StereoMQL

API Reference 2.00

(V 2031)

Contents

1. API for Developers 2.0 **4**
1.1 The Estimator 4
1.2 Easy coding 5
1.3 Architecture of SEAs 5
1.4 Steps to build 6
1.5 Event-Processing 7
2. Processing logic **10**
2.1 Orientation 10
3. Accessing bars **13**
3.1 The _Bars object 13
3.1.1 Basic data 13
3.1.2 Shapes and measuring 14
3.1.3 Bar interaction and pattern 14
3.1.4 Advanced 15
3.1.5 Indications 15
3.2 Single bars 16
3.2.1 Predefined single bar objects 16
3.2.2 Data of the current bar 17
3.2.3 Basic data of single CBar objects 18
3.2.4 Member functions for shapes and measuring 19
3.1.5 Bar interaction 20
4. Signals **21**
4.1 Setup functions 21
4.2 Basic commands 23
4.1.1 Open/close positions 23
4.1.2 Blocking / Unblocking 24
4.2 Pending orders 24
4.2.1 Order settings 25
4.2.2 Order modification 26
4.3 Managing trades 26
4.3.1 Stereo Future mode 26
4.3.2 Functions for Stereo Hedge mode 27
4.3.3 Commands for manual exits 29
4.4 Commands for automated exits 31
4.5 Commands for strategic orders 32
4.7 Messages 34
4.8 Other functions 34

4.9 API variables 35
4.9.1 Common 35
4.9.2 Trading data 35
4.10 Status requests 37
4.11 Source code example 38

5. Drawing 40

6. Dialog fields 41

7. Notes for advanced developers 43
7.1 Class frame 43

8. Further classes & functions of the API 44
8.1 Compatibility MT4/MT5 44
8.2 File __MT_native.mqh 44
8.3 File __ChartExt.mqh 44
8.4 File XVars.mqh 44
8.5 File Comment.mqh 45

Copyright / Impressum 46

1. API for Developers 2.0

The API 2.0 is now called **StereoMQL** and is dedicated to all those traders and developers, who are looking for an easy way to develop automations. Any automations which are created using StereoMQL are compatible to StereoTrader for MetaTrader 4 and MetaTrader 5 without any changes.

Any automations which are created for StereoTrader are called **Stereo Expert Advisors** (SEA). Such an SEA is developed as an indicator which sends trading signals to StereoTrader.

One hand, the API itself is the interface but on the other hand it is also a very powerful class construct, which makes it very easy to evaluate bars, indications etc. and to built automated processes in a very "human" way, which is normally not available to MetaTrader.

1.1 The Estimator

StereoTrader comes with a module called "Estimator". The Estimator allows you to backtest strategies visually. Trades, including visualisation of trailing stops become visible in the chart, results are displayed on the fly without running the Strategy-Tester. This makes is very comfortable to develop and verify strategies. As soon as a parameter is changed at the panel, the Estimator visualizes the changes in the chart. The following picture shows the RangeBreakOut strategy, which is included as sample with the API.

(StereoTrader with Estimator – trades become visible in the chart)

1.2 Easy coding

A short example? The following code evaluates the current ATR (average true range) and opens a position, if the last bar exceeds the ATR twice.

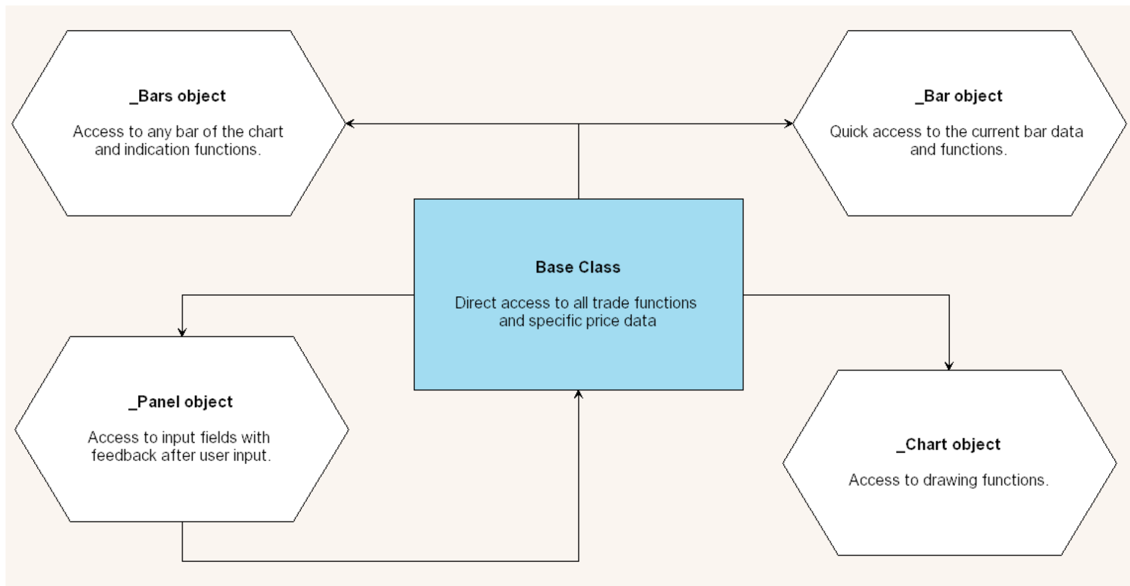
```
//+------------------------------------------------------------------+
//| Include the API and define the SEA                               |
//+------------------------------------------------------------------+
#include <StereoTrader_API\StereoAPI.mqh>
DECLARE_SEA_BEGIN("CandleATR01")

//+------------------------------------------------------------------+
//| Iteration                                                        |
//+------------------------------------------------------------------+
SEA_EVALUATE
   {
      //--- Get out if last bar is below ATR x 2.0
      if (_LastBar.Range()<_Bars.ATR(14)* 2.0)
         return;


      //--- Bullish candle, buy ...
      else if (_LastBar.IsBullish())
            Buy();

      //--- Bearish candle, sell
      if (_LastBar.IsBearish())
            Sell();

      return;
   }

//+------------------------------------------------------------------+
//| Indication                                                       |
//+------------------------------------------------------------------+
SEA_INDICATE
   {
   // Optional drawing functions
   }

//+------------------------------------------------------------------+
//| End declaration of StereoEA                                      |
//+------------------------------------------------------------------+
DECLARE_SEA_END
```

That´s it!

With native MQL, the code for the same purpose would be more than a hundred lines. But nevertheless you still have full access to all functions of MQL4/5 and in case if you are a skilled programmer, there will be no limitations at all.

1.3 Architecture of SEAs

The API of StereoTrader is designed purely object oriented. To understand this, especially for those who are not familiar with object oriented programming, the following figure should help to demonstrate the logic.



(The signal flow)

The blue box is the environment/scope, where "you" are. From here, you have direct access to any trade commands, trade functions, trade variables and so on. Therefore, if a command e. g. for buying is send, there is no prefix. Example:

```
Buy();
Flat();
...
```

The other objects represent the elements as described. Whenever you access a function of these objects, such function calls are always preceeded by the name of the object. Example:

```
double atr=_Bars.ATR(14,1);
```

And so on. This is how object oriented programming works – basically. The result is less code and better understanding of the code.

1.4 Steps to build

The API allows for very quick and very efficient development of Stereo Expert Advisors (SEAs) which are actually indicators that send trade commands to the StereoTrader, which acts as a host. All this without all the hassle of order management, because this is done entirely by StereoTrader.

To create such an SEA, use the MetaTrader Editor and create a new indicator. Then remove all the code and use the following base:

1. Include the API

```
#include <StereoTrader_API\StereoAPI.mqh>
```

2. Declare the SEA using the macro

```
DECLARE_SEA_BEGIN("My SEA")
```

3. Define the function SEA_EVALUATE and evaluate your signals here

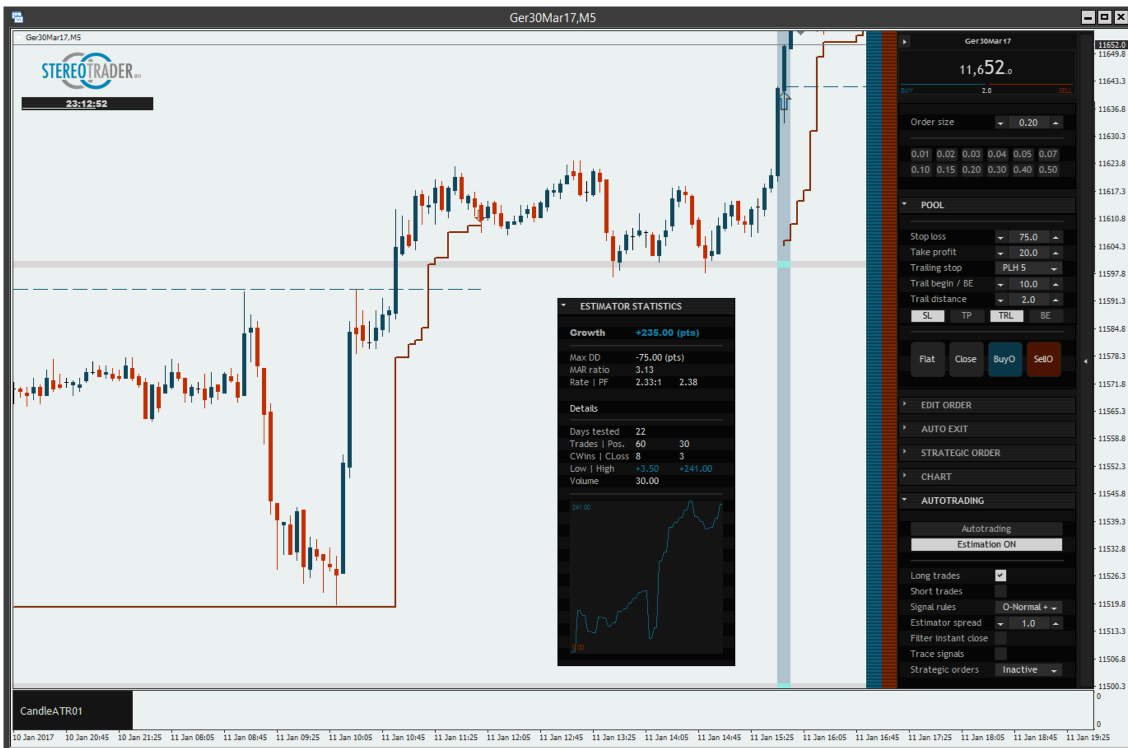
```
SEA_EVALUATE
   {
   if (…)
      Buy();
```

```
        else
           Sell();
        }
```

4. At the end of the file, use the macro

```
DECLARE_SEA_END
```

This is all what needs to be coded basically.



(StereoTrader with Sample SEA loaded, trades are visualized by the Estimator)

1.5 Event-Processing

The code of every SEA is executed within event function blocks. Such functions are processed during initialization, timer processing or when the user changes a parameter. For each event there is a pre defined function and the code should be located only within such functions. Every function is represented by a macro, these are explained below.

Do not confuse with the complexity. In most cases you will need only the main functions which are SEA_INIT, SEA_INDICATE, SEA_EVALUATE.

```
SEA_INIT {}
```

Executed one time at initialization/start of the SEA. This block is used for variable initializations and to set up parameters of the SEA.

Return value: *true* in case of success, *false* if not.

```
SEA_DEINIT {}
```

Executed one time before the SEA is terminated

SEA_DEACTIVATE {}

Executed when SEA is deactivated, e. g. when AutoTrading was switched off. The function is also executed prior to SEA_DEINIT.

SEA_RESET {}

Executed every time when the SEA is resetted. Such a reset occurs whenever the calculation/indication becomes invalid. In other words, when the user changes a parameter at the panel, at StereoTrader panel etc.. The function is also executed after SEA_INIT during the first reset.

SEA_INITPANEL {}

Executed one time before the first execution of SEA_EVALUATE

SEA_INDICATE {}

Executed with every bar. Used for calculations and drawings, does NOT accept any trade commands such as *Buy() or Flat()*

SEA_EVALUATE {} *

Executed with every bar or tick after SEA_INDICATE. Shall be used for calculations and trade command executions such as *Buy(), BuyOrder(), Flat()* etc.

SEA_TIMER {} *

Executed on timer intervals, as specified by *SetTimer()* function during SEA_INIT. The function does also accept trade commands.

SEA_USER {} *

Executed whenever the user changes a parameter at the SEA-panel or StereoTrader-panel, prior to SEA_RESET. The function does accept trade commands.

SEA_CHARTEVENT {}

Executed with any chart-event and identical to the MT4/MT5 function. Please refer to the documentation of MetaTrader for additional information. The function parameters *id, lparam, dparam* and *sparam* are passed as usual and return code is *void*.

SEA_LOADDATA {}

Executed when SEA loads preset data. You may use this function to load additional data, which may also be saved with the preset file. Available functions are

```
double LoadDouble(void)
bool   LoadBool(void)
int    LoadInt(void)
string LoadText(void)
```

Please note that all data is stored sequentially, therefore the order and count of data during loading and saving must be always identical.

If you need to access the preset file on your own, the file handle is stored in the variable *m_filehandle*.

SEA_SAVEDATA {}

Executed when SEA saves preset data. You may use this function to save additional data, which may also be saved with the preset file. Available functions are

```
void   SaveDouble(double value)
void   SaveBool(bool value)
void   SaveInt(int value)
void   SaveText(string value)
```

Please note that all data is stored sequentially, therefore the order and count of data during loading and saving must be always identical.

If you need to access the preset file on your own, the file handle is stored in the variable *m_filehandle*.

SEA_LOADDEFAULT {}

Executed instead of SEA_LOADDATA when preset file does not exist.

** All functions with an asterisk allow for the usage of trade commands*

2. Processing logic

It´s also necessary to understand the flow of processing, before you start to develop own strategies.

There are two key functions, SEA_INDICATE and SEA_EVALUATE. SEA_INDICATE is called for every bar in the chart, SEA_EVALUATE is only called when evaluation is requested, e. g. when autotrading is used or estimation is activated.

The function macro SEA_INDICATE can be used to separate indications, such as drawings, calculations etc. from evaluation processes. The function SEA_INDICATE is called one time for each bar.

SEA_EVALUATE is processed for every bar from left to right which could produce a signal. In case of operating tick by tick, SEA_EVALUATE is called on every tick. In case of operating with finished bars only, SEA_EVALUATE is called only when a new bar is opened.

2.1 Orientation

As mentioned before, the evaluation functions SEA_INDICATE and SEA_EVALUATE are processed per bar or tick, from left to right, from past to present. In other words, you "are" always at the "current" bar, no matter if "current" is somewhere in the past or if it is the last bar in the chart. You don´t need to care about it, the API manages this for you.

The bars in the chart are represented by the **_Bars** object. The functions of this object mostly require a **shift** value to adress a specific bar or a specific set of bars. A shift value of zero (default) points to the current bar which either has just been opened (finished bar mode) or which is just about to be processed (tick mode).

This method of adressing bars is completely different to native MQL programming, but makes it much more easy to produce readable and compact code.

For example, if you want to evaluate the ATR of the recently finished 14 bars, the function call would just be:

```
int atr=_Bars.ATR(14,1);
```

And if you need to know if the bar before the previously closed bar is a bullish hammer, the code would be:

```
if (_Bars.IsHammer(2) && _Bars.IsBullish(2)) ...
```

Additionaly, there are some functions to support the orientation:

IsFirstBar ()

returns true if the very first bar (left side) is processed

IsLastBar ()

returns true if the very last bar (right side) is processed

IsNewHour()

returns true if the current bar starts in a new hour

IsNewDay ()

returns true if the current bar starts in a new day

IsNewWeek()

returns true if the current bar starts in a new week

IsNewBar ()

returns true if a new bar is built and another one is just finished. If one does not operate on every tick, this value is always true.

_IsCurrentDay(int shift=0)

True if the date of the adressed bar is in the current day (local time)

_IsCurrentWeek(int shift=0)

True if the adressed bar is in the current week (local time)

_IsInPeriod(int days, int shift=0)

True if the adressed bar is in in a time range, measured from local time until the specified number of days backwars. This function is useful, if an indication shall be displayed only for e. g. 14 days backwards.

_BarsCount

Contains the number of all bars in the chart

_Index

Contains the index of the current processed bar from left to right, whereby 0 is the very first bar on the left side. This variable may be used, if an SEA needs access to an array/buffer which is part of an indication built in *OnCalculate*().

For example, the following code is an extract from an SEA which builds and visualizes the ATR by the usage of *OnCalculate*() and stores the results into the buffer *m_atr[]*. Within *SEA_EVALUATE*, the buffer values are accessible directly by *_Index*.

```
SEA_EVALUATE
   {
   double atr=m_atr[_Index];
   }
```

_Shift

Current shift value, if it´s necessary to calculate indications which are not part of the API. Without any headache, it´s easy to add additional calculations which are based on native MQL.

```
SEA_EVALUATE
   {
   double mfi=iMFI(NULL,PERIOD_CURRENT,20,_Shift);
   }
```

3. Accessing bars

3.1 The _Bars object

As mentioned before, specific bars or specific sets of bars are accessible by the object **_Bars**. The parameter *shift* points to the relative position of the bar in the chart, measured from the current bar. A *shift* value of zero points to the current bar whereby a value of 1 points to the previously closed bar and so on.

3.1.1 Basic data

```
double _Bars.Open(shift=0)
double _Bars.Close(shift=0)
double _Bars.High(shift=0)
double _Bars.Low(shift=0)
```

Returns the rates of the selected bar.

```
datetime _Bars.Time(shift=0)
```

Returns the raw time data of the selected bar.

```
string  _Bars.TimeString(shift=0)
```

Returns the time data of the selected bar as string.

```
string  _Bars.DateString(shift=0)
```

Returns the date of the selected bar as string.

```
int _Bars.Year(shift=0)
int _Bars.Month(shift=0)
int _Bars.Day(shift=0)
int _Bars.Hour(shift=0)
int _Bars.Minute(shift=0)
int _Bars.Second(shift=0)
int _Bars.DayOfWeek(shift=0)
int _Bars.DayOfYear(shift=0)
```

Returns time and date values of a selected bar.

```
long _Bars.RealVolume(shift=0)
```

Returns the real volume (MT5) of the selected bar.

```
long _Bars.TickVolume(shift=0)
```

Returns the tick volume of the selected bar.

```
double _Bars.Gap(shift=0)
```

Returns the range of the gap between the selected bar and the bar before.

3.1.2 Shapes and measuring

```
bool _Bars.IsBullish(shift=0)
bool _Bars.IsBearish(shift=0)
bool _Bars.IsDoji(double tolerance=0, shift=0)
```

Evaluates if a bar is bullish or bearish or a doji.

```
double _Bars.Range(shift=0)
```

The range of a bar from high to low.

```
double _Bars.TrueRange(shift=0)
```

The True Range of a bar including gap – if present.

```
double _Bars.BodyRange(shift=0)
double _Bars.WickRange(shift=0)
double _Bars.TailRange(shift=0)
```

The range of the bars body, wick or tail.

```
double _Bars.BodyPart(shift=0)
double _Bars.WickPart(shift=0)
double _Bars.TailPart(shift=0)
```

The part of either the body, the wick or the tail of bar. The return value is always between 0.0 and 1.0 (0% to 100%)

```
bool _Bars.IsHammer(shift=0)
bool _Bars.IsShootingStar(shift=0)
```

Returns the shape of a bar.

```
double _Bars.FiboPrice(double levelmult, shift=0)
```

Returns the fibonacci price of a specific bar. The parameter *levelmult* specificies the level as multiplication value.

```
double _Bars.PivotPrice(ENUM_PIVOT_POINT p, shift=0)
```

Returns the pivot price of a specific bar. The parameter *p* specificies the requested pivot point, which is one of the following:

```
PIVOT_POINT_MAIN
PIVOT_POINT_S1
PIVOT_POINT_S2
PIVOT_POINT_R1
PIVOT_POINT_R2
```

3.1.3 Bar interaction and pattern

```
bool _Bars.IsCrossing(double price, int shift=0)
```

Returns true if the bar is crossing a price level

```
bool _Bars.IsTouchingHigh(double price, int shift=0)
```

Returns true if the bars wick is touching or crossing a price level

```
bool _Bars.IsTouchingLow(double price, int shift=0)
```

Returns true if the bars tail is touching or crossing a price level

```
bool _Bars.IsEngulfingLong(int shift=0)
bool _Bars.IsEngulfingShort(int shift=0)
```

Returns true in case of a long/short engulfing pattern

```
bool _Bars.IsReversingLong(int shift=0)
bool _Bars.IsReversingShort(int shift=0)
```

Returns true in case of a long/short reversal pattern

3.1.4 Advanced

```
Bool GetOLHC(double &o, double &l, double &h, double &c, int shift=0)
```

Returns open, low, high and close values of a specific bar

```
CBar * _Bars.BarGet(int shift=0)
```

Returns an object pointer to a CBar object which holds data of a specific bar.

```
bool BarGetSeries(CBar &bars[], int cntbars, int shift=0)
```

Fills an CBar array with data.

```
bool BarUpdate(CBar &bar, int shift=0)
```

Updates a CBar object

3.1.5 Indications

The **_Bars** object allows also access to multiple bars, for indications etc. The currently implemented functions are:

```
bool _Bars.IsCrossingUp (series[], int shift=0)
bool _Bars.IsCrossingDown (series[], int shift=0)
```

Returns true if the selected bar is crossing a series buffer.

```
double _Bars.ATR (int span=14, int shift=0)
```

Calculates the Average True Range.

```
double _Bars.AR (int span=14, int shift=0, ENUMARMODE mode=ARMODENORMAL,
ENUMCANDLEBASE base=WRONGVALUE)
```

Calculates the Average Range of a specific number of bars. The function is able to calculate the range only of open and close prices, if base is set to CANDLE_BASE_OPENCLOSE.

Furhtermore, when mode is set to ATR_MODE_BEARISH or ATR_MODE_BULLISH, the function calculates only the range of bullish or bearish bars.

```
double _Bars.HighestHigh (int span, int shift=0)
double _Bars.LowestLow (int span, int shift=0)
```

Returns the highest high or lowest low in a specific range of bars.

```
double _Bars.SMA (int periods, int shift=0)
double _Bars.EMA (int periods, int shift=0)
double _Bars.SMMA (int periods, int shift=0)
double _Bars.LWMA (int periods, int shift=0)
```

Calculates different types of moving averages.

```
long _Bars.TickVolumeAverage(int periods=14, int shift=0)
long _Bars.RealVolumeAverage(int periods=14, int shift=0)
```

Calculates averages of volume.

```
double  _Bars.RSI(int periods=14, int shift=0)
```

Calculates the RSI

```
double _Bars.StochasticMain(int period_k=5, int period_d=3, int slowing=3, int shift=0)
double _Bars.StochasticSignal(int period_k=5, int period_d=3, int slowing=3, int shift=0)
SIGNAL _Bars.Stochastic(int period_k, int period_d, int slowing, double signallinedistance=20, int shift=0)
```

Calculates stochastic.

```
bool _Bars.IsLocalHigh(int bars, int shift=0, double tolerance=0)
bool _Bars.IsLocalLow(int bars, int shift=0, double tolerance=0)
```

Evaluates if a bar marks a local high or low

```
bool _Bars.IsSMASlopingUp(int period, int span=2, int shift=0)
bool _Bars.IsEMASlopingUp(int period, int span=2, int shift=0)
bool _Bars.IsLWMASlopingUp(int period, int span=2, int shift=0)
bool _Bars.IsSMMASlopingUp(int period, int span=2, int shift=0)
bool _Bars.IsSMASlopingDown(int period, int span=2, int shift=0)
bool _Bars.IsEMASlopingDown(int period, int span=2, int shift=0)
bool _Bars.IsLWMASlopingDown(int period, int span=2, int shift=0)
bool _Bars.IsSMMASlopingDown(int period, int span=2, int shift=0)
```

Evaluates if moving average is sloping upwards or downwards

3.2 Single bars

Besides accessing a series of bars, there are also objects which represent a single bar. These objects belong to the class **CBar**. Of course it´s possible to define also own instances of such an CBar object.

3.2.1 Predefined single bar objects

The predefined single bar objects are:

_CurBar

The current bar.

_PrevBar

The previously finished bar

_CurHour

The current hour

_PrevHour

The previous hour

_CurDay, **_PrevDay**, **_CurWeek** and **_PrevWeek**.

Example:

```
bool hammer=_CurBar.IsHammer();
datetime t=_PrevBar.Time;
double weekhigh=_CurWeek.High;
```

3.2.2 Data of the current bar

To make coding easier, the following variables contain already some information about the current bar, copied from the ***_CurBar*** object.

_Price

The deal price. In case if an SEA operates with each tick, the variable returns the current close price of the unfinished bar. Otherwise it´s the open price of the next bar.

_Bid / _Ask

The corresponding bid or ask price

Some self explaining variables which return results of the current – the new – bar. In case if the SEA operates with finished bars only, the variables _Low, _High, _Open and _Close have all the same value: The open price.

_Open

_Low

_High

_Close

_TickVolume

_RealVolume

_Time

_Hour

_Minute

_Day

_DayOfWeek

_Month

_Year

Further predefined variables which contain data of the current bar:

_SymbolName

Contains the current symbol name

_TimeFrame

Contains the current timeframe

_BarsCount

The number of bars in the chart

3.2.3 Basic data of single CBar objects

Analogue to the functions of the **_Bars** object, CBar objects have also functions and member variables to access the data of bars. Of course, single bars never use a *shift* parameter, because such objects always represent one specific bar.

Member variables:

`double` **.Open**

`double` **.Close**

`double` **.High**

`double` **.Low**

Contain the specific rate of a bar.

Please note: In case of the **_CurBar** object while not operating in tick mode, these member variables have all the same value to avoid miscalculations when the Estimator is active.

`datetime` **.Time**

Contains the raw time data.

`int` **.Year**

`int` **.Month**

`int` **.Day**

`int` **.Hour**

`int` **.Minute**

`int` **.Second**

`int` **.DayOfWeek**

`int` **.DayOfYear**

Contain time and date values.

`long .RealVolume`

Contains the real volume (MT5 only).

`long .TickVolume`

Contains the tick volume.

`double .Gap`

Contains the range of the gap between the bar in relation to the previous bar.

Member functions/methods:

`string .TimeString()`

Returns the time data of the selected bar as string.

`string .DateString()`

Returns the date of the selected bar as string.

3.2.4 Member functions for shapes and measuring

```
bool .IsBullish()
bool .IsBearish()
bool .IsDoji(double tolerance=0)
```

Evaluates if a bar is bullish or bearish or a doji.

`double .Range()`

The range of a bar from high to low.

`double .TrueRange()`

The True Range of a bar including gap – if present.

```
double .BodyRange()
double .WickRange()
double .TailRange()
```

The range of the bars body, wick or tail.

```
double .BodyPart()
double .WickPart()
double .TailPart()
```

The part of either the body, the wick or the tail of bar. The return value is always between 0.0 and 1.0 (0% to 100%)

```
bool .IsHammer(double tailmin=0.55)
bool .IsShootingStar(double wickmin=0.55)
```

Return the shape of a bar.

```
bool .IsSolid(double wickmax=0.15, double tailmax=0.15)
```

Returns true if the bar is solid whereby wick and tail must not exceed the specified maximal values for wick and tail.

```
double .FiboPrice(double levelmult, shift=0)
```

Returns the fibonacci price of a specific bar. The parameter *levelmult* specificies the level as multiplication value.

```
double .PivotPrice(ENUM_PIVOT_POINT p, shift=0)
```

Returns the pivot price of a specific bar. The parameter *p* specificies the requested pivot point, which is one of the following:

```
PIVOT_POINT_MAIN
PIVOT_POINT_S1
PIVOT_POINT_S2
PIVOT_POINT_R1
PIVOT_POINT_R2
```

3.1.5 Bar interaction

```
bool .IsCrossing(double price)
```

Returns true if the bar is crossing a price level

```
bool .IsBreaking(double price)
```

Returns true if the body of the bar crosses a specific price level.

```
bool .IsBreakingLong(double price, double exceed=0)
bool .IsBreakingShort(double price, double exceed=0)
```

Returns true if the body of the bar crosses *price* bullish/bearish.

```
bool .IsTouchingHigh(double price)
```

Returns true if the bars wick is touching or crossing a price level

```
bool .IsTouchingLow(double price)
```

Returns true if the bars tail is touching or crossing a price level

4. Signals

4.1 Setup functions

Any SEA gets registered automatically by the base class. The function SEA_INIT (*OnInit*()) may be used to define behaviour and parameters of the StereoEA, but is not needed in any case. Any such property should be set during *OnInit.*

```
SEA_INIT
   {
   SetEveryTick(true);
   SetTradeModes(TRADE_MODE_STEREO_FUTURE);
   //--- Other stuff
   //...
   //--- Success
   return true;
   }
```

Any initialization is optional. All of these function may be used for own purposes if needed only.

```
void SetOnCalculate (bool flag, bool arraysasseries)
```

Indicates that the SEA is using a standard *OnCalculate*() function block for indications, as it is the normal way when developing indicators with MQL. If this property is not set, the OnCalculate() will not be called.

The parameter *arrayasseries* specifies, that the passed arrays such as *close [], open[]* will be preprocessed before *OnCalculate*() is used. For more informations about *OnCalculate*(), please refer to the MQL language reference of MetaQuotes.

```
void SetIndicate (bool flag)
```

Informs the API if the SEA uses an indication block. This provides an alternative way to use indications which may draw on the chart. The SEA_INDICATE block is always called prior to SEA_EVALUATE and within its not possible to use any trade commands. By default, this property is activated.

```
SEA_INDICATE
   {
   if (_Hour==m_time1.Hour() && _Minute==m_time1.Minute())
      {
      DrawLine("Timeline @%price (%time)",_Price,clrDarkOrange,_Time);
      return;
   }
```

```
void SetEvaluate (bool flag)
```

Tells the API if the SEA uses an evaluation block, which is used to evaluate candles and to send trade commands. By default, this property is activated.

```
SEA_EVALUATE
   {
   if (...)
      Buy();
   }
```

```
void SetDaysToProcess(int days=0)
```

Limits the number of the past days to process for indication and estimation. You may use this function to speed up processing when using complex indications.

```
void SetTimeWindow(int beginh, int beginm, int endh, int endm)
```

Limits the hours of processing to a specific time range.

```
void SetDefaultShift(shift=1)
```

The default number of bars which are shifted backwards in any evaluations.

(Default shift 1 – left side – and default shift 0 – right side)

On the left side, when dealing with finished candles, the default shift is 1, which means any shift-value is added by 1 and the not yet finished bar is accessible by a shift value of -1.

_Bars. Example:

```
SetDefaultShift(1);

if (_Bars.IsHammer())     //--- last closed bar
   Buy();
```

During SEA_EVALUATE, this example detects a hammer in the previous bar. In case if the default shift would be 0, the same example would have to look like this

```
DefaultShift(0);

if (_Bars.IsHammer(1))    //--- last closed bar
   Buy();
```

```
void SetEveryTick (bool flag)
```

Indicates if the function macro *SEA_EVALUATE* is called on every tick. If this property is not set, the function will be called only when a new bar/candle is about to be created. Nevertheless, during *SEA_EVALUATE* you still have the additional possibility to test, if a new bar has just been created by using the *IsNewBar*() function to avoid unnecessary calculation time.

```
void SetSignalRules (ENUM_SEA_OPPOSITEFLAG flag, bool overlap)
```

Defines the behaviour in case of counter signals, e. g. Buy() when short positions are opened, or Sell() when long positions are active. Possible values are

```
SEA_OPPOSITE_NONE           // Open the position
SEA_OPPOSITE_NOTRADE        // Prohibit opening
SEA_OPPOSITE_CLOSE          // Close opposite
```

The parameter *overlap* specifies if multiple signals/positions of the same direction (Buy() / Sell()) are allowed at the same time. By default, this is restricted to one position/signal.

```
void SetSignalMinDistance (int bars)
```

Specifies the minimal distance between Buy() and Sell() signals, meaured in bars. By default, this value is 1, which means, only one signal per bar is allowed.

```
void SetRoom(double roompoints)
```

Predefines a minimal room between orders to prevent overlapping orders. The parameter *roompoints* defines the space around any new order which shall be verified first. If *roompo ints* is set to 10, StereoTrader would not create an order if there is already another order or position of the same direction within in a distance of 10 points.

```
void SetTradeModes (supportedmode=0xFFFF)
```

supportedmode defines the mode in which the SEA works, whereby – *0xFFFF* (standard) means all modes. The following enum constants are defined and may be combined by a bitwise OR operation:

```
TRADE_MODE_SINGLE_HEDGE = 1
TRADE_MODE_SINGLE_NOHEDGE = 2
TRADE_MODE_STEREO_HEDGE = 4
TRADE_MODE_STEREO_FUTURE = 8
```

```
void SetTrueRangeMode (supportedmode=0xFFFF)
```

supportedmode defines the true range measurement method. If such a mode is set, all bar values are calculated on the base of their true range including gaps. Hereby two different modes may be used, the normal mode, which adds gaps to the bar after the gap, and the forwarding mode, which adds the gap to the previous bar.

```
TRUERANGE_MODE_NONE = 0
TRUERANGE_MODE_NORMAL = 1
TRUERANGE_MODE_FORWARD = 2
```

```
void SetUsesVars(bool flag=true)
```

Indicates that local trading variables are used. By default, this is enabled. A disabling increased the backtesting speed, in case if no variables are used.

```
void SetExpirationDate(datetime date)
```

Limits the usablity of an SEA to a specific date.

4.2 Basic commands

When talking about signals, primilary the functions Buy() and Sell() are meant.

4.1.1 Open/close positions

```
void Buy (string info=NULL)
void Sell (string info=NULL)
```

Sends a signal to open a buy/sell position or to activate the strategic long/short order in dependence of which order mode is selected at the *AutoTrading* panel.

The parameter info is used as tooltip text of the signal arrow which is displayed on the chart.

```
void Flat ();
void FlatLong ();
void FlatShort ();
```

Close all positions and delete all orders.

```
void CloseBuy (double profitpoints=-9999999)
void CloseSell (double profitpoints=-9999999)
```

Close all buy/sell positions

```
void CloseAll (double profitpoints=-999999)
```

Close all positions

4.1.2 Blocking / Unblocking

```
void BlockBuy()
```

Sends a signal to prevent autotrading from opening any buy positions or sending any buy orders of any SEA.

```
void BlockSell()
```

Sends a signal to prevent autotrading from opening any sell positions or sending any sell orders of any SEA

```
void BlockAll()
```

Sends a signal to prevent autotrading from opening any positions or sending any orders of any SEA.

4.2 Pending orders

```
void BuyOrder (double price)
void SellOrder (double price)
```

Sends a signal to place a buy or sell order at the level of the parameter *price.* If price is below the current price, a limit order is sent, if the price is above, a stop order is generated.

```
void BuyMTO (double distance=2)
void SellMTO (double distance=2)
```

Sends a signal to open a market trail order (MTO) for either buying or selling. The MTO variant can be more effective than a normal market order, especially when sending after a bar was just closed.

```
void DeleteBuyOrders()
```

`void DeleteSellOrders()`

Remove all buy/sell orders.

`void DeleteAllOrders()`

Guess … ;)

4.2.1 Order settings

Any attribute/setting of an order must be sent prior to the actual command which places an order.

`void OrderId (string id)`

Defines an order id. The value is used to identify orders sent by the SEA for further modification, selection or deletion. There is no obligation to use this function, orders will also be placed without an id.

`void OrderSize (double size)`

Sets the absolute size of a new order or position as lot value.

`void OrderSizeRel (double factor)`

Multiplies the current order size with the value in *factor.*

`void OrderComment (string comment)`

Defines the comment for any following new order. In case of multiple orders in a row with different comments, this function must be called prior to each function, which generates an order or opens a position.

`void OrderLimitPullback (double points)`

Defines the limit pullback for any following new limit order. In case of multiple orders in a row with different values, this function must be called prior to each function, which generates a limit order.

`void OrderAttrOCO (bool flag=true)`

Sets the OCO flag (order cancels others) for any following new order. In case of multiple orders in a row with different flags, this function must be called prior to each function, which generates an order.

`void OrderAttrCO (bool flag=true)`

Sets the CO flag (cancel opposite orders) for any following new order. In case of multiple orders in a row with different flags, this function must be called prior to each function, which generates an order.

`void OrderAttrREV (bool flag=true)`

Sets the REV flag (reverse position) for any following new order. In case of multiple orders in a row with different flags, this function must be called prior to each function, which generates an order.

`void OrderAttrNET (bool flag=true)`

Sets the NET flag (order compensates opposite positions) for any following new order. In case of multiple orders in a row with different flags, this function must be called prior to each function, which generates an order.

```
void OrderTrail (ENUM_SEA_TRAILMODE mode, double distancepts, int period=0)
```

Indicates that the next order which is sent shall be trailed with the given parameters. The following values can be used for the parameter *mode*

```
SEA_TRAIL_NONE =       0
SEA_TRAIL_DST =        1
SEA_TRAIL_PLH =        2
SEA_TRAIL_MA =         3
SEA_TRAIL_PLH_PEAK =   5
SEA_TRAIL_PLH_CLOSE =  6
```

distancepts is the trailing distance as point value, *period* is only used for trailing with moving averages or other periodic trails.

4.2.2 Order modification

```
void MoveOrder (string id, double price)
```

Moves a pending order to another price. To use this function proper, the command *OrderId()* which defines the id of an order has to be used prior with unique names for any new order.

4.3 Managing trades

4.3.1 Stereo Future mode

The following functions are to be used in single mode and the Stereo Future mode.

```
void SL (double points)
```

Sets or modifies the stop loss.

```
void SLAbsolute (double price)
```

Sets the stop loss to an absolute price. If the postion is not opened yet, the value is saved and the TP will be adjusted as soon as the position is opened.

Please note, that this commands have to be used after any order command such as *.Buy()* or *.BuyOrder()*, otherwise prior calls of the functions have no effect.

```
void TP (double points)
```

Sets the take profit.

```
void TPAbsolute (double price)
```

Sets the take profit to an absolute price. If the postion is not opened yet, the value is saved and the TP will be adjusted as soon as the position is opened.

Please note, that this commands have to be used after any order command such as *Buy()* or *BuyOrder()*, otherwise prior calls of the functions have no effect.

```
void TrailMode (ENUM_SEA_TRAILMODE mode, double distancepoints, int periods=0)
```

Defines the trailing mode.

The parameter *mode* defines the mode, *distancepoints* the distance as points. In case of periodic low/high trailing or trailing based on moving averages, the paramter *periods* defines the periods to be used. *mode* is one of the following:

```
SEA_TRAIL_NONE =       0
SEA_TRAIL_DST =        1
SEA_TRAIL_PLH =        2
SEA_TRAIL_MA =         3
SEA_TRAIL_EOP_DST =    4
SEA_TRAIL_PLH_PEAK =   5
SEA_TRAIL_PLH_CLOSE =  6
```

`void` **`TrailDistance`** `(double points)`

Defines the trailing distance.

`void` **`TrailBegin`** `(double points)`

Defines where trailing begins, relative to the break even.

The value defines also the break even trigger, in case if *BEActivate* is used to save the break even.

`void` **`BEAdd`** `(double points)`

Defines the additional points to add to break even.

`void` **`SLActivate`** `(bool flag)`

Activates or deactivates the stop loss.

`void` **`TPActivate`** `(bool flag)`

Activates or deactivates the take profit.

`void` **`TrailActivate`** `(bool flag)`

Activates or deactivates the trailing stop.

`void` **`BEActivate`** `(bool flag)`

Activates or deactivates the break even save function.

4.3.2 Functions for Stereo Hedge mode

The following functions are to be used in virtual mode only.

`void` **`SLBuy`** `(double points)`

`void` **`SLSell`** `(double points)`

Sets the stop loss as points of opened buy/sell positions within the pool.

`void` **`SLBuyAbsolute`** `(double price)`

`void` **`SLSellAbsolute`** `(double price)`

Sets the stop loss of opened positions within the pool to an absolute price. If the postion is not opened yet, the value is saved and the TP will be adjusted as soon as the position is opened.

Please note, that this commands have to be used after any order command such as *.Buy()* or *.BuyOrder()*, otherwise prior calls of the functions have no effect.

```
void TPBuy (double points)
```

```
void TPSell (double points)
```

Sets the take profit as points of opened buy/sell positions within the pool.

```
void TPBuyAbsolute (double price)
```

```
void TPSellAbsolute (double price)
```

Sets the take profit of opened positions within the pool to an absolute price. If the postion is not opened yet, the value is saved and the TP will be adjusted as soon as the position is opened.

Please note, that this commands have to be used after any order command such as *.Buy()* or *.BuyOrder()*, otherwise prior calls of the functions have no effect.

```
void TrailModeBuy (ENUM_SEA_TRAILMODE mode)
```

```
void TrailModeSell (ENUM_SEA_TRAILMODE mode, double distancepoints,
int periods=0)
```

Sets the trailing mode of opened buy/sell positions within the pool.

The parameter *mode* defines the mode, *distancepoints* the distance as points. In case of periodic low/high trailing or trailing based on moving averages, the paramter *periods* defines the periods to be used. *mode* is one of the following:

```
SEA_TRAIL_NONE =       0
SEA_TRAIL_DST =        1
SEA_TRAIL_PLH =        2
SEA_TRAIL_MA =         3
SEA_TRAIL_EOP_DST =    4
SEA_TRAIL_PLH_PEAK =   5
SEA_TRAIL_PLH_CLOSE =  6
```

```
void TrailDistanceBuy (double points)
```

```
void TrailDistanceSell (double points)
```

Sets the trailing distance in points of opened buy/sell positions within the pool.

```
void TrailBeginBuy (double points)
```

```
void TrailBeginSell (double points)
```

Defines where trailing begins. The value defines also the break even trigger, in case if *BEActivateBuy or BEActivateSell* is used to save the break even.

Sets the break even trigger value in points of opened buy/sell positions within the pool.

```
void BEAddBuy (double points)
```

```
void BEAddSell (double points)
```

Defines the additional points to add to break even of opened buy/sell positions within the pool.

```
void SLActivateBuy (bool flag)
void SLActivateSell (bool flag)
```

Stop loss activation/deactivation of opened buy/sell positions within the pool.

```
void TPActivateBuy (bool flag)
void TPActivateSell (bool flag)
```

Take profit activation/deactivation of opened buy/sell positions within the pool.

```
void TrailActivateBuy (bool flag)
void TrailActivateSell (bool flag)
```

Trailing stop activation/deactivation of opened buy/sell positions within the pool.

```
void BEActivateBuy (bool flag)
void BEActivateSell (bool flag)
```

Break even save activation/deactivation of opened buy/sell positions within the pool.

4.3.3 Commands for manual exits

The following functions are to be used for manual exits and are almost similar to the buttons below the order panels.

```
void Flat() / FlatLong() / FlatShort()
```

Closes all positions and deletes all orders.

```
bool IsFlat (bool checkorders=false)
```

Returns true when there is not opened position. If *checkorders* is set to *true*, pending orders are also recognized.

```
void CloseBuy (string id=NULL, double minprofit=-999999999)
```

Sends a signal to close all buy positions of the SEA. In case of strategic ordering, this also forces a deactivation of the corresponding *Stay* functionality.

If *id* is specified, the function affects only orders which match the id, whereby wildcards my be used in the id.

If *minprofit* is specified, the function forces only the closing of all particular buy positions which have a profit of at least the specified value as points.

```
void CloseSell (string id=NULL, double minprofit=-999999999)
```

Sends a signal to close all sell positions of the SEA. In case of strategic ordering, this also forces a deactivation of the corresponding *Stay* functionality.

If *id* is specified, the function affects only orders which match the id, whereby wildcards my be used in the id.

If *minprofit* ist specified, the function forces only the closing of all particular sell positions which have a profit of at least the specified value as points.

```
void CloseAll (string id=NULL, double minprofit=-999999999)
```

Sends a signal to close all positions of the SEA. In case of strategic ordering, this forces a deactivation of the *Side* algorithm and its *Stay* functionality.

If *id* is specified, the function affects only orders which match the id, whereby wildcards my be used in the id.

If *minprofit* ist specified, the function forces only the closing of all particular buy or sell positions which have a profit of at least the specified value as points.

```
void DeleteBuyOrders(string id=NULL)
```

Sends a signal to deletes any pending buy orders of the SEA. If *id* is specified, the command will only affect such orders which match the specified id value. Hereby the value for id may be used with wildcards. Example for wildcards:

DeleteBuyOrders("OrderName")* – would delete all orders from this SEA with an id that starts with the letters "OrderName".

*DeleteBuyOrders("*OrderName")* – would delete all orders from this SEA with an id that ends with the letters "OrderName".

DeleteBuyOrders("OrderName") – would delete all orders created by this SEA with an id that matches "OrderName" exactly.

In case of the usage of strategic orders, this also forces a deactivation of the corresponding *Force* functionality

```
void DeleteSellOrders(string id=NULL)
```

Sends a signal to deletes any pending sell orders of the SEA. If *id* is specified, the command will only affect such orders which match the specified id value. Hereby the value for id may be used with wildcards.

In case of strategic ordering, this also forces a deactivation of the corresponding *Stay* functionality

```
void DeleteAllOrders(string id=NULL)
```

Sends a signal to deletes any pending orders of the SEA. If *id* is specified, the command will only affect such orders which match the specified id value. Hereby the value for id may be used with wildcards.

In case of strategic ordering, this also forces a deactivation of the *Stay Side* functionality.

4.4 Commands for automated exits

The following functions are to be used to control further automated exists, such as displayed at the *AutoExit* panel. Please refer to the description of auto exiting to learn about the functionality of all its parameters.

`void AEPoolSL (double points)`

Sets the pool stop loss in points.

`void AEPoolTP (double points)`

Sets the pool take profit in points.

`void AEPoolSLActivate (bool flag)`

Activates or deactivates the Pool stop loss.

`void AEPoolTPActivate (bool flag)`

Activates or deactivates the Pool take profit.

`void AETradeMaxReduction (double amount)`

Sets the Trade P/L stop loss as amount. If *amount* is negative, it´s interpreted as percentage value of the current equity.

`void AETradeMaxGrowth (double amount)`

Sets the Trade P/L take profit as amount. If *amount* is negative, it´s interpreted as percentage value of the current equity.

`void AETradeMaxReductionActivate (bool flag)`

Activates or deactivates the Trade P/L stop loss.

`void AETradeMaxGrowthActivate (bool flag)`

Activates or deactivates the Trade P/L take profit.

`void AETradeTrailBegin (double amount)`

Specifies the beginning of the Trade P/L trailing stop as amount.

`void AETradeTrailDistance (double amount)`

Specifies the distance of the Trade P/L trailing stop as amount.

`void AETradeTrailActivate (bool flag)`

Activates or deactivates trailing stop loss for Trade PL

`void AEEquitySL (double amount)`

Sets the Equity stop loss as amount. If *amount* is negative, it´s interpreted as percentage value of the current equity.

`void AEEquityTP (double amount)`

Sets the Equity take profit as amount. If *amount* is negative, it´s interpreted as percentage value of the current equity.

`void` **`AEEquitySLActivate`** `(bool flag)`

Activates or deactivates the Equity stop loss.

`void` **`AEEquityTPActivate`** `(bool flag)`

Activates or deactivates the Equity take profit.

`void` **`AEEquityDeactivate`** `(bool flag)`

Enables or disables the automated deactivation of any automated processes when the Equity SL or Equity TP was triggered.

`void` **`AETimeBegin`** `(int hh, int mm)`

Sets the begin time.

`void` **`AETimeFlat`** `(int hh, int mm)`

Sets the flat time.

`void` **`AETimeFlatActivate`** `(bool flag)`

Enables or disables the time based exit.

`void` **`AERemoveOrdersFlat`** `(bool flag)`

Activates or deactivates the automated deletion of remaining orders whenever all positions are closed.

4.5 Commands for strategic orders

The following functions are for modification of the settings for strategic orders as shown at the *Strategic Order* panel. Please refer to the description of strategic orders to understand the functionality of all its parameters.

`void` **`SOrderATRMode`** `(string mode)`

Defines the ATR mode as displayed in the drop down list.

`void` **`SOrderDistance`** `(double points)`

Defines the distance between particular orders as points. If ATR is used, the parameter points becomes a multiplier for the calculated ATR (average true range).

`void` **`SOrderSLRel`** `(double points)`

Defines the relative distance for stop loss, based on the absolute distance between orders.

`void` **`SOrderTrailBeginRel`** `(double points)`

Defines the relative distance for the trail begin, based on the absolute distance between orders.

`void` **`SOrderTrailDistanceRel`** `(double points)`

Defines the relative distance for trailing, based on the absolute distance between orders.

```
void SOrderProgressLimit (double factor)
```

Sets the factor of order size progression of limit orders.

```
void SOrderProgressStop (double factor)
```

Sets the factor of order size progression of stop orders.

```
void SOrderSizeMultiplyBuy(double size)
```

Sets the initial lot size of the first buy order or position.

```
void SOrderSizeMultiplySell (double size)
```

Sets the initial lot size of the first sell order or position.

```
void SOrderAttributes (bool attr_oco, bool attr_co, bool attr_rev, bool attr_net)
```

Activates or deactivates the attributes.

```
void SOrderInitial (string enumvalue, double distance=1)
```

Defines the type of the initial order. In case of "*MTO*" or "*EOP*", the parameter *distance* holds the trailing distance for the order.

```
void SOrderChainLimits(bool flag)
```

Activates/deactivates chaining of MIT limit pullback orders

```
void SOrderQuantity (int cntlimits, int cntstops)
```

Defines the count of orders which are to be generated.

```
void SOrderLong ()
```

Sends a signal to execute a long order strategy which equates a manual click on the button.

```
void SOrderSide()
```

Sends a signal to execute a side order strategy which equates a manual click on the button.

```
void SOrderShort()
```

Sends a signal to execute a short order strategy which equates a manual click on the button.

```
void SOrderStayLong(bool flag)
```

Activates/deactivates the stay-functionality for long strategy orders.

```
void SOrderStayShort(bool flag)
```

Activates/deactivates the stay-functionality for side strategy orders.

```
void SOrderStayShort(bool flag)
```

Activates/deactivates the stay-functionality for short strategy orders.

4.7 Messages

`void MessageFloat (string text, string title=NULL)`

Displays a floating message on the chart in a semi-transparent overlay window

`void MessageOK (string text, string title=NULL)`

Displays a message in a window which the user needs to confirm

`void MessageChart (string text, string title=NULL)`

Displays a scrolling message in the upper left corner of the chart

4.8 Other functions

`double PriceToPoints (double pricevalue)`

Converts a price based value to a point based value in accordance to the local definition of a point.

`double PointsToPrice (double pointvalue)`

Converts a point based value to a price based value in accordance to the local definition of a point.

4.9 Options

`void OptionMITLimitOrders (bool flag)`

Enables/disables the MIT functionality for limit orders.

`void OptionMITStopOrders (bool flag)`

Enables/disables the MIT functionality for stop orders.

`void OptionMITSLTP (bool flag)`

Enables/disables the MIT functionality for stop loss and take profit.

`void OptionMITSLSoft (int smaperiods)`

Defines the periods which is used to calculate the moving average for soft SL. This option takes only effect if *OptionMITSLTP(true)* was executed before or if the corresponding option was enabled at the Setup. In case of *smaperiods=0* the SL works as a normal SL and is triggered as soon the level was reached, otherwise a simple moving average is used to calculate the trigger level, which softens the stop loss on high volatility.

`void OptionSLTPAdjust (bool flag)`

If activated, this results in adjustment of stop loss and/or take profit after a further order was filled, based on the specified amount of points for SL and/or TP. If this option is

disabled, the position of stop loss and/or take profit stays fixed at the distance to the first filled order in the pool. (This option takes effect only in virtual modes)

```
void OptionOrderRoom (double roompoints)
```

Predefines a minimal room between orders to prevent overlapping orders. The parameter *roompoints* defines the space around any new order which shall be verified first. If *roompoints* is set to 10, StereoTrader would not create an order if there is already another order or position of the same direction within in a distance of 10 points.

4.9 API variables

StereoTrader provides several trade variables which are updated on every tick. These variables are called API variables.

Due to the circumstance, that SEAs work in a message queue, the provided variables are valid for the duration of one tick. This means, e. g., if a *CloseAll()* is executed or any other trade command such as Buy(), the number of opened positions, returned by *GetCntPos()* remains unchanged until the next tick forces the next execution of SEA_EVALUATE. Clear, because StereoTrader gets informed about *CloseAll()* when the code exits the SEA_EVALUATE block.

4.9.1 Common

The following functions return the values of global API variables, which are common for all SEAs. Use these functions instead of the corresponding MetaTrader4 functions to ensure, that you get correct results.

```
datetime GetServerTime()
```

Returns the server time.

```
double GetEquity()
```

Returns the current equity as double.

```
double GetBalance()
```

Returns the current account balance as double.

```
double GetSpread ()
```

Returns the current absolute spread

4.9.2 Trading data

```
double GetOrderSize()
```

Size of new order as set in the order panels.

```
double GetPL()
```

Returns the current Pool P/L as currency amount.

```
double GetPLBuy()
```

Returns the current P/L of all buy positions as currency amount.

`double` **`GetPLSell()`**

Returns the current P/L of all sell positions as currency amount.

`double` **`GetPLPoints()`**

Returns the current Pool P/L as point value.

`double` **`GetPLBuyPoints()`**

Returns the current P/L of all buy positions as point value.

`double` **`GetPLSellPoints()`**

Returns the current P/L of all sell positions as point value.

`double` **`GetPLTrade()`**

Returns the current Trade P/L as currency amount.

`double` **`GetBE()`**

Returns the accumulated break even of all positions in the pool as price value.

`double` **`GetBEBuy()`**

Returns the break even / average price of all buy positions in the pool as price value.

`double` **`GetBESell()`**

Returns the break even / average price of all positions in the pool as price value.

`int` **`GetCntPos()`**

Returns the accumulated number of all opened positions within the pool. Accumulation means in this case, if the pool contains 10 sell positions and 12 buy positions, the function would return +2.

`int` **`GetCntPosBuy()`**

Returns the number of all opened buy positions within the pool.

`int` **`GetCntPosSell()`**

Returns the number of all opened sell positions within the pool.

`int` **`GetCntOrders()`**

Returns the accumulated number of all orders within the pool. Accumulation means in this case, if the pool contains 10 sell orders and 12 buy orders, the function would return +2.

`int` **`GetCntOrdersBuy()`**

Returns the number of all buy orders within the pool.

`int` **`GetCntOrdersSell()`**

Returns the number of all sell orders within the pool.

double **GetSize()**

Returns the accumulated size of all opened positions within the pool. Accumulation means in this case, if the pool contains 10 lots on the sell side and 12 on the buy side, the function would return +2.

double **GetSizeBuy()**

Returns the summarized size of all opened buy positions within the pool.

double **GetSizeSell()**

Returns the summarized size of all opened sell positions within the pool.

If *local* is set to true, the function returns only the number of positions which belong to this client.

4.10 Status requests

The following functions are provided as soon as the SEA client was registered. Same as with initialization, the usage of these functions is optional.

bool **IsRegistered()**

Returns the registration state of the client.

ENUM_TRADE_MODE **TradeMode()**

In case if an SEA supports multiple modes, this function is used to identify the mode, in which StereoTrader is operating. The value is one of the following:

```
TRADE_MODE_SINGLE_HEDGE =   1
TRADE_MODE_SINGLE_NOHEDGE = 2
TRADE_MODE_STEREO_HEDGE = 4
TRADE_MODE_STEREO_FUTURE = 8
```

bool **IsHedging()**

Returns *true* if the current mode is capable of hedging.

bool **IsAcknowledged ()**

Checks if the SEA was acknowledged by the host.

bool **IsHostPresent ()**

Checks if StereoTrader is loaded.

bool **IsLongAllowed ()**

Returns true if long trades are allowed

bool **IsShortAllowed ()**

Returns true if short trades are allowed

4.11 Source code example

The following shows the full source code in which the functionality of two crossing moving averages is implemented.

```
//+------------------------------------------------------------------+
//|                                              CrossingMA_01.mq4 |
//|                                    Copyright 2017 by Dirk Hilger |
//|                                            https://www.mql5.com |
//+------------------------------------------------------------------+
#property copyright "Copyright 2017, Dirk Hilger"
#property link      "https://www.stereotrader.net"
#property version   "1.00"
#property strict

/*

   DESCRIPTION:

   CrossingMA buys or sells when one moving average crosses another.
   It can furhtermore test the curve of such if its sloping up or down.

   The example also shows the usage of an input panel.

*/

//+------------------------------------------------------------------+
//| Includes                                                         |
//+------------------------------------------------------------------+
#include <StereoTrader_API\StereoAPI.mqh>

//--- Remove these lines if indicator shall not appear in a separate window
#property indicator_separate_window
#property indicator_height 48

//+------------------------------------------------------------------+
//| Declaration of StereoEA                                          |
//+------------------------------------------------------------------+
DECLARE_SEA_BEGIN("CrossingMA01")
//+------------------------------------------------------------------+
//| Fields                                                           |
//+------------------------------------------------------------------+
CSNumEdit      ma_fast;            // Fast MA
CSNumEdit      ma_slow;            // Slow MA
CSButton       ma_checkcurve;      // Check curve direction

//+------------------------------------------------------------------+
//| Optional initialization function                                 |
//+------------------------------------------------------------------+
SEA_INIT
  {
      //--- Add input fields
      _Panel.AddNumEdit(ma_fast,"Fast MA",20);
      _Panel.AddNumEdit(ma_slow,"Slow MA",200);
      _Panel.AddButton(ma_checkcurve,"Curve");
      //--- Return success
      return(true);
  }
//+------------------------------------------------------------------+
//| Iteration                                                        |
//+------------------------------------------------------------------+
SEA_EVALUATE
   {

//--- MA levels
      double mafast=_Bars.SMA((int)ma_fast.Value());
      double mafastspan=_Bars.SMA((int)ma_fast.Value(),2);
      double maslow=_Bars.SMA((int)ma_slow.Value());
      double maslowspan=_Bars.SMA((int)ma_slow.Value(),2);

//--- Long signal
      bool buysignal=true;
      //--- Fast MA is above/equal slow MA
      buysignal&=mafast>=maslow;
      //--- Fast MA was below __e_ma_slow MA
```

```
      buysignal&=mafastspan<maslowspan;
      //--- Direction check
      if (ma_checkcurve.Pressed())
         {
         buysignal&=mafast>mafastspan;
         buysignal&=maslow>maslowspan;
         }

      if (buysignal)
         {
         Buy();
         return;
         }

//--- Short signal ?
      bool shortsignal=true;
      //--- Fast MA is below/equal __e_ma_slow MA
      shortsignal&=mafast<=maslow;
      //--- Fast MA was above __e_ma_slow MA
      shortsignal&=mafastspan>maslowspan;
      //--- Direction check
      if (ma_checkcurve.Pressed())
         {
         shortsignal&=maslow<maslowspan;
         shortsignal&=mafast<mafastspan;
         }

      if (shortsignal)
         {
         Sell();
         return;
         }

//--- No signal
   return;

   }


//+------------------------------------------------------------------+
//| Declaration of StereoEA                                          |
//+------------------------------------------------------------------+
DECLARE_SEA_END
```

5. Drawing

StereoTrader allows for easy drawing of clusters, highlight zones, arrows and lines. The drawing functions may be used within SEA_INDICATE as well as within SEA_EVALUATE

This chapter is not implemented yet, please refer to the samples such as RangeBreakOut or DstFromTime. The functions are:

```
DrawLine(string text, double pricebegin, color clr, datetime timebegin=0, datetime timeend=0, int width=1, ENUM_LINE_STYLE style=STYLE_SOLID, string tooltip=NULL)

DrawLineFibo(string text, double low, double high, double fibomult,datetime timebegin, datetime timeend, color clr, int width=1,ENUM_LINE_STYLE style=STYLE_SOLID,string tooltip=NULL)

DrawLineTrend (string text, datetime, datetime timebegin, double pricebegin, datetime timeend, double priceend, color clr, int width=1,ENUM_LINE_STYLE style=STYLE_SOLID,string tooltip=NULL)

bool DrawClusterLong(string text, datetime timebegin, double pricebegin, datetime timeend, double priceend, string tooltip=NULL) ;

bool DrawClusterShort(string text, datetime timebegin, double pricebegin, datetime timeend, double priceend, string tooltip=NULL) ;

bool DrawCluster(string text, datetime timebegin, double pricebegin, datetime timeend, double priceend, color clr_cluster, color clrtext, string tooltip=NULL)

bool DrawArrowLong(double price, string text=NULL, int size=2, int shift=0)

bool DrawArrowShort(double price, string text=NULL, int size=2, int shift=0)

DrawArrow(int filter, datetime timebegin, double pricebegin, string text=NULL, int size=3)

bool DrawHighlightLong(int shift=0)

bool DrawHighlightShort(int shift=0)

bool DrawHighlightLong(datetime timebegin)

bool DrawHighlightShort(datetime timebegin)

bool DrawHighlight(color clr, int shift=0)

bool DrawHighlight(datetime timebegin, color clr)

bool DrawFilterLong(int shift=0)

bool DrawFilterShort(int shift=0)

bool DrawFilterNone(int shift=0)

bool DrawFilter(int filter, datetime timebegin=0, datetime timeend=0, int shift=0) ;

bool DrawTraceLineInit(int id=-1, color clr=clrBlue, int width=1, ENUM_LINE_STYLE style=STYLE_SOLID, string tooltip=NULL)

bool DrawTraceLineBegin(int id, double price, color clr=clrNONE, int shift=0)

bool DrawTraceLineBegin(int id, double price, datetime timebegin, color clr=clrNONE)

bool DrawTraceLineUpdate(int id, double price, color clr=clrNONE, int shift=0)

bool DrawTraceLineUpdate(int id, double price, datetime time, color clr=clrNONE)
```

6. Dialog fields

StereoTrader allows for easy implementing of dialog fields which are located on a panel. The object which allows access to all such fields is defined as **_Panel**.

(This chapter is not implemented yet.)

```
//+------------------------------------------------------------------+
//|                                            VolumeBreakout.mq4 |
//|                                   Copyright 2016, Dirk Hilger |
//|                                  https://www.stereotrader.net |
//+------------------------------------------------------------------+
#property copyright "Copyright 2015, Dirk Hilger"
#property link      "https://www.stereotrader.net"
#property version   "1.00"
#property strict

//+---------------------------- -------------------------------------+
//| Indicator properties                                             |
//+------------------------------------------------------------------+
#property indicator_separate_window
#property indicator_height 48
//+------------------------------------------------------------------+
//| Includes                                                         |
//+------------------------------------------------------------------+
#include <StereoTrader_API\StereoAPI.mqh>
//+------------------------------------------------------------------+
//| StereoEA declaration                                             |
//+------------------------------------------------------------------+
DECLARE_SEA_BEGIN("VolBreakOut")
//+------------------------------------------------------------------+
//| Objects                                                          |
//+------------------------------------------------------------------+
CSButton          m_mode;        //--- Intrabar / prev bar
CSNumEdit         m_lookback;    //--- Edit field shift
CSNumEdit         m_mult;        //--- Edit field span
CSNumEdit         m_minpips;     //--- Edit field Slow MA

//+------------------------------------------------------------------+
//| Initialization function                                          |
//+------------------------------------------------------------------+
SEA_INIT
  {
      //--- Settings

      // We start operating on the previous bar
      SetDefaultShift(1);
      //--- Allow multiple trades signals same direction
      SetSignalOverlap(false);
      //--- Close opposite in case of countersignal
      SetSignalOpposite(SEA_OPPOSITE_CLOSE);

//--- Add panel title
      _Panel.AddTitle(m_name);
//--- Add edit fields
      _Panel.AddButton(m_mode,"Mode","Close","Tick",false);
      _Panel.AddNumEdit(m_lookback,"Vol Span",14);
      _Panel.AddNumEdit(m_mult,"Vol Th",2.0,1);
      _Panel.AddNumEdit(m_minpips,"Body pips",10);

//---
   return(true);
  }
//+------------------------------------------------------------------+
//| Check signal                                                     |
//+------------------------------------------------------------------+
SEA_EVALUATE
   {
      static long averagevolume=0;
      static int span=0;
```

```
      static double mult=0;
      static double minbarrange=0;

      //--- Calculate on new bar and fetch fields on new bar (saves time)
      if (IsNewBar())
         {
         //--- Fields
         span=MAX(1,(int)m_lookback.Value());
         mult=mult=m_mult.Value();
         minbarrange = PointsToPrice(m_minpips.Value());

         if (m_mode.Pressed())
            {
            SetEveryTick(true);
            averagevolume=_Bars.TickVolumeAverage(span,1);
            }
         else
            {
            SetEveryTick(false);
            averagevolume=_Bars.TickVolumeAverage(span);
            }

         //--- Average volume
         }

      //--- Checks
      if (averagevolume==0)
         {
         Print("No volume info present! Bar: ",_Index);
         return;
         }

      if (_Bars.BodyRange()<minbarrange)
         return;

      //--- Check volume
      long volThis=_Bars.TickVolume();
      if (volThis>=(averagevolume*mult))
         {
         if (_Bars.IsBullish())
            Buy();
         else
            Sell();
         }
   }


DECLARE_SEA_END
```

7. Notes for advanced developers

7.1 Class frame

Any code between DECLARE_SEA_BEGIN and DECLARE_SEA_END is part of a class which derives from the base classes *CStereoEA* and *CStereoEAEvents*, the macros define the frame to embed any function into the class CSEADev. In case if you used earlier versions of the API, the code is still compatible. The name of the public CStereoEAEvents class object is __SEA.

- SEA_INIT is replaced by virtual bool OnInit()
- SEA_EVALUATE is a macro which is replaced by virtual void OnEvaluate()
- SEA_INDICATE is a macro which is replaced by virtual void OnIndicate()
- SEA_DEINIT is replaced by virtual void OnDeinit(const int reason)
- SEA_CHARTEVENT by virtual void OnChartEvent(…)
- SEA_CALCULATE by virtual void OnCalculate(…).

Instead of using these macros, you may feel free to use the normal function names.

8. Further classes & functions of the API

The functions described before are local functions of the SEA framework, inside the class CSEADev, defined by DECLARE_SEA_BEGIN and DECLARE_SEA_END. The API contains much more functions and classes which may be used by an SEA.

8.1 Compatibility MT4/MT5

It´s highly recommended to **NOT** use native MetaTrader functions, rather you should use only the functions of the API files instead. The reason is the compatibility between different builds of MetaTrader as well as compatiblity between MT4 and MT5. When using these functions, you are always 100% compatible. If you take a look at the forums of MetaQuotes, you will find hundreds of desperate users who worry about porting from MT4 to MT5. If you decide to develop with the framework/API of StereoTrader, you don´t have to worry about it all – you are always compatible without any change.

8.2 File __MT_native.mqh

This file contains public functions for the most common reasons. The functions replace many native MT language commands, it also contains useful macros. Almost every function name begins with the prefix "__MT_". Please feel also free to take a look at the file and to find additional functions for your purposes.

The file also contains several public objects. These are

- **__Terminal** by CMTTerminal class – properties and functions regarding the environment
- **__Account** by CMTAccount class – properties and functions regarding the account and broker
- **__Time** by CTimeNative class – access to time functions

8.3 File __ChartExt.mqh

This file contains the class CChartExt as well as a public object named

__Chart

The object allows for quick access and enhanced functionality of/with the chart.

8.4 File XVars.mqh

This file contains class which allow for extended variable sharing between different charts and instances of MT4 and MT5. It replaces and enhances the functionality of global variables, which you may know from MQL. Besides the class CXVars there are several predefined public objects:

- **__ChartVars** – allows for sharing variables inside one chart and may be accessed by multiple SEAs
- **__LocalVars** – allows for sharing variables inside the MT4 or MT5 environment and between different charts
- **__GlobalVars** – variable sharing between multiple instances of either MT4 or MT5
- **__GlobalMTXVars** – variable sharing between all instances of MetaTrader

These objects can be accessed without any initialization or deinitialization.

The most important functions are:

```
bool   SetDouble(string name, double value, bool permanent=false)
```

Stores a double value by a given name. If permanent is set to true, the variable will overdue the runtime of an SEA.

```
bool   SetString(string name, string value, bool permanent=false)
```

Stores a string value by a given name. If permanent is set to true, the variable will overdue the runtime of an SEA.

```
double  GetDouble(string name)
```

Reads a double value by a given name.

```
string  GetString(string name)
```

Reads a string value by a given name.

```
bool   IsDefined(string name)
```

Returns true if the variable is defined

```
bool   Delete(string name)
```

Reads a string value by a given name.

If you want to use the class for furhter purposes, these functions allow to create own variable databases:

```
bool   InitByChart(string prefix=NULL, CHART id=NULL, bool commonfolder=false)
```

Creates or opens a database by using the current chart.

```
bool   Init(string filename, bool commonfolder=false)
```

Creates or opens an individual database.

8.5 File Comment.mqh

You may know the MQL command "Comment". The CComment class enhances the functionality of printing text in the upper left corner of the chart. Comments may displayed in separate blocks, whereby each inheritant/instance defines a block. You may define your own instance or you may use the predefined public object **__Comment .**

Please note that SEAs should not use the class object, rather you should use the function **MessageChart().** This way you ensure that StereoTrader handles the output and manages it with outputs of other SEAs.

```
void   Append(string value)
```

Adds a text to the comment block

```
void   Clear()
```

Deletes all comments

Copyright / Impressum

StereoTrader is developed under german copyright protection by

Dirk Hilger
Gottesweg 64

50969 Cologne
Germany

info@stereotrader.net
www.stereotrader.net

All rights are reserved. Usage of StereoTrader is always on your own risk.

Get ready for success.