MetaTrader 4 Expert Advisors

MetaQuotes Language 4
Articles

In this section you can find articles dealing with various aspects of creating and using expert advisors and custom indicators. Before starting to write your own programs in MQL4, read these articles carefully. You are sure to find answers to your questions here.

Along with expert advisors and custom indicators, it is also possible to create and use scripts and libraries. The main difference between a script and an expert advisor is the way they are used: Expert Advisors are executed for each tick, while Scripts are executed only once, on request. For example, with a script, you can implement various sequences of actions for the client terminal, displaying additional messages or waiting for some actions from the user.

Libraries are used to create archives with custom functions. It means that a software developer can collect her/his original functions in library files and call them from any custom program. As soon as a library is created all functions described in it are compiled and saved for the future use. It is worth mentioning that functions being not called are removed from the code of expert advisors, custom indicators, and scripts during the compilation process in order to optimize it. This is the peculiarity of creating libraries in MQL 4 as compared to other programs.

Features of Experts Advisors creation

Creation of expert advisors in the MetaTrader trading system has a number of features.

- Before opening a position, you should check if there is money available on the account. If there is not enough money on the account, the operation of opening a position will not be successful. The "FreeMargin" value must not be less than 1000 only during tests because the price of one lot is 1000 during tests.

```
if(AccountFreeMargin() < 1000) return(0);   // not enough money
```

- You can access history data by using the predefined arrays Time, Open, Low, High, Close, Volume. Due to historical reasons, index in these arrays increases from the end to the beginning. It means that the latest data have index 0. Index 1 indicates data shifted one period backwards, index 2 means data shifted two periods backwards, 3 is three periods backwards, etc.

```
// if Close on the previous bar is less than
// Close on the bar before previous
if(Close[1] < Close[2]) return(0);
```

- It is also possible to access historical data using other time intervals and even using other currency pairs. To get such data, it is necessary to define a one-dimensional array first and perform a copy operation using the "ArrayCopySeries" function. Mind that, while calling the function, it is possible to pass a less number of parameters and do not specify default parameters.

```
double eur_close_m1[];
int number_copied = ArrayCopySeries(eur_close_m1, MODE_CLOSE, "EURUSD",
PERIOD_M1);
```

- In the process of writing an expert advisor, as well as any other software, it is sometimes necessary to get some additional debugging information. The MQL 4 language provides several methods for getting such information.

  - The "Alert" function displays a dialog box with some data defined by the user.

```
Alert("FreeMargin grows to ", AccountFreeMargin(), "!");
```

  - The "Comment" function displays data defined by the user in the upper-left corner of the chart. The character sequence "\n" is used to start a new line.

```
Comment("FreeMargin is ", AccountFreeMargin(), ".");
```

  - The "Print" function saves data defined by the user to the system log.

```
Print("FreeMargin is ", AccountFreeMargin(), ".");
```

- To get the information about errors in the program, the "GetLastError" function is very useful. For example, an operation with an order always returns the ticket number. If the ticket number equals 0 (some error has occurred in the process of performing the operation), it is necessary to call the "GetLastError"" function to get additional information about the error:

```
int iTickNum = 0;
int iLastError = 0;
...
iTickNum = OrderSet (OP_BUY, g_Lots, Ask, 3, 0, Ask + g_TakeProfit *
g_Points, Red);
if (iTickNum <= 0)
  {
   iLastError = GetLastError();
   if (iLastError != ERROR_SUCCESS) Alert("Some Message");
  }
```

You should remember that calling the "GetLastError" function displays the code of the last error and resets its value. That is why calling this function once again in a row will always return value 0.

- How to define the beginning of a new bar? (It can be necessary to find out that the previous bar has just been finished.) There are several methods to do it.

The first method is based on checking the number of bars:

```
int prevbars = 0;
...
if(prevbars == Bars) return(0);
prevbars = Bars;
...
```

This method can fail to work while loading the history. That is, the number of bars is changed while the "previous" one has not been finished yet. In this case you can make checking more complicated by introducing a check for difference between the values equal to one.

The next method is based on the fact that the "Volume" value is generated depending on the number of ticks that have come for each bar and the first tick means that the "Volume" value of a new bar equals 1:

```
if( Volume > 1) return(0);
...
```

This method can fail to work when there are a lot of incoming price ticks. The matter is that incoming price tricks are processed in a separate thread. And if this thread is busy when the next tick comes, this new incoming tick is not processed to avoid overloading the processor! In this case you can also make checking more complicated by saving the previous "Volume" value.

The third method is based on the time a bar is opened:

```
datetime prevtime=0;
...
if(prevtime == Time[0]) return(0);
prevtime = Time[0];
...
```

It is the most reliable method. It works in all cases.

- An example of working with a file of the "CSV" type:

```
int h1;
h1 = FileOpen("my_data.csv", MODE_CSV | MODE_WRITE, ";");
if(h1<0)
  {
   Print("Unable to open file my_data.csv");
   return(false);
  }
FileWrite(h1, High[1], Low[1], Close[1], Volume[1]);
FileClose(h1);
```

- Some explanations to the code. The file of the "CSV" format is opened first. In case there occurs an error while opening the file, the program is exited. In case the file is successfully opened, its content gets cleared, data are saved to the file and the file is closed. If you need to keep the content of the file being opened, you should use the MODE_READ opening mode:

```
int h1;
h1 = FileOpen("my_data.csv", MODE_CSV | MODE_WRITE | MODE_READ, ";");
if(h1<0)
```

```
   {
     Print("Unable to open file my_data.csv");
     return(false);
    }
  FileSeek(h1, 0, SEEK_END);
  FileWrite(h1, High[1], Low[1], Close[1], Volume[1]);
  FileClose(h1);
```

- In this example, data are added to the end of the file. To do it, we used the "FileSeek" function right after it was opened.

Features of Custom Indicators creation

Creation of Custom Indicators in the MetaTrader trading system has a number of features.

- For the program to be considered a Custom Indicator, it must fall under one of two definitions:

```
#property  indicator_chart_window      // an indicator is drawn in the main chart window
```

or

```
#property  indicator_separate_window   // an indicator is drawn in a separate window
```

- To set the scale of a separate indicator window, the following definitions are used:

```
#property  indicator_minimum Min_Value
#property  indicator_maximum Max_Value
```

where "Min_Value" and "Max_Value" are corresponding values. For example, these values must be 0 and 100 respectively for custom indicator RSI.

- The number of indicator arrays necessary for drawing an indicator must be defined as follows:

```
#property  indicator_buffers N
```

where N can take on values from 1 to 8.

- The color of lines in an indicator is set by the following definitions:

```
#property  indicator_color1  Silver

#property  indicator_color2  Red

...

#property  indicator_colorN  <SomeColor>
```

where N is the number of indicator arrays defined by "#define indicator_buffer".

- There are functions allowing to control the process of indicator calculation and visualization. Custom Indicator of Ishimoku Kinko Hyo is used here to illustrate it:

```
//+------------------------------------------------------------------+
//|                                                   Ichimoku.mq4 |
//|                      Copyright © 2004, MetaQuotes Software Corp. |
//|                                       http://www.metaquotes.net/ |
//+------------------------------------------------------------------+
#property copyright "Copyright © 2004, MetaQuotes Software Corp."
#property link      http://www.metaquotes.net/

#property indicator_chart_window
#property indicator_buffers 7
#property indicator_color1 Red
#property indicator_color2 Blue
#property indicator_color3 SandyBrown
#property indicator_color4 Thistle
#property indicator_color5 Lime
#property indicator_color6 SandyBrown
#property indicator_color7 Thistle
//---- input parameters
extern int Tenkan=9;
extern int Kijun=26;
extern int Senkou=52;
//---- indicator buffers
```

```
double Tenkan_Buffer[];

double Kijun_Buffer[];

double SpanA_Buffer[];

double SpanB_Buffer[];

double Chinkou_Buffer[];

double SpanA2_Buffer[];

double SpanB2_Buffer[];

//---- span_a drawing begin

int a_begin;

//+------------------------------------------------------------------+

//| Custom indicator initialization function                         |

//+------------------------------------------------------------------+

int init()

  {

//----

   SetIndexStyle(0,DRAW_LINE);

   SetIndexBuffer(0,Tenkan_Buffer);

   SetIndexDrawBegin(0,Tenkan-1);

   SetIndexLabel(0,"Tenkan Sen");

//----

   SetIndexStyle(1,DRAW_LINE);

   SetIndexBuffer(1,Kijun_Buffer);

   SetIndexDrawBegin(1,Kijun-1);

   SetIndexLabel(1,"Kijun Sen");
```

```
//----
   a_begin=Kijun; if(a_begin<Tenkan) a_begin=Tenkan;
   SetIndexStyle(2,DRAW_HISTOGRAM,STYLE_DOT);
   SetIndexBuffer(2,SpanA_Buffer);
   SetIndexDrawBegin(2,Kijun+a_begin-1);
   SetIndexShift(2,Kijun);
   SetIndexLabel(2,NULL);
   SetIndexStyle(5,DRAW_LINE,STYLE_DOT);
   SetIndexBuffer(5,SpanA2_Buffer);
   SetIndexDrawBegin(5,Kijun+a_begin-1);
   SetIndexShift(5,Kijun);
   SetIndexLabel(5,"Senkou Span A");
//----
   SetIndexStyle(3,DRAW_HISTOGRAM,STYLE_DOT);
   SetIndexBuffer(3,SpanB_Buffer);
   SetIndexDrawBegin(3,Kijun+Senkou-1);
   SetIndexShift(3,Kijun);
   SetIndexLabel(3,NULL);
   SetIndexStyle(6,DRAW_LINE,STYLE_DOT);
   SetIndexBuffer(6,SpanB2_Buffer);
   SetIndexDrawBegin(6,Kijun+Senkou-1);
   SetIndexShift(6,Kijun);
   SetIndexLabel(6,"Senkou Span B");
//----
```

```
   SetIndexStyle(4,DRAW_LINE);

   SetIndexBuffer(4,Chinkou_Buffer);

   SetIndexShift(4,-Kijun);

   SetIndexLabel(4,"Chinkou Span");

//----

   return(0);

  }

//+------------------------------------------------------------------+

//| Ichimoku Kinko Hyo                                               |

//+------------------------------------------------------------------+

int start()

  {

   int    i,k;

   int    counted_bars=IndicatorCounted();

   double high,low,price;

//----

   if(Bars<=Tenkan || Bars<=Kijun || Bars<=Senkou) return(0);

//---- initial zero

   if(counted_bars<1)

     {

      for(i=1;i<=Tenkan;i++)    Tenkan_Buffer[Bars-i]=0;

      for(i=1;i<=Kijun;i++)     Kijun_Buffer[Bars-i]=0;

      for(i=1;i<=a_begin;i++) { SpanA_Buffer[Bars-i]=0; SpanA2_Buffer
[Bars-i]=0; }

      for(i=1;i<=Senkou;i++)  { SpanB_Buffer[Bars-i]=0; SpanB2_Buffer
[Bars-i]=0; }

     }

//---- Tenkan Sen

   i=Bars-Tenkan;

   if(counted_bars>Tenkan) i=Bars-counted_bars-1;

   while(i>=0)

     {

      high=High[i]; low=Low[i]; k=i-1+Tenkan;

      while(k>=i)

        {

         price=High[k];

         if(high<price) high=price;

         price=Low[k];

         if(low>price)  low=price;

         k--;

        }

      Tenkan_Buffer[i]=(high+low)/2;

      i--;

     }

//---- Kijun Sen

   i=Bars-Kijun;

   if(counted_bars>Kijun) i=Bars-counted_bars-1;

   while(i>=0)

     {
```

```
      high=High[i]; low=Low[i]; k=i-1+Kijun;
      while(k>=i)
        {
         price=High[k];
         if(highprice)  low=price;
         k--;
        }
      Kijun_Buffer[i]=(high+low)/2;
      i--;
     }
//---- Senkou Span A
   i=Bars-a_begin+1;
   if(counted_bars>a_begin-1) i=Bars-counted_bars-1;
   while(i>=0)
     {
      price=(Kijun_Buffer[i]+Tenkan_Buffer[i])/2;
      SpanA_Buffer[i]=price;
      SpanA2_Buffer[i]=price;
      i--;
     }
//---- Senkou Span B
   i=Bars-Senkou;
   if(counted_bars>Senkou) i=Bars-counted_bars-1;
   while(i>=0)
```

```
     {
      high=High[i]; low=Low[i]; k=i-1+Senkou;
      while(k>=i)
        {
         price=High[k];
         if(high<price) high=price;
         price=Low[k];
         if(low>price)  low=price;
         k--;
        }
      price=(high+low)/2;
      SpanB_Buffer[i]=price;
      SpanB2_Buffer[i]=price;
      i--;
     }
//---- Chinkou Span
   i=Bars-1;
   if(counted_bars>1) i=Bars-counted_bars-1;
   while(i>=0) { Chinkou_Buffer[i]=Close[i]; i--; }
//----
   return(0);
  }
//+------------------------------------------------------------------+
```

- "SetIndexStyle" function controls drawing parameters of an indicator array. The drawing mode of DRAW_LINE assumes that lines between values defined in a corresponding indicator array are drawn. The drawing mode of DRAW_HISTOGRAM having been applied to the main window indicator has its special features, as well. A histogram is drawn between corresponding values of two index arrays: an even one (here: SpanA_Buffer) and an odd one (here: SpanB_Buffer). At that, the color of the index array is used the value of which is higher.
- "SetIndexDrawBegin" function specifies which element significant data of indicator array start at.
- "SetIndexBuffer" function allows to declare any one-dimensional array of the "double" type as an index array. At that, the system will manage index arrays. It is this reason for which the of these arrays does not need to be specified.

```
//---- indicator buffers

double Tenkan_Buffer[];

double Kijun_Buffer[];

double SpanA_Buffer[];

double SpanB_Buffer[];

double Chinkou_Buffer[];

double SpanA2_Buffer[];

double SpanB2_Buffer[];
```

The ArrayResize function cannot be applied to indicator arrays, either, as it is useless. It is useless, as well, to apply the ArrayInitialize function to indicator arrays, especially as 'init' function, when indicator arrays have not been allocated yet. Indicator arrays are initialized automatically during memory allocation and reallocation. EMPTY_VALUE, or the value specified by SetIndexEmptyValue function, are used as initializing values. "Empty" values are not displayed.

- The "SetIndexLabel" function sets the name to be displayed in tool tips and in data window along with the corresponding value (the "ValueN" is set by default, where N is the number of the index array). If NULL is transferred instead of a name, the corresponding value will be displayed neither in tool tips, nor in the data window. In the given case, clouds are hatched using a histogram and limited by a line. At that, the values of corresponding "line" and "histogram" arrays are the same, and it is possible to show only one of them.
- The "IndicatorCounted" function allows to organize an economic calculation of an indicator. This function returns the amount of bars by the moment of the preceding launch of the indicator, i.e., the amount of bars having already been calculated (potentially, if there were no errors or early termination during the preceding launch) which do not need any recalculation. At reinitialization of the custom indicator or significant updating of history data, this amount will be automatically reset to 0.
- Let us discuss one more example. The custom indicator named Accelerator/Decelerator Oscillator:

```
//+------------------------------------------------------------------+
//|                                              Accelerator.mq4 |
//|                      Copyright © 2005, MetaQuotes Software Corp. |
//|                                       http://www.metaquotes.net/ |
//+------------------------------------------------------------------+
#property  copyright "Copyright © 2005, MetaQuotes Software Corp."
#property  link      http://www.metaquotes.net/
//---- indicator settings
#property  indicator_separate_window
#property  indicator_buffers 3
#property  indicator_color1  Black
#property  indicator_color2  Green
#property  indicator_color3  Red
//---- indicator buffers
double     ExtBuffer0[];
double     ExtBuffer1[];
double     ExtBuffer2[];
double     ExtBuffer3[];
double     ExtBuffer4[];
//+------------------------------------------------------------------+
//| Custom indicator initialization function                         |
//+------------------------------------------------------------------+
int init()
```

```
   {
//---- 2 additional buffers are used for counting.
    IndicatorBuffers(5);
//---- drawing settings
    SetIndexStyle(0,DRAW_NONE);
    SetIndexStyle(1,DRAW_HISTOGRAM);
    SetIndexStyle(2,DRAW_HISTOGRAM);
    IndicatorDigits(Digits+2);
    SetIndexDrawBegin(0,38);
    SetIndexDrawBegin(1,38);
    SetIndexDrawBegin(2,38);
//---- 4 indicator buffers mapping
    SetIndexBuffer(0,ExtBuffer0);
    SetIndexBuffer(1,ExtBuffer1);
    SetIndexBuffer(2,ExtBuffer2);
    SetIndexBuffer(3,ExtBuffer3);
    SetIndexBuffer(4,ExtBuffer4);
//---- name for DataWindow and indicator subwindow label
    IndicatorShortName("AC");
    SetIndexLabel(1,NULL);
    SetIndexLabel(2,NULL);
//---- initialization done
    return(0);
   }
```

```
//+------------------------------------------------------------------+
//| Accelerator/Decelerator Oscillator                               |
//+------------------------------------------------------------------+
int start()
  {
   int    limit;
   int    counted_bars=IndicatorCounted();
   double prev,current;
//---- last counted bar will be recounted
   if(counted_bars>0) counted_bars--;
   limit=Bars-counted_bars;
//---- macd counted in the 1-st additional buffer
   for(int i=0; i<limit; i++)
      ExtBuffer3[i]=iMA(NULL,0,5,0,MODE_SMA,PRICE_MEDIAN,i)-
                    iMA(NULL,0,34,0,MODE_SMA,PRICE_MEDIAN,i);
//---- signal line counted in the 2-nd additional buffer
   for(i=0; i=0; i--)
     {
      current=ExtBuffer3[i]-ExtBuffer4[i];
      prev=ExtBuffer3[i+1]-ExtBuffer4[i+1];
      if(current>prev) up=true;
      if(current<prev) up=false;
      if(!up)
        {
             ExtBuffer2[i]=current;
             ExtBuffer1[i]=0.0;
            }
          else
            {
             ExtBuffer1[i]=current;
             ExtBuffer2[i]=0.0;
            }
           ExtBuffer0[i]=current;
         }
//---- done
     return(0);
    }
//+------------------------------------------------------------------+
```

- The "IndicatorBuffers" function specifies the amount of buffers to be used in indicator claculation. Generally, this function is called if more index arrays are used than it is necessary for drawing an indicator. At that, the system manages additional arrays.
- The "SetIndexDigits" function manages the precision of information output. In this case, when the difference between two moving averages as well as the further difference between the result and the signal line is calculated, the standard precision of within 4 characters after point is obviously insufficient.
- The "SetIndexDrawBegin" function specifies the element the significant data of indicator array begin at. In our example, the signal line is calculated as simple moving average of another simple moving average. This is why the first 38 values of the indicator are considered as empty ones and are not to be drawn.
- The "IndicatorShortName" function sets a so-called short name of the indicator to be displayed in the upper left corner of the indicator window and in "DataWindow". If the short name has not been set up, the custom indicator name will be used as the former one. In the given example, there is no need to use the SetIndexLabel function, because only one value is output. So, the name of the indicator is enough for output of a single value.
- The "SetIndexStyle" function manages drawing parameters of an indicator array. The drawing mode of DRAW_NONE means that the line does not need to be drawn. The matter is that histogramm of the indicator presented must be colored in 2 different colors. Data from ExtBuffer0 are allocated in two other arrays, ExtBuffer1 and ExtBuffer2. In order not to output doubling data in tool tips or in data window, the SetIndexLabel function having NULL parameter

is used. The drawing mode of DRAW_HISTOGRAM being applied to the indicator of a separate window allows to draw histogram between zero value and the value of a corresponding array (compare with drawing of a histogram in the main window described above).

- Input parameters used for calculation by custom indicators and functions must be defined as "extern" and may be of any type.
- If input parameters have not been set the corresponding custom indicator will be called in the simplest format.

```
double current_AC = iCustom( NULL, 0, "Accelerator", 0, 0 );
```

The trasnfer of the first two values "NULL" и "0" means that the current chart will be used. The name of the corresponding file (without extension of mq4) is used as custom indicator name. If the last but one parameter is 0, it means that we are interested in the data from the very first indicator array. The last parameter being 0 means that we are interested in the value of the last element (i.e., the most recent, current value) of indicator array requested.

- Parameters are transferred in the function of custom indicator calculation in the order they have been described. For example, custom indicator named "Ichimoku" and having parameters of (9,26,52) will be called as follows:

```
iCustom( NULL, 0, "Ichimoku", 9, 26, 52, 0, shift );
```

Strictly speaking, custom indicator parameters do not need to be necessarily transferred to the function. If no one extern variable is defined in the program, it is useless to trasnfer parameters. Or, if necessary, initial values can be used that are used in description of parameters. For example, the same custom indicator without parameters will be called as follows:

```
iCustom( NULL, 0, "Ichimoku", 0, shift );
```

This means that the values will be used that initialize variables "Tenkan", "Kijun", "Senkou", i.e., 9, 26, and 52. However, if one custom indicator having different sets of parameters is called in one Expert Advisor, default settings are highly not recommended to be used.

It must be noted that the surplus of custom indicators as well as incorrectly written ones can decelerate the work of client terminal significantly!

Strategy Tester: Modes of Modeling during Testing

Introduction

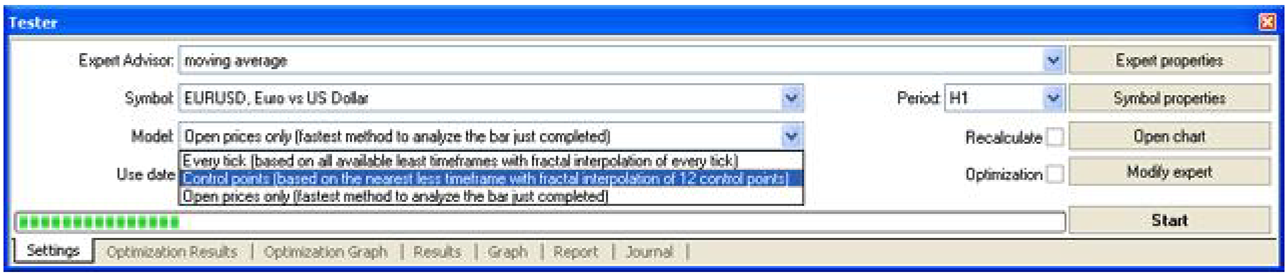
Many programs of technical analysis allow to test trading strategies on history data. In the most cases, the testing is conducted on already completed data without any attempts to model the trends within a price bar. It was made quickly, but not precisely enough.

It is important to choose a relevant way of modeling development of price bars to make a quality testing of a trading strategy. Indeed, there cannot exist an ideal situation when there is a full tick history for an extremely accurate testing. It is very difficult for a normal trader to find the full tick history for a period of several years, in order to analize it.

To solve this problem, the data of more precise periods can be used as reference points with modeling price changes between them.

Ways of Modeling Price Bars

There are three ways of modeling development of bars used in MetaTrader 4 Client Terminal:


Tester
Expert properties
Symbol properties
Open chart
Modify expert
Start
Settings | Optimization Results | Optimization Graph | Results | Graph | Report | Journal

Expert Advisor: moving average

Symbol: EURUSD, Euro vs US Dollar

Period: H1

Model: Open prices only (fastest method to analyze the bar just completed)

Every tick (based on all available least timeframes with fractal interpolation of every tick)

Control points (based on the nearest less timeframe with fractal interpolation of 12 control points)

Open prices only (fastest method to analyze the bar just completed)

Use date

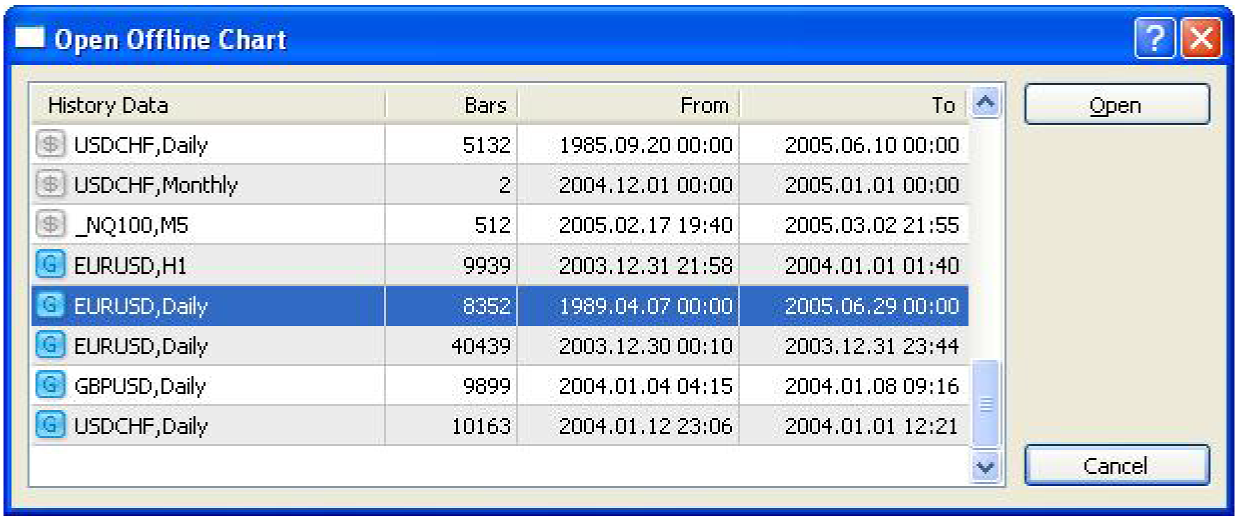
Recalculate

Optimization

- Every tick (based on all available least timeframes with fractal interpolation of every tick)
- Control points (the nearest timeframe with fractal interpolation are used)
- Open prices (quick method on completed bars)

Intermediate price bars are generated before testing starts, the results being saved in file (for example: **/tester/history/eurusd_1440_1.fxt**). Data cached in this way allow accelerate the tester hugely afterwards. After "Recalculate" has been enabled, recalculation of intermediate data can be managed.

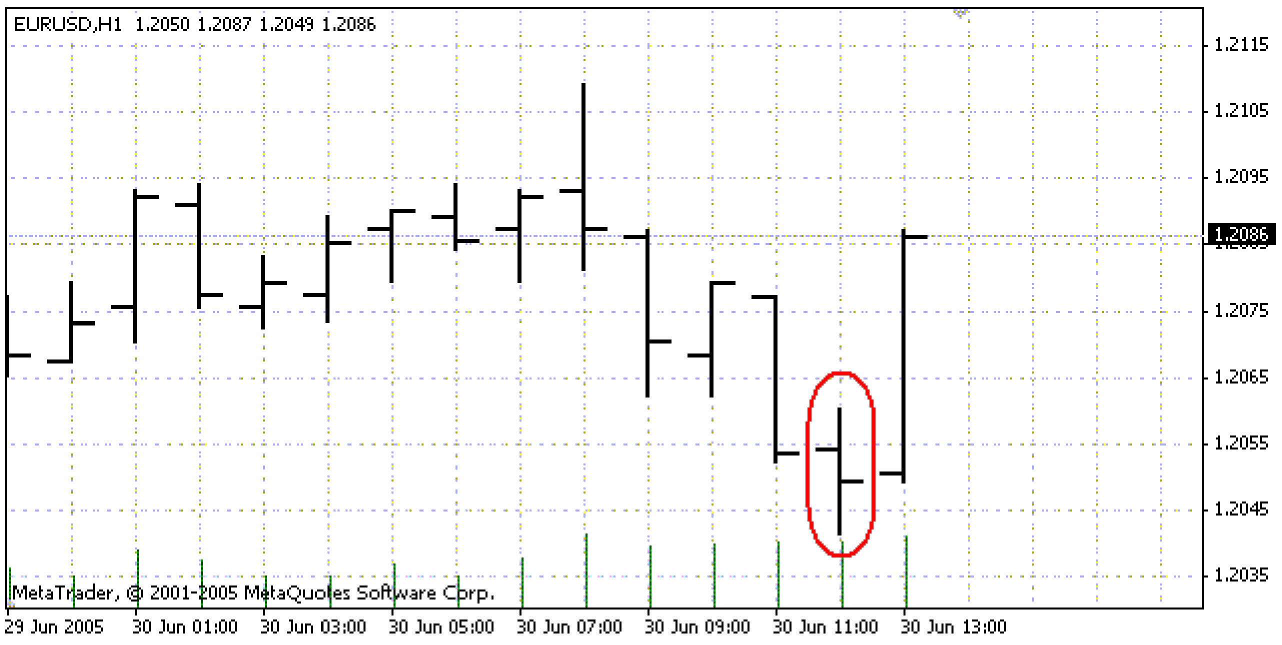
The use of data cached before allows to perform tests on the basis of the own intermediate data. To do so, it is sufficient to save file in a proper format (*.FXT format, completely open)into **/tester/history/** directory. These files are easy to open in terminal as offline charts through **File -> Open offline**.



History Data	Bars	From	To
USDCHF,Daily	5132	1985.09.20 00:00	2005.06.10 00:00
USDCHF,Monthly	2	2004.12.01 00:00	2005.01.01 00:00
_NQ100,M5	512	2005.02.17 19:40	2005.03.02 21:55
EURUSD,H1	9939	2003.12.31 21:58	2004.01.01 01:40
EURUSD,Daily	8352	1989.04.07 00:00	2005.06.29 00:00
EURUSD,Daily	40439	2003.12.30 00:10	2003.12.31 23:44
GBPUSD,Daily	9899	2004.01.04 04:15	2004.01.08 09:16
USDCHF,Daily	10163	2004.01.12 23:06	2004.01.01 12:21

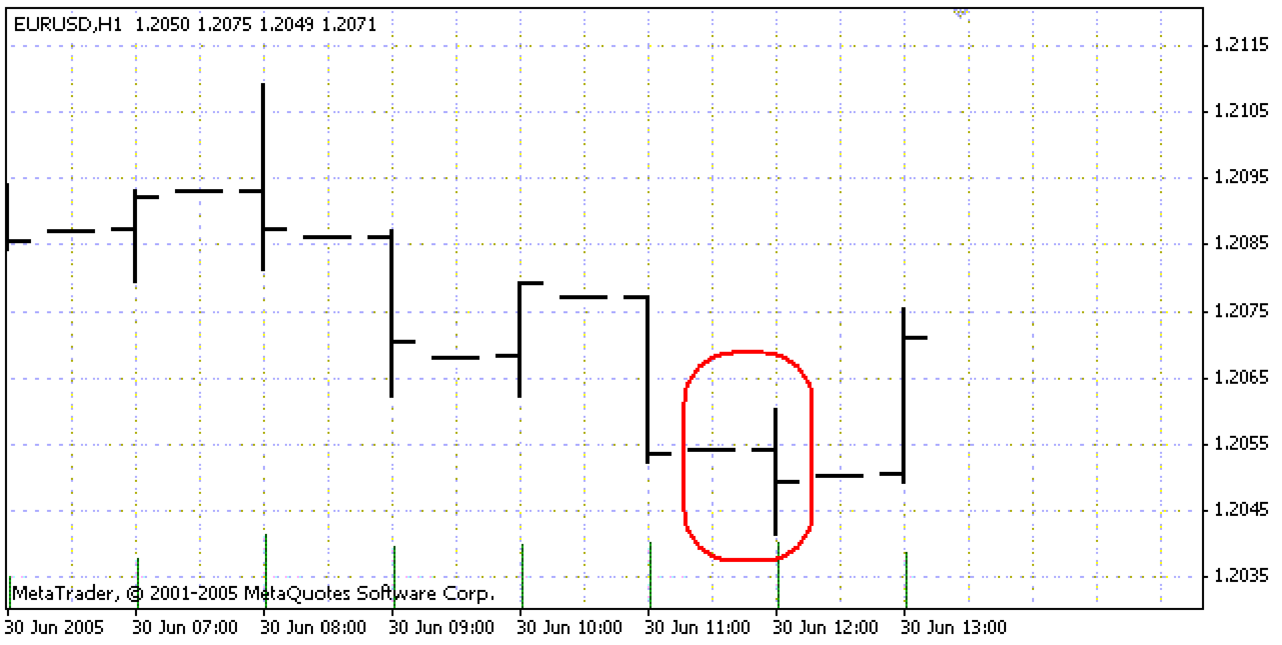
Examples of Modeling

Let us begin with the simplest modeling method based on hourly chart. Now, let us study the hour bar (June, 30 2005, 12:00) highlighted with red:



Open Price

Some traders do not wish to depend on particliuarities of intrabar modeling and create experts trading on the bars completed. The fact that the current price bar is fully completed can only be known when the next one appears. These are the experts for which the mode of "**Open Price**" modeling is intended.

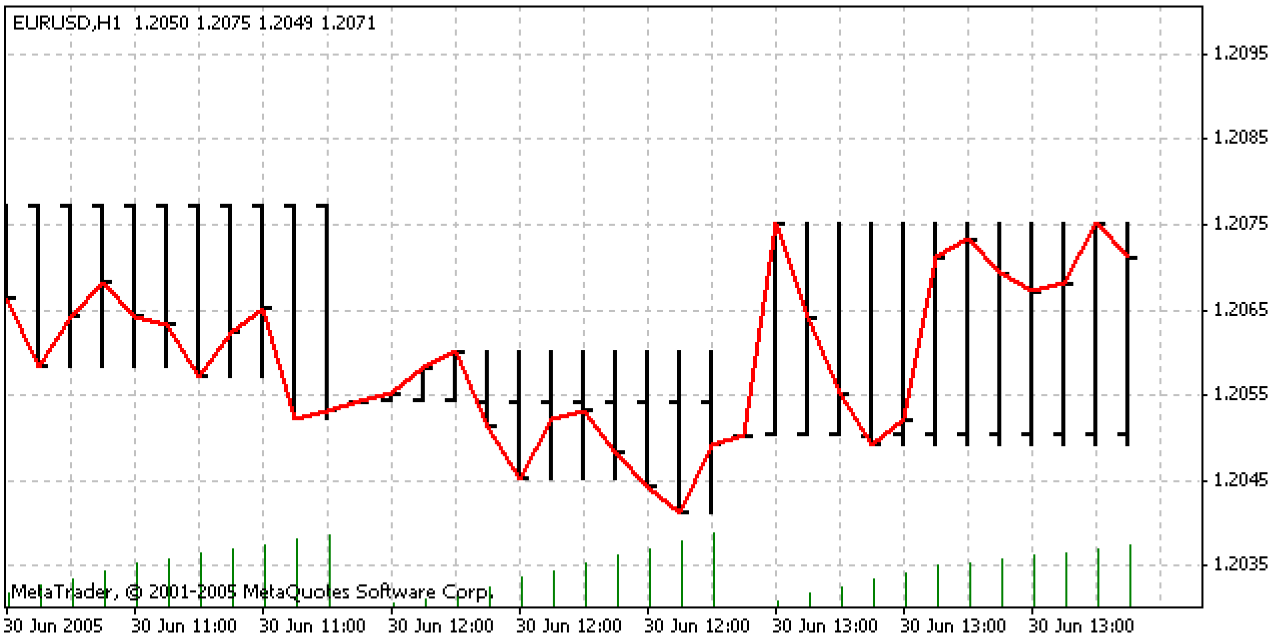


In this mode, first, bar is opened (Open = High = Low = Close, Volume=1), and this allows expert to identify the end of completion of the preceding price bar. It is this incipient bar on which expert testing is launched. At the next stage, the current bar, fulle completed, is yielded, but no testing is performed on it!

Control Points (the nearest less timeframe)

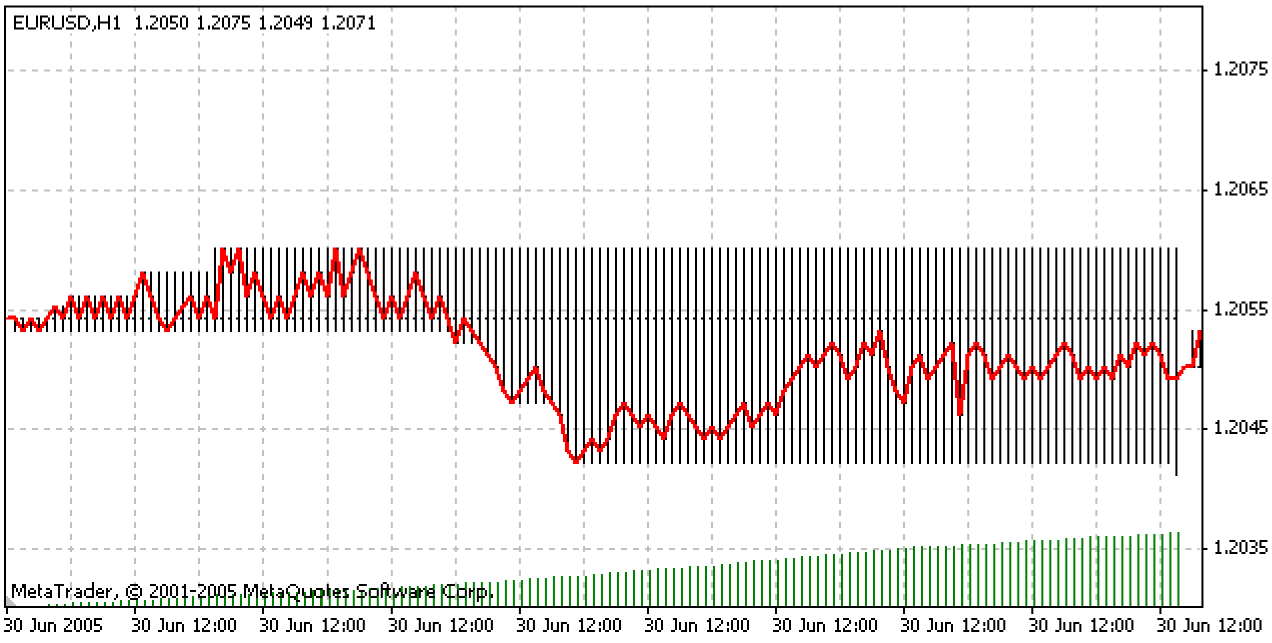
Method of modeling control points is intended for crude estimate of experts trading within a bar. To use this method, history data of the nearest less timeframe must be available. In the most cases, the data of a smaller timeframe available do not completely cover the time span of the timeframe under test. If there are no data of a smaller timeframe, the bar development is generated on basis of close prices of the preceding 12 bars. I.e., the movement within a bar repeats the price movement for the last 12 timeframes, this is that is called fractal interpolation.

This generation method is totally different from the flooding method used in the preceding vewrsions of Client Terminal, since that method allowed a strictly determined bar development. As soon as history data of a smaller timeframe appear, fractal interpolation will apply to these new data. However, not 12, but only 6 preceding bars are used. I.e., real prices of Open, High, Low, Close plus 2 more generated prices are reproduced. The value and location of these two generated prices depends on the price movement on the 6 preceding timeframes.



Every Tick (based on all available least timeframes with fractal interpolation of every tick)

This mode allows to model price movement within a bar the most precisely. Unlike "control points" the every-tick method uses for generation not only data of the nearest smaller timeframe, but also those of all available smaller timeframes. At that, if there are data of more than one timeframe for a certain time span simlutaneously, the data of the smalles timeframe are used for generation. Like the preceding method, this method uses fractal generation of control points. Fractal interpolation is used again to generate price movement between control points. It may happen that several identical ticks are generated one after another. In this case, duplicated quotes are filtered, and the volume of the last of such successive quotes is fixed.



A possible large amount of tick data generated must be considered. This may influence consumption of operating system and speed of testing.

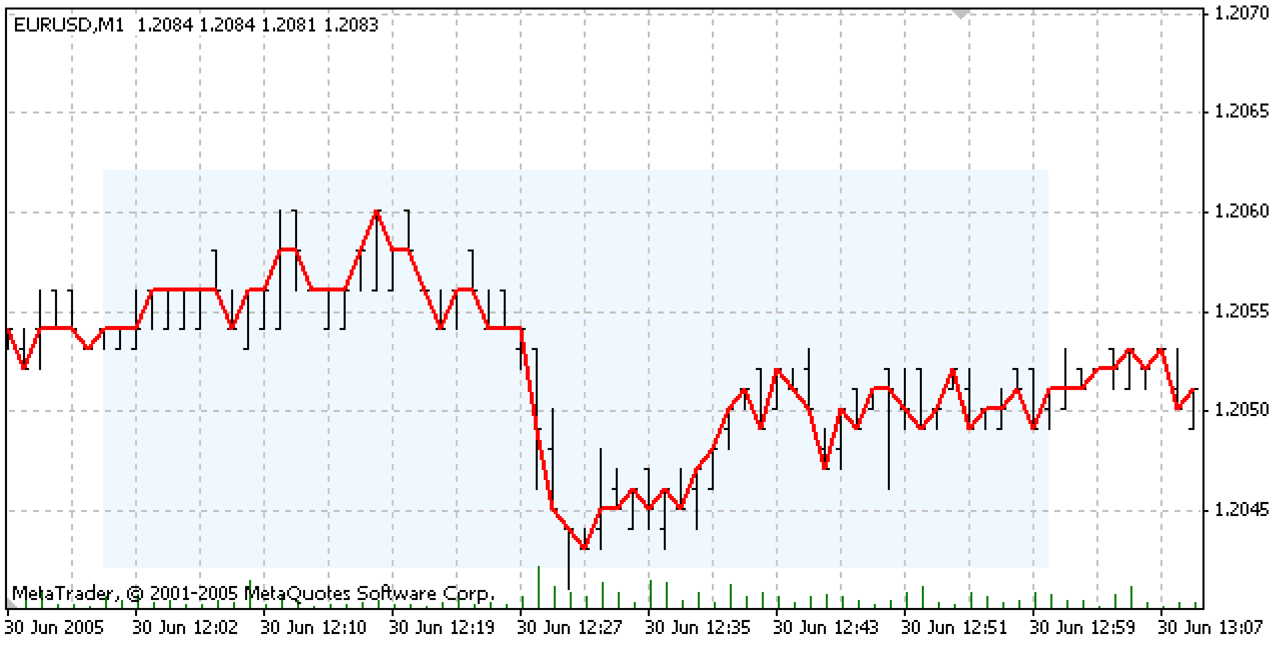
Attention: There is no need to launch the all tick testing if there are no smaller timeframes fully covering the timeframe under test. This modeling is intended only for testing on the basis of data of smaller timeframes!

Using the Range of Dates When Modeling

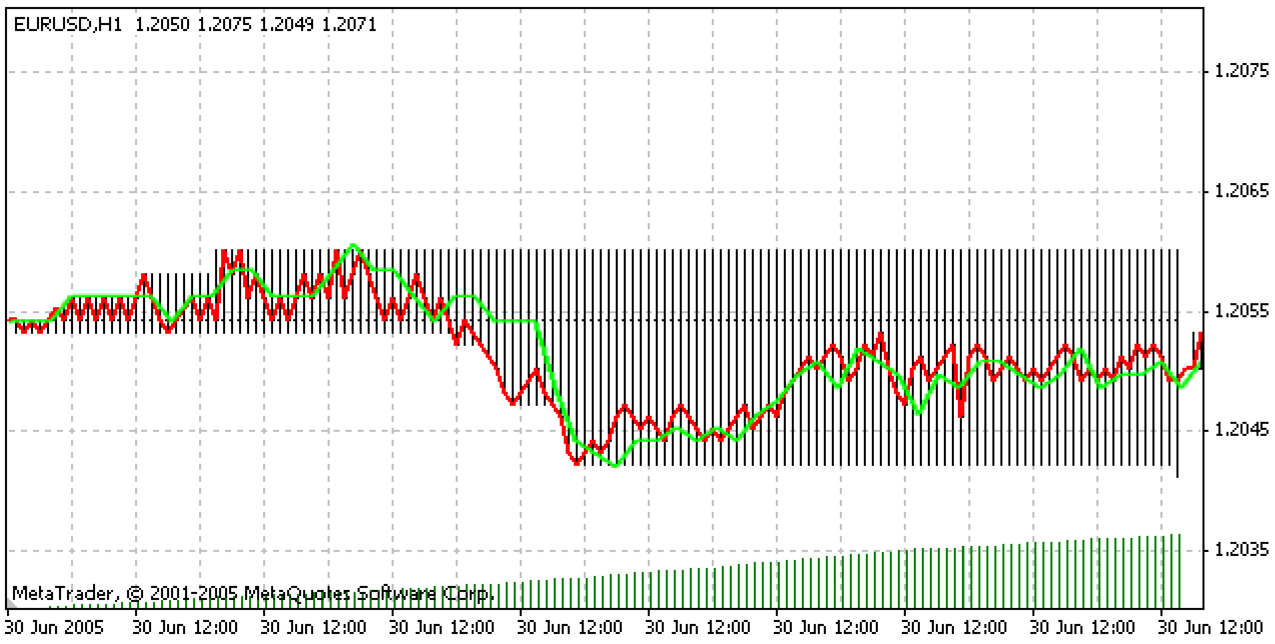
The range of dates can be set up in the Settings tab: check the "Use date" box and specify dates in "From:" and "To:" fields. The range of dates can be used not only for testing an expert advisor, but also for generating the test sequence of bars. There is often no need to generate data of the whole history, especially, for Every Tick modeling, where the amount of unused data can be very large. So, if at the initial generation of test sequence (or always, if the "Recalculate" box is checked) there was a possibility to use the range of dates, then bars, exceeding the given range were not generated, but just overwritten in the output sequence. The data are not excluded from the sequence in order the possibility to remain that allows to calculate indicators correctly for the whole history received. It must be noted that the first 100 bars are not generated either, this limitation does not depend on the range of dates being set up.

Referencing M1 Timeframe

To check the accuracy of intrabar modeling, the minutes chart of June, 30 2005, between 12:00 a.m. and 1:00 p.m. will be used.



A simple scaling of the original minutes chart (green line is the Close price) and its superimpositioning on the chart of all-tick modeled data results in a practically exact coincidence:



Conclusions

Maximally precise testing can be performed and simulation veracity can be well assured if there are auxiliary smaller timeframes 100% covering the timeframe under test. I.e., the problem of a qualitative every-tick testing turns into the search for detailed history data...

Testing Features and Limits in MetaTrader 4

Introduction

This article allows to find out more about features and limits of Strategy Tester in MetaTrader 4.

Special Features of Testing Strategies on History Data

- ***Some functions are processed/passed without output***
  These are Sleep(), Alert(), SendMail(), SpeechText(), PlaySound(), MessageBox(), WindowFind(), WindowHandle(), WindowIsVisible()

- ***Trading is permitted for the symbol under test only, no portfolio testing***
  Attempts to trade using another symbol will return error

- ***Lot sizes including initial size and increment step, commissions and swaps should be taken from the active account settings***
  Before testing, it is necessary to make sure that there is at least one activated account in the list in "Navigator" window of the terminal.

- ***All swaps, margin requirements, expirations, GTC-orders are modeled***
  Testing is performed maximally closely to trading server conditions. But the can occur some inaccuracies in estimation of margin requirements on cross currencies because of lack of precise information about conversion prices at each moment.

- ***Zero bar of another timeframe for the same symbol under test is modeled approximately***
  Open = correct Open, Close = correct Close, Low = min (Open,Close), High = max (Open,Close), Volume = final Volume (false)

- ***Instant Execution mode is assumed to be used in trades, being processed without slippage***

- ***Processing orders, Open/Close without slippages***

- ***Testing stops after StopOut***

- ***Weekly, monthly, and irregular timeframes are not tested***

- ***The deposit currency can be changed, but conversion prices are set, and the current available ones are used***

- ***There are still no delays in execution of trade operations***
  A setup delay is planned to be introduced in processing of transactions

- ***Account History is fully available, and it does not depend on settings***

- ***If other symbols and periods are actively used, it is desirable to download them to all possible depth***

- ***At every-tick modeling, tester pumps all necessary timeframes for the symbol under test independently***

- ***Using of MarketInfo function generates error ERR_FUNCTION_NOT_ALLOWED_IN_TESTING_MODE(4059)***, however, correct information about current prices for the symbol under test, about stop level dimensions, about point size, about spread size of any symbol being present in the quotes window is provided.

Special Features of Optimization Process

- ***Nothing is output in the journal (either Print() function)***
  This was done in order to accelerate the testing and save disk space. If complete logs are output the journal files will need hundreds of MByte.

- ***Draw objects are not really set***
  The objects are disabled in order to accelerate the testing.

- ***"Skip useless results" function is used***
  In order not to garble the table and chart with testing results, the possibility to skip very bad results is used. This function can be enabled in context menu of "Optimization Results" -> "Skip useless results" tab.