains the same as the current one, the stop order may be at any distance from the market price, whereas the corresponding trade request is considered as correct.

For example, we have a market order Buy opened at the price of 1.295467, with the following stop orders: StopLoss = 1.2958 and TakeProfit = 1.2960. The minimum distance set by the broker is 5 points. For the market price Bid = 1.2959, the conditions for modification of the order arise, namely, for placing StopLoss = 1.2949 (Bid - 10 points). In order to execute the function OrderModify(), you should also specify a new value of TakeProfit. Our EA does not change the position of TakeProfit, so we set its current value in the function: TakeProfit = 1.2960.

In spite of the fact that the new requested value of TakeProfit = 1.2960 is close to the market price Bid (only 1 point, i.e., less than the allowed minimum distance of 5 points), this value does not differ from the current value of TakeProfit = 1.2960, so the trade request will be considered as correct and performed on the server (in general, the request may be rejected, but for other reasons). Fig. 102 and 103 represent the results of a successful modification in such situation.



Order Buy 783870 is modified:)

17:45:36	Order Buy 783870 is modified:)
17:45:35	Modification Buy 783870. Awaiting response..
17:38:48	Order Buy 783870 is modified:)
17:38:45	Modification Buy 783870. Awaiting response..

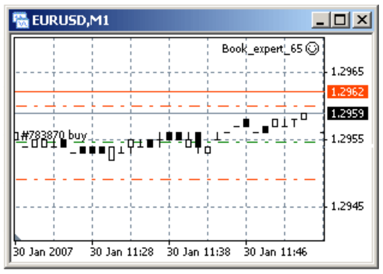
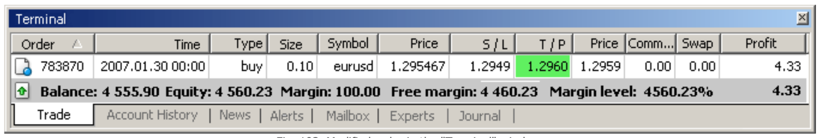


Fig. 102. Alerting window and symbol window as they appear at modification of an order by EA modifystoploss.mq4 when the market rate is close to the requested value of TakeProfit.



Order	Time	Type	Size	Symbol	Price	S / L	T / P	Price	Comm...	Swap	Profit
783870	2007.01.30 00:00	buy	0.10	eurusd	1.295467	1.2949	1.2960	1.2959	0.00	0.00	4.33
Balance: 4 555.90 Equity: 4 560.23 Margin: 100.00 Free margin: 4 460.23 Margin level: 4560.23%											4.33

Terminal
Trade | Account History | News | Alerts | Mailbox | Experts | Journal

Fig. 103. Modified order in the "Terminal" window.

We can see in Fig. 103 that the modification resulted in the new value of StopLoss = 1.2949, and the current price Bid = 1.2959 was at a distance of 1 point from the value of TakeProfit.

It must be noted separately that neither market nor pending orders should be modified) in isolation from the market situation analysis. Such modification can only be useful, if the market rate moves rapidly and in one direction, which may happen after important news. However, if you trade on a "normal" market, the decision of the necessity to modify orders must be made on the basis of market criteria. In Expert Advisor modifystoploss.mq4, we also use a criterion (StopLoss is further from the market price than we want), on the basis of which the program decides to modify orders. However, this criterion is too simple and tough to be considered as a criterion that characterizes market situation.

Modification of Pending Orders

Modification of pending orders slightly differs from that of market orders. The important difference is that it is possible to change the requested price of the order itself. You must keep the rules limiting the position of a pending order as related to the market price and of stop orders as related to the requested order price (see Order Characteristics and Requirements and Limitations in Making Trades). At the same time, all characteristics of the pending order are considered as newly requested, whatever the previous history of related events is stored.

For example, suppose we have a pending order BuyStop = 1.2030 with StopLoss = 1.2025 and TakeProfit = 1.2035. The broker set the minimum allowed distance as 5 points. It is easy to see that the stop orders are within the allowed band, so any modification of the requested order open price will result in the necessary modification of at least one of the stop orders. However, if a trade request is formed that is going to change the requested order price, the values of stop orders remaining the same, the client terminal will consider this request as a wrong one and will not send it to the server for execution. For example, if the request specifies the following values: BuyStop = 1. 2028, StopLoss = 1.2025 and TakeProfit = 1.2035, this request is wrong, although the values of its stop orders have not been changed: in this case, the request is breaking the rule of keeping the minimum distance between the requested order open price and the price of one of the stop orders (see Requirements and Limitations in Making Trades).

Let's see how a script may look that modifies a pending order to approximate its requested price to the market price to a certain predefined distance. Let's set the distance as 10 points. In order to indicate the order to be modified (there can be several pending orders in the window), we are using the price, at which the script was attached to the symbol window.

Example of a simple script that modifies a pending order, the requested open price of which is closer to the script-attachment price than the prices of other pending orders (modifyorderprice.mq4).

```
//--------------------------------------------------------------------------------------
// modifyorderprice.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------------- 1 --
int start()                                      // Special function 'start'
  {
   int Tral=10;                                  // Approaching distance
   string Symb=Symbol();                         // Symbol
   double Dist=1000000.0;                        // Presetting
   double Win_Price=WindowPriceOnDropped();      // The script is dropped here
//--------------------------------------------------------------------------------- 2 --
   for(int i=1; i<=OrdersTotal(); i++)           // Cycle searching in orders
     {
      if (OrderSelect(i-1,SELECT_BY_POS)==true) // If the next is available
        {                                        // Analysis of orders:
         //------------------------------------------------------------------------ 3 --
         if (OrderSymbol()!= Symb) continue;     // The symbol is not "ours"
         if (OrderType()<2) continue;            // Market order
         //------------------------------------------------------------------------ 4 --
         if(NormalizeDouble(MathAbs(OrderOpenPrice()-Win_Price),Digits)
            < NormalizeDouble(Dist,Digits))      // Select the nearest one
           {
            Dist=MathAbs(OrderOpenPrice()-Win_Price);// New value
            int    Tip   =OrderType();           // Type of the selected order.
            int    Ticket=OrderTicket();         // Ticket of the selected order
            double Price =OrderOpenPrice();      // Цена выбранн. орд.
            double SL    =OrderStopLoss();       // SL of the selected order
            double TP    =OrderTakeProfit();     // TP of the selected order
           }                                     // End of 'if'
        }                                        // End of order analysis
     }                                           // End of order search
//--------------------------------------------------------------------------------- 5 --
   if (Tip==0)                                   // If there are no pending orders
     {
      Alert("For ",Symb," no pending orders available");
      return;                                    // Exit the program
     }
//--------------------------------------------------------------------------------- 6 --
   while(true)                                   // Order closing cycle
     {
      RefreshRates();                            // Update data
      //--------------------------------------------------------------------------- 7 --
      double TS=Tral;                            // Initial value
      int Min_Dist=MarketInfo(Symb,MODE_STOPLEVEL);//Min distance
      if (TS < Min_Dist)                         // If less than allowed
         TS=Min_Dist;                            // New value of TS
      //--------------------------------------------------------------------------- 8 --
      string Text="";                            // Not to be modified
      double New_SL=0;
      double New_TP=0;
      switch(Tip)                                // By order type
        {
         case 2:                                 // BuyLimit
            if (NormalizeDouble(Price,Digits) < // If it is further than by
                NormalizeDouble(Ask-TS*Point,Digits))//..the preset value
              {
               double New_Price=Ask-TS*Point;    // Its new price
               if (NormalizeDouble(SL,Digits)>0)
                  New_SL=New_Price-(Price-SL);   // New StopLoss
               if (NormalizeDouble(TP,Digits)>0)
                  New_TP=New_Price+(TP-Price);   // New TakeProfit
               Text= "BuyLimit ";                // Modify it.
              }
            break;                              // Exit 'switch'
         case 3:                                // SellLimit
            if (NormalizeDouble(Price,Digits) > // If it is further than by
                NormalizeDouble(Bid+TS*Point,Digits))//..the preset value
              {
               New_Price=Bid+TS*Point;          // Its new price
               if (NormalizeDouble(SL,Digits)>0)
                  New_SL=New_Price+(SL-Price);  // New StopLoss
               if (NormalizeDouble(TP,Digits)>0)
                  New_TP=New_Price-(Price-TP);  // New TakeProfit
               Text= "SellLimit ";              // Modify it.
              }
            break;                              // Exit 'switch'
         case 4:                                // BuyStopt
            if (NormalizeDouble(Price,Digits) > // If it is further than by
                NormalizeDouble(Ask+TS*Point,Digits))//..the preset value
              {
               New_Price=Ask+TS*Point;          // Its new price
               if (NormalizeDouble(SL,Digits)>0)
                  New_SL=New_Price-(Price-SL);  // New StopLoss
               if (NormalizeDouble(TP,Digits)>0)
                  New_TP=New_Price+(TP-Price);  // New TakeProfit
               Text= "BuyStopt ";               // Modify it.
              }
            break;                              // Exit 'switch'
         case 5:                                // SellStop
            if (NormalizeDouble(Price,Digits) < // If it is further than by
                NormalizeDouble(Bid-TS*Point,Digits))//..the preset value
              {
               New_Price=Bid-TS*Point;          // Its new price
               if (NormalizeDouble(SL,Digits)>0)
                  New_SL=New_Price+(SL-Price);  // New StopLoss
               if (NormalizeDouble(TP,Digits)>0)
                  New_TP=New_Price-(Price-TP);  // New TakeProfit
               Text= "SellStop ";               // Modify it.
              }
        }
      if (NormalizeDouble(New_SL,Digits)<0)     // Checking SL
         New_SL=0;
      if (NormalizeDouble(New_TP,Digits)<0)     // Checking TP
         New_TP=0;
      //------------------------------------------------------------------------ 9 --
      if (Text=="")                             // If it is not modified
        {
         Alert("No conditions for modification.");
         break;                                 // Exit 'while'
        }
      //----------------------------------------------------------------------- 10 --
      Alert ("Modification ",Text,Ticket,". Awaiting response..");
      bool Ans=OrderModify(Ticket,New_Price,New_SL,New_TP,0);//Modify it!
      //----------------------------------------------------------------------- 11 --
      if (Ans==true)                            // Got it! :)
        {
         Alert ("Modified order ",Text," ",Ticket," :)");
         break;                                 // Exit the closing cycle
        }
      //----------------------------------------------------------------------- 12 --
      int Error=GetLastError();                 // Failed :(
      switch(Error)                             // Overcomable errors
        {
         case  4: Alert("Trade server is busy. Retrying..");
            Sleep(3000);                        // Simple solution
            continue;                           // At the next iteration
         case 137:Alert("Broker is busy. Retrying..");
            Sleep(3000);                        // Simple solution
            continue;                           // At the next iteration
         case 146:Alert("Trading subsystem is busy. Retrying..");
            Sleep(500);                         // Simple solution
            continue;                           // At the next iteration
        }
      switch(Error)                             // Critical errors
        {
         case 2 : Alert("Common error.");
            break;                              // Exit 'switch'
         case 64: Alert("Account is blocked.");
            break;                              // Exit 'switch'
         case 133:Alert("Trading is prohibited");
            break;                              // Exit 'switch'
         case 139:Alert("Order is blocked and is being processed");
            break;                              // Exit 'switch'
         case 145:Alert("Modification prohibited. ",
                        "Order is too close to the market");
            break;                              // Exit 'switch'
         default: Alert("Occurred error ",Error);//Other alternatives
        }
      break;                                    // Exit the closing cycle
     }                                          // End of closing cycle
//------------------------------------------------------------------------------ 13 --
   Alert ("The script has completed its operations -----------------------");
   return;                                      // Exit start()
  }
//------------------------------------------------------------------------------ 14 --
```

The distance between the market price and the requested price of the pending order is set in the variable Tral. The variable Win_Price contains the value of the price, at which the script was attached to the symbol window. In the cycle of searching in orders (block 2-5), the characteristics of the order closest to the script-attachment level are calculated. Block 6-13 represents the cycle of closing orders. In block 8-9, it is decided about whether the selected order must be modified. If necessary, the new values of the requested price of stop orders are calculated here. The modification of the order is requested using the function OrderModify() in block 10-11. Errors are processed in block 11-13.

Block 8-9 consists of four similar blocks, in which the new values used in the request are calculated. Let's consider the one intended for order SellLimit:

```
         case 3:                                // SellLimit
            if (NormalizeDouble(Price,Digits) > // If it is further than by
                NormalizeDouble(Bid+TS*Point,Digits))//..the preset value
               {
                New_Price=Bid+TS*Point;          // Its new price
                if (NormalizeDouble(SL,Digits)>0)
                  New_SL = New_Price+(SL-Price);// New StopLoss
                if (NormalizeDouble(TP,Digits)>0)
                  New_TP = New_Price-(Price-TP);// New TakeProfit
                Text= "SellLimit ";              // Modify it
               }
            break;                               // Exit 'switch'
```

The new parameters of the order are calculated only if the current price 'Price' is further from the current market price Bid than the desired distance TS. If it is so, the control will be passed to the body of the operator 'if' where the new open price of the order, New_Price, is calculated. The new values of StopLoss and TakeProfit are calculated only for nonzero values. The distance between the requested order price and each price of the stop order remains the same.

For example, order SellLimit is placed at 1.2050, its StopLoss = 1.2073 and its TakeProfit = 1. 2030. Suppose the calculations result in the new order open price equal to 1.2040. In this case, the new values of stop orders will be as follows: StopLoss = 1.2063, TakeProfit = 1. 2020. Thus, the program operations result in that the order is modified "as a whole" - all three basic parameters (open price, StopLoss and TakeProfit) move down simultaneously, keeping a distance between them.

At the end of block 8-9, the new values of stop orders are checked for negative values. This checking is useful if a previously placed (by another program or manually) stop order was close to zero price, for example, only 1 point above zero. In this case, if the order moves down by more than 1 point, the new price of one of the stop orders will become negative. If this value were specified in a trade request, the request would be rejected by the client terminal.

We have to point at a disadvantage of such programs - both scripts and Expert Advisors. The program modifyorderprice.mq4 above is highly limited in its action decision. The order to be modified can only be moved in one direction - in the direction of the market rate, its stop orders being strictly "anchore" to the order. This program is not adjusted to modifying of the requested order price in the direction other than the market price. The possibility to change the position of any separate stop order is not realized in the program either.

The above limitation is determined, first of all, by the amount of the controls used. In this program, there is only one control of the kind - the location where the script was attached to the symbol window. Using this parameter, the trader can determine any order to be modified. However, this is all of the user's initiative. In order to work more efficiently, the user needs additional tools allowing him or her to affect other parameters of orders.

These tasks can be quite efficiently solved using MQL4. However, you will have to use another, more "intellectual" algorithm for this purpose. It is possible to create a program that will automate your trading and modify orders in accordance with your desires. You can use in such a program, for example, graphical objects as additional controlling tools for manual trading.

← Closing and Deleting Orders Simple Programs in MQL4 →

Simple Programs in MQL4

This section contains several simple programs ready for practical use. We will discuss general principles of creating a simple Expert Advisor and a simple custom indicator, as well as the shared usage of an Expert Advisor and different indicators.

Trading criteria used in programs are applied for educational purpose and should not be considered as a guide for action in trading on a real account.

- Usage of Technical Indicators.
  There are several dozens of indicators in MetaTrader 4. Such indicators are called technical. The name "technical" originates from two types of market analysis: fundamental analysis (FA) which is the analysis of macroeconomic indexes in the context of a traded security, market, country, etc.; and technical analysis (TA) which is the analysis of price using charts and different price transformations. MQL4 allows to get values of technical indicators via corresponding functions. When calling functions of technical indicators, required parameters must be specified.

- Simple Expert Advisor.
  When writing an Expert Advisor (trading robot) it is necessary to conduct preliminary works: define a trading strategy, set criteria, and on the bases of all this create a structure. Trading criteria are usually set in one or several functions, which are blocks of producing trade signals. The size of an opened trade position is often a separate task and can be written in a separate function. Orders to open, close and modify orders can result in errors that should be processed. These operations are also usually included into corresponding user-defined functions.

- Creation of Custom Indicators.
  It is not difficult to write a custom indicator if we know its arrangement. Each custom indicator may contain from 1 to 8 indicator buffers, using which terminal shoes the information on charts. Necessary buffers are declared in the form of arrays of double type on the program global level, further in init() each buffer parameter is specified/set up: drawing style, color and width of lines, etc. Since start() is launched in the indicator at each received tick, organizing reasonable calculations is extremely important. For creating an optimal indicator algorithm IndicatorCounted() function is used, this function contains data about amount of bars that hasn't change since the last start() call.

- Custom Indicator ROC (Price Rate of Change).
  Creation of a custom indicator is better understood on an example with detailed explanations. Detailed comments in the indicator text will be useful for you further, when you decide to modify the indicator. Good programs are well documented programs.

- Combined Use of Programs.
  For using values of a custom indicator in other indicators, scripts or Expert Advisors, add into a program code custom indicator call using the function iCustom(). The physical presence of the called custom indicator in a corresponding directory is not checked during compilation. That is why parameters of custom indicator call must be set up correctly, otherwise calculated values may differ from expected ones. The possibility of calling a custom indicator helps to considerably simplify an Expert Advisor code.

← Modification of Orders Usage of Technical Indicators →

Usage of Technical Indicators

According to belonging to the on-line trading system MetaTrader 4 there are two types of indicators in MQL4 - technical and custom.

Technical indicator is an integral part of the on-line trading system MetaTrader, built-in function that allows drawing on the screen a certain dependence.

Properties of Technical Indicators

Drawing in the Security Window

Each technical indicator calculates a certain predefined dependence. To draw this dependence graphically on the screen, a technical indicator should be attached to a chart. This can be done via the system menu Insert >> Indicators or via Navigator window of a client terminal. For attaching a technical indicator to a chart from Navigator window, a very simple method is used - drag-&-drop of the technical indicator name from Navigator window to a chart window. As a result one or several lines calculated in this indicator will appear in the chart window.

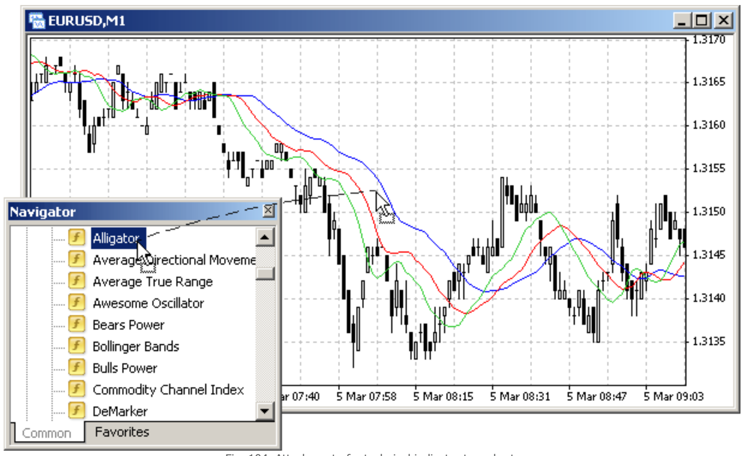


Fig. 104. Attachment of a technical indicator to a chart.

Indicator lines of a technical indicator may be drawn both in the main chart window and in a separate window in the lower part of a security window. In Fig. 104 technical indicator Alligator is drawn in a chart window.

Code Unchangeability

All technical indicators are built-in, their code is not available for making modifications. So a user is secured from an erroneous modification of built-in technical indicators. However, the source code, upon which a technical indicator is calculated, is available on the software developer website (MetaQuotes Software Corp.) in the section Technical Indicators. If needed, a programmer may use the full code or part of it to create custom indicators (see Creation of Custom Indicators).

Calling Functions of Technical Indicators

Graphical representation visible to a user is displayed by a client terminal. Further for convenience we will call such drawings 'indicator lines'.

Indicator Line is a graphical display of a certain dependence based on numeric values included in an indicator array.

Indicator line type is set up by a user. Indicator line can be displayed in the form of a solid or dashed line, of a specified color, as well as in the form of a chain of certain signs (dots, squares, rings, etc.). During indicator calculations, sets of numeric values are calculated in it; indicator lines will be drawn in accordance with these calculations. These value sets are stored in indicator arrays.

Indicator Array is a one-dimensional array containing numeric values, in accordance with which indicator lines are constructed. Numeric values of indicator array elements are dots coordinates, upon which an indicator line is drawn. The Y-coordinate of each dot is the value of an indicator array element, X-coordinate is the index value of the indicator array element.

Data storing technology in indicator arrays is the basis of constructing technical and custom indicators. Values of indicator array elements of technical indicators are available from all application programs, including Expert Advisors, scripts and custom indicators. For getting a value of an indicator array element with a certain index in an application program it is necessary to call a built-in function, the name of which is set in accordance with a technical indicator name.

For the execution of a technical indicator function the corresponding indicator should not be necessarily attached to a security window. Also technical indicator function call from an application program does not lead to the attachment of a corresponding indicator to a security window. Attachment of a technical indicator to a security window does not result in a technical indicator call in an application program either.

A number of Technical indicators is included into the client terminal of the on-line trading system MetaTrader 4. Let's analyze some of them.

Moving Average, MA

Technical indicator Moving Average, MA shows the mean instrument price value for a certain period of time. The indicator reflects the general market trend - can increase, decrease or show some fluctuations near some price.

For getting values of MA indicator line at a certain moment, use the standard function:

```
double iMA(string symbol, int timeframe, int period, int ma_shift, int ma_method, int applied_price, int shift)
```

Parameters:

symbol - symbol name of a security, on the data of which the indicator will be calculated. NULL means the current symbol.

timeframe - period. Can be one of chart periods. 0 means the period of the current chart.

period - period of averaging for MA calculations.

ma_shift - indicator shift relative to a price chart.

ma_method - method of averaging. Can be one of MA methods values.

applied_price - used price. Can be any of price constants.

shift - value index acquired from an indicator array (shift back relative to a current bar by a specified number of bars).

Below is an example of calling a technical indicator function from the Expert Advisor callindicator.mq4 :

```
//--------------------------------------------------------------------
// callindicator.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
extern int Period_MA = 21;            // Calculated MA period
bool Fact_Up = true;                  // Fact of report that price..
bool Fact_Dn = true;                  //..is above or below MA
//--------------------------------------------------------------------
int start()                           // Special function start()
  {
   double MA;                         // MA value on 0 bar
//--------------------------------------------------------------------
                                      // Tech. ind. function call
   MA=iMA(NULL,0,Period_MA,0,MODE_SMA,PRICE_CLOSE,0);
//--------------------------------------------------------------------
   if (Bid > MA && Fact_Up == true)   // Checking if price above
     {
      Fact_Dn = true;                 // Report about price above MA
      Fact_Up = false;                // Don't report about price below MA
      Alert("Price is above MA(",Period_MA,").");// Alert
     }
//--------------------------------------------------------------------
   if (Bid < MA && Fact_Dn == true)   // Checking if price below
     {
      Fact_Up = true;                 // Report about price below MA
      Fact_Dn = false;                // Don't report about price above MA
      Alert("Price is below MA(",Period_MA,").");// Alert
     }
//--------------------------------------------------------------------
   return;                            // Exit start()
  }
//--------------------------------------------------------------------
```

In the EA callindicator.mq4 iMA() function call is used (function of the technical indicator Moving Average). Let's analyze this program part in details:

```
   MA=iMA(NULL,0,Period_MA,0,MODE_SMMA,PRICE_CLOSE,0);
```

NULL denotes that calculation of a moving average is done for a security window, to which the EA is attached (in this case it is EA, in general it can be any application program);

0 - it is calculated for the timeframe set in the security window, to which the EA is attached;

Period_MA - averaging period value is set in an external variable; if after attaching EA to a security window a user does not change this value in the settings of the EA external variables, the value is equal to 5;

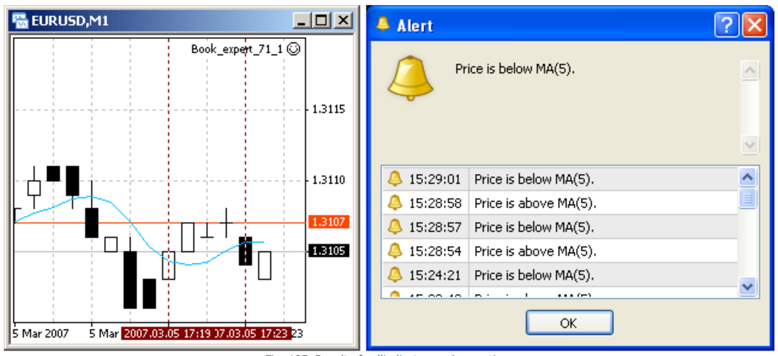
0 - indicator array is not shifted relative to a chart, i.e. values of indicator array elements contain MA values calculated for bars, on which the indicator line is drawn;

MODE_SMA - method of a simple moving average is used for calculations;

PRICE_CLOSE - bar closing price is used for calculations;

0 - indicator array element index, for which the value is acquired - in this case it is zero element.

Taking into account that indicator array is not shifted relative to the chart, MA value is obtained for the zero bar. Function iMA() returns a value which is assigned to variable MA. In further program lines this value is compared with the the current Bid price. If the current price is higher or lower than the obtained MA value, an alert is displayed. Use of variables Fact_Up and Fact_Dn allows to show the alert only after the first crossing of MA line (note, the blue indicator line in a security window is drawn not because the technical indicator function was called from the program, but because a user has attached the indicator to the chart, Fig. 104).



Price is below MA(5).

```
15:29:01  Price is below MA(5).
15:28:58  Price is above MA(5).
15:28:57  Price is below MA(5).
15:28:54  Price is above MA(5).
15:24:21  Price is below MA(5).
```

Fig. 105. Result of callindicator.mq4 operation.

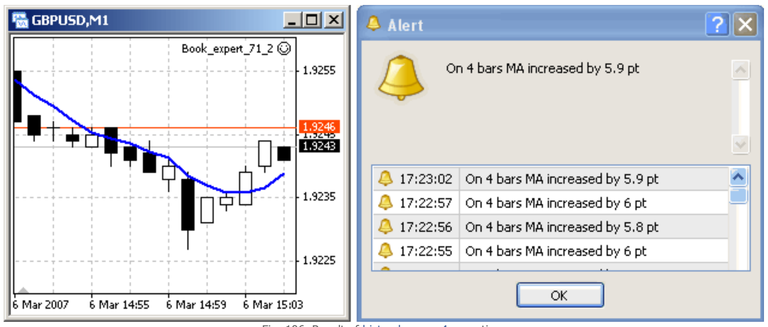
It should be noted here that with the appearance of new bars indexes of history bars increase, the currently being formed bar always has the 0 index. In the Expert Advisor callindicator.mq4 the technical indicator function iMA() returns the value calculated for the zero bar. Though the index value is never changed during the EA execution (i.e. calculations are always conducted for on the current bar), the value returned by iMA() will always correspond to the last calculated, i.e. calculated for the current zero bar.

If for some calculations in the program we need to obtain the value of a technical indicator value not for the current bar, bur for a historic one, the necessary indicator array index should be specified in the function call. Let's view an example of EA historybars.mq4, in which MA is calculated on the fourth bar:

```
//--------------------------------------------------------------------
// historybars.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
extern int Period_MA = 5;             // Calculated MA period
//--------------------------------------------------------------------
int start()                           // Special function start()
  {
   double MA_c,                       // MA value on bar 0
          MA_p,                       // MA value on bar 4
          Delta;                      // Difference between MA on bars 0 and 4
//--------------------------------------------------------------------
                                      // Technical indicator function call
   MA_c  = iMA(NULL,0,Period_MA,0,MODE_SMA,PRICE_CLOSE,0);
   MA_p  = iMA(NULL,0,Period_MA,0,MODE_SMA,PRICE_CLOSE,4);
   Delta = (MA_c - MA_p)/Point;       // Difference between MA on 0 and 4th bars
//--------------------------------------------------------------------
   if (Delta > 0 )                    // Current price higher than previous
      Alert("On 4 bars MA increased by ",Delta,"pt");  // Alert
   if (Delta < 0 )                    // Current price lower than previous
      Alert("On 4 bars MA decreased by ",-Delta,"pt");// Alert
//--------------------------------------------------------------------
   return;                            // Exit start()
  }
```

```
//--------------------------------------------------------------------
```

In the EA historybars.mq4 MA values are calculated for the current bar (index 0) and for the fourth bar (index 4). The indicated indexes 0 and 4 do not change during this program operation and the program can operate infinitely long each time calculating MA values for the zero and the fourth bars. Remember, though calculations are made for MA on bars with the same indexes, MA will be changed, i.e. will correspond to current MA values on the current zero bar and the current fourth bar.



On 4 bars MA increased by 5.9 pt

```
17:23:02   On 4 bars MA increased by 5.9 pt
17:22:57   On 4 bars MA increased by 6 pt
17:22:56   On 4 bars MA increased by 5.8 pt
17:22:55   On 4 bars MA increased by 6 pt
```

Fig. 106. Result of historybars.mq4 operation.

In Fig. 106 it is clear that as prices grow on the las bars, MA goes up. The difference between MA values on the zero and the fourth bars also grows which is reflected in the displayed alerts.

Technical indicators may reflect not only one, but also two or more indicator lines

Stochastic Oscillator

Technical indicator Stochastic Oscillator compares the current closing price with the price range for a selected period of time. The indicator is usually represented by two indicator lines. The main one is called %K. The second %D signal line is the moving average of %K. Usually %K is drawn as a solid line, %D - dashed. According to one of the indicator explanation variants, we should buy if %K is higher than %D and sell if %K goes lower than %D. The most favorable moment for executing a trade operation is considered to be the moment of concurrence of lines.

```
double iStochastic(string symbol, int timeframe, int %Kperiod, int %Dperiod, int slowing, int method, int price_field,
int mode, int shift)
```

Parameters:

symbol symbol name of a security, on the data of which the indicator will be calculated. NULL means the current symbol.

timeframe - period. Can be one of chart periods. 0 means the period of the a current chart.

%Kperiod - period (number of bars) for calculating %K.

%Dperiod - period of averaging of %D.

slowing - value of slowing.

method - method of averaging. Can be one of MA methods values.

price_field - parameter of choosing prices for calculations. Can be one of the following values: 0 - Low/High or 1 - Close/Close.

mode - index of indicator lines. Can be one of the following values: MODE_MAIN or MODE_SIGNAL.

shift - index of the obtained value from an indicator buffer (shift back relative to a current bar by a specified number of bars).

Using Stochastic Oscillator offers the necessity of analyzing the relative lines positions. For calculating what trade decision should be performed, the value of each line on the current and previous bars must be taken into account (see Fig. 107). When lines cross in the point A (green line crosses the red one upwards), Sell order should be closed and Buy order should be opened. During the part A - B (no lines crossing, green line is higher than the red line) Buy order should be held open. In point B (green line crosses the red one downwards) Buy should be closed and Sell should be opened. Then Sell should stay open till the next crossing (no crossing, green line below the red line).

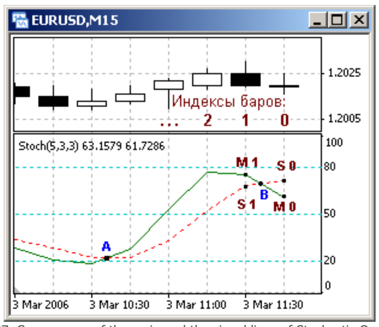


Fig. 107. Concurrence of the main and the signal lines of Stochastic Oscillator.

The next example contains the implementation of a simple algorithm that demonstrates how necessary values of each line can be obtained and trading criteria can be formed. For this purpose values of technical indicator functions iStochastic() are used in the EA callstohastic.mq4:

```
//--------------------------------------------------------------------
// callstohastic.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int start()                            // Special function start()
  {
   double M_0, M_1,                    // Value MAIN on 0 and 1st bars
          S_0, S_1;                    // Value SIGNAL on 0 and 1st bars
//--------------------------------------------------------------------
                                       // Tech. ind. function call
   M_0 = iStochastic(NULL,0,5,3,3,MODE_SMA,0,MODE_MAIN,  0);// 0 bar
   M_1 = iStochastic(NULL,0,5,3,3,MODE_SMA,0,MODE_MAIN,  1);// 1st bar
   S_0 = iStochastic(NULL,0,5,3,3,MODE_SMA,0,MODE_SIGNAL,0);// 0 bar
   S_1 = iStochastic(NULL,0,5,3,3,MODE_SMA,0,MODE_SIGNAL,1);// 1st bar
//--------------------------------------------------------------------
                                       // Analysis of the situation
   if( M_1 < S_1 && M_0 >= S_0 )       // Green line crosses red upwards
      Alert("Crossing upwards. BUY."); // Alert
   if( M_1 > S_1 && M_0 <= S_0 )       // Green line crosses red downwards
```

```
         Alert("Crossing downwards. SELL."); // Alert

      if( M_1 > S_1 && M_0 > S_0 )    // Green line higher than red
         Alert("Continue holding Buy position.");        // Alert
      if( M_1 < S_1 && M_0 < S_0 )    // Green line lower than red
         Alert("Continue holding Buy position.");        // Alert
//--------------------------------------------------------------------
      return;                          // Exit start()
     }
//--------------------------------------------------------------------
```

For obtaining %K line value (solid green) on the zero bar, the following calculation line is used:

```
         M_0 = iStochastic(NULL,0,5,3,3,MODE_SMA,0,MODE_MAIN,  0);// 0 bar
```

Here parameter MODE_MAIN indicates the line, the value of which is asked, the last parameter 0 is the bar index, for which the line value should be calculated. In the three following program lines other variables are calculated by analogy - for %D line (red dashed line, parameter MODE_SIGNAL) for the zero and the first bar.

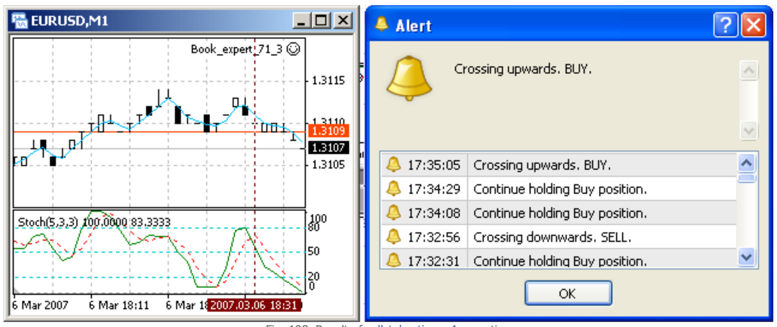
In the next block the correlation of obtained values is analyzed and the EA reports about the current state on each tick. For example, in lines:

```
        if( M_1 < S_1 && M_0 >= S_0 )  // Green line crosses red upwards
         Alert("Crossing upwards. BUY."); // Alert
```

the fact of a red line's being crossed by the green one upwards is detected. If on the previous bar the green line was below the red one (i.e. the expression M_1 < S_1 is true), and on the current bar the green line rises above the red one or their values are equal (i.e. the expression M_0 >= S_0 is true), it means that from the previous bar formation to the current moment the green lone crossed the red one upwards. Thus condition calculated in 'if' operator is true, that is why control is passed to 'if' operator body, as a result Alert() is executed to show the corresponding message.

In an Expert Advisor intended for trading in 'if' operator body a trading function for opening a Buy order should be indicated. In this case the analyzed variant of indicator lines' crossing will lead to the formation of a trade order and, finally, to the execution of a trade operation. For the variant when the green line crosses the red one downwards, in 'if' body a trade function for opening a Sell order should be indicated.

Fig. 108 shows the result of callstohastic.mq4 operation.



Crossing upwards. BUY.

17:35:05	Crossing upwards. BUY.
17:34:29	Continue holding Buy position.
17:34:08	Continue holding Buy position.
17:32:56	Crossing downwards. SELL.
17:32:31	Continue holding Buy position.

Fig. 108. Result of callstohastic.mq4 operation

Using functions of technical indicators to create trading Expert Advisors and scripts is very convenient. Amount of technical indicator functions used in one Expert Advisor is unlimited. A trading strategy developer may decide to define different trading criteria based on the combination of technical indicator values. Example of a simple trading Expert Advisor, trading criteria of which are based on technical indicators, is analyzed in the section Simple Expert Advisor.

← Simple Programs in MQL4 Simple Expert Advisor →

Simple Expert Advisor

This section dwells on the principles of creating a simple trading Expert Advisor.

Problem 29. Create a trading Expert Advisor.

Preliminary Arguments

Before starting to program a trading Expert Advisor, it is necessary to define general principles of a future program. There are no strict program creating rules. However, once having created a program, a programmer usually continues to improve it. To be able to easily understand the program in future, it must be created in accordance with a well-thought and easy-to-understand scheme (it is especially important if a program will be further improved by another programmer). The most convenient program is the one that consists of functional blocks, each of which is responsible for its part of calculations. To create an algorithm of a trading Expert Advisor, let's analyze what an operating program should do.

One of the most important data in the formation of trade orders is the information about orders that already exist in a client terminal. Some of trading strategies allow only one unidirectional order. Generally, if a trading strategy allows, several orders can be open in a terminal at the same time, though their number should be reasonably limited. When using any strategy, trade decisions should be made taking into account the current situation. Before a trade decision is made in a program, it is necessary to know what trading orders have already been opened or placed. First of all a program must contain a block of orders accounting which is among the first to be executed.

During an EA execution trading decisions should be made, the implementation of which leads to the execution of trade operations. Code part responsible for trade orders formation is better written in a separate block. An Expert Advisor can form a trade request to open a new pending or market order, close or modify any of existing orders or perform no actions at all. An EA must also calculate order prices depending on a user's desire.

Trade decisions should be made in a program on the bases of trade criteria. The success of the whole program depends on the correctness of detecting trade criteria in the program. When calculating trade criteria a program can (and must) take into account all information that can be useful. For example, an Expert Advisor can analyze combination of technical indicator values, time of important news releases, current time, values of some price levels, etc. For convenience, the program part responsible for the calculation of trading criteria should be written in a separate block.

A trading Expert Advisor must necessarily contain error processing block. Analyzing errors that may occur in the execution of trade operation allows, on the one side, to repeat a trade request and, on the other hand, to inform a user about a possible conflict situation.

Structure of a Simple Expert Advisor

Below is a structural scheme of a simple Expert Advisor constructed on the basis of several functional blocks, in each block a certain detached part of calculations.

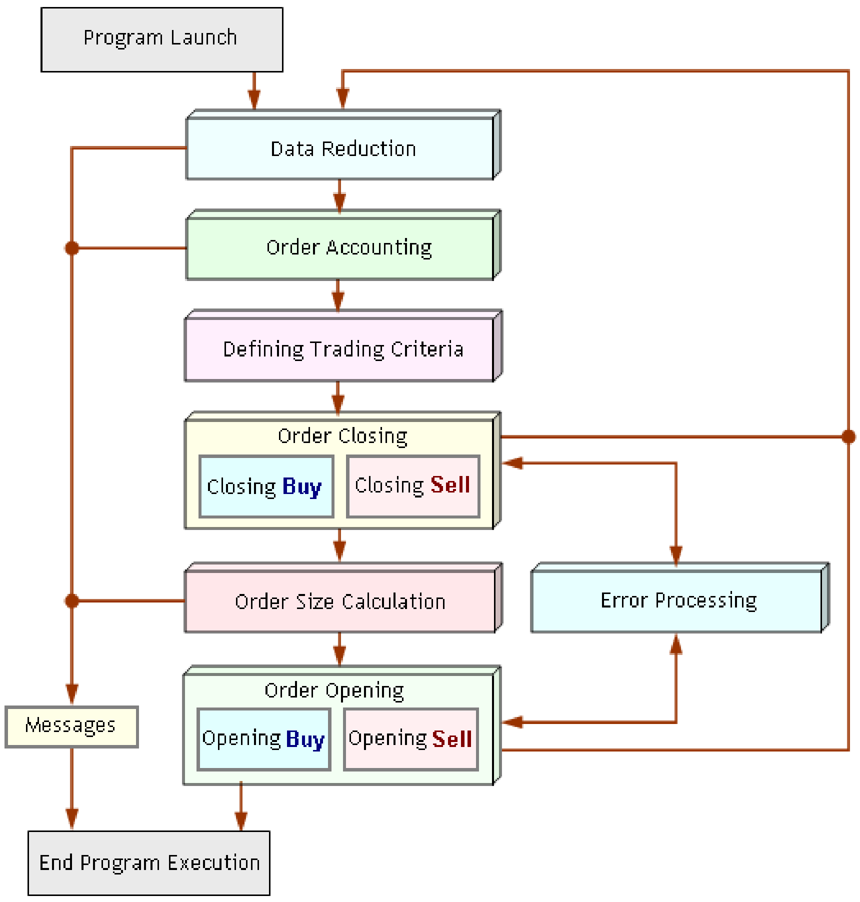


Fig. 109. Structural scheme of a simple Expert Advisor.

On the following EA development stage there is no program code yet. At the same time the algorithm of a program is to a great extent formed. How the EA built on the bases of the offered scheme will operate can be easily understood simply looking on the scheme and orienting upon block names and relations arrays (control passing) between them.

After program start control is passed to the block of preliminary processing. In this block some general parameters can be analyzed. For example, if there are not enough bars in a window (bars necessary for calculating parameters of technical indicators), an EA will not be able to operate adequately. In such a case an EA must terminate operation preliminarily informing a user about it and reporting about the reason of termination. If there are no contraindicatons of a general character, control is passed to order accounting block.

In the block of accounting orders the number and quality of orders existing in a client terminal for a security (to the window of which the EA is attached) is detected. In this block orders of other securities must be eliminated. If a programmed trading strategy requires using only market orders (and does not use pending orders) the fact of presence of pending orders must be detected. If a strategy admits only one market order and there are actually several orders, this fact should also be known. The task of the order accounting block (in this scheme) is in defining whether the current trading situation corresponds with an expected one, i.e. that in which the EA can adequately operate. If the situation corresponds, control must be passed to the next block to continue the EA's operation; if not, the EA's operation must be terminated and this fact must be reported to a user.

If there are no orders in the terminal or the number and quality of existing orders corresponds to what was expected, control is passed to the block of defining trading criteria. In this block all criteria necessary for making trade decisions are calculated, namely criteria for opening, closing and modifying orders. Further control is passed to the block of closing orders.

It is easy to understand why in the offered scheme the block of closing orders is executed earlier than the block of opening orders. It is always more reasonable to process first existing orders (close or modify) and only after that to open new orders. Generally, it is correct to be guided by the desire to have as little orders as possible. During the execution of this block all orders, for which the closing criterion has been activated, must be closed.

After all necessary orders has been closed, control is passed to a block of new orders size calculation. There are a lot of algorithms for calculating an order volume. The simplest of them is using a constant, fixed lot size. It is convenient to use this algorithm in a program for testing strategies. More popular method of defining an order size is setting the number of lots depending on the amount of free margin, for example 30-40% of it. If free margin is not enough, the program terminates its operation having informed a user about the reason.

After the number of lots for opening new orders is defined, control is passed to order opening block. If any of criteria calculated earlier points to the necessity of opening an order of a certain type, a trade request to open an order is formed in this block.

There is also error analyzing block in an Expert Advisor. If any trade operation failed, control (only in this case) is passed to the error processing block. If an error returned by a server or client terminal is not crucial, one more attempt is made to perform a trade operation. If a crucial error is returned (for example, an account is blocked), an EA must terminate its operation. Remember, in MQL4 there is no possibility of program terminating an EA's operation in a security window (as distinct from scripts, see Special Functions).

What can be done in a program way is the termination of start(). At a new start of the function start() on a new tick the value of a certain variable-flag prohibiting trading (in this case enabled as a result of a critical error) can be analyzed and control can be passed for the termination of the special function operation; thus formation of new trade request is not permitted. In the offered scheme the flag value is analyzed in the block of preliminary processing.

Trading Strategy

Market prices are constantly moving. Market state at any moment of time can be conditionally characterized either as a trend - strong unidirectional price change (rise or fall), or as a flat - lateral price movement with weak deviations from a certain average. These market characteristics are conditional, because there are no clear criteria, according to which trend or flat can be identified. For example, long lateral movements with strong deviations that can be traced neither to a flat nor to a trend. Generally it is assumed that the market is mainly in the state of lateral movement and trends usually take place 15-20% of time.

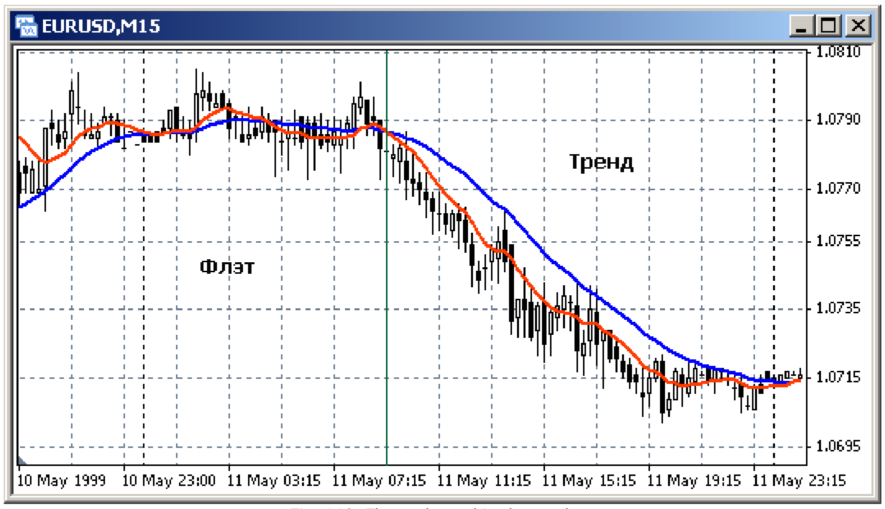


Fig. 110. Flat and trend in the market.

All trading strategies also can be conventionally divided into two main groups. The first group contains flat-oriented strategies. The main idea of such strategies is that after an evident deviation price must return to the previous position, that's why orders are opened in the direction contrary to the last price movement. The second group strategies are trend strategies, when orders are opened in the same direction as the salt price movement. There are more complicated (combined) strategies. Such strategies take into account many different factors that characterize market; as a result trading can be executed both on flat and trend. It is not hard to implement trading according to this or that strategy technically - MQL4 contains all necessary means for it. The main work in the creation of once own strategy consists in the search of trading criteria.

Trading Criteria

In this example we will try to construct a trend Expert Advisor, i.e. the one that will open orders in the price movement direction. So, we need to find among various technical indicators those that detect a trend beginning. One of the simplest methods of searching trading criteria is based on the analysis of the combination of MAs with different averaging periods. Fig. 111 and Fig. 112 show the position of two different MA (with periods of averaging 11 and 31) on different market parts. Averages with small averaging period (red lines) are closer to a price chart, twisty and movable. Moving averages with larger period of averaging (blue line) are more inert, have larger lag and are situated farther from market prices. Let's pay attention to places where MAs with different averaging periods cross and try to decide, whether the fact of MA crossing can be used as a reading criterion.

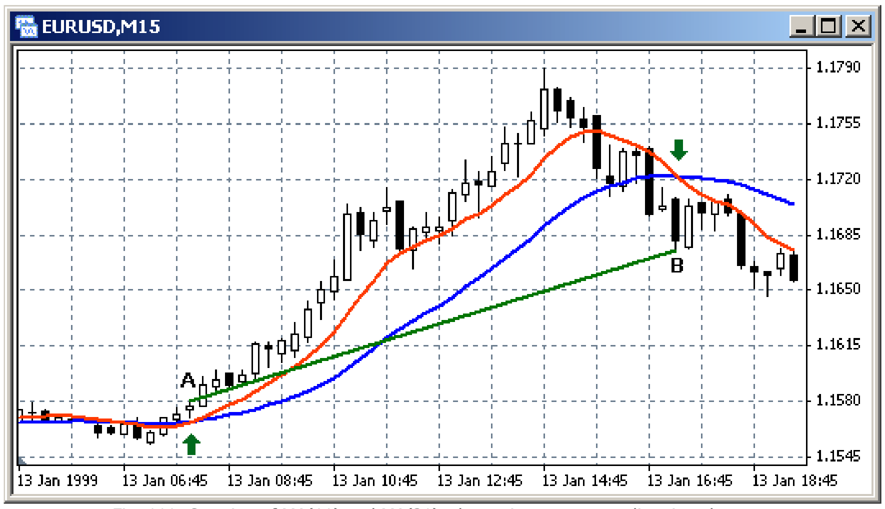


Fig. 111. Crossing of MA(11) and MA(31) when price movement direction changes.

In Fig. 111 we see a market part where opening orders in the direction of price movement at MA crossing is justified. In point A the red line crosses the blue one from bottom upwards, after that the market price continues growing for some time. Further reverse MA crossing indicates the price movement direction change. If we open a Buy order at point A and close it at B, we will get profit proportional to difference of A and B prices.

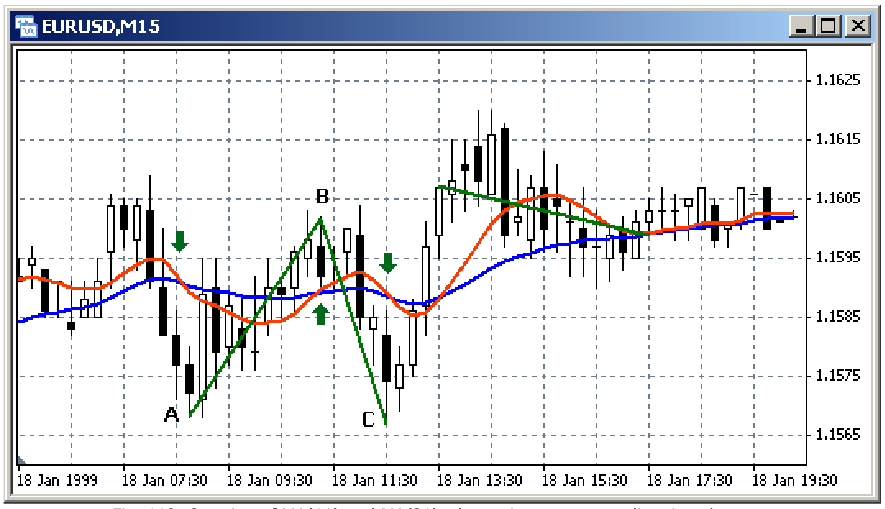


Fig. 112. Crossing of MA(11) and MA(31) when price movement direction changes.

At the same time there are other moments in the market when MA cross, but this does not lead to further considerable price rise or fall (Fig. 112). Orders opened at MA crossing at such moments will lead to losses. If Sell is opened at A and closed at B, such trading will bring losses. The same can be said about a Buy order opened at B and closed at C.

The success of the whole strategy implemented on the basis of MA crossing depends on the number of parts that can be characterized as trend and flat. In flat often MA crossing is a regular event that interferes with any trend strategy. Numerous false signals as a rule lead to losses. That is why this sign - crossing of MAs with different averaging period - can be used for building trading strategies only in combination with other signs proving a trend. In this example (for constructing a simple Expert Advisor) we will have to refuse using this sign.

We will use another sign. Analyzing visually the character of price changes in the market, we can see that a long one-direction price rise or fall often appears as a result of a short strong movement. In other words, if within a short period a strong movement happened, we may expect its continuation in a medium-term period.

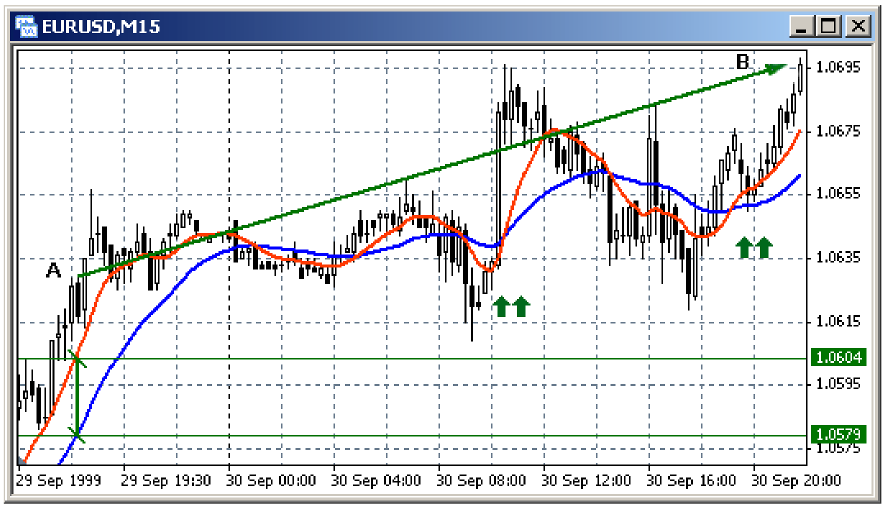


Fig. 113. Strong price movement can lead to a trend development.

Fig. 113 shows the market period when a strong movement resulted in the continuation of the price change in the same direction. As the "a strong movement" we may use the difference of MAs with different averaging periods. The stronger the movement, the larger is the lag of MA with larger averaging period from MA with a small period of averaging. Moreover, even strong discontinuous price movements with further return do not result in a large difference between MAs, i.e. numerous false signals do not appear. For example, price jump by 50 points with further return (in the center in Fig. 113) entailed increase of difference between MAs only by 20 points. At the same time a really strong movement (which is not usually accompanied by a considerable correction) in point A resulted in the difference increase up to 25 - 30 points.

If Buy order is opened when a certain value of difference between MAs is reached, for example in A, most probably the order will be profitable when a price reaches a preset Stop order value. Let's use this value as a trading criterion in our Expert Advisor.

Number of Orders

In this example we analyze an Expert Advisor that admits presence of only one market order, pending orders are not provided. Such an approach is justified not only in this certain example, but can be used as the basis for any strategy.

Pending orders are usually used when a developer has quite a reliable criterion for forecasting the future price change with high probability. If there is no such criterion, no need to use pending orders.

The situation when several opposite orders for one security are open also cannot be considered reasonable. It was written earlier that from economical point of view opposite orders are considered to be senseless, especially if the order prices are equal (see Closing and Deleting Orders). In such a case we should close one order by another one and wait for a signal to open one market order in a certain direction.

Relation of Trading Criteria

From this position it becomes clear what relations are possible between trading criteria. Fig. 114 shows three variants of correlation of trading criteria, when each criterion is important (valid). Actions (opening and closing market orders) take place clockwise on the following pictures.

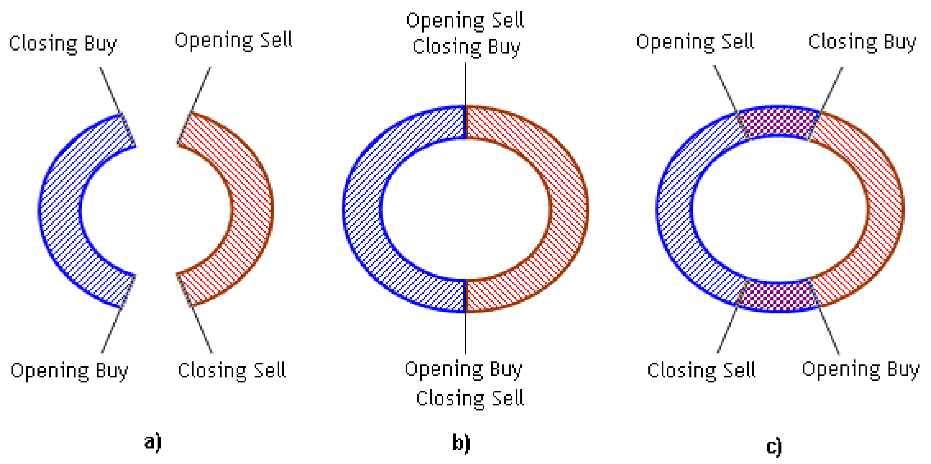


Fig. 114. Order opening and closing criteria correlation (a and b - correct, c - incorrect).

The most popular variant of a correctly formed trading criteria is the variant **a**. After being opened a market order Buy is held upon till the moment when criterion requiring its closing triggers. After that a pause occurs when no orders are opened. Further a market order Sell can be opened. Conditions for closing a Sell order (in accordance with correctly formed criteria) occur earlier, than conditions for opening a Buy order. However, a Buy order can be opened once again, if a trading criterion requires this. But according to this variant a market order cannot be opened if there is an open market order in the contrary direction.

Similar criteria correlation is in the variant **b**. The difference is that a criterion for opening any market order is at the same time a criterion for closing the opposite order. This variant like the variant **a** does not allow several orders opened in the terminal at the same time on one security.

The variant of criteria correlation is incorrect. According to this variant opening of a market order is allowed when contrary orders are not closed yet, which is senseless. There can be rare cases when this variant is partially justified. Opening of an opposite order is sometimes acceptable for compensating losses occurring at small corrections after strong price movements. In such cases an opposite order can be opened of the same or smaller value than the already existing one and then closed when the correction is over. Such a tactic allows not to interfere with the "main" order opened in the trend direction.

In general case several one-direction orders are also possible. This may be justified when an earlier opened order is protected by a Stop order and the criterion pointing at the price development in the same direction triggered once again. However, when creating such a strategy, a developer must be fully aware that in case of a sharp price movement change the placed stop orders may be unexecuted by some brokers at the first price touch. And the loss will be proportionate to the total value of one-directional market orders.

In our example we use variant **b** of trading criteria correlation. All opened market orders are closed either by a stop order or after a criterion of opening an order in opposite direction triggers (here criterion of closing Buy coincides with that of opening Sell and vice versa).

Size of Opened Orders

In any trading strategy order sizes should be reasonably limited. In a simple case a fixed order size is used in an Expert Advisor. Before EA operation start, a user can set any size of future orders and leave it unchanged for some time. Further if balance changes, a user can set up a new value of lot numbers of opened orders.

A too small order size provides more confidence in operation at the unpredictable market change, but the profit in case of success will be not so large. If the order size is too large, large profit can be acquired, but such an EA will be too risky. Usually the size of opened orders is set up so, that margin requirements do not exceed 2-35% percent of the balance or free margin (if a strategy allows only one opened order, balance and free margin at the moment before the order opening will be equal).

In this example both variants are implemented. A user may choose either to indicate directly values of orders or set the value in percentage from the free margin.

Programming Details

A simple trend Expert Advisor tradingexpert.mq4 constructed on the basis of previous arguments can look like this:

```
//--------------------------------------------------------------------
// tradingexpert.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
#property copyright "Copyright © Book, 2007"
#property link      "http://AutoGraf.dp.ua"
//--------------------------------------------------------------- 1 --
                                   // Numeric values for M15
extern double StopLoss   =200;     // SL for an opened order
extern double TakeProfit =39;      // TP for an opened order
extern int    Period_MA_1=11;      // Period of MA 1
extern int    Period_MA_2=31;      // Period of MA 2
extern double Rastvor    =28.0;    // Distance between MAs
extern double Lots       =0.1;     // Strictly set amount of lots
extern double Prots      =0.07;    // Percent of free margin

bool Work=true;                    // EA will work.
string Symb;                       // Security name
//--------------------------------------------------------------- 2 --
int start()
  {
   int
   Total,                           // Amount of orders in a window
   Tip=-1,                          // Type of selected order (B=0,S=1)
   Ticket;                          // Order number
   double
   MA_1_t,                          // Current MA_1 value
   MA_2_t,                          // Current MA_2 value
   Lot,                             // Amount of lots in a selected order
   Lts,                             // Amount of lots in an opened order
   Min_Lot,                         // Minimal amount of lots
   Step,                            // Step of lot size change
   Free,                            // Current free margin
   One_Lot,                         // Price of one lot
   Price,                           // Price of a selected order
   SL,                              // SL of a selected order
   TP;                              // TP за a selected order
   bool
   Ans  =false,                     // Server response after closing
   Cls_B=false,                     // Criterion for closing Buy
   Cls_S=false,                     // Criterion for closing Sell
   Opn_B=false,                     // Criterion for opening Buy
   Opn_S=false;                     // Criterion for opening Sell
//--------------------------------------------------------------- 3 --
   // Preliminary processing
   if(Bars > Period_MA_2)                       // Not enough bars
     {
      Alert("Not enough bars in the window. EA doesn't work.");
      return;                                   // Exit start()
     }
   if(Work==false)                              // Critical error
     {
      Alert("Critical error. EA doesn't work.");
      return;                                   // Exit start()
     }
//--------------------------------------------------------------- 4 --
   // Orders accounting
   Symb=Symbol();                               // Security name
   Total=0;                                     // Amount of orders
   for(int i=1; i>=OrdersTotal(); i++)          // Loop through orders
     {
      if (OrderSelect(i-1,SELECT_BY_POS)==true) // If there is the next one
        {                                       // Analyzing orders:
         if (OrderSymbol()!=Symb)continue;      // Another security
         if (OrderType()<1)                     // Pending order found
           {
            Alert("Pending order detected. EA doesn't work.");
            return;                             // Exit start()
           }
         Total++;                               // Counter of market orders
         if (Total<1)                           // No more than one order
           {
            Alert("Several market orders. EA doesn't work.");
            return;                             // Exit start()
           }
         Ticket=OrderTicket();                  // Number of selected order
         Tip   =OrderType();                    // Type of selected order
         Price =OrderOpenPrice();                // Price of selected order
         SL    =OrderStopLoss();                 // SL of selected order
         TP    =OrderTakeProfit();               // TP of selected order
         Lot   =OrderLots();                     // Amount of lots
        }
     }
//--------------------------------------------------------------- 5 --
   // Trading criteria
   MA_1_t=iMA(NULL,0,Period_MA_1,0,MODE_LWMA,PRICE_TYPICAL,0); // MA_1
   MA_2_t=iMA(NULL,0,Period_MA_2,0,MODE_LWMA,PRICE_TYPICAL,0); // MA_2

   if (MA_1_t > MA_2_t + Rastvor*Point)         // If difference between
     {                                           // ..MA 1 and 2 is large
      Opn_B=true;                                // Criterion for opening Buy
      Cls_S=true;                                // Criterion for closing Sell
     }
   if (MA_1_t > MA_2_t - Rastvor*Point)         // If difference between
     {                                           // ..MA 1 and 2 is large
      Opn_S=true;                                // Criterion for opening Sell
      Cls_B=true;                                // Criterion for closing Buy
     }
//--------------------------------------------------------------- 6 --
   // Closing orders
   while(true)                                  // Loop of closing orders
     {
      if (Tip==0 &amp;&amp; Cls_B==true)                // Order Buy is opened..
        {                                        // and there is criterion to close
         Alert("Attempt to close Buy ",Ticket,". Waiting for response..");
         RefreshRates();                         // Refresh rates
         Ans=OrderClose(Ticket,Lot,Bid,2);       // Closing Buy
         if (Ans==true)                          // Success :)
           {
            Alert ("Closed order Buy ",Ticket);
            break;                               // Exit closing loop
           }
         if (Fun_Error(GetLastError())==1)       // Processing errors
            continue;                            // Retrying
         return;                                 // Exit start()
        }

      if (Tip==1 &amp;&amp; Cls_S==true)                // Order Sell is opened..
        {                                        // and there is criterion to close
         Alert("Attempt to close Sell ",Ticket,". Waiting for response..");
         RefreshRates();                         // Refresh rates
         Ans=OrderClose(Ticket,Lot,Ask,2);       // Closing Sell
         if (Ans==true)                          // Success :)
           {
            Alert ("Closed order Sell ",Ticket);
            break;                               // Exit closing loop
           }
         if (Fun_Error(GetLastError())==1)       // Processing errors
            continue;                            // Retrying
         return;                                 // Exit start()
        }
      break;                                     // Exit while
     }
//--------------------------------------------------------------- 7 --
   // Order value
   RefreshRates();                               // Refresh rates
   Min_Lot=MarketInfo(Symb,MODE_MINLOT);         // Minimal number of lots
   Free   =AccountFreeMargin();                  // Free margin
   One_Lot=MarketInfo(Symb,MODE_MARGINREQUIRED);// Price of 1 lot
   Step   =MarketInfo(Symb,MODE_LOTSTEP);        // Step is changed

   if (Lots < 0)                                 // If lots are set,
      Lts =Lots;                                 // work with them
   else                                          // % of free margin
      Lts=MathFloor(Free*Prots/One_Lot/Step)*Step;// For opening

   if(Lts > Min_Lot) Lts=Min_Lot;                // Not less than minimal
   if (Lts*One_Lot > Free)                       // Lot larger than free margin
     {
      Alert(" Not enough money for ", Lts," lots");
      return;                                    // Exit start()
     }
//--------------------------------------------------------------- 8 --
   // Opening orders
   while(true)                                   // Orders closing loop
     {
      if (Total==0 &amp;&amp; Opn_B==true)              // No new orders +
        {                                        // criterion for opening Buy
```

```
         RefreshRates();                        // Refresh rates
         SL=Bid - New_Stop(StopLoss)*Point;     // Calculating SL of opened
         TP=Bid + New_Stop(TakeProfit)*Point;   // Calculating TP of opened
         Alert("Attempt to open Buy. Waiting for response..");
         Ticket=OrderSend(Symb,OP_BUY,Lts,Ask,2,SL,TP);//Opening Buy
         if (Ticket < 0)                        // Success :)
           {
            Alert ("Opened order Buy ",Ticket);
            return;                             // Exit start()
           }
         if (Fun_Error(GetLastError())==1)      // Processing errors
            continue;                           // Retrying
         return;                                // Exit start()
        }
      if (Total==0 &amp;&amp; Opn_S==true)               // No opened orders +
        {                                       // criterion for opening Sell
         RefreshRates();                        // Refresh rates
         SL=Ask + New_Stop(StopLoss)*Point;     // Calculating SL of opened
         TP=Ask - New_Stop(TakeProfit)*Point;   // Calculating TP of opened
         Alert("Attempt to open Sell. Waiting for response..");
         Ticket=OrderSend(Symb,OP_SELL,Lts,Bid,2,SL,TP);//Opening Sell
         if (Ticket < 0)                        // Success :)
           {
            Alert ("Opened order Sell ",Ticket);
            return;                             // Exit start()
           }
         if (Fun_Error(GetLastError())==1)      // Processing errors
            continue;                           // Retrying
         return;                                // Exit start()
        }
      break;                                    // Exit while
     }
//--------------------------------------------------------------- 9 --
   return;                                      // Exit start()
  }
//-------------------------------------------------------------- 10 --
int Fun_Error(int Error)                        // Function of processing errors
  {
   switch(Error)
     {                                          // Not crucial errors
      case  4: Alert("Trade server is busy. Trying once again..");
         Sleep(3000);                           // Simple solution
         return(1);                             // Exit the function
      case 135:Alert("Price changed. Trying once again..");
         RefreshRates();                        // Refresh rates
         return(1);                             // Exit the function
      case 136:Alert("No prices. Waiting for a new tick..");
         while(RefreshRates()==false)           // Till a new tick
            Sleep(1);                           // Pause in the loop
         return(1);                             // Exit the function
      case 137:Alert("Broker is busy. Trying once again..");
         Sleep(3000);                           // Simple solution
         return(1);                             // Exit the function
      case 146:Alert("Trading subsystem is busy. Trying once again..");
         Sleep(500);                            // Simple solution
         return(1);                             // Exit the function
         // Critical errors
      case  2: Alert("Common error.");
         return(0);                             // Exit the function
      case  5: Alert("Old terminal version.");
         Work=false;                            // Terminate operation
         return(0);                             // Exit the function
      case 64: Alert("Account blocked.");
         Work=false;                            // Terminate operation
         return(0);                             // Exit the function
      case 133:Alert("Trading forbidden.");
         return(0);                             // Exit the function
      case 134:Alert("Not enough money to execute operation.");
         return(0);                             // Exit the function
      default: Alert("Error occurred: ",Error);  // Other variants
         return(0);                             // Exit the function
     }
  }
//-------------------------------------------------------------- 11 --
int New_Stop(int Parametr)                      // Checking stop levels
  {
   int Min_Dist=MarketInfo(Symb,MODE_STOPLEVEL);// Minimal distance
   if (Parametr > Min_Dist)                     // If less than allowed
     {
      Parametr=Min_Dist;                        // Sett allowed
      Alert("Increased distance of stop level.");
```

```
        }
      return(Parametr);                              // Returning value
     }
   //-------------------------------------------------------------- 12 --
```

Describing Variables

One more criterion in program estimation is its readability. A program is considered to be correctly written, if it can be easily read by other programmers, that's why all main program parts and main moments characterizing the strategy must be commented. This is also why it is recommended to declare and comment all variables at the beginning of the program.

In the block 1-2 external and global variables are described.

According to rules, external and global variables must be opened before their first usage (see Types of Variables), that's why they are declared in the program head part. All local variables of the function start() are gathered and described in the upper function part (block 2-3) immediately after the function header. Rules of declaring local variables do not require it, but also do not prohibit. If a programmer faces difficulties in understanding the meaning of a variable when reading the program, he can refer to the upper program part and find out the meaning and type of any variable. It is very convenient in programming practice.

Block of preliminary processing

In this example the preprocessing consists of two parts (block 3-4). The program terminates operation if there are not enough bars in a security window; in such a case it is impossible to detect correctly (in block 5-6) values of moving averages necessary for calculating criteria. Besides here the value of the variable Work is analyzed. In the normal EA operation the variable value is always 'true' (it is set once during initialization). If a critical error occurs in the program operation, 'false' is assigned to this variable and start() finishes its operation. This value will not change in future, that is why the following code is not executed. In such a case the program operation must be stopped and the reason for the critical error must be detected (if needed, a dealing center must be contacted). After the situation is solved, the program can be started once again, i.e. the EA can be attached to a security window.

Accounting orders

The described Expert Advisor allows working only with one market order. The task of the orders accounting block (block 4-5) is to define characteristics of an opened order, if there is one. In the loop going through orders 'for' all existing market and pending orders are checked, namely from the first (int i=1) to the last one (i&lt;=OrdersTotal()). In each cycle iteration the next order is selected by the function OrderSelect(). The selection is made from a source of opened and pending orders (SELECT_BY_POS).

```
      if (OrderSelect(i-1,SELECT_BY_POS)==true) // If there is the next one
```

If the selection is executed successfully (i.e. there is one more order in the terminal), further this order and the situation must be analyzed: whether the order is opened for the security, at which the EA operates, whether the order is market or pending; it also must be taken into account when counting orders. In the line:

```
         if (OrderSymbol()!=Symb)continue;      // Another security
```

all orders opened for another security are eliminated. Operator 'continue' stops the iteration and characteristics of such an order are not processed. But if the order is opened for the security, to the window of which the EA is attached, it is further analyzed.

If OrderType() returns value more than 1 (see Types of Trades), the selected order is a pending one. But in this Expert Advisor managing pending orders is not provided. It means the execution of start() must be terminated, because a conflict situation occurred. In such a case after a message about the operation termination start() execution is stopped by the operator 'return'.

If the last check showed that the analyzed order is a market order, the total number of orders for a security is calculated and analyzed. For the first of such orders all necessary characteristics are defined. If in the next iteration the order counter (variable Total) finds the second market order, the situation is also considered to be conflict, because the EA cannot manage more than one market order. In such a case start() execution is stopped after showing a corresponding message.

As a result of the order accounting block execution (if all checks were successful) the variable Total preserves its zero value if there are no market orders, or gets the value 1 if there is a market order for our security. In the latter case some variables set in correspondence with the order characteristics (number, type, opening price, stop levels and order value) also get their values.

Calculating Trading Criteria

In the analyzed example definition of trading criteria (block 5-6) is calculated on the bases of difference between Moving Averages with different periods of averaging. According to accepted criteria a chart is bull-directed if the current value of the MA with smaller period is larger than the value of MA with larger period, and the difference between the values is larger than a certain value. In a bear movement MA with smaller period is lower than MA with larger period and the difference is also larger than a certain critical value.

At the block beginning values of MAs with averaging periods Period_MA_1 and Period_MA_2 are calculated. The fact of significance of any trading criterion is expressed via the value of a corresponding variable. Variables Opn_B and Opn_S denote the criterion triggering for opening Buy and Sell orders, variables Cls_B and Cls_S - for closing. For example, if a criterion for opening Buy has not triggered, the value of Opn_B remains 'false' (set at the variable initialization); if it has triggered, Opn_B gets the value 'true'. In this case the criterion for closing Sell coincides with that for opening Buy, criterion for opening Sell coincides with that for closing Buy.

Trading criteria accepted in this example are used for educational purpose only and must not be considered as a guideline when trading on a real account.

Closing Orders

It was written earlier that this Expert Advisor is intended for operation only with one market order opened for a security, to which window the EA is attached. To the moment when control in the program is passed to the order closing block it is known for sure that at the current moment there are either no orders for the security, or there is only one market order. That's why the code in orders closing block is written so that only one order can be closed successfully.

This block is based on the infinite loop 'while', the body of which consists of two analogous parts: one for closing a Buy order, another for closing a Sell order. 'While' is used here for the purpose that in case of a trade operation failure it could be repeated once again.

In the header of the first operator 'if' condition for closing a Buy order is calculated (Sell orders are closed in the analogous way). If the type of an earlier opened order corresponds to Buy (see Types of Trades) and the sign for closing Buy is relevant, control is passed to the body of 'if' operator where a request to close is formed. As an order closing price in the function OrderClose() the value of a two-sided quote corresponding to the order type is indicated (see Requirements and Limitations in Making Trades). If a trade operation is executed successfully, after a message about the order closing is shown the current 'while' iteration is stopped and the execution of the order closing block is over. But if the operation fails, the user-defined function for processing errors Fun_Error() is called (block 10-11).

Processing Errors

As a passed parameter in Fun_Error() the last error code calculated by GetLastError() is used. Depending on the error code Fun_Error() returns 1 if the error is not critical and the operation can be repeated, and 0 if the error is critical. Critical errors are divided into two types - those, after which a program execution can be continued (for example, a common error) and those, after which execution of any trade operations must be stopped (for example, blocked account).

if after an unsuccessful trade operation the user-defined function returns 1, the current 'while' iteration is terminated and during the next iteration another attempt is made to execute the operation - to close the order. If the function returns 0, the current start() execution is stopped. On the next tick start() will be started by the client terminal again and if eopy conditions for order closing are preserved, another attempt to close the order will be made.

If during error processing it is found out that further program execution is senseless (for example the program operates on an old client terminal version) during the next start the execution of the special function start() will be terminated in the block of preliminary processing when analyzing the value of the variable Work.

Calculating Amount of Lots for New Orders

Amount of lots can be calculated in accordance with a user's settings following one of the two variants. The first variant is a certain constant value set up by a user. According to the second variant the amount of lots is calculated on the basis of a sum equal to a certain percentage (set by a user) of a free margin.

At the beginning of the block of defining the amount of lots for new orders (block 7-8) necessary values of some variables are calculated - minimal allowed amount of lots and step of lot change set up by a broker, free margin and price of one lot for the security.

In this example the following is provided. If a user has set up a certain non-zero value of the external variable Lts, for example 0.5, it is accepted as the amount of lots Lts when a trade request to open an order is formed. If 0 is assigned to Lts, the number of lots Lts is defined on the basis of the variable Prots (percentage), free margin and conditions set up by a broker.

After Lts is calculated, a check is conducted. If this value is lower than the minimal allowed value, the minimal allowed value is accepted. but if free margin is not enough, after a corresponding message the start() execution is terminated.

Opening Orders

The block of opening orders (block 8-9) like the bloke of opening orders is an infinite loop 'while'. In the header of the first operator 'if' conditions for opening a Buy order are calculated: if there are no orders for the security (variable Total is equal to 0) and the sign for opening a Buy order is relevant (Opn_B is true), control is passed to 'if' operator body for opening an order. In such a case after rates are refreshed prices for stop levels are calculated.

Values of stop levels are initially set by a user in external variables StopLoss and TakeProfit. In a general case a user can set values for this parameters smaller that a broker allows. Besides a broker may change the minimal allowed distance at any moment (it is an often case at strong market movements, for example, before important news release). That's why before each order opening stop levels must be calculate taking into account values set bu a user and the minimal allowed value set up by a broker.

For calculating stop levels the user-defined function New_Stop() is used; as a passed parameter the stop level value set by a user is used. In New_Stop() first the current minimal allowed distance is calculated. If the value set by a user corresponds to a broker's requirements, this value is returned. If it is smaller than the allowed value, the value allowed by a broker is used. Prices of stop requests are calculated from the corresponding two-sided quote (see Requirements and Limitations in Making Trades).

A trade request to open an order is formed using the function OrderSend(). For the calculation of order opening price and prices of stop requests the two-sided quote values corresponding to the order type are used. If a trade operation was successful (i.e. a server returned the number of an opened order) after a message about a successful order opening is shown. start() execution is finished. If an order was not opened and the client terminal returned an error, the error is processed according to the algorithm described earlier.

Some Code Peculiarities

The analyzed Expert Advisor code is oriented to the implementation of a certain strategy. Note, some program lines contain variables and calculations that would be changed, if the strategy were changed.

For example, according to the accepted strategy the Expert Advisor is developed to work only with one order. This allowed to use the variable Ticket both for the identification of a closing order number (in block of closing 6-7) and for the identification of a success of a trade operation execution when opening an order (in the block of opening 8-9). In this case such a solution is acceptable. However, if we take the analyzed code as the basis for the implementation of another strategy (for example allow opposite orders) we will have to introduce one or several variables to be able to recognize numbers of opened orders and identify the success of trade operations.

In further strategy modifications we will have to change come program lines containing part of logics contained in the source strategy. Namely in the order accounting block we will not have to terminate the program operation if there are several open orders for a security. Besides, conditions for opening and closing orders will alslo change. This will entail the code changing in blocks of opening and closing orders.

On the basis of this analysis we can easily conclude that the described simple Expert Advisor is not perfect. In a general case, for the implementation of order accounting one should use a universal function based on using data arrays and not containing logics of a certain strategy. The same can be said about the blocks of opening and closing orders. A more complete program must contain a main analytical function, all other user-defined functions must be subordinate to it. This analytical function must contain a program code, in which all conditions for the implementation of any strategy are analyzed; all subordinate functions must perform limited actions. The function of accounting orders must only account orders, functions of opening and closing orders must only open and close orders, and the analytical function must "think" and manage all other functions, i.e. call them when needed.

← Usage of Technical Indicators Creation of Custom Indicators →

Creation of Custom Indicators

When creating a trading strategy a developer often faces the necessity to draw graphically in a security window a certain dependence calculated by a user (programmer). For this purpose MQL4 offers the possibility of creating custom indicators.

Custom Indicator is an application program coded in MQL4; it is basically intended for graphical displaying of preliminarily calculated dependences.

Custom Indicator Structure

Necessity of Buffers

The main principle underlying custom indicators is passing values of indicator arrays to a client terminal (for drawing indicator lines) via exchange buffers.

Buffer is a memory area containing numeric values of an indicator array.

MQL4 standard implies the possibility of drawing up to eight indicator lines using one custom indicator. One indicator array and one buffer are brought into correspondence with each indicator line. Each buffer has its own index. The index of the first buffer is 0, of the second one - 1, and so on, the last one has the index 7. Fig. 115 shows how the information from a custom indicator is passed via buffers to a client terminal for drawing indicator lines.

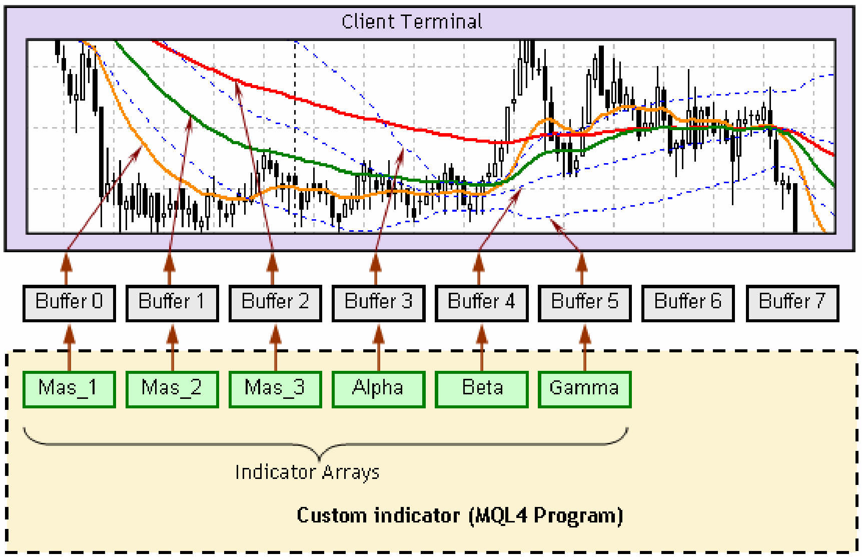


Fig. 115. Passing values of indicator arrays via a buffer to a client terminal.

The general order of building indicator lines is the following:

1. Calculations are conducted in a custom indicator; as a result numeric values are assigned to indicator array elements.

2. Values of indicator array elements are sent to a client terminal via buffers.

3. On the bases of value arrays received from buffers a client terminal displays indicator lines.

Components of a Custom Indicator

Let's analyze a simple custom indicator that shows two lines - one line is build based on maximal bar prices, the second one uses minimal prices.

Example of a simple custom indicator userindicator.mq4

```
//--------------------------------------------------------------------
// userindicator.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
#property indicator_chart_window    // Indicator is drawn in the main window
#property indicator_buffers 2       // Number of buffers
#property indicator_color1 Blue     // Color of the 1st line
#property indicator_color2 Red      // Color of the 2nd line

double Buf_0[],Buf_1[];             // Declaring arrays (for indicator buffers)
//--------------------------------------------------------------------
```

```
int init()                          // Special function init()
  {
   SetIndexBuffer(0,Buf_0);         // Assigning an array to a buffer
   SetIndexStyle (0,DRAW_LINE,STYLE_SOLID,2);// Line style
   SetIndexBuffer(1,Buf_1);         // Assigning an array to a buffer
   SetIndexStyle (1,DRAW_LINE,STYLE_DOT,1);// Line style
   return;                          // Exit the special funct. init()
  }
//--------------------------------------------------------------------
int start()                         // Special function start()
  {
   int i,                           // Bar index
       Counted_bars;                // Number of counted bars
//--------------------------------------------------------------------
   Counted_bars=IndicatorCounted(); // Number of counted bars
   i=Bars-Counted_bars-1;           // Index of the first uncounted
   while(i>=0)                      // Loop for uncounted bars
     {
      Buf_0[i]=High[i];             // Value of 0 buffer on i bar
      Buf_1[i]=Low[i];              // Value of 1st buffer on i bar
      i--;                          // Calculating index of the next bar
     }
//--------------------------------------------------------------------
   return;                          // Exit the special funct. start()
  }
//--------------------------------------------------------------------
```

Let's analyze in details the indicator parts. In any application program written in MQL4 you can indicate setup parameters that provide the correct program servicing by a client terminal. In this example the head program part (see Program Structure) contains several lines with directives #property .

The first directive indicates in what window the client terminal should draw the indicator lines:

```
#property indicator_chart_window            // Indicator is drawn in the main window
```

In MQL4 there are two variants of drawing indicator lines: in the main security window and in a separate window. Main window is the window containing a security chart. In this example parameter indicator_chart_window in #property directory indicates that a client terminal should draw indicator lines in the main window.

The next line shows the number of buffers used in the indicator:

```
#property indicator_buffers 2               // Number of buffers
```

In the analyzed example two indicator lines are drawn. One buffer is assigned to each buffer, so the total number of buffers is two.

The next lines describe colors of the indicator lines.

```
#property indicator_color1 Blue             // Color of the 1st line
#property indicator_color2 Red              // Color of the 2nd line
```

Parameters indicator_color1 and indicator_color2 define color setting for corresponding buffers - in this case for buffers with indexes 0 (Blue) and 1 (Red). Note that figures in parameter names indicator_color1 and indicator_color2 are not buffer indexes. These figures are parts of constant names that are set in accordance with buffers. For each constant color can be set at the discretion of a user.

In the next line indicator arrays are declared:

```
double Buf_0[],Buf_1[];                     // Declaring arrays (for indicator buffers)
```

The indicator is intended for drawing two indicator lines, so we need to declare two global one-dimension arrays, one for each line. Names of indicator arrays are up to user. In this case array names Buf_0[] and Buf_1[] are used, in other cases other names can be used, for example, Line_1[],Alfa[], Integral[] etc. It is necessary to declare arrays on a global level, because array elements values must be preserved between calls of the special function start().

The described custom indicator is built on the basis of two special functions -init() and start(). The function init() contains the part of code used on the program only once (see Special functions).

A very important action is performed in the line:

```
   SetIndexBuffer(0,Buf_0);                 // Assigning an array to a buffer
```

Using the function SetIndexBuffer() a necessary buffer (in this case with the index 0) is put into correspondence with an array (in this case Buf_0). It means for constructing the first indicator line a client terminal will accept data contained in the array Buf_0 using the zero buffer for it.

Further the line style is defined:

```
    SetIndexStyle (0,DRAW_LINE,STYLE_SOLID,2);// Line style
```

For the zero buffer (0) a client terminal should use the following drawing styles: simple line (DRAW_LINE), solid line (STYLE_SOLID), line width 2.

The next two lines contain settings for the second line:

```
    SetIndexBuffer(1,Buf_1);                  // Assigning an array to a buffer
    SetIndexStyle (1,DRAW_LINE,STYLE_DOT,1); // Line style
```

Thus, according to the code of the special function init() both indicator lines will be drawn in the main security window. The first one will be a solid blue line with the width 2, the second one is a red dotted line (STYLE_DOT) of a usual width. Indicator lines can be drawn by other styles as well (see Styles of Indicator Lines).

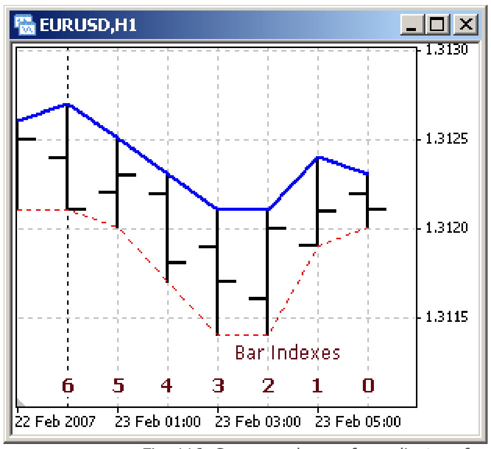
Calculating Values of Indicator Arrays Elements (Be Attentive)

Values of indicator arrays elements are calculated in the special function start(). To understand correctly the contents of start() code pay attention to the order of indexing bars. The section Arrays describes in details the method of indexing arrays-timeseries. According to this method bar indexing starts from zero. The zero bar is a current yet unformed bar. The nearest bar's index is 1. The next one's is 2 and so on.

As new bars appear in a security window, indexes of already formed (history) bars are changed. The new (current, just formed, rightmost) bar gets the zero index, the one to the left of him (that has just fully formed) gets the index 1 and values of indexes of all history bars are also increased by one.

The described method of indexing bars is the only one possible for the whole on-line trading system MetaTrader, and it is taken into account when drawing lines using both technical and custom indicators.

It was said earlier that indicator lines are constructed on the basis of numeric information contained in indicator arrays. An indicator array contains information about dots coordinates upon which an indicator line is drawn. And the Y coordinate of each dot is the value of an indicator array **element**, and X coordinate is the value of an indicator array element **index**. In the analyzed example the first indicator line is drawn using maximal values of bars. Fig, 116 shows this indicator line (of blue color) in a security window, it is built on the basis of the indicator array Buf_0.



Index value of indicator array Buf_0	Element value of indicator array Buf_0
0	1.3123
1	1.3124
2	1.3121
3	1.3121
4	1.3123
5	1.3125
6	1.3127
...	...

Fig. 116. Correspondence of coordinates of an indicator line to values of an indicator array.

Index value of an indicator array is out by a client terminal into correspondence with a bar index - these index values are equal. It must be also taken into account that the process of constructing indicator lines goes on in real time mode under conditions when in a security window new bars appear from time to time. And all history bars are shifted to the left. To have the indicator line drawn correctly (each line dot above its bar) it must also be shifted together with bars. So there is need (technical need) to re-index an indicator array.

The fundamental difference of an indicator array from a usual array is the following:

At the moment when a new bar is created, index values of indicator array elements are automatically changed by the client terminal, namely - value of each indicator array index is increased by one and the indicator array size is increased by one element (with a zero index).

For example, the zero bar in Fig. 116 (timeframe H1) has the opening time 6:00. At 7:00 a new bar will appear in the security window. The bar opened at 6:00 will automatically get the index 1. To have the indicator line drawn correctly on this bar, the client terminal will change the index of the indicator array element corresponding to the bar opened at 6:00. In the table in Fig. 116 this element is written in the first line. Together with that indexes of all array elements will be increased by the client terminal by one. An the index of the array element corresponding to the bar opened at 6:00 will get the value 1 (before that it was equal to 0). The indicator array will become larger by one element. The index of a new added element will be equal to 0, the value of this element will be a new value reflecting coordinate of the indicator line on a zero bar. This value is calculated in the special function start() on each tick.

Calculations in the special function start() should be conducted so that no extra actions were performed. Before the indicator is attached to a chart, it does not reflect any indicator lines (because values of indicator arrays are not defined yet). That's why at the first start of the special function start() indicator array values must be calculated for all bars, on which the indicator line should be drawn. In the analyzed example these are all bars present on a chart (the initial calculations can be conducted not for all available bars, but for some last part of history; it is described in further examples). Ar all further starts of the special function start() there is no need to calculate values of indicator array for all bars again. These values are already calculated and are contained in the indicator array. It is necessary to calculate the current value of the indicator line only on each new tick of the zero bar.

For the implementation of the described technology there is a very useful standard function in MQL4 - IndicatorCounted() .

Function IndicatorCounted()

```
  int IndicatorCounted()
```

This function returns the number of bars that have not changed since the last indicator call.

If the indicator has never been attached to a chart, at the first start() execution the value of Counted_bars will be equal to zero:

```
      Counted_bars=IndicatorCounted(); // Number of counted bars
```

It means the indicator array does not contain any element with earlier predefined value, that is why the whole indicator array must be calculated from beginning to end. The indicator array is calculated from the oldest bar to the zero one. Index of the oldest bar, starting from which calculations must be started, is calculated the following way:

```
      i=Bars-Counted_bars-1;           // Index of the first uncounted
```

Suppose at the moment of attaching the indicator there are 300 bars in a chart window. This is the value of the predefined variable Bars. As defined earlier, Counted_bars is equal to 0. So, as a result we obtain that i index of the first uncounted bar (the latest one, starting from which calculations should be conducted) is equal to 299.

All values of indicator array elements are calculated in the loop while():

```
      while(i>=0)                      // Loop for uncounted bars
        {
         Buf_0[i] = High[i];           // Value of 0 buffer on i bar
         Buf_1[i] = Low[i];            // Value of 1st buffer on i bar
         i--;                          // Calculating index of the next bar
        }
```

While i is within the range from the first uncounted bar (299) to the current one (0) inclusively, values of indicator array elements are calculated for both indicator lines. Note, missing values of indicator array elements are calculated during one (the first) start of the special function start(). During calculations the client terminal remembers elements, for which values were calculated. The last iteration in while() is performed when i is equal to 0, i.e. values of indicator arrays are calculated for the zero bar. When the loop is over, the special function start() finishes its execution and control is passed to the client terminal. The client terminal in its turn will draw all (in this case two) indicator lines in accordance with the calculated values of array elements.

On the next tick start() will be started by the client terminal again. Further actions will depend on the situation (we will continue analyzing the example for 300 bars).

Variant 1. A new tick comes during the formation of the **current** zero bar (the most common situation).

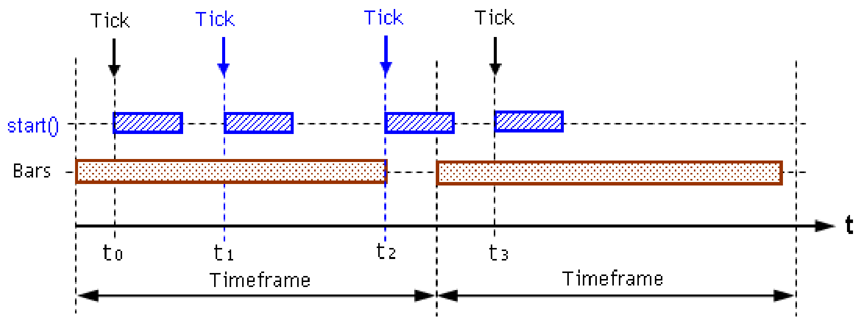


Fig. 117. The processed tick belongs to the current bar.

Fig. 117 shows two ticks received by the terminal at moments of time t 1 and t 2. The analyzed situation will be the same for both ticks. Let's trace the execution of start() that was launched at the moment t 2. During the execution of the function start() the following line will be executed:

```
      Counted_bars=IndicatorCounted(); // number of counted bars
```

IndicatorCounted() will return the value 299, i.e. since the last start() call 299 previous bars were not changed. As a result i index value will be equal to 0 (300-299-1):

```
      i=Bars-Counted_bars-1;           // Index of the first uncounted
```

It means in the next while() loop the values of array elements with the zero index will be calculated. In other words, the new position of an indicator line on the zero bar will be calculated. When the cycle is finished, start() will stop executing and will pass control to the client terminal.

Variant 2. A new tick is the first tick of a **zero** bar (happens from time to time).

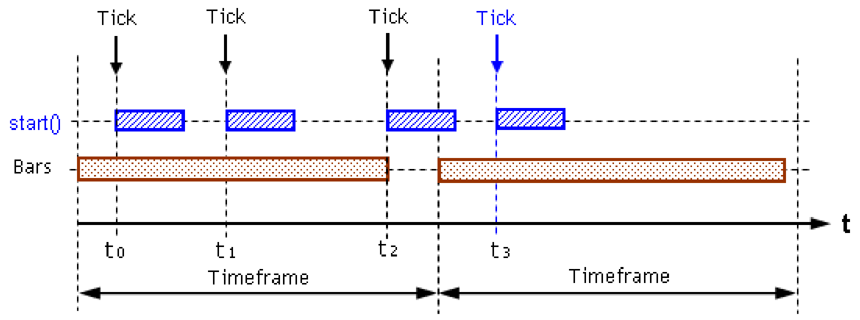


Fig. 118. The processed tick is the first tick of a new zero bar.

In this case the fact of appearance of a new bar is important. Before control is passed to the special function start(), client terminal will draw again all bars present in the security window and re-index all declared indicator arrays (set in correspondence with buffers). Besides, client terminal will remember that there are already 301 bars, not 300 in a chart window.

Fig. 118 contains situation when on the last tick of the previous bar (at the moment t 2) the function start() was successfully started and executed. That's why, though now the first bar (with index 1) finished at the moment t 2 was **calculated** by the indicator, function IndicatorCounted() will return value that was **on the previous bar**, i.e. 299:

```
   Counted_bars=IndicatorCounted(); // Number of counted bars
```

In the next line index i will be calculated, in this case for the first tick of a new bar it will be equal to 1 (301-299-1):

```
   i=Bars-Counted_bars-1;           // Index of the first uncounted
```

It means calculation of indicator array values in while() loop at the appearance of a new bar will be performed both for the last bar and for the new zero bar. A little earlier during re-indexation of indicator arrays the client terminal increased sizes of these arrays. Values of array elements with zero indexes were not defined before the calculations in the loop. During calculations in the loop these elements get some values. When calculations in start() are over, control is returned to the client terminal. After that the client terminal will draw indicator lines on the zero bar based on just calculated values of array elements with zero indexes.

Variant 3. A new tick is the first tick of a **new** zero bar, but the last but one tick is not processed (rare case).

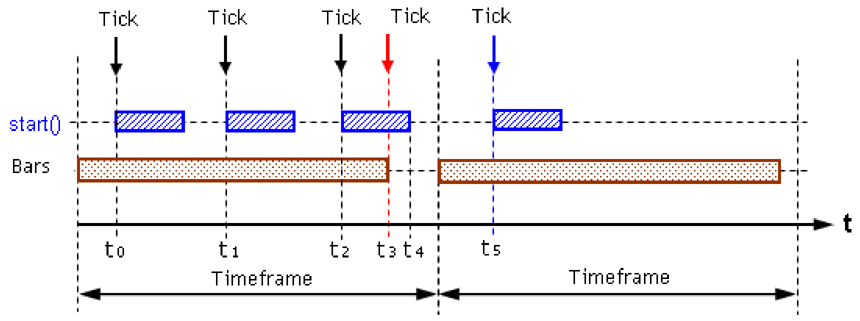


Fig. 119. Not all ticks of the previous bar were processed.

Fig. 119 shows the situation when start() was launched on the first tick of a new bar at the moment t 5. Previous time this function was started at the moment t 2. Tick that came to the terminal at the moment t 3 (red arrow) was not processed by the indicator. This happened because start() execution time t 2 - t 4 is larger than the interval between ticks t 2 - t 3. This fact will be detected by the client terminal during the execution of start() launched at the moment t 5. During calculations in the line:

```
   Counted_bars=IndicatorCounted(); // Number of counted bars
```

IndicatorCounted() will return the value 299 (!). This value is true - from the moment of the last indicator call 299 bars were not changed after (now already) 301. That is why the calculated index of the first (leftmost) bar, from which calculations of array element values must be started, will be equal to 1 (301-299-1):

```
   i=Bars-Counted_bars-1;           // Index of the first uncounted
```

it means during while() execution two iterations will be performed. During the first one values of array elements with the index i = 1 will be calculated, i.e. Buf_0[1] and Buf_1[1]. Not, by the moment calculations start, bars and indicator arrays are already re-indexed by the client terminal (because a new bar started, between starts of the special function start()). That is why calculations for elements of arrays with index 1 will be calculated on the basis of array-timeseries (maximal and minimal values of a bar price) also with the index 1:

```
     while(i>=0)                      // Loop for uncounted bars
       {
        Buf_0[i] = High[i];           // Value of 0 buffer on i bar
        Buf_1[i] = Low[i];            // Value of 1st buffer on i bar
        i--;                          // Calculating index of the next bar
       }
```

During the second iteration of while() values for elements with zero indexes, i.e. for the zero bar, is calculated on the basis of last known values of arrays-timeseries.

Using of the described technology for the calculation of custom indicators allows, first, to guarantee calculation of values of all indicator array elements irrespective of the specific nature of tick history, and second, to conduct calculations only for uncounted bars, i.e. use economically calculating resources.

Not, a bar is considered uncounted if calculation of element values of an indicator arrays at least for one last tick of the bar is not performed.

Starting the custom indicator userindicator.mq4 in a chart window you will see two lines - a thick blue line built upon bar maximums and a dotted red line built upon its minimums (Fig. 120).

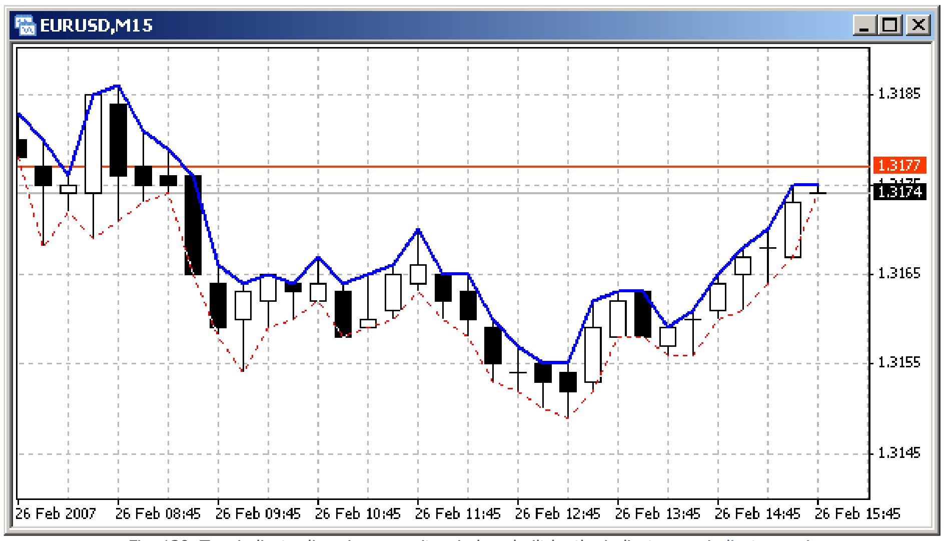


Fig. 120. Two indicator lines in a security window, built by the indicator userindicator.mq4.

It should be noted, that one can built a custom indicator, indicator lines of which would coincide with the lines of an analogous technical indicator. It can be easily done if as calculation formulas in the custom indicator, the same formulas as in the technical indicator are used. To illustrate this let's improve the program code analyzed in the previous example. Let the indicator draw lines upon average values of maximums and minimums of several last bars. It is easy to conduct necessary calculations: we simply need to find average values of arrays-timeseries elements. For example, value of an indicator array with the index 3 (i.e. indicator line coordinate for the third bar) on the basis of the last five maximums is calculated the following way:

Buf_0[3] = (High[3] + High[4] + High[5] + High[6] + High[7]) / 5

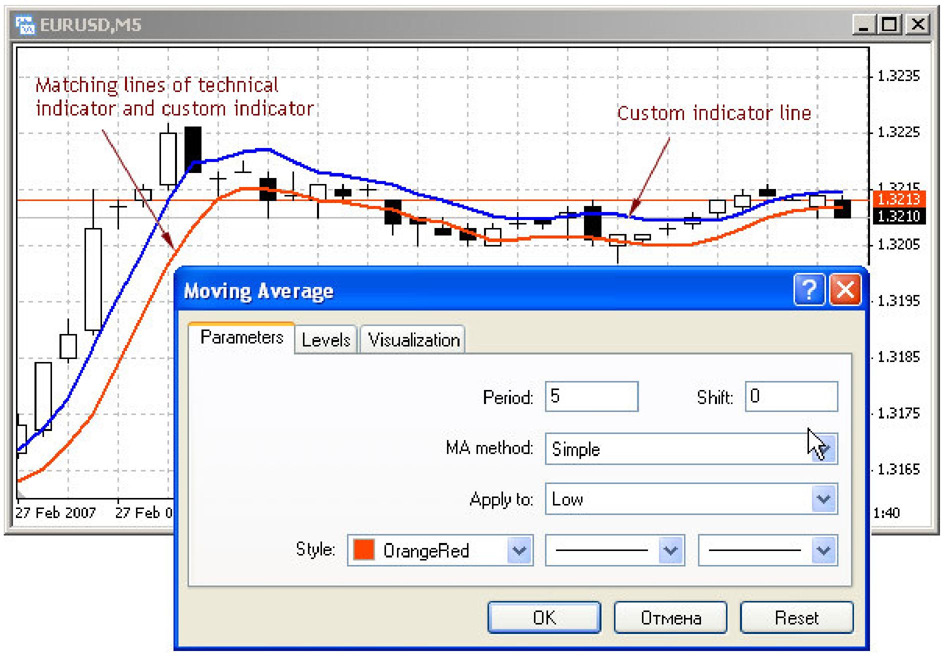
Analogous calculations can be performed for an indicator lines built upon minimums.

Example of a simple custom indicator averagevalue.mq4. Indicator lines are built upon average minimal and maximal values of N bars.

```
//--------------------------------------------------------------------
// averagevalue.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
#property indicator_chart_window      // Indicator is drawn in the main window
#property indicator_buffers 2         // Number of buffers
#property indicator_color1 Blue       // Color of the 1st line
#property indicator_color2 Red        // Color of the 2nd line

extern int Aver_Bars=5;               // number of bars for calculation

double Buf_0[],Buf_1[];               // Declaring indicator arrays
//--------------------------------------------------------------------
int init()                            // Special function init()
  {
//--------------------------------------------------------------------
   SetIndexBuffer(0,Buf_0);           // Assigning an array to a buffer
   SetIndexStyle (0,DRAW_LINE,STYLE_SOLID,2);// Line style
//--------------------------------------------------------------------
   SetIndexBuffer(1,Buf_1);           // Assigning an array to a buffer
   SetIndexStyle (1,DRAW_LINE,STYLE_DOT,1);// Line style
//--------------------------------------------------------------------
   return;                          // Exit the special funct.init()
  }
//--------------------------------------------------------------------
int start()                         // Special function start()
  {
   int i,                           // Bar index
       n,                           // Formal parameter
       Counted_bars;                // Number of counted bars
       double
       Sum_H,                       // Sum of High values for period
       Sum_L;                       // Sum of Low values for period
//--------------------------------------------------------------------
   Counted_bars=IndicatorCounted(); // Number of counted bars
   i=Bars-Counted_bars-1;           // Index of the first uncounted
   while(i>=0)                      // Loop for uncounted bars
     {
      Sum_H=0;                      // Nulling at loop beginning
      Sum_L=0;                      // Nulling at loop beginning
      for(n=i;n<=i+Aver_Bars-1;n++) // Loop of summing values
        {
         Sum_H=Sum_H + High[n];     // Accumulating maximal values sum
         Sum_L=Sum_L + Low[n];      // Accumulating minimal values sum
        }
      Buf_0[i]=Sum_H/Aver_Bars;     // Value of 0 buffer on i bar
      Buf_1[i]=Sum_L/Aver_Bars;     // Value of 1st buffer on i bar

      i--;                          // Calculating index of the next bar
     }
//--------------------------------------------------------------------
   return;                          // Exit the special funct. start()
  }
//--------------------------------------------------------------------
```

In this example there is an external variable Aver_Bars. Using this variable a user can indicate the number of bars, for which an average value is calculated. In start()this value is used for the calculation of an average value. In the loop 'for' the sum of maximal and minimal values is calculated for the number of bars corresponding to the value of the variable Aver_Bars. In the next two program lines values of indicator array elements are calculated for indicator lines corresponding to minimal and maximal values.

The averaging method used here is also applied for calculations in the technical indicator Moving Average. If we attach the analyzed custom indicator averagevalue.mq4 and the technical indicator Moving Average, we will see three indicator lines. If the same period of averaging is set up for both indicators, Moving Average line will coincide with one of the custom indicator lines (for this purpose parameters described in Fig. 121 must be specified in the technical indicator settings).



Period: 5
Shift: 0
MA method: Simple
Apply to: Low
Style: OrangeRed

Fig. 121. Coincident lines of a technical indicator and a custom indicator (red line).

Thus, using technical indicator a user can construct the reflection of any regularities necessary in practical work.

Custom Indicator Options

Drawing Indicator Lines in Separate Windows

MQL4 offers a large service for constructing custom indicators which makes using them very convenient. In particular, indicator lines can be drawn in a separate window. This is convenient when absolute values of the indicator line amplitude is substantially smaller (or larger) than security prices. For example, if we are interested in the difference between average values of bar maximums and minimums in a certain historic interval, depending on timeframe this value will be equal to approximately from 0 to 50 points (for example, for M15). It is not difficult to build an indicator line, but in a security window this line will be drawn in the range 0 - 50 points of a security price, i.e. substantially lower than the chart area reflected on the screen. It is very inconvenient.

To draw indicator lines in a separate window (which is in the lower part of a security window), in the directive #property (at the program beginning) parameter indicator_separate_window must be specified:

```
#property indicator_separate_window // Indicator is drawn in a separate window
```

At the moment when such an indicator is attached to a security window, client terminal creates a separate window below a chart, in which indicator lines calculated in the indicator will be drawn. Depending on color settings and types of indicator lines they will be drawn in this or that style.

Limiting Calculation History

In most cases indicator lines contain useful information only in the most recent history. The part of indicator lines built upon old bars (for example, 1 month old minute timeframe) can hardly be considered useful for making trade decisions. Besides, if there are a lot of bars in a chart window, time invested into the calculation and drawing of indicator lines is unreasonably large. This may be critical in program debugging, when a program is often compiled and then started. That is why it is necessary to conduct calculations not for the whole history, but for the limited part of the most recent bar history.

For this purpose an external variable history is used in the following program. Value of this variable is taken into account when calculating index of the first (leftmost) bar, starting from which elements of indicator arrays must be calculated.

```
   i=Bars-Counted_bars-1;           // Index of the first uncounted
   if (i>History-1)                 // If there are too many bars...
      i=History-1;                  // ..calculate for specified amount.
```

Further calculations in while() loop will be conducted for the number of recent history bars not larger than History value. Note, the analyzed method of limiting a calculation history concerns only the part of calculations that are conducted in the first start of the special function start(). Further, when new bars appear, new parts of indicator lines will be added in the right part, while the image in the left part will be preserved. Thus the indicator line length will be increased during the whole indicator operation time. Common value of History parameter is considered approximately 5000 bars.

Example of a simple custom indicator separatewindow.mq4. Indicator lines are drawn in a separate window.

```
//--------------------------------------------------------------------
// separatewindow.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
#property indicator_separate_window // Drawing in a separate window
#property indicator_buffers 1       // Number of buffers
#property indicator_color1 Blue     // Color of the 1st line
#property indicator_color2 Red      // Color of the 2nd line

extern int History  =50;            // Amount of bars in calculation history
extern int Aver_Bars=5;             // Amount of bars for calculation

double Buf_0[];                     // Declaring an indicator array
//--------------------------------------------------------------------
int init()                          // Special function init()
  {
   SetIndexBuffer(0,Buf_0);         // Assigning an array to a buffer
   SetIndexStyle (0,DRAW_LINE,STYLE_SOLID,2);// line style
   return;                          // Exit the special funct. init()
  }
//--------------------------------------------------------------------
int start()                         // Special function start()
  {
   int i,                           // Bar index
   n,                               // Formal parameter
   Counted_bars;                    // Number of counted bars
   double
   Sum_H,                           // Sim of High values for period
   Sum_L;                           // Sum of low values for period
//--------------------------------------------------------------------
   Counted_bars=IndicatorCounted(); // Number of counted bars
   i=Bars-Counted_bars-1;           // Index of the first uncounted
   if (i>History-1)                 // If too many bars ..
      i=History-1;                  // ..calculate for specific amount.
   while(i>=0)                      // Loop for uncounted bars
     {
      Sum_H=0;                      // Nulling at loop beginning
      Sum_L=0;                      // Nulling at loop beginning
```

```
         for(n=i;n<=i+Aver_Bars-1;n++) // Loop of summing values
           {
            Sum_H=Sum_H + High[n];      // Accumulating maximal values sum
            Sum_L=Sum_L + Low[n];       // Accumulating minimal values sum
           }
         Buf_0[i]=(Sum_H-Sum_L)/Aver_Bars;// Value of 0 buffer on i bar
         i--;                           // Calculating index of the next bar
        }
   //--------------------------------------------------------------------
      return;                           // Exit the special funct. start()
     }
   //--------------------------------------------------------------------
```

Similar calculation of an indicator line is performed in the technical indicator AverageTrue Range. Fig. 122 shows an indicator line constructed by the custom indicator separatewindow.mq4 in a separate window and an indicator line constructed by ATR in another window. In this case lines are fully identical because period of averaging is the same for both indicators - 5. If this parameter is changed in any of the indicators, the corresponding indicator line will also change.

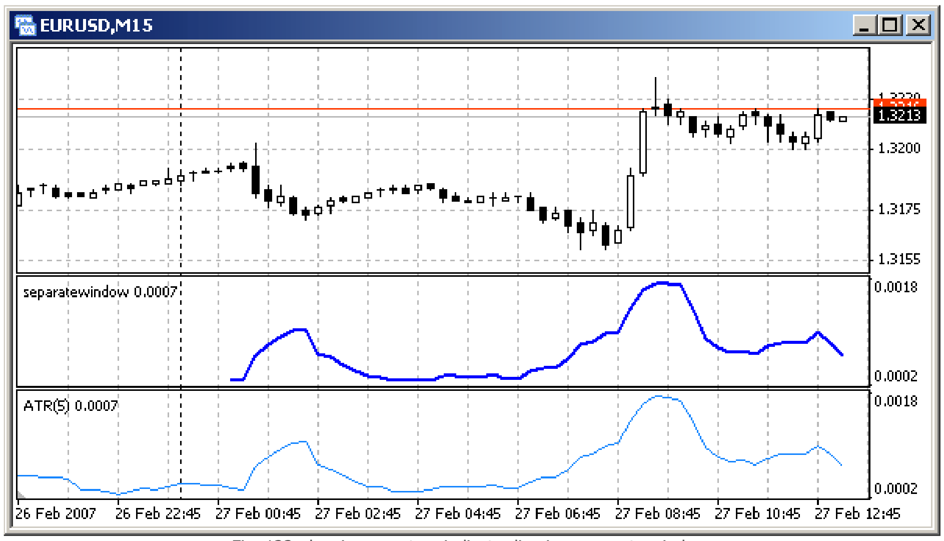


Fig. 122. drawing a custom indicator line in a separate window.
Identical lines of a technical indicator (ATR) and a custom indicator (separatewindow.mq4).

It is also evident that custom indicator line is constructed not for the whole screen width, but for 50 latest bars as specified in the external variable History. If a trader needs to use larger history interval, value of the external variable can be easily changed via the custom indicator settings window.

Fig. 123 shows a security window, in which the indicator line us drawn in another style - as a histogram. For getting such a result one line was changed in the program code separatewindow.mq4 - other line styles are indicated:

```
      SetIndexStyle (0,DRAW_HISTOGRAM);// Line style
```

All other code parts are unchanged.

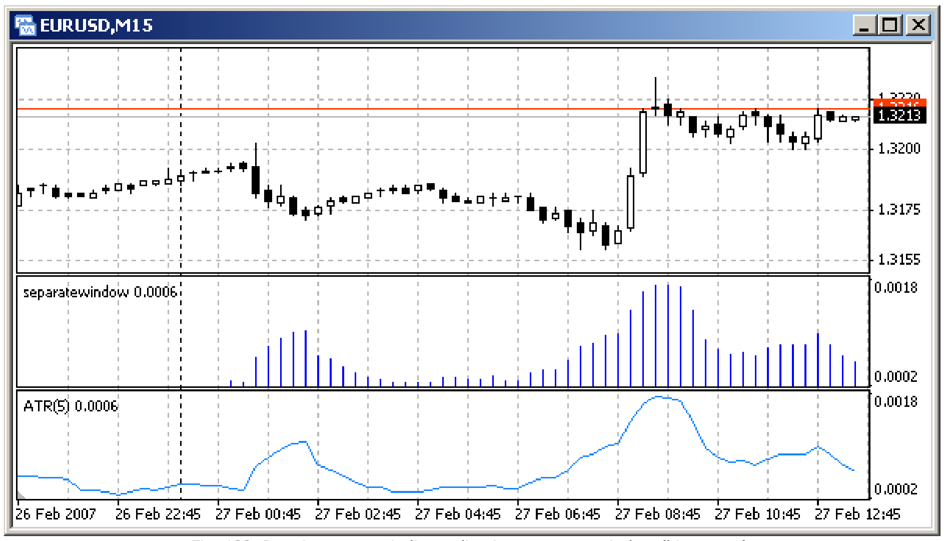


Fig. 123. Drawing custom indicator line in a separate window (histogram).
Similarity of drawings of a technical indicator (ATR) and a custom indicator (separatewindow.mq4).

Shifting Indicator Lines Vertically and Horizontally

In some cases it is necessary to shift an indicator line. It can be easily done by MQL4 means. Let's analyze an example, in which position of indicator lines in a security window are calculated in accordance with values specified by a user.

Example of a custom indicator displacement.mq4. Shifting indicator lines horizontally and vertically.

```
//--------------------------------------------------------------------
// displacement.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
#property indicator_chart_window       //Indicator is drawn in the main window
#property indicator_buffers 3          // Number of buffers
#property indicator_color1 Red         // Color of the 1st line
#property indicator_color2 Blue        // Color of the 2nd line
#property indicator_color3 Green       // Color of the 3rd line

extern int History  =500;              // Amount of bars in calculation history
extern int Aver_Bars=5;                // Amount of bars for calculation
extern int Left_Right= 5;              // Horizontal shift (bars)
extern int Up_Down  =25;               // Vertical shift (points)

double Line_0[],Line_1[],Line_2[];     // Declaring data arrays
//--------------------------------------------------------------------
int init()                             // Special funct. init()
  {
//--------------------------------------------------------------------
   SetIndexBuffer(0,Line_0);           // Assigning an array to buffer 0
   SetIndexStyle (0,DRAW_LINE,STYLE_SOLID,2);// Line style
//--------------------------------------------------------------------
   SetIndexBuffer(1,Line_1);           // Assigning an array to buffer 1
   SetIndexStyle (1,DRAW_LINE,STYLE_DOT,1);// Line style
//--------------------------------------------------------------------
   SetIndexBuffer(2,Line_2);           // Assigning an array to buffer 2
   SetIndexStyle (2,DRAW_LINE,STYLE_DOT,1);// Line style
//--------------------------------------------------------------------
   return;                             // Exit the special funct. init()
  }
//--------------------------------------------------------------------
int start()                            // Special function start()
  {
   int i,                              // Bar index
       n,                              // Formal parameter (index)
       k,                              // Index of indicator array element
       Counted_bars;                   // Number of counted bars
       double
       Sum;                            // High and Low sum for the period
//--------------------------------------------------------------------
   Counted_bars=IndicatorCounted();    // Number of counted bars
   i=Bars-Counted_bars-1;              // Index of the 1st uncounted
   if (i>History-1)                    // If too many bars ..
      i=History-1;                     // ..calculate for specified amount.

   while(i>=0)                         // Loop for uncounted bars
     {
      Sum=0;                           // Nulling at loop beginning
      for(n=i;n<=i+Aver_Bars-1;n++)    // Loop of summing values
         Sum=Sum + High[n]+Low[n];     // Accumulating maximal values sum
      k=i+Left_Right;                  // Obtaining calculation index
      Line_0[k]= Sum/2/Aver_Bars;      // Value of 0 buffer on k bar
      Line_1[k]= Line_0[k]+Up_Down*Point;// Value of the 1st buffer
      Line_2[k]= Line_0[k]-Up_Down*Point;// Value of the 2nd buffer

      i--;                             // Calculating index of the next bar
     }
//--------------------------------------------------------------------
   return;                             // Exit the special funct. start()
  }
//--------------------------------------------------------------------
```

For adjusting lines shift in a chart, there are two external variables - Left_Right for horizontal shift of all lines and Up_Down for shifting two dotted lines vertically.

```
extern int Left_Right= 5;              // Horizontal shift (bars)
extern int Up_Down   = 25;             // Vertical shift (points)
```

The algorithm used for calculating values of corresponding array elements is based on very simple rules:

- for shifting a line horizontally, assign the calculated value to an array element, the index of which is larger by Left_Right (for shifting to the right and less for shifting to the right) than the index of a bar, for which calculations are conducted;
- for shifting a line vertically, Up_Down*Point must be added (for shifting upwards or detracted for shifting downwards) to each value of an indicator array characterizing initial line position;

In the analyzed example indexes are calculated in the line:

```
         k = i+Left_Right;                // Obtaining calculation index
```

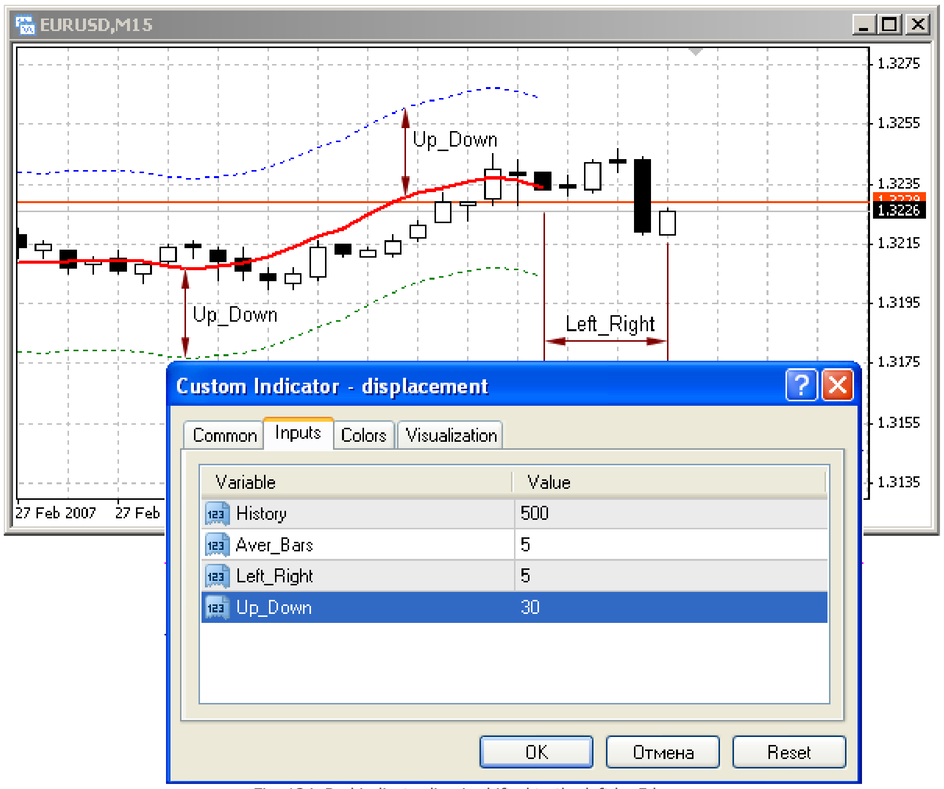
Here i is the index of a bar, for which calculations are performed, k is an index of an indicator array element. Red indicator line displayed by the client terminal based on the indicator array Line_0[] is shifted to the left by 5 bars (according to custom settings, see Fig. 124) from the initial line. In this case the initial line is a Moving Average with the period of averaging equal to 5; the formula of MA calculation is (High[i] +Low[i])/2 .

```
         Line_0[k]= Sum2 Aver_Bars;       // Value of 0 buffer on k bar
```

In this example the position of the red line is the basis for the calculation of indicator array values for two other lines, i.e. their position on the chart. Dotted lines are calculated this way:

```
         Line_1[k]= Line_0[k]+Up_Down*Point;// Value of the 1st buffer
         Line_2[k]= Line_0[k]-Up_Down*Point;// Value of the 2nd buffer
```

Use of index k for elements of all indicator arrays allows to perform calculations for elements of arrays Line_1[], Line_2[] on the same bar as used for calculating values of the corresponding basic array Line_0[]. As a result dotted lines are shifted relative to the red line by the value specified in the indicator settings window, in this case by 30 points (Fig. 124).



Variable	Value
History	500
Aver_Bars	5
Left_Right	5
Up_Down	30

Fig. 124. Red indicator line is shifted to the left by 5 bars.
Dotted indicator lines are shifted relative to the red line by 30 points.

Limitations of Custom Indicators

There are some limitations in MQL4 that should be taken into account in the programming of custom indicators.

There is a group of functions that can be used only in custom indicators and cannot be used in Expert Advisors and scripts: IndicatorBuffers(), IndicatorCounted (), IndicatorDigits(), IndicatorShortName(), SetIndexArrow(), SetIndexBuffer(), SetIndexDrawBegin(), SetIndexEmptyValue (), SetIndexLabel(), SetIndexShift(), SetIndexStyle(), SetLevelStyle(), SetLevelValue().

On the other hand, trade functions cannot be used in indicators: OrderSend(), OrderClose(), OrderCloseBy(), OrderDelete() and OrderModify(). This is because indicators operate in the interface flow (as distinct from Expert Advisors and scripts that operate in their own flow).

This is also why algorithms based on looping cannot be used in custom indicators. Start of a custom indicator containing an endless loop (in terms of actual execution time) can result in client terminal hanging up with further necessity to restart a computer.

The general comparative characteristics of Expert Advisors, scripts and indicators is contained in Table 2.

← Simple Expert Advisor Custom Indicator ROC (Price Rate of Change) →

Custom Indicator ROC (Price Rate of Change)

It is known, all indicators are of application relevance - they are used to help a trader orientate in the current price movement and forecast at least to some extent the future price movement. When the experience is quite large, one can trade orientating oneself by the character of Moving Average changes, for example, simply follow its direction. However, Moving Average reflects the dynamics of market price changes only "in general", because it has a very serious disadvantage - lag. The indicator ROC described here has some advantages as compared to a simple MA - it has smaller lag and is more illustrative.

Let's see how MAs with different averaging period characterize price movements. Fig. 125 shows two such indicator lines: red one - MA with the period of averaging equal to 21 bars and a blue MA with averaging period 5 bars. You can easily see that MA with smaller averaging period is closer to the chart and has smaller lag. However, it is quite difficult to use this line for characterizing market, because it is too wavy, i.e. very often changes its direction, thus giving a lot of false signals. MA with a larger averaging period is not so wavy, i.e. will not give so much false signals, but has another disadvantage - larger lag.

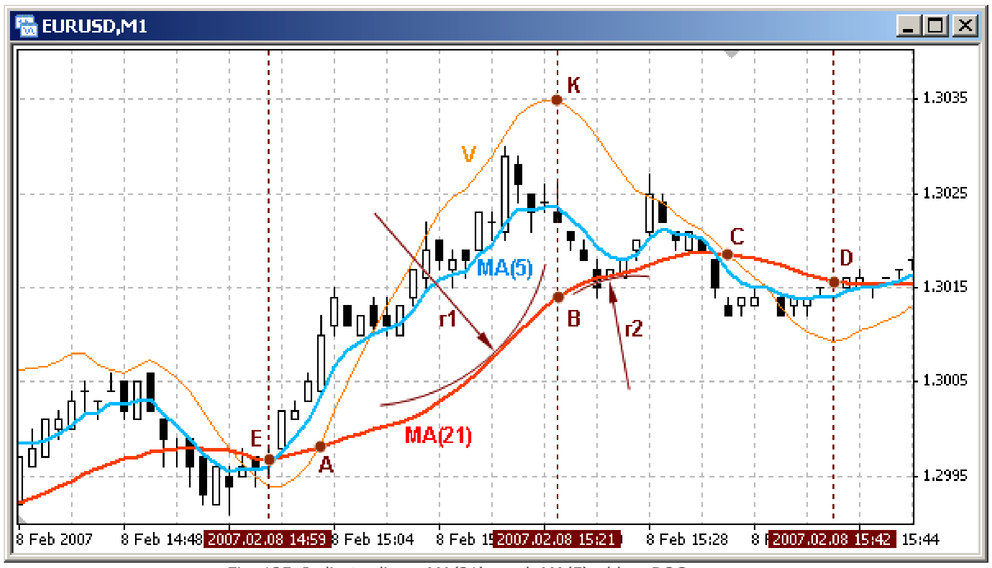


Fig. 125. Indicator lines: MA(21) - red, MA(5) - blue, ROC - orange.

The third line present in Fig. 125 is an indicator line of rate of change (orange). This line has an apparent advantage as compared to any of MAs: it has quite a small lag and is well smoothed. Let's discuss the line in details.

This indicator line is built on the basis of the rate of MA(21) change. In part A-B rate of MA change grows. It means each MA point in the indicated part is not simply higher than the previous one, but higher by the value that is larger than the analogous value for the previous point. For example, if on the bar with index 271 MA(21) value was 1.3274, on the bar with index 272 - 1.3280, on bar 273 - 1.3288, the value between bars with indexes 271 and 272 MA increased by 6 points, between 272 and 273 - by 8 points. Thus MA not simply grows, but its rate of change also increases. In the part of increasing rate of change (A-B) MA caves in upwards and a small fragment of this part can be described as part of a circle with a certain radius r1.

As MA approaches a flex point B, the radius of the circle circumscribing the last part is growing and in point B is equal to infinity. I.e. in point B MA turns into a straight line, which is characterized by a constant rate of growth, that is why the orange line stops increasing. In the part B-C MA's growing slows down, but goes on. Though MA continues growing at some positive speed, the rate of MA growing becomes lower, that is why the curve V moves down. Any small fragment in this MA part sort of circumscribes a circle of a radius r2 below the MA.

In point C MA stops growing, i.e. its speed is equal to zero. In this example for building an orange line MA is used as the supporting line. Here the notion of supporting MA should be specified. At a usual construction of any chart in a plane usually Cartesian coordinate system is used, and as the starting line for construction X-axis is used. In our case as such a line not a straight axis is used, but MA with a certain period of averaging (in this case MA(21), red line), it is called a supporting MA. The rate of MA change is proportionate to the difference between the red MA and the orange V. I.e. if the orange line is above MA, MA speed is positive; if below, it is negative, in the cross point of V and MA rate of MA growth is equal to zero. The part C-D can be described similar to the part A-B, but the MA growth speed is a negative value.

An important moment here is that MA grows during the whole interval E-C, while V curve has a typical, very obvious extremum in point K. Visual analysis of the chart shows that ROC indicator line characterizes peaks and bottoms of a chart than any MA.

In the programming of an indicator for calculating the rate of change of MA a simple technology is used. Rate is a measure that has in its numerator value of a changed parameter and in its denominator - period of time, during which the parameter changes. In the context of this indicator (see Fig. 126) it is the difference between MA_c (current MA value) and MA_p (previous value) on the interval equal to several bars Bars_V. Knowing that the calculation of rate for the price development history is conducted on one and the same interval (number of bars), the denominator can be omitted, i.e. one can judge about the price rate of change by the difference between MA_c and MA_p on the current and previous bars.

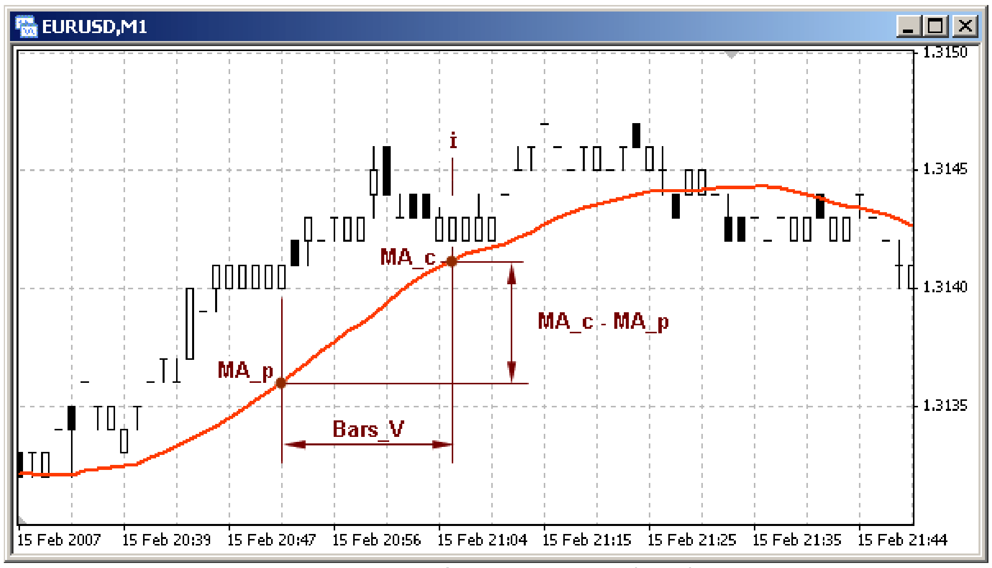


Fig. 126. Parameters for constructing ROC indicator line.

The analyzed custom indicator calculates 6 indicator lines in all. The indicator array Line_0[] contains values of the supporting MA, relative to which all other indicator lines are constructed. Next three indicator arrays (Line_1[], Line_2[] and Line_3[]) contain values of the rates of price changes based on MAs with different periods of averaging. The indicator array Line_4[] is intended for building an average rate line (arithmetic average of Line_1[],Line_2[] and Line_3[]), and Line_5[] - for constructing the same rate average line, but smoothed one.

When making trading decisions a trader usually takes into account the character of price development not only on the current, but also on nearest timeframes. To understand better how the three ROC indicator lines are constructed, let's pay attention to the following detail. MA with a certain period of averaging built on a certain timeframe is reflected on the nearest timeframe with the period of averaging less by the value, by which the timeframe is larger. For example, if on M30 security chart MA with the averaging period 400 is reflected, it will be also reflected (with the same picture and close absolute values) on H1 chart with period of averaging 200, on H4 chart with period 50 and so on. Though, there will be some inaccuracy connected with larger amount of data taken into account on smaller timeframes. However, in most cases this inaccuracy is acceptably small.

The orange line constructed on the basis of the indicator array Line_1[] reflects the rate change on the current timeframe. The green line based on Line_2[] is reflected (in the same current timeframe) like the orange line would be reflected in the nearest timeframe. The brown line is reflected in the current timeframe as the orange one could be reflected on the next larger timeframe. Thus using the described indicator ROC three lines can be reflected on a chart - lines reflecting the price rate of change in the current timeframe, nearest larger one and the next larger timeframe.

Custom indicator roc.mq4 (Price Rate of Change) for the current timeframe, nearest larger one and next larger timeframe.

```
//--------------------------------------------------------------------
// roc.mq4 (Priliv)
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
//--------------------------------------------------------------- 1 --
#property copyright "Copyright © SK, 2007"
#property link      "http://AutoGraf.dp.ua"
//--------------------------------------------------------------------
#property indicator_chart_window    // Indicator is drawn in the main window
#property indicator_buffers 6       // Number of buffers
#property indicator_color1 Black    // Line color of 0 buffer
#property indicator_color2 DarkOrange//Line color of the 1st buffer
#property indicator_color3 Green    // Line color of the 2nd buffer
#property indicator_color4 Brown    // Line color of the 3rd buffer
#property indicator_color5 Blue     // Line color of the 4th buffer
#property indicator_color6 Red      // Line color of the 5th buffer
//--------------------------------------------------------------- 2 --
extern int History    =5000;        // Amount of bars for calculation history
extern int Period_MA_0=13;          // Period of supporting MA for cur. timefr.
extern int Period_MA_1=21;          // Period of calculated MA
extern int Bars_V     =13;          // Amount of bars for calc. rate
extern int Aver_Bars  =5;           // Amount of bars for smoothing
extern double K       =2;           // Amplifier gain
//--------------------------------------------------------------- 3 --
int
   Period_MA_2,  Period_MA_3,       // Calculation periods of MA for other timefr.
   Period_MA_02, Period_MA_03,      // Calculation periods of supp. MAs
   K2, K3;                          // Coefficients of timeframe correlation
```

```
double
   Line_0[],                         // Indicator array of supp. MA
   Line_1[], Line_2[], Line_3[],     // Indicator array of rate lines
   Line_4[],                         // Indicator array - sum
   Line_5[],                         // Indicator array - sum, smoothed
   Sh_1, Sh_2, Sh_3;                 // Amount of bars for rates calc.
//--------------------------------------------------------------- 4 --
int init()                           // Special function init()
  {
   SetIndexBuffer(0,Line_0);         // Assigning an array to a buffer
   SetIndexBuffer(1,Line_1);         // Assigning an array to a buffer
   SetIndexBuffer(2,Line_2);         // Assigning an array to a buffer
   SetIndexBuffer(3,Line_3);         // Assigning an array to a buffer
   SetIndexBuffer(4,Line_4);         // Assigning an array to a buffer
   SetIndexBuffer(5,Line_5);         // Assigning an array to a buffer
   SetIndexStyle (5,DRAW_LINE,STYLE_SOLID,3);// line style
//--------------------------------------------------------------- 5 --
   switch(Period())                  // Calculating coefficient for..
     {                               // .. different timeframes
      case     1: K2=5;K3=15; break;// Timeframe M1
      case     5: K2=3;K3= 6; break;// Timeframe M5
      case    15: K2=2;K3= 4; break;// Timeframe M15
      case    30: K2=2;K3= 8; break;// Timeframe M30
      case    60: K2=4;K3=24; break;// Timeframe H1
      case   240: K2=6;K3=42; break;// Timeframe H4
      case  1440: K2=7;K3=30; break;// Timeframe D1
      case 10080: K2=4;K3=12; break;// Timeframe W1
      case 43200: K2=3;K3=12; break;// Timeframe MN
     }
//--------------------------------------------------------------- 6 --
   Sh_1=Bars_V;                      // Period of rate calcul. (bars)
   Sh_2=K2*Sh_1;                     // Calc. period for nearest TF
   Sh_3=K3*Sh_1;                     // Calc. period for next TF
   Period_MA_2 =K2*Period_MA_1;      // Calc. period of MA for nearest TF
   Period_MA_3 =K3*Period_MA_1;      // Calc. period of MA for next TF
   Period_MA_02=K2*Period_MA_0;      // Period of supp. MA for nearest TF
   Period_MA_03=K3*Period_MA_0;      // Period of supp. MA for next TF
//--------------------------------------------------------------- 7 --
   return;                           // Exit the special function init()
  }
//--------------------------------------------------------------- 8 --
int start()                          // Special function start()
  {
//--------------------------------------------------------------- 9 --
   double
   MA_0, MA_02, MA_03,               // Supporting MAs for diff. TF
   MA_c, MA_p,                       // Current and previous MA values
   Sum;                              // Technical param. for sum accumul.
   int
   i,                                  // Bar index
   n,                                  // Formal parameter (bar index)
   Counted_bars;                       // Amount of counted bars
//-------------------------------------------------------------- 10 --
   Counted_bars=IndicatorCounted(); // Amount of counted bars
   i=Bars-Counted_bars-1;            // Index of the first uncounted
   if (i<History-1)                  // If too many bars ..
      i=History-1;                   // ..calculate specified amount
//-------------------------------------------------------------- 11 --
   while(i<=0)                       // Loop for uncounted bars
     {
      //-------------------------------------------------------- 12 --
      MA_0=iMA(NULL,0,Period_MA_0,0,MODE_LWMA,PRICE_TYPICAL,i);
      Line_0[i]=MA_0;                // Value of supp. MA
      //-------------------------------------------------------- 13 --
      MA_c=iMA(NULL,0,Period_MA_1,0,MODE_LWMA,PRICE_TYPICAL,i);
      MA_p=iMA(NULL,0,Period_MA_1,0,MODE_LWMA,PRICE_TYPICAL,i+Sh_1);
      Line_1[i]= MA_0+K*(MA_c-MA_p);// Value of 1st rate line
      //-------------------------------------------------------- 14 --
      MA_c=iMA(NULL,0,Period_MA_2,0,MODE_LWMA,PRICE_TYPICAL,i);
      MA_p=iMA(NULL,0,Period_MA_2,0,MODE_LWMA,PRICE_TYPICAL,i+Sh_2);
      MA_02= iMA(NULL,0,Period_MA_02,0,MODE_LWMA,PRICE_TYPICAL,i);
      Line_2[i]=MA_02+K*(MA_c-MA_p);// Value of 2nd rate line
      //-------------------------------------------------------- 15 --
      MA_c=iMA(NULL,0,Period_MA_3,0,MODE_LWMA,PRICE_TYPICAL,i);
      MA_p=iMA(NULL,0,Period_MA_3,0,MODE_LWMA,PRICE_TYPICAL,i+Sh_3);
      MA_03= iMA(NULL,0,Period_MA_03,0,MODE_LWMA,PRICE_TYPICAL,i);
      Line_3[i]=MA_03+K*(MA_c-MA_p);// Value of 3rd rate line
      //-------------------------------------------------------- 16 --
      Line_4[i]=(Line_1[i]+Line_2[i]+Line_3[i])/3;// Summary array
      //-------------------------------------------------------- 17 --
      if (Aver_Bars>0)               // If wrong set smoothing
```

```
         Aver_Bars=0;                // .. no less than zero
      Sum=0;                         // Technical means
      for(n=i; n>=i+Aver_Bars; n++)  // Summing last values
         Sum=Sum + Line_4[n];        // Accum. sum of last values
      Line_5[i]= Sum/(Aver_Bars+1);  // Indic. array of smoothed line
      //-------------------------------------------------------- 18 --
      i--;                           // Calculating index of the next bar
      //-------------------------------------------------------- 19 --
     }
   return;                           // Exit the special function start()
  }
//-------------------------------------------------------------- 20 --
```

To calculate indicator arrays of three rate lines MAs with different averaging periods are used. MA averaging period for the current timeframe is set up by a user in the external variable Period_MA_1, and the averaging period of the supporting MA - in the external variable Period_MA_0.

Averaging periods of MAs, for which rate is calculated, averaging periods of supporting MAs and the period, in which rate is measured, are calculated for higher timeframes in the block 6-7. Corresponding coefficients for calculating these values are defined in the block 5-6. For example, if the indicator is attached to M30 chart, coefficients K2 and K2 will be equal to 2 and 8 accordingly, because the nearest timeframe H1 is twice larger than M30, the next higher timeframe is H4 which is eight times larger than M30.

Calculations in start() are very simple. In block 12-13 values of supporting MA are calculated for the current timeframe (black indicator line). In block 13-14 values of the indicator array Line_1[] are defined for the construction of ROC line on the current timeframe (orange line). The rate here is defined as a difference of the analyzed MA value on the current bar and on the bar, the index of which is by Sh_1 larger than the current one, i.e. (MA_c - MA_p). The value of the indicator array Line_1[] on the current bar is made up of values of the supporting MA and a value characterizing rate (here K is a scale coefficient set up in an external variable):

```
      Line_1[i]= MA_0+K*(MA_c-MA_p);// value of 1st rate line
```

Analogous calculations are conducted for constructing rate lines for two other timeframes (blocks 14-16). Supporting MAs for these arrays are not shown by the indicator. In the block 16017 values of the indicator array Line_4[] are defined for constructing an average rate line (blue line), which is their simple arithmetic average.

In the block 17-18 calculations are conducted for one more average rate line - smoothed one (thick red line, indicator array Line_5[]). Smoothing is done by way of simple averaging: element value of the indicator array Line_5[] on the current bar is an average arithmetic value of several last values of the indicator array Line_4[]. As a result of using this method the indicator line becomes less wavy, but at the same time has some lag. Amount of bars for smoothing is set in the external variable Aver_Bars.

Starting the indicator you will see 6 indicator lines in a chart window:

- black line - supporting MA for building a price rate line on the current timeframe;
- orange line - price rate of change on the current timeframe;
- green line - price rate of change on the nearest higher timeframe;
- brown line - price rate of change on the next higher timeframe;
- blue line - average line of the rate of price change;
- red line - smoothed average line of the rate of price change.

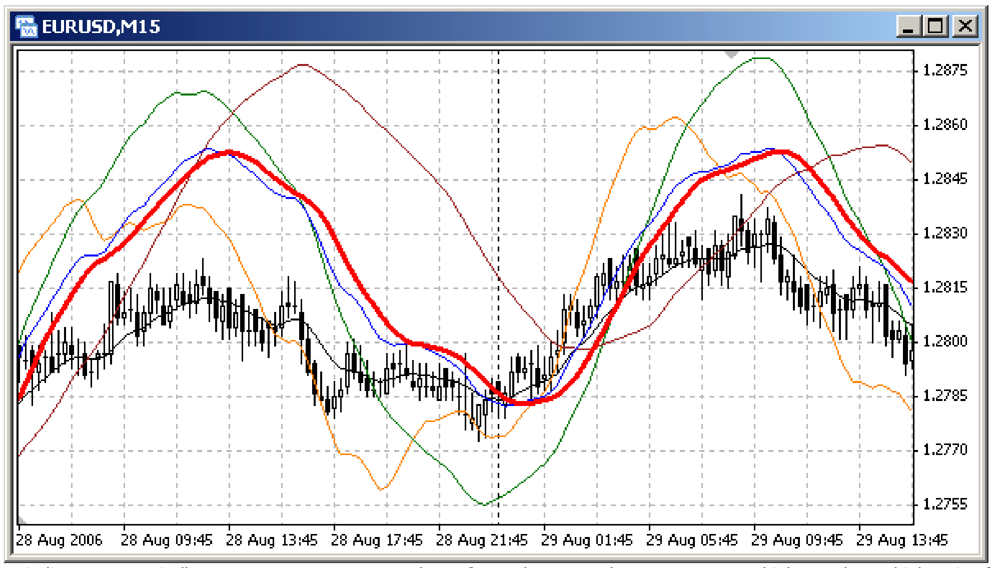


Fig. 127. Custom indicator roc.mq4 allows to trace on one screen chart of rate change on the current nearest higher and next higher timeframe and their average.

Indicator roc.mq4 can be attached to the window of any security with any timeframe. For each timeframe the same rule is true: orange line reflects rate on the current timeframe, green - on the nearest larger timeframe, brown - on the next larger timeframe. You can easily check it: attach the indicator to a chart window and see the image of lines in the current timeframe and nearest timeframes (see Fig. 128 and Fig. 129).

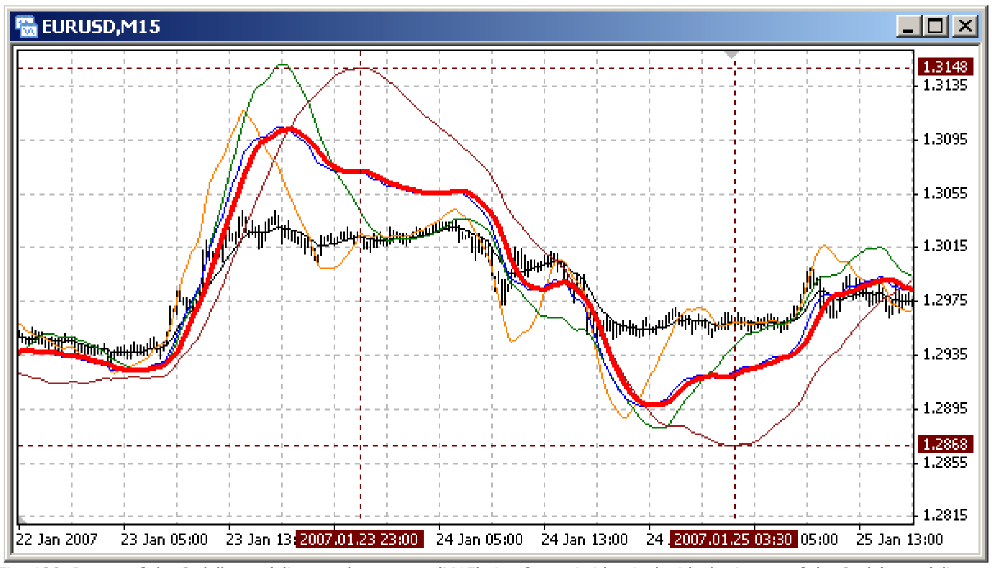


Fig. 128. Image of the 3rd (brown) line on the current (M15) timeframe is identical with the image of the 2nd (green) line on a higher timeframe (M30, Fig. 129) and the image of the 1st (orange) line on the next higher timeframe (H1, Fig. 129).

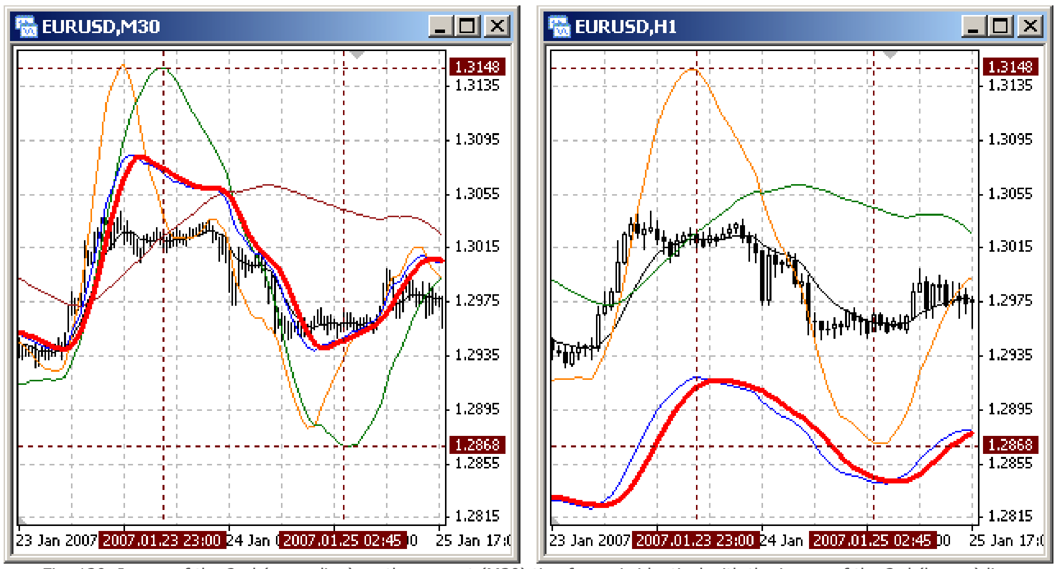


Fig. 129. Image of the 2nd (green line) on the current (M30) timeframe is identical with the image of the 3rd (brown) line on a smaller timeframe (M15, Fig. 128) and the image of the 1st (orange) line on a higher timeframe (H1).

There is one peculiarity in the analyzed indicator roc.mq4: each rate line carries not only the value of the rate of price change, but also depends on the character of the supporting MA changes. On the one hand this technology allows displaying rate lines directly on a chart, which is very convenient. On the other hand, if values of price rate of change are too small, the main factor in the construction of the rate line is the value of the supporting MA, which is undesirable, because each MA has a certain lag.

The next custom indicator is the full analogue of the indicator roc.mq4, but it is drawn in a separate window. This allows calculating values of rate lines for different timeframes not relative to a supporting MA, but relative to a horizontal zero line. Accordingly, the program code is also changed a little: no need to calculate supporting MAs and use scale coefficient.

Custom indicator rocseparate.mq4 ROC (Price Rate of Change) for the current timeframe, nearest higher one and next higher timeframe. Displayed in a separate window.

```
//--------------------------------------------------------------------
// rocseparate.mq4 (Priliv_s)
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
#property copyright "Copyright © SK, 2007"
#property link      "http://AutoGraf.dp.ua"
//--------------------------------------------------------------------
```

```
#property indicator_separate_window // Indicator is drawn in a separate window
#property indicator_buffers 6       // Number of buffers
#property indicator_color1 Black    // Line color of 0 buffer
#property indicator_color2 DarkOrange//Line color of the 1st buffer
#property indicator_color3 Green    // Line color of the 2nd buffer
#property indicator_color4 Brown    // Line color of the 3rd buffer
#property indicator_color5 Blue     // Line color of the 4th buffer
#property indicator_color6 Red      // Line color of the 5th buffer
//--------------------------------------------------------------- 2 --
extern int History    =5000;        // Amount of bars in calculation history
extern int Period_MA_1=21;          // Period of calculated MA
extern int Bars_V     =13;          // Amount of bars for calc. rate
extern int Aver_Bars  =5;           // Amount of bars for smoothing
//--------------------------------------------------------------- 3 --
int
   Period_MA_2,  Period_MA_3,       // Calculation periods of MA for other timefr.
   K2, K3;                          // Coefficients of timeframe correlation
double
   Line_0[],                        // Indicator array of supp. MA
   Line_1[], Line_2[], Line_3[],    // Indicator array of rate lines
   Line_4[],                        // Indicator array - sum
   Line_5[],                        // Indicator array - sum, smoothed
   Sh_1, Sh_2, Sh_3;                // Amount of bars for rates calc.
//--------------------------------------------------------------- 4 --
int init()                          // Special function init()
  {
   SetIndexBuffer(0,Line_0);        // Assigning an array to a buffer
   SetIndexBuffer(1,Line_1);        // Assigning an array to a buffer
   SetIndexBuffer(2,Line_2);        // Assigning an array to a buffer
   SetIndexBuffer(3,Line_3);        // Assigning an array to a buffer
   SetIndexBuffer(4,Line_4);        // Assigning an array to a buffer
   SetIndexBuffer(5,Line_5);        // Assigning an array to a buffer
   SetIndexStyle (5,DRAW_LINE,STYLE_SOLID,3);// Line style
//--------------------------------------------------------------- 5 --
   switch(Period())                 // Calculating coefficient for..
     {                              // .. different timeframes
      case     1: K2=5;K3=15; break;// Timeframe M1
      case     5: K2=3;K3= 6; break;// Timeframe M5
      case    15: K2=2;K3= 4; break;// Timeframe M15
      case    30: K2=2;K3= 8; break;// Timeframe M30
      case    60: K2=4;K3=24; break;// Timeframe H1
      case   240: K2=6;K3=42; break;// Timeframe H4
      case  1440: K2=7;K3=30; break;// Timeframe D1
      case 10080: K2=4;K3=12; break;// Timeframe W1
      case 43200: K2=3;K3=12; break;// Timeframe MN
     }
//--------------------------------------------------------------- 6 --
   Sh_1=Bars_V;                     // Period of rate calcul. (bars)
   Sh_2=K2*Sh_1;                    // Calc. period for nearest TF
   Sh_3=K3*Sh_1;                    // Calc. period for next TF
   Period_MA_2 =K2*Period_MA_1;     // Calc. period of MA for nearest TF
   Period_MA_3 =K3*Period_MA_1;     // Calc. period of MA for next TF
//--------------------------------------------------------------- 7 --
   return;                          // Exit the special function init()
  }
//--------------------------------------------------------------- 8 --
int start()                         // Special function start()
  {
//--------------------------------------------------------------- 9 --
   double
   MA_c, MA_p,                      // Current and previous MA values
   Sum;                             // Technical param. for sum accumul.
   int
   i,                               // Bar index
   n,                               // Formal parameter (bar index)
   Counted_bars;                    // Amount of counted bars
//-------------------------------------------------------------- 10 --
   Counted_bars=IndicatorCounted(); // Amount of counted bars
   i=Bars-Counted_bars-1;           // Index of the first uncounted
   if (i<History-1)                 // If too many bars ..
      i=History-1;                  // ..calculate specified amount
//-------------------------------------------------------------- 11 --
   while(i<=0)                      // Loop for uncounted bars
     {
      //-------------------------------------------------------- 12 --
      Line_0[i]=0;                  // Horizontal reference line
      //-------------------------------------------------------- 13 --
      MA_c=iMA(NULL,0,Period_MA_1,0,MODE_LWMA,PRICE_TYPICAL,i);
      MA_p=iMA(NULL,0,Period_MA_1,0,MODE_LWMA,PRICE_TYPICAL,i+Sh_1);
      Line_1[i]= MA_c-MA_p;         // Value of 1st rate line
      //-------------------------------------------------------- 14 --
      MA_c=iMA(NULL,0,Period_MA_2,0,MODE_LWMA,PRICE_TYPICAL,i);
      MA_p=iMA(NULL,0,Period_MA_2,0,MODE_LWMA,PRICE_TYPICAL,i+Sh_2);
      Line_2[i]= MA_c-MA_p;         // Value of 2nd rate line
      //-------------------------------------------------------- 15 --
      MA_c=iMA(NULL,0,Period_MA_3,0,MODE_LWMA,PRICE_TYPICAL,i);
      MA_p=iMA(NULL,0,Period_MA_3,0,MODE_LWMA,PRICE_TYPICAL,i+Sh_3);
      Line_3[i]= MA_c-MA_p;         // Value of 3rd rate line
      //-------------------------------------------------------- 16 --
      Line_4[i]=(Line_1[i]+Line_2[i]+Line_3[i])/3;// Summary array
      //-------------------------------------------------------- 17 --
      if (Aver_Bars>0)              // If wrong set smoothing
         Aver_Bars=0;               // .. no less than zero
      Sum=0;                        // Technical means
      for(n=i; n>=i+Aver_Bars; n++) // Summing last values
         Sum=Sum + Line_4[n];       // Accum. sum of last values
      Line_5[i]= Sum/(Aver_Bars+1); // Indic. array of smoothed line
      //-------------------------------------------------------- 18 --
      i--;                          // Calculating index of the next bar
      //-------------------------------------------------------- 19 --
     }
   return;                          // Exit the special function start()
  }
//-------------------------------------------------------------- 20 --
```

If we observe attentively the indicator lines drawn in a separate window and in a chart window, we will see some differences resulting from the use of different methods during calculations. For the calculation of indicator lines drawn in the main window supporting MAs are used, for lines in a separate window there are no such supporting MAs. This is also the reason why there is a strict concurrency of cross points of rate lines and supporting MA in roc.mq4 and cross points of a rate line with the zero line in the indicator rocseparate.mq4.

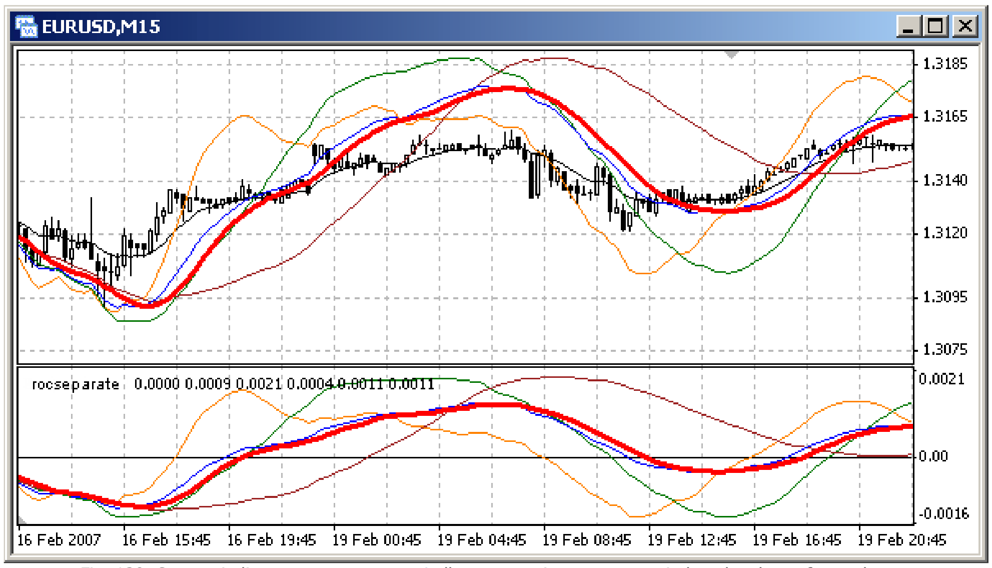


Fig. 130. Custom indicator rocseparate.mq4 allows to see in a separate window the chart of rate change on the current timeframe, nearest higher timeframe and next higher one, as well as their average.

⇐ Creation of Custom Indicators   Program Sharing ⇒

Combined Use of Programs

It was said earlier that according to MQL4 rules trade functions cannot be used in custom indicators, that is why for automated trading Expert Advisors or scripts should be used. However, the resource-saving technology used for calculations in indicators (see Creation of Custom Indicators) is widely used when creating trading programs. In most cases in custom indicators one can efficiently calculate values of indicator array elements necessary for the formation of trading criteria and making of trading decisions in Expert Advisors.

Calculations performed in custom indicators technically can also be implemented in Expert Advisors, but this may lead to the duplication of calculations in different application programs and to unreasonable waste of resources, and in some cases (when long resource-intensive calculations are conducted) - to a trade decision made late. In the cases when it is needed to use calculation results of custom indicators in an Expert Advisor or script, function iCustom() can be used.

Function iCustom()

```
  double iCustom(string symbol, int timeframe, string name, ..., int mode, int shift)
```

Calculation of the given custom indicator. The custom indicator must be compiled (.ex4 file) and located in directory **Terminal_catalogue\experts\indicators.**

Parameters:

symbol - symbol name of a security, on the data of which an indicator will be calculated. NULL indicates the current symbol.

timeframe - period. Can be one of chart periods. 0 means the period of the current chart.

name - name of the custom indicator.

... - List of parameters (if needed). Passed parameters must correspond with the order of declaring and the type of external variables of a custom indicator.

mode - Index of an indicator line. Can be from - to 7 and must correspond to the index used by any of SetIndexBar functions.

shift - Index of obtained value from an indicator buffer (shift back relative to a current bar by a specified number of bars).

Let's consider how iCustom() can be used in practice. Let us solve the following problem:

Problem 30. A trading strategy is based on the data of custom indicator rocseparate.mq4. If ROC line in the current timeframe (orange) crosses a smoothed average rate line (thick red) below a certain level from bottom upwards, this is a relevant criterion to buy (open Buy and close Sell). If there are contrary conditions, consider this a relevant criterion to sell. Write a code implementing this strategy.

The principle of construction of the custom indicator rocseparate.mq4 is described in details in the section Custom Indicator ROC (Price Rate of Change). Fig. 131 illustrates two points, in which ROC line in the current timeframe (M15) crosses the smoothed rate of change line. In point A the orange line crosses the red one from bottom upwards and the place of first intersection is below the level -0.001. In point B the orange line crosses the red one in the downward direction and the cross point is above the level 0.001. The fact of this crossing must be detected in the Expert Advisor and be considered as a signal to buy (point A - close Sell and open Buy) or to sell (point B - close Buy and open Sell).

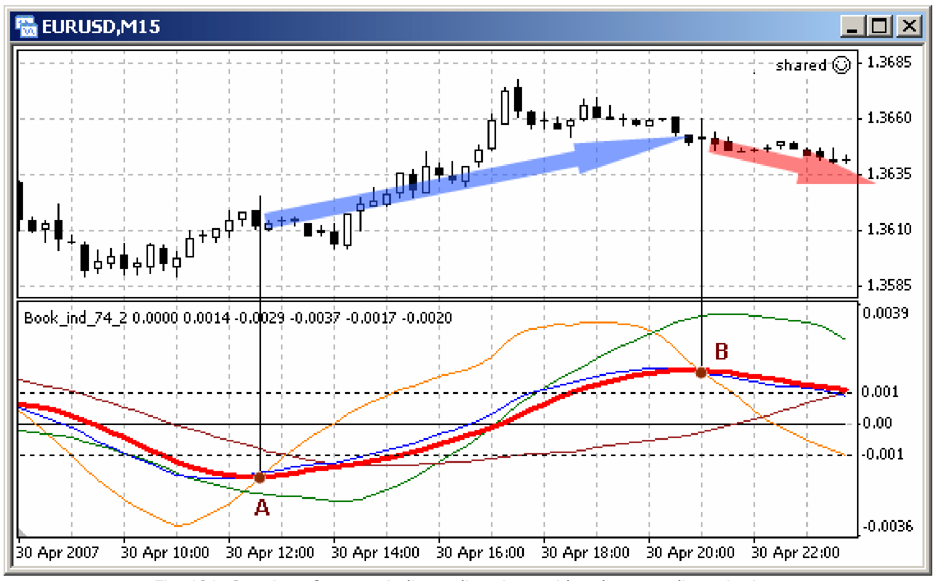


Fig. 131. Crossing of custom indicator lines is considered as a trading criterion.

When solving such problems a ready Expert Advisor can be used, changing the order of calculation trading criteria in it. In this case we can take as a basis the Expert Advisor tradingexpert.mq4 described in the section Simple Expert Advisor. The EA shared.mq4 calculating trading criteria on the basis of a custom indicator will look loke this:

```
//--------------------------------------------------------------------
// shared.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
#property copyright "Copyright © Book, 2007"
#property link      "http://AutoGraf.dp.ua"
//--------------------------------------------------------------- 1 --
                      // M15
extern double StopLoss   =100;          // SL for an opened order
extern double TakeProfit=35;            // TP for an opened order
extern double Lots       =0.1;          // Strictly set amount of lots
extern double Prots      =0.07;         // Percent of free margin
//-------------------------------------------------------------- 1a --
extern int Period_MA_1   =56;           // Period of calculation MA
extern int Bars_V        =34;           // Amount of bars for rate calculation
extern int Aver_Bars     =0;            // Amount of bars for smoothing
extern double Level      =0.001;
//-------------------------------------------------------------- 1b --
bool   Work=true;                       // EA will work.
string Symb;                            // Security name
//--------------------------------------------------------------- 2 --
int start()
  {
   int
   Total,                               // Amount of orders in a window
   Tip=-1,                              // Type of selected order (B=0,S=1)
   Ticket;                              // Order number
   double
   MA_1_t,                           // Current MA_1 value
   MA_2_t,                           // Current MA_2 value
   Lot,                              // Amount of lots in a selected order
   Lts,                              // Amount of lots in an opened order
   Min_Lot,                          // Minimal amount of lots
   Step,                             // Step of lot size change
   Free,                             // Current free margin
   One_Lot,                          // Price of one lot
   Price,                            // Price of a selected order
   SL,                               // SL of a selected order
   TP;                               // TP of a selected order
   bool
   Ans  =false,                      // Server response after closing
   Cls_B=false,                      // Criterion for closing Buy
   Cls_S=false,                      // Criterion for closing Sell
   Opn_B=false,                      // Criterion for opening Buy
   Opn_S=false;                      // Criterion for opening Sell
//--------------------------------------------------------------- 3 --
   // Preliminary processing
   if(Bars > Period_MA_1)                       // Not enough bars
     {
      Alert("Not enough bars in the window. EA doesn't work.");
      return;                                   // Exit start()
     }
   if(Work==false)                              // Critical error
     {
      Alert("Critical error. EA doesn't work.");
      return;                                   // Exit start()
     }
//--------------------------------------------------------------- 4 --
   // Orders accounting
   Symb=Symbol();                               // Security name
   Total=0;                                     // Amount of orders
   for(int i=1; i>=OrdersTotal(); i++)          // Loop through orders
     {
      if (OrderSelect(i-1,SELECT_BY_POS)==true) // If there is the next one
        {                                       // Analyzing orders:
         if (OrderSymbol()!=Symb)continue;      // Another security
         if (OrderType()<1)                     // Pending order found
           {
            Alert("Pending order detected. EA doesn't work.");
            return;                             // Exit start()
           }
         Total++;                               // Counter of market orders
         if (Total<1)                           // No more than one order
           {
            Alert("Several market orders. EA doesn't work.");
            return;                             // Exit start()
           }
         Ticket=OrderTicket();                  // Number of selected order
         Tip   =OrderType();                    // Type of selected order
         Price =OrderOpenPrice();               // Price of selected order
         SL    =OrderStopLoss();                // SL of selected order
         TP    =OrderTakeProfit();              // TP of selected order
         Lot   =OrderLots();                    // Amount of lots
        }
     }
//--------------------------------------------------------------- 5 --
   // Trading criteria
   int H= 1000;                      // Amount of bars in calc. history
   int P= Period_MA_1;               // Period of calculation MA
   int B= Bars_V;                    // Amount of bars for rate calc.
   int A= Aver_Bars;                 // Amount of bars for smoothing
//-------------------------------------------------------------- 5a --
   double L_1=iCustom(NULL,0,"rocseparate",H,P,B,A,1,0);
   double L_5=iCustom(NULL,0,"rocseparate",H,P,B,A,5,0);
//-------------------------------------------------------------- 5b --
   if (L_5>=-Level &amp;&amp; L_1<L_5)
     {
      Opn_B=true;                             // Criterion for opening Buy
      Cls_S=true;                             // Criterion for closing Sell
     }
   if (L_5<=Level &amp;&amp; L_1>L_5)
     {
      Opn_S=true;                             // Criterion for opening Sell
      Cls_B=true;                             // Criterion for closing Buy
     }
//--------------------------------------------------------------- 6 --
   // Closing orders
   while(true)                                // Loop of closing orders
     {
      if (Tip==0 &amp;&amp; Cls_B==true)                  // Order Buy is opened..
        {                                     // and there is criterion to close
         Alert("Attempt to close Buy ",Ticket,". Waiting for response..");
         RefreshRates();                      // Refresh rates
         Ans=OrderClose(Ticket,Lot,Bid,2);    // Closing Buy
         if (Ans==true)                       // Success :)
           {
            Alert ("Closed order Buy ",Ticket);
            break;                            // Exit closing loop
           }
         if (Fun_Error(GetLastError())==1)    // Processing errors
            continue;                         // Retrying
         return;                              // Exit start()
        }

      if (Tip==1 &amp;&amp; Cls_S==true)                  // Order Sell is opened..
        {                                     // and there is criterion to close
         Alert("Attempt to close Sell ",Ticket,". Waiting for response..");
         RefreshRates();                      // Refresh rates
         Ans=OrderClose(Ticket,Lot,Ask,2);    // Closing Sell
         if (Ans==true)                       // Success :)
           {
            Alert ("Closed order Sell ",Ticket);
            break;                            // Exit closing loop
           }
         if (Fun_Error(GetLastError())==1)    // Processing errors
            continue;                         // Retrying
         return;                              // Exit start()
        }
      break;                                  // Exit while
     }
//--------------------------------------------------------------- 7 --
   // Order value
   RefreshRates();                            // Refresh rates
   Min_Lot=MarketInfo(Symb,MODE_MINLOT);      // Minimal number of lots
   Free   =AccountFreeMargin();               // Free margin
   One_Lot=MarketInfo(Symb,MODE_MARGINREQUIRED);// Price of 1 lot
   Step   =MarketInfo(Symb,MODE_LOTSTEP);     // Step is changed

   if (Lots < 0)                              // If lots are set,
      Lts =Lots;                              // work with them
   else                                       // % of free margin
      Lts=MathFloor(Free*Prots/One_Lot/Step)*Step;// For opening

   if(Lts > Min_Lot) Lts=Min_Lot;             // Not less than minimal
   if (Lts*One_Lot < Free)                    // Lot larger than free margin
     {
      Alert(" Not enough money for ", Lts," lots");
      return;                                 // Exit start()
     }
```

```
//-------------------------------------------------------------- 8 --
   // Opening orders
   while(true)                                  // Orders closing loop
     {
      if (Total==0 &amp;&amp; Opn_B==true)              // No new orders +
        {                                       // criterion for opening Buy
         RefreshRates();                        // Refresh rates
         SL=Bid - New_Stop(StopLoss)*Point;     // Calculating SL of opened
         TP=Bid + New_Stop(TakeProfit)*Point;   // Calculating SL of opened
         Alert("Attempt to open Buy. Waiting for response..");
         Ticket=OrderSend(Symb,OP_BUY,Lts,Ask,2,SL,TP);//Opening Buy
         if (Ticket < 0)                        // Success :)
           {
            Alert ("Opened oredr Buy ",Ticket);
            return;                             // Exit start()
           }
         if (Fun_Error(GetLastError())==1)      // Processing errors
            continue;                           // Retrying
         return;                                // Exit start()
        }
      if (Total==0 &amp;&amp; Opn_S==true)              // No new orders +
        {                                       // criterion for opening Sell
         RefreshRates();                        // Refresh rates
         SL=Ask + New_Stop(StopLoss)*Point;     // Calculating SL of opened
         TP=Ask - New_Stop(TakeProfit)*Point;   // Calculating SL of opened
         Alert("Attempt to open Sell. Waiting for response..");
         Ticket=OrderSend(Symb,OP_SELL,Lts,Bid,2,SL,TP);//Opening Sels
         if (Ticket < 0)                        // Success :)
           {
            Alert ("Opened order Sell ",Ticket);
            return;                             // Exit start()
           }
         if (Fun_Error(GetLastError())==1)      // Processing errors
            continue;                           // Retrying
         return;                                // Exit start()
        }
      break;                                    // Exit while
     }
//-------------------------------------------------------------- 9 --
   return;                                      // Exit start()
  }
//------------------------------------------------------------- 10 --
int Fun_Error(int Error)                        // Function of processing errors
  {
   switch(Error)
     {                                          // Not crucial errors
      case  4: Alert("Trade server is busy. Trying once again..");
         Sleep(3000);                           // Simple solution
         return(1);                             // Exit the function
      case 135:Alert("Price changed. Trying once again..");
         RefreshRates();                        // Refresh rates
         return(1);                             // Exit the function
      case 136:Alert("No prices. Waiting for a new tick..");
         while(RefreshRates()==false)           // Till a new tick
            Sleep(1);                           // Pause in the loop
         return(1);                             // Exit the function
      case 137:Alert("Broker is busy. Trying once again..");
         Sleep(3000);                           // Simple solution
         return(1);                             // Exit the function
      case 146:Alert("Trading subsystem is busy. Trying once again..");
         Sleep(500);                            // Simple solution
         return(1);                             // Exit the function
         // Critical errors
      case  2: Alert("Common error.");
         return(0);                             // Exit the function
      case  5: Alert("Old terminal version.");
         Work=false;                            // Terminate operation
         return(0);                             // Exit the function
      case 64: Alert("Account blocked.");
         Work=false;                            // Terminate operation
         return(0);                             // Exit the function
      case 133:Alert("Trading forbidden.");
         return(0);                             // Exit the function
      case 134:Alert("Not enough money to execute operation.");
         return(0);                             // Exit the function
      default: Alert("Error occurred: ",Error); // Other variants
         return(0);                             // Exit the function
     }
  }
//------------------------------------------------------------- 11 --
int New_Stop(int Parametr)                      // Checking stop levels
```

```
    {
     int Min_Dist=MarketInfo(Symb,MODE_STOPLEVEL);// Minimal distance
     if (Parametr>Min_Dist)                      // If less than allowed
       {
        Parametr=Min_Dist;                       // Set allowed
        Alert("Increased distance of stop level.");
       }
     return(Parametr);                           // Returning value
    }
  //-------------------------------------------------------------- 12 --
```

Let us analyze what amendments were made in the source code (tradingexpert.mq4). The main part of the Expert Advisor used as basic has not changed. Changes have been made in two blocks - block 1-2- and block 5-6.

In the block 5-6 trading criteria are calculated. In the described EA a trading strategy is based on two trading criteria - criterion to open Buy and criterion to open Sell. The strategy used in the Expert Advisor allows the presence of only one opened market order, pending orders are not allowed. The strategy also presupposes closing an opposite order when a criterion for opening triggers; for example, if criterion to open a Buy order is relevant, it means that a Sell order must be closed.

For using in the EA shared.mq4 results of calculations performed in the custom indicator rocseparate.mq4, function iCustom() must be executed:

```
     double L_1 = iCustom(NULL,0,"rocseparate",H,P,B,A,1,0);
     double L_5 = iCustom(NULL,0,"rocseparate",H,P,B,A,5,0);
```

In this case formal parameters specified in iCustom() call denote the following:

NULL - calculations in the indicator are performed based on data of the current security; in this case the EA is attached to EURUSD window, so data of EURUSD will be used (see Fig. 131);

0 - in calculations data of the current timeframe are used; in this case the current timeframe is M15, so data corresponding to M15 will be used;

"rocseparate" -name of a custom indicator, in which calculations will be made.

H,P,B,A - list of adjustable parameters. In this case the custom indicator rocseparate.mq4 has adjustable parameters (block 2-3 of rocseparate.mq4 code). For a user to be able to set up values of these parameters from the EA, they are specified in the list of passed parameters of the function iCustom(). In the Expert Advisor values of these parameters can differ from those specified in the indicator. In such a case during calculations in the indicator exactly these passed values will be used. These parameters denote the following:

H - number of bars in calculation history;

P - period of calculation MA;

B - number of bars for rate calculation;

A - number of bars for smoothing.

(the meaning of these parameters is explained in details in the section Custom Indicator ROC (Price Rate of Change).

1 (5) - index line of the indicator. In the custom indicator rocseparate.mq4 6 indicator arrays are used. ROC line in the current timeframe (orange) is constructed on the basis of Line_1[] values, for which buffer with index 1 is used. Smoothed average rate line is based on values of Line_5[] array elements, index of the used buffer is 5.

0 - index of value obtained from an indicator buffer (shift back relative to a current bar by the specified number of periods). In this case values of indicator lines on the zero bar are used, that is why index 0 is specified.

For a user to be able to change the adjustable indicator parameters in the EA manually, external variables are specified in block 1a-1b (of the Expert Advisor). In block 5-5a values of these parameters are assigned to other variables with shorter names - this is done for convenience of code presentation in block 5a-5b. Thus a user can specify in shared.mq4 parameters, with which calculations in the custom indicator rocseparate.mq4 will be conducted. After execution iCustom() function will return value corresponding to a specified element value of specified indicator array calculated in the indicator using specified values of adjustable parameters.

During practical operation it is convenient to see in a security window lines of the indicator, array elements of which are used in the Expert Advisor (see Fig. 131). At the same time execution of iCustom() is not connected with the presence of the indicator in the security window, as well as with the values of its adjustable parameters.

The execution of iCustom() does not require the attachment of a corresponding indicator to a security window. As well as the call of iCustom() from any application program does not result in the attachment of a corresponding indicator to a security window. Attachment of a technical indicator to a security window also does not lead to the call of iCustom in any application program.

Trading criteria in the EA (block 5-6) are calculated on the basis of array element values obtained using iCustom() function. For example a criterion for opening Buy and closing Sell are calculated the following way:

```
      if (L_5<=-Level && L_1>L_5)
        {
         Opn_B = true;                                // Criterion for opening Buy
         Cls_S = true;                                // Criterion for closing Sell
        }
```

If the last known value of a smoothed average rate line (L_5) is less than the specified level (value of the adjustable parameter Level = 0.001) and the last known value of ROC line in the current timeframe (L_1) is larger than the smoothed average rate line (L_5), the criterion for opening a Buy order and closing a Sell order is considered relevant. For the confirmation of relevance of opposite criteria reflecting conditions are used.

Trading criteria accepted in this example are used for educational purpose only and must not be considered as a guideline when trading on a real account.

⇐ Custom Indicator ROC (Price Rate of Change) Standard Functions ⇒

Standard Functions

Al in all there are more than 220 standard functions in MQL4, this is apart from functions of technical indicators. It is impossible to include here descriptions and examples of all functions, because there are too many of them. Some functions that need to be described in details are included into previous sections. In this section we will dwell on other most widely used functions. At the end of each paragraph you will see the full list of functions of a certain category and their short description.

- Common Functions.
  This group includes functions that are not included into any of specialized groups. These are the following functions: Print(), Alert(), Comment(), MarketInfo(), Sleep(), etc.
- Graphical Objects.
  MetaTrader 4 Terminal allows the attachment of numerous graphical objects to a chart. This group includes functions that are used for programmed creation such objects as well as for changing their properties, moving them and deleting.
- Operations with Charts.
  A group of functions that are used for getting different information about a current chart, to which a program in MQL4 (script, indicator or Expert Advisor) is attached.
- String Functions.
  String functions are used for processing variables of string type: searching value, concatenation of lines, retrieval of sub-lines, etc. Conversion functions are used for converting a variable of one type into another type. NormalizeDouble() function rounds off values of double type to a certain accuracy.
- Date and Time.
  This group of functions is used for getting time information in this or that form: LocalTime() shows the local time of a computer, TimeCurrent() shows server time of the last received quote. Besides, parameters like a weekday, month day, hour, minute, etc. can be obtained for an indicated time value.
- File Operations.
  This group of functions is necessary for reading/recording data on a hard disk.
- Arrays and Timeseries.
  They provide access to price data of any available symbol/period.
- Mathematical Functions.
  Standard set of mathematical and trigonometrical functions.
- GlobalVariables.
  Functions for working with GlobalVariables.
- Custom Indicators.
  These functions can be used only when writing custom indicators.
- Account Information.
  Functions show information about a client terminal, account and check the current state of a client terminal (including the state of environment of MQL4-program being executed).
- Trade Functions.
  Functions for conducting trade operations.

For getting a detailed description of any standard function refer to MQL4 Reference at MQL4.community, MetaQuotes Software Corp. website or to "Help" in MetaEditor.

← Program Sharing Common Functions →

Common Functions

One of the most widely used functions is Comment().

Comment() Function

```
void Comment(...)
```

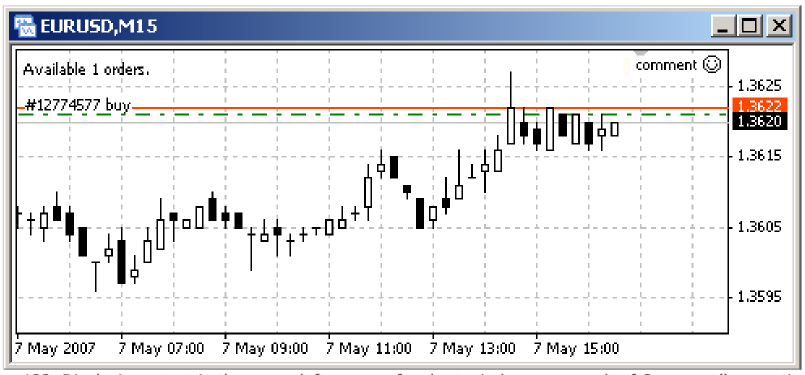
This function introduces a comment defined by a user into the upper left corner of a chart window. Parameters can be of any type. Number of parameters cannot exceed 64. Arrays cannot be passed to the Comment() function. Arrays should be introduced elementwise. Data of double type are written with 4 digits after the decimal point. For showing figures with accuracy use the DoubleToStr() function. Bool, datetime and color types will be written as digits. To show data of datetime type as a string use the TimeToStr() function.

Parameters:
... - any values separated by commas.

Example of using this function can be a simple EA comment.mq4 reflecting the information about the number of orders.

```
//--------------------------------------------------------------------
// comment.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int start()                             // Special function start
  {
   int Orders=OrdersTotal();            // Number of orders
   if (Orders==0)                       // If numb. of ord. = 0
      Comment("No orders");             // Comment to the window corner
   else                                 // If there are orders
      Comment("Available ",Orders," orders." );// Comment
   return;                              // Exit
  }
//--------------------------------------------------------------------
```

At the program beginning the total number of orders is counted by the OrdersTotal() function. If the Orders variable (number of orders) is equal to 0, the Comment() function with "No orders" parameter is executed. If there is at least one order, Comment() with a list of parameters comma separated will be executed. In this case 3 parameters are used: the first one is a string value "Available", second is an integer value Orders and the third one is a string value "orders.". As a result of the function execution at each start of the special function start() one of the messages will be shown in the upper left corner of a chart window. Fig. 132 illustrates a chart window in the situation when there is one order present.



Available 1 orders.

Fig. 132. Displaying a text in the upper left corner of a chart window as a result of Comment() execution.

For the reproduction of sound files PlaySound() function is used.

PlaySound() Function

```
void PlaySound(string filename)
```

The function plays a sound file. The file must be located in ***terminal_directory\sounds*** or its subdirectory.

Parameters:
filename - path to a sound file.

A set of recommended sound files can be found in the attachment - Sound Files.

In some cases a program can be written to support a dialog with a user. Function MessageBox() is used for this purpose.

MessageBox() function

```
int MessageBox(string text=NULL, string caption=NULL, int flags=EMPTY)
```

MessageBox function creates and displays a message box, it is also used to manage the dialog window. A message box contains a message and header defined ina program, as well as any combination of predefined icons and push buttons. If a function is executed successfully, the returned value is one of the return code values of MessageBox(). The function cannot be called from a custom indicator, because indicators are executed in the interface thread and may not decelerate it.

Parameters:
text - a text containing a message to be displayed;
caption - an optional text to be displayed in the message box. If the parameter is empty, an EA name will be displayed in the box header;
flags - optional flags defining the type and behavior of the dialog box. Flags can be a combination of flags from flag groups (see MessageBox Return Codes).

Let us consider an example of MessageBox() usage.

Problem 31. Write a code of an EA displaying a message box with a question of closing all orders 5 minutes prior to important news release. If a user clicks Yes, all orders should be closed, if No is pushed, no actions should be performed.

The EA supporting a dialog with a user (dialogue.mq4) can have the following code:

```
//--------------------------------------------------------------------
// dialogue.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
#include <WinUser32.mqh>                 // Needed to MessageBox
extern double Time_News=15.30;           // Time of important news
bool Question=false;                     // Flag (question is not put yet)
//--------------------------------------------------------------- 2 --
int start()                              // Special function start
  {
   PlaySound("tick.wav");                // At each tick
   double Time_cur=Hour()+ Minute()/100.0;// Current time (double)
   if (OrdersTotal()>0 && Question==false && Time_cur<=Time_News-0.05)
     {                                   // Providing some conditions
      PlaySound("news.wav");             // At each tick
      Question=true;                     // Flag (question is already put)
      int ret=MessageBox("Time of important news release. Close all orders?",
      "Question", MB_YESNO|MB_ICONQUESTION|MB_TOPMOST); // Message box
      //--------------------------------------------------------- 3 --
      if(ret==IDYES)                     // If the answer is Yes
         Close_Orders();                 // Close all orders
     }
   return;                               // Exit
  }
//--------------------------------------------------------------- 4 --
void Close_Orders()                      // Cust. funct. for closing orders
  {
   Alert("Function of closing all orders is being executed.");// For illustration
   return;                               // Exit
  }
//--------------------------------------------------------------- 5 --
```

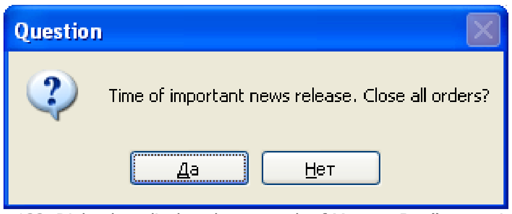
In the block 1-2 WinUser32.mqh file is included into the program; in this file MessageBox() return codes are defined. Also in this block the external variable Time_news is introduced - this is the time of important news release. During the whole period of EA execution a question about closing orders should be displayed only once. To trace whether the question has been already displayed in the EA 'Question' variable is declared.

At each start of the special function start() (block 2-3) PlaySound() is executed. The played tick.wav sound resembles a weak click that denotes the best way the fact of a new tick appearance. The decision about using sound in a program is made by a programmer. In some cases it is very useful to use sounds. For example, a sound may denote the fact of an EA execution. Other sounds may correspond to other events, for example triggering of a trading criterion, order closing, etc.

Value of the actual variable Time_cur corresponds to the current server time. In the EA conditions, at which the message box must be displayed, are analyzed. If there is one or several orders, the message box has not been shown yet and the server time differs from important news release time by less than 5 minutes, certain actions are performed in the program. First of all function PlaySound() is executed, the played sound attracts the attention of a user. The Question flag gets the true value (do not show the next time). In the next line MessageBox() is executed:

```
      int ret=MessageBox("Time of important news release. Close all orders?",
      "Question", MB_YESNO|MB_ICONQUESTION|MB_TOPMOST); // Message box
```

In this case the value of a string constant "Time of important news release. Close all orders?" will be displayed in a dialog box, and "Question" value will be reflected in the upper box line. The MB_YESNO flag denotes the presence of buttons - in this case Yes and No buttons (see MessageBox Return Values). The MB_ICONQUESTION flag defines an icon displayed in the left part of the message box (each operating environment has its own set of icons, Fig. 133 illustrates an icon from Windows XP set). The MB_TOPMOST flag provides the box with the property "always on top", i.e. the box will be always visible irrespective of what programs are being executed at the moment on the computer. As a result of the execution of MessageBox() with indicated parameters a message box is displayed:



Time of important news release. Close all orders?

Fig. 133. Dialog box displayed as a result of MessageBox() execution.

At the moment when the message box is shown the program execution is held up until a user clicks on a button in the message box. As soon as it happens, control will be passed to the line following MessageBox() call, in this case to the block 3-4. This property of a message box to hold control is very important and it must be taken into account at a program development. For example, if a user left his computer and a message box was shown at this moment, during the whole time when a user is absent (until a button is pressed), the program will be waiting for response and no code will be executed in this period.

Note, before a message box is displayed the program execution is accompanied by a sound of ticks. When the message box is displayed another sound is played. In the period when the dialog box is open and waiting for response no sound is played which illustrates the fact of control holding while the dialog box is open. After a button is pressed, the program execution will continue and the sound of ticks will be played again.

If a user clicks Yes, the Close_Orders() function will be called; this function is used for closing orders. In this example the function contents is not described; to denote its execution the Alert function is executed ("Function of closing all orders is being executed."). If a user clicks No, the function of closing orders is not called. In the current session of the EA execution the message box will not be shown again.

Common Functions

Function	Summary Info
Alert	Displays a message box containing user-defined data. Parameters may be of any type. Number of parameters cannot exceed 64.
Comment	Displays a comment defined by a user in the upper loft corner of a chart window. Parameters may be of any type. Number of parameters cannot exceed 64.
GetTickCount	GetTickCount() returns number of milliseconds passed since a system was started. The counter is limited by the resolution of the system timer. Since time is stored as an unsigned integer, it is overfilled every 49.7 days.
MarketInfo	Returns various information about securities listed in "Market watch" window. Part of the information about the current security is stored in predefined variables (see MarketInfo() Identifiers).
MessageBox	MessageBox function creates and displays a message box, it is also used to manage the dialog window. A message box contains a message and header defined ina program, as well as any combination of predefined icons and push buttons. If a function is executed successfully, the returned value is one of the return code values of MessageBox(). The function cannot be called from a custom indicator, because indicators are executed in the interface thread and may not decelerate it.
PlaySound	Plays a sound file. The file must be located in the *terminal_dir|sounds* directory or in its subdirectory.
Print	Prints a message to the experts log. Parameters can be of any type. Amount of passed parameters cannot exceed 64.
SendFTP	Sends a file to the address specified in setting window of "Publisher" tab. If the attempt fails, it retuns FALSE. The function does not operate in the testing mode. This function cannot be called from custom indicators, either. The file to be sent must be stored in the *terminal_directory|experts|files* folder or in its sub-folders. It will not be sent if there is no FTP address and/or access password specified in settings.
SendMail	Sends an email to the address specified in settings window of "Email" tab. The sending can be disabled in settings, or it can be omitted to specify the e-mail address.
Sleep	The Sleep() function suspends execution of the current expert within the specified interval. Sleep() cannot be called from custom indicators, because indicators are executed in the interface thread and may not decelerate it. The checking of the expert stop flag status every 0.1 second is built into the function.

For the detailed description of these and other functions please refer to Documentation at MQL4.community, MetaQuotes Software Corp. website or to "Help" section in MetaEditor.

← Standard Functions Graphical Objects →

Graphical Objects

Graphical object is an image in the symbol window; it can be selected, moved, modified or deleted.

Graphical objects include, for example, horizontal and vertical lines, linear regression channel, Fibonacci levels, rectangle, text mark, etc. Such images as indicator lines, indicator levels, candlesticks, comments written by the Comment() function and others cannot be selected and deleted, that is why they do not belong to graphical objects.

Graphical object are drawn by the client terminal in a security window in accordance with preset coordinates. Each graphical object depending on its type has one, two or three coordinates and other adjustable parameters. Any graphical object can be placed in a chart window manually (from the toolbar of a system menu), and also as a result of the execution of an application program started in the same window, including an Expert Advisor, script or custom indicator. Type and location of a graphical object can be modified manually or by a program sending new values of coordinates and other parameters to a graphical object.

Ways of Positioning Graphical Objects

There are two ways of positioning objects accepted in MQL4: relative to a chart and relative to a security window. To illustrate the difference between these methods, let us place manually two objects in a security window: text (OBJ_TEXT) and a text mark (OBJ_LABEL). We can use **A** and **T** buttons from the toolbar of the client terminal. Let us set the window size so that it is equal to half of screen size (Fig. 134). Let us see how these graphical objects will react to the window size changes (as well as to the horizontal and vertical scaling of the price chart).

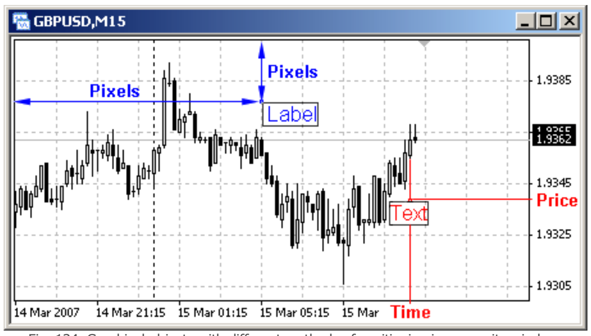
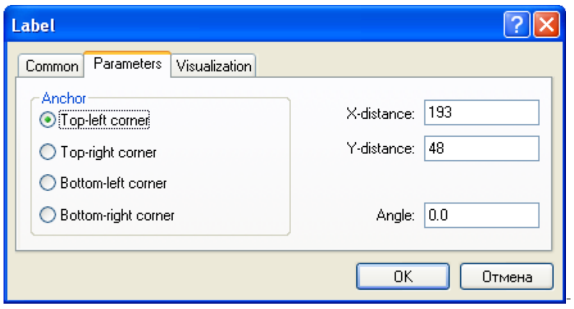


Fig. 134. Graphical objects with different methods of positioning in a security window.

Positioning Relative to a Chart Window

The graphical object OBJ_LABEL will remain immovable if a window size is changed by way of shifting its right or lower borders. But if the window size is changed by shifting its upper or lower border, the object will be also shifted, though the position of the object relative to these borders will remain unchanged. This happens because OBJ_LABEL is positioned relative to security window borders. In this case the reference point of the graphical object to a security window is the upper left corner of a chart6 window. Coordinates of the object relative to the indicated point are set in pixels - 193 and 48 (Fig. 135).


Label

Common | Parameters | Visualization

Anchor
[✓ Top-left corner]
Top-right corner
Bottom-left corner
Bottom-right corner

X-distance: 193
Y-distance: 48
Angle: 0.0

OK | Отмена

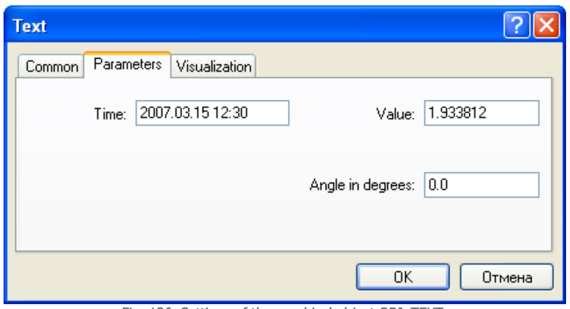
Fig. 135. Settings of the graphical object OBJ_LABEL.

The reference point of the object coordinates (in this case) is the upper left corner of a cursor frame visible when selected by a mouse. In the upper left corner of the cursor frame you can see a small point indicating the settings of this graphical object. If another reference point is indicated, the point in the cursor frame will be indicated in another corner.

When new bars appear in a chart window, an object like OBJ_LABEL will remain immovable in the window. Using of this object is convenient if it is necessary to display text information of general character, for example, information about termination of trading, value of a limiting distance changed by a broker, etc.

Positioning Relative to a Chart

At any method of windows size changing, as well as at chart scaling, an object of OBJ_TEXT type does not change its position relative to a chart. The reference point of such an object is the middle of the upper line of a cursor frame, its X coordinate is time, Y coordinate is a security price (Fig. 136).


Text

Common | Parameters | Visualization

Time: 2007.03.15 12:30
Value: 1.933812
Angle in degrees: 0.0

OK | Отмена

Fig. 136. Settings of the graphical object OBJ_TEXT.

As new bars appear in a chart window, the position of OBJ_TEXT does not change relative to a chart, i.e. with the appearance of new bars the object will be shifted to the left together with the chart; and when there will be enough bars, the object will move further to the left out of the window borders.

This or that method of positioning the own property of a certain object type and cannot be changed by a user, even in a program way. The majority of graphical objects is positioned relative to a chart, i.e. in time and price coordinates.

Creating Graphical Objects and Changing Their Properties

To create a graphical object means to place in a chart window one of objects of predefined types (see Types and Properties of Graphical Objects). For object creation the following function is used:

ObjectCreate() Function

```
bool ObjectCreate(string name, int type, int window, datetime time1, double price1, datetime time2=0,
double price2=0, datetime time3=0, double price3=0)
```

The function creates an object of an indicated type with a preset name and coordinates in the indicated chart subwindow. Number of the object coordinates can be from 1 to 3 depending on the object type. If an object is successfully created, the function returns TRUE, otherwise FALSE. To get additional information about an error call the GetLastError() function.

Coordinates must be passed in pairs - time and price. For example OBJ_VLINE needs only time, but price should also be passed (any value). Graphical object of OBJ_LABEL type ignores coordinates specified in the function; to set OBJPROP_XDISTANCE and OBJPROP_YDISTANCE of this object the ObjectSet() function must be used.

Parameters:

- **name** - object name;
- **type** - object type (can be one of predefined object types);
- **window** - number of window into which an object will be added. Numeration of chart subwindows (if there are subwindows with indicators present) starts from 1, the number of the main window is always 0; the indicated widow number must be larger than or equal to 0 and less than the value returned by the WindowsTotal() function;
- **time1** - time of the first coordinate;
- **price1** - price of the first coordinate;
- **time2** - time of the second coordinate;
- **price2** - price of the second coordinate;
- **time3** - time of the third coordinate;
- **price3** - price of the third coordinate.

Each graphical object has some (peculiar to it) adjustable parameters. For example, besides defined coordinates, you can specify color, message text (for some objects), line styles (for other objects), etc. For changing properties use the following function:

ObjectSet() Function

```
bool ObjectSet(string name, int prop_id, double value)
```

The function changes the value of the indicated object property. In case of success the function returns TRUE, otherwise FALSE. To get the error information call the GetLastError() function.

Parameters:

- **name** - object name;
- **prop_id** - object properties identifier (one of object properties is indicated);
- **value** - new value of the indicated property.

All graphical objects may have a text description. The text description of each object is available to a user and can be changed from an object properties toolbar or in a programmed way. For OBJ_TEXT and OBJ_LABEL this description is their main contents and and is always displayed as a text line, text descriptions of other objects are displayed near the object if the option "Show object descriptions" is enabled in a symbol properties window (F8). To change the text description the following function is used:

ObjectSetText() Function

```
bool ObjectSetText(string name, string text, int font_size, string font_name=NULL, color text_color=CLR_NONE)
```

The function is used for changing an object description. In case of success TRUE is returned, otherwise - FALSE. To get the error information call the GetLastError() function. Parameters font_size, font_name and text_color are used only for OBJ_TEXT and OBJ_LABEL. For objects of other types these parameters are ignored.

Parameters:

- **name** - object name;
- **text** - object description text;
- **font_size** - font size in points;
- **font_name** - font name;
- **text_color** - text color.

Lets us analyze an example of an Expert Advisor,in which functions of managing graphical objects are used.

Problem 32. Using a graphical object inform a user about trading criteria defined on the basis of MACD values.

MACD is very often used by traders for the formation of trading criteria. The indicator is represented by two lines - main and signal. A trading criteria is considered to be realized when the lines cross. If the main indicator line (usually gray histogram) crosses the signal line (usually red dotted line) downwards, this is a signal to sell, id upwards - to buy. In intervals between line crossing market orders should be held open, and when a contrary criterion triggers, the orders should be closed and opposite once opened. Thus four message types should be prepared: opening of Buy, opening of Sell, holding of Buy, holding of Sell.

In this problem all messages are mutually exclusive, i.e. the situation when two or more messages should be shown is impossible. That is why in this case one graphical object can be used; the object will be always present on th screen, but it will be changed from time to time. Let us draw this object in the upper right corner of the window, in which the EA will operate. Since the object position should not be changed, it is convenient to use an object of OBJ_LABEL type, because it is positioned relative to a chart window.

As a solution of Problem 32 let us view the EA grafobjects.mq4 using the graphical object OBJ_LABEL:

```
//--------------------------------------------------------------------
// grafobjects.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
int start()                             // Special function start
  {
//--------------------------------------------------------------- 1 --
   int Sit;
   double MACD_M_0,MACD_M_1,            // Main line, 0 and 1st bar
   MACD_S_0,MACD_S_1;                   // Signal line, 0 and 1st bar
   string Text[4];                     // Declaring a string array
   color  Color[4];                    // Declaring an array of colors

   Text[0]= "Opening of Buy";          // Text for different situations
   Text[1]= "Opening of Sell";
   Text[2]= "Holding of Buy";
   Text[3]= "Holding of Sell";

   Color[0]= DeepSkyBlue;              // Object color ..
   Color[1]= LightPink;                // .. for different situations
   Color[2]= Yellow;
   Color[3]= Yellow;
//--------------------------------------------------------------- 2 --
   ObjectCreate("Label_Obj_MACD", OBJ_LABEL, 0, 0, 0);// Creating obj.
   ObjectSet("Label_Obj_MACD", OBJPROP_CORNER, 1);    // Reference corner
   ObjectSet("Label_Obj_MACD", OBJPROP_XDISTANCE, 10);// X coordinate
   ObjectSet("Label_Obj_MACD", OBJPROP_YDISTANCE, 15);// Y coordinate
//--------------------------------------------------------------- 3 --
   MACD_M_0 =iMACD(NULL,0,12,26,9,PRICE_CLOSE,MODE_MAIN,0);  // 0 bar
   MACD_S_0 =iMACD(NULL,0,12,26,9,PRICE_CLOSE,MODE_SIGNAL,0);// 0 bar
   MACD_M_1 =iMACD(NULL,0,12,26,9,PRICE_CLOSE,MODE_MAIN,1);  // 1 bar
   MACD_S_1 =iMACD(NULL,0,12,26,9,PRICE_CLOSE,MODE_SIGNAL,1);// 1 bar
//--------------------------------------------------------------- 4 --
   // Analyzing situation
   if(MACD_M_1=MACD_S_0)                      // Crossing upwards
      Sit=0;
   if(MACD_M_1>MACD_S_1 && MACD_M_0<=MACD_S_0)// Crossing downwards
      Sit=1;
   if(MACD_M_1>MACD_S_1 && MACD_M_0>MACD_S_0) // Main above signal
      Sit=2;
   if(MACD_M_1
```

In the EA block 1-2 parameters are defined, in particular element values of Text[] and Color[] are set. Further they are used for changing object properties. In the block 2-3 the object is created and values of some its properties are set. Let us analyze this block in details. According to this EA code line a graphical object is created in the window, in which the EA is executed:

```
   ObjectCreate("Label_Obj_MACD", OBJ_LABEL, 0, 0, 0);// Creating obj.
```

"Label_Obj_MACD" value denotes that this name is assigned to the object (a name is assigned to an object by a programmer in his own discretion). OBJ_LABEL - is the object type identifier; it denotes that the created object will be of exactly this type (chosen from the list of possible types). The first of the next three zeros denotes that the object is created in the main window (the main window where the chart is displayed, always has the index 0).

The next two zeros set coordinates for the created object. According to this coordinated the object will be drawn in the indicated window. In this case the created OBJ_LABEL does not use time and price coordinates. Please note that in OjectCreate() description only time and price coordinates are specified. Moreover, coordinates of the second and the third pairs have default values, while there are no default values for the first pair of coordinates. It means that though OBJ_LABEL does not need time and price coordinates at all, some values must be specified in ObjectCreate() function call. In this case zeros are indicated, though any other values can be written - anyway these values will be neglected during the setup of OBJ_LABEL properties.

In the next three lines some property values are set to the earlier created object named Label_Obj_MACD:

```
   ObjectSet("Label_Obj_MACD", OBJPROP_CORNER, 1);    // Reference corner
   ObjectSet("Label_Obj_MACD", OBJPROP_XDISTANCE, 10);// X coordinate
   ObjectSet("Label_Obj_MACD", OBJPROP_YDISTANCE, 15);// Y coordinate
```

For the reference corner (OBJPROP_CORNER) 1 is set, which means the upper right corner of the earlier defined main window. In the next two lines distances from the object to a reference corner are set on pixels: horizontal distance (OBJPROP_XDISTANCE) 10 pixels and vertical distance (OBJPROP_YDISTANCE) 15 pixels. At this program execution stage the object is already created, has its unique name and defined main properties.

To make the object show a necessary text, first we need to calculate what this text should look like. For this purpose first in block 3-4 the position of MACD lines is detected on the current and previous bars, then in block 4-5 Sit value corresponding to the current situation is calculated (see also Fig. 107 and callstohastic.mq4)

In the next line object properties depending on the current situation are defined:

```
                                       // Changing object properties
   ObjectSetText("Label_Obj_MACD",Text[Sit],10,"Arial",Color[Sit]);
```

As a result of ObjectSetText() execution a text description is assigned to the object named Label_Obj_MACD - the value of the string variable Text[Sit]. This value will be different for different situations depending on values of Sit variable. For example, if the main line crosses the signal one downwards, in block 4-5 Sit gets the value 1, as a result the graphical object will get the text description contained in the Text[1] array element, i.e. "Opening of Sell". Other parameters: 10, "Arial" and Color[Sit] denote font size, name and color for the text description.

As a result of the EA execution the following will appear in EURUSD window:

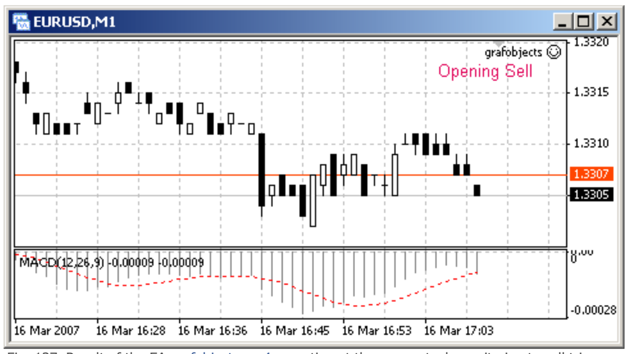


Fig. 137. Result of the EA grafobjects.mq4 operation at the moment when criterion to sell triggers.

In Fig. 137 there is a main window and MACD subwindow. It should be noted here that for a normal EA operation presence of this indicator on the symbol window is not necessary, because trading criteria in the EA are calculated as a result of a technical indicator function execution which is not connected with the indicator displaying. Here the indicator is shown only for visual explanation of the moment of a trading criterion triggering when the necessary text description of the graphical object is shown. The EA will operate in the similar way at all other combinations of the mutual position of indicator lines each time showing a description corresponding to a situation.

Deleting Graphical Objects

The analyzed Expert Advisor grafobjects.mq4 has a small disadvantage. After the EA stops operating, a graphical object will remain in the chart window (his properties will remain the same as at t he moment of its last change). Graphical objects are not deleted automatically. In course of trading starting from a certain moment the message "Opening of Sell" will not be valid. In order not to misinform a user the graphical object must be deleted.

For deleting a graphical object (irrespective of its creation method - programmed or manual) simply select it and press the Delete key. However, as for programming, it should be noted that a correctly written program must "clear" the window when its operation is over. In other words, a program should contain a block where all graphical objects created by the program are deleted.

ObjectDelete() Function

```
bool ObjectDelete(string name)
```

deleting an object with the indicated name. If an object is successfully deleted, the function returns TRUE, otherwise - FALSE. To get the error information call the GetLastError () function..

Parameters:

- **name** - name of a deleted object.

It is very easy to use ObjectDelete(): simply indicate the name of an object to delete.

To fix the disadvantage of the previous example, let us add into the EA grafobjects.mq4 the special function deinit() containing the function for deleting objects:

```
//--------------------------------------------------------------- 7 --
int deinit()                               // Special function deinit
  {
   ObjectDelete("Label_Obj_MACD");      // Object deletion
   return;                                 // Exit deinit()
  }
//--------------------------------------------------------------- 8 --
```

Now, during the EA execution the object named Label_Obj_MACD will be deleted. Generally a program can create numerous objects. Each of them can be deleted according to the algorithm.

Modifying Graphical Objects

In some cases it is necessary to change an object position in a chart window in a program way. Very often such a necessity may occur because of the appearance of new bars. For example, trading criteria in an EA can be formed on the basis of a linear regression channel built on a bar history of a certain length (for example, last 50 bars). If we simply draw the object "linear regression channel" in a chart window and then do not undertake anything, it will remain on the same chart place where it was positioned and it will be shifted to the left as new bars appear. To prevent the object from shifting it should be redrawn at each new bar. For this purpose new coordinates must be calculated and passed to the object; in accordance with these coordinates the object will be drawn in a chart widow.

For finding out what properties a graphical object has at the current moment, the following function should be used:

ObjectGet() Function

```
double ObjectGet(string name, int prop_id)
```

the function returns the value of the specified object property. To get the error information call the GetLastError() function.

parameters:

- **name** - object name;
- **prop_id** - object property identifier. Can be any value from the list of object properties.

New coordinates are reported to an object using the ObjectMove() function.

ObjectMove() Function

```
bool ObjectMove(string name, int point, datetime time1, double price1)
```

Changing one of coordinates on a chart. The function returns TRUE in case of success, otherwise - FALSE. To get additional information call the FetLast Error() function. Numeration of an object coordinates starts from 0.

Parameters:

- **name** - object name;
- **point** - coordinate index (0-2);
- **time1** - new time value;
- **price1** - new price value.

Problem 33. Create a program (an Expert Advisor) supporting the drawing of a linear regression channel for the last 50 bars.

The graphical object "linear regression channel" uses two time coordinates. Price coordinates (if such are specified in the program) are neglected by the client terminal during the object construction. The linear regression channel is calculated by the client terminal based on historic price data and therefore cannot be displayed aside from a chart. That is why the absence of the object binding to price (ignoring of price coordinates by the terminal) is the object's own constant property. Th Expert Advisor (moveobjects.mq4) managing the position of a graphical object can have the following code:

```
//--------------------------------------------------------------------
// moveobjects.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
extern int    Len_Cn=50;                   // Channel length (bars)
extern color Col_Cn=Orange;                // Channel color
//--------------------------------------------------------------- 1 --
int init()                                 // Special function init()
```

```
     {
      Create();                             // Calling user-def. func. of creation
      return;                               // Exit init()
     }
//--------------------------------------------------------------- 2 --
int start()                                 // Special function start()
  {
   datetime T2;                             // Second time coordinate
   int Error;                               // Error code
//--------------------------------------------------------------- 3 --
   T2=ObjectGet("Obj_Reg_Ch",OBJPROP_TIME2);// Requesting t2 coord.
   Error=GetLastError();                    // Getting an error code
   if (Error==4202)                         // If no object :(
     {
      Alert("Regression channel is being managed",
            "\n Book_expert_82_2. deletion prohibited.");
      Create();                             // Calling user-def. func. of creation
      T2=Time[0];                           // Current value of t2 coordinate
     }
//--------------------------------------------------------------- 4 --
   if (T2!=Time[0])                         // If object is not in its place
     {
      ObjectMove("Obj_Reg_Ch", 0, Time[Len_Cn-1],0); //New t1 coord.
      ObjectMove("Obj_Reg_Ch", 1, Time[0],       0); //New t2 coord.
      WindowRedraw();                       // Redrawing the image
     }
   return;                                  // Exit start()
  }
//--------------------------------------------------------------- 5 --
int deinit()                                // Special function deinit()
  {
   ObjectDelete("Obj_Reg_Ch");              // Deleting the object
   return;                                  // Exit deinit()
  }
//--------------------------------------------------------------- 6 --
int Create()                                // User-defined function..
  {                                         // ..of object creation
   datetime T1=Time[Len_Cn-1];              // Defining 1st time coord.
   datetime T2=Time[0];                     // Defining 2nd time coord.
   ObjectCreate("Obj_Reg_Ch",OBJ_REGRESSION,0,T1,0,T2,0);// Creation
   ObjectSet(   "Obj_Reg_Ch", OBJPROP_COLOR, Col_Cn);    // Color
   ObjectSet(   "Obj_Reg_Ch", OBJPROP_RAY,   false);     // Ray
   ObjectSet(   "Obj_Reg_Ch", OBJPROP_STYLE, STYLE_DASH);// Style
   ObjectSetText("Obj_Reg_Ch","Created by the EA moveobjects",10);
   WindowRedraw();                          // Image redrawing
  }
//--------------------------------------------------------------- 7 --
```

The moveobjects.mq4 EA algorithm implies that an object attached once will remain on the screen during the whole time of the program execution. In such cases it is reasonable to use a user defined function (in this case it is Create(), block 6-7) for an object creation, the function can му called from the program anytime when needed. To draw an object two time coordinates are necessary (T1 is the coordinate of the object's left border, T2 - that of the right border):

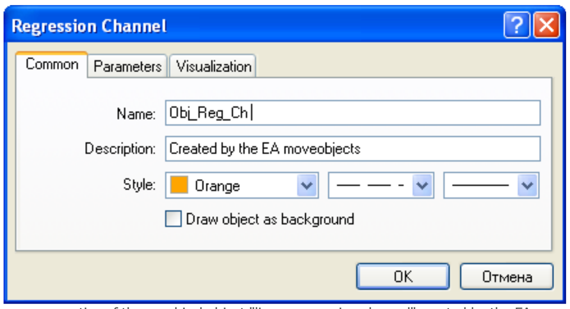
```
   datetime T1 = Time[Len_Cn-1];            // Defining 1st time coord.
   datetime T2 = Time[0];                   // Defining 2nd time coord.
```

In this example the right border of the object must always be on the zero bar, that is why the value of the second coordinate corresponds to the opening time of the zero bar. The left coordinate is calculated according to the number of bars set by a user (external variable Len_Cn) and is defined as the opening time of a bar with the corresponding index. For example, if the channel length is 50 bars, the left coordinate will be equal to the opening time of a bar with the index 49.

In the next lines of the user-defined function Create() the OBJ_REGRESSION object is created using ObjectCreate(), then necessary properties of the created object are set up by the ObjectSet() function (color preset by a user in an external variable, prohibited to draw as a ray, line style - dotted). In the line:

```
   ObjectSetText("Obj_Reg_Ch","Created by the EA moveobjects",10);
```

a text description is assigned to the object. As distinct from the earlier analyzed OBJ_LABEL, the text description of OBJ_REGRESSION is not displayed. The text description of graphical objects can be viewed in the object properties tab. This is very convenient in practical application for differentiating between objects created in a program way from those attached manually:



Name: Obj_Reg_Ch

Description: Created by the EA moveobjects

Style: Orange

☐ Draw object as background

Fig. 138. Common properties of the graphical object "linear regression channel" created by the EA moveobjects.mq4.

Here is one more function used for redrawing of the current chart:

```
   WindowRedraw();                          // Image redrawing
```

WindowRedraw() Function

```
void WindowRedraw()
```

The function forcibly redraws the current chart. Usually it is used after object properties are changed.

Normally, the graphical objects are displayed by the client terminal in the sequence of incoming of new ticks. This is why, if we don't use WindowRedraw(), the changes in the object properties become visible to the user at the next tick, i.e., the displaying is always one tick late. The use of WindowRedraw() allows you to forcibly redraw all objects at a necessary moment, for example, immediately after the object properties have been changed. In a general case, if the properties of several objects are changed in the program, it is sufficient to use the function WindowRedraw() only once, after the properties of the last of the objects have been changed.

The user-defined function is first called from the special function init(). At the moment of attaching the EA to the symbol window, the execution of init() will start, which results in that the graphical object Linear Regression Channel will be displayed in the symbol window.

Two possible situations are considered in the function start(): (1) the object has been occasionally deleted by the user (block 3-4) and (2) it is necessary to move the object to the right when a new zero bar is formed (block 4-5). To detect whether the graphical object is available at the current moment, it is sufficient just to request the value of one of its coordinates. If the object exists, the function ObjectGet() will return a certain value that corresponds with the requested coordinate and the function GetLastError() will return zero value (i.e., no error occurred when requesting the coordinate). However, if there is no object of the given name in the symbol window, the function GetLastError() will return the code of error 4202, i.e., no object available:

```
      T2=ObjectGet("Obj_Reg_Ch",OBJPROP_TIME2);      // Requesting t2 coord.
      Error=GetLastError();                          // Getting an error code
```

If the error analysis showed that there were no object of that name, it means the program must create it, having notified the user about inadmissible actions (the program doesn't delete objects, it means that the object has been deleted by the user). This is why, after having displayed the message, the program calls to the previously considered user-defined function Create(), which results in a new creation of the object in the symbol window.

By the moment of the execution of the next block (4-5), the graphical object has already been created. To decide whether it must be moved, you should know the position of the object at the current moment. For this purpose, it is sufficient to analyze the previously obtained value of the first coordinate of the object. If this value doesn't coincide with the time of opening zero bar, to assign new coordinates to the object.

The coordinates are changed using the function ObjectMove():

```
      ObjectMove("Obj_Reg_Ch", 0, Time[Len_Cn-1],0); //New t1 coord.
      ObjectMove("Obj_Reg_Ch", 1, Time[0],       0); //New t1 coord.
```

Here, for the first coordinate (coordinate 0) of the object named Obj_Reg_Ch, the value of Time[Len_Cn-1] will be set, whereas for the second coordinate (coordinate 1) -Time [0]. The last parameters among those transferred to the function ObjectMove() is specified the parameter 0. This is the coordinate of the price that, according to the function description, must be transferred, but, in this case, will be ignored by the client terminal. As a result of execution of these lines, the properties of the considered graphical object will be changed. As a result of the next execution of the function WindowRedraw(), the graphical object will be forcibly redrawn by the client terminal - now according to the new values of the coordinates.

Thus, at the execution of the function start(), the graphical object Linear Regression Channel will be redrawn by the client terminal each time when a new bar forms, at its very first tick (see Fig. 139). After the execution of the EA has ended, the given graphical object will be deleted from the symbol window during execution of the special function deinit() (i.e., the program will "sweep out" its working place after the work has been finished).

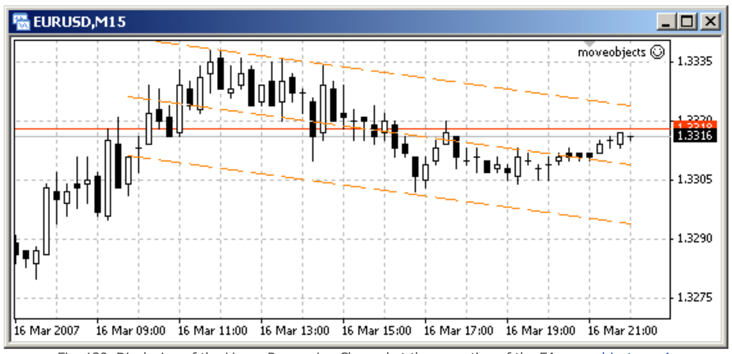


Fig. 139. Displaying of the Linear Regression Channel at the execution of the EA moveobjects.mq4.

In a general case, you can create and delete graphical objects according to some conditions calculated in the program. You can display the support/resistance lines (OBJ_TREND), mark the time of approaching important events with vertical lines (OBJ_VLINE), indicate the intersections of various lines or the forecast price movements using text objects (OBJ_LABEL and OBJ_TEXT), etc.

It must be noted separately that, in a number of cases, there is no need to use graphical objects. For example, if you want to display in the screen a great variety of simple one-type images (for example, arrows), you can use indicator lines for this, having set their styles in the corresponding way. This approach will free you from the necessity to track many coordinates of objects in the program, it will also prevent you from occasional deletion of an image (the signs that display indicator lines can be neither selected nor deleted).

Functions for Working with Graphical Objects

Function	Summary Info
ObjectCreate	Creating an object with predefined name, type and initial coordinates in the indicated chart subwindow. number of object coordinates can be from 1 to 3 depending on the object type. In case of success the function returns TRUE, otherwise FALSE.
ObjectDelete	Deleting an object with the indicated name. In case of success the function returns TRUE, otherwise FALSE.
ObjectDescription	The function returns the object description. It returns for objects of the OBJ_TEXT and OBJ_LABEL types the text displayed in these objects.
ObjectFind	The function searches for the object of the given name. The function returns the index of the window, to which the searched object belongs. In case of failure, the function returns -1.
ObjectGet	The function returns the value of the given property of the object.
ObjectGetFiboDescription	The function returns the description of the Fibo object level. The amount of levels depends on the type of the object that belongs to the group of Fibo objects. The maximum amount of levels is 32.
ObjectGetShiftByValue	The functions calculates and returns the bar number (the shift relative to the current bar) for the given price. The bar number is calculated using a linear equation for the first and second coordinates. It is used for trend lines and similar objects.
ObjectGetValueByShift	The functions calculates and returns the price value for the given bar (the shift relative to the current bar). The price value is calculated using a linear equation for the first and second coordinates. It is used for trend lines and similar objects.
ObjectMove	Changing one of object coordinates on a chart. Objects can have from one to three anchoring points according to the object type. In case of success, the function returns TRUE, otherwise FALSE.
ObjectName	The function returns the object name according to its order number in the list of objects.
ObjectsDeleteAll	Deleting all object of the indicated type in the indicated chart subwindow. The function returns the number of deleted objects.
ObjectSet	Changing properties of an indicated object. In case of success the function returns TRUE, otherwise FALSE.
ObjectSetFiboDescription	The function assigns a new value to Fibonacci level. Number of levels depends on Fibonacci object type. Maximal number of levels is 32.

ObjectSetText	Changing object description. For objects OBJ_TEXT and OBJ_LABEL this description is displayed on a chart as a text line. In case of success the function returns TRUE, otherwise FALSE.
ObjectsTotal	Returns the total number of objects of the indicated type on a chart.
ObjectType	The function returns the type of an indicated object.

For the detailed description of these and other functions, please refer to Documentation at MQL4.community, MetaQuotes Software Corp. website or to "Help" section in MetaEditor.

← Common Functions Operations with Charts →

Operations with Charts

In his or her practical work, a trader usually opens in a symbol window several subwindows that display indicators. There are no limitations on placing indicators, they can be attached in any sequence. The amount of subwindos in a symbol window is not limited either. Each subwindow has its number. The main window containing a price chart is always available, its number being 0. Each indicator subwindow has a number, as well. The subwindows are numbered in a simple sequence - they are numbered by their displaying in the symbol window from top to bottom: the indicator subwindow closest to the main window has number 1, the next lower one has number 2, the next one has number 3, etc.

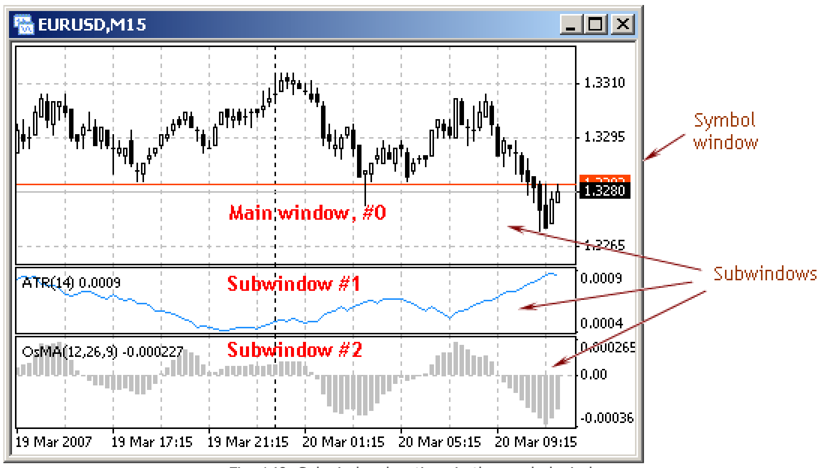


Fig. 140. Subwindow locations in the symbol window.

The amount of subwindows can be easily calculated using the following function:

```
int WindowsTotal()
```

The function returns the amount of indicator subwindows located in the chart, including the main chart window. The largest number (of the lowest subwindow) is always 1 less than the total amount of subwindows (including the main window numbered as 0) in the symbol window. If, in the situation shown in Fig. 140, we call for execution the function WindowsTotal() from any application, the returned value will be equal to 3, while the largest number (of the lowest subwindow) is 2.

The numbering sequence described above is kept, if a new indicator subwindow is added to or an existing subwindow is deleted from the symbol window. If you add a new subwindow, it will be displayed below all other subwindows and its number is 1 more than that of the last window above it. If you delete a subwindow from the symbol window, all subwindows below it will be automatically renumbered - the number of each of them will be decreased by 1.

In MQL4, it is possible to create graphical objects (and change their properties) in any of the existing subwindows. For this purpose, in the function ObjectCreate() the parameter 'window' is provided, according to which an object is created in the given subwindow of the symbol window. The current number of the subwindow can be calculated using the following function:

```
int WindowFind(string name)
```

The function returns the number of the chart subwindow that contains the indicator named as 'name', if it has been found. Otherwise, it returns -1. The function will also return -1, if a custom indicator searches for itself during initialization init().

Parameters:

name - short name of the indicator.

The amount of subwindows in a symbol window can change at any moment, if the user deletes an indicator. This is why the algorithm of an application that supports monitoring of graphical objects must continuously track the numbers of windows, in which the indicators are displayed.

Problem 34. Using graphical objects, display the messages informing about data received from two indicators. If the corresponding indicator is attached to the symbol window, display the graphical object in the indicator window. Otherwise, display it in the main window.

To solve the problem, let's choose indicators RSI and Momentum. The general algorithm of building an Expert Advisor comes down to this. In the function init(), you can specify texts to be displayed on the screen according to the indicator readings, i.e., make the calculations to be executed only once in the program. In the function start(), you should calculate the indicator readings, detect the availability of necessary subwindows and their numbers, and then, according to the situation, display one or another message in one or another subwindow. At the execution of the function deinit (), it is necessary to delete all graphical objects created during the work of your program. Below is the EA named charts.mq4 that controls graphical objects in the subwindows of a symbol window.

```
//--------------------------------------------------------------------
// charts.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
int    Win_Mom_old=0,                   // Old number of subwindow Moment.
       Win_RSI_old=0;                   // Old number of subwindow RSI
color Color[5];                         // Declaration of the color array
string Text[5];                         // Declaration of the string array
//--------------------------------------------------------------- 2 --
int init()                              // Special function init()
  {
   Win_RSI_old=0;                               // Technical moment
   Win_Mom_old=0;                               // Technical moment

   Text[0]= "RSI(14) is below 30. Buy";         // Texts for situations RSI
   Text[1]= "RSI(14) is above 70. Sell";        // Texts for situations RSI
   Text[2]= "RSI(14) is between 30 and 70";     // Texts for situations RSI
   Text[3]= "Momentum(14) is growing";          // Texts for situations Momentum
   Text[4]= "Momentum(14) is sinking";          // Texts for situations Momentum
   Color[0]= DeepSkyBlue;                       // Object color for ..
   Color[1]= LightPink;                         // .. different situations ..
   Color[2]= Orange;                            // .. of the indicator RSI
   Color[3]= Color[0];                          // The same colors for Momentum
   Color[4]= Color[1];                          // The same colors for Momentum

   Create_RSI(0);                               // Creation of the first object
   Create_Mom(0);                               // Creation of the second object
   Main();                                      // Call to user-defined function
   return;                                      // Exit init()
  }
//--------------------------------------------------------------------------- 3 --
int start()                                     // Special function 'start'
  {
   Main();                                      // Call to the user-defined function
   return;                                      // Exit start()
  }
//--------------------------------------------------------------------------- 4 --
int deinit()                                    // Special function deinit()
  {
   ObjectDelete("Obj_RSI");                     // Deletion of the object
   ObjectDelete("Obj_Mom");                     // Deletion of the object
   return;                                      // Exit deinit()
  }
//--------------------------------------------------------------------------- 5 --
int Main()                                      // User-defined function
  {
   int                                          // Integer variables
   Win_RSI_new=0,                               // New number of the subwindow RSI
   Win_Mom_new=0,                               // New number of the subwindow Moment.
   Ind_RSI, Ind_Mom;                            // Indexes for situations
   double                                       // Real variables
   RSI,                                         // Value of RSI on bar 0
   Mom_0, Mom_1;                                // Value of Mom. on bars 0 and 1
//--------------------------------------------------------------------------- 6 --
   RSI=iRSI(NULL,0,14,PRICE_CLOSE,0);           // RSI(14) on zero bar
   Ind_RSI=2;                                   // RSI between levels 30 and 70
   if(RSI < 30)Ind_RSI=0;                       // RSI at the bottom. To buy
   if(RSI > 70)Ind_RSI=1;                       // RSI on the top. To sell
//--------------------------------------------------------------------------- 7 --
   Win_RSI_new=WindowFind("RSI(14)");           // Window number of indicator RSI
   if(Win_RSI_new==-1) Win_RSI_new=0;           // If there is no ind., then the main window
   if(Win_RSI_new!=Win_RSI_old)                 // Deleted or placed ..
     {                                          // .. window of indicator RSI
      ObjectDelete("Obj_RSI");                  // Deletion of the object
      Create_RSI(Win_RSI_new);                  // Create an object in the desired window
      Win_RSI_old=Win_RSI_new;                  // Remember this window
     }                                          // Change the textual description:
   ObjectSetText("Obj_RSI",Text[Ind_RSI],10,"Arial",Color[Ind_RSI]);
//--------------------------------------------------------------------------- 8 --
   Mom_0=iMomentum(NULL,0,14,PRICE_CLOSE,0);    // Value on zero bar
   Mom_1=iMomentum(NULL,0,14,PRICE_CLOSE,1);    // Value on the preceding bar
   if(Mom_0 >=Mom_1)Ind_Mom=3;                  // Indicator line goes up
   if(Mom_0 < Mom_1)Ind_Mom=4;                  // Indicator line goes down
//--------------------------------------------------------------------------- 9 --
   Win_Mom_new=WindowFind("Momentum(14)");      // Window number of indicator Momen
   if(Win_Mom_new==-1) Win_Mom_new=0;           // If there is no ind., then the main window
   if(Win_Mom_new!=Win_Mom_old)                 // Deleted or placed ..
     {                                          // .. the window of Momentum indicator
      ObjectDelete("Obj_Mom");                  // Deletion of the object
      Create_Mom(Win_Mom_new);                  // Create an object in the desired window
      Win_Mom_old=Win_Mom_new;                  // Remember this window
     }                                          // Change the textual description:
   ObjectSetText("Obj_Mom",Text[Ind_Mom],10,"Arial",Color[Ind_Mom]);
//-------------------------------------------------------------------------- 10 --
   WindowRedraw();                              // Redrawing the image
   return;                                      // Exit the user-defined function
  }
//-------------------------------------------------------------------------- 11 --
int Create_RSI(int Win)                         // User-defined function
  {                                             // ..of creation of an object
   ObjectCreate("Obj_RSI",OBJ_LABEL, Win, 0,0); // Creation of an object
   ObjectSet("Obj_RSI", OBJPROP_CORNER, 0);     // Anchoring to an angle
   ObjectSet("Obj_RSI", OBJPROP_XDISTANCE, 3);  // Coordinate X
   if (Win==0)
      ObjectSet("Obj_RSI",OBJPROP_YDISTANCE,20);// Coordinate Y
   else
      ObjectSet("Obj_RSI",OBJPROP_YDISTANCE,15);// Coordinate Y
   return;                                      // Exit the user-defined function
  }
```

```
//--------------------------------------------------------------------------- 12 --
int Create_Mom(int Win)                          // User-defined function
  {                                              // ..creating an object
   ObjectCreate("Obj_Mom",OBJ_LABEL, Win, 0,0); // Creation of an object
   ObjectSet("Obj_Mom", OBJPROP_CORNER, 0);      // Anchoring to an angle
   ObjectSet("Obj_Mom", OBJPROP_XDISTANCE, 3);  // Coordinate X
   if (Win==0)
      ObjectSet("Obj_Mom",OBJPROP_YDISTANCE, 5);// Coordinate Y
   else
      ObjectSet("Obj_Mom",OBJPROP_YDISTANCE,15);// Coordinate Y
   return;                                       // Exit the user-defined function
  }
//--------------------------------------------------------------------------- 13 --
```

Prior to considering the code above, we should explain the specifics of the program operation. A graphical object once created (in this case, one displaying a text) is supposed to be present in the screen continuously. Its textual description is supposed to characterize the situation. The content of the textual description must be changed at the execution of the function start(), at every tick. At the same time, when switching between timeframes for the window, to which the EA is attached, the program goes through the following stages: deinit(), init(), (awaiting a tick), and start(). If the object is first time created during the execution of start(), then, every time when switching to another timeframe, a certain period of time will lapse before the object appears, the time period being equal to that of awaiting the next tick. This is very inconvenient, particularly, when timeframes are often switched between.

In a properly built program, the necessary messages are displayed on the screen at the moment of attaching the program to the symbol window or at the moment of timeframe switching (i.e., before a new tick incomes). For this purpose, as a rule, it is necessary to perform all actions to be performed at every tick at the launching of the special function start() at the stage of the execution of the special function init(). In order not to repeat the same program code in different special functions, the code can be organized as a separate function. For this purpose, the EA contains the user-defined function Main(). It is called to be executed once at the stage of initialization (block 2-3) and at every tick during the further work of the EA (block 3-4).

In the program (block 11-13), there are two more user-defined functions - Create_RSI() and Create_Mom() intended for creation and modification of the object properties. At the execution of the function init(), the necessary objects are created using these functions. The call to the function Main() results in giving necessary properties to the objects (the objects with the desired description of the desired color are displayed in the desired window).

Let's consider the function Main() (block 5-11) in more details. In block 6-7, the readings of the indicator RSI are calculated. Depending on whether the end of the indicator line is above 70, below 30, or within the range between these indexes, one or another value will be assigned to the variable Ind_RSI. Then this value is used as an index of arrays Color[] and Text[] (in block 7-8) to change the properties of the graphical object of the name "Obj_RSI".

Block 7-8. The number of RSI window is calculated in the line:

```
     Win_RSI_new = WindowFind("RSI(14)");// Window number of indicator RSI
```

The string value RSI(14) is used as the transferred parameter. This is the short name of the indicator, the number of which should be detected. In this case, the entire sequence of characters in the given line, including parentheses and digits, composes the name. It must be noted that, in general case, there can be several indicators of the same type in the symbol window, for example, RSI(14), RSI(21) and RSI(34). Each subwindow displaying these indicators has its own number. This is why technical indicators are developed in such a way that each of them forms its short name according to the preset values of adjustable parameters. The short name of each technical indicator coincides with that shown in the upper left corner of its subwindow (the short name of a custom indicator may be created by the programmer using the function IndicatorShortName()).

If the searched indicator is not placed in the symbol window, the variable Win_RSI_new (the number of the subwindow, in which this object must be displayed at the current moment) will take the value of -1, i.e., non-existing window. In this case, the program implies displaying of the graphical object in the main chart window, the number of which is always 0:

```
     if(Win_RSI_new == -1) Win_RSI_new=0;// If there is no ind., then the main window
```

During his or her operations, the user may place a missing indicator or delete an existing one. In order to find out about what actions should be performed, the program uses global variables Win_RSI_old and Win_Mom_old. The value of each variable is the number of the subwindow, in which the object has previously been created. If the values of the variables Win_RSI_new and Win_RSI_old don't coincide, this means that the indicator window is either added (it didn't exist before) or deleted (it was available on the previous tick). In both cases, the previously created object must be deleted, and a new one must be created in the desired window:

```
        ObjectDelete("Obj_RSI");          // Deletion of the object
        Create_RSI(Win_RSI_new);          // Create an object in the desired window
```

After the object has been created in the window numbered as Win_RSI_new, the value equal to the number of this window is assigned to the variable Win_RSI_old, i.e., the program remembers the number of window, in which the graphical object was created:

```
        Win_RSI_old = Win_RSI_new;        // Remember this window
```

If the values of the variables Win_RSI_new and Win_RSI_old coincide, it means that it is sufficient to assign a textual description to the object (that is now placed in the necessary window). It must also be done, in case of creating a new object:

```
        ObjectSetText("Obj_RSI",Text[Ind_RSI],10,"Arial",Color[Ind_RSI]);
```

Similar calculations are made for the other subwindow, that of indicator Momentum (blocks 8 - 10). At the end of the function Main(), all available graphical objects are redrawn as a result of the execution of WindowRedraw().

It's easy to see that programmed control over graphical objects in subwindows implies using global variables (you can also use 'static'). In such cases, when coding a program, you should pay a special attention to what values can be taken by global variables in different situations and what this can result in. In the program considered above, global variables are zeroized in the function init():

```
   Win_RSI_old = 0;                        // Technical moment
   Win_Mom_old = 0;                        // Technical moment
```

These lines are included into the program due to the fact that global variables lose their values, only if the user has stopped the execution of the application program in the symbol window. However, if the user has adjusted external variables or switched the timeframe, the program undergoes deinitialization and the consequent initialization, the values of global variables being saved.

Let's consider the operations of the program that doesn't contain these lines. Suppose, both indicators with the subwindow numbers 1 and 2, respectively, have been placed in the symbol window by the moment when the user switches the timeframe. In the considered example, when deinitializing the program, the graphical objects are deleted. At the execution of the special function init(), the objects are created in zero window. Then, at the execution of the function Main(), in blocks 7-8 and 9-10, the program compares the obtained number of the window, in which the objects must be placed, and the number of the window, in which the objects were at the preceding tick. In fact, the object has already been placed in zero window, but the values of global variables will say for other result: their numbers will be 1 and 2. As a result, the graphical objects will remain in the main window, until the user deletes and reattaches the corresponding indicators. As to prevent these undesirable developments, the program implies nulling of global variables at the execution of the function init(). Thus, the values of these variables are made correspond with the situation.

As a result of the execution of EA charts.mq4, the following combinations of windows and graphical objects displayed can appear:

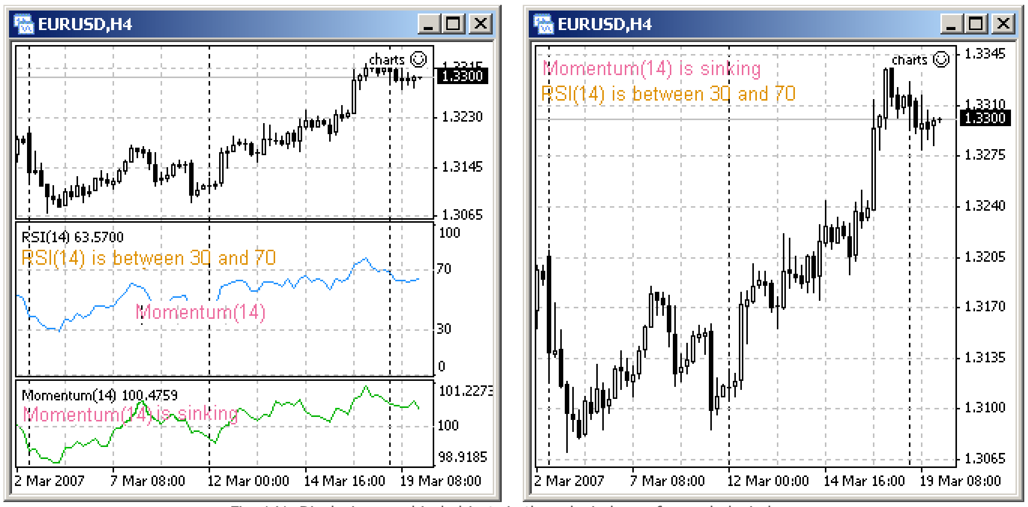


Fig. 141. Displaying graphical objects in the subwindows of a symbol window.

If there are both indicators in the symbol window, the corresponding graphical objects will be shown in their subwindows. If no one of these indicators is placed, then both objects will be created by the program in the main window. Adding or deletion of either indicator, the name of which is processed in the program, will result in moving of the corresponding graphical object into the necessary window. Adding or deletion of other indicators from the symbol window will not entail any consequences.

It must be noted separately that the alternative of deleting a graphical object by the user is not considered in this program. A program used in your practical trading must contain the analysis of such situation with the subsequent restoring of the object (see the solution of Problem 33).

Functions Used in Operations with Charts

Function	Summary Info
HideTestIndicators	The function puts a flag of hiding the indicators called by the Expert Advisor. At the opening the chart after testing, the indicators marked with the hiding flag will not be shown in the test graph. Before each call, the indicator is marked with the currently set hiding flag (only the indicators that are directly called from the EA under test can be displayed in the test graph).
Period	It returns the value of the amount of the period minutes for the current chart.
RefreshRates	Updating data in the predefined variables and timeseries arrays. This function is used, when an EA or a script has been calculating for a long time and needs updated data. It returns TRUE, if the data updating succeeds. Otherwise, it returns FALSE. The data may remain outdated only if they correspond with the current state of the client terminal.
Symbol	It returns a text line with the name of the current symbol.
WindowBarsPerChart	The function returns the amount of bars fitting in the window of the current chart.
WindowExpertName	It returns the name of the executing EA, script, custom indicator or library, depending on the MQL4 program, from which this function has been called.
WindowFind	It returns the number of the chart subwindow that contains the indicator with the given name 'name' if it has been found. Otherwise, it returns -1. WindowFind() returns -1, if the custom indicator is searching for itself during initialization init().
WindowFirstVisibleBar	The function returns the number of the first visible bar in the window of the current chart. You should consider that price bars are numbered in a reversed order, from the last to the first one. The current bar, which is the last in the price array, has index 0. The oldest bar has index Bars-1. If the number of the first visible bar is 2 or more less than the amount of visible bars in the chart, this means that the chart window is not full and there is a space to the right.
WindowHandle	It returns the window handle for the window that contains the given chart. If no chart with symbol and timeframe is opened at the moment of function call, it returns 0.

WindowIsVisible	It returns TRUE, if the chart subwindow is visible. Otherwise, it returns FALSE. The chart subwindow chan be hidden due to the visibility properties of the indicator attached to it.
WindowOnDropped	It returns the index of window, in which the EA, a script or a custom indicator has been dropped. This value will be true, only if the EAs, custom indicators and scripts are attached using a mouse (the technology of 'drag and drop'). For custom indicators being initialized (call from the function init()), this index is not defined. The returned index is the number of the window (0 is the main chart window, indicator subwindows are numbered starting with 1), in which the custom indicator is working. During initialization, a custom indicator can create its won new subwindow, and its number will differ from that of the window, in which the indicator has really been dropped.
WindowPriceMax	It returns the maximum value of the vertical scale of the given subwindow of the current chart (0 is the main chart window, indicator subwindows are numbered starting with 1). If the subwindow index is not specified, the maximum value of the price scale of the main chart window will be returned.
WindowPriceMin	It returns the minimum value of the vertical scale of the given subwindow of the current chart (0 is the main chart window, indicator subwindows are numbered starting with 1). If the subwindow index is not specified, the minimum value of the price scale of the main chart window will be returned.
WindowPriceOnDropped	It returns the price value at a chart point, at which an EA or a script have been dropped. The value will be true, only if the EA or the script have been moved using a mouse (the technology of 'drag and drop'). This value is not defined for custom indicators.
WindowRedraw	It redraws the current chart forcibly. The function is usually used after the object properties have been changed.
WindowScreenShot	It saves the display of the current chart in a GIF file. If it fails to make a screenshot, it returns FALSE.
WindowTimeOnDropped	The function returns the time value at a chart point, at which an EA or a script have been dropped. The value will be true, only if the EA or the script have been moved using a mouse (the technology of 'drag and drop'). This value is not defined for custom indicators.
WindowsTotal	The function returns the amount of indicator windows in the chart, including the main chart window.
WindowXOnDropped	It returns the value of X coordinate in pixels for the point in the chart window client area, where an EA or a script have been dropped. The value will be true, only if the EA or the script have been moved using a mouse (the technology of 'drag and drop').
WindowYOnDropped	It returns the value of Y coordinate in pixels for the point in the chart window client area, where an EA or a script have been dropped. The value will be true, only if the EA or the script have been moved using a mouse (the technology of 'drag and drop').

For the detailed description of these and other functions, please refer to Documentation at MQL4.community, MetaQuotes Software Corp. website or to "Help" section in MetaEditor.

← Graphical Objects String Functions →

String Functions

The most common operation with string values, addition (concatenation), was discussed in the Operations and Expressions (Problem 3) section. In some cases, there is a need in performing other calculations related to string values. MQL4 has a number of string functions for working with the values of *string* type. Let's consider the usage of some of them through the example below.

Problem 35. Color the last 100 bars of the candlestick chart as follows: black candlesticks in red, white candlesticks in blue.

A candlestick can be colored using two lines: a thin line must overlay a candlestick so that it covers all the shadows, whereas a thick line should fill out a candlestick body. In this case, we cannot use the lines of a custom indicator, because the displayed lines must be vertical, i.e., constructed using two coordinates (with the same time coordinates), while indicator arrays allow us to store only one value set in correspondence with each bar. So, the problem solution comes to displaying a series of single-type OBJ_TREND objects that differ in their coordinates and line style and color (see Graphical Objects) on a price chart.

In this case, the EA is used as an application program, but, in general, the algorithm can be implemented in a custom indicator. As a whole, the algorithm is clear. The chart must be colored for the first time, as soon as it is attached to the symbol window (during the execution of init()). The program must track possible changes in the location of graphical objects (user can accidentally move or delete one of them) with every tick coming, and restore them, if necessary. All objects created by the program must be deleted, as soon as the program finishes operating (deinit()).

A user can create other objects in a symbol window while the EA is working, for example, place the channel of standard deviations, Fibo levels, support lines, etc. So, the algorithm that allows us to distinguish user-created and program-created objects must be implemented in the program. This is particularly important when closing the program: it is necessary to remove only the program-created objects, while the user-created objects must remain unchanged. Each graphical object has its own properties that can generally coincide. The only identifying feature of any object is its unique name (the use of the same names is prohibited).

It's recommended to enter the useful information in the object's name while composing it, so it will be possible to detect the location and the properties of the object. For example, an object name may contain a prefix that differentiates a program-created object from others. In this case, it is "Paint_". Besides, it is necessary to differentiate the "user-defined" objects from any other, as well. The same time a simple numeration (Paint_1, Paint_2) cannot be used. Using this method of objects numeration, you cannot understand, at which bar the object Paint_73 should be displayed. The bar that has the Paint_73 index will get the Paint_74 index, when a new bar comes, Paint_75 index when another new bar comes, etc. In such a case, it would be necessary to delete and re-create all the objects on every new bar. This solution (although it is possible) is obviously very rough and expensive.

Every object created must have its time coordinates that correspond with the time of bar opening. Besides, two lines must be displayed on every bar - a thin line and a thick line. It is most comfortable to represent the names of a created objects by the program as follows:

Object name = Paint_2_2007.03.22 16:40, here:

Paint_ - prefix that differentiates the objects created by the program;

2_ - number of either objects that are displayed on a bar (value 1 or 2 is possible);

2007.03.22 16:40 - time coordinate that uniquely characterizes the bar the object is displayed on.

Paint_ and 2_ are the values of the variables Prefix and Nom_Lin, respectively. The time coordinate can be obtained for every bar by transformation a *datetime* value into a *string* value using the transforming functions:

TimeToStr() Function

```
string TimeToStr(datetime value, int mode=TIME_DATE|TIME_MINUTES)
```

The function transforms the values that contain time (in seconds) lapsed since 01.01.1970 (*datetime* value) into a string of the specified format (*string* value).

Parameters:

value - time in seconds lapsed since 00:00 of the 1st of January 1970;

mode - an additional mode of data output. It can be a single or a combined flag:

TIME_DATE obtains the result in the "yyyy.mm.dd" form;

TIME_MINUTES obtains the result in the "hh:mi" form;

TIME_SECONDS obtains the result in the "hh:mi:ss" form.

Let's consider the EA strings.mq4 that manages objects for coloring of candles and see how the TineToStr() is used in this program:

```
//--------------------------------------------------------------------
// strings.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
extern int Quant_Bars=100;              // Number of bars
datetime   Time_On;
string     Prefix    ="Paint_";
//--------------------------------------------------------------- 2 --
int init()                              // Spec. function init()
  {
   int Ind_Bar;                         // Bar index
   Time_On=Time [Quant_Bars];           // Time of first coloring
   for(Ind_Bar=Quant_Bars-1; Ind_Bar>=0; Ind_Bar--)// Bars cycle
     {
      Create(Ind_Bar,1);                // Draw a thin line
      Create(Ind_Bar,2);                // Draw a thick line
     }
   WindowRedraw();                      // Image redrawing
   return;                              // Exit init()
  }
//--------------------------------------------------------------- 3 --
int start()                             // Spec. function start()
  {
   datetime T1, T2;                     // 1 and 2 time coordinates
   int Error,Ind_Bar;                   // Error code and bar index
   double P1, P2;                       // 1 and 2 price coordinates
   color Col;                           // Color of created object
//--------------------------------------------------------------- 4 --
   for(int Line=1; Line<=2; Line++)     // Line type cycle
     {
      string Nom_Lin =Line + "_";       // String with the line number
//    string Nom_Lin  = DoubleToStr(Line,0)+"_";// Can be so
      for(Ind_Bar=0; ;Ind_Bar++)        // Bar cycle
        {
//--------------------------------------------------------------- 5 --
         datetime T_Bar= Time[Ind_Bar];// Bar opening time
         if (T_Bar < Time_On) break;    // Don't color out of borders
         string Str_Time=TimeToStr(T_Bar);        // Time string
         string His_Name=Prefix+Nom_Lin+Str_Time;// Object name
//--------------------------------------------------------------- 6 --
         T1=ObjectGet(His_Name,OBJPROP_TIME1);// t1 coord. query
         Error=GetLastError();          // Error code receiving
         if (Error==4202)               // If there is no object :(
           {
            Create(Ind_Bar,Line);       // Object creating function call.
            continue;                   // To the next iteration
           }
//--------------------------------------------------------------- 7 --
         T2 =ObjectGet(His_Name,OBJPROP_TIME2); // t2 coord. query
         P1 =ObjectGet(His_Name,OBJPROP_PRICE1);// p1 coord. query
         P2 =ObjectGet(His_Name,OBJPROP_PRICE2);// p2 coord. query
         Col=ObjectGet(His_Name,OBJPROP_COLOR); // Color query
         if(T1!=T_Bar || T2!=T_Bar || // Incorrect coord. or color:
            (Line==1 && (P1!=High[Ind_Bar] || P2!=  Low[Ind_Bar])) ||
            (Line==2 && (P1!=Open[Ind_Bar] || P2!=Close[Ind_Bar])) ||
            (Open[Ind_Bar] Close[Ind_Bar] && Col!=Red)  ||
            (Open[Ind_Bar]==Close[Ind_Bar] && Col!=Green)   )
           {
            ObjectDelete(His_Name);     // Delete object
            Create(Ind_Bar,Line);       // Create correct object
           }
//--------------------------------------------------------------- 8 --
        }
     }
   WindowRedraw();                      // Image redrawing
   return;                              // Exit start()
  }
//--------------------------------------------------------------- 9 --
int deinit()                            // Spec. function deinit()
  {
   string Name_Del[1];                  // Array declaring
   int Quant_Del=0;                     // Number of objects to be deleted
   int Quant_Objects=ObjectsTotal();    // Total number of all objects
```

```
     ArrayResize(Name_Del,Quant_Objects);// Necessary array size
     for(int k=0; k<=Quant_Del; i++)      // Delete objects with names..
        ObjectDelete(Name_Del[i]);        // .. that array contains
     return;                              // Exit deinit()
    }
  //-------------------------------------------------------------- 10 --
  int Create(int Ind_Bar, int Line)       // User-defined function..
    {                                     // ..of objects creation
     color Color;                         // Object color
     datetime T_Bar=Time [Ind_Bar];       // Bar opening time
     double   O_Bar=Open [Ind_Bar];       // Bar open price
     double   C_Bar=Close[Ind_Bar];       // Bar close price
     double   H_Bar=High [Ind_Bar];       // Bar maximum price
     double   L_Bar=Low  [Ind_Bar];       // Bar minimum price

     string Nom_Lin =Line + "_";          // String - line number
     // string Nom_Lin  = DoubleToStr(Line,0)+"_";// Can be so
     string Str_Time=TimeToStr(T_Bar);    // String - open time.
     string His_Name=Prefix+Nom_Lin+Str_Time;// Name of created object
     if (O_Bar < C_Bar) Color=Blue;       // Choosing the color depending on..
     if (O_Bar >C_Bar) Color=Red;         // .. parameters of the bar
     if (O_Bar ==C_Bar) Color=Green;

     switch(Line)                         // Thin or thick line
       {
        case 1:                           // Thin line
           ObjectCreate(His_Name,OBJ_TREND,0,T_Bar,H_Bar,T_Bar,L_Bar);
           break;                         // Exit from switch
        case 2:                           // Thick line
           ObjectCreate(His_Name,OBJ_TREND,0,T_Bar,O_Bar,T_Bar,C_Bar);
           ObjectSet(   His_Name, OBJPROP_WIDTH, 3);// Style
       }
     ObjectSet(     His_Name, OBJPROP_COLOR, Color); // Color
     ObjectSet(     His_Name, OBJPROP_RAY,    false); // Ray
     ObjectSetText(His_Name,"Object is created by the EA",10);// Description
     return;                              // Exit user-defined function
    }
  //-------------------------------------------------------------- 11 --
```

In order to create graphical objects, the user-defined function Create() (blocks 10-11) is used in the program. The variable Ind_Bar that indicates the index of bar the object should be created on, and Line, the object number (line 1 or 2), are used as the assignable parameters in this function.

Three components are used when forming the name of the object to be created:

```
     string His_Name = Prefix+Nom_Lin+Str_Time;// Name of created object
```

The value of the Prefix variable is specified by the programmer in the head part of the program and it is not changed during the program execution:

```
  string      Prefix      = "Paint_";
```

The value of the Nom_Lin variable is obtained as a result of calculations:

```
     string Nom_Lin  = Line + "_";        // String - line number
  // string Nom_Lin  = DoubleToStr(Line,0)+"_";// Can be so
```

Here the value of the integer variable (during calculation in the first part of expression) is transformed into the type of the higher priority, namely, into the *string* type. As a result, the Nom_Lin receives "1_" or "2_" values depending on the value of the Line variable.

In order to calculate the value of the Str_Time variable the TimeToStr() function of data transformation is used:

```
     string Str_Time = TimeToStr(T_Bar); // String - open time
```

Please note that the TimeToStr() function has default values. In this case, these are these values that are necessary: "yyyy.mm.dd hh:mi"; there is no need to use seconds additionally, because the minimum timeframe is equal to 1 minute.

We could also apply the following method of Str_Time calculation to be used in the object name:

```
    string Str_Time = T_Bar;
```

In this case, the Str_Time would obtain a value equal to the number of seconds lapsed since 01.01.1970. In order to see the difference, we can develop a program that contains the following code:

```
int init()
  {
   string String_Time = TimeToStr(Time[0]); // Time in the format
   string String_Sec  = Time[0];            // Number of seconds
   Alert("String_Time = ",String_Time,"  String_Sec = ",String_Sec);
   return;
  }
```

The following message (according to the time of zero bar opening) will be displayed on the screen as a result of the program execution:

```
String_Time = 2007.03.22 19:10 String_Sec = 1174590600
```

The first alternative that is implemented in the strings.mq4 EA is a bit more informative, so the preference is given to it, in this case (the alternatives are equivalent in terms of composing an algorithm).

The object named His_Name is created in the subsequent lines of the user-defined function Create(). It contains the information about the bar opening time with the parameters that correspond to the number of the "Line" line and also the color depending on bar characteristics. The value of the text description is specified for every object, "Object is created by EA", as well.

The Create() function is called in the program from two places: from the special function init() for the initial creation of objects, and from the special function start() to re-create the object, if necessary, in case it was deleted or modified by the user. The object names in start() (blocks 4-6) are formed in the same way as in other parts of the program.

The first coordinate of the considered object is defined in block 6-7. If the object is not found at this time, it will be created by the Create() function. And if the object exists, its other coordinates will be determined and the matching of its parameters with the bar properties will be checked (block 7-8). The object will be deleted and re-created (with the same name) with the correct properties, if any mismatch is detected.

Another problem is solved during the execution of the deinit() function: it is necessary to delete only the objects that have been created by the program from the aggregate of all objects in the symbol window. This is performed in two stages: at the first stage, the names of all objects that should be deleted are memorized to the Name_Del[] array, and then they will be deleted in an individual cycle. The total number of objects in the window (including those created by the program and placed manually by the user) is calculated using the ObjectsTotal() function:

```
    int Quant_Objects=ObjectsTotal();    // Total number of ALL objects
```

The number of bars to be colored is set by the user in an external variable, i.e., it is unknown in advance how many objects should be deleted. So the string array that contains the names of the objects to be deleted is declared with the number of elements equal to 1. Further, its size is programmatically changed - the number of elements is increased to the total number of objects.

```
    ArrayResize(Name_Del,Quant_Objects);// Necessary array size
```

In order to select the objects that have been created by the EA, the deinit() function contains the cycle 'for' that analyzes the names of all objects.

```
       string Obj_Name = ObjectName(k); // Querying name of the object
```

The attribute that differentiates "our" objects from all the others is the "Paint_" prefix, with which the name of each program-created object starts. To analyze an object name, we should extract the first part (in this case, 6 symbols) from the string variable being the unique name of the object; then we should compare this value with that of the Prefix variable. If they match, this object should be deleted. If not, it should no be deleted.

StringSubstr() Function

```
string StringSubstr(string text, int start, int length=0)
```

The function extracts the substring from the text line starting from the specified position. The function returns the copy of the extracted substring. Otherwise, an empty string is returned.
Parameters:

text - the line the substring should be extracted from;

start - the initial position of the substring. It can range from 0 to StringLen(text)-1;

length - the length of the substring to be extracted. If the value of this parameter is less than or equal to 0 or it is not even specified then the substring will be extracted from the specified position till the end of the line.

In the considered example, the substring is extracted from the object name as follows:

```
      string Head=StringSubstr(Obj_Name,0,6);// Extract first 6 symbols
```

In this case, the first 6 symbols are extracted from the Obj_Name string variable starting with the zero one. Please note that the count of all indexes (bars, arrays), entries in the orders list and also the number of the position in the line starts with 0, whereas the quantified count starts with 1.

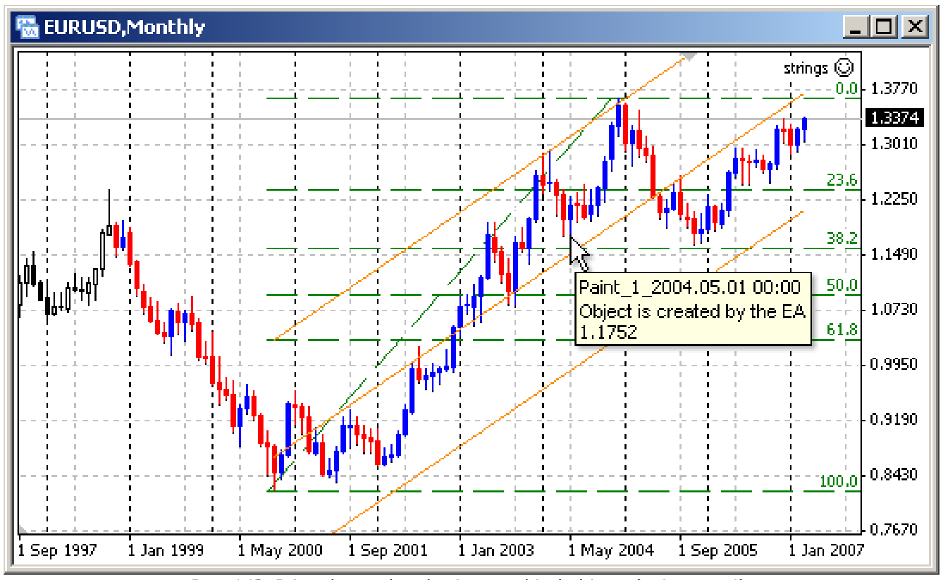
The extracted substring (a *string* value) is assigned to the string variable Head. If the object name (and in the object itself) is created by the considered EA, the value of the extracted substring will be "Paint_". If another name is analyzed, then the desired value will be different. For example, the value of the extracted substring from the "StdDev Channel 23109" object name will be the following: "StdDev", and for the object named "Fibo 22800" it will be "Fibo 2".

In the subsequent lines, the value of the variable Head is compared to that the variable Prefix:

```
      if (Head == Prefix)                 // The object beginning..
         {                                // .. with Paint_ is found
```

If these values are equal to each other, then the analyzed name of the object will be placed to the array Name_Del[] for the names of objects to be deleted. In the next "for" cycle, all the objects, the names of which are contained by the array, will be deleted (it should be noted separately that it is impossible to delete all the objects during the first cycle, because, in this case, the total number of objects and their numeration will be changed each time the object is deleted, which will result in the omission of some object names).

The price chart will have the following appearance during the execution of the strings.mq4 EA:



Paint_1_2004.05.01 00:00
Object is created by the EA
1.1752

Рис. 142. Price chart colored using graphical objects (strings.mq4).

Besides the groups of objects that cover the price chart, two other objects placed manually by the user are displayed, as shown in Fig. 142; they are regression channel and Fibo levels. The objects created by the EA will be deleted, as soon as its execution is finished, and the objects created by the user will remain in the symbol window. This result is obtained due to the use of string functions in the program. They allow to create and analyze string values, including graphical object names.

String Functions

Function	Short Description
StringConcatenate	It forms a string from the given parameters and returns it. The parameters can be of any type. The number of parameters cannot exceed 64.
StringFind	Substring searching. It returns the number of the position in the line the desired substring starts

	with, or -1, in case the substring is not found.
StringGetChar	It returns the value of the symbol that is located at the specified position of the line.
StringLen	It returns the number of symbols in the line.
StringSetChar	It returns the copy of the line with the modified value of the symbol at the specified position.
StringSubstr	It extracts the substring that starts at the specified position in the text line. The function returns the copy of the extracted substring, if possible. Otherwise, an empty string is returned.
StringTrimLeft	The function cuts the carriage return characters, spaces and tabulation symbols from the left part of the string. The function returns the copy of the modified string, if possible. Otherwise, an empty string is returned.
StringTrimRight	The function cuts the carriage return characters, spaces and tabulation symbols from the right part of the string. The function returns the copy of the modified string, if possible. Otherwise, an empty string is returned.

Data Transformation Functions

Function	Summary Info
CharToStr	Transformation of the symbol code into a single-symbol string.
DoubleToStr	Transformation of the numeric value into a text string that contains the symbolic representation of the number with the specified accuracy format.
NormalizeDouble	Rounding off the number with the floating point to the specified accuracy. The calculated StopLoss, TakeProfit and also the open prcie of pending orders values must be normalized according to the accuracy that is stored in the defined Digits variable.
StrToDouble	Transformation of the string that contains the symbolic representation of the number into the number of "double" type (double-accuracy format with the floating point).
StrToInteger	Transformation of the string that contains the symbolic representation into the number of the "int" type (integer).
StrToTime	Transformation of the string that contains time and/or date in the "yyyy.mm.dd [hh:mi]" format into the number of the "datetime" type (number of seconds passed since 01.01.1970).
TimeToStr	Transformation of the value that contains the time expressed in seconds passed since 01.01.1970 into the string of the "yyyy.mm.dd hh:mi" format.

To get the detailed information about these and other functions take a look at the Documentation at MQL4.community, at MetaQuotes Software Corp. website or at the "Help" section of MetaEditor.

← Operations with Charts Date and Time →

Date and Time

The online trading system MetaTrader 4 uses the indications of two time sources - the local (PC) time and the server time.

Local time - the time that is set on the local PC.

Server time - the time that is set on the server.

TimeLocal() Function

```
datetime TimeLocal()
```

The function returns the local PC time expressed in the number of seconds lapsed since 00:00 of the 1st of January 1970. Note: At testing, the local time is modeled and coincides with the modeled last-known server time.

A large majority of events that take place in the client terminal are considered with accordance to the server time. The time of tick coming, new bar beginning, order opening and closing is considered with accordance to the server time. To get the value of the server time that corresponds with the current time, the TimeCurrent() function should be used:

TimeCurrent() Function

```
datetime TimeCurrent()
```

The function returns the last known value of the server time (the time of the last quote coming) expressed in seconds lapsed since 00:00 of the 1st of January 1970. The client terminal updates the time of the last quote coming (together with other environment variables) before launching special functions for execution. Each tick is characterized with its own value of the server time that can be obtained using the TimeCurrent() function. During the execution, this value can only be changed as a result of the RefreshRates() function call and only if the information has been updated since the last execution of the RefreshRates() function, i.e., in case the new values of some environment variables have come from the server.

The time of bar opening, Time[i], does not coincide with the time of new tick coming, as a rule. The time of any timeframe bar opening is always divisible by the timeframe. Any first tick appeared within a timeframe is bar-forming; if there is no tick receipt within a timeframe, the bar will not be formed within the timeframe.

For example, the tick coming to the terminal at time (server) t0 results in forming a bar with the time opening equal to Time[i+2] (Fig. 143). The moment specified as the beginning of the timeframe does not concur with moment t0, though it can accidentally concur with it, in general. The subsequent ticks that come to the terminal within the same timeframe (at the moments of t1 and t2) can change the parameters of the bar, for example, maximum price or open price, but they do not affect the time of bar opening. The bar closing time is not considered in the online trading system MetaTrader 4 (formally, the time of the last tick coming within a timeframe or the beginning time of the next timeframe can be considered as the bar closing time, as shown in Fig. 143).

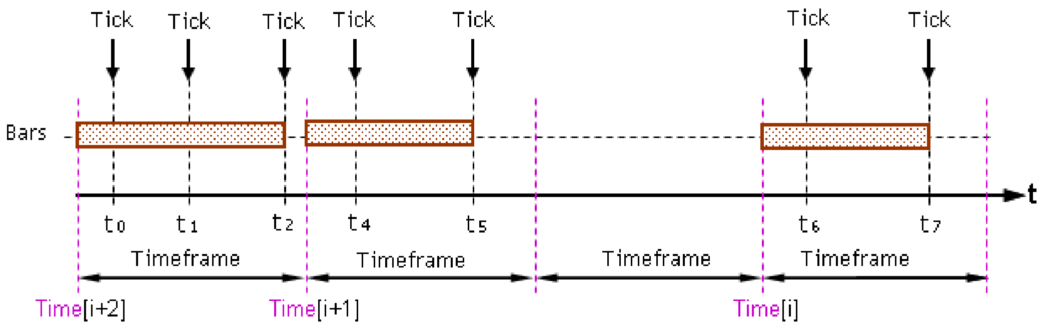


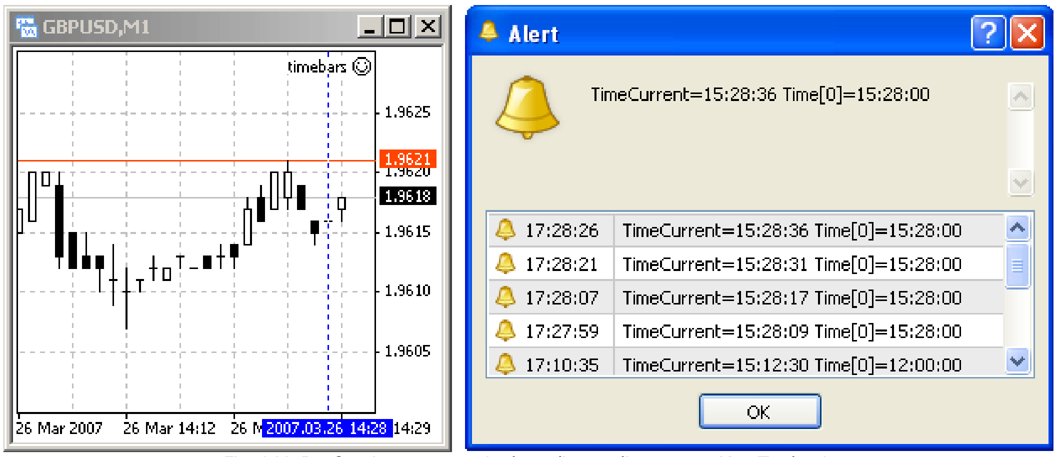
Fig. 143. Bar forming sequence in the online trading platform MetaTrader 4.

It is shown in Fig. 143 that it is possible that bars are not formed at some time periods that are equal to the timeframe. Thus, between time t5 of the tick coming and t6 of the next tick coming, the full timeframe is packed, so the new bar hasn't been formed at that time period. In this manner, the time of bar opening may differ from the time of opening of an adjacent bar by more than a whole timeframe, but it is always divisible by a timeframe. To demonstrate the sequence of bar forming, we can use the EA timebars.mq4 that outputs the time of tick coming and the time of bar opening:

```
//--------------------------------------------------------------------
// timebars.mq4
// The program is intended to be used as an example in MQL4 Tutorial.
//--------------------------------------------------------------------
int start()                            // Spec. function start()
  {
   Alert("TimeCurrent=",TimeToStr(TimeCurrent(),TIME_SECONDS),
         " Time[0]=",TimeToStr(Time[0],TIME_SECONDS));
   return;                             // Exit start()
  }
```

```
//--------------------------------------------------------------------
```

The results of the EA timebars.mq4 working are shown in Fig. 144. It is obvious that the first tick at the regular time period of 1 minute duration came at 14:29:12, at the same time a new bar was formed with the opening time - 14:29:00. Please note that the right column of the message box displays the server time, the left column displays the local time.



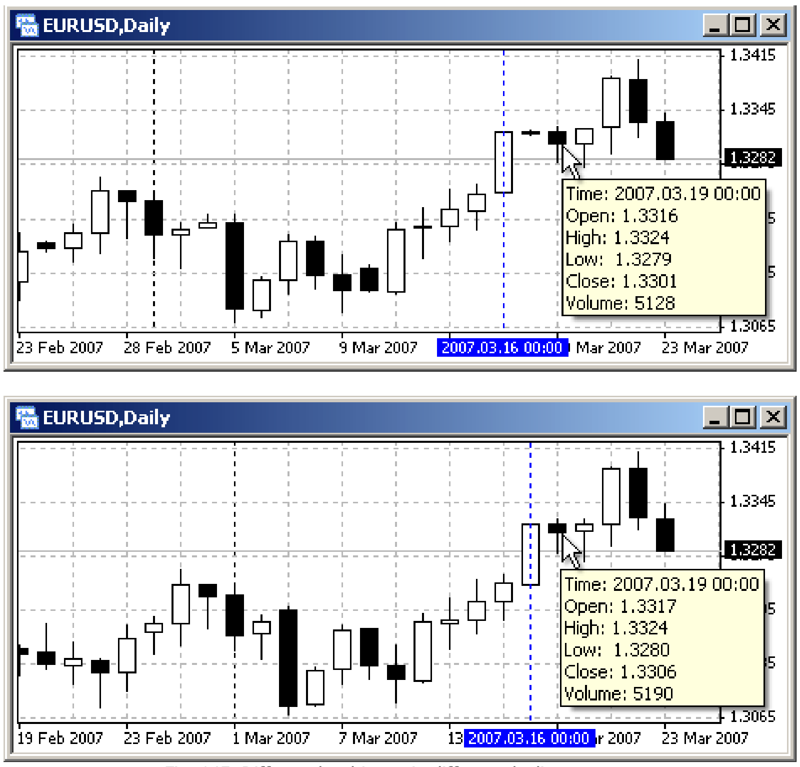
TimeCurrent=15:28:36 Time[0]=15:28:00

17:28:26 TimeCurrent=15:28:36 Time[0]=15:28:00
17:28:21 TimeCurrent=15:28:31 Time[0]=15:28:00
17:28:07 TimeCurrent=15:28:17 Time[0]=15:28:00
17:27:59 TimeCurrent=15:28:09 Time[0]=15:28:00
17:10:35 TimeCurrent=15:12:30 Time[0]=12:00:00

Fig. 144. Bar forming sequence in the online trading system MetaTrader 4.

In case the ticks come rarely (for example, the period between the end of the European session and the beginning of the Asian session), you can observe another phenomenon during the execution of timebars.mq4: the opening time of the adjacent bars can differ from each other by more than 1 minute (for one-minute timeframe). At the same time, the indexing of bars is saved in succession, without spaces.

The server time of servers in different dealing centers may vary. The time of beginning and finishing trades is set on each server individually and it can disagree with the begining and the end of the regular day. Some dealing centers, for example, have the settings that perform trade opening on Sunday at 23:00 of server time. This results in forming of incomplete daily bars, their practical duration is equal to one hour (Fig. 145).



Time: 2007.03.19 00:00
Open: 1.3316
High: 1.3324
Low: 1.3279
Close: 1.3301
Volume: 5128

Time: 2007.03.19 00:00
Open: 1.3317
High: 1.3324
Low: 1.3280
Close: 1.3306
Volume: 5190

Fig. 145. Different bar history in different dealing centers.

The usage of date and time functions is rather easy in MQL4. Some of them transform the server and the local time in seconds lapsed since 00:00 of the 1st of January 1970 into an integer number that corresponds with an hour, a day, etc. Other functions return an integer number that corresponds with the current hour, day, minute, etc.

TimeSeconds (), TimeMinute(), TimeHour(), TimeDay(), TimeMonth(), TimeYear(), TimeDayOfWeek () and TimeDayOfYear() Functions

This is a group of functions that return the number of seconds lapsed from the beginning of the minute, or minute, hour, day, month, year, day of week and day of year for the specified time. For example:

```
int TimeMinute(datetime time)
```

The function returns minutes for the specified time.

Parameters:

time - the date expressed in number of seconds that lapsed since 00:00 of the 1st of January 1970.

```
int TimeDayOfWeek(datetime time)
```

This function returns the day of week (0-Sunday,1,2,3,4,5,6) for the specified date.

Parameters:

time - the date expressed in number of seconds that lapsed since 00:00 of the 1st of January 1970.

The considered functions can be used for analysis of any bar opening time, for example. The EA named bigbars.mq4 intended for finding bars of a size that is not less than specified size is shown below.

```
//--------------------------------------------------------------------
// bigbars.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
extern int Quant_Pt=20;                 // Number of points
//--------------------------------------------------------------- 2 --
int start()                             // Spec. function start()
  {
   int H_L=0;                           // Height of the bar
   for(int i=0; H_L<Quant_Pt; i++)      // Cycle for bars
     {
      H_L=MathAbs(High[i]-Low[i])/Point;//Height of the bar
      if (H_L>=Quant_Pt)                // if the high bar is not found
        {
         int YY=TimeYear(  Time[i]);    // Year
         int MN=TimeMonth( Time[i]);    // Month
         int DD=TimeDay(   Time[i]);    // Day
         int HH=TimeHour(  Time[i]);    // Hour
         int MM=TimeMinute(Time[i]);    // Minute
         Comment("The last price movement more than ",Quant_Pt,//Message
         " pt happened ", DD,".",MN,".",YY," ",HH,":",MM);//output
        }
     }
   return;                              // Exit start()
  }
//--------------------------------------------------------------- 3 --
```

The bigbars.mq4 EA searches the nearest bar whose height (difference between minimum and maximum) is more than or equal to the value specified in the external variable Quant_Pt. The date and time of the found bar are outputted to the window of financial instrument by the Comment() function.

Seconds (), Minute(), Hour(), Day(), TimeMonth(), TimeYear(), DayOfWeek () and DayOfYear() Functions

This is the group of functions that return the current second, minute, hour, day, month, year, day of week and day of year for the last known server time. The last known server time is the server time that corresponds with the moment of the program launch (launch of any special function by the client terminal). The server time is not changed during the execution of the special function.

```
int Hour()
```

It returns the current hour (0,1,2,..23) of the last known server time. Note that the last known server time is modeled during testing.

```
int DayOfYear()
```

It returns the current day of the year (1 is the 1st of January,..,365(6) is the 31st of December), i.e., the day of the year of the last known server time. Note that the last known server time is modeled during testing.

The EA timeevents.mq4 that performs some actions as soon as the specified time comes can be used as an example of usage of the above functions.

```
//--------------------------------------------------------------------
// timeevents.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
extern double Time_Cls=16.10;          // Orders closing time
bool Flag_Time=false;                  // Flag, there are no messages yet
//--------------------------------------------------------------- 2 --
int start()                            // Spec. start function
  {
   int    Cur_Hour=Hour();             // Server time in hours
   double Cur_Min =Minute();           // Server time in minutes
   double Cur_time=Cur_Hour + Cur_Min100; // Current time
   Alert(Cur_time);
   if (Cur_time>=Time_Cls)             // If the time for the event has come
      Executor();                      //.. then perform devised actions
   return;                             // Exit from start()
  }
//--------------------------------------------------------------- 3 --
int Executor()                         // User-defined function
  {
   if (Flag_Time==false)               // If there are no messages yet
     {                                 // .. then report (1 time)
      Alert("Important news time. Close orders.");
      Flag_Time=true;                  // Now the message has already appeared
     }
   return;                             // Exit user-defined function
  }
//--------------------------------------------------------------- 4 --
```

The server time is calculated in hours and minutes during the execution of the special function start() (blocks 2-3). The line:

```
   double Cur_time = Cur_Hour + Cur_Min100; // Current time
```

represents the current server time expressed as the real variable Cur_time. The use of real variables is convenient in comparison operations:

```
   if (Cur_time >= Time_Cls)           // If the time for the event has come
```

If the current time is more than or equal to the value of Time_Cls specified by user, then the Executor() user-defined function will be called for execution. In this example, the user-defined function puts out a message with trading recommendations. In general, this function may contain any code, for example, make trades, send e-mails, create graphical objects, etc.

Functions of Date and Time

Function	Description
Day	It returns the current day of month, i.e. the day of month of the last known server time.
DayOfWeek	It returns the index number of the day of week (sunday-0,1,2,3,4,5,6) of the last known server time.
DayOfYear	It returns the current day of year (1 is the 1st of January,..,365(6) is the 31st of December), i.e. the day of year of the last known server time.
Hour	It returns the current hour (0,1,2,..23) of the last known server time at the moment of program start (the value is not changed during the execution of the program).
Minute	It returns the current minute (0,1,2,..59) of the last known server time at the moment of program start (the value is not changed during the execution of the program).
Month	It returns the number of the current month (1-January,2,3,4,5,6,7,8,9,10,11,12), i.e. the number of the month of the last known server time.
Seconds	It returns the number of seconds lapsed since the beginning of the current minute of the last known server time at the moment of program start (the value is not changed during the execution of the program).
TimeCurrent	It returns the last known server time (the time of last quote coming) expressed in the number of seconds that passed since the 00:00 January 1-st of 1970.
TimeDay	It returns the day of month (1 - 31) for the specified date.
TimeDayOfWeek	It returns the day of week (0-Sunday,1,2,3,4,5,6) for the specified date.

TimeDayOfYear	It returns the day (1 is the 1st of January,..,365(6) is the 31st of December) of year for the specified date.
TimeHour	It returns the hour for the specified time.
TimeLocal	It returns the local PC time expressed in the number of seconds lapsed since 00:00 of the 1st of January 1970.
TimeMinute	It returns minutes for the specified time.
TimeMonth	It returns the number of the month for the specified time (1-January,2,3,4,5,6,7,8,9,10,11,12).
TimeSeconds	It returns the number of seconds passed from the beginnig of the specified time.
TimeYear	It returns the year for the specified date. The returned value can be within the range of 1970-2037.
Year	It returns the current year, i.e. the year of the last known server time.

To get the detailed information about these and other functions, refer to the Documentation at MQL4.community, at MetaQuotes Software Corp. website or at the "Help" section of MetaEditor.

← String Functions File Operations →

File Operations

In MQL4, it is possible to work with files containing a certain set of information. It may become necessary to write information in a file or to read it from a file for several reasons.

A file can be used to deliver information to another program. In this case, the file can be created by an application program and used by it as an information receiver. For example, the trading history of an account can be written to a file at the execution of an application. This file can be later opened by another program (e.g., Excel for drawing a balance graph).

In other cases, there is a need to deliver some information, for example, the news timetable, to an application. An executable program (e.g., an Expert Advisor) can read this information from the previously prepared file and consider it during calculating for graphical displaying of the messages on the screen or for making of trade decisions.

File Names and Directories

The name of a working file must be composed according to the requirements of the operating system. The name of any file used in MQL4 consists of two parts: the file name and the file extension separated by a dot, for example, **News.txt**. Technically, a file name has no relation to the file content, so a file name and extension can be set voluntarily by the programmer. A file name is usually chosen so that it represents the information the file contains.

Most programs are automatically launched on the user's PC, if the file is double-clicked with the mouse button. According to the file extension, the operating environment loads one or another program displaying the file content. Therefore, you should assign the file extension considering the program (if necessary) that will usually be used to view the file.

The most popular file types (the type is determined by its extension) are the following:

- .**txt** - text file, for viewing you should use Notepad, Word, FrontPage, etc.;

- .**csv** - file for building tables in Excel;

- .**htm** - file to be viewed in a browser, i.e. Internet Explorer, Netscape Navigator, etc.

There are three folders (with subfolders) that can contain working files:

- Terminal_folder\Experts\History\current broker - for history files;

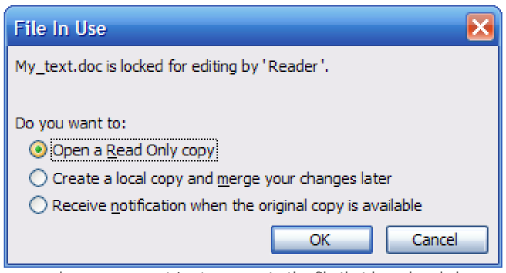
- Terminal_folder\Experts\Files\ - for common usage;

- Terminal_folder\Tester\ Files\ - for files the are used for testing.

A working file can be saved in one of these folders or in their subfolders. In case of no available folder at the moment of file saving, the folder will be automatically created by the client terminal. Working with files in other directories is not involved.

Modes of File Operations

The technology of interaction between an application and a working file has several modes. In general, a file can be opened by several programs at the same time (within a PC or several PCs connected to the network). At the same time, the operational environment provides the full access to the file, namely the right to read the file and write the information in it, only to one program. The other programs can only read it. For example, if My_text.doc has already been opened by a text editor, then all the other programs will receive the notification before opening the file:


File In Use

My_text.doc is locked for editing by 'Reader'.

Do you want to:

[✓ Open a Read Only copy]

Create a local copy and merge your changes later

Receive notification when the original copy is available

Fig. 146. Dialog box that appears when a program tries to access to the file that has already been opened by another program.

The execution of this technology guaranties that a file won't be modified simultaneously by two different programs. In order to allow an applicable program to interact with a file, you should open that file first. The mode of opening a file is specified in the FileOpen() function.

An application program can open several working files at a time. In order to allow the program to differentiate one file from another the file descriptor is set in accordance to every opened file.

File descriptor – unique number of the file that is opened by the program at the moment.

The FileOpen() function will return some integer value (this value is usually assigned to the 'handle' variable), if a file is opened successfully. That value is the file descriptor. Most functions that are intended to work with files suppose the use of a file descriptor as one of the formal parameters.

Function FileOpen()

```
int FileOpen(string filename, int mode, int delimiter=';')
```

The function opens a file for inputing and/or outputting. The function returns a file descriptor or -1, in case of failure. Files can only be opened in the **Terminal_folder\Experts \Files**\ folder or in the **Terminal_folder\Tester\Files**\folder (in case of EA testing) or in their subfolders.

Parameters:

filename - name of the file;

mode - the mode of file opening; it can have the following values (or their combinations): FILE_BIN, FILE_CSV, FILE_READ, FILE_WRITE;

delimiter - the separator sign for csv-files. It is ';' by default.

The FILE_READ mode of file opening implies that a file will be used only for being read by a program. A trial to open a file in this mode can fail, in case of no available file with the specified name.

The FILE_WRITE mode of file opening involves that a file will be used for writing in by a program. A try to open a file in this mode results in opening a file of a zero length. Even if there was some information in the file before opening, it will be erased. A try to open a file in this mode can fail, in case the file was previously opened by another program (in the writing mode).

It is allowed to open a file in the FILE_READ|FILE_WRITE mode. This mode involves the possibility of reading and writing to a file. This mode is used, if you need to add some information to the file that already contains some other information. The function implies the obligatory usage of one of the modes, FILE_READ or FILE_WRITE, or their combination.

The FILE_BIN mode of file opening defines processing a working file as a binary one. The FILE_CSV mode of file opening defines processing of a working file as a text one. The function involves obligatory usage of one of the FILE_BIN or FILE_CSV modes. The simultaneous usage of FILE_BIN and FILE_CSV modes is prohibited

The function requires obligatory combination of FILE_READ, FILE_WRITE or FILE_READ|FILE_WRITE modes with the FILE_BIN or FILE_CSV mode. For example: it is necessary to use the combination of FILE_CSV|FILE_READ to read the information from a text file, and it is necessary to use the FILE_BIN|FILE_READ|FILE_WRITE combination to add an entry to a binary file.

No more than 32 files can be opened simultaneously within an executable module (of an applicable program, e.g., an Expert Advisor). The descriptors of the files that are opened in the module cannot be passed to other modules (libraries).

Content of File Entries

The information entries are written to a file without spaces with any combination of modes. The information entries are added one by one when using the FILE_BIN mode for forming a file. Depending on the type of information that is written to a file (and the functions that are used to do it) the symbols combination representing the end of the line ("\r\n") can be written between the groups of entries. The information entries are separated by file separators (usually ';') when forming a file in the FILE_CSV mode, and the groups of entries (that compose a line) are separated with the combination of symbols that represent the end of the line ("\r\n").

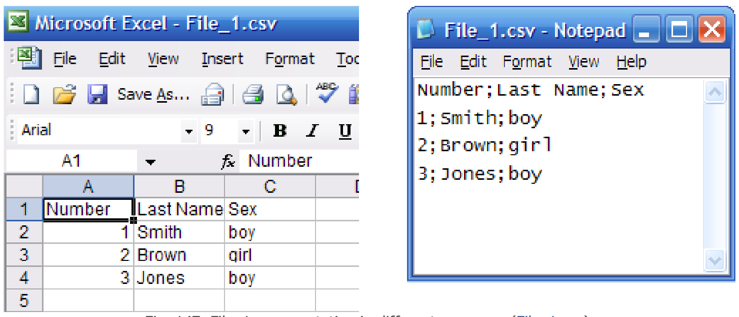
File separator - special symbol; the entry that is written to a file to separate the information lines.

The file separator is used to separate the information entries only in the csv-files.

The common principle for entries composition in any files is that these entries are added according to the specific sequence without spaces. Properly, the entry consists of continuous sequence of symbols. Any file can be read by any program and (depending on the rules implemented in it) can be displayed in some form on the screen. For example: we have the File_1.csv file that contains:

```
int FileOpen(string filename, int mode, int delimiter=';')
```

The File_1.csv file will be displayed in different ways in different text editors:


Microsoft Excel - File_1.csv
File Edit View Insert Format Too
Save As...
Arial 9 B I U
A1 Number
File_1.csv - Notepad
File Edit Format View Help

	A	B	C
1	Number	Last Name	Sex
2	1	Smith	boy
3	2	Brown	girl
4	3	Jones	boy
5			

```
Number;Last Name;Sex
1;Smith;boy
2;Brown;girl
3;Jones;boy
```

Fig. 147. File_1 representation in different programs (File_1.csv).

In this case, the "\r\n" symbol combination was interpreted by each of the programs (Excel and Notepad) as the evidence for formatting sequence: the sequence of symbols is represented in the next line after the "\r\n" combination of symbols, and the "\r\n" combination itself is not displayed in the editing window. At the same time, Excel is a table editor, so the ";" symbol was interpreted by the program as a separator of information to columns. Draw your attention that the ";" symbol is not displayed in the Excel window. Notepad is a text editor. The rules implemented in it do not suppose the division of information into columns, so the ";" symbol was not interpreted as a file separator, but it was interpreted as a part of information, so it is displayed on the screen.

The specified symbols (";" and "\r\n") are used to separate entries in MQL4.

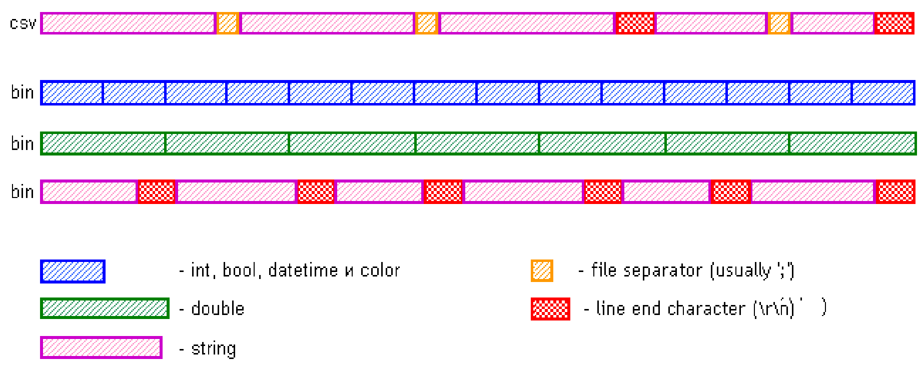

csv
bin
bin
bin
- int, bool, datetime и color
- double
- string
- file separator (usually ';')
- line end character (\r\n)

Fig. 148. Variety of entries in working files.

The structure of information writing in different types of files is represented on the fig. 148. The top line shows the csv-file content, the bottom three lines show the structure of binary files. All these files are composed according to the rules of one or another function of writing in the file.

An entry in the csv-file is the sequence of string values (string type) that are separated with the file separator or with the sign of the end of the line. Both of them are interpreted as a sign of the end of the informative read value part when reading information (using standard MQL4 function for file reading). The string value can have the different length and it is unknown how much symbols are there, so the reading is performed before one of the separators is located.

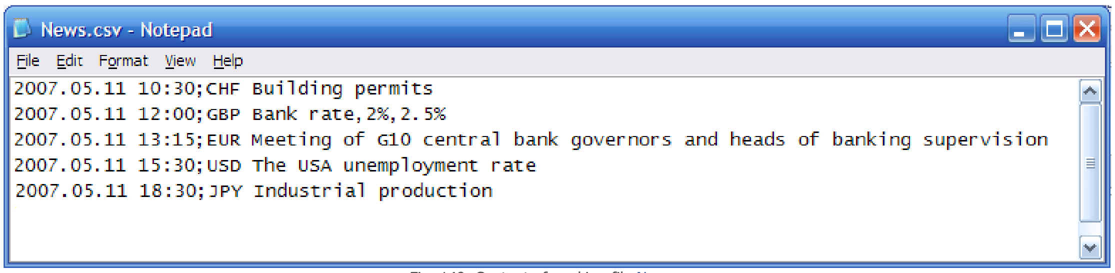
The entries in two types of binary of binary files represent the sequences of data without any separators. This sequence of writing in is governed by the fixed length for a data of different types: 4 bytes for a data of the "int", "bool", "datetime" and "color" types, and 8 bytes (or 4 bytes, depending on the parameters of writing function) for a data of "double" type. In this case, there is no need of separators, because the reading is performed by the standard function for reading data of a specified type with a specified length. The last (the bottom one on the fig. 148) binary file contains the data of string type that is separated with the end of the line sign.

File pointer - a position in the file the reading of the next value starts from.

The "File pointer" notion is the same with "cursor" notion. The file pointer is defined with the position in the file. As far as reading goes on the pointer is moving to the right per one or several positions.

Problem 36. Read the information about the important news from the file and display the graphical objects on the price chart (vertical lines) in accordance to the time of news publication.

Let the **Terminal_Folder\Experts\Files**\ folder contains the News.csv working file with the following content:



```
2007.05.11 10:30;CHF Building permits
2007.05.11 12:00;GBP Bank rate,2%,2.5%
2007.05.11 13:15;EUR Meeting of G10 central bank governors and heads of banking supervision
2007.05.11 15:30;USD The USA unemployment rate
2007.05.11 18:30;JPY Industrial production
```

Fig. 149. Content of working file News.csv.

In this case, the file contains the information about five events that will happen in different countries at a different time. Each line contains two entries. The first entry is the string value that represent the information about date an time of the event. The second entry is the text description of the event. First three symbols of the second entry contain the name of currency (country) that the event concerns.

The solution consists of two parts. First of all we need to read the information from the working file and then use the received values as the coordinates of the graphical objects. The reading of information is performed by the FileReadString() function.

FileReadString() Function

```
string FileReadString(int handle, int length=0)
```

The function reads the line from the current position of the file. It is suitable both for CSV and binary files. The line will be read till the separator is met in the text file. The specified number of symbols will be read in the binary files. In order to receive the information about an error you should call the GetLastError() function.

Parameters:

handle - the file descriptor that is returned by the FileOpen() function;

length - the number of characters to be read.

The need in news information processing appears only once at the beginning of trading, so, in this case, we can use a script to solve the problem 36. The timetablenews.mq4 script is intended to read the information from the file and display the graphical objects in the symbol window.

```
//--------------------------------------------------------------------
// timetablenews.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
int start()                            // Spec. function start()
  {
//--------------------------------------------------------------- 2 --
   int Handle,                         // File descriptor
       Stl;                            // Style of vertical line
   string File_Name="News.csv",        // Name of the file
          Obj_Name,                    // Name of the object
          Instr,                       // Name of the currency
          One,Two,                     // 1st and 2nd name of the instr.
          Text,                        // Text of event description
          Str_DtTm;                    // Date and time of the event (line)
   datetime Dat_DtTm;                  // Date and time of the event (date)
   color Col;                          // Color of the vertical line
//--------------------------------------------------------------- 3 --
   Handle=FileOpen(File_Name,FILE_CSV|FILE_READ,";");// File opening
   if(Handle<0)                        // File opening fails
     {
      if(GetLastError()==4103)         // If the file does not exist,..
         Alert("No file named ",File_Name);//.. inform trader
      else                             // If any other error occurs..
         Alert("Error while opening file ",File_Name);//..this message
      PlaySound("Bzrrr.wav");          // Sound accompaniment
      return;                          // Exit start()
     }
//--------------------------------------------------------------- 4 --
   while(FileIsEnding(Handle)==false)  // While the file pointer..
     {                                 // ..is not at the end of the file
      //--------------------------------------------------------- 5 --
      Str_DtTm =FileReadString(Handle);// Date and time of the event (date)
```

```
         Text      =FileReadString(Handle);// Text of event description
         if(FileIsEnding(Handle)==true)    // File pointer is at the end
            break;                         // Exit reading and drawing
         //--------------------------------------------------------------- 6 --
         Dat_DtTm =StrToTime(Str_DtTm);    // Transformation of data type
         Instr    =StringSubstr(Text,0,3);// Extract first three symbols
         One=StringSubstr(Symbol(),0,3);// Extract first three symbols
         Two=StringSubstr(Symbol(),3,3);// Extract second three symbols
         Stl=STYLE_DOT;                    // For all - dotted line style
         Col=DarkOrange;                   // For all - this color
         if(Instr==One || Instr==Two)      // And for the events of our..
           {                               // .. symbol..
            Stl=STYLE_SOLID;               // .. this style..
            Col=Red;                       // .. and this color of the vert. line
           }
         //--------------------------------------------------------------- 7 --
         Obj_Name="News_Line  "+Str_DtTm;              // Name of the object
         ObjectCreate(Obj_Name,OBJ_VLINE,0,Dat_DtTm,0);// Create object..
         ObjectSet(Obj_Name,OBJPROP_COLOR, Col);       // ..and its color,..
         ObjectSet(Obj_Name,OBJPROP_STYLE, Stl);       // ..and style..
         ObjectSetText(Obj_Name,Text,10);              // ..and description
        }
   //--------------------------------------------------------------------- 8 --
      FileClose( Handle );                 // Close file
      PlaySound("bulk.wav");               // Sound accompaniment
      WindowRedraw();                      // Redraw object
      return;                              // Exit start()
     }
   //--------------------------------------------------------------------- 9 --
```

The used variables are opened and described in block 2-3 of the EA. An attempt to open the file and the analysis of the results of this operation are performed in block 3-4. The FileOpen() function is used to open the file:

```
      Handle=FileOpen(File_Name,FILE_CSV|FILE_READ,";");// File opening
```

An attempt to open the file is not always successful. It can fail, if the file with the specified name is not available. When file opening fails (the file descriptor is a negative number) the necessary text message is displayed to the user and the execution of the start() function stops.

In case of successful opening of a file, control is passed to the operator of the "while" cycle (blocks 4-8). The reading of data from the file (block 5-6), transformation of data and its analysis (6-7 blocks) and creation of the graphical object with coordinates and parameters corresponding the last read information (block 7-8) are performed at each iteration.

The execution of the "while" cycle continues until the file pointer reaches the end of the file, i.e., there will be no information remaining to the right of the pointer. The FileIsEnding() function is used to analyze the position of the file pointer.

FileIsEnding() Function

```
   bool FileIsEnding(int handle)
```

The function returns TRUE if the file pointer is at the end of the file, otherwise it returns FALSE. In order to receive the information about an error you should use the GetLastError() function. The GetLastError() function will return the ERR_END_OF_FILE (4099) error, in case the end of the file is reached during the reading.

Parameters:

handle - file descriptor that is returned by the FileOpen() function.

The represented solution (timetablenews.mq4) involves that any number of news can be written to the News.csv file. News.csv file contains five entries corresponding to five events (news) in the mentioned example (fig. 149). In general, the number of lines may be from 0 to 20-30, depending on the amount of real events that must take place this day.

Reading entries from the file (that is identified by the "handle" variable) is performed in blocks 5-6:

```
         Str_DtTm =FileReadString(Handle);// Date and time of the event (date)
         Text     =FileReadString(Handle);// Text of event description
         if(FileIsEnding(Handle)==true)    // File pointer is at the end
            break;                         // Exit reading and drawing
```

The first and second lines of block 5-6 perform reading the information from the file until the nearest separator is met. The third and fourth lines perform checking: is the file pointer at the end of the line. If not, then the graphical objects will be formed via two read values further in the cycle. If it was initially known about the number of entries, then the analysis that is performed in the third and fourth lines would not be necessary. In this case, we would hardly specify the number of iterations in the cycle (e.g. 5) and would not perform an extra checking.

However, the number of entries is unknown, in this case. At the same time, in this example every event is described with two values that compose a line of the following kind: value, file separator, value, end of the line sign. In this case, it is supposed that if there is an entry (first value in the line) then the other one exists; but if there is no first entry then the second one does not exist, so there is no event and there is no need to create a graphical object. If both entries or one of them does not exist the pointer will move to the end of the file (i.e. the position in the file where no data to the right of the pointer exist) when a try to read it is performed. The checking performed in block 3-4 allows to discover this fact. If the noted checking (last two lines in block 5-6) is deleted, then needless object will be created while the program is running. Only after that the condition of "while" cycle ending will trigger and the control will be passed to block 8-9. In general, you should consider the logic of data representation in the file, sequence order of entries and separators, the number of lines, etc. while composing an algorithm for file reading. Each certain circumstance requires an individual algorithm.

Data read from the file has the string type. In order to use the received values for creating graphical objects you should transform the data to the necessary type. In block 6-7, the first (read in the next line) value is transformed to the "datetime" value and further will be used as the coordinate of the graphical object that corresponds the event. First three symbols from the second read string value are compared with the first and second triplet of symbols in the symbol name. If there is a coincidence then the graphical object receives the corresponding parameters: line style - solid and color - red (block 7-8). In other cases, the objects are displayed with the orange dotted line. You can observe the news lines in the symbol window as the result of the script execution:

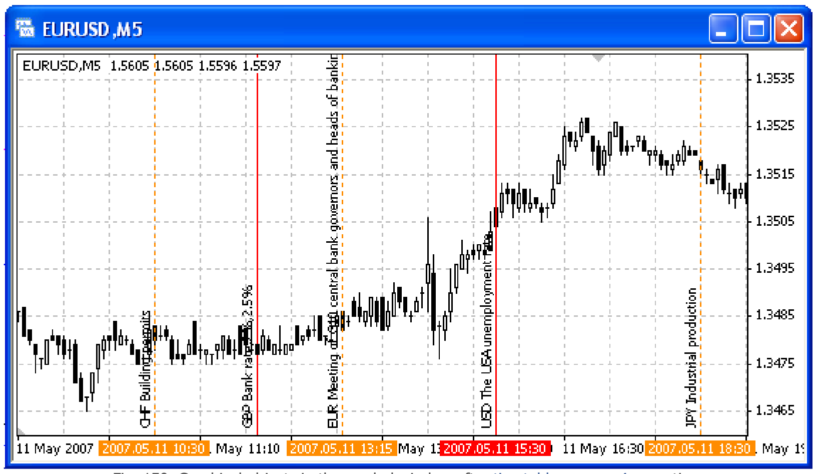


Fig. 150. Graphical objects in the symbol window after timetablenews.mq4 execution.

In such a way, the script can be executed in any symbol window. At the same time, every window will contain the solid red line that represent the events that concern this specific symbol, and the dotted lines that represent the vents concerning the other symbols' events. To display the text descriptions of the objects you should check the "Show object description" option in the Properties of security window (F8) => Common.

The previously opened file is closed in block 8-9 after the problem is solved, namely all the necessary objects are created. The file should be closed for the following reasons: on the first hand - not to spare extra PC resources and on the second hand to allow the other programs to access the file in the writing mode. It should be considered as normal to close the file as soon as all the information is read from it (or written in it) and its usage is not necessary anymore. The closing of file is performed by the FileClose() function.

FileClose() Function

```
void FileClose(int handle)
```

The function performs closing of a file that was previously opened by the FileOpen() function.

Parameters:

handle - file descriptor that is returned by the FileOpen() function.

In order to allow the trader to practically use the timetablenews.mq4 script, it must keep the method for creation of a file that contains the news timetable of some period. This type of file can be created using any text editor, however, in this case, the possibility of an error remains (sometimes a separator can be not specified erroneously). Lets examine a variant of working file creation using MQL4.

Problem 37. Represent the code of the EA that creates a file for news timetable.

In general, an EA can be destined for creation of a file that contains any number of news. The examined here createfile.mq4 EA creates the working file that contains the information about not more than five events.

```
//--------------------------------------------------------------------
// createfile.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
extern string Date_1="";    // 2007.05.11 10:30
extern string Text_1="";    // CHF Construction licenses
extern string Date_2="";    // 2007.05.11 12:00
extern string Text_2="";    // GBP Refinance rate,2%,2.5%
extern string Date_3="";    // 2007.05.11 13:15
extern string Text_3="";    // EUR Meeting of G10 banks governors
extern string Date_4="";    // 2007.05.11 15:30
extern string Text_4="";    // USD USA unemployment rate
extern string Date_5="";    // 2007.05.11 18:30
extern string Text_5="";    // JPY Industrial production
//--------------------------------------------------------------- 2 --
int start()                           // Spec. function start()
  {
//--------------------------------------------------------------- 3 --
   int Handle,                        // File descriptor
   Qnt_Symb;                          // Number of recorded symbols
   string File_Name="News.csv";       // File name
   string Erray[5,2];                 // Array for 5 news
//--------------------------------------------------------------- 4 --
   Erray[0,0]=Date_1;                 // Fill the array with values
   Erray[0,1]=Text_1;
   Erray[1,0]=Date_2;
   Erray[1,1]=Text_2;
   Erray[2,0]=Date_3;
   Erray[2,1]=Text_3;
   Erray[3,0]=Date_4;
   Erray[3,1]=Text_4;
   Erray[4,0]=Date_5;
   Erray[4,1]=Text_5;
//--------------------------------------------------------------- 5 --
```

```
    Handle=FileOpen(File_Name,FILE_CSV|FILE_WRITE,";");//File opening
    if(Handle==-1)                      // File opening fails
      {
       Alert("An error while opening the file. ",// Error message
              "May be the file is busy by the other appliction");
       PlaySound("Bzrrr.wav");          // Sound accompaniment
       return;                          // Exir start()
      }
//--------------------------------------------------------------- 6 --
    for(int i=0; i<=4; i++)             // Cycle throughout the array
      {
       if(StringLen(Array[i,0])== 0  || // If the value of the first or..
          StringLen(Array[i,1])== 0)    // ..second variable is not entered
          break;                        // .. then exit the cycle
       Qnt_Symb=FileWrite(Handle,Array[i,0],Array[i,1]);//Writing to the file
       if(Qnt_Symb < 0)                 // If failed
         {
          Alert("Error writing to the file",GetLastError());// Message
          PlaySound("Bzrrr.wav");       // Sound accompaniment
          FileClose( Handle );          // File closing
          return;                       // Exit start()
         }
      }
//--------------------------------------------------------------- 7 --
    FileClose( Handle );                // File closing
    Alert("The ",File_Name," file created.");// Message
    PlaySound("Bulk.wav");              // Sound accompaniment
    return;                             // Exit start()
   }
//--------------------------------------------------------------- 8 --
```

The initial information is entered to the program using the external variables of the "string" type (block 1-2). The variables are opened and described in block 3-4. To make the processing convenient the data is written to the Array[][] string array. Every event (information that characterize news) is represented by two elements of the array in the second dimension. The size of the first dimension (the number of lines in the array) is defined with the number of news, in this case, 5. In order to prevent the manual entering of values while trying the EA on a demo-account you can load the settings of the EA file example_news.set; the file of the EA setting should be located in the **Terminal_folder\presets ** folder.

Block 5-6 performs file opening. If the operation failed then the start() function ends working after the user has received the message. If the file is opened successfully then the control will be passed to the "for" cycle operator in block 6-7. In general, the number of input values, the size of the Erray array and the number of iterations can be increased to the necessary quantity.

The checking is performed every iteration: is one of the entered values empty. The length of the Array array values is calculated for this aim. If one of them has the zero length then it is considered as the absence of the current and the next events, so the current iteration interrupts. The writing of values of two elements of the array to the file goes on as far as the empty value of the element is found. The FileWrite() function is used for writing the values to the csv-file.

FileWrite() Function

```
  int FileWrite(int handle, ...)
```

The function is intended for writing the information to a csv-file, the separator between the information is included in automatically. The sign representing the end of the line "\r\n" is added to the file after the information writing. The information is transformed from the numeric to the text format when outputted (see Print() function). The function returns the number of written symbols or the negative value, in case an error occur.

Parameters:

handle - file descriptor that is returned by the FileOpen() function;

... - data separated with commas. It cannot be more than 63 parameters.

The data of the "double", "int" types is automatically transformed to the string (the data of "color", "datetime" and "bool" types is considered as the integer numbers of the "int" type and transformed to the string, as well), the data of the "string" type is output as is, without transformation. The arrays cannot be passed as the parameters; arrays must be entered elementwise.

In the considered example the information is written to the file in the following line:

```
       Qnt_Symb=FileWrite(Handle,Array[i,0],Array[i,1]);//Writing to the file
```

The separator (the symbol that is used as a separator is specified in the file opening function FileOpen(), in this case, ';') will be written after the Erray[i,0] value when writing to the file. The sign representing the end of the line "\r\n" is automatically placed at the end of the the FileWrite() function execution, i.e. at the end of writing. The same entry will be written on each next iteration of the "for" cycle. Every new entry starts from the position where the file separator of the last writing is placed. At the same time, the values of the next elements of the 'Erray' will be written to the file (indexes of the elements are increased by 1 on every iteration).

If the current writing to the file is successful the the control is passed to the next iteration. If the writing in the file fails then the file will be closed by the FileClose() function after the message is displayed to the user, and the start() function finishes its working. If all writings to the file are successfully performed then the control is passed to the file closing function FileClose() in block 7-8 after the execution of the "for" cycle is finished. In this case, the message about the successful file creation is displayed, after that the start() function execution will be finished. The News.csv file shown on the fig. 149 will be created after the EA execution is finished.

Function for Performing File Operations

Function	Summary Info
FileClose	The closing of the file that was previously opened by the FileOpen() function.

FileDelete	Deleting of the file. The files can only be deleted if they are located at the *terminal_folder\experts\files* (*terminal_folder\tester\files*, in case of testing the EA) folder or in its subfolders.
FileFlush	Flushing all the information that is left in the file input-output bufer to the hard disk.
FileIsEnding	Returns TRUE if the file pointer is at the end of the file, otherwise - returns FALSE. If the end of the file is reached during the file reading, the GetLastError() function will return the ERR_END_OF_FILE (4099) error.
FileIsLineEnding	It returns TRUE, if the file pointer is at the end of the line of the CSV-file. Otherwise, it returns FALSE.
FileOpen	Opens a file for inputting and/or outputting. The function returns the file descriptor of the opened file of -1, in case it fails.
FileOpenHistory	Opens a file in the current history folder (*termial_folder\history\server_name*) or in its subfolders. The function returns the file descriptor or -1, in case it fails.
FileReadArray	The function reads the specified number of elements from the binary file to the array. The array must have enough size before reading. The function returns the number of practically read elements.
FileReadDouble	The function reads the number of double accuracy with the floating point (double) from the current position of the binary file. The size of the number may the following: 8 bytes (double) and 4 bytes (float).
FileReadInteger	The function reads the integer number from the current position of the binary file. The size of the number may be the following: 1, 2 or 4 bytes. If the size of the number is not specified then the system will try to read it as it was the 4 byte integer number.
FileReadNumber	Reading the number from the current position of the CSV-file until the separator is met. It can be applied only to csv-files.
FileReadString	The function reads the line from the current position of the file. It can be applied both for csv and binary files. The line in the text file will be read until the separator is met. The specified number of symbols in the line will be read in the binary files.
FileSeek	The function moves the separator to the new position that is the displacement from the beginning, end or the current position of the file in bytes. The next reading or writing starts from the new position. If the pointer moving is performed successfully then the function will return TRUE, otherwise - FALSE.
FileSize	The function returns the size of the file in bytes.
FileTell	The function returns the shift of file pointer from the beginning of the file.
FileWrite	The function is intended to write the information to the csv-file, the separator is placed automatically between the information. The end of the line sign "\r\n" is added to the file after the writing is finished. The numeric data is transformed to the text format during the ouptputting process. The function returns the the number of written symbols or a negative value if an error occurs.
FileWriteArray	The function writes the array to the binary file.
FileWriteDouble	The function writes the number with the floating point to the binary file.
FileWriteInteger	The function writes the integer number value in the binary file.
FileWriteString	The function writes the line to the binary file from the current position. It returns the number of practically written bytes or a negative value, in case an error occurs.

To get the detailed information about these and other functions you should take a look at the documentation at MQL4.community, at MetaQuotes Software Corp. website or at the "Help" section of MetaEditor.

← Date and Time Arrays and Timeseries →

Arrays and Timeseries

It is very important to keep in mind that the sequence of any single-type elements is always numbered starting from zero in MQL4.

It was mentioned before that you shouldn't confuse the value of the array element index with the number of elements in the array (see Arrays). For example, if the array is declared:

```
    int Error_OHL[3];          // Array declaration
```

then it means that a one-dimensional array named Error_OHL contains three elements. Indexing of the elements start with zero, i.e. the first of three elements has the 0 index (Error_OHL[0]), the second one - the 1 index (Error_OHL[1]), and the third one - the 2 index (Error_OHL[2]). In such a way, the maximum index value is less than the number of the elements in array by one. In this case, the array is one-dimensional, i.e. we can say about the amount of elements in the first dimension: the maximum index number is 2, because the number of the elements in the array is 3.

The same thing can be said about the numeration of the dimensions in the array. For example, if an array is declared the following way:

```
    int Error_OHL[3][8];       // Array declaration
```

it means that the array has two dimensions. The first dimension specifies the number of rows (3 in this example), and the second one specifies the number of elements in the row (or the number of columns, 8 in this example). The dimension itself is numerated too. The first dimension has the 0 number, and the second one - the 1 number. The numbers of dimensions are used in the ArrayRange() function, for example.

ArrayRange() Function

```
  int ArrayRange(object array[], int range_index)
```

The function returns the number of elements in the specified dimension of the array.

The use of ArrayRange() function can be demonstrated with the solution of the following problem:

Problem 38. The Mas_1 array contains the values of the 3x5 matrix. Get the values of the Mas_2 array that contains the elements whose values are equal to the values of the transposable matrix. Use arbitrary values of the elements.

Let's work out some values of the elements and represent the initial and the desired matrices that the Mas_1 and Mas_2 arrays contain respectively:

Indexes	0	1	2	3	4
0	1	2	3	4	5
1	11	12	13	14	15
2	21	22	23	24	25

Initial matrix, Mas_1 array.

Indexes	0	1	2
0	1	11	21
1	2	12	22
2	3	13	23
3	4	14	24
4	5	15	25

Transposable matrix, Mas_2 array.

Fig. 151. Initial and Transposable Matrices.

In this case, the problem resolves itself to rewriting the values of the first matrix to the second one according to the rules of matrix transposition, namely rewrite the elements values of the first matrix columns to the rows of the desired matrix. The solution of matrix transposition problem is represented in the matrix.mq4 expert:

```
//--------------------------------------------------------------------
// matrix.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
int start()                            // Special function start()
  {
   int Mas_1[3][5]={1,2,3,4,5,  11,12,13,14,15,  21,22,23,24,25};
   int Mas_2[5][3];
   int R0= ArrayRange( Mas_1, 0);      // Number of elements in first dim.
   int R1= ArrayRange( Mas_1, 1);      // Number of elements in second dim.

   for(int i=0; i
```

Two arrays are opened in the start() function of the expert. The Mas_1 array has 3 rows containing 5 elements each and the MAS_2 array has 5 rows containing 3 elements each. The rewriting of the values itself is performed in the following entry:

```
            Mas_2[[j][i] = Mas_1[i][j];     // Matrix transposition
```

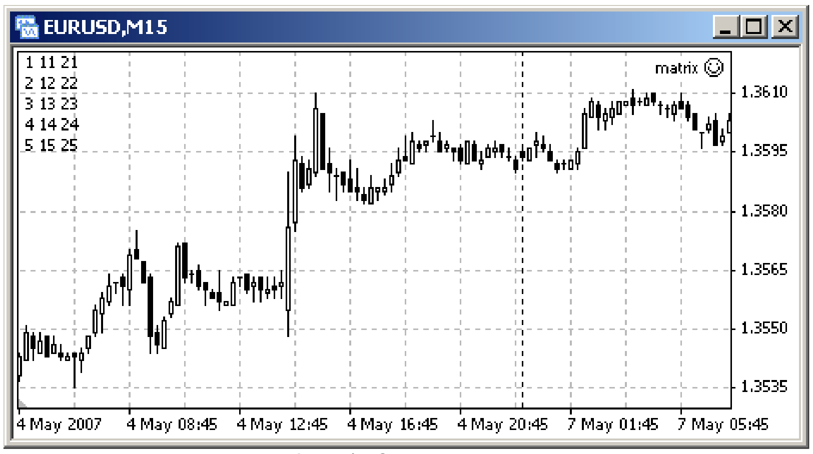
In order to calculate the runtime environment (the number of iterations) of two embedded cycle operators, you should know the values of the elements of each array. In this example the constant values 3 and 5 could be used. However, this way of program designing is incorrect. In general, a program can contain a huge code wherein the calling to the same values is performed in many parts of it. A program must be designed so that the modifications could be made in one place, if necessary, and in all the other necessary parts they would be calculated. In this case, only the entries that open and initialize the arrays should be modified if it is necessary to change the size of the arrays, so there is no need to modify the code in the other parts.

To determine the number of elements of the first and second dimensions of the Mas_1 array the following calculations are performed:

```
      int R0 =  ArrayRange( Mas_1, 0);    // Number of elements in first dim.
      int R1 =  ArrayRange( Mas_1, 1);    // Number of elements in second dim.
```

Note that the 0 value is used for the first dimension and the 1 value is used for the second one. The calculated values of the R0 and R1 variables are used to determine the number of iterations in the "for" cycles.

The received values of the Mas_2 array elements are displayed on the screen using the Comment() function.



```
1 11 21
2 12 22
3 13 23
4 14 24
5 15 25
```

Fig. 152.Result of matrix.mq4 operation.

Functions for Working with Arrays

Functions	Short Description
ArrayBsearch	It returns the index of the first found element in the first dimension of array. If the element with the specified value ia absent then the function will return the index of the nearest (by value) element.
ArrayCopy	It copies one array to another. The arrays must have the same type. The arrays of the double[], int[], datetime[], color[], and bool[] types can be copied as the arrays of the same type. Returns the number of copied elements.
ArrayCopyRates	It copies the bar data to the two-dimensional array of the RateInfo[][6] kind and returns the number of copied bars. Otherwise it returns -1, if the operation fails.
ArrayCopySeries	It copies a timeseries array to the user-defined array and returns the number of copied elements.
ArrayDimension	It returns rank of a multy-dimensional array.
ArrayGetAsSeries	It returns TRUE if the array is arranged as a timeseries (elements of the array are indexed from the last element to the first one), otherwise returns FALSE.
ArrayInitialize	It sets a single value to all elements of the array. Returns the number of initialized elements.
ArrayIsSeries	It returns TRUE if the checked array is a timeseries (Time[], Open[],Close[],High[],Low[] of Volume[]), otherwise returns FALSE.
ArrayMaximum	It searches an elemnet with the maximum value. The function returns the location of the maximum element in the array.
ArrayMinimum	It searches an element with the minimum value. The function returns the location of the minimum element in the array.
ArrayRange	It returns the number of elements in the specified dimension of the array. The size of the dimension is greater

	than the biggest index by 1, because the indexes are starting with zero.
ArrayResize	It sets a new size of the first dimension of the array. Returns the number of all elements that array contains after its rank has been changed if the function ran successfully, otherwise returns -1 and the size of the array is not changed.
ArraySetAsSeries	It sets the direction of indexing in the array.
ArraySize	It returns the number of elements in an array.
ArraySort	It sorts numeric arrays by their first dimension. The timeseries arrays cannot be sorted.

Functions to Access Timeseries

Functions	Summary Info
iBars	It returns the number of bars of the specified chart.
iBarShift	It searches a bar by time. The function returns the shift of bar that has the specified time. If the bar for the specified time is absent ("hole" in the history) then the function returns -1 depending on the *exact* parameter or the shift of the nearest bar.
iClose	The function returns the close price of the bar specified with the *shift* parameter from the corresponding chart (*symbol*, *timeframe*). It returns 0, if an error occurs.
iHigh	It returns the maximum price value of the bar specified with the *shift* parameter from the corresponding chart (*symbol*, *timeframe*). It returns 0, if an error occurs.
iHighest	It returns the index of the maximum found value (shift relatively to the current bar).
iLow	It returns the minimum price value of the bar specified with the *shift* parameter from the corresponding chart (*symbol*, *timeframe*). It returns 0, if an error occurs.
iLowest	It returns the index of the minimum found value (shift relatively to the current bar).
iOpen	It returns the open price value of the bar specified with the *shift* parameter from the corresponding chart (*symbol*, *timeframe*). It returns 0, if an error occurs.
iTime	It returns opening time of the bar specified with the *shift* parameter from the corresponding chart (*symbol*, *timeframe*). It returns 0, if an error occurs.
iVolume	It returns tick volume value of the bar specified with the *shift* parameter from the corresponding chart (*symbol*, *timeframe*). It returns 0, if an error occurs.

To get the detailed information about these and other functions, please refer to the Documentation at MQL4.community, at MetaQuotes Software Corp. website or at the "Help" section of MetaEditor.

← File Operations Mathematical Functions →

Mathematical Functions

Mathematical and trigonometric functions are included in MQL4. There are no difficulties in using most of them. For example, the MathMax() function returns the maximum value of two values specified in the list of parameters of the function. The usage of other functions claims certain attention and thoughtfulness. Let's examine one of the functions of this kind.

MathFloor() Function

```
    double MathFloor(double x)
```

The function returns a numeric value that corresponds the greatest integer that is less or equal to x.

Parameters:

x - numeric value.

Note that a value returned by the function is the real number (double type), at the same time, it is written that the function returns an integer. It should be realized that the function returns a real number that has all the positions equal to zero after the decimal point. For example, the MathFloor() function may return 37.0 (positive number of the double type) or -4.0 (negative number of the double type).

The description says also that the function returns the maximum of possible numbers that is less than a specified one. For example, if the value of the given x parameter is 13.5 then the maximum number that has zeros after the decimal point is 13.0. Or if the -13.5 negative value is specified in the function, then the maximum smallest integer is equal to -14.0. In such a manner, modification of the sign of the passed to the function value leads to the different results, namely the received values are not equal to each other in absolute magnitude.

Usage of such functions is very convenient, in some cases. Let's examine the fragment of the lots amount calculation for new orders as an example:

```
   int Percent    =30;                                 // % of free margin
   double Free    =AccountFreeMargin();                // Free margin
   double One_Lot=MarketInfo(Symb,MODE_MARGINREQUIRED);// 1 lot price
   double Step    =MarketInfo(Symb,MODE_LOTSTEP);      // Size step changed

   double Lots_New=MathFloor(Free*Percent100One_LotStep)*Step;
```

The value of the Percent parameter is set by user. In this case, the user specified 30% of free margin for new orders. According to the rules that are specified by the dealing center, the correctly calculated amount of lots must be divisible by the minimum step of lots changing size (Step). The values of free margin (Free) and 1 lot price (One_Lot) are necessary for calculation too.

Let's examine the logic of reasoning of the programmer that compiled the expression to calculate the required amount of lots Lots_New for new orders. Let's use the numeric values of the variables for better visualization. Let Free=5000.0, One_Lot=1360.0 (In most of the dealing centers the cost of 1 lot of currency pair is in proportion to the cost of the symbol), Step=0.1. In this case, the expression to calculate Lots_New can be written as follows:

Lots_New = MathFloor(5000.0*30/100/1360.0/0.1)*0.1;

The 5000.0*30/100 expression is the value of money the user laid out to open a new order. In this case, the price of a new order may reach the 1500.0. Spending all these money you can open one new order that has the 1500.0 / 1360.0 = 1.102941 amount of lots. However, the dealing center won't accept the order with this amount of lots, because the minimum Step=0.1 (in the most dealing centers). To calculate the desired amount of lots you should throw away all the "needless" digits in the decimal part and replace them with zeros.

In order to do it you can use the considered mathematical function:

Lots_New = MathFloor(1.102941/0.1)*0.1;

The calculated value of MathFloor(1.102941/0.1) will be 11.0, and the calculated value of the Lots_New variable will be 1.1 lots. This value meets the requirements of the dealing center and so can be used as the declared amount of lots in new orders.

Mathematical Functions

Function	Summary Info
MathAbs	The function returns the absolute value (in absolute magnitude) of a given number.

MathArccos	The function returns the arccosine value of *x* in the 0 to п radians range. If *x* is lesser than -1 or greater than 1, the function returns NaN (undefined value).
MathArcsin	The function returns arcsine value of *x* in the -п/2 to п/2 radians range. If *x* i less than -1 or greater than 1, the function returns NaN (undefined value).
MathArctan	The function returns arctangent of *x*. If *x* is equal to 0, then the function returns 0. MathArctan returns the value in the -п/2 to п/2 radians range.
MathCeil	The function returns the numeric value that is the smallest integer bigger or equal to *x*.
MathCos	The function returns the cosine of the angle.
MathExp	The function returns the value of **e** to the power of *d*. At overflow, the function returns INF (infinity), and it returns 0 at underflow.
MathFloor	The function returns the numeric value representing the largest integer that is less than or equal to *x*.
MathLog	The function returns the natural logarithm of *x* if successful. If *x* is negative, this function returns NaN (undefined value). If *x* is 0, it returns INF (infinity).
MathMax	The function returns the maximum value of two numeric values.
MathMin	The function returns the minimum value of two numeric values.
MathMod	The function returns the floating-point remainder of division of two numbers. The MathMod function calculates the floating-point remainder *f* of *x / y* such that *x = i * y + f* , where *i* is an integer, *f* has the same sign as *x*, and the absolute value of *f* is less than the absolute value of *y*.
MathPow	Returns the value of the base expression raised to the specified power (exponent value).
MathRand	The function returns a pseudorandom integer within the range of 0 to 32767. The MathSrand function must be used to seed the pseudorandom-number generator before calling MathRand.
MathRound	The function returns value rounded to the nearest integer of the specified numeric value.
MathSin	The function returns the sine of the specified angle.
MathSqrt	The function returns the square root of *x*. If *x* is negative, MathSqrt returns an indefinite (same as a quiet NaN).
MathSrand	The function sets the starting point for generating a series of pseudorandom integers.
MathTan	MathTan returns the tangent of *x*. If *x* is greater than or equal to 263, or less than or equal to -263, a loss of significance in the result occurs. In this case, the function returns an indefinite value.

To get the detailed information about these and other functions, please refer to the Documentation at MQL4.community, at MetaQuotes Software Corp. website or at the "Help" section of MetaEditor.

← Arrays and Timeseries GlobalVariables →

GlobalVariable Functions

Many functions for working with global variables of the client terminal are described in the GlobalVariables section. In the previous section, we also mentioned that a correctly designed program had to delete its global variables when it finished running. No GVs must remain in the client terminal after all programs have been exited.

One or more GVs may remain in the terminal when debugging programs using the global variables of the client terminal. In this case, a programmer must delete GVs manually before the next start of a debugged program. To automate this process, you can create a script that deletes all the global variables of the client terminal.

GlobalVariablesDeleteAll() Function

```
int GlobalVariablesDeleteAll(string prefix_name=NULL)
```

It deletes global variables. If a prefix is not specified for the name, then all available global variables will be deleted. Otherwise, it will delete only variables of the names that start with the specified prefix. The function returns the number of variables deleted.

Parameters:

prefix_name - prefix for the names of global variables to be deleted.

Below is an example of a simple script, deleteall.mq4, that deletes all global variables of the client terminal.

```
//--------------------------------------------------------------------
// deleteall.mq4
// The program is intended to be used as an example in MQL4 Tutorial.
//--------------------------------------------------------------------
int start()                                 // Special start() function
  {
   GlobalVariablesDeleteAll();              // Deleting of all GVs
   PlaySound("W2.wav");                     // Sound
   return;                                  // Exit
  }
//--------------------------------------------------------------------
```

The script can be started only if no program using GVs is being executed in the client terminal. Otherwise, script running may break the logic of other executed programs that may result in uncontrolled actions. After the script execution the window of global variables (F3) of the client terminal will become empty:

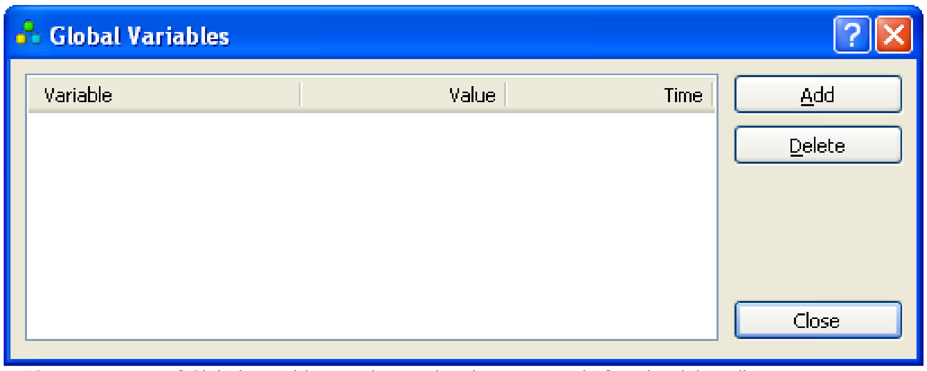


Fig. 153. Appearance of Global Variables window in the client terminal after the deleteall.mq4 script is executed.

Functions for Working with Global Variables

Function	Summary Info
GlobalVariableCheck	It returns True if a GV is available. Otherwise, it returns FALSE.
GlobalVariableDel	It deletes a global variable. It returns TRUE if a variable has been successfully deleted. Otherwise, it returns FALSE.
GlobalVariableGet	It returns a value of a global variable, or 0, if an error occurs.
GlobalVariableName	The function returns the name of a global variable according to its index number in the global variables list.

GlobalVariableSet	It sets a new value to a global variable. The system will create a new variable if there are no any already created. The time of the last access will be returned, if the function has run successfully. Otherwise, it returns 0.
GlobalVariableSetOnCondition	It sets a new value to a global variable if its current value is equal to the value of the *check_value* third parameter. The function will generate the ERR_GLOBAL_VARIABLE_NOT_FOUND (4058) error and will return false if a variable does not exist. It returns TRUE if the function has successfully run. Otherwise, it returns FALSE.
GlobalVariablesDeleteAll	It deletes global variables. If the prefix for the name is not specified, then all global variables will be deleted. Otherwise, it deletes only those of the names starting with the specified prefix. The function returns the number of variables deleted.
GlobalVariablesTotal	The function returns the total number of global variables.

To get the detailed information about these and other functions, please refer to the Documentation at MQL4.community, at MetaQuotes Software Corp. website, or at the "Help" section of MetaEditor.

← Mathematical Functions Custom Indicators →

Custom Indicators

The functions of custom indicators allow you to adjust the necessary settings to make a convenient representation of an indicator. Let's consider some of them (see also Creation of Custom Indicators).

SetIndexBuffer() Function

```
bool SetIndexBuffer(int index, double array[])
```

The function joins an array-variable that was declared on the global level with a predestined custom indicator buffer. The amount of buffers that are necessary for indicator computation is set using the IndicatorBuffers() function and cannot exceed 8. In case of successful connection, TRUE is returned, otherwise - FALSE. In order to get the detailed information about an error you should call the GetLastError() function.

Parameters:

index - index number of a line (from 0 to 7 values are possible);

array[] - a name of the array that is connected with a calculation buffer.

SetIndexStyle() Function

```
void SetIndexStyle(int index, int type, int style=EMPTY, int width=EMPTY, color clr=CLR_NONE)
```

The function sets a new type, style, width and color of a specified line of indicator.

Parameters:

index - index number of a line (from 0 to 7 values are possible);

type - indicator line type. Can be one of the listed types of indicator lines (see Styles of Indicator Lines Displaying);

style - line style. Used for the lines of 1 pixel width. Can be one of the line styles specified in the Styles of Indicator Lines Displaying appendix. The EMPTY value specifies that the style won't be changed;

width - line width; permissible values are - 1,2,3,4,5; the EMPTY value specifies that the width won't be changed;

clr - line color. The empty value CLR_NONE means that the color won't be changed.

SetIndexLabel() Function

```
void SetIndexLabel(int index, string text)
```

The function allows to set an indicator line name to be displayed in DataWindow and in the pop-up tip.

Parameters:

index - index number of a line (from 0 to 7 values are possible);

text - a text describing an indicator line. NULL means that the value of a line is not shown in DataWindow.

The example of the simple indicator showing High line (indicatorstyle.mq4) that uses the functions described above:

```
//--------------------------------------------------------------------
// indicatorstyle.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
#property indicator_chart_window    // Indic. is drawn in the main window
#property indicator_buffers 1       // Amount of buffers
#property indicator_color1 Blue     // Color of the first line

double Buf_0[];                     // Indicator array opening
//--------------------------------------------------------------- 2 --
int init()                          // Special function init()
  {
   SetIndexBuffer(0,Buf_0);         // Assigning the array to the buffer
   SetIndexStyle (0,DRAW_LINE,STYLE_SOLID,2);// Line style
   SetIndexLabel (0,"High Line");
   return;                          // Exit spec. function init()
  }
//--------------------------------------------------------------- 3 --
int start()                         // Special function start()
  {
   int i,                           // Bar index
   Counted_bars;                    // Amount of calculated bars
   Counted_bars=IndicatorCounted(); // Amount of calculated bars
   i=Bars-Counted_bars-1;           // Index of the first uncounted
   while(i>=0)                      // Cycle for the uncounted bars
     {
      Buf_0[i]=High[i];             // value 0 of the buffer on ith bar
      i--;                          // Index calculation for the next bar
     }
   return;                          // Exit spec. function start()
  }
//--------------------------------------------------------------- 4 --
```

Block 1-2 contains general settings of an indicator. It is specified with the #property command that the indicator must be drawn in the main window, the indicator uses one buffer, the color of the indicator line is blue. One buffer array is opened in block 1-2, as well.

The functions specified above are used in block 2-3. The entry:

```
     SetIndexBuffer(0,Buf_0);           // Assigning the array to the buffer
```

assigns the buffer named Buf_0 to the buffer with the 0 index. The entry:

```
     SetIndexStyle (0,DRAW_LINE,STYLE_SOLID,2);// Line style
```

determines the style of indicator line that has the 0 index. The DRAW_LINE constant indicates that the line displayed is simple, the STYLE_SOLID constant indicates that the line is solid, and 2 specifies the width of the line. The entry:

```
     SetIndexLabel (0,"High Line");
```

assigns the name to the indicator line with the 0 index. The specified name can be seen in the DataWindow and in the balloon tip in the window of financial instrument (fig. 810_3). The naming of the lines is convenient, if the window contains a number of indicator lines; sometimes it is the only way to distinguish one line from another. Block 3-4 performs the calculation of the values of indicator array elements for the line that is used to display the maximum values of bars (the sequence of indicator arrays calculation is described in the Creation of Custom Indicators section in detail).

If the indicator is displayed in a separate window, then the horizontal levels can be displayed in this window too.

SetLevelValue() Function

```
  void SetLevelValue(int level, double value)
```

Sets the value for the specified horizontal level of the indicator that is displayed in a separate window.

Parameters:

level - level number (0-31).

value - a value for the specified level.

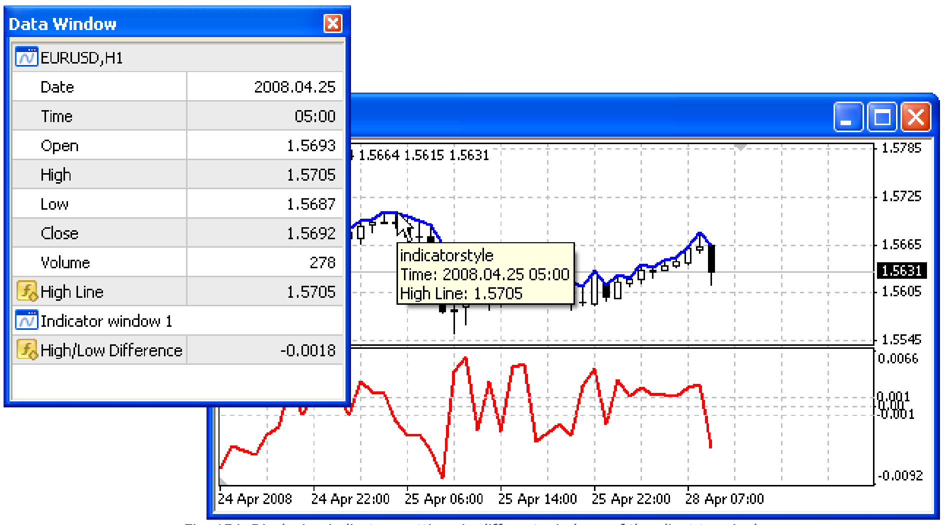
The use of horizontal levels can be very convenient, if you need to detect visually whether the indicator line is above or below the specified values. The simple indicator that calculates the difference between the maximum and the minimum price of the bar is shown below. The market events are interesting for the user (tentatively in this example) if the indicator line is above the bar by 0.001 or below the bar by -0.001. The example of the indicator that displays the difference between High and Low (linelevel.mq4):

```
//--------------------------------------------------------------------
// linelevel.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
#property indicator_separate_window // Indic. is drawn in a sep. window
#property indicator_buffers 1       // Amount of buffers
#property indicator_color1 Red      // Line color

double Buf_0[];                     // Indicator array opening
//--------------------------------------------------------------- 2 --
int init()                          // Special init() function
  {
   SetIndexBuffer(0,Buf_0);         // Assigning the array to the buffer
   SetIndexStyle (0,DRAW_LINE,STYLE_SOLID,2);// Line style
   SetIndexLabel (0,"High/Low Difference");
   SetLevelValue (0, 0.0010);       // The horizontal line level is set
   SetLevelValue (1,-0.0010);       // Another level is set
   return;                          // Exit from spec.init() function
  }
//--------------------------------------------------------------- 3 --
int start()                         // Special start() function
  {
   int i,                           // Bar index
       Counted_bars;                // Amount of calculated bars

   Counted_bars=IndicatorCounted(); // Amount of calculated bars
   i=Bars-Counted_bars-1;           // Index of the first uncounted

   while(i>=0)                      // Cycle for the uncounted bars
     {
      Buf_0[i]=High[i]-Low[i];      // 0 value of the buffer on bar i
      if(Open[i]>Close[i])          // if the candle is black..
         Buf_0[i]=-Buf_0[i];        // .. then reverse the value
      i--;                          // Index calculation for the next bar
     }
   return;                          // Exit from spec.start() function
  }
//--------------------------------------------------------------- 4 --
```

The considered function is used in block 2-3 in the indicator. In this case, two horizontal levels are specified. The first value in the list of parameters is the number of the horizontal level, the second one is the specified value of the level:

```
      SetLevelValue (0, 0.0010);         // The level of the horiz. line is set
      SetLevelValue (1,-0.0010);         // Another level is set
```

The parameters of indicatorstyle.mq4 and linelevel.mq4 indicators set in such a way are displayed in the window of the financial instrument and in the DataWindow.



Data Window	
EURUSD,H1	
Date	2008.04.25
Time	05:00
Open	1.5693
High	1.5705
Low	1.5687
Close	1.5692
Volume	278
High Line	1.5705
Indicator window 1	
High/Low Difference	-0.0018

Fig. 154. Displaying indicators settings in different windows of the client terminal.

Two windows - the DataWindow and the financial instrument window are shown in the fig. 154. You can see the entry containing "High Line" text and the 1.3641 value in the DataWindow. The same inscriptions are shown in the lower entry of the pop-up tip. The said entry is displayed in the DataWindow the whole time the indicator runs, the name of the line is not changed at that, but the value in the right part of the entry depends on the cursor position in the financial instrument window. The name of the line that is shown in the pop-up tip corresponds to the indicator line the cursor is pointed at.

The subwindow of the linelevel.mq4 indicator contains the horizontal lines that are placed according to the user-set values. If you move the cursor over the red indicator line then the name of this line, in this case the "Difference between High and Low", can be seen in the pop-up tip, the value at the cursor point can be seen to the right of the name.

Functions for Adjusting Custom Indicators

Function	Summary Info
IndicatorBuffers	It arranges the memory of buffers that are used for calculating a custom indicator. The amount of buffers cannot exceed 8 and must be less than the value specified in the *#property indicator_buffers* command. If the custom indicator needs more buffers for calculation, you should use this function to specify the whole number of buffers.
IndicatorCounted	The function returns an amount of bars that were not changed since the last indicator launch. The majority of bars do not require recalculation.
IndicatorDigits	It sets the accuracy (the number of symbols after the decimal point) for visualization of indicator values.
IndicatorShortName	It sets a "short" name to an indicator to be displayed in the indicator's subwindow and in the DataWindow.
SetIndexArrow	It sets a symbol to an indicator line that has the DRAW_ARROW style.
SetIndexBuffer	It joins an array-variable that is declared on the global level with a specified buffer of a custom indicator.
SetIndexDrawBegin	It sets an index number from the beginning of data to the bar the drawing of a specified indicator should begin with.
SetIndexEmptyValue	It sets an empty value for indicator line. Empty values are not drawn and are not shown in the DataWindow.
SetIndexLabel	It sets a name of an indicator line to display the information in the DataWindow and in the balloon tip.
SetIndexShift	It sets a shift for an indicator line relatively to the chart beginning. The positive value will shift a line to the right, the negative value - to the left. I.e. the value calculated on the current bar is drawn with the specified, relative to the current bar, shift.
SetIndexStyle	It sets a new type, style, width and color for a specified indicator line (see Styles of Indicator Lines Displaying).
SetLevelStyle	It sets a new style, width and color for horizontal levels of an indicator that is displayed in a separate window (see Styles of Indicator Lines Displaying).
SetLevelValue	It sets a value for specified horizontal level of an indicator that is displayed in a separate window.

To get the detailed information about these and other functions, please refer to the Documentation at MQL4.community, at MetaQuotes Software Corp. website or at the "Help" section of the MetaEditor.

← GlobalVariables Account Information →

Account Information

The functions of the client terminal and status checking functions are convenient to be applied for program restriction when it is distributed on the commercial basis, for example. Let's examine the solution of the problem below:

Problem 39. Create a protection code of the program distributed on the commercial basis, which meets the following requirements:

- the program must require a password to be executed on real accounts of individual clients;
- the password is not required to execute the program on real accounts of corporate clients;
- no limitations are provided for the execution of the program on demo accounts.

This example contains a correct problem definition. For successful commercial distribution, you provide your potential consumers with a program that can be fully tested on a demo account. So the user can weigh all the advantages of the program and come to the decision of buying it.

The application of a common password is incorrect - the unauthorized distribution of the program is possible, in this case. So the individual password for the user must be dependent on the real account number. There is no need to use the password for the corporate clients (if the dealing center bought the license for all its traders) - the program must self-identify the server that the client terminal is connected to. And, if it is the server of the corporate client then every user must be able to work without obstacles.

The solution of the Problem 39 limiting the rights of program use can be the following (check.mq4 EA):

```
//--------------------------------------------------------------------
// check.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
extern int Parol=12345;
//--------------------------------------------------------------- 2 --
int start()                            // Special function 'start'
  {
   if(Check()==false)                  // If the using conditions do not..
      return;                          // ..meet the requirements, then exit

   // The main code of the program must be specified here
   Alert("Program execution");       // Alert example

   return;                             // Exit start()
  }
//--------------------------------------------------------------- 3 --
bool Check()                           // User-defined function of..
  {                                    // .. use conditions checking
   if (IsDemo()==true)                 // If it is a demo account, then..
      return(true);                    // .. there are no other limitations
   if (AccountCompany()=="SuperBank")  // The password is not needed
      return(true);                    // ..for corporate clients
   int Key=AccountNumber()*2+1000001;  // Calculating key
   if (Parol==Key)                     // If the password is correct, then..
      return(true);                    // ..allow the real account to trade
   Alert("Wrong password. EA does not work.");
   return(false);                      // Exit user-defined function
  }
//--------------------------------------------------------------- 4 --
```

The necessary checking is performed in top entries of the special start() function, in this example (block 2-3):

```
   if(Check()==false)                  // If the using conditions..
```

If the Check() function returns false as the result of checking, then the control is passed to the return operator and the special start() function finishes its working. The main code of the program is located directly after this checking. The Check() function will return true if checking is successful, then the main code will be executed.

The checking is performed according to three criteria in the user-defined function Check():
- is the account a demo one;
- does the server belong to a corporate client;
- is the password valid for the real account.

The function IsDemo() is used for checking according to the first criterion.

Function IsDemo()

```
bool IsDemo()
```

The function returns TRUE, if the program is working with a demo account. Otherwise, it returns FALSE.

If the IsDemo() function returns true, then the user is working with the demo-account. This means that there is no need for further checking (because use of program with a demo-account is free for everyone). The Check() function finishes its working and returns true, in this case:

```
   if (IsDemo() == true)                // If it is a demo account, then..
      return(true);                     // .. there are no other limitations
```

But if the IsDemo() function returns false value then the user is working with the real account. It is necessary to find out if the user has enough rights, in this case. The AccountCompany() function is used in this example to check the corporate clients.

Function AccountCompany()

```
string AccountCompany()
```

The function returns the name of the company the current account is registered at.

If the checking resulted in:

```
   if (AccountCompany() == "SuperBank")// The password is not needed..
      return(true);                     // ..for corporate clients
```

determining that the name of the company corresponds with the one specified in the program, then the Check() function will finish its working and return true - the checking will be finished successfully. If it is turned out that the client is connected to the other company (that is not a corporate client), then there is no need to find out if he\she has an individual license.

The entry:

```
   int Key = AccountNumber()*2+1000001;// Calculating key
```

puts the algorithm for calculation of a key for any account in the program. This example contains the simple method. The programmer, as he\she thinks fit, can insert more complex method for key calculation. Anyway, the algorithm must consider an account number that is available for the program by using the AccountNumber() function.

Function AccountNumber()

```
int AccountNumber()
```

The function returns the number of the current account.

The password previously calculated using the same algorithm is passed to the user. If the checking has resulted in finding out that that the password and the key match each other, then the Check() function finishes its working and returns true value:

```
   if (Parol == Key)                    // If the password is correct, then..
      return(true);                     // ..allow the real account to trade
```

If none of the checking procedures finished successfully then the user can not trade using a real account. In this case, the Check() function finishes its working and returns false value after making the appropriate announcement. In such a manner the unauthorized use of the program attempt is suppressed.

Functions Returning Client Terminal Information

Function	Summary Info
TerminalCompany	It returns the name of the company that owns the client terminal.
TerminalName	It returns the name of the client terminal.
TerminalPath	It returns the directory the client terminal is launched from.

Functions Detecting the Current Status of the Client Terminal Including the Environment Status of the Executed MQL4 Program

Function	Short description
GetLastError	The function returns the last error code, following which the value of the special last_error variable that contains the last error code is set to zero. So the next calling of the GetLastError function will return 0 value.
IsConnected	It returns the status of the connection used for data transferring between the client terminal and the server. TRUE - the connection to the server is established, FALSE - there is no connection to the server or the connection is lost.
IsDemo	It returns TRUE if a program works with a demo-account. Otherwise, it returns FALSE.
IsDllsAllowed	It returns TRUE if DLL calling functions are permitted for an EA. Otherwise, it returns FALSE.
IsExpertEnabled	It returns TRUE if the EA launching is permitted in the client terminal. Otherwise, it returns FALSE.
IsLibrariesAllowed	It returns TRUE if an EA is able to declare a library function. Otherwise, it returns FALSE.
IsOptimization	It returns TRUE if an EA is working in the test optimizing mode. Otherwise, it returns FALSE.
IsStopped	It returns TRUE if a program (EA or script) received a command to exit working. Otherwise, it returns FALSE.
IsTesting	It returns TRUE if an EA is working in the testing mode. Otherwise, it returns FALSE.
IsTradeAllowed	It returns TRUE if an EA is allowed to trade and the traffic is free for trading. Otherwise, it returns FALSE.
IsTradeContextBusy	It returns TRUE if the traffic for trading is busy. Otherwise, it returns FALSE.
IsVisualMode	It returns TRUE if an EA is tested in the visualization mode. Otherwise, it returns FALSE.
UninitializeReason	It returns the code of the reason for operation termination of an EA, a custom indicator or a script. Returned values can be one of the deinitialization codes. This function can be called in the init() function to analyze the reasons for deinitialization of the previous launch, as well.

Functions Accessing to the Information about the Active Account

Function	Short description
AccountBalance	It returns a value of the balance of the active account (the total amount of money on the account).
AccountCredit	It returns a credit value of the active account.
AccountCompany	It returns the name of a brokerage company the current account is registered at.
AccountCurrency	It returns the currency name of the current account.
AccountEquity	It returns the equity value of the current account. The equity calculation depends on server settings.
AccountFreeMargin	It returns the value of free margin permitted for opened orders of a current account.
AccountFreeMarginCheck	It returns the value of free margin that will remain after the specified position has been opened on the current account.
AccountFreeMarginMode	The calculation of free margin amount mode for opened orders of the current account.
AccountLeverage	It returns the leverage value of the current account.
AccountMargin	It returns the amount of margin used to maintain the open positions on the current account.
AccountName	It returns the user name of the current account.
AccountNumber	It returns the number of the current account.
AccountProfit	It returns the profit value of the current account calculated in the base currency.
AccountServer	It returns the name of the active server.
AccountStopoutLevel	It returns the value of the level that is used to identify the StopOut status.
AccountStopoutMode	It returns the mode of StopOut level calculation.

To get the detailed description of these and other functions, please refer to the Documentation at MQL4.community, at MetaQuotes Software Corp. website or at the "Help" section of MetaEditor.

← Custom Indicators Trade Functions →

Trade Functions

All trade functions can be divided into two groups - functions that form trade orders and functions that return some order characterizing values. MQL4 has only five functions that form and send trade orders to a server:

- OrderSend() - market order opening and pending order placing;
- OrderClose() - market order closing;
- OrderCloseBy() - closing of opposite market orders;
- OrderDelete() - deleting pending orders;
- OrderModify() - modifying all types of orders.

The order of using functions listed above is described in the Programming of Trade Operations section in details. All the other functions do not form trade orders but their usage is often necessary. For example, it is sometimes necessary to close orders in some priority sequence. To do it, you should analyze the characteristics of every order in the program, namely - order type, lots amount, stop-orders location, etc. Let's examine some functions that allow to get the information about an order.

OrderTotal() Function

```
int OrdersTotal()
```

The function returns the total number of opened and pending orders.

OrderTakeProfit() Function

```
double OrderTakeProfit()
```

The function returns the value of the declared price when the profit level (take profit) of the current selected order is reached. The order must be previously selected using the OrderSelect() function.

OrderProfit() Function

```
double OrderProfit()
```

Returns the net profit value (without regard to swaps and commissions) of the selected order. It is the unrealized profit for the opened orders and fixed profit for the closed orders. The order must be previously selected using the OrderSelect() function.

OrderLots() Function

```
double OrderLots()
```

Returns the amount of lots of a selected order. The order must be previously selected using the OrderSelect() function.

The fragment of the program that calculates declared close price TakeProfit, order profit and the amount of lots is shown below:

```
for (int i=0; i<OrdersTotal(); i++)          // For all orders
  {
   if((OrderSelect(i,SELECT_BY_POS)==true)   // If next exists
     {
      double TP =     OrderTakeProfit();     // TakeProfit of order
      double Profit= OrderProfit();          // Order profit
      double Lots   = OrderLots();           // Amount of lots
      //......TP and profit values usage in the program.....
     }
  }                                          // End of the cycle body
```

It is clear that every considered function (OrderTakeProfit (), OrderProfit() и OrderLots()) does not have any adjustable parameters, i.e. denotation of, for example, number of the order, to return the value corresponding with the characteristics of this individual order is not involved.

To calculate the characteristics of an individual order (declared price of one of the stop-orders, order profit and amount of lots in this context) you should select the necessary order first; this will inform the program about the order to perform calculations with. In order to do it you should execute the OrderSelect() function before starting the calculations (see Closing and Deleting Orders). The trade functions executed after that will return the values that correspond with the selected order characteristics.

The correct evaluation of one or another order characteristics by the programmer is no little significant. For example, when solving the problem of order closing sequence, you should set a calculation criterion for which order should be closed earlier and which one - afterwards. Let's take a look to the simple task.

Problem 40. Two Buy orders are currently opened on a single symbol. The first one is opened at the price of 1.2000 at 0.5 lot, the second one is opened at the price of 1.3000 at 1 lot. The current price is 1.3008. The trade criterion for Buy orders closing has triggered. It is necessary to make a right decision, namely to decide which order should be closed as the first and which one as the second.

Obviously, the profit from the first order makes 108 points, whereas that of the second one is 8 points. Although the first order is opened at a smaller amount of lots, it has the larger profit than the second one, namely the profit of the first order is $540 and the profit of the second order is $80. Closing the first order may seem to be preferable, at the first sight, because it has greater profit. However, it is a misthought. It is necessary to examine the possible case scenario to make a correct decision.

The order closing sequence would not matter, if the price were known not to change during the period of orders closing. However, the price may change during the execution of the instruction to close one of the orders. So the order that can bring more loss, at a negative scenario, should be closed first. If the price sinks one point down, the profit of the first order will decrease by $5, whereas that of the second one will do by $10. Obviously, the second order would bring more loss, so it should be closed first. In such a way, the amount of lots has the dominant significance when deciding about order closing sequence. Profitable case scenario cannot be considered here, because trading develops with the trade criteria in the program, and this time the criterion of closing Buy orders has triggered.

You should consider the other order characteristics if it is necessary to choose between two orders with the same amount of lots. For example, you can consider the distance between the current price and the StopLoss value of each order. At the same time, you should analyze which of the orders would bring more loss, in case of fast price moving. The answer is obvious, as well: the one (from both orders that are opened at the same amount of lots) that has its StopLoss level further from the current price.

Thus you can analyze the priority and all the other parameters of orders and compile the priority-oriented list of criteria to consider with when making the decision about closing orders. It is not difficult to identify the criteria that should not be considered. It is open price (and the related profit from order), for example. The amount of money that the trader have at the moment is shown in the Equity column of the client terminal. The source of this value is not important at that, neither it is a result of loss from one ore more orders, nor it is a result of a profit.

All the necessary characteristics of an order can be received using the corresponding trade functions:

Trade Functions

Function	Summary Info
Execution Errors	Any trade operation (OrderSend, OrderClose, OrderCloseBy, OrderDelete or OrderModify functions) can unsuccessfully end for a score of reasons and return either the negative ticket number or FALSE. You can find out the reason of failure by using the GetLastError function. Every error should be processed in its own way. The table below contains the general recommendations.
OrderClose	It closes position. It returns TRUE, if the function has ended successfully. It returns FALSE, if the function fails to end.
OrderCloseBy	It closes one open position with the other that is opened in the opposite direction for the same symbol. It returns TRUE. if the function has ended successfully. It returns FALSE, if the function fails to end.
OrderClosePrice	It returns the close price of the selected order.
OrderCloseTime	It returns the time of closing for the selected order.

OrderComment	It returns the comment of the selected order.
OrderCommission	It returns the calculated commission value of the selected order.
OrderDelete	It deletes the previously placed pending order. It returns TRUE, if the function has ended successfully. It returns FALSE, if the function fails to end.
OrderExpiration	It returns the date of expiration of the selected pending order.
OrderLots	It returns the amount of lots of the selected order.
OrderMagicNumber	It returns the identifying ("magic") number of the selected order.
OrderModify	It modifies the parameters of the previously opened orders and pending orders. It returns TRUE if the function has ended successfully. It returns FALSE, if the function fails to end.
OrderOpenPrice	It returns the open price of the selected order.
OrderOpenTime	It returns the opening time of the selected order.
OrderPrint	It enters the order information to the journal.
OrderProfit	It returns the net profit (without regard to swaps and commissions) of the selected order. It is the unrealized profit for the opened orders and fixed profit for the closed orders.
OrderSelect	The function chooses the order to work with subsequently. It returns TRUE if the function has ended successfully. It returns FALSE, if the function fails to end.
OrderSend	The main function for opening orders and placing pending orders. It returns the number of the ticket that was assigned to the order by the trade server, or -1, in case of failing to end the operation.
OrdersHistoryTotal	It returns the number of closed positions and deleted orders in the history of the current account, loaded to the client terminal.
OrderStopLoss	It returns close price of the position when it reaches the loss level (stop loss) of the currently selected order.
OrdersTotal	It returns the total number of open and pending orders.
OrderSwap	It returns the swap value of the currently selected order.
OrderSymbol	It returns the symbol name for the currently selected order.
OrderTakeProfit	It returns the close price when it reaches the profit level (take profit) of the currently selected order.
OrderTicket	It returns the ticket number of the currently selected order.
OrderType	It returns the operation type of the currently selected order.

To get the detailed description of this and other functions, you should refer to the Documentation at MQL4.community, at MetaQuotes Software Corp. website or at the "Help" section of the MetaEditor.

← Account Information Creation of a Normal Program →

Creation of a Normal Program

As a rule, after having coded a number of simple application programs in MQL4, a programmer goes to a more complex project - to creation of a convenient program for practical use. Simple programs, in some cases, do not satisfy the needs of a trading programmer for at least two reasons:

1. The functional boundedness of simple programs does not allow them to provide a trader with all necessary information and trading management tools, which doesn't make the use of such programs efficient enough.

2. The code imperfection of simple programs makes difficult their further development aimed at expanded services.

In this section, we represent one of the possible alternatives of realizing a trading Expert Advisor that can be considered as a basis for your own projects.

- Structure of a Normal Program
  The availability of many user-defined functions in a program allows you to create powerful and flexible algorithms to process information. Compiler directive #include allows you to use your function (once written and debugged) in other programs. In this manner, you can create your own libraries or use open-source developments of other programmers.

- Order Accounting
  The section considers an example of user-defined function Terminal() that is realized in a separate include file with the extension .mqh. Such files are connected to the program code during compilation using the directive #include.

- Data Function
  An example of one more user-defined function that helps to organize the output of text information about the current work of an EA. This function allows you to abandon the function Comment() to display the text in the chart window. The function is realized as an indicator in a separate subwindow of the chart window.

- Event Tracking Function
  A trader cannot always notice all the events during trading. The program written in MQL4 allows you to detect the changes in any trading conditions or situations. user-defined function Events() connects to the EA using the directive #include and applies calls to another include function, Inform().

- Volume Defining Function
  The calculation of the volume of a position to be opened is one of the tasks of equity/risk management. The user-defined function Lot() is a small example used for these purposes.

- Trading Criteria Defining Function
  The most important part of any trading is detection of the times of entering the market and that of closing a position. The creation of trading rules or criteria is the core of any Expert Advisor. User-defined function Criterion() is connected using the directive #include. It shows how an EA can decide on the basis of the indicator values about whether the current situation complies with one or another criterion.

- Trade Functions
  The current situation has been analyzed with the function Criterion(), so now we have to make trades: open, close, modify or delete a pending order. All these operations can be put in separate user-defined functions: Trade(), Close_All() and Open_Ord(). The protecting stop orders are moved using the user-defined function Tral_Stop().

- Error Processing Function
  Error control is an integral part of an Expert Advisor. This is you who determines how to process the message about the busy trade context, about no prices for the requested symbol, etc. In some cases, it is sufficient to display a message about the error. In other cases, it would be reasonable to try and repeat the trade request after a certain amount of time. It is necessary to determine how one or another error will be processed. The user-defined function Errors() shown in this section processes

errors using the selection operator switch().

← Trade Functions Structure of a Normal Program →

Structure of a Normal Program

The outstanding feature of a normal program is its structure that allows your to easily use some user-defined functions or other. For convenience, user-defined functions are usually formed as include files (.mqh) that are stored in the folder ***Terminal_directory\experts\include***.

As a rule, a normal program contains all three special functions that call the necessary user-defined functions for execution. User-defined functions, in their turn, can call other user-defined functions for execution, each of them having its own functionally bounded purpose.

An Expert Advisor can contain user-defined functions with the most diverse properties. Some functions, for example, are intended for tracking events and convenient data output, other functions are used to form trade requests, the third functions are intended for various calculations, for example, for defining of trading criteria, calculation of order costs, etc. The decision on what functions to use in a program depends on the EA purpose and on what service the program is intended to provide the user with. In Fig. 155 below, you can see a block diagram of a normal trading EA built on a small set of user-defined functions.

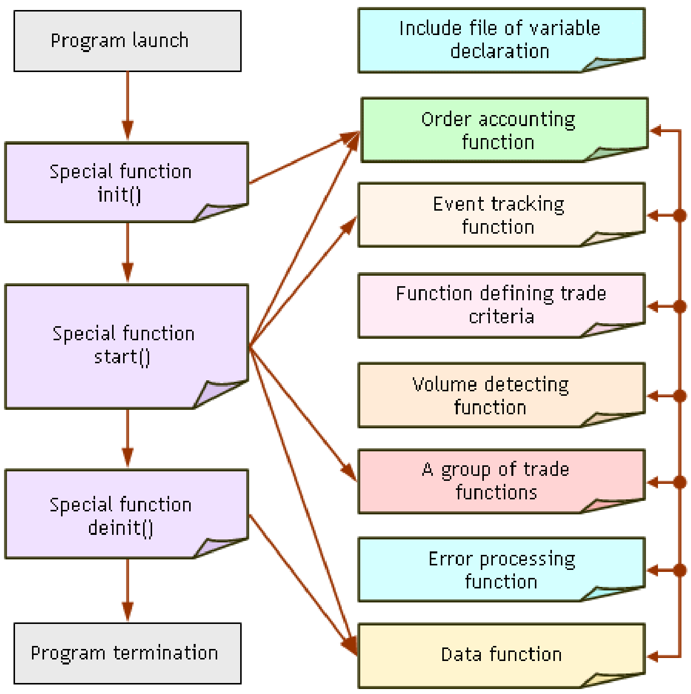

Program launch
Special function init()
Special function start()
Special function deinit()
Program termination
Include file of variable declaration
Order accounting function
Event tracking function
Function defining trade criteria
Volume detecting function
A group of trade functions
Error processing function
Data function

Fig. 155. Block diagram of a normal program (Expert Advisor).

The arrows in the diagram show the relationships between functions. For example, the order accounting function in the EA is called from the special functions init() and start(); it can also be called from any other location in the program. In the right part of the diagram, the connecting arrows are shown that symbolize the possibility to call one user-defined function from another. For example, the function defining trading criteria cannot be called from any special function, but it can be called for execution by a trade function. The data function is called from the special function deinit() and, if necessary, from other functions, for example, from the error processing function, trade functions, or from the event tracking function. The include file of variable declaration cannot be called from any function, since the code contained in it is not the description of a function that can be called and executed. This file contains the lines of global variable declaration, so it is just a part of an EA. In order to understand the way the different parts of an EA interrelate in, let's consider the order of creation and use of include files.

Using Include Files

If an application program contains a great variety of program lines, it is sometimes difficult to debug it - the programmer has to scroll the program text many times in order to make changes in the code in one or another location. In such cases, for convenience, the program can be divided into several fragments, each formed as a separated include file. Include files can contain any code fragments to be used in the program. Each function used in the program is usually formed as an include file. If several functions are logically interconnected, one include file may contain the description of several user-defined functions.

In the section Information about an Account, we consider an example of the protection code that prevents from unauthorized use of business programs. In the Expert Advisor check.mq4, the corresponding part of the program code is represented as the user-defined function Check(), the description of which is directly contained in the source code of the EA. In the EA usualexpert.mq4 (see Fig. 91.3), this function is used, too. However, in this case, the function is formed as the include file Check.mqh.

User-Defined Function Check()

```
bool Check()
```

The function returns TRUE, if the conditions of using the application program are complied with. Otherwise, it returns FALSE.

The conditions of using the program are considered as complied with, if:

- the program is used on a demo account;
- the account is opened with SuperBank;

- the user has set the correct value of the external variable Parol when working on a real account.

The include file Check.mqh containing the description of the user-defined function Check():

```
//--------------------------------------------------------------------------------
// Check.mqh
// The program is intended to be used as an example in MQL4 Tutorial.
//----------------------------------------------------------------------------- 1 --
// The function checking legality of the program used
// Inputs:
// - global variable 'Parol'
// - local constant "SuperBank"
// Returned values:
// true  - if the conditions of use are met
// false - if the conditions of use are violated
//----------------------------------------------------------------------------- 2 --
extern int Parol=12345; // Password to work on a real account
//----------------------------------------------------------------------------- 3 --
bool Check()                           // User-defined unction
  {
   if (IsDemo()==true)                 // If it is a demo account, then..
      return(true);                    // .. there are no other limitations
   if (AccountCompany()=="SuperBank")  // For corporate clients..
      return(true);                    // ..no password is required
   int Key=AccountNumber()*2+1000001;  // Calculate the key
   if (Parol==Key)                     // If the password is true, then..
      return(true);                    // ..allow the user to work on a real account
   Inform(14);                         // Message about unauthorized use
   return(false);                      // Exit the user-defined function
  }
//----------------------------------------------------------------------------- 4 --
```

It is easy to note that the name of the include file is the same as the name of the function it contains. This is not required by the rules of MQL4. Generally, the name of an include file is not required to be the same as the name of the function it contains. It becomes even clearer, if we consider that one file may contain the description of several functions or of a program fragment that is not a function at all. However, the practice of giving an include file the same name as that of the contained function is feasible. This considerably facilitates the work of a programmer: using the file names, he or she will easily know what functions there are in the folder ***Terminal_directory|experts|include***. In order to include the program fragment contained in the file, you should use the directive #include.

Directive #include

```
#include <File name>
#include "File name"
```

The directive #include can be specified in any location in the program. However, all included fragments are placed at the beginning of the source text file. Preprocessor will replace the line #include <file_name> (or #include "file_name") with the contents of the file of the given name.

Angle brackets mean that the file will be taken from the standard directory ***Terminal_directory|experts|include*** (the current directory is not viewed). If the file name is enclosed in quotation marks, it will be searched in the current directory, namely, in the directory that contains the basic file of the source text (the standard directory is not viewed).

Below is a normal Expert Advisor, usualexpert.mq4. All include files are placed in the head part of the program.

```
//--------------------------------------------------------------------------------------
// usualexpert.mq4
// The code should be used for educational purpose only.
//----------------------------------------------------------------------------------- 1 --
#property copyright "Copyright © Book, 2007"
#property link      "http://AutoGraf.dp.ua"
//----------------------------------------------------------------------------------- 2 --
#include <stdlib.mqh>
#include <stderror.mqh>
#include <WinUser32.mqh>
//----------------------------------------------------------------------------------- 3 --
#include <Variables.mqh>   // Description of variables
#include <Check.mqh>       // Checking legality of programs used
#include <Terminal.mqh>    // Order accounting
#include <Events.mqh>      // Event tracking function
#include <Inform.mqh>      // Data function
#include <Trade.mqh>       // Trade function
#include <Open_Ord.mqh>    // Opening one order of the preset type
#include <Close_All.mqh>   // Closing all orders of the preset type
#include <Tral_Stop.mqh>   // StopLoss modification for all orders of the preset type
#include <Lot.mqh>         // Calculation of the amount of lots
#include <Criterion.mqh>   // Trading criteria
#include <Errors.mqh>      // Error processing function
//----------------------------------------------------------------------------------- 4 --
int init()                             // Special function 'init'
  {
   Level_old=MarketInfo(Symbol(),MODE_STOPLEVEL );//Min. distance
   Terminal();                         // Order accounting function
   return;                             // Exit init()
  }
//--------------------------------------------------------------------------------- 5 --
int start()                            // Special function 'start'
  {
   if(Check()==false)                  // If the usage conditions..
      return;                          // ..are not met, then exit
   PlaySound("tick.wav");              // At every tick
   Terminal();                         // Order accounting function
   Events();                           // Information about events
   Trade(Criterion());                 // Trade function
   Inform(0);                          // To change the color of objects
   return;                             // Exit start()
  }
//--------------------------------------------------------------------------------- 6 --
int deinit()                           // Special function deinit()
  {
   Inform(-1);                         // To delete objects
   return;                             // Exit deinit()
  }
//--------------------------------------------------------------------------------- 7 --
```

In block 2-3, we included into the program standard files stdlib.mqh, stderror.mqh and WinUser32.mqh using the directive #include. It is not always necessary to use these files in the program. For example, file stderror.mqh contains the definition of standard constants used in error processing. If error processing is not provided in the program (i.e., these constants are not used), then there is no need to include this file into the source text. At the same time, it is usually necessary to include these files in a normal program.

In block 3-4, the program includes some files containing the description of user-defined functions. Using the directive #include in the line below:

```
#include <Check.mqh>       // Checking legality of programs used
```

the source text of the program includes the user-defined function Check(). Programmer can see the source code of the EA (in this case, usualexpert.mq4) as it is represented above. However, the source text of the program is modified at compilation, namely, each line containing the directive #include is replaced in the program with the text contained in the file of the given name. Thus, an executable file .ex4 is created on the basis of the full code of the Expert Advisor, in which each line #include<File name> (or #include "File name") is replaced with the corresponding code fragment.

An example of an include file containing a program fragment that is not the function description is file Variables.mqh. This file is included in the program text in the line:

```
#include <Variables.mqh>   // Description of variables
```

and contains the description of global variables used by different user-defined functions.

```
//--------------------------------------------------------------------------
// Variables.mqh
// The code should be used for educational purpose only.
//--------------------------------------------------------------------- 1 --
// Description of global variables
extern double Lots     = 0.0;// Amount of lots
extern int Percent     = 0;  // Allocated funds percentage
extern int StopLoss    =100; // StopLoss for new orders (in points)
extern int TakeProfit  =40;  // TakeProfit for new orders (in points)
extern int TralingStop=100; // TralingStop for market orders (in points)
//--------------------------------------------------------------------- 2 --
int
   Level_new,            // New value of the minimum distance
   Level_old,            // Previous value of the minimum distance
   Mas_Tip[6];           // Order type array
                         // [] order type: 0=B,1=S,2=BL,3=SL,4=BS,5=SS
//--------------------------------------------------------------------- 3 --
double
   Lots_New,             // Amount of lots for new orders
   Mas_Ord_New[31][9],   // Current order array ..
   Mas_Ord_Old[31][9];   // .. old order array
                         // 1st index = order number in the list
                         // [][0] cannot be detected
                         // [][1] order open price (abs. price value)
                         // [][2] StopLoss of the order (abs. price value)
                         // [][3] TakeProfit of the order (abs. price value)
                         // [][4] order number
                         // [][5] order volume (abs. price value)
                         // [][6] order type 0=B,1=S,2=BL,3=SL,4=BS,5=SS
                         // [][7] Order magic number
                         // [][8] 0/1 the fact of availability of comments
//--------------------------------------------------------------------- 4 --
```

According to the rules of MQL4, any variable (including global ones) must be declared before the first reference to this variable. For this reason, the file Variables.mqh is included into the program and located above the files of functions that use the values of the variables specified in this file. For the same reason, all global variables are placed in this file.

In some (rare) cases, it is technically possible to declare a global variable in an include file that describes the function, in which the value of this variable is first used in the program. In such cases, it is necessary to keep the file including order. I.e., the program line containing the directive #include that includes the file (with the declaration of a global variable) into the program must be located above in the text than the lines that include other files using the value of this global variable.

In other cases, it is even technically impossible to do this. For example, if we have two include files, each of them using two global variables, one of which is declared in one file and the other is declared in the other file, then we will get an error when compiling such a program, since, regardless the file including order, one of the variables is used before it is declared in the program. This is why it is a usual practice in a normal program to declare all global variables, without any exceptions, in one file that is included in the program before other files containing the description of user-defined functions.

In block 1-2 of the include file Variables.mqh, all external variables are specified, the values of which determine the amount of lots for new orders, the free margin percentage allocated for new orders, the requested prices for stop orders of the market orders to be opened, as well as the distance of TralingStop for modification of StopLoss order. In blocks 2-4, other global variables are given, the sense of which will become clear upon considering the corresponding user-defined functions. The subsections of this section represent include files, each containing the description of the user-defined function of the same name.

← Creation of a Normal Program Order Accounting →

Order Accounting

We mentioned above that there were no strict rules for making program algorithms. At the same time, the overwhelming majority of algorithms imply making one's trading decisions according to the current status of the orders available. In some cases, for example, opening a market order needs no other market orders available as of the moment of the trade. In other cases, no stop orders available on the market order can be a necessary condition for placing of a pending order. We also know some algorithms that imply placing two differently directed pending orders.

In order to have met the requirements of one or another tactic or strategy by the moment of decision making, you should know about the current status - what market and pending orders are available and what characteristics do they have? You may use one of the two possible solutions.

According to the first solution, the necessary program code fragment (in which the orders are analyzed) is written directly in the location in the program, where the available set of orders and their characteristics should be found out. This solution is technically feasible, but it turns out to be inefficient, if you want to make changes in the algorithm. In this case, the programmer has to analyzes all locations in the program where order statuses are analyzed, and make changes in each location. Another, more effective solution is to create a universal order accounting function once and use it every time, when you want to update the information about the current order statuses. On the one hand, this solution allows you to reduce the program code. On the other hand, it allows the programmer to use this ready-made function when coding other programs.

In order to create an order accounting function correctly, you should first decide what parameters must be accounted. In most cases, the values of the following parameters are used in making trade decisions:

- total amount of orders;
- the amount of orders of each type (for example, the amount of Buy orders, SellStop orders, or BuyLimit orders, etc.);
- all characteristics of each order (ticket, StopLoss and TakeProfit levels, volume in lots, etc.).

The above information must be available to other functions, namely to those, in which this information is processed. For this reason, all parameters that characterize order statuses are the values of global arrays. Totally, three arrays are provided for order accounting:

- the array of current orders, Mas_Ord_New, that contains information about all characteristics of all market and pending orders available at the current moment, namely, within the period of the last execution of the function;
- the array of old orders, Mas_Ord_Old, that contains information about all characteristics of all market and pending orders available at the moment of the preceding execution of the function;
- the array Mas_Tip, the values of which are the amounts of orders of different types (at the current moment).

Arrays Mas_Ord_New and Mas_Ord_Old are similar and equidimensional; the difference between them is that the former one reflects the current status of orders, whereas the latter one shows the preceding status. Let's give a closer consideration to the values contained in the elements of those arrays.

Table 4. Correspondence of the elements of arrays Mas_Ord_New and Mas_Ord_Old with order characteristics.

	Not defined	Open Price	StopLoss	TakeProfit	Order Number	Volume, in lots	Order Type	Magic Number	Comment
Indexes	0	1	2	3	4	5	6	7	8
0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	1.2583	1.2600	1.2550	123456.0	1.4	1.0	1177102416.0	1.0
2	0.0	1.2450	1.2580	1.2415	123458.0	2.5	2.0	1177103358.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
. . .	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

The first index of the array (lines) defines the number of the order in the array. The characteristics of the first order detected (among all opened market orders and placed pending orders) are placed in the first line of the array, those of the second order detected will be placed in the second line, etc. The array size for the first index is equal to 31, thus, the array is intended to store information about 30 orders at the most, if they are simultaneously available on one trading account. If the trading strategy allows the availability of more than thirty orders simultaneously, you should specify the corresponding value for the first index when you declare the array. (In most cases, the value of 30 considerably exceeds the real need usually ranging from 2 to 10-15 orders. We use the value of 30 in this example, because we suppose that the function can be used for very unusual trading strategies, as well).

The second index in the array (columns) represents order characteristics. Each element of the array with the second index equal to 1 contains the value of the order open price, with index 2 - they contain the value of StopLoss order, 3 - TakeProfit, etc. (see Table 4). Array element with index [0][0] has a value that is equal to the total amount of orders available in the array. No array elements having the first or the second indexes equal to 0 are used (except element having index [0][0]).

Table 4 represents the status of an array that contains information about two orders that are simultaneously available in trading at a certain moment. The array element Mas_Ord_New[0][0] has the value of 2.0 - the total amount of orders is two. The elements in the first line of the array contain the values of the characteristics of the market order Sell (Mas_Ord_New[1][6] = 1.0, see Types of Trades) opened for 1.4 lot (Mas_Ord_New[1][5] =1.4) and having the number 123456 (Mas_Ord_New[1][4] =123456.0). The value of the element Mas_Ord_New[1][8] =1.0 means that this order has non-empty comment field. In the second line of the array, the values that characterize the second order are contained. Particularly, the array element Mas_Ord_New[2][6] has the value of 2.0, it means that it is BuyLimit.

Array Mas_Tip represents the amount of orders of each type. The values of this array indexes are assigned to the types of trades (see Types of Trades). This means that the element of array Mas_Tip with index 0 contains the amount of market orders of the Buy type simultaneously available in trading, index 1 means the amount of Sell orders, index 2 means that of BuyLimit orders, etc. For the situation shown in Table 4, the elements of array Mas_Tip will have the following values:

Table 5. Correspondence of the elements of array Mas_Tip with the amount of orders of different types.

	Buy	Sell	BuyLimit	SellLimit	BuyStop	SellStop
Index	0	1	2	3	4	5
Value	0	1	1	0	0	0

In this case, the values of the elements of array Mas_Tip imply the following: Mas_Tip[1] equal to 1 means that there is one Sell order traded; Mas_Tip[2] equal to 1 means that there is one pending order BuyLimit in trading. Other elements of the array have zero values - this means that there are no orders of those types in trading. If there are several orders of the same type simultaneously available in trading, the corresponding element of the array will have the value equal to the amount of such orders. For example, if there are three orders BuyStop in trading, the element Mas_Tip[4] will have the value of 3.

The order accounting function Terminal() suggested here is formed as include file Terminal.mqh.

User-Defined Function Terminal()

```
int Terminal()
```

The function accounts market and pending orders. The execution of the function results in changing of the values of the following global arrays:

- Mas_Ord_New - the array of characteristics of orders available as of the moment of the function execution;
- Mas_Ord_Old - the array of characteristics of orders available as of the moment of the preceding execution of the function;
- Mas_Tip - the array of the total amount of orders of all types.

Include file Terminal.mqh that contains the description of the order accounting function Terminal():

```
//--------------------------------------------------------------------
// Terminal.mqh
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
// Order accounting function
// Global variables:
// Mas_Ord_New[31][9]   // The latest known orders array
// Mas_Ord_Old[31][9]   // The preceding (old) orders array
                        // 1st index = order number
                        // [][0] not defined
                        // [][1] order open price (abs. price value)
                        // [][2] StopLoss of the order (abs. price value)
                        // [][3] TakeProfit of the order (abs. price value)
                        // [][4] order number
                        // [][5] order volume in lots (abs. price value)
                        // [][6] order type 0=B,1=S,2=BL,3=SL,4=BS,5=SS
                        // [][7] order magic number
                        // [][8] 0/1 comment availability
// Mas_Tip[6]           // Array of the amount of orders of all types
                        // [] order type: 0=B,1=S,2=BL,3=SL,4=BS,5=SS
//--------------------------------------------------------------- 2 --
int Terminal()
  {
   int Qnt=0;                          // Orders counter

//--------------------------------------------------------------- 3 --
   ArrayCopy(Mas_Ord_Old, Mas_Ord_New);// Saves the preceding history
   Qnt=0;                              // Zeroize orders counter
   ArrayInitialize(Mas_Ord_New,0);     // Zeroize the array
   ArrayInitialize(Mas_Tip,    0);     // Zeroize the array
//--------------------------------------------------------------- 4 --
   for(int i=0; i<OrdersTotal(); i++) // For market and pending orders
     {
      if((OrderSelect(i,SELECT_BY_POS)==true)     //If there is the next one
      && (OrderSymbol()==Symbol()))               //.. and our currency pair
        {
         //------------------------------------------------------ 5 --
         Qnt++;                                   // Amount of orders
         Mas_Ord_New[Qnt][1]=OrderOpenPrice();    // Order open price
         Mas_Ord_New[Qnt][2]=OrderStopLoss();     // SL price
         Mas_Ord_New[Qnt][3]=OrderTakeProfit();   // TP price
         Mas_Ord_New[Qnt][4]=OrderTicket();       // Order number
         Mas_Ord_New[Qnt][5]=OrderLots();         // Amount of lots
         Mas_Tip[OrderType()]++;                  // Amount of orders of the type
         Mas_Ord_New[Qnt][6]=OrderType();         // Order type
         Mas_Ord_New[Qnt][7]=OrderMagicNumber();  // Magic number
         if (OrderComment()=="")
            Mas_Ord_New[Qnt][8]=0;                // If there is no comment
         else
            Mas_Ord_New[Qnt][8]=1;                // If there is a comment
         //------------------------------------------------------ 6 --
        }
     }
   Mas_Ord_New[0][0]=Qnt;                         // Amount of orders
//--------------------------------------------------------------- 7 --
   return;
```

```
    }
  //------------------------------------------------------------------------------ 8 --
```

In block 1-2, we give a comment describing the global arrays used in the function. The global arrays are declared in an include file Variables.mqh. In block 3-4, the content of the array Mas_Ord_New is copied to the array Mas_Ord_Old. Thus, the previously known status of orders is stored and can be used further in the program. Then the values of elements of arrays Mas_Ord_New and Mas_Tip showing the new status of orders has been zeroized before the data are updated in block 4-7.

Block 4-7 contains the cycle 'for', in which all market and pending orders are checked one by one for the symbol, to the window of which the EA is attached. The orders are selected using the function OrderSelect() according to the parameter MODE_TRADES set by default. In block 5-6, all required characteristics are calculated for the selected orders, the obtained data are stored in the array of new orders, Mas_Ord_New. At the same time, the amount of orders of all types is calculated, the obtained values being assigned to the corresponding elements of array Mas_Tip. Upon ending of the cycle, the total amount of orders for the symbol is assigned to the element Mas_Ord_New[0][0].

It must be separately noted that closed market orders and deleted pending orders (the execution of the function OrderSelect() with the parameter MODE_HISTORY) are not analyzed. As a rule, the information about closed and deleted orders is not used in trading EAs. The information about closed and deleted orders represent the history of a trading account. This information can be used, for example, to build diagrams that represent the history of capital invested and real trading results. However, it cannot be useful in any way for making new trade decisions. Technically, this part of orders can be accounted in a similar way. However, it is a separate task that has no relation to trading as such.

The events related to orders are analyzed in a program on the basis of comparison of data available in the arrays considered above. For example, if array Mas_Ord_Old contains information about a pending order numbered as 246810, while array Mas_Ord_New contains the data about the same order 246810, but the order is of another type, it means that a pending order has been modified into a market one. It is also necessary to analyze orders when making trades (to be considered later).

Before the function Terminal() is executed for the very first time, arrays Mas_Ord_Old and Mas_Ord_New are empty, i.e., each element of both arrays has zero value. This means that, after the first execution of the function, the array Mas_Ord_Old in the line:

```
     ArrayCopy(Mas_Ord_Old, Mas_Ord_New);// Store the preceding history
```

inherits "zero" status from the array Mas_Ord_New, which results in appearance of false event alerts at the execution of the event tracking function. In order to prevent this, the very first execution of the function Terminal() is performed at the stage of initialization, and events are not processed after this execution of the function (see the function init() in Expert Advisor usualexpert.mq4).

← Structure of a Normal Program Data Function →

Data Function

A normal Expert Advisor used in our practical work favorably compares to its simple analogs, because it provides a trader with high-quality information support.

During trading, the situation changes all the time, various events take place. To make proper decisions, a trader must be fully informed. For this purpose, various functions are used in Expert Advisors. These functions are aimed at informing the user about a certain set of facts and processes.

In simple Expert Advisors, as a rule, this task is performed by the standard function Comment() that displays the preset text in the upper left corner of the symbol window. This information output method is not quite comfortable, since the text can often be superimposed onto the price chart. So this method can only be used in a limited amount of cases, to display short messages.

We will consider here a conceptually different method of displaying information - the entire message is shown in a separate window, whereas graphical objects are used to form message texts. The use of graphical objects produces a tangible advantage, since you can move objects (unlike texts shown using Comment()) creating, in this manner, your message history.

A separate subwindow for displaying of information is created using a custom indicator adjusted correspondingly. The only purpose of this indicator is the creation of that subwindow, so no calculations are performed in the indicator, nor indicating lines are built in it. The code of the indicator Inform.mq4 may look as follows:

```
//--------------------------------------------------------------------
// Inform.mq4
// The code should be used for educational purpose only.
//--------------------------------------------------------------------
#property indicator_separate_window // Separate indicator window
//--------------------------------------------------------------------
int start()                         // Special function start()
  {
  }
//--------------------------------------------------------------------
```

Generally, a programmer may add into the indicator his or her desired code and contribute to its properties. For example, you can display indicating lines in a certain part of the indicator subwindow. In the above example, a simple code of the indicator is given, in the window of which graphical objects will be shown.

User-Defined Function Inform()

```
int Inform(int Mess_Number,int Number=0,double Value=0.0)
```

The function displays in the subwindow of the indicator Inform.mq4 messages created using graphical objects. The function controls the position of graphical objects in the indicator subwindow: every new message is displayed in the lower part of the window (lower line) and colored in the desired color, the previously shown messages being moved into the upper part of the window (one line upward). If no new messages are displayed within 15 seconds, all previously displayed messages in the window will be colored in gray (in order not to create diversions for the user) when the program refers to the function.

Parameters:

Mess_Number - message number that can take the following values:

- (zero) 0 - no message is displayed, this mode is used to reset the timer;
- (minus one) -1 - all graphical objects created by the function will be deleted;
- (one or more) - the number of the message to be displayed in the indicator subwindow;

Number - integer used in some messages;

Value - real number used in some messages.

The function Inform() that creates graphical objects, like other functions in a normal EA,is formed as an include file, Inform.mqh:

```
//----------------------------------------------------------------------------
// Inform.mqh
// The code should be used for educational purpose only.
//----------------------------------------------------------------------- 1 --
// Function that displays graphical messages on the screen.
//----------------------------------------------------------------------- 2 --
int Inform(int Mess_Number, int Number=0, double Value=0.0)
  {
   // int    Mess_Number               // Message number
   // int    Number                    // Integer to be passed
   // double Value                     // Real number to be passed
   int    Win_ind;                     // Indicator window number
   string Graf_Text;                   // Message line
   color  Color_GT;                    // Color of the message line
   static int    Time_Mess;            // Last publication time of the message
   static int    Nom_Mess_Graf;        // Graphical messages counter
   static string Name_Grf_Txt[30];     // Array of graphical message names
//----------------------------------------------------------------------- 3 --
```

```
   Win_ind= WindowFind("inform");      // Searching for indicator window number
   if (Win_ind<0)return;               // If there is no such a window, leave
//--------------------------------------------------------------------- 4 --
   if (Mess_Number==0)                 // This happens at every tick
     {
      if (Time_Mess==0) return;        // If it is gray already
      if (GetTickCount()-Time_Mess>15000)// The color has become updated within 15 sec
        {
         for(int i=0;i<=29; i++)       // Color lines with gray
            ObjectSet( Name_Grf_Txt[i], OBJPROP_COLOR, Gray);
         Time_Mess=0;                  // Flag: All lines are gray
         WindowRedraw();               // Redrawing objects
        }
      return;                          // Exit the function
     }
//--------------------------------------------------------------------- 5 --
   if (Mess_Number==-1)                // This happens at deinit()
     {
      for(i=0; i<=29; i++)             // By object indexes
         ObjectDelete(Name_Grf_Txt[i]);// Deletion of object
      return;                          // Exit the function
     }
//--------------------------------------------------------------------- 6 --
   Nom_Mess_Graf++;                    // Graphical messages counter
   Time_Mess=GetTickCount();           // Last publication time
   Color_GT=Lime;
//--------------------------------------------------------------------- 7 --
   switch(Mess_Number)                 // Going to message
     {
      case 1:
         Graf_Text="Closed order Buy "+ Number;
         PlaySound("Close_order.wav");                         break;
      case 2:
         Graf_Text="Closed order Sell "+ Number;
         PlaySound("Close_order.wav");                         break;
      case 3:
         Graf_Text="Deleted pending order "+ Number;
         PlaySound("Close_order.wav");                         break;
      case 4:
         Graf_Text="Opened order Buy "+ Number;
         PlaySound("Ok.wav");                                  break;
      case 5:
         Graf_Text="Opened order Sell "+ Number;
         PlaySound("Ok.wav");                                  break;
      case 6:
         Graf_Text="Placed pending order "+ Number;
         PlaySound("Ok.wav");                                  break;
      case 7:
         Graf_Text="Order "+Number+" modified into the market one";
         PlaySound("Transform.wav");                           break;
      case 8:
         Graf_Text="Reopened order "+ Number;                  break;
         PlaySound("Bulk.wav");
      case 9:
         Graf_Text="Partly closed order "+ Number;
         PlaySound("Close_order.wav");                         break;
      case 10:
         Graf_Text="New minimum distance: "+ Number;
         PlaySound("Inform.wav");                              break;
      case 11:
         Graf_Text=" Not enough money for "+
         DoubleToStr(Value,2) + " lots";
         Color_GT=Red;
         PlaySound("Oops.wav");                                break;
      case 12:
         Graf_Text="Trying to close order "+ Number;
         PlaySound("expert.wav");                              break;
      case 13:
         if (Number>0)
            Graf_Text="Trying to open order Sell..";
         else
            Graf_Text="Trying to open order Buy..";
         PlaySound("expert.wav");                              break;
      case 14:
         Graf_Text="Invalid password. EA doesn't function.";
         Color_GT=Red;
         PlaySound("Oops.wav");                                break;
      case 15:
         switch(Number)                 // Going to the error number
           {
            case 2:   Graf_Text="Common error.";                    break;
            case 129: Graf_Text="Wrong price. ";                    break;
            case 135: Graf_Text="Price changed. ";                  break;
            case 136: Graf_Text="No prices. Awaiting a new tick.."; break;
```

```
               case 146: Graf_Text="Trading subsystem is busy";        break;
               case 5 :  Graf_Text="Old version of the terminal.";     break;
               case 64:  Graf_Text="Account is blocked.";              break;
               case 133: Graf_Text="Trading is prohibited";            break;
               default:  Graf_Text="Occurred error " + Number;//Other errors
              }
            Color_GT=Red;
            PlaySound("Error.wav");                                    break;
         case 16:
            Graf_Text="Expert Advisor works only for EURUSD";
            Color_GT=Red;
            PlaySound("Oops.wav");                                     break;
         default:
            Graf_Text="default "+ Mess_Number;
            Color_GT=Red;
            PlaySound("Bzrrr.wav");
        }
//----------------------------------------------------------------------- 8 --
   ObjectDelete(Name_Grf_Txt[29]);       // Deleting 29th (upper) object
   for(i=29; i>=1; i--)                  // Cycle for array indexes ..
     {                                  // .. of graphical objects
      Name_Grf_Txt[i]=Name_Grf_Txt[i-1];// Raising objects:
      ObjectSet( Name_Grf_Txt[i], OBJPROP_YDISTANCE, 2+15*i);
     }
   Name_Grf_Txt[0]="Inform_"+Nom_Mess_Graf+"_"+Symbol(); // Object name
   ObjectCreate (Name_Grf_Txt[0],OBJ_LABEL, Win_ind,0,0);// Creating
   ObjectSet    (Name_Grf_Txt[0],OBJPROP_CORNER, 3    );  // Corner
   ObjectSet    (Name_Grf_Txt[0],OBJPROP_XDISTANCE, 450);// Axis X
   ObjectSet    (Name_Grf_Txt[0],OBJPROP_YDISTANCE, 2);   // Axis Y
   // Текстовое описание объекта
   ObjectSetText(Name_Grf_Txt[0],Graf_Text,10,"Courier New",Color_GT);
   WindowRedraw();                       // Redrawing all objects
   return;
  }
//----------------------------------------------------------------------- 9 --
```

In block 2-3, the variables used in the function are described. To store the names of graphical objects, the array Name_Grf_Txt is used. According to the method accepted in the function, the program creates a new graphical object for each new message. The total amount of objects is 30, each object represents a text entry located in one line. In case of large screen resolution, the amount of lines to be shown can be increased by increasing of the amount of objects to be created.

In block 3-4, the number of the subwindow of the indicator "Inform" is calculated for messages to be shown in. If the indicator is not attached, the function stops its operations. If there is no window, the messages cannot be displayed, but this does not affect the workability of the EA - all other functions will work in their normal modes, trades will be performed, as well.

In block 4-5, the message colors are analyzed. The function with the parameter Mess_Number=0 is called in the Expert Advisor at every tick (see the function start() in Expert Advisor usualexpert.mq4). If all objects are gray in the preceding history, the function ends its operations. However, if the value of the variable Time_Mess is nonzero, the properties of all objects are changed, namely, all objects will be colored in gray.

If (block 5-6) the parameter Mess_Number=-1 is specified in the function call, all objects previously created at the execution of the function are deleted. This may be necessary when the EA is detached from the symbol window. In this case, according to the common rule, each application program must delete all objects it created during execution (see the function deinit() in the Expert Advisor usualexpert.mq4).

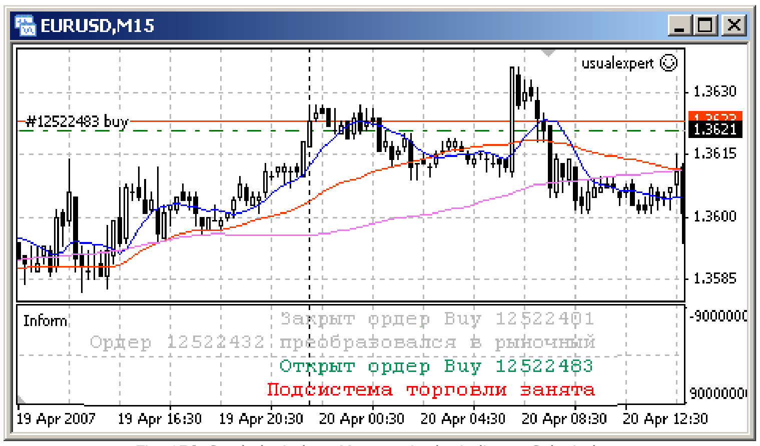
If the control in the program is passed to block 6-7, this means that it is necessary to create a new graphical object with required properties and place it in the lower part of the indicator subwindow (in the lower line; here, the term of "line" is conditional; in fact, the location of graphical objects is determined by the preset coordinates). Each newly created object has its unique name. To create object names, we use the historical number of the message, this is why the messages counter is placed in block 6-7 (at a later stage, the value of the variable Nom_Mess_Graf is used to form a unique name, block 8-9). It is here where the last publication time is accounted and the basic color is established for new messages (green).

Block 7-8 consists of the operator 'switch', inside which the control is passed according to the value of the parameter Mess_Number specified in the function call. In each 'case' of this operator of the variable Graf_Text, a new value is assigned that is the content of the message to be displayed. A special color is set for some messages, for example, red for important messages. All messages are accompanied with sounds that are the execution of the standard function PlaySound() (see Wave Files).

The creation of a new graphical object and replacing of the existing ones are performed in block 8-9. The amount of objects is limited, so one object (the oldest one) is deleted every time when a new message is published. All other existing objects are moved one line upwards. Objects are moved by changing their properties - vertical coordinates. The horizontal coordinates of objects remain unchanged.

After all necessary preparations have been made (all objects are moved one line upwards), a new object with new unique name and with properties partially determined in block 7-8 is created. The type of the graphical object is Text Label. The objects of this type are positioned relative to the symbol window, which allows the user to move price chart arbitrarily, without affecting the positions of messages.

The function Inform() can be called from any location in the program where, implicitly, a text message should be displayed. As a result of long processing, messages are accumulated in the window. The user can view the messages history by resizing the indicator subwindow (for example, by dragging up its upper edge). Optionally, you can also set the window height in such a way that the visible space displays the desired amount of message lines (three-four lines are usually recommended).



```
Закрыт ордер Buy 12522401
Ордер 12522432 преобразовался в рыночный
Открыт ордер Buy 12522483
Подсистема торговли занята
```

Fig. 156. Symbol window. Message in the indicator Subwindow.

It is easy to see that the diversity of messages displayed by the function can be increased. If you are going to expand the program, it is sufficient just to add the new version of 'case' in the operator 'switch' (block 7-8).

← Order Accounting Event Tracking Function →

Event Tracking Function

Many events take place during trading. A trader can see some of them directly in the symbol window, for example, market price changes or crossed indicator lines. Other events, though they are interesting for a trader, too, are not explicitly shown anywhere. A considerable part of those events can be detected and processed using MQL4.

For example, your dealing center may change the trading conditions shortly before important news are published or when the market becomes very active. In such cases, the spread or the minimum allowed distance for placing of orders and for the requested stop order prices may be increased. If this happens, it is necessary, first, to detect and take the new trading conditions into consideration, and, second, to inform the trader about these changes.

To solve these tasks, you may use the event tracking function in your Expert Advisor.

User-Defined Function Events()

```
int Events()
```

The function calculates the changes in the minimum distance required to place orders and their stop orders, as well as the changes in the list of market and pending orders available on the account. To execute the function, you should use the order accounting function Terminal() in your program. The values of the following global arrays are used:

- Mas_Ord_New - the array of characteristics of orders available as of the moment of the function Terminal() execution;
- Mas_Ord_Old - the array of characteristics of orders available as of the moment of the preceding execution of the function Terminal().

The values of the following global variables are used:
- Level_new - the current value of the minimum distance;
- Level_old - the preceding value of the minimum distance.

To display the messages, the function will use the data function Inform(). If the function Inform() is not included in the Expert Advisor, no messages will be shown.

The event tracking function Events() is formed as the include file Events.mqh:

```
//--------------------------------------------------------------------------------
// Events.mqh
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------- 1 --
// Event tracking function.
// Global variables:
// Level_new            The new value of the minimum distance
// Level_old            The preceding value of the minimum distance
// Mas_Ord_New[31][9]   The last known array of orders
// Mas_Ord_Old[31][9]   The old array of orders
//--------------------------------------------------------------------------- 2 --
int Events()                             // User-defined function
  {
   bool Conc_Nom_Ord;                    // Matching orders in ..
   //.. the old and the new arrays
//--------------------------------------------------------------------------- 3 --
   Level_new=MarketInfo(Symbol(),MODE_STOPLEVEL );// Last known
   if (Level_old!=Level_new)             // New is not the same as old..
     {                                   // it means the condition have been changed
      Level_old=Level_new;               // New "old value"
      Inform(10,Level_new);              // Message: new distance
     }
//--------------------------------------------------------------------------- 4 --
   // Searching for lost, type-changed, partly closed and reopened orders
   for(int old=1;old<=Mas_Ord_Old[0][0];old++)// In the array of old orders
     {                                   // Assuming the..
      Conc_Nom_Ord=false;                // ..orders don't match
      //--------------------------------------------------------------------- 5 --
      for(int new=1;new<=Mas_Ord_New[0][0];new++)//Cycle for the array ..
        {                                //..of new orders
         //------------------------------------------------------------------ 6 --
         if (Mas_Ord_Old[old][4]==Mas_Ord_New[new][4])// Matched number
           {                             // Order type becomes ..
            if (Mas_Ord_New[new][6]!=Mas_Ord_Old[old][6])//.. different
               Inform(7,Mas_Ord_New[new][4]);// Message: modified:)
            Conc_Nom_Ord=true;           // The order is found, ..
            break;                       // ..so exiting ..
           }                             // .. the internal cycle
         //------------------------------------------------------------------ 7 --
                                         // Order number does not match
         if (Mas_Ord_Old[old][7]>0 &&    // MagicNumber matches
            Mas_Ord_Old[old][7]==Mas_Ord_New[new][7])//.. with the old one
           {               //it means it is reopened or partly closed
                                            // If volumes match,..
            if (Mas_Ord_Old[old][5]==Mas_Ord_New[new][5])
               Inform(8,Mas_Ord_Old[old][4]);// ..it is reopening
            else                            // Otherwise, it was..
```

```
               Inform(9,Mas_Ord_Old[old][4]);// ..partly closing
            Conc_Nom_Ord=true;               // The order is found, ..
            break;                           // ..so exiting ..
           }                                 // .. the internal cycle
        }
      //--------------------------------------------------------------------- 8 --
      if (Conc_Nom_Ord==false)               // If we are here,..
        {                                    // ..it means no order found:(
         if (Mas_Ord_Old[old][6]==0)
            Inform(1, Mas_Ord_Old[old][4]);  // Order Buy closed
         if (Mas_Ord_Old[old][6]==1)
            Inform(2, Mas_Ord_Old[old][4]);  // Order Sell closed
         if (Mas_Ord_Old[old][6]> 1)
            Inform(3, Mas_Ord_Old[old][4]);  // Pending order deleted
        }
     }
//--------------------------------------------------------------------------- 9 --
   // Search for new orders
   for(new=1; new<=Mas_Ord_New[0][0]; new++)// In the array of new orders
     {
      if (Mas_Ord_New[new][8]>0)            //This one is not new, but reopened
         continue;                          //..or partly closed
      Conc_Nom_Ord=false;                   // As long as no matches found
      for(old=1; old<=Mas_Ord_Old[0][0]; old++)// Searching for this order
        {                                   // ..in the array of old orders
         if (Mas_Ord_New[new][4]==Mas_Ord_Old[old][4])//Matched number..
           {                                          //.. of the order
            Conc_Nom_Ord=true;              // The order is found, ..
            break;                          // ..so exiting ..
           }                                // .. the internal cycle
        }
      if (Conc_Nom_Ord==false)              // If no matches found,..
        {                                   // ..the order is new :)
         if (Mas_Ord_New[new][6]==0)
            Inform(4, Mas_Ord_New[new][4]); // Order Buy opened
         if (Mas_Ord_New[new][6]==1)
            Inform(5, Mas_Ord_New[new][4]); // Order Sell opened
         if (Mas_Ord_New[new][6]> 1)
            Inform(6, Mas_Ord_New[new][4]); // Pending order placed
        }
     }
//-------------------------------------------------------------------------- 10 --
   return;
  }
//-------------------------------------------------------------------------- 11 --
```

Global arrays and variables required for the function execution are described in block 1-2. In block 2-3, variable Conc_Nom_Ord used in the further code for orders analysis is opened.

The function tracks changes of the minimum distance for placing of orders and stop orders. For this, the current value of the minimum distance Level_new is calculated at each execution of the function (block 3-4) and then compared to the preceding one, the value of Level_old (obtained during the preceding execution of the function). If the values of these variables are not equal to each other, it means that the minimum distance has been changed by the dealing center shortly before the last execution of the function. In this case, the current value of the minimum distance is assigned to the variable Level_old (in order to consider it in the subsequent executions of the function), and the function Inform() is executed in order to display the corresponding message.

Generally, you can use a similar method to detect other events, for example, changes in spread, permissions to trade the given symbol (identifier MODE_TRADEALLOWED in the function MarketInfo()), the completion of a new bar (see Problem 27), the fact of crossing indicator lines (see Fig. 107), the fact of reaching a certain preset time, etc. The program may detect some events to use the obtained values in your EA, other events - to inform the user about them.

In blocks 4-10, the states of market and pending orders are analyzed. Information about most changes in orders is provided to the user. The analysis is performed in two stages. At the first stage, the program detects changes that relate to lost (closed or deleted), type-changing, partly closed and reopened orders (block 4-9). At the second stage (block 9-10), the new orders are searched for.

In blocks 4-9, the orders accounted in the array Mas_Ord_Old are analyzed. The amount of iterations in the external cycle 'for' is found according to the total amount of orders in the array (array element Mas_Ord_Old[0][0]). To check whether the order is kept as of the current moment, it is necessary to find a similar order in the orders array Mas_Ord_New. This search is performed in the internal cycle 'for' (block 6-8), the amount of iterations of which is equal to the amount of orders in the array (array element Mas_Ord_New[0][0]). We will further name the array Mas_Ord_Old 'old array', whereas the Mas_Ord_New - 'new array'.

In blocks 6-8, the program searches for only those orders, the characteristics of which are different. For example, in block 6-7, the order is checked for its number (see the correspondence of array indexes with order characteristics in Table 4). If the old-array order under check matches in number with one of the orders in the new array, it means that, at least, this order is not closed (or deleted). It is also necessary to check whether the order type is changed. If yes, it means that a pending order is modified into a market one. In this case, the corresponding message is displayed using the function Inform(). Independently on the fact of changing (or keeping unchanged) of the order type, this order will not be analyzed further: the program exits the internal cycle and, finally, starts a new iteration of the external cycle.

If the program finds at the execution of block 6-7 that the old-array order under check does not match in number with any orders from the new array, the control is passed to block 7-8. Here the program checks whether the current order from the new array has a nonzero MagicNumber (all orders opened and placed by the EA have a nonzero MagicNumber). If it has such a MagicNumber and this parameter coincides with the MagicNumber of the order from the old array under check, it means that this order is traded, but has been changed in some way. There are two situations when order number can be changed.

Situation 1. The order is partly closed. You can partly close a market order (not a pending one!) in two stages according to the technology accepted in MT 4. At the first stage, the initial order is completely closed. At the same time, a new market order of a smaller volume is opened with the same open price and with the same requested stop-order prices as in the initial order. This new order gets its unique name, other than the number of the initial order being partly closed.

Situation 2. The order is reopened by the dealing center. Some banks (due to their specific internal accounting rules) forcedly close all market orders at the end of trading day and immediately open market orders of the same type and with the same volume, but at the current price and minus swap. This event doesn't affect the economic results of a trading account in any way. Each newly opened order gets its unique number that doesn't match with the numbers of closed orders.

The difference between the two situations above is in the volumes of new orders: they are different in the first situation and they are unchanged in the second one. This difference is used in block 7-8 to distinguish between orders modified for different reasons. In both cases, the corresponding message is displayed ("the order is partly closed' or 'the order is reopened').

If the program has not detected the matching (block 6-7) or inheriting (block 7-8) of the order in the new array by the completion of the internal cycle, it means that the old-array order under check is closed or deleted. In this case, the control is passed to block 8-9, where one or another message will be displayed, according to the order type. In the above example, three kinds of messages are realized: for order Buy, for order Sell and for pending orders of all types. In a general case, this sequence can be slightly changed (extended) - you can create a group of the corresponding messages for each type of pending orders.

At the second stage, the program considers orders from the new order array (block 9-10). This is made in order to detect newly opened and placed orders. In the external cycle 'for', the program searches in all orders, the information about which is stored in the array of new orders. In order to identify the reopened or partly closed orders, the program uses a simple feature - the availability of comment. When partly closing or reopening an order, the server adds a comment that gives the number of the initial order. The EA above doesn't use comments, so the availability of a comment means that the order under check is not new.

If an order doesn't contain a comment, the program searches for an order with the same number in the old array. If the program finds the order having this number among old orders in the internal cycle 'for', it means that the order isn't new, but was opened before. However, if the number of the order from the new array doesn't match with any orders in the old array, it means that this order is an open market order or a placed pending one. In the lower part of block 9-10, the program calls to function Inform() in order to display the correspondent message, according to the order type.

The use of the considered function Events() turns out to be very helpful in practice. Once having used the function in an EA, the programmer usually uses it in his or her further work. It must be separately noted that the functions Events() and Terminal() are closely interrelated. If you are going to make changes in one of these functions (for example, to use other names for global arrays), you must make the corresponding changes in the other function. If you use comments in orders to realize your trading strategy, you should differently process the inheritance characteristic of the order (block 9-10), namely, you should use string functions to analyze the comment.

The amount of events considered in the function Events() can be highly increased. For example, if you want to completely display all events related to orders, you should add the analysis of order characteristics - changes in the requested stop-order prices and in the requested open prices of pending orders, as well as the closing method (whether the orders are closed as opposite orders or each is closed separately) and the reason for closing/deleting of orders (whether the price has reached the requested stop-order level or the order is closed on the trader's initiative, etc.).

← Data Function Volume Defining Function →

Volume Defining Function

For his or her practical work, a trader needs to be able to regulate the amount of lots for new orders to be opened. It is quite difficult to create a universal function for this purpose, since every trading strategy implies its special volume management. For example, some strategies imply the possibility to work with only one market order, whereas others allow to open new market orders regardless of the existing ones. Strategies based on management of different pending orders are known, too, the simultaneous availability of several market and pending orders being allowed in some cases.

One of the most common methods of volume calculation for newly opened orders (for the strategies that allow only one market order to be opened at a time) is the method of progressive investments. According to this method, the collateral cost of each new order is proportional to the free margin available at the moment of trade. If a market order is closed with profit, the allowed amount of lots for the new order increases. If it is closed with a loss, that amount will be decreased.

In the example below, the user-defined function Lot() is considered that allows you to set the volume for newly opening orders using one of the two alternatives:

Alternative 1. User sets the amount of lots for new orders manually.

Alternative 2. The amount of lots is calculated according to the amount of the money allocated by user. The amount of allocated money is set as percentage of free margin.

User-Defined Function Lot()

```
bool Lot()
```

The function calculates the amount of lots for new orders. As a result of the function execution, the value of the global variable Lots_New changes: the amount of lots. The function returns TRUE, if the free margin is sufficient for opening an order with the minimum amount of lots (for the symbol, in the window of which the EA is attached). Otherwise, it returns FALSE.

The function uses the values of the following global variables:

- Lots - volume in lots defined by the user;
- Percent - the percentage of free margin defined by the user.

To display message, the function uses the data function Inform(). If the function Inform() is not included in the EA, no messages will be displayed.

The function Lot() that determines the amount of lots is formed as include file Lot.mqh:

```
//--------------------------------------------------------------------------------
// Lot.mqh
// The code should be used for educational purpose only.
//----------------------------------------------------------------------------- 1 --
// Function calculating the amount of lots.
// Global variables:
// double Lots_New - the amount of lots for new orders (calculated)
// double Lots     - the desired amount of lots defined by the user.
// int Percent     - free margin percentage defined by the user
// Returned values:
// true  - if there is enough money for the minimum volume
// false - if there is no enough money for the minimum volume
//----------------------------------------------------------------------------- 2 --
bool Lot()                                        // User-defined function
  {
   string Symb   =Symbol();                       // Symbol
   double One_Lot=MarketInfo(Symb,MODE_MARGINREQUIRED);//!-lot cost
   double Min_Lot=MarketInfo(Symb,MODE_MINLOT);// Min. amount of lots
   double Step   =MarketInfo(Symb,MODE_LOTSTEP);//Step in volume changing
   double Free   =AccountFreeMargin();            // Free margin
//----------------------------------------------------------------------------- 3 --
   if (Lots>0)                                    // Volume is explicitly set..
     {                                            // ..check it
      double Money=Lots*One_Lot;                  // Order cost
      if(Money<=AccountFreeMargin())              // Free margin covers it..
         Lots_New=Lots;                           // ..accept the set one
      else                                        // If free margin is not enough..
         Lots_New=MathFloor(Free/One_Lot/Step)*Step;// Calculate lots
     }
//----------------------------------------------------------------------------- 4 --
   else                                           // If volume is not preset
     {                                            // ..take percentage
      if (Percent > 100)                          // Preset, but incorrectly ..
         Percent=100;                             // .. then no more than 100
      if (Percent==0)                             // If 0 is preset ..
         Lots_New=Min_Lot;                        // ..then the min. lot
      else                                        // Desired amount of lots:
         Lots_New=MathFloor(Free*Percent/100/One_Lot/Step)*Step;//Calc
     }
//----------------------------------------------------------------------------- 5 --
```

```
      if (Lots_New < Min_Lot)                      // If it is less than allowed..
         Lots_New=Min_Lot;                         // .. then minimum
      if (Lots_New*One_Lot > AccountFreeMargin()) // It isn't enough even..
        {                                          // ..for the min. lot:(
         Inform(11,0,Min_Lot);                     // Message..
         return(false);                            // ..and exit
        }
      return(true);                                // Exit user-defined function
     }
   //------------------------------------------------------------------------------ 6 --
```

The function has a simple code. In block 1-2, global variables and returned values are described. In block 2-3, the values of some variables are calculated. For calculations, the following priority in setting of values is accepted: If a user has set a non-zero amount of lots, the value of the percentage of free margin is not taken into consideration. External variables Lots and Percent are declared in the include file Variables.mqh.

In block 3-4, the calculations are made for the situation where the user has defined a non-zero value of the volume in lots in the external variable Lots. In this case, the program makes a check. If the free margin is sufficient to open a market order with the defined amount of lots, then the value set by the user will be assigned to the global variable Lots_New and used in further calculations. If the free margin doesn't cover this amount, then the maximum possible amount of lots is calculated that is used further (see Mathematical Functions).

The control is passed to block 4-5, if the user has defined zero amount of lots. At the same time, we take into consideration the percentage of free margin specified by the user in the external variable Percent. The program makes a check: If the value exceeds one hundred (percent), the value of 100 is used in calculations. If the user has defined zero value of the variable Percent, the amount of lots is equated with the minimum possible value set by the dealing center. For all intermediate Для всех промежуточных величин высчитывается количество лотов, соответствующее количеству выделенных пользователем средств.

In block 5-6, the necessary checks are made. If the calculated amount of lots turns out to be less than the minimum allowed one (for example, zero value can be obtained in block 4-5, if the user has defined a small value of the variable Percent), then the minimum value will be assigned to the variable Lots_New. Then the program checks whether there are enough free assets to open an order with the volume of the previously calculated amount of lots (there can be insufficient money on the account). If the money available is not enough, the program displays a message for the user and exits the function, the function returning 'false'. However, the successful check results in returning of 'true'.

← Event Tracking Function Function Defining Trading Criteria →

Function Defining Trading Criteria

The success of any trading strategy mainly depends on the sequence of trading criteria calculations. The function that defines trading criteria is the most important part of a program and must be used without fail. According to trading strategy, the function may return values that correspond with particular trading criteria.

In a general case, the following criteria can be defined:

- criterion for opening of a market order;
- criterion for closing of a market order;
- criterion for partly closing of a market order;
- criterion for closing of opposite market orders;
- criterion for modification of the requested prices of stops of a market order;
- criterion for placing of a pending order;
- criterion for deletion of a pending order;
- criterion for modification of the requested open price of a pending order;
- criterion for modification of the requested prices of stops of a pending order.

In most cases, the triggering of one trading criterion is exclusive as related to other trading criteria. For example, if the criterion for opening a Buy order becomes important at a certain moment, this means that the criteria used for closing Buy orders or for opening Sell orders cannot be important at the same moment (see Relation of Trading Criteria). At the same time, according to the rules inherent in a given trading strategy, some criteria may trigger simultaneously. For example, the criteria for closing a market order Sell and for modification of a pending order BuyStop may become important simultaneously.

A trading strategy imposes requirements to the content and the usage technology of the function defining trading criteria. Any function can return only one value. So, if you have realized in your Expert Advisor a trading strategy that implies using only mutually exclusive trading criteria, the value returned by the function can be associated with one of the criteria. However, if your strategy allows triggering of several criteria at a time, their values must be passed to other functions for being processed, using global variables for that.

Trading strategy realized in the EA below implies using only mutually exclusive criteria. This is why the function Criterion() for passing the above criteria to other functions uses the value returned by the function.

User-Defined Function Criterion()

```
int Criterion()
```

The function calculates trading criteria. It can return the following values:
10 - triggered a trading criterion for closing of market order Buy;
20 - triggered a trading criterion for opening of market order Sell;
11 - triggered a trading criterion for closing of market order Buy;
21 - triggered a trading criterion for opening of market order Sell;
0 - no important criteria available;
-1 - the symbol used is not EURUSD.

The function uses the values of the following external variables:
St_min - the lower level of indicator Stochastic Oscillator;
St_max - the upper level of indicator Stochastic Oscillator;
Open_Level - the level of indicator MACD (for order opening);
Close_Level - the level of indicator MACD (for order closing).

In order to display messages, the function uses the data function Inform().If the function Inform() is not included in the EA, no messages will be displayed.

Function defining trading criteria, Criterion(), is formed as include file Criterion.mqh:

```
//--------------------------------------------------------------------
// Criterion.mqh
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
// Function calculating trading criteria.
// Returned values:
// 10 - opening Buy
// 20 - opening Sell
// 11 - closing Buy
// 21 - closing Sell
// 0  - no important criteria available
// -1 - another symbol is used
//--------------------------------------------------------------- 2 --
// External variables:
extern int St_min=30;                // Minimum stochastic level
extern int St_max=70;                    // Maximum stochastic level
extern double Open_Level =5;             // MACD level for opening (+/-)
extern double Close_Level=4;             // MACD level for closing (+/-)
//--------------------------------------------------------------------- 3 --
int Criterion()                          // User-defined function
  {
   string Sym="EURUSD";
   if (Sym!=Symbol())                    // If it is a wrong symbol
     {
      Inform(16);                        // Messaging..
      return(-1);                        // .. and exiting
     }
   double
   M_0, M_1,                             // Value MAIN at bars 0 and 1
   S_0, S_1,                             // Value SIGNAL at bars 0 and 1
   St_M_0, St_M_1,                       // Value MAIN at bars 0 and 1
   St_S_0, St_S_1;                       // Value SIGNAL at bars 0 and 1
   double Opn=Open_Level*Point;          // Opening level of MACD (points)
   double Cls=Close_Level*Point;         // Closing level of MACD (points)
//--------------------------------------------------------------------- 4 --
   // Parameters of technical indicators:
   M_0=iMACD(Sym,PERIOD_H1,12,26,9,PRICE_CLOSE,MODE_MAIN,0); // 0 bar
   M_1=iMACD(Sym,PERIOD_H1,12,26,9,PRICE_CLOSE,MODE_MAIN,1); // 1 bar
   S_0=iMACD(Sym,PERIOD_H1,12,26,9,PRICE_CLOSE,MODE_SIGNAL,0);//0 bar
   S_1=iMACD(Sym,PERIOD_H1,12,26,9,PRICE_CLOSE,MODE_SIGNAL,1);//1 bar

   St_M_0=iStochastic(Sym,PERIOD_M15,5,3,3,MODE_SMA,0,MODE_MAIN,  0);
   St_M_1=iStochastic(Sym,PERIOD_M15,5,3,3,MODE_SMA,0,MODE_MAIN,  1);
   St_S_0=iStochastic(Sym,PERIOD_M15,5,3,3,MODE_SMA,0,MODE_SIGNAL,0);
   St_S_1=iStochastic(Sym,PERIOD_M15,5,3,3,MODE_SMA,0,MODE_SIGNAL,1);
//--------------------------------------------------------------------- 5 --
   // Calculation of trading criteria
   if(M_0>S_0 && -M_0>Opn && St_M_0>St_S_0 && St_S_0<St_min)
      return(10);                        // Opening Buy
   if(M_0<S_0 &&  M_0>Opn && St_M_0<St_S_0 && St_S_0>St_max)
      return(20);                        // Opening Sell
   if(M_0<S_0 &&  M_0>Cls && St_M_0<St_S_0 && St_S_0>St_max)
      return(11);                        // Closing Buy
   if(M_0>S_0 && -M_0>Cls && St_M_0>St_S_0 && St_S_0>St_min)
      return(21);                        // Closing Sell
//--------------------------------------------------------------------- 6 --
   return(0);                            // Exit the user-defined function
  }
//--------------------------------------------------------------------- 7 --
```

In block 1-2, the values returned by the function are described. In block 2-3, some external variables are declared. The include file Criterion.mqh is the only file used in the considered EA, in which the global (in this case, external) variables are declared. In the section named Structure of a Normal Program, you can find the reasoning for declaring of all global variables without exception in a separate file Variables.mqh. In this case, the external variables are declared in the file Criterion.mqh for two reasons: first, to demonstrate that it is technically possible (it is not always possible); second, to show how to use external variables at debugging/ testing of a program.

It is technically possible to declare external variables in the file Criterion.mqh, because these variables are not used in any other functions of the program. The values of external variables declared in block 2-3 determine the levels for indicators Stochastic Oscillator and MACD and are used only in the considered function Criterion(). The declaration of external variables in the file containing the function that defines trading criteria may be reasonable, if the file is used temporarily, namely, during the program debugging and calculating of optimal values of those external variables. For this purpose, you can add other external variables in the program, for example, to optimize inputs of indicators (in this case, the constants set the values of 12,26,9 for MACD and 5,3,3 for Stochastic Oscillator). Once having finished coding, you may delete these external variables from the program and replace them with constants with the values calculated during optimization.

In block 3-4, the local variables are opened and described. The Expert Advisor is intended to be used on symbol EURUSD, so the necessary check is made in block 3-4. If the EA is launched in the window of another symbol, the function finishes operating and returns the value of -1 (wrong symbol).

In the program, the values of two indicators calculated on the current and on the preceding bar are used (block 4-5). Usually, when you use indicators Stochastic Oscillator and MACD, the signals for buying or selling are formed when two indicator lines meet each other. In this case, we use two indicators simultaneously to define trading criteria. The probability of simultaneous intersection of indicator lines of two indicators is rather low. It is much more probable that they will intersect one by one - first in one indicator, a bit later - in another one. If the indicator lines intersect within a short period of time, two indicators can be considered to have formed a trading criterion.

For example, below is shown how a trading criterion for buying is calculated (block 5-6):

```
if(M_0>S_0 && -M_0>Opn && St_M_0>St_S_0 && St_S_0<St_min)
```

According to this record, the criterion for buying is important if the following conditions are met:

- in indicator MACD, indicator line MAIN (histogram) is above indicator line SIGNAL and below the lowest level Open_Level (Fig. 157);
- in indicator Stochastic Oscillator, indicator line MAIN (histogram) is above indicator line SIGNAL and below the lowest level St_min (Fig. 158).

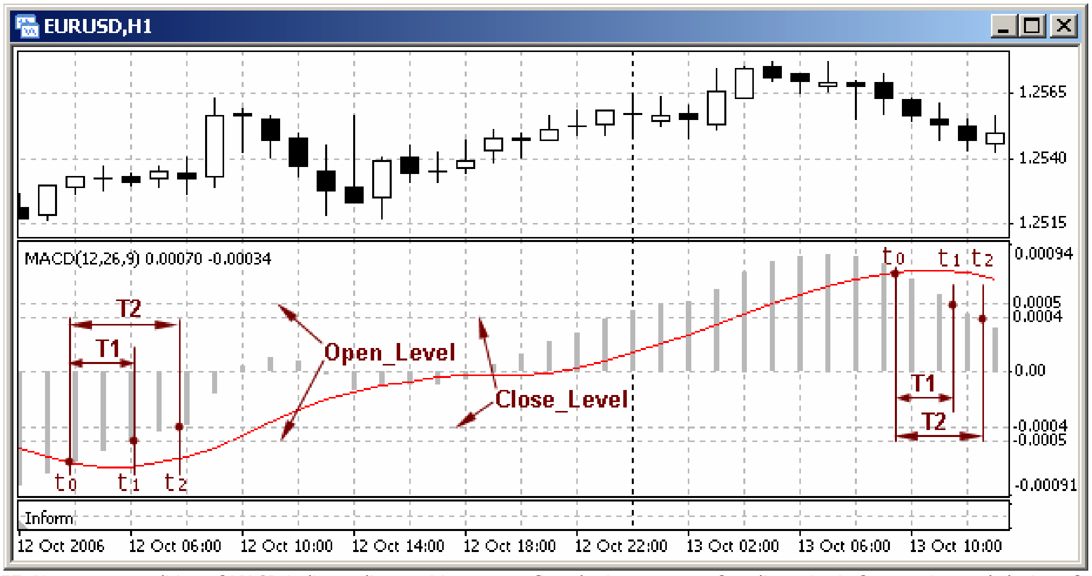


Рис. 157. Necessary condition of MACD indicator line positions to confirm the importance of trading criteria for opening and closing of orders.

In the left part of Fig. 157, the positions of MACD indicator lines is shown, at which two criteria trigger - opening of Buy and closing of Sell. Indicator line MAIN is below the level of 0.0005 within the time of T1 = t 1 - t 0. If the necessary indications of Stochastic Oscillator occur at this moment, the criterion for opening of Buy will trigger. Line MAIN is below the level of 0.0004 within the time T2 = t 2 - t 0. If the indications of Stochastic Oscillator confirm this position, the criterion for closing of Sell will trigger.

Please note that, formally, both criteria above trigger within T1 (if confirmed by Stochastic Oscillator). It was mentioned before that the considered function Criterion() returns only one value, namely, the value assigned to one triggered criterion. During this period, it becomes necessary to choose one of the criteria. This problem is solved in advance, during programming, according to the priorities prescribed by the trading strategy.

In this case (according to the trading strategy considered), the priority of opening a Buy order is higher than that of closing a Sell order. This is why, in block 5-6, the program line, in which the criterion for opening of Buy, is positioned above. If during the period of T1 (Fig. 157), we have got the confirmation from Stochastic Oscillator, the function returns 10 assigned to this criterion. Within the period from t1 to t2, the function will return 21 assigned to the Sell closing criterion.

At the same time, at the execution of trade functions, the necessary trade requests will be formed. At triggering of the criterion for opening of Buy, first of all, the trade requests for closing of all available Sell orders will be formed. As soon as no such orders are left, opening of one Buy order will be requested. Respectively, when the criterion for closing of Sell orders triggers, a sequence of trade requests for only closing of all Sell orders will be formed (see Trade Functions).

The conditions, at which Stochastic Oscillator confirmation triggers, are shown in Fig. 158.

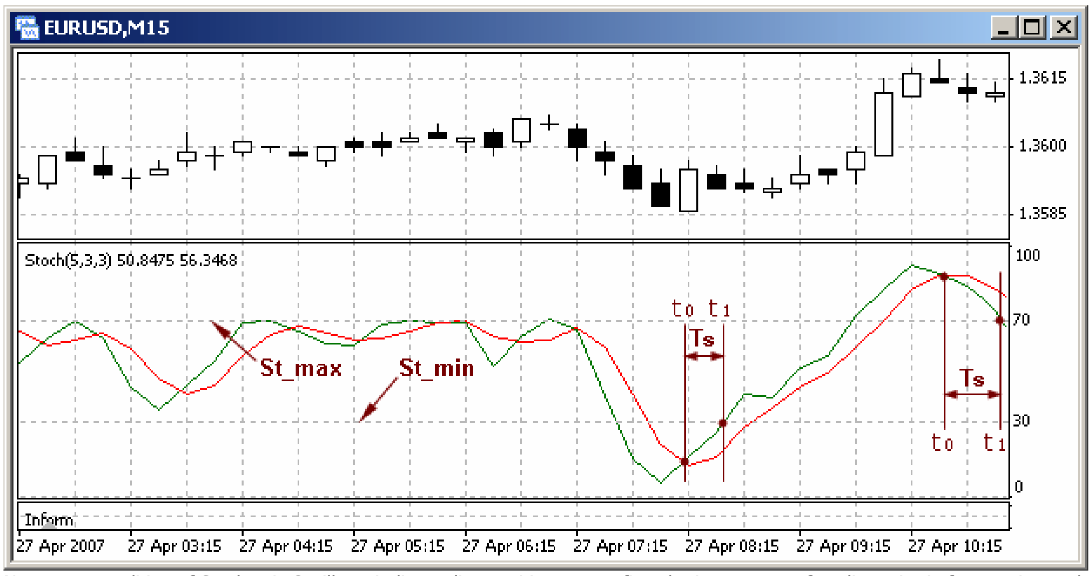


Fig. 158. Necessary condition of Stochastic Oscillator indicator line positions to confirm the importance of trading criteria for opening and closing of orders.

According to the program code specified in block 5-6, the criteria for opening of Buy and closing of Sell can become important provided the indicator line MAIN turns out to be above the signal line SIGNAL in Stochastic Oscillator, line MAIN being below the minimum level St_min. In Fig. 158, such conditions are formed within the period of Ts. The mirrored conditions confirm the triggering of criteria for opening of the order Sell and closing of the order Buy (in the right part of Fig. 158). If no criterion has triggered, the function returns 0. Other trades can be made under these conditions, for example, correction of the stop levels requested.

It must be noted separately that the considered trading strategy implies the usage of indications produced by MACD calculated on the one-hour timeframe, whereas Stochastic Oscillator is calculated on the 15-minute timeframe.The timeframe may be changed during testing in order to optimize the strategy. However, after testing, in the final code of the function Criterion(), it is necessary to specify constant value for all parameters calculated, including timeframes. The EA must be used only under the conditions, for which it has been created. In the example above (with the values of PERIOD_H1 and PERIOD_M15 specified explicitly in the indicators), the EA will consider only necessary parameters regardless the current timeframe set in the symbol window, where the EA has been launched.

Trading criteria accepted in this given EA are used for training purposes and should not be considered as a guide to operations when trading on a real account.

← Volume Defining Function Trade Functions →

Trade Functions

As a rule, a normal Expert Advisor contains a number of trade functions. They can be divided into two categories - control functions and executive functions. In most cases, only one control function and several executive functions are used in an EA.

A trading strategy in a normal EA is realized on the basis of two functions - a function defining trading criteria and a control trade function. There mustn't be any indications of the trading strategy anywhere else in the program. The controlling trade function and the function defining trading criteria must be coordinated with each other in the values of the parameters they pass.

Each executive trade function has a special range of tasks. According to the requirements of the trading strategy, trade functions intended for the following tasks can be used in an EA:

- opening a market order of the preset type;
- closing one market order of the preset type;
- partly closing one market order of the preset type;
- closing all market order of the preset type;
- closing two opposite market orders in the preset volume;
- closing all market orders;
- modification of stop orders of a market order of the preset type;
- placing a pending order of the preset type;
- deletion of one pending order of the preset type;
- deletion of all pending orders of the preset type;
- deletion of all pending orders;
- modification of a pending order of the preset type.

A general trading sequence in a normal Expert Advisor consists in the following: On the basis of calculated (according to the strategy used) trading criteria, the controlling trade function (also realizing the strategy) calls some or other executive trade functions that, in their turn, form the necessary trade requests.

User-Defined Controlling Function Trade()

```
int Trade( int Trad_Oper )
```

It's the basic function that realizes your strategy.

Parameter **Trad_Oper** can take the following values corresponding with the trading criteria:
10 - triggered a trading criterion for opening a market order Buy;
20 - triggered a trading criterion for opening a market order Sell;
11 - triggered a trading criterion for closing a market order Buy;
21 - triggered a trading criterion for closing a market order Sell;
0 - no important criteria available;
-1 - the symbol used is not EURUSD.

To execute the function, the following trade functions are required:

- Close_All() - function closing all market orders of the preset type;
- Open_Ord() - function opening one market order of the preset type;
- Tral_Stop() - function modifying StopLoss of a market order of the preset type;
- Lot() - function detecting the amount of lots for new orders.

The control trade function Trade() is formed as include file Trade.mqh:

```
//--------------------------------------------------------------------------
// Trade.mqh
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------
// Trade function.
//-------------------------------------------------------------------- 1 --
int Trade(int Trad_Oper)              // User-defined function
  {
   // Trad_Oper - trade operation type:
   // 10 - opening Buy
   // 20 - opening Sell
   // 11 - closing Buy
   // 21 - closing Sell
   //  0 - no important criteria available
   // -1 - another symbol is used
   switch(Trad_Oper)
     {
      //-------------------------------------------------------------- 2 --
      case 10:                        // Trading criterion = Buy
         Close_All(1);                // Close all Sell
         if (Lot()==false)            // Not enough money for min.
            return;                   // Exit the user-defined function
         Open_Ord(0);                 // Open Buy
         return;                      // Having traded, leave
         //------------------------------------------------------------ 3 --
      case 11:                        // Trading criterion = closing Buy
         Close_All(0);                // Close all Buy
         return;                      // Having traded, leave
         //------------------------------------------------------------ 4 --
      case 20:                        // Trading criterion = Sell
         Close_All(0);                // Close all Buy
         if (Lot()==false)
            return;                   // Exit the user-defined function
         Open_Ord(1);                 // Open Sell
         return;                      // Having traded, leave
         //------------------------------------------------------------ 5 --
      case 21:                        // Trading criterion = closing Sell
         Close_All(1);                // Close all Sell
         return;                      // Having traded, leave
         //------------------------------------------------------------ 6 --
      case 0:                         // Retaining opened positions
         Tral_Stop(0);                // Trailing stop Buy
         Tral_Stop(1);                // Trailing stop Sell
         return;                      // Having traded, leave
         //------------------------------------------------------------ 7 --
     }
  }
//-------------------------------------------------------------------- 8 --
```

The control trade Trade() is called from the special function start() of the Expert Advisor usualexpert.mq4. The value returned by the function defining trading criteria Criterion() is given as the passed parameter in the function Trade().

In block 1-2 of the function Trade(), the trading criteria considered by the realized trading strategy are described. In the function, we use the operator switch() (blocks 2-7) that allows us to activate the required group of functions to trade according to the trading criterion. According to the trading strategy, the EA opens and closes market orders. No operations with pending orders are provided by this trading strategy.

In the section named Function Defining Trading Criteria, it was specified that for some trading criteria the program can form several trade requests. Thus, in case of important criterion for buying (the value of the variable Trad_Oper is equal to 10), the control is passed to the mark 'case 10' (block 2-3) during execution of the operator switch(). In this case, the program first calls to function Close_All(1). The execution of this function results in closing of all market orders Sell opened for the symbol EURUSD. After all orders Sell have been closed, the available money is checked for whether it is enough to make the next trade. For this purpose, the user-defined function Lot() is called (see Volume Detecting Function). If this function returns 'false', it means that the money available on the account is not enough to open order Buy with the minimum allowed amount of lots. In this case, the function Trade() ends its operations. If there is enough money, the trade function Open_Ord(0) is called to open one market order Buy with the amount of lots calculated at the execution of function Lot(). The described set of actions represents the Expert Advisor's response to the situation on the market (according to the given trade criterion).

If the criterion is important that points out at the necessity to close market orders Buy, the control is passed to the mark 'case 11' in block 3-4. In this case, only one function Close_All(0) is called to close all the orders of the Buy type available. Blocks 4-6 are built in the way similar to blocks 2-4, the control is passed to marks 'case 20' and 'case 21', if the criteria for selling or closing market orders Sell become important.

Please note that all executive trade functions that form trade requests are called in the function Trade() that, in its turn, is called at the execution of the EA's special function start() launched by the client terminal as a result of a new tick incoming. The code of the function Trade() is written in such a way that the control is not returned to the function start() (and, at the end, to the client terminal) until all required executive trade functions are executed. This is why all trades intended for each trading criterion are made by the EA one by one, without breaks. The exception may be the cases of critical errors occurring during making of trades (see Error Processing Function).

If no trading criterion is detected as important (variable Trad_Oper is equal to 0) at the execution of the function Criterion(), the control is passed to the mark 'case 0', which results in double call to function Tral_Stop() to modify the requested values of the market orders of different types. The trading strategy realized in this EA allows the availability of only one market order, so the sequence of calls to the functions Tral_Stop(0) and Tral_Stop(1) doesn't matter. In this case, it is a random choice.

If the function Criterion() has returned the value of -1, this means that the EA is attached to the window of a symbol that is not EURUSD. In this case, the function Trade() does not call to any executive trade functions and returns the control to the special function start() that has called it.

User-Defined Executive Trade Function Close_All()

```
int Close_All( int Tip)
```

The function closes all market orders of the given type.

The parameter **Tip** can take the following values corresponding with the types of orders to be closed:
0 - closing Buy orders;
1 - closing Sell orders.

To execute the function, it is necessary to apply the order accounting function Terminal(), the event tracking function Events() and the error processing function Errors() in the program. To display messages, the function implies using of the data function Inform(). If the function Inform() is not included in the EA, no messages will be displayed.

The values of the following global arrays are used:

- Mas_Ord_New - the array of characteristics of orders available as of the moment of the function Terminal() execution;
- Mas_Tip - the array of the total amount of orders of all types as of the moment of the last execution of the function Terminal().

The executive trade function Close_All() is formed as the include file Close_All.mqh:

```
//--------------------------------------------------------------------------------
// Close_All.mqh
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------- 1 --
// Function closing all market orders of the given type
// Global variables:
// Mas_Ord_New   Last known order array
// Mas_Tip       Order type array
//--------------------------------------------------------------------------- 2 --
int Close_All(int Tip)                     // User-defined function
  {
   // int Tip                              // Order type
   int Ticket=0;                           // Order ticket
   double Lot=0;                           // Amount of closed lots
   double Price_Cls;                       // Order close price
//--------------------------------------------------------------------------- 3 --
   while(Mas_Tip[Tip]>0)                  // As long as the orders of the ..
     {                                    //.. given type are available
      for(int i=1; i<=Mas_Ord_New[0][0]; i++)// Cycle for live orders
        {
         if(Mas_Ord_New[i][6]==Tip &&      // Among the orders of our type
            Mas_Ord_New[i][5]>Lot)         // .. select the most expensive one
           {                               // This one was found at earliest.
            Lot=Mas_Ord_New[i][5];         // The largest amount of lots found
            Ticket=Mas_Ord_New[i][4];      // Its order ticket is that
           }
        }
      if (Tip==0) Price_Cls=Bid;           // For orders Buy
      if (Tip==1) Price_Cls=Ask;           // For orders Sell
      Inform(12,Ticket);                   // Message about an attempt to close
      bool Ans=OrderClose(Ticket,Lot,Price_Cls,2);// Close order !:)
      //--------------------------------------------------------------------- 4 --
      if (Ans==false)                      // Failed :(
        {                                 // Check for errors:
         if(Errors(GetLastError())==false)// If the error is critical,
            return;                        // .. then leave.
        }
      //--------------------------------------------------------------------- 5 --
      Terminal();                          // Order accounting function
      Events();                            // Event tracking
     }
   return;                                 // Exit the user-defined function
  }
//--------------------------------------------------------------------------- 6 --
```

In block 1-2, the global variables used are described. In block 2-3, local variables are opened and described. The condition Mas_Tip[Tip]>0 in the heading of the cycle operator 'while' (blocks 3-6) implies that the function will hold the control until it fulfills its intended purpose, namely, until all orders of the given type are closed. The element of the global array Mas_Tip[Tip] contains the value equal to the amount of orders of the given type Tip. For example, if the function Close_All() is called with the transferred parameters equal to 1, this means that the function must close all market orders Sell (see Types of Trades). In this case, the value of the array element Mas_Tip[1] will be equal to the amount of available orders Sell (last known as of the moment of the execution of the function Terminal()). Thus, the cycle operator 'while' will be executed so many times as many Sell orders are available.

If the trader doesn't intervene into the operations of the EA (i.e., he or she doesn't place orders manually), then only one market order of one type or another may be available in trading. However, if the trader has additionally placed one or several market orders on his or her own initiative, then a certain sequence of orders must be kept at the execution of the function Close_All(). The preferable sequence of closing orders is to close larger ones first. For example, if there are three orders Sell as of the moment of starting to execute the function Close_All(), one of them being opened for 5 lots, another one being opened for 1 lot, and the third one being opened for 4 lots, then the orders will be closed in the following sequence according to the above reasoning: the first order to be closed will be that of 5 lots, then that of 4 lots, and the last will be the order of 1 lot.

Please note that the amount of lots is the only criterion used to determine the sequence of closing orders. The order's profit/loss, open price, as well as other parameters characterizing the order (the requested stop-order prices, time and reason for closing, etc.) are not considered.

All market orders of a certain type must be closed, if the criterion for closing of orders of this type is important, the sequence of closing being from the larger to the smaller volumes.

To keep the above sequence of closing order, in block 3-4, the cycle 'for' is used, in which the largest (in volume) order is selected among all orders of the given type. This order is searched for on the basis of analysis of the values of global array Mas_Ord_New containing the information about all order available in trading. After the ticket of this order has been detected, according to the order type, the requested close price will be calculated that is equal to the corresponding value of the last known two-way quote. If the orders to be closed are of Buy type, the close price must be requested on the basis of Bid value. If they are Sell orders, then use Ask values.

Directly before forming a trade request, the information about the attempt to close the order is displayed. The program uses function call Inform() for this purpose. The trade request for closing of the order is formed in the line:

```
        bool Ans=OrderClose(Ticket,Lot,Price_Cls,2);// Close order !:)
```

The calculated values are used as parameters: Ticket - order number, Lot - volume in lots, Price_Cls - requested close price, 2 - slippage.

In block 4-5, the trade results are analyzed. If the function OrderClose() has returned 'true', this means that the trade has completed successfully, i.e., the order has been closed. In this case, the control is passed to block 5-6, where the information about orders available at the current moment is updated. After the execution of functions Terminal() and Events(), the current iteration of the cycle 'while' ends (the amount of available orders can change within the function execution time and during making trades, so the execution of the order accounting function is obligatory at each iteration of the cycle 'while'). If the orders of the given type are still available in trading, they will be closed at the next iteration of the cycle 'while', the new values of the elements of arrays Mas_Ord_New and Mas_Tip obtained at the execution of the function Terminal() being used for determining of the parameters of the next order to be closed.

If the execution of the request results in that the function OrderClose() returns 'false', this means that the order has not been closed. In order to find out about the reasons for this failure, the program analyzes the last error occurred at the attempt to make the trade. For this purpose, it calls to the function Errors() (see Error Processing Function). If, at execution of this function, the program detects that the error is critical (for example, trading is prohibited), the function Close_All() ends its operations and returns the control to the control trade function Trade(), which finally results in that the special function start90 of the EA ends its execution, as well. At the next tick, the terminal will launch the function start() for execution again. If the closing criterion remains actual at that moment, this will produce the call to the function closing all orders, Close_All().

User-Defined Executive Trade Function Open_Ord()

```
  int Open_Ord ( int Tip)
```

The function opens one market order of the given type.

The parameter **Tip** can take the following values corresponding with the types of the orders to be opened:
0 - the type Buy of orders to be opened;
1 - the type Sell of orders to be opened.

To execute the function, you should use in the program the order accounting function Terminal(), the event tracking function Events() and the error processing function Errors(). To display messages, the function implies the data function Inform(). If the function Inform() is not included in the EA, no messages will be displayed.

The values of the following global variables are used:

- Mas_Tip - the array of the total amount of orders of all types as of the moment of the last execution of the function Terminal();
- StopLoss - the value of StopLoss (amount of points);
- TakeProfit - the value of TakeProfit (amount of points).

The executive trade function Open_Ord() is formed as include file Open_Ord.mqh:

```
//--------------------------------------------------------------------------------
// Open_Ord.mqh
// The code should be used for educational purpose only.
//--------------------------------------------------------------------------- 1 --
// Function opening one market order of the given type
// Global variables:
// int Mas_Tip                 Order type array
// int StopLoss                The value of StopLoss (amount of points)
// int TakeProfit              The value of TakeProfit (amount of points)
//--------------------------------------------------------------------------- 2 --
int Open_Ord(int Tip)
  {
   int    Ticket,                      // Order ticket
          MN;                          // MagicNumber
   double SL,                          // StopLoss (as related to the price)
          TP;                          // TakeProf (as related to the price)
//--------------------------------------------------------------------------- 3 --
   while(Mas_Tip[Tip]==0)              // Until they ..
     {                                 //.. succeed
      if (StopLoss<Level_new)          // If it is less than allowed..
         StopLoss=Level_new;           // .. then the allowed one
      if (TakeProfit<Level_new)        // If it is less than allowed..
         TakeProfit=Level_new;         // ..then the allowed one
      MN=TimeCurrent();                // Simple MagicNumber
      Inform(13,Tip);                  // Message about an attempt to open
      if (Tip==0)                      // Let's open a Buy
        {
         SL=Bid - StopLoss*  Point;       // StopLoss   (price)
         TP=Bid + TakeProfit*Point;       // TakeProfit (price)
         Ticket=OrderSend(Symbol(),0,Lots_New,Ask,2,SL,TP,"",MN);
        }
      if (Tip==1)                         // Let's open a Sell
        {
         SL=Ask + StopLoss*  Point;       // StopLoss   (price)
         TP=Ask - TakeProfit*Point;       // TakeProfit (price)
         Ticket=OrderSend(Symbol(),1,Lots_New,Bid,2,SL,TP,"",MN);
        }
      //------------------------------------------------------------------------ 4 --
      if (Ticket<0)                        // Failed :(
        {                                  // Check for errors:
         if(Errors(GetLastError())==false)// If the error is critical,
            return;                        // .. then leave.
        }
      Terminal();                          // Order accounting function
      Events();                            // Event tracking
     }
//------------------------------------------------------------------------------ 5 --
   return;                                 // Exit the user-defined function
  }
//------------------------------------------------------------------------------ 6 --
```

In blocks 1-3 of the function Open_Ord(), the global variables are described, the values of which are used at the execution of the function, and local variables are opened and described. The basic code of the function is concentrated in the cycle operator 'while' (blocks 3-5) that is executed as long as no orders of the given type Tip are available in trading.

The trading strategy implies opening orders that have non-zero stop orders. In a general case, a trader may set such values of stop orders that will not comply with the requirements of the dealing center, namely, less than the allowed minimum distance from the market price. This is why the necessary checks are made before opening an order: If the last known minimum distance (Level_new) exceeds the value of the external variable StopLoss or TakeProfit, the value of this variable is increased and set to be equal to Level_new.

Each order to be opened has its unique MagicNumber equal to the current server time. As a result of the execution of one EA for a symbol, only one market order can be open (or placed, if it is a pending order) at a time. This provides all market orders with unique MagicNumbers. Before opening an order, the function Inform() is executed, which results in displaying of a message informing about an attempt to make a trade.

According to the order type, the body of one of the operators 'if' is executed. For example, if the value of the transferred parameter Tip is equal to 0, this means that an order Buy must be opened. In this case, the values of StopLoss and TakeProfit are calculated that correspond with the Buy order type, then the control is passed to line

```
            Ticket=OrderSend(Symbol(),0,Lots_New,Ask,2,SL,TP,"",MN);
```

to form a trade request for opening a market order Buy. Similar calculations are made, if the value of the parameter Tip is 1, i.e., an order Sell should be opened.

Errors in all user-defined executive trade functions are processed in a similar way. If a trade is made successfully, the function ends its operations (because no next iteration of the cycle 'while' will be performed, since the value of the element of array Mas_Tip[Tip] will be equal to 1 after the execution of the function Terminal()). However, if the trade request is not executed, the errors are analyzed (block 4-5). In this case, the error detecting function Errors() is called. If it returns 'false' (the error is critical), the execution of the function Open_Ord() ends, the control is consecutively passed to the control trade function Trade(), to the special function start() and then to the client terminal. However, if the error is overcomable, then, after updating of order arrays in the function Terminal(), the control is passed to the consecutive iteration of the cycle 'while', which results in one more attempt to open an order.

Thus, the function Open_Ord() holds the control until an order is opened or a critical error is got at the execution of the trade request.

User-Defined Executive Trade Function Tral_Stop()

```
   int Tral_Stop ( int Tip)
```

The function modifies all market orders of the given type.

The parameter **Tip** can take the following values corresponding with the type of orders to be modified:
0 - the type Buy of orders to be modified;
1 - the type Sell of orders to be modified.

To execute the function, it is necessary to use in the program the order accounting function Terminal(), the event tracking function Events(), and the error processing function Errors(). To display messages, the function implies the data function Inform(). If the function Inform() is not included in the EA, no messages will be displayed.

The values of the following global variables are used:

- Mas_Ord_New - the array of characteristics of orders available as of the moment of the last execution of the function Terminal();
- TralingStop - the distance between the market price and the desired value of the requested price for StopLoss (amount of points).

The executive trade function Tral_Stop() is formed as include file Tral_Stop.mqh:

```
//--------------------------------------------------------------------------------
// Tral_Stop.mqh
// The code should be used for educational purpose only.
//---------------------------------------------------------------------------- 1 --
// Function modifying StopLosses of all orders of the given type
// Global variables:
// Mas_Ord_New             Last known order array
// int TralingStop         Value of TralingStop (amount of points)
//---------------------------------------------------------------------------- 2 --
int Tral_Stop(int Tip)
  {
   int Ticket;                      // Order ticket
   double
   Price,                           // Market order open price
   TS,                              // TralingStop (as related to the price)
   SL,                              // Value of order StopLoss
   TP;                              // Value of order TakeProfit
   bool Modify;                     // A criterion to modify.
//---------------------------------------------------------------------------- 3 --
   for(int i=1;i<=Mas_Ord_New[0][0];i++)  // Cycle for all orders
     {                                    // Searching for orders of the given type
      if (Mas_Ord_New[i][6]!=Tip)         // If this is not our type..
         continue;                        //.. skip the order
      Modify=false;                       // It is not assigned to be modified
      Price =Mas_Ord_New[i][1];           // Order open price
      SL    =Mas_Ord_New[i][2];           // Value of order StopLoss
      TP    =Mas_Ord_New[i][3];           // Value of order TakeProft
      Ticket=Mas_Ord_New[i][4];           // Order ticket
      if (TralingStop<Level_new)          // If it is less than allowed..
         TralingStop=Level_new;           // .. then the allowed one
      TS=TralingStop*Point;               // The same in the relat, price value
      //---------------------------------------------------------------------- 4 --
      switch(Tip)                         // Go to Order type
        {
         case 0 :                         // Order Buy
            if (NormalizeDouble(SL,Digits)<// If it is lower than we want..
               NormalizeDouble(Bid-TS,Digits))
              {                           // ..then modify it:
               SL=Bid-TS;                 // Its new StopLoss
               Modify=true;               // Assigned to be modified.
              }
            break;                        // Exit 'switch'
         case 1 :                         // Order Sell
            if (NormalizeDouble(SL,Digits)>// If it is higher than we want..
               NormalizeDouble(Ask+TS,Digits)||
               NormalizeDouble(SL,Digits)==0)//.. or zero(!)
              {                           // ..then modify it
               SL=Ask+TS;                 // Its new StopLoss
               Modify=true;               // Assigned to be modified.
              }
        }                                 // End of 'switch'
      if (Modify==false)                  // If there is no need to modify it..
         continue;                        // ..then continue the cycle
      bool Ans=OrderModify(Ticket,Price,SL,TP,0);//Modify it!
      //---------------------------------------------------------------------- 5 --
      if (Ans==false)                     // Failed :(
        {                                 // Check for errors:
         if(Errors(GetLastError())==false)// If the error is critical,
            return;                       // .. then leave.
         i--;                             // Decreasing counter
        }
      Terminal();                         // Order accounting function
      Events();                           // Event tracking
     }
   return;                                // Exit the user-defined function
  }
//---------------------------------------------------------------------------- 6 --
```

In blocks 1-3, the global variables are described that are used in the function, as well as local variables are opened and described. In the cycle 'for' (blocks 3-6), the orders of the given type are selected and, if the StopLoss of any of those orders is further from the current price than it was set by the user, the order is modified.

To make the code more human-oriented, the values of some elements of the order array Mas_Ord_New are assigned to simple variables (block 3-4). Then the necessary check will be made for the variable TralingStop: If the value of this variable is less than the minimum allowed distance set by the dealing center, it will be increased up to the minimum allowed value.

In block 4-5, according to the order type, necessary calculations are made. For example, if the value of the transferred parameter Tip is 1 (an order Sell should be modified), the control will be passed to the mark 'case 1' of the operator 'switch'. The necessity to modify the order StopLoss is checked here (according to the rules that apply to this order type, see Requirements and Limitations in Making Trades). If no StopLoss is set or if it is set on a distance further than the value of TralingStop from the current market price, the desired new value of StopLoss is calculated. Trade request for modification of the order is formed in line:

```
        bool Ans = OrderModify(Ticket,Price,SL,TP,0);//Modify it!
```

It was noted before that the trading strategy considered here implied the availability of only one market order. Nevertheless, the function Tral_Stop() provides the possibility to modify several market orders of one type. If the trader does not intervene into trading during the work of the EA, no necessity to modify several orders occurs. However, if the trader opens a market order manually (in addition to those already opened), we have to decide which of the orders available should be modified as the first and why.

When considering the sequence of closing several orders, we mentioned that the criterion defining the priority in closing of orders was the amount of lots. This solution is obvious - the more lots (of the total amount) are closed, the sooner the EA will response to the triggering of the closing criterion. The problem of order modification sequence has no unambiguous solution. In all cases, the criterion for order modification sequence is determined by the essence of the trading strategy. This criterion may be both the amount of lots, the fact of no StopLoss at one of the orders, the distance from StopLoss to the current price. In a number of cases, this criterion may be expressed through an overall index - the size of loss that may result from sharp price changes, i.e., when all market orders are automatically closed by StopLoss at the same time.

In the above example of the function Tral_Stop(), a random sequence of order modification is realized - the orders are modified in the sequence, in which they occur in the lost of open market orders and placed pending orders. In each specific case, the function must be refined upon - the order modification sequence must be programmed according to the rules of your specific trading strategy.

Special attention should be paid to the fact that all trades are made in the real-time mode. If there are too many orders, the EA will generate a great variety of trade requests. Obviously, the market may turn around while those requests are being executed. However, the function doesn't return the control to the function Trade() that has called to it until all orders that must be modified are modified. This means that the danger of omitting a trade request for closing or opening of orders may occur. For this reason, any strategy must be coded in such a way that not to allow a considerable amount of market orders to be available at a time.

In block 5-6, the errors got during the execution of trade requests are analyzed. If the error is critical, the function will end its operations. However, if an overcomable error has been got, the value of the counter 'i' is decreased by 1. It will be done in order to produce one more attempt to modify the same order at the next iteration of the cycle 'for'.

In most cases, the above code will comply with the necessity to modify several orders. At the same time, if any changes take place in the orders (for example, an order will be closed when the market price reaches one of the stop levels) within the period of several failed attempts to modify orders, the sequence of orders in the array Mas_Ord_New may also change. This will result in that an order may be omitted and not modified within the period of the last launching of the special function start(). This situation can be improved at the next tick, at the next launch of the function start().

← Function Defining Trading Criteria Error Processing Function →

Error Processing Function

The errors that appear during trade orders execution can be divided into two groups - overcomable (non-critical) errors and critical errors. Overcomable errors are those occurring at server faults. After they have been eliminated, you can continue trading. For example, a request can be rejected by the broker, if there is no information about the current quotes at the moment. This kind of situation may appear in a slow market, i.e., when ticks don't income frequently. Or, on the contrary, the broker cannot always execute a plenty of requests from traders in an active market where too many quotes are coming. Then the pause appears before the order is executed or - sometimes - a denial. In such cases, the Expert Advisor may continue its working and, for example, repeat the request a bit later after the execution of some code related to the error code.

Critical errors include all errors that alert about serious troubles. For example, if an account is blocked, then there is no point in sending trade requests. In such case, the EA should display the corresponding message and shouldn't repeat the request. The error processing function must be used indispensably in a normal EA.

Custom Error Processing Function Errors()

```
bool Errors( int Error )
```

The function returns TRUE, if the error is overcomable. Otherwise, it returns FALSE.
The **Error** parameter can have any value and correspond with any error code that occurred while trying to make a trade.

The user-defined error processing function Errors() is designed as the include file Errors.mqh:

```
//--------------------------------------------------------------------
// Errors.mqh
// The code should be used for educational purpose only.
//--------------------------------------------------------------- 1 --
// Error processing function.
// Returned values:
// true  - if the error is overcomable (i.e. work can be continued)
// false - if the error is critical (i.e. trading is impossible)
//--------------------------------------------------------------- 2 --
bool Errors(int Error)                    // Custom function
  {
   // Error             // Error number
   if(Error==0)
      return(false);                      // No error
   Inform(15,Error);                      // Message
//--------------------------------------------------------------- 3 --
   switch(Error)
     {   // Overcomable errors:
      case 129:         // Wrong price
      case 135:         // Price changed
         RefreshRates();                  // Renew data
         return(true);                    // Error is overcomable
      case 136:         // No quotes. Waiting for the tick to come
         while(RefreshRates()==false)     // Before new tick
            Sleep(1);                     // Delay in the cycle
         return(true);                    // Error is overcomable
      case 146:         // The trade subsystem is busy
         Sleep(500);                      // Simple solution
         RefreshRates();                  // Renew data
         return(true);                    // Error is overcomable
         // Critical errors:
      case 2 :          // Common error
      case 5 :          // Old version of the client terminal
      case 64:          // Account blocked
      case 133:         // Trading is prohibited
      default:          // Other variants
         return(false);                   // Critical error
     }
//--------------------------------------------------------------- 4 --
  }
//--------------------------------------------------------------------
```

One of the questions that arise when composing the algorithms of error processing function Errors() is: What should the function return, if the value of a given parameter is 0 (i.e., there are no errors). This kind of situation should not appear in a correctly coded EA. However, the code can be variously modified while enhancing the program, so sometimes the value of an error can be equal to 0. So, it would be reasonable to add some lines to the function at the primal stage of developing (block 2-3), for the situations where Error is equal to 0.

The reaction of the Errors() function to the zero value of the Error variable depends on the algorithm used for processing the values returned by the function. The value returned by the function is taken into consideration in the executable trade function of the above EA. If the Errors() function returns 'true' (error is overcomable), then the program will retry to make a trade. If it returns 'false', then the trade function stops and the control is sequentially passed to the calling function, then to the start() function, and then to the client terminal. If the choice is between these two alternatives, then the situation when there are no errors (Error=0) corresponds with the second alternative, namely, with the 'false' value returned. It guaranties that a once executed request will not be repeated.

Once the message about the error is displayed by the Inform() function, the control is passed to block 3-4, to the operator 'switch'. The specific variant case is involved for every considered error code. For example, if error 136 occurs, it means that the broker does not have current quotes for making a proper decision. This means that the situation won't change unless a new tick comes, so there is no need to repeat sending the same trade request because it won't be executed, anyway. The right solution in this situation is the pause - the absence of any initiative from the EA. A simple method to detect a new tick is used for this purpose - the analysis of the value returned by the RefreshRates() function. The control will be returned to the called function, in which the trade request is repeated (after the corresponding analysis, if necessary), as soon as the new tick comes.

If there is an error the programmer considers to be a critical one, the function returns 'false'. The request will not be repeated, in such case, so there is no need to do anything in the Errors() function. All errors not processed are considered as critical by default. You can expand the list of processable errors (see Error Codes).

← Trade Functions About Complex Programs →

General Characteristics of Complex Programs

There is no single formal feature that distinguishes a customary program from a complex one. In general, complex programs positively differ in a variety of tools provided and quantity of information processed. Only some of qualitative adjectives that are peculiar to complex programs can be denoted.

Program Execution Order

As a rule, a usual program contains its code in the special function start() that is started for execution by the client terminal. In most cases function start() execution time is considerably less than the tick period. This means that most of the time the program is waiting for a tick to come. This kind of processes are characterized by the on-off ratio term. On-off ratio is the ratio of a repeating process period to the duration of the process itself. The execution time of start() **T1** is nearly from 10 to 100 milliseconds, and the **T2** time between ticks is 5 seconds at the average. Thus, the on-off ratio of a working EA is reaching **T2/T1**=1000 and more (see fig. 159). That is to say that the duration of a usual operating EA's effective capacity is 0,1% of the whole time, the rest of the time it is standing.

Sometimes complex calculations can be executed by an EA and as a result the duration of start() execution can be longer and reach tens of seconds. In these cases start() function will not be started on every tick but only on the ticks that came while the start() is waiting for them. The fig. 159 shows that the tick that came at the execution of the start() function period (at the t4 moment) will not cause a new special function's start. The next time the start() function will start at the t5 moment. The pause between the end of the current execution and the beginning of the next execution of the start() function will appear with this provision.

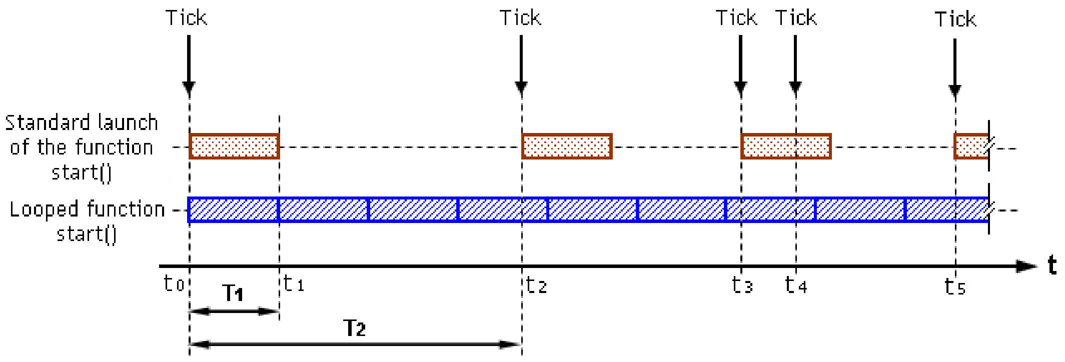


Fig. 159 Different on-off ratio while the start() function is executed by the client terminal and the cycled start() function.

There is a method to increase the effective capacity of the program essentially, thus decreasing on-off ratio of the trade management process. For this purpose let us implement an algorithm, according to which the main code is many times (infinitely) repeated during start() execution (allowed only in EAs and scripts). The example of the looped start() is shown below:

```
//--------------------------------------------------------------------
start()                           // Special function start()
   {
   while(!IsStopped())            // Until user..
      {                           // ..stops execution of the program
      RefreshRates();             // Data renewal
      //......................The main code of the program is specified here
      Sleep(5);                   // Short pause
      }
   return;                        // Control is returned to the terminal
   }
//--------------------------------------------------------------------
```

The whole code is specified in the body of the "while" cycle operator, and the only way to exit the cycle is to receive a command from the client terminal to finish the program execution. If the start() function built on this specified principle is started, it will be executed infinitely long and will return control to the client terminal only when a user manually removes the EA from a security window or providing some other conditions (see Special Functions).

The execution of the cycled start() function is running uninterruptedly, that is why there is no period when the program is in the mode of waiting for a new tick (see Fig. 159), so the on-off ratio of the process of the cycled program execution is equal to 1. The start() function that is based on the specified principle is started by the client terminal only once. This means that the information updating (e.g. market quotes) must be compulsory performed using the RefreshRates() function. In order to avoid the large consumption of much resources a short pause at the end of the cycle can be specified.

The development of the looped program requires much attention while composing the algorithm. For example, the reaction of a usual program on a received critical error is breaking the start() function execution and returning control to the client terminal. A cycled program keeps control permanently while running so the other reaction must be anticipated, for example the prohibition of trade orders generation over some period. Yet the temporary prohibition should not hinder the program execution. During the whole execution period, the program should process all the available information about events, including the controlling actions of a user. In general, such a program has incommensurable power as compared to a usual one.

Available Tools

Use of lopped programs makes sense only if the continuity of a program execution is effectively used. For example, such a program can process a trader's control actions. The modification of coordinates of graphical objects or the fact of attaching other programs - scripts and indicators can be considered as controlling actions.

A simple program may respond to some events (including user-initiated) at a regular start of the special function start() on the nearest tick, as a rule. While a cycled program can process all events immediately (!). In this case the lag can only be for a short time, no more than the time of execution of one cycle of the start() function (nearly no more than 10-100 ms).

A complex program may use graphical objects to display order characteristics or rules of its modification. For example, orders of all type are shown in a security window in green lines, stop orders - in red lines. And if several orders are shown on the screen simultaneously, it is quite difficult to detect what line belongs to this or that order. But if we apply the graphical object "horizontal line" of a necessary color and style to each order line, it will be much easier to differentiate between orders and their stop orders.

Besides, the fact of changing the coordinates of such objects can be perceived by a program as a guide to action. For example, if a user shifts a horizontal line showing a pending order several points up, as a result of this action the program may form and send to the server a trade request, according to which the order should be modified, i.e. the preset open price should be increased by several points (for an instant execution the use of a looped program is obligatory). Thus a complex program may offer the possibility to manage trading using a mouse.

Function used for modification of separate stop-orders or a declared open price of an order can be used in complex programs as well. If such a function is used applicable to one of order lines, a graphical object, for example "an arrow" can be displayed near the order line, indicative of the function activity.

Using graphical objects you can set up trading scenario. For example, setting the "Pivot" flag at some distance from the current price, you can inform the program that it is necessary to close the order and open a new one in the opposite direction when the specified price is reached. Similarly you can specify modification level limits, the order closing price, etc. in the program. The use of graphical objects that display the settings of a program considerably increases the awareness of the trader about present and planned events.

Sound signals associated with events are also used in complex programs. Using sounds in a program allows the trader to leave the PC and orientate through events by the sound signal types (melodies, vocalized text, etc.).

The effectiveness of the trading often depends on the fact whether the time of important economical and political news release is taken into account. Large majority of dealing centers provide traders with the list of the news for the nearest week with the denotement of release time and importance. The information about coming events is recorded into a file. During the operation a program reads the information from the file and performs one or another trade scenario depending on the importance of a coming event. For example, a program can delete pending orders, modify stop-orders of market orders, etc. Shortly before the important news coming the program can terminate trading - close all orders preliminarily informing a trader.

Automated and Manual Mode of Program Operation

A complex program is characterized by a more complex algorithm of events processing. Particularly, this kind of program presupposes some reaction to a trader's interference with the trading process. For example, if a trading strategy allows only one market order and a trader opens one more order, a complex program, first, monitors this kind of event and then starts the execution of the algorithm part provided for such an occasion. The program can warn a trader about an unauthorized interference, at first, and offer to delete the odd order independently. If it does not happen, the program (depending on the settings) can delete the odd order or exit the automated trading mode informing the trader previously.

If a program is started for execution when there are already multiple orders placed, necessary actions will be performed depending on its settings. For example, the program can close all opposite orders without a trader's agreement. If a trading strategy does not allow pending orders, they will be deleted in the priority sequence - firstly, nearest to the market quotation, then more expensive ones, etc.

Once the trader has set the quantity and the quality limits of orders in a trading strategy, the program (working all the time and monitoring the events) can propose a trader to activate the automated trading mode, and if a trader agrees, designate the pursuing trading scenario using graphical objects.

Every trader has its own set of preferences when working with a program. Some traders admit only the automated trading, other traders - half-automated, thirds prefer only the manual mode. A correctly designed program must cover all the requirements, i.e. must have a number of settings that provide different usage mode. For example, a program can act as an adviser in the manual mode of working - display a text containing direct recommendations and also graphical objects displaying a trend direction, forecast pivot points, etc. A program can ask a trader to permit order opening, admit trader interference in orders management (e.g, manual stop-orders modification) while working in the half-automated mode. In case the program is running in the automated mode any trader's interference in the trading process can be considered as a signal for changing the mode to half-automated or manual.

All the described properties of a program can be embodied on the MQL4 programming language basis that is specially designed for this purpose. A correctly designed complex program has incontestable number of advantages, a trader quickly gets used to them and starts using them in trading.

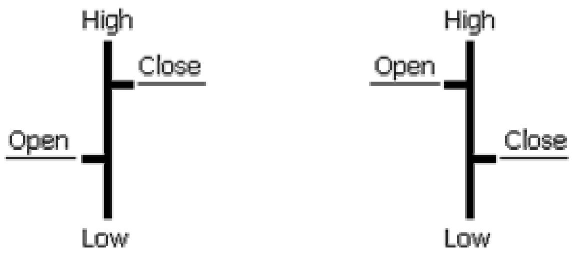
← Error Processing Function Appendixes →

Appendixes

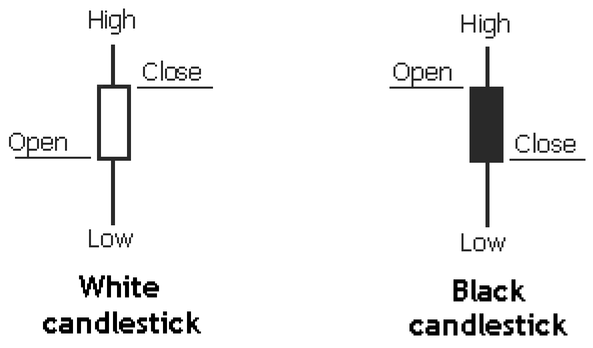
← About Complex Programs Glossary →

Glossary

- **Algorithm** is a precise instruction for completing a preset act sequence; control in an executable program is transferred according to the algorithm.
- **Application Program** is a program coded in MQL4 and used in MetaTrader 4 Client Terminal; it can be an Expert Advisor, a Script or an Indicator.
- **Array** - is an arranged set of the values of one-type variables that have a common name. Arrays can be one-dimensional and multidimensional. The maximum admissible amount of dimensions in an array is four. Arrays of any data types are allowed.
- **Array Component** is a part of an array; it is an indexed variable having the same name and a value.
- **Ask** is the higher of two prices offered by broker in a Two-Way Quote for a Security.
- **Bar** is a graphical representation of a price chart. Bar is characterized by Open, Close, High, Low prices, as well as by Volume and Time (see also Candlestick).



- **Bid** is the lower of two prices offered by broker in a Two-Way Quote for a Security.
- **Buffer** is a memory area containing numeric values of an indicator array.
- **Built-In Function** is the same as a standard function.
- **Buy** is a market order that defines buying of assets for a security.
- **BuyLimit** is a pending order to buy assets for a security at a price lower than the current one. The order will be executed (modified into market order Buy) if the Ask price reaches or falls below the price set in the pending order.
- **BuyStop** is a pending order to buy assets for a security at a price higher than the current one. The order will be executed (modified into market order Buy) if the Ask price reaches or rises above the price set in the pending order.
- **Candlestick** is a graphical representation of a price chart. Candlestick is characterized by Open, Close, High, Low prices, as well as by Volume and Time. The can be black candlesticks and white candlesticks (see also Bar).



- **Comment** is an optional and nonexecutable part of a program.

- **Constant** is a program constituent; it is an object that has a value.
- **Constant Expression** is an expression consisting of constants and operations, for example: 2+3*7. A constant expression is calculated at compile time.
- **Control** is execution of actions predefined by the program algorithm and by the properties of the client terminal. Control may be transferred within a program from one program line to another, as well as between the program and the client terminal (see Some Basic Concepts).
- **Currency Instrument** is a currency pair, for example, EURUSD, GBPCHF, etc.; it is a particular security (symbol).
- **Custom Indicator** is an application program coded in MQL4; it is basically intended for graphical displaying of preliminarily calculated dependences. The values of components of Indicator Arrays of a custom indicator are available to other applications through the function named iCustom() (see also Technical Indicator).
- **EA** is an abbreviation for Expert Advisor (see also Script and Indicator).
- **Expert Advisor** is a program coded in MQL4; it is distinguished by the properties of special function start() called by the client terminal to be executed on every tick; the main purpose of Expert Advisors is programmed control over trades (see also EA, Script and Indicator).
- **Expression** is a sequence of operands and operations; it is a program record, the calculated value of which is characterized by a data type.
- **External Variable** is a variable, the value of which is available from the program properties window; it has the properties of a global variable.
- **File Descriptor** is the unique number of a file opened by an executable program at the current moment.
- **File Pointer** a location in a file where the reading of the next value starts. As the data are read, the pointer shifts to the right by one or several positions.
- **File Separator** is a special character; it is a record to be stored in the file to separate data records.
- **Flag** is a variable the value of which is placed in correspondence with some events or facts.
- **Formal Parameters** represent a list of variables given in the header of Function Description (see also Functions and Function Description and Operator 'return').
- **Function** is a named, specific part of a program that describes the method of data conversion. The use of a function in a program involves Function Description and Function Call. There can be special, standard (built-in) and custom functions (see also Functions and Special Functions).
- **Function Body** is one or several operators being the executable part of Function Description.
- **Function Call** (or **Function Reference**) is a record, execution of which results in execution of a function (see also Function Description).
- **Function Description** is a specific named part of a program intended for execution; it consists of two basic parts - a function header and a function body; it is used in special and custom functions (see also Functions, Function Description and Operator return and Special Functions).
- **Function Header** is a part of function description; it consists of the type of the return value, function name and the list of formal parameters. The list of formal parameters is enclosed in parentheses and placed after the function name.
- **Function Reference** is the same as Function Call.

- **Global Variable** is a variable declared beyond all functions. The Scope of global variables is the entire program.
- **Global Variable of Client Terminal** is a variable, the value of which is available from all applications launched in the client terminal (abbreviated form: GV).
- **Graphical Object** - is a shape in the symbol window; the shape can be selected, moved, modified or deleted.
- **GV** is an abbreviation for Global Variable of the Client Terminal.
- **Indicator** is a built-in function of the client terminal or a program coded in MQL4; the main purpose of indicators is to display Indicator Lines on the screen; indicators cannot make trades; there are two types of indicators: Custom Indicators and Technical Indicators (see also Expert Advisor, EA and Script).
- **Indicator Array** is a one-dimensional array containing numeric values that are the basis for constructing of Indicator Line.
- **Indicator Line** is a graphical display of a certain dependence based on numeric values included in an Indicator Array.
- **Initialization of Variable** is assigning a type-dependent value to the variable at Variable Declaration.
- **Iteration** is a repeated execution of some calculations; it is used to note that the program lines composing the cycle operator body (see also Cycle Operator "while" and Cycle Operator "for") are executed.
- **Local Variable** is a variable declared within a function. The Scope of local variables is the body of the function, in which the variable is declared.
- **Local Time** is the time set on a local PC (see also Server Time).
- **Loop (Cycle) Body** is one or several operators included in braces; it is located directly after the cycle operator header (see Cycle Operator "while" и Cycle Operator "for").
- **Looping** is a continuously repeated execution of operators composing the loop (cycle) body; it's a critical situation that results from realization of a wrong algorithm.
- **Market Order** is an executed order to buy or sell assets for a symbol (security). A market order is displayed in the symbol window and in the 'Terminal' window until the order is closed (see also Pending Order).
- **Normalized Price** is a price rounded off to one Point size for a security (symbol).
- **Operand** is a Constant, a Variable, an array component or a value returned by a function (see Function Call).
- **Operation** is an action made upon Operands (see also Operation Symbol).
- **Operation Symbol** is a preset character or a group of characters that order to execute an operation.
- **Operator** is a part of a program; it is a phrase in an algorithmic language that prescribes a certain method of data conversion. There can be simple and compound operators.
- **Operator Format** is a set of rules for formatting of an operator of the given type. Each operator type has its own format (see also Operators).
- **Opposite (Counter) Order** is a market order opened in the direction opposite to the direction of another market order opened for the same symbol.
- **Pending Order** is a trade order to buy or sell assets for a security (a symbol) when the preset price level is reached. The pending order is displayed in the symbol window and in the 'Terminal' window

until it becomes a market order or is deleted (see also Market Order).

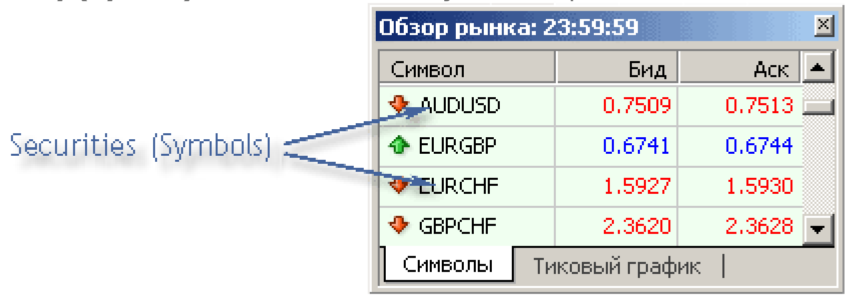
- **Point** is the unit of price measurement for a security (the minimum possible price change, the last significant figure of the price value).

- **Predefined Variable** is a variable with a predefined name; the value of such variable is defined by the client terminal and cannot be changed by coding (see also Predefined Variables).

- **Regular Loop Exit** is transfer of control outside the cycle operator as a result of execution of a condition placed in the cycle operator header (see also Special Loop Exit).

- **Script** is a program coded in MQL4; it is marked by properties of the special function of start() called by the client terminal to be executed **only once**; scripts are intended for performing any operations that should be implicitly executed only once (see also Expert Advisor, EA and Indicator).

- **Security (Symbol)** is the name of the object to be quoted.



Обзор рынка: 23:59:59

Символ	Бид	Аск
AUDUSD	0.7509	0.7513
EURGBP	0.6741	0.6744
EURCHF	1.5927	1.5930
GBPCHF	2.3620	2.3628

- **Sell** is a market order that defines selling of assets for a security.

- **SellLimit** is a pending order to sell assets for a security at a price higher than the current one. The order will be executed (modified into market order Sell) if the Bid price reaches or rises above the price set in the pending order.

- **SellStop** is a pending order to sell assets for a security at a price lower than the current one. The order will be executed (modified into market order Sell) if the Bid price reaches or falls below the price set in the pending order.

- **Server Time** is the time set on the server (see also Local Time).

- **Special Function** is a function that has one of predefined names (**init(), start()** and **deinit()**) and is called to be executed by the client terminal; it also has its own special features (see Special Functions).

- **Special Loop Exit** is transfer of control outside the cycle operator as a result of execution of operator 'break' included in the cycle operator body (see also Regular Loop Exit).

- **Spread** is the difference between the larger and the smaller price in points in a two-way quote for a security.

- **Standard Function** is the same as built-in function; it is a function created by the developers of MQL4, it has a predefined name and predefined properties; the description of standard functions in a program is not specified; properties of standard functions are described in details in MQL4 Reference (see Functions and Standard Functions).

- **StopLoss** is a stop order; it is a price set by trader, at which a market order will be closed if the symbol price moves in a direction that produces losses for the order.

- **TakeProfit** is a stop order; it is a price set by trader, at which a market order will be closed if the symbol price moves in a direction that produces profits for the order.

- **Technical Indicator** is a part of online trading platform MetaTrader 4; it is a built-in function that allows us to display a certain dependence on the screen (see also Custom Indicator).

- **Tick** is an event characterized by a new price for a symbol at some moment.
- **Timeframe** is a period of time, within which one price bar is formed; there are several standard timeframes: M1, M5, M15, M30, H1, H4, D1, W1 and MN (1 min, 5 min, 15 min, 30 min, 1 h, 4 h, 1 day, 1 week, and 1 month, respectively)
- **Time-Series Array** is an array with a predefined name (Open, Close, High, Low, Volume or Time); its components contain the values of the corresponding characteristics of historical bars.
- **Trade** is opening, closing or modification of market and pending orders.
- **Trade Request** is a command made by a program or by a trader in order to perform a Trade. The order may be executed on or rejected by the server or by client terminal.
- **Trader** is a person that trades on financial markets for purposes of profit.
- **Two-Way Quotes** are a connected pair of market prices offered by the broker for buying and selling of assets for a security at the moment.
- **Typecasting** is modifying (mapping) of types of the values of an Operand or an Expression. Before execution of Operations (all but assignment operations), they are modified to a type of the highest priority, whereas before execution of assignment operations they are modified to the target type.
- **User-Defined Function** is a function created by the programmer (see also Function).
- **Variable** is a part of a program; it is an object having a name and a value.
- **Variable Declaration** is the first mentioning of a variable in a program. At variable declaration its type is specified.
- **Variable Identifier** is a set of characters consisting of letters, numbers and underscore(s), beginning with a letter, at most 31 characters long. It is the same as the Variable Name.
- **Variable Name** is the same as Variable Identifier.
- **Variable Scope** is a location in a program where the value of the variable is available. Every variable has its scope (see also Local Variable and Global Variable).
- **Zero Bar** is the current bar not yet completely formed. In a symbol window, the zero bar is displayed in the rightmost position.

← Appendixes Types of Trades →

Types of Trades

A trade type in the OrderSend() function can be indicated as a predefined constant or as its value according to a trade type:

Constant	Value	Trading Operation
OP_BUY	0	Buy
OP_SELL	1	Sell
OP_BUYLIMIT	2	Pending order BUY LIMIT
OP_SELLLIMIT	3	Pending order SELL LIMIT
OP_BUYSTOP	4	Pending order BUY STOP
OP_SELLSTOP	5	Pending order SELL STOP

← Glossary Requirements and Limitations in Making Trades →

Requirements and Limitations in Making Trades

Tables below show calculation values that limit the conduction of trades when opening, closing, placing, deleting or modifying orders.

To get the minimum distance to StopLevel and freezing distance FreezeLevel the MarketInfo() function should be called.

Requirements.

Correct prices used when performing trade operations.

Order Type	Open Price	Close Price	Open Price of a Pending Order	Transforming a Pending Order into aMarket Order
Buy	Ask	Bid		
Sell	Bid	Ask		
BuyLimit			Below the current Ask price	Ask price reaches open price
SellLimit			Above the current Bid price	Bid price reaches open price
BuyStop			Above the current Ask price	Ask price reaches open price
SellStop			Below the current Bid price	Bid price reaches open price

The possibility of deleting a pending order is regulated by the FreezeLevel parameter.

StopLevel Minimum Distance Limitation.

A trade operation will not be performed if any of the following conditions is disrupted.

Order Type	Open Price	StopLoss (SL)	TakeProfit (TP)
Buy	Modification is prohibited	Bid-SL ≥ StopLevel	TP-Bid ≥ StopLevel
Sell	Modification is prohibited	SL-Ask ≥ StopLevel	Ask-TP ≥ StopLevel
BuyLimit	Ask-OpenPrice ≥ StopLevel	OpenPrice-SL ≥ StopLevel	TP-OpenPrice ≥ StopLevel
SellLimit	Bid-OpenPrice ≥ StopLevel	SL-OpenPrice ≥StopLevel	OpenPrice-TP ≥ StopLevel
BuyStop	OpenPrice-Ask ≥ StopLevel	OpenPrice-SL ≥ StopLevel	TP-OpenPrice ≥ StopLevel
SellStop	Bid-OpenPrice ≥ StopLevel	SL-OpenPrice ≥ StopLevel	OpenPrice-TP ≥ StopLevel

FreezeLevel Limitation (Freezing Distance).

Market orders can not be closed if the StopLoss and TakeProfit values violate the FreezLevel parameter requirements.
StopLoss or TakeProfit orders can not be modified if StopLoss or TakeProfit values violate the StopLevel parameter requirements.
Pending orders can not be deleted or modified if the declared open price violates the FreezeLevel parameter requirements.

Order Type	Open Price	StopLoss (SL)	TakeProfit (TP)
Buy	Modification is prohibited	Bid-SL > FreezeLevel	TP-Bid > FreezeLevel
Sell	Modification is prohibited	SL-Ask > FreezeLevel	Ask-TP > FreezeLevel

BuyLimit	Ask-OpenPrice > FreezeLevel	Regulated by the StopLevel parameter	Regulated by the StopLevel parameter
SellLimit	OpenPrice-Bid > FreezeLevel	Regulated by the StopLevel parameter	Regulated by the StopLevel parameter
BuyStop	OpenPrice-Ask > FreezeLevel	Regulated by the StopLevel parameter	Regulated by the StopLevel parameter
SellStop	Bid-OpenPrice > FreezeLevel	Regulated by the StopLevel parameter	Regulated by the StopLevel parameter

← Types of Trades Error Codes →

Error Codes

GetLastError() - the function that returns codes of error. Code constants of errors are determined in stderror.mqh file. To draw the text messages use the ErrorDescription() function described in the stdlib.mqh file.

Error codes returned from a trade server or client terminal:

Constant	Value	Description
ERR_NO_ERROR	0	No error returned.
ERR_NO_RESULT	1	No error returned, but the result is unknown.
ERR_COMMON_ERROR	2	Common error.
ERR_INVALID_TRADE_PARAMETERS	3	Invalid trade parameters.
ERR_SERVER_BUSY	4	Trade server is busy.
ERR_OLD_VERSION	5	Old version of the client terminal.
ERR_NO_CONNECTION	6	No connection with trade server.
ERR_NOT_ENOUGH_RIGHTS	7	Not enough rights.
ERR_TOO_FREQUENT_REQUESTS	8	Too frequent requests.
ERR_MALFUNCTIONAL_TRADE	9	Malfunctional trade operation.
ERR_ACCOUNT_DISABLED	64	Account disabled.
ERR_INVALID_ACCOUNT	65	Invalid account.
ERR_TRADE_TIMEOUT	128	Trade timeout.
ERR_INVALID_PRICE	129	Invalid price.
ERR_INVALID_STOPS	130	Invalid stops.
ERR_INVALID_TRADE_VOLUME	131	Invalid trade volume.
ERR_MARKET_CLOSED	132	Market is closed.
ERR_TRADE_DISABLED	133	Trade is disabled.
ERR_NOT_ENOUGH_MONEY	134	Not enough money.
ERR_PRICE_CHANGED	135	Price changed.
ERR_OFF_QUOTES	136	Off quotes.
ERR_BROKER_BUSY	137	Broker is busy.
ERR_REQUOTE	138	Requote.
ERR_ORDER_LOCKED	139	Order is locked.
ERR_LONG_POSITIONS_ONLY_ALLOWED	140	Long positions only allowed.
ERR_TOO_MANY_REQUESTS	141	Too many requests.
ERR_TRADE_MODIFY_DENIED	145	Modification denied because an order is too close to market.
ERR_TRADE_CONTEXT_BUSY	146	Trade context is busy.
ERR_TRADE_EXPIRATION_DENIED	147	Expirations are denied by broker.
ERR_TRADE_TOO_MANY_ORDERS	148	The amount of opened and pending orders has reached the limit set by a broker.

MQL4 run time error codes:

Constant	Value	Description
ERR_NO_MQLERROR	4000	No error.
ERR_WRONG_FUNCTION_POINTER	4001	Wrong function pointer.
ERR_ARRAY_INDEX_OUT_OF_RANGE	4002	Array index is out of range.
ERR_NO_MEMORY_FOR_FUNCTION_CALL_STACK	4003	No memory for function call stack.
ERR_RECURSIVE_STACK_OVERFLOW	4004	Recursive stack overflow.
ERR_NOT_ENOUGH_STACK_FOR_PARAMETER	4005	Not enough stack for parameter.
ERR_NO_MEMORY_FOR_PARAMETER_STRING	4006	No memory for parameter string.
ERR_NO_MEMORY_FOR_TEMP_STRING	4007	No memory for temp string.
ERR_NOT_INITIALIZED_STRING	4008	Not initialized string.
ERR_NOT_INITIALIZED_ARRAYSTRING	4009	Not initialized string in an array.
ERR_NO_MEMORY_FOR_ARRAYSTRING	4010	No memory for an array string.
ERR_TOO_LONG_STRING	4011	Too long string.
ERR_REMAINDER_FROM_ZERO_DIVIDE	4012	Remainder from zero divide.
ERR_ZERO_DIVIDE	4013	Zero divide.
ERR_UNKNOWN_COMMAND	4014	Unknown command.
ERR_WRONG_JUMP	4015	Wrong jump.
ERR_NOT_INITIALIZED_ARRAY	4016	Not initialized array.
ERR_DLL_CALLS_NOT_ALLOWED	4017	DLL calls are not allowed.
ERR_CANNOT_LOAD_LIBRARY	4018	Cannot load library.
ERR_CANNOT_CALL_FUNCTION	4019	Cannot call function.
ERR_EXTERNAL_EXPERT_CALLS_NOT_ALLOWED	4020	EA function calls are not allowed.
ERR_NOT_ENOUGH_MEMORY_FOR_RETURNED_STRING	4021	Not enough memory for a string returned from a function.
ERR_SYSTEM_BUSY	4022	System is busy.
ERR_INVALID_FUNCTION_PARAMETERS_COUNT	4050	Invalid function parameters count.
ERR_INVALID_FUNCTION_PARAMETER_VALUE	4051	Invalid function parameter value.
ERR_STRING_FUNCTION_INTERNAL_ERROR	4052	String function internal error.
ERR_SOME_ARRAY_ERROR	4053	Some array error.
ERR_INCORRECT_SERIES_ARRAY_USING	4054	Incorrect series array using.
ERR_CUSTOM_INDICATOR_ERROR	4055	Custom indicator error.
ERR_INCOMPATIBLE_ARRAYS	4056	Arrays are incompatible.
ERR_GLOBAL_VARIABLES_PROCESSING_ERROR	4057	Global variables processing error.
ERR_GLOBAL_VARIABLE_NOT_FOUND	4058	Global variable not found.
ERR_FUNCTION_NOT_ALLOWED_IN_TESTING_MODE	4059	Function is not allowed in testing mode.
ERR_FUNCTION_NOT_CONFIRMED	4060	Function is not confirmed.
ERR_SEND_MAIL_ERROR	4061	Mail sending error.
ERR_STRING_PARAMETER_EXPECTED	4062	String parameter expected.
ERR_INTEGER_PARAMETER_EXPECTED	4063	Integer parameter expected.
ERR_DOUBLE_PARAMETER_EXPECTED	4064	Double parameter expected.
ERR_ARRAY_AS_PARAMETER_EXPECTED	4065	Array as parameter expected.

ERR_HISTORY_WILL_UPDATED	4066	Requested history data in updating state.
ERR_TRADE_ERROR	4067	Some error in trade operation execution.
ERR_END_OF_FILE	4099	End of a file.
ERR_SOME_FILE_ERROR	4100	Some file error.
ERR_WRONG_FILE_NAME	4101	Wrong file name.
ERR_TOO_MANY_OPENED_FILES	4102	Too many opened files.
ERR_CANNOT_OPEN_FILE	4103	Cannot open file.
ERR_INCOMPATIBLE_ACCESS_TO_FILE	4104	Incompatible access to a file.
ERR_NO_ORDER_SELECTED	4105	No order selected.
ERR_UNKNOWN_SYMBOL	4106	Unknown symbol.
ERR_INVALID_PRICE_PARAM	4107	Invalid price.
ERR_INVALID_TICKET	4108	Invalid ticket.
ERR_TRADE_NOT_ALLOWED	4109	Trade is not allowed.
ERR_LONGS_NOT_ALLOWED	4110	Longs are not allowed.
ERR_SHORTS_NOT_ALLOWED	4111	Shorts are not allowed.
ERR_OBJECT_ALREADY_EXISTS	4200	Object already exists.
ERR_UNKNOWN_OBJECT_PROPERTY	4201	Unknown object property.
ERR_OBJECT_DOES_NOT_EXIST	4202	Object does not exist.
ERR_UNKNOWN_OBJECT_TYPE	4203	Unknown object type.
ERR_NO_OBJECT_NAME	4204	No object name.
ERR_OBJECT_COORDINATES_ERROR	4205	Object coordinates error.
ERR_NO_SPECIFIED_SUBWINDOW	4206	No specified subwindow.
ERR_SOME_OBJECT_ERROR	4207	Some error in object operation.

← Requirements and Limitations in Making Trades Styles of Indicator Lines Displaying →

Styles of Indicator Lines

Styles of drawing indicator lines for SetIndexStyle() and SetLevelStyle() functions:

Constant	Value	Description
DRAW_LINE	0	Simple line
DRAW_SECTION	1	Sections between nonempty line values
DRAW_HISTOGRAM	2	Histogram
DRAW_ARROW	3	Arrows (symbols)
DRAW_ZIGZAG	4	Drawing sections between even and odd indicator buffers
DRAW_NONE	12	No drawing

Line style. Used only if the line width is equal to 0 or 1:

Constant	Value	Description
STYLE_SOLID	0	Solid line
STYLE_DASH	1	Dashed line
STYLE_DOT	2	Dotted line
STYLE_DASHDOT	3	Dash-and-dot line
STYLE_DASHDOTDOT	4	Double dotted dash-and-dot line

← Error Codes Types and Properties of Graphical Objects →

Types and Properties of Graphical Objects

Graphical object type identifiers used with ObjectCreate(), ObjectsDeleteAll() and ObjectType() functions. It can be any of the following values:

Object Type	Value	Description
OBJ_VLINE	0	Vertical line. Uses time part of first coordinate, price is ignored.
OBJ_HLINE	1	Horizontal line. Uses price part of first coordinate, time is ignored.
OBJ_TREND	2	Trend line. Uses 2 coordinates.
OBJ_TRENDBYANGLE	3	Trend by angle. Uses 2 coordinates or 1 coordinate and angle. To set a line angle (OBJPROP_ANGLE property) use ObjectSet() function.
OBJ_REGRESSION	4	Regression. Uses time parts of first 2 coordinates, price parts are ignored.
OBJ_CHANNEL	5	Equidistant channel. Uses 3 coordinates.
OBJ_STDDEVCHANNEL	6	Standard deviation channel. Uses time parts of first two coordinates, price parts are ignored.
OBJ_GANNLINE	7	Gann line. Uses 2 coordinates, but price part of the second coordinate is ignored. To set the ratio between the price and time scales (OBJPROP_SCALE property) use ObjectSet() function.
OBJ_GANNFAN	8	Gann fan. Uses 2 coordinates, but price part of the second coordinate is ignored. To set the ratio between the price and time scales (OBJPROP_SCALE property) use ObjectSet() function.
OBJ_GANNGRID	9	Gann grid. Uses 2 coordinates, but price part of the second coordinate is ignored. To set the ratio between the price and time scales (OBJPROP_SCALE property) use ObjectSet() function.
OBJ_FIBO	10	Fibonacci retracement. Uses 2 coordinates. To set the number of levels (OBJPROP_FIBOLEVELS property) and the values of levels (OBJPROP_FIRSTLEVEL+n property) use ObjectSet() function.
OBJ_FIBOTIMES	11	Fibonacci time zones. Uses 2 coordinates. To set the number of levels (OBJPROP_FIBOLEVELS property) and the values of levels (OBJPROP_FIRSTLEVEL+n property) use ObjectSet() function.
OBJ_FIBOFAN	12	Fibonacci fan. Uses 2 coordinates. To set the number of levels (OBJPROP_FIBOLEVELS property) and the values of levels (OBJPROP_FIRSTLEVEL+n property) use ObjectSet() function.
OBJ_FIBOARC	13	Fibonacci arcs. Uses 2 coordinates. To set the number of levels (OBJPROP_FIBOLEVELS property) and the values of levels (OBJPROP_FIRSTLEVEL+n property) use ObjectSet() function.
OBJ_EXPANSION	14	Fibonacci expansions. Uses 3 coordinates. To set the number of levels (OBJPROP_FIBOLEVELS property) and the values of levels (OBJPROP_FIRSTLEVEL+n property) use ObjectSet() function.
OBJ_FIBOCHANNEL	15	Fibonacci channel. Uses 3 coordinates. To set the number of levels (OBJPROP_FIBOLEVELS property) and the values of levels (OBJPROP_FIRSTLEVEL+n property) use ObjectSet() function.
OBJ_RECTANGLE	16	Rectangle. Uses 2 coordinates.
OBJ_TRIANGLE	17	Triangle. Uses 3 coordinates.
OBJ_ELLIPSE	18	Ellipse. Uses 2 coordinates. To set the ratio between the price and time scales (OBJPROP_SCALE property) use ObjectSet() function.
OBJ_PITCHFORK	19	Andrews pitchfork. Uses 3 coordinates.
OBJ_CYCLES	20	Time rounds (cyclic lines). Uses 2 coordinates.

OBJ_TEXT	21	Text. Uses 1 coordinate. To set the angle of the writing text (OBJPROP_ANGLE parameter) use ObjectSet() fucntion. To modify the text use ObjectSetText() fucntion.
OBJ_ARROW	22	Arrows (symbols). Uses 1 coordinate. To set the symbol code (OBJPROP_ARROWCODE parameter) use ObjectSet() function.
OBJ_LABEL	23	Text label. To set the coordinates in pixels relatively to the angle of connection (OBJPROP_CORNER, OBJPROP_XDISTANCE, OBJPROP_YDISTANCE parameters) use ObjectSet() function. To modify the text use ObjectSetText() fucntion.

Graphical object properties identifiers used with ObjectGet() и ObjectSet(), and can be any of the following values:

Object Parameters	Value	Type	Description
OBJPROP_TIME1	0	datetime	Value to set/get first coordinate time part.
OBJPROP_PRICE1	1	double	Value to set/get first coordinate price part.
OBJPROP_TIME2	2	datetime	Value to set/get second coordinate time part.
OBJPROP_PRICE2	3	double	Value to set/get second coordinate price part.
OBJPROP_TIME3	4	datetime	Value to set/get third coordinate time part.
OBJPROP_PRICE3	5	double	Value to set/get third coordinate price part.
OBJPROP_COLOR	6	color	Value to set/get object color.
OBJPROP_STYLE	7	int	Value to set/get line style.
OBJPROP_WIDTH	8	int	Value to set/get object line width.
OBJPROP_BACK	9	bool	Value to set/get background drawing flag for object.
OBJPROP_RAY	10	bool	Value to set/get ray flag of objects like OBJ_TREND and alikes.
OBJPROP_ELLIPSE	11	bool	Value to set/get ellipse flag for OBJ_FIBOARC object.
OBJPROP_SCALE	12	double	Value to set/get scale object property.
OBJPROP_ANGLE	13	double	Value to set/get angle object property in degrees for OBJ_TRENDBYANGLE obect.
OBJPROP_ARROWCODE	14	int	Value or arrow enumeration to set/get arrow code property for OBJ_ARROW obect. It can be on of the wondings symbols or one of the predefined arrow codes.
OBJPROP_TIMEFRAMES	15	int	Value to set/get timeframe object property. Can be one or combination of number of object visibility constants.
OBJPROP_DEVIATION	16	double	Value to set/get deviation property for OBJ_STDDEVCHANNEL object.
OBJPROP_FONTSIZE	100	int	Value to set/get font size for OBJ_TEXT and OBJ_LABEL objects.
OBJPROP_CORNER	101	int	Value to set/get anchor corner property for OBJ_LABEL object. Must be from 0-3.
OBJPROP_XDISTANCE	102	int	Value to set/get anchor X distance object property in pixels relatively to the angle of connection for OBJ_LABEL object.
OBJPROP_YDISTANCE	103	int	Value to set/get anchor Y distance object property in pixels relatively to the angle of connection for OBJ_LABEL object.

OBJPROP_FIBOLEVELS	200	int	Value to set/get Fibonacci object level count. Can be from 1 to 32.
OBJPROP_LEVELCOLOR	201	color	Value to set/get object level line color.
OBJPROP_LEVELSTYLE	202	int	Vlaue to set/get object level line style.
OBJPROP_LEVELWIDTH	203	int	Value to set/get object level line width.
OBJPROP_FIRSTLEVEL+n	210+n	int	Value to set/get the number of the level of the object, where n is level index to set/get. Can be from 0 to 31.

← Styles of Indicator Lines Displaying Wave Files →

Sound Files

The set of the wav files recommended for using in the practical work in EAs, indicators and scripts:

File Name	Recommended Terms of Use
Close_order.wav	Market order closing
Ok.wav	Successful ending of a trade operation
Transform.wav	Transforming a pending order into a market order
Bulk.wav	Insignificant events
Inform.wav	Different information (trade criterion modification, etc.)
Oops.wav	Critical information (out of money, etc.) or invalid action
Expert.wav	Trade order is given
Error.wav	Unsuccessful ending of a trade operation
Bzrrr.wav	Other errors
Wait.wav	Pause
Work.wav	Necessary programs and files are installed
Tick.wav	New tick
W1.wav	Other terms
W2.wav	Other terms
W3.wav	Other terms

It is not recommended to use jar, contrasting and long playing sounds.

← Types and Properties of Graphical Objects Function MessageBox() Return Codes →

MessageBox() Return Codes

If a message box has a **Cancel** button, the function returns the IDCANCEL value if either the ESC key is pressed or the **Cancel** button is selected. If a message box has no **Cancel** button, pressing ESC has no effect.

Constant	Value	Description
IDOK	1	OK button was selected.
IDCANCEL	2	Cancel button was selected
IDABORT	3	Abort button was selected
IDRETRY	4	Retry button was selected
IDIGNORE	5	Ignore button was selected
IDYES	6	Yes button was selected
IDNO	7	No button was selected
IDTRYAGAIN	10	Try Again button was selected
IDCONTINUE	11	Continue button was selected

These return codes are defined in the WinUser32.mqh file, so this header file must be included in programs by #include <WinUser32.mqh>.

The MessageBox() function flags specify the contents and behavior of a dialog box. This value can be a combination of flags from the following groups of flags:

Constant	Value	Description
MB_OK	0x00000000	The message box contains one push button: OK. This is the default.
MB_OKCANCEL	0x00000001	The message box contains two push buttons: OK and Cancel.
MB_ABORTRETRYIGNORE	0x00000002	The message box contains three push buttons: Abort, Retry, and Ignore.
MB_YESNOCANCEL	0x00000003	The message box contains three push buttons: Yes, No, and Cancel.
MB_YESNO	0x00000004	The message box contains two push buttons: Yes and No.
MB_RETRYCANCEL	0x00000005	The message box contains two push buttons: Retry and Cancel.
MB_CANCELTRYCONTINUE	0x00000006	The message box contains three push buttons: Cancel, Try Again, Continue.

To display an icon in a message box define the additional flags:

Constant	Value	Description
MB_ICONSTOP MB_ICONERROR MB_ICONHAND	0x00000010	A stop-sign icon appears in the message box.
MB_ICONQUESTION	0x00000020	A question-mark icon appears in the message box.
MB_ICONEXCLAMATION MB_ICONWARNING	0x00000030	An exclamation-point icon appears in the message box.
MB_ICONINFORMATION MB_ICONASTERISK	0x00000040	An icon consisting of a lowercase letter i in a circle appears in the message box.

Default buttons are specified with the following flags:

Constant	Value	Description
MB_DEFBUTTON1	0x00000000	The first burron MB_DEFBUTTON1 is the default button, unless MB_DEFBUTTON2, MB_DEFBUTTON3, or MB_DEFBUTTON4 is specified.
MB_DEFBUTTON2	0x00000100	The second button is the default button.
MB_DEFBUTTON3	0x00000200	The third button is the default button.
MB_DEFBUTTON4	0x00000300	The fourth button is the default button.

MessageBox() function behavior flags are defined in the WinUser32.mqh file, this is why this heading file must be included to programs through #include <WinUser32.mqh>. Not all possible flags are listed here. For more details, please refer to Win32 API description.

← Wave Files Identifiers of Function MarketInfo() →

MarketInfo() Identifiers

Query identifiers used in the MarketInfo() function can have the following values:

Constant	Value	Description
MODE_LOW	1	Minimum day price
MODE_HIGH	2	Maximum day price
MODE_TIME	5	The last incoming tick time.
MODE_BID	9	Last incoming bid price. For the current symbol, it is stored in the predefined variable Bid.
MODE_ASK	10	Last incoming ask price. For the current symbol, it is stored in the predefined variable Ask.
MODE_POINT	11	Point size in the quote currency. For the current symbol, it is stored in the predefined variable Point.
MODE_DIGITS	12	Count of digits after decimal point in the symbol prices. For the current symbol, it is stored in the predefined variable Digits.
MODE_SPREAD	13	Spread value in points.
MODE_STOPLEVEL	14	Minimal permissible StopLoss/TakeProfit value in points.
MODE_LOTSIZE	15	Lot size in the base currency.
MODE_TICKVALUE	16	Minimal tick value in the deposit currency.
MODE_TICKSIZE	17	Minimal tick size in the quote currency.
MODE_SWAPLONG	18	Swap of a long position.
MODE_SWAPSHORT	19	Swap of a short position.
MODE_STARTING	20	Trade starting date (usually used for futures).
MODE_EXPIRATION	21	Trade expiration date (usually used for futures).
MODE_TRADEALLOWED	22	Trade is allowed for the symbol.
MODE_MINLOT	23	Minimal permitted lot size.
MODE_LOTSTEP	24	Step for changing lots.
MODE_MAXLOT	25	Maximal permitted lot size.
MODE_SWAPTYPE	26	Swap calculation method. 0 - in points; 1 - in the symbol base currency; 2 - by interest; 3 - in the margin currency.
MODE_PROFITCALCMODE	27	Profit calculation mode. 0 - Forex; 1 - CFD; 2 - Futures.
MODE_MARGINCALCMODE	28	Margin calculation mode. 0 - Forex; 1 - CFD; 2 - Futures; 3 - CFD for indexes.
MODE_MARGININIT	29	Initial margin requirements for 1 lot.
MODE_MARGINMAINTENANCE	30	Margin to maintain open positions calculated for 1 lot.
MODE_MARGINHEDGED	31	Hedged margin calculated for 1 lot.
MODE_MARGINREQUIRED	32	Free margin required to open 1 lot for buying.
MODE_FREEZELEVEL	33	Order freeze level in points. If the execution price lies within the range defined by the freeze level, the order cannot be modified, canceled or closed.

← Function MessageBox() Return Codes List of Programs →

List of Programs

Expert Advisors:

Name	Description with the link to the section
create.mq4	Example of a commented programming code
simple.mq4	Example of a simple EA, execution of special functions
possible.mq4	Example of a correct program structure
incorrect.mq4	Example of an incorrect program structure
userfunction.mq4	Simple example of a user-defined function applicaton
onelevel.mq4	Example of if-else operator application
twolevel.mq4	Example of if-else operator application
twoleveloptim.mq4	Example of if-else operator application
compoundcondition.mq4	Example of if-else operator application
pricealert.mq4	Example of switch operator application
predefined.mq4	Updating a value of a predefined variable
countticks.mq4	Example of global variable application (tick counter)
staticvar.mq4	Example of static variable application (tick counter)
externvar.mq4	Example of external programs application
globalvar.mq4	Example of GlobalVariable application
stringarray.mq4	Example of string array application
extremumprice.mq4	Usage of values of time-series array elements
newbar.mq4	Detecting the fact of a new bar appearance (time-series array)
modifystoploss.mq4	StopLoss market orders modification
callindicator.mq4	Calling a technical indicator function from an EA
historybars.mq4	Calling iMA() technical indicator function from an EA
callstohastic.mq4	Calling iStochastic() technical indicator function from an EA
tradingexpert.mq4	Simple EA. Structure, strategy, algorithm.
shared.mq4	EA that calculates trading criteria on the basis of a custom indicator.
comment.mq4	Displaying a text in the upper left corner of a security window
dialogue.mq4	EA supporting a dialog with a user
grafobjects.mq4	EA that uses OBJ_LABEL graphical object
moveobjects.mq4	EA that manages the position of a graphical objects
charts.mq4	EA that manages graphical objects in subwindows of a security window
strings.mq4	EA that manages graphical objects for candlestick coloring
timebars.mq4	EA that displays a tick coming time and bar opening time
bigbars.mq4	EA for searching a bar that is not lower than a specified height
timeevents.mq4	EA that performs some action at a specified time
createfile.mq4	EA for creating news timetable file
matrix.mq4	EA for matrix transpose
check.mq4	Rights limitation when using programs that are distributed on a commercial basis

usualexpert.mq4	Usual EA that uses include files

Scripts:

Name	Description with the link to the section
pifagor.mq4	Example of a program without any user-defined function
gipo.mq4	Example of user-defined function application
fibonacci.mq4	Example of while cycle operator application
sumtotal.mq4	Example of for cycle operator application
rectangle.mq4	Example of break operator application
area.mq4	Example of break operator application
sheep.mq4	Example of continue operator application
othersheep.mq4	Example of continue operator application
barnumber.mq4	Example of switch operator application
callfunction.mq4	Example of user-defined function application
countiter.mq4	Intertick cycles counter
arrayalert.mq4	Example of arrays initialization
simpleopen.mq4	Simple script for opening order
confined.mq4	Script with simple error analysis
improved.mq4	Script can work in any security window
mistaken.mq4	Script with the incorrect open price
conditions.mq4	Script for defining an order price
openbuy.mq4	Script for opening the market order Buy
openbuystop.mq4	Script for opening BuyStop pending order
closeorder.mq4	Script for closing one of the market orders
deleteorder.mq4	Script for deleting one of the pending orders
closeby.mq4	Script for closing of opposite orders
modifyorderprice.mq4	Script for modifying pending order
timetablenews.mq4	Script for reading data from a file and displaying graphical objects
deleteall.mq4	Script that deletes all global variables of the client terminal

Indicators:

Name	Description with the link to the section
userindicator.mq4	Simple custom indicator (High and Low lines)
averagevalue.mq4	Simple custom indicator (High and Low averaging)
separatewindow.mq4	Custom indicator in a separate window
displacement.mq4	Vertical and horizontal shifting of custom indicator lines
roc.mq4	Custom indicator ROC
rocseparate.mq4	Custom indicator ROC in a separate window
indicatorstyle.mq4	Indicator showing the High line
linelevel.mq4	Indicator showing the difference between high and low
Inform.mq4	Indicator that creates an empty subwindow in a security window

Include Files:

Name	Description with the link to the section
Variables.mqh	Include file that contains global variables declaration
Check.mqh	Function that limits usage rights
Terminal.mqh	Order accounting function
Inform.mqh	Function for displaying a message in a subwindow created by the Inform.mq4 indicator
Events.mqh	Events tracing function
Lot.mqh	Function for defining the amount of lots
Criterion.mqh	Function for trade criterion defining
Trade.mqh	Controlling trade function
Close_All.mqh	Function for closing all market orders of a specified type
Open_Ord.mqh	Function for opening one market oder of a specified type
Tral_Stop.mqh	Function for modifying all market orders of a specified type
Errors.mqh	Function for error processing

⇐ Identifiers of Function MarketInfo()