



NanoTrader Express

Language Reference

Document Version 2.4.0

www.fipertec.com

Contents

1	Introduction	4
2	Express – Execute Only	4
3	Structure of an Express-Program.....	5
4	A Sentimentor Example.....	5
5	Adding an Express-Sentimentor to a Study.....	7
6	Working with the Express Editor	10
6.1	Open the Express Editor	10
6.2	Keyboard Shortcuts.....	10
6.3	Verifying an Express Program	10
6.4	Understanding how Express Programs are Saved	10
7	Express Language Elements	11
7.1	Types	11
7.2	Reserved Words	12
7.3	Expressions	12
7.3.1	Numerical Expressions	12
7.3.2	Relational Expressions	12
7.3.3	Logical Expressions	13
7.3.4	String Expressions	13
7.4	Variable Declarations	13
7.5	Input Variables	16
7.6	Accessing Variables and Series data.....	17
7.7	Working with Arrays	18
7.8	Assignments	18
7.9	Assigning Sentiments.....	18
7.10	Predefined Series.....	19
7.11	Importing a Series from another Sentimentor	19
7.12	Importing Price Data from another Symbol	21
7.13	Date and Time constants	22
7.14	Statements.....	22
7.15	Control Structures	23
7.15.1	if then	23
7.15.2	If then else	23
7.15.3	While Loop.....	23
7.15.4	For Loop	24

7.16	The Need for Speed.....	25
7.17	Interpretation – Computing the Sentiments.....	25
7.17.1	Interpretation Using the Built-in Schemes.....	26
7.17.2	Programming the Interpretation Explicitly	26
7.18	Plotting.....	27
8	A Blocker Example.....	28
9	A Stop Example	29
10	A Stop/Tactic Example with intraperiod updates	30
11	Encrypting Express-Sentimentors	32
12	Tips for Debugging	33
13	Built-In Functions and Procedures.....	34

1 Introduction

NanoTrader-Express allows to program sentimentors, stops and filters that can be used in exactly the same way as the built-in sentimentors, i.e., they can be combined with other sentimentors and of course they can be optimized. Thus, the NanoTrader framework in conjunction with the Express environment gives you an unparalleled power for specifying, optimizing, backtesting, and applying your trading ideas.

Note that it is not a prerequisite to have the NanoTrader-TradingSystem permission to take advantage of Express. You might use Express to compute and plot classical indicators as well as creating graphical annotations to the chart or issuing messages and alarms.

The scope of this document is to provide a description of the language NanoTrader-Express. It is not intended to give an introduction into the theory and practice of programming or algorithms in general. A reader unfamiliar with the concepts of programming may have a look at introductory books for, e.g., Visual Basic, Pascal, Excel programming, or EasyLanguage for TradeStation. Once the main concepts like *variables*, *loops*, or *conditional expressions* are understood, working with Express will be very easy.

Besides a working knowledge of programming languages it is assumed that the reader is familiar with the terminology used in the “NanoTrader – Charting & Trading” manual.

Users having experience with other programming languages for building indicators of trading systems should be aware of the overall “Sentimentor” approach used by NanoTrader as this carries over to sentimentors programmed in *Express*, i.e., you will not find statements like “buy at open” – instead, an Express based Sentimentor is a building block used generating sentiments that eventually lead to trading actions through the combination with the MetaSentimentor and the applied trading approach.

2 Express – Execute Only

Access to the programming environment of Express requires a specific permission. However, NanoTrader allows users *without* the Express permission to *execute* and *view* Express scripts, but they cannot change or create scripts on their own.

In order to enable an Express script (or a study containing an Express script) to be executed by an "ExecuteOnly" user that script needs to be opened once with the Express editor by a user having the Express permission. When closing/saving the editor NanoTrader silently adds a watermark to the script which is required for being executable by "ExecuteOnly" users.

We encourage programmers to spread their Express creations among the NanoTrader users.

3 Structure of an Express-Program

The way a Sentimentor is computed is as follows:

- Declare and initialize variables needed for the computation
- For each bar, carry out the required calculations.
- Compute the sentiments, i.e., how is the result of the calculation to be interpreted
- Define one or more charts to be plotted

An Express program reflects this by enforcing the following structure:

Variable Declarations Section

Calculation Section

Interpretation Section

Plot Section

4 A Sentimentor Example

Let's have a look at a simple Sentimentor that computes an exponential moving average (EMA). Buy and Sell sentiments are generated if the EMA crosses the close price. This Sentimentor is part of the NanoTrader distribution.

(Hint: This code is just an example to show the exact syntax of Express)

<code>//(c) Fipertec</code>	1.
<code>Express EMA</code>	2.
<code>Vars</code>	3.
<code>input \$span (1, 200, 10);</code>	4.
<code>numeric factor (0);</code>	5.
<code>series ema;</code>	6.
<code>Calculation</code>	7.
<code>factor = 2 / (\$span + 1);</code>	8.
<code>if close[1] = void then //we need one lookback entry</code>	9.
<code> ema = close;</code>	10.
<code>else</code>	
<code> ema = factor * close + (1 - factor) * ema[1];</code>	11.
<code>interpretation TriggerLine(close, ema);</code>	12.
<code>plot (ema, "blue", 2);</code>	13.
<code>plot (close, "black", 1);</code>	14.

Explanation:

1. A double slash is used to start a comment reaching until the end of the line. It is also possible to use curly braces { } for comments anywhere in the code.

You can use this function to deactivate specific code lines.

2. The program always starts with `Express <name>`, where `name` becomes the name of the Sentimentor used in the DesignerBar.
3. With the keyword `Vars` the Variable Declarations Section is started.
4. This line declares an integer variable that is subject to optimization. The minimal value is 1, the maximal value is 200, and the initial value is 10. The name of the variable, `span`, is displayed in the DesignerBar. By using the `$` character, it is immediately visible in the code where optimization variables are used.

Now the word „span“ is automatically shown in the workspace bar as a variable and you can change it's name into a preferred name. The „\$“ figure in front of the variable name means that this is an input variable.

5. `factor` is defined as a numeric variable that may hold integer or float values. By using `(0)` it is initialized to zero. However, as `Express` guarantees to initialize numeric values to zero, the `(0)` could be omitted. `Express` is not case sensitive, i.e. `factor`, `Factor`, `FACTOR` all refer to the same variable. The same holds for keywords.
6. `ema` is defined as a `series` of float data. The series has the same length as the analyzed `MasterChart`. The elements of the series are automatically initialized to zero. By using the `(val)` mechanism, `val` would be used as the initial value of all the elements of the series.
7. After the reserved word `Calculation` the statements for performing the computations begin. These statements are executed for each bar in turn, starting with the oldest or “left most”.
8. The `factor` for the EMA computation is determined, based on the input variable `$span`. The usual operators `(+, -, *, /)` and parenthesis can be used for mathematical operations.
9. `Express` defines a number of series implicitly, e.g., `close`, `open`, `high`, `low`, `volume`, to access the data of the `MasterChart`. For computing the EMA value of the actual period the EMA value of the previous period is also required. To access previous data of a series, an indexing is used: `close[1]` denotes the close of 1 one period ago. Generally, `close[n]` denotes the closing price of `n` periods ago, and `close` is simply a synonym for `close[0]`. Assume we are calculating the EMA for the very first period then `close[1]` will not be available. In this case, the value of `close[1]` will

be `void`, a reserved word that is used to identify non existing data.
 The conditional statement `if` executes the then-part if the condition is fulfilled. In case the then-part consists of more than one statement, the then-part has to be started with `begin` and ended with `end`.

10. The first `ema` series element is assigned an initial value.
11. The `ema` for the current period is computed.
12. The conversion of the `ema/close` crossings into sentiments– called the *interpretation* – can be carried out by the standard interpreter *TriggerLine*. It would also be possible to compute the sentiments explicitly by assigning sentiment values to the predefined series `sentiment`.

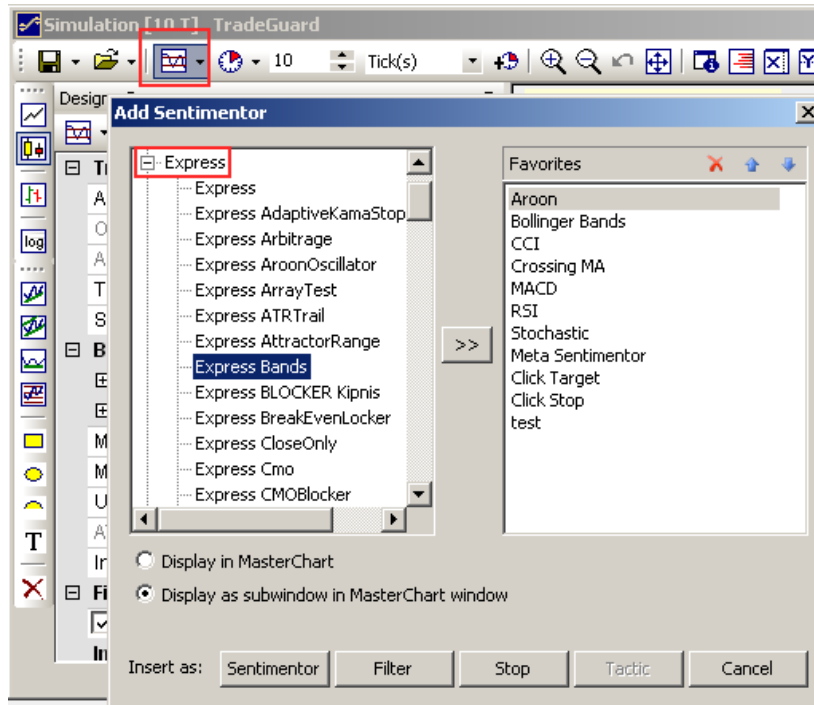
In case an interpretation scheme isn't used it is possible to create the sentiment value with individual code in the interpretation section. In this case the sentiment values need to be declared to the predefined series variable „`sentiment`“ and all code must begin with the words „`begin`“ and „`end`“. Example:

```
interpretation
begin
  if ... then
    sentiment = 100;
  else
    ...
end
```

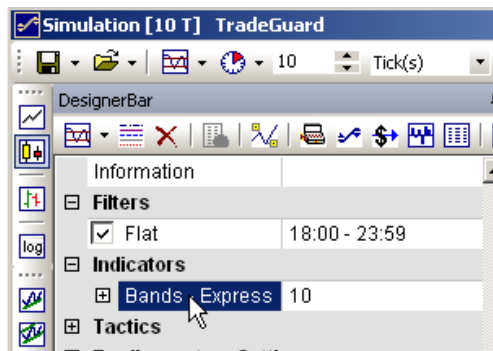
13. The `ema` series is to be plotted in blue using a pen width of 2.
14. The `close` series is to be plotted in black using a pen width of 1.


5 Adding an Express-Sentimentor to a Study

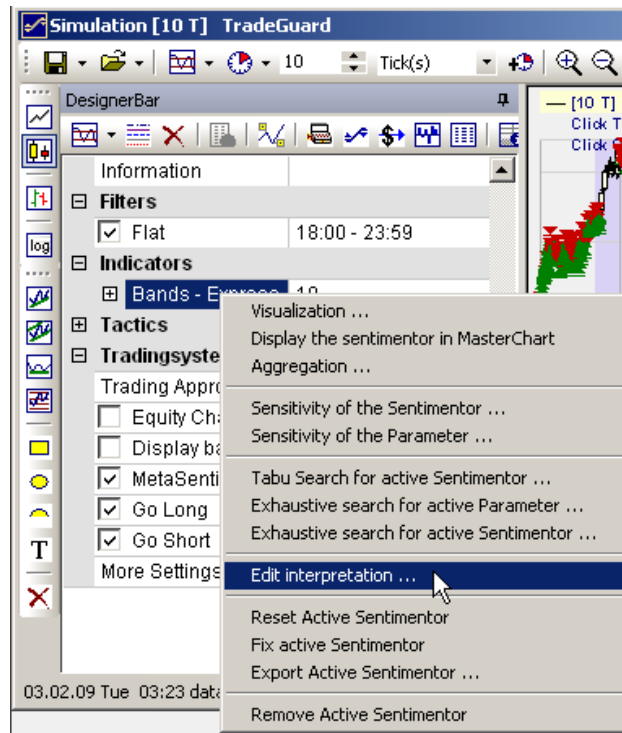
To add an Express Sentimentor to a study, choose the desired Sentimentor from the Express section of the Add Sentimentor dialog:



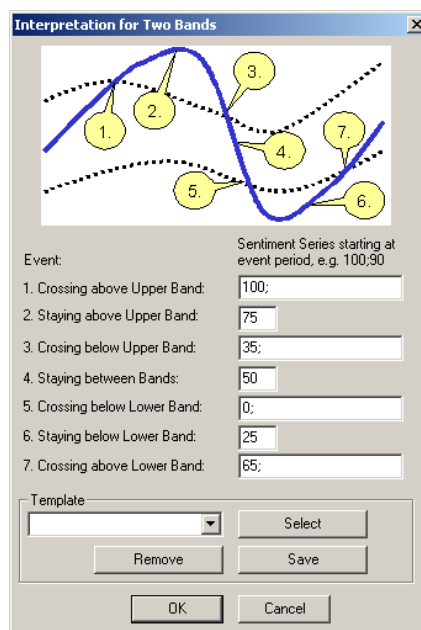
To edit the code of an Express Sentimentor double click the corresponding line in the DesignerBar:



In case the Express Sentimentor applies a default interpretation scheme (see below), the scheme can be configured by clicking the  icon or by rightclicking the Express Sentimentor in the DesignerBar and then choosing Edit interpretation from the context menu:



This will bring up the associated sentiment editor:

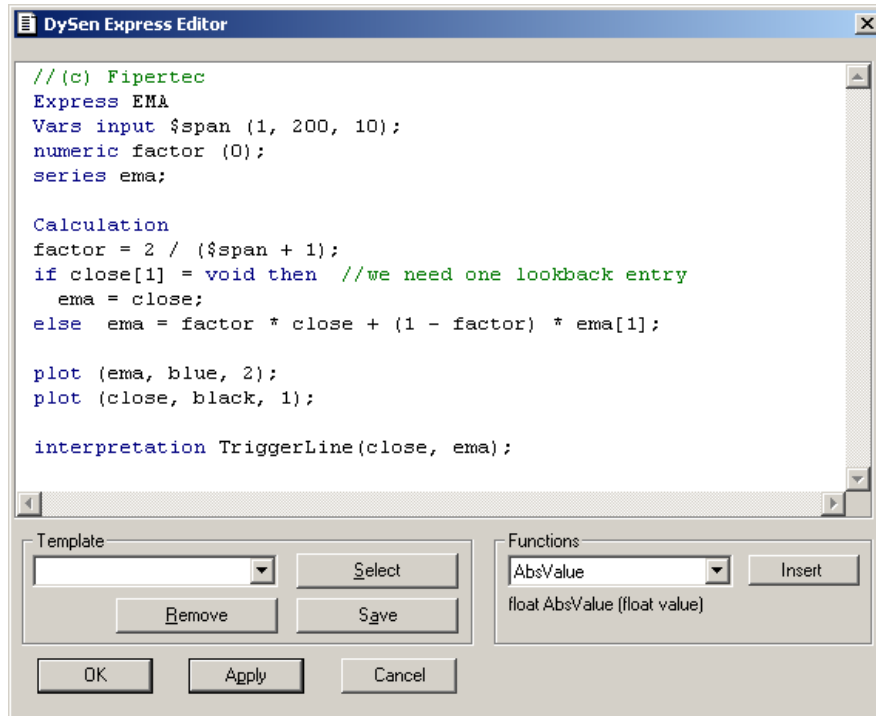


More details about the interpretation scheme can be found in the document „trading systems „.

6 Working with the Express Editor

6.1 Open the Express Editor

Double clicking on an Express Sentimentor in the DesignerBar will open the Express editor:



The editor displays the reserved words of Express in blue and comments in green.

6.2 Keyboard Shortcuts

The default Windows shortcuts for cutting and pasting text can be applied, i.e., Ctrl-C to copy selected text, Ctrl-V to paste text, Ctrl-A to select the complete text, and Ctrl-S to save the text.


The Express editor also provides unlimited undo/redo using the standard Windows shortcuts Ctrl-Z for undo and Ctrl-Y for redo.

6.3 Verifying an Express Program

Clicking the Apply-button will execute the Express program and the results in terms of calculated series, plots, and signals are immediately displayed in the charts. In case the code contains errors, NanoTrader will display appropriate error messages.

6.4 Understanding how Express Programs are Saved

When quitting the Express editor by clicking the OK-Button, the program is associated with the Express Sentimentor that has been double clicked to start

the editor. Note that you need to save the study explicitly using  from the chart window otherwise the program might be lost.

However, the usual way is to save the Express program as a template before quitting the Express editor. This will save the programs text in the directory „Express“ below the installation directory of NanoTrader. This will also make the program available as a Sentimentor in the Add Sentimentor-dialog and the Editor window.

From within the Editor, you may load any template and adapt it to your specific needs.

It is good style to use the name of the Express program when saving it as a template. Suppose the program starts with
`Express UltimateSenti`
 then `UltimateSenti` is the name of the Sentimentor. This name appears in the DesignerBar and in the legend of its chart window. When saving the program as a template, it is desirable to use `UltimateSenti` as the templates name, although this is not enforced by the editor.

7 Express Language Elements

7.1 Types

Express is a so-called *typed language*, i.e. every *entity* representing a value is of a certain type. This allows NanoTrader to catch a broad range of potential programming errors immediately.

The following types are supported by Express:

- `numeric`
 A numeric value can be a float (e.g. 3.75) or an integer (e.g. 10). If needed, *NanoTrader* automatically converts floats into integer values by stripping the fractional portion.
- `series`
 A `series` is a series of elements of type `numeric`. Suppose the analyzed MasterChart consists of 200 bars then a series will also automatically consist of 200 elements. If the MasterChart is connected to a realtime data feed, then a `series` will grow automatically.
- `array`
 An array is a specified number of elements of type `numeric`. The size of an `array` *does not change automatically* as a series does. The entries of an `array` are initialized to 0.
- `string`
 A string is a sequence of characters, like `Hello World!` When using string constants in Express, the characters have to be enclosed in quotation marks: `"Hello World!"`
- `bool`
 A boolean entity can take the values `true` or `false`.

- `time`
 Entities of type `time` are used to work with times and dates, e.g.,

```
if timeOpen < 10:00 then sentiment = 50;
if date = 2_8_2002 then sentiment = 100;
```

7.2 Reserved Words

Express uses a number of words that have a specific meaning. These words cannot be used for naming variables.

Currently the reserved words are:

```
express, sentimentor, blocker, stop, vars,
calculation, interpretation, numeric, bool, info, export,
senti_block, senti_flat, senti_pass, series, string, if,
then, else, begin, end, for, to, downto, while, void, and,
or, plot, plotband, plotcrossinglines, plotline,
plotcandles, plotbars.
```

The following reserved words are for future use:

```
function, procedure, import, plothistogram
```

7.3 Expressions

An expression is a combination of *operators* and *operands*, like

```
5 + 3
```

where 5 and 3 are the operands and + is the operator.

Moreover, the value of a variable, a series, or the return value of a function are also expressions.

Express distinguishes between three types of expressions: numerical, string, and boolean.

7.3.1 Numerical Expressions

Numeric entities can be combined using the operators +, -, *, / with their usual mathematical meaning, i.e.,

```
5 * close - 3*close[1];
```

Parenthesis can be used to group expressions as in

```
(high + low + close) / 3
```

7.3.2 Relational Expressions

A relational expression evaluates to `true` or `false`, as in
`close > open`.

The available relational operators are

Operator	Meaning
-----------------	----------------

>	greater than
<	less than
=	equal to
>=	greater equal than
<=	less equal than
<>	not equal to

Relational operations may be used with numeric, time, and string entities. In the latter case the relational operation is evaluated with respect to the *lexicographical ordering*, i.e. "abc" is less than "xyz".

7.3.3 Logical Expressions

A logical expression is a combination of expressions evaluating to `true` or `false` with the operators `and` and `or`, e.g.,

```
(close > open) and (volume <= 1000)
```

```
(close > close[1]) and ((open > close) or (volume > 1000))
```

Note: Always use parenthesis to indicate exactly the grouping of the expressions. This enhances the readability of the code and avoids unnecessary programming errors.

7.3.4 String Expressions

The only operator for two entities of type string is `+` that is used for concatenating the strings, e.g.,

```
"Hello "+ "World!"
```

results in the new string "Hello World!".

7.4 Variable Declarations

All variables used by an Express program have to be declared in the `Variable Declarations` Section by stating the type and the name of the variable.

The name must begin with a character, e.g.,

```
numeric weight;
```

The initial value of a variable may be given in its declaration:

```
numeric weight(0.5);
```

To specify some variables of the same type the following notation can be used:

```
numeric weight, factor, delta;
```

Hint: Choose names for the variables that explain their meaning to make the code easier to understand later.

Valid variable types are `numeric`, `bool`, `series`, and `string`. In case the initial values are not specified in the declaration, Express uses the following values:

Type	Initialized to
<code>numeric</code>	0
<code>bool</code>	<code>false</code>
<code>series</code>	all elements set to 0
<code>array</code>	all elements set to 0
<code>string</code>	<code>""</code> , i.e., the empty string

A `series` variable can be bound to a `series` of another Express sentimentor or to a `series` of a built-in sentimentor that is part of the study. See Section [Importing a series from another sentimentor](#) for details.

Exporting Variables

`numeric` and `string` variables can have an optional `export` clause:

```
export "label;format";
```

Example:

```
numeric result export "Trending Days;%.02f";
string comment export "Comment";
```

The `“;format”`-portion is optional. (See function `NumericToString()` for a description of valid format strings.)

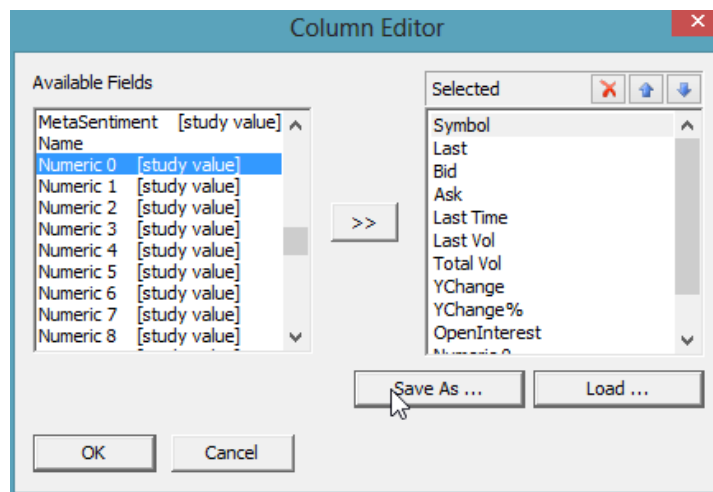
Exported variables are shown as separate columns in the result table of Screeners:

Screeners			
Name	Trending Days	Comment	
Carrefour	11.00	strong trend	
Bank of America	5.00	weak trend	

The exported variables are also shown in the InfoBar of the study:

Item	Value		
Time from	12:30:38		
Date from	06.02.15	Fri	
Time to	12:30:38		
Date to	06.02.15	Fri	
Open	138.88		
Close	138.88		
High	138.88		
Low	138.88		
Volume	0.00		
Periods	79		
Periods in Zoom	78		
range	0.00 %		
avg. range	0.06 %		
Average True Range	0.06 %		
Trending Days	5		
Comment	weak trend		

The configuration dialog for Scanners (LiveTables) allows to show variables exported from Express scripts. As a LiveTable can hold various studies the column headers can only be generic. Up to ten exported Numeric/Series variables and ten exported Strings can be shown:



Symbol	Study	Last	Bid	Ask	Last Time	Last Vol	Numeric 0	String 0
Simulation	MACD [5 Min.]	5944.0	5943.5	5944.5	15:01:20	62	n/a	n/a
BUNDsim	Plain [3333 Se...]	139.44	139.43	139.45	15:01:20	0	5	weak trend
Simulation	Plain [5 Min.]	5944.0	5943.5	5944.5	15:01:20	62	5906.22	Up

7.5 Input Variables

Input variables are numeric variables that are exposed to the outer world. They appear as parameters in the DesignerBar and can be altered by the end user, just as any parameter of the built-in sentimentors. Moreover, the input variables are subject to optimization, so they need a minimal, a maximal, and an initial value. The declaration of an input variable is as follows:

```
input $<var name> (<min value>, <max value>, <initial value>);
```

Example:

```
input $span (1, 200, 20);
```

The \$ sign is used as prefix to indicate anywhere in the code that the referenced value is an input variable that is subject to optimization and hence is of great importance for the overall computation.

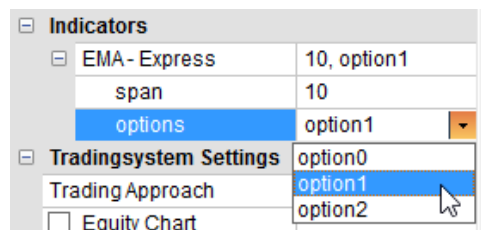
A numeric variable can be defined to show a list of text options in the DesignerBar using the following syntax:

```
input $<var name> (<"semicolon separated options", <initial value>);
```

Example:

```
input $options ("option0;option1;option2", 1);
```

This will be shown in the DesignerBar as follows:



Internally the options are numbered starting with 0. The above example is equivalent to the following definition without using option strings:

```
input $options (0, 2, 1);
```

Floating Point Variables

Very often, the input variables will only take integer values. However, sometimes you may need float values as input variables. For defining input variables of type float two more specifications are required: the precision and the step size. The latter is used by the optimization as the minimal change of the variables value. The declaration of an input variable holding float values is:

```
input $<var name> (<min value>, <max value>, <initial value>, <step size>, <precision>,);
```


Example:

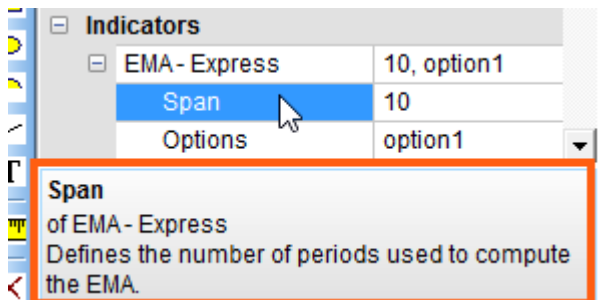
```
input $factor (1.00, 3.00, 2.00, 0.01, 2);
```

Showing Explanations

An input variable can have an optional `info` clause. E.g.:

```
input $span (1, 100, 20) info "Defines the number of
periods used to compute the EMA.";
```

When clicking the parameter in the DesignerBar, the text is displayed in the Description Area of the DesignerBar:



This allows to have a short description of each input parameter so the user does not need to refer to some external documentation.

7.6 Accessing Variables and Series data

The value of a variable is accessed by simply using the variables name, as in `factor * delta`

where `factor` and `delta` are declared as numeric variables.

An element of a variable of type `series` is always referenced *relative* to the currently processed bar. The syntax is as follows:

```
<series name> [n]
```

where `n` denotes a positive integer or an expression.

Example:

```
close[1] or close[span], where span is a numeric variable.
```

The example references the close price of the previous bar (1 bar ago).

To access the current bar, it is possible to use the abbreviation `close` instead of `close[0]`.

7.7 Working with Arrays

An array has a fixed number of entries of type `numeric`.

An array can be defined in the Variable Declarations Section as follows:

```
Array a[100];
```

This defines an `array` containing 100 numeric entries.

There is a major difference in accessing `array` entries as opposed to `series` entries, although the syntax is the same:

```
<array name>[n]
```

Eg. `a[0]` refers to the first entry of the array.

Note that the indexing starts with 0, i.e., valid indices for the array `a[100]` are 0 to 99.

Also note the difference to working with `series` in that the index always represents an absolute index inside the `array`, not a relative offset with respect to the current bar index.

An array might be resized using the function `SetArraySize()`, e.g.

```
SetArraySize(a, 250);
```

All entries of an array might be set to a specific value using the function

```
SetArrayTo(), e.g.,
```

```
SetArrayTo(a, 500);
```

7.8 Assignments

A declared variable can be assigned a value using the `=` operator:

```
span = high - low;
```

When assigning a value to a variable of type `series`, Express automatically uses the element currently processed in the calculation section: Suppose `median` is a variable of type `series` and Express performs the calculation for the 25th bar of the MasterChart, then

```
median = (high + low + close) / 3;
```

assigns the result of the expression to the 25th element of the `median` series.

7.9 Assigning Sentiments

The sentiments of a `Sentimentor` programmed in Express have to be stored in the predefined series `sentiment`. As a sentiment has to be a value between 0 and 100, Express automatically enforces this, i.e., if a value greater than 100 is assigned, Express uses 100 instead, and in case a negative value is assigned, Express uses 0.

When computing sentiments, it is sometimes very convenient to assign a sentiment not only for the current period but also for the next 3, say, periods.

This can be achieved using the following notation:

```
sentiment = [100; 90; 80;];
```

With this function the sentiment value is assigned to 100 for the 1st period, for the next period the sentiment will be 90 and for the next 80.

Note that this so-called *list assignment* is only valid for the predefined series `sentiment`.

7.10 Predefined Series

The following series referring to the MasterChart are always available:

`open`, `close`, `high`, `low`, `volume`. You may also use the abbreviations `o`, `c`, `h`, `l`, `v`.

For Renko charts the series `renkoHigh`, `renkoLow` are available to access the high and low of a brick.

The following series of type time are also available:

Series	Meaning
<code>date</code>	the date of the end of the period
<code>dateOpen</code>	the date of the beginning of the period
<code>time</code>	the time of the end of the period
<code>timeOpen</code>	the time of the beginning of the period
<code>dateTime</code>	the date and time of the end of the period
<code>dateTimeOpen</code>	the date and time of the beginning of the period

Finally, the series `sentiment` has to be used for storing the computed sentiments unless one of the default interpretation schemes is used.

With the exception of the sentiment series, all predefined series are *read only*, i.e., it is not possible to assign a value to them.

7.11 Importing a Series from another Sentimentor

A `series` variable can be bound to a `series` of another Express sentimentor or to a `series` of a built-in sentimentor that is part of the study by using the following syntax:

```
series myseries (sentimentorName.seriesName);
```

The variable `myseries` will be a read-only reference to `seriesName` of the sentimentor named `sentimentorName`.

This scheme is extremely helpful when using the computational result from a sentimentor creating signals in another sentimentor that is used as a stop.

The following example demonstrates this.

```
Express EMA_Mid
Vars
```

```

series emaHigh, emaLow, emaMid;
input $span (0, 200, 10);

Calculation
if IsFirstBar() then
begin
  ExpMovingAverage(high, emaHigh, $span);
  ExpMovingAverage(low, emaLow, $span);
end

emaMid = (emaHigh + emaLow) / 2;
interpretation TriggerLine(close, emaMid);
plot (emaMid, "blue", 2);

```

The EMA_Mid sentimentor creates signals based on a trigger price calculated from the EMAs off the period highs and lows.

Now assume a stop sentimentor should rely on exactly that trigger series. Instead of replicating the code and forcing the input parameters to be the same the stop could simply import the trigger series names `emaMid`.

An example would be as follows:

```

Express Stop EMA_MidStop
vars
input $ofs (-25, 25, 10);
series anchor (EMAMidExpress.emaMid);
//The stop trails along the imported series with $ofs
//ticks.
numeric last;
calculation
if MarketPosition() = 1 then
begin
  if IsIntradayEntry() then
    last = -999999;
    last = max (last, anchor - $ofs * TickSize());
    SetStopPrice(last);
  end
else
begin
  if IsIntradayEntry() then
    last = 999999;
    last = min (last, anchor + $ofs * TickSize());
    SetStopPrice (last);
  end
end

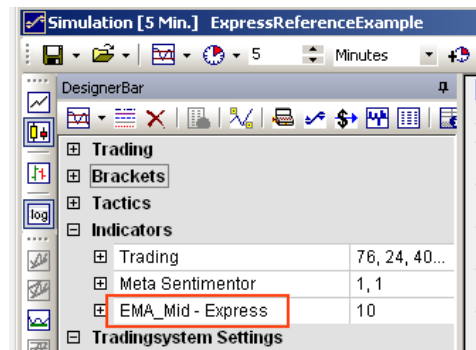
```

Naming Convention

The imported sentimentor has to be written as it is shown in the DesignerBar with dropping all non-alphabetical characters including spaces.

Therefore „EMA_Mid – Express“ as shown to the right will become EMAMidExpress.

The name is not case sensitive, hence EmamidExpress would also be feasible.



If the study contained multiple „EMA_Mid – Express“ sentimentors then they have to be indexed:

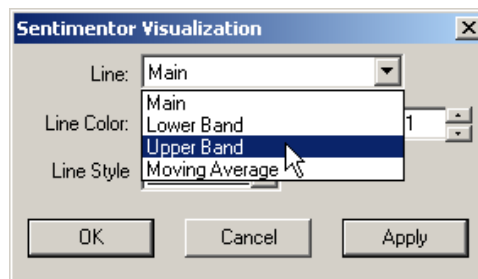
```
series anchor (EMAMidExpress2.emaMid);
```

Import from a Built-in Sentimentor

It is also possible to import a series from a built-in sentimentor, e.g.:

```
series myBollinger (BollingerBands.upperBand);
```

The series that can be imported are listed in the Sentimentor Visualization dialog. For Bollinger Bands this dialog looks as follows:



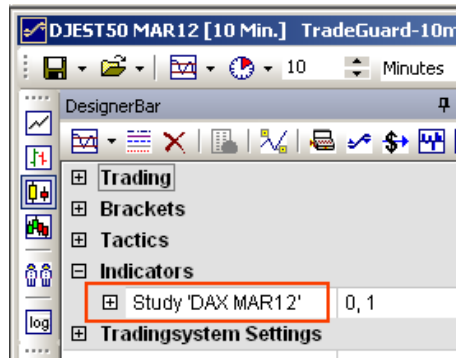
Calling Sequence

If sentimentor A imports a series from sentimentor B then A must be recalculated whenever the settings for B are changed. NanoTrader ensures the correct call sequence automatically. Moreover, there is no limit of the number of imports a sentimentor can have. Imports can be nested in any depths, e.g., A might import from B which imports from C and so forth.

7.12 Importing Price Data from another Symbol

As a special case of the above discussed importing of series from another sentimentor the price data of a symbol used by the Study sentimentor can be accessed. Recall that one usage of the Study sentimentor is to just display the price data of the accessed symbol.

Assume a Study sentimentor accesses the DAX future:



An Express sentimentor could access the price data as follows:

```
series dax (StudyDaxMar.close); //or .open, .high, .low
```

As Future contracts often carry their expiry date at the end of their name, hence forcing to adapt the code whenever the contract rolls over, it is possible to provide only the beginning sequence of symbol name, e.g.:

```
series dax (StudyDax.close);
```

7.13 Date and Time constants

Time constants may be used in boolean expressions. A time has the format HH:MM or HH:MM:SS

Example:

```
14:15    14:50:40
```

If the seconds are omitted they are automatically set to 0.

The format of a date is:

DD_MM_YYYY

Example:

```
22_10_2002
```

Time and date constants can be used in boolean expressions, as in

```
if time < 10:00 then
    sentiment = 50;           //don't buy during the first hour of the session
```

7.14 Statements

A statement is a complete Express instruction composed out of reserved words, operators, operands and ended by a semicolon. E.g.

```
delta = high - low;
```

7.15 Control Structures

7.15.1 if then

The if control structure is used to execute statements only if a specified condition is met. The syntax is:

```
if <boolean expression> then
    <statement>;
```

If several statements are to be executed the following syntax has to be used:

```
if <boolean expression> then
begin
    <statement>;
    <statement>;
    ...
    <statement>;
end
```

Example:

```
if close > open then
    upMoves = upMoves + 1;
```

7.15.2 If then else

The if control structure may also contain statements to be executed in case the condition is *not* met:

```
if <boolean expression> then
    <statement>;
else
    <statement>;
```

Again, use `begin` and `end` to group a number of statements to be executed.

Example:

```
if close > open then
    upMoves = upMoves + 1;
else
begin
    downMoves = downMoves + 1;
    downVol = downVol + volume;
end;
```

7.15.3 While Loop

The syntax of the While loop is as follows:

```
while <boolean expression>
begin
    <statement>;
    <statement>;
```

```
...
  <statement>;
end
```

The statements are executed until the <boolean expression> evaluates to true.

Example:

```
lastHigh = 1;
vol = 0;
while (close[lastHigh] <> void) and (close > close[lastHigh])
begin
  vol = vol + volume[lastHigh];
  lastHigh = lastHigh + 1;
end
```

Note: Double check that the <boolean expression> will finally evaluate to false, otherwise the while loop would run endlessly and the system will be blocked. There is no way for NanoTrader to verify the finiteness of a <boolean expression>. Therefore, if a While loop does not terminate within five seconds, the Express program is terminated by NanoTrader.

7.15.4 For Loop

The syntax of a `For` loop is as follows:

```
for <variable> = <start value> to <numerical expression>
begin
  <statement>;
  <statement>;
  ...
  <statement>;
end
```

At the beginning of the `For` loop, <variable> is set to the <start value>. If <variable> does not exceed <numerical expression> then the statements are executed and <variable> is increased by one. This process repeats until <variable> finally exceeds <numerical expression>.

Example:

```
upMoves = 0;
for i = 0 to 9
begin
  if close[i] > open[i] then
    upMoves = upMoves + 1;
end
```

Sometimes it is desirable to *decrease* the <variable> and to stop the loop if the <variable> falls below <numerical expression>. This can be achieved by using the following variant of the `For` loop:

```
for <variable> = <start value> downto <numerical expression>
begin
```



```

    <statement>;
    <statement>;
    ...
    <statement>;
  end

```

Note: Double check that the `<variable>` will finally exceed `<numerical expression>` (or falls below it in case of the `downTo` variant).

There is no way for NanoTrader to verify the finiteness of a For loop. Therefore, if a For loop does not terminate within five seconds, the Express program is terminated by NanoTrader.

7.16 The Need for Speed

Whenever `While` or `For` loops are used, make sure that the computation you are carrying out is *efficient*. It is very easy to implement a calculation in a naive way that works, but that requires an enormous amount of computation time.

Take for example the calculation of a 50-bar moving average. The naive approach would sum up the close price of the current and the previous 49 bars and then divide the result by 50. A more intelligent approach would take advantage of the fact that whenever moving to the next bar, the new sum could be computed by subtracting the “leftmost” price and adding the price of the current bar.

Hence, the naive approach is 50 times slower (*in words: fifty*) than the more intelligent approach. For a 200-bar moving average it would be 200 times slower. Now suppose what happens if you use the “naive” implementation within an optimization...

Quite often some calculations can be performed after all bars have been processed, e.g., you want to apply a moving average on a complete series you have computed. This can be achieved easily by using the boolean built-in function `IsFinalBar()`:

```

...
series result;
input $span (1, 200, 10);
Calculation:
..result = ...;
  if IsFinalBar() then //true, if currently the final bar is processed
    MovingAverage (result, result, $span); //built-in function

```

Whenever you assume that your Express script requires a lot of computation time, make sure to call `CalculateAtEveryTick(false)`; at the beginning of the script. This ensures that the script is only executed at the end of a period and not with each incoming tick. Obviously this will tremendously decrease the overall workload.

7.17 Interpretation – Computing the Sentiments

The main aspect of a Sentimentor is obviously the computation of a sentiment for each period. This is done in the `Interpretation` Section of an Express

program. The Interpretation Section starts is introduced with the reserved word Interpretation.

7.17.1 Interpretation Using the Built-in Schemes

Very often the computation of the sentiments can be performed by one of the built-in interpretation schemes that are also used by the built-in sentimentors. Moreover, when using a built-in scheme, the corresponding editor for configuring the scheme details is available. So relying on a built-in scheme greatly simplifies the programming of an Express Sentimentor.

A built-in scheme can be called like a normal function.

Example:

```
interpretation TwoThresholds (mySeries, $upZone, $downZone);
```

or

```
interpretation TriggerLine (close, mySeries);
```

The TriggerLine scheme would compute the sentiments based on the close series crossing the series mySeries.

In case a built-in scheme requires input variables they have to be provided as parameters.

The following built-in schemes are available:

Built-in scheme	Typical usage
TwoThresholds (series curve, input upThreshold, input downThreshold)	RSI
TriggerLine (series curve, series trigger)	Crossing MA
Swing (series curve, input spanLeft, input spanRight)	Momentum
Bands (series curve, series lower, series upper)	Bollinger Bands

When using a built-in scheme, the plot statements may be omitted – Express will automatically plot the series used in the built-in scheme. However, if at least one plot statement is given, this standard mechanism is not applied.

7.17.2 Programming the Interpretation Explicitly

In case no built-in scheme matches the intended interpretation the sentiments can be computed explicitly using the syntax:

```
interpretation
begin
  <statement>
...
  <statement>
```

end

Note the `begin` and `end` surrounding the statements for computing the sentiments.

The process for computing the sentiments equals the process in the calculation section, i.e., the statements are executed for each bar, starting with the oldest („left most“) bar.

The sentiments have to be assigned to the predefined series `sentiment`. NanoTrader initializes the elements of the sentiment series with 50, i.e., *neutral*.

Example

```
interpretation
begin
  if CrossesAbove (close, mySeries) then
    sentiment = 100;
end
```

Instead of assigning the sentiment for the current bar only, it is also possible to assign sentiments for the following bars. With this technique, an *event* can be of significance not only in the period where it happens but also in the following periods.

Example:

```
interpretation
begin
  if CrossesAbove (close, mySeries) then
    sentiment = [100; 90; 80;];
  else if close > mySeries then //staying above mySeries
    if sentiment = 50 then //do not overwrite crossing event
      sentiment = 65;
end
```

The so-called *list assignment*

```
sentiment = [value; value;...;];
```

is only valid for the series `sentiment`.

7.18 Plotting

The final statements of an Express program are one or more `plot` statements following the syntax:

```
plot (<series name>, <colorname>, <pen width>);
```

or

```
plotline (<constant or variable>, <colorname>, <pen width>);
```

Example:

```
plot (mySeries, "blue", 2);
plotline ($threshold, "green" 1);
```

The following colors are predefined:

red, lightRed, green, lightGreen, blue, lightBlue, magenta,

lightMagenta, yellow, lightYellow, cyan, lightCyan, grey, black, white.

The colors are not case sensitive. In case of a typo, i.e., an undefined colour, blue is chosen.

A color can also be defined as a so-called RGB-value (RGB= Red-Green-Blue) using the syntax:

```
plot (<series name>, <red>, <green>, <blue>, <pen width>);
```

The red/green/blue values are integers in the range of 0 to 255 defining the strength of the respective color.

Sometimes it is interesting to plot candles or bars based on real or modified price data. This can be achieved by the following plot routines:

```
plotcandles (<open series>, <close series>, <high series>, <low series>);
```

or

```
plotbars (<open series>, <close series>, <high series>, <low series>);
```

To fill the area between two series use `plotband` (e.g. for sentimentors like Bollinger Bands):

```
plotband (<upper series name>, <colorname>, <pen width>,  
         <lower series name>, <colorname>, <pen width>,  
         <fillcolor>);
```

If two series that are crossing each other are to be plotted and the enclosed areas should be filled in dedicated colors use `plotcrossinglines` (e.g. for sentimentors like Crossing Moving Averages):

```
plotcrossinglines (<series1 name>, <colorname>, <pen width>,  
                  <series2 name>, <colorname>, <pen width>,  
                  <fillcolor series1 above series2>,  
                  <fillcolor series1 below series2>);
```

8 A Blocker Example

In addition to the sentiment values in the range from 0 to 100, *NanoTrader* supports two specific sentiment states that are used in conjunction with filters:

- **BLOCK**
Long *and* Short signals are rejected
- **FLAT**
Long *and* Short signals are rejected. In addition, a possible open position is closed.

<pre> if IsFirstBar () then MovingAverage (close, ma, 10); if MarketPosition() = 1 then //long begin if IsIntradayEntry() then //we just opened the position SetStopPrice (EntryPrice() - 15); else SetStopPrice (ma - 100 + \$increase* BarsSinceEntry()); end else if MarketPosition() = -1 then //short begin if IsIntradayEntry() then //we just opened the position SetStopPrice (EntryPrice() + 15); else SetStopPrice (ma + 100 - \$increase* BarsSinceEntry()); end end </pre>	2. 3. 4. 5.
--	----------------------

Explanation:

1. The keyword `Stop` declares this sentimentor to work as a price based stop. This enables the usage of functions only available for this kind of sentimentors. Moreover, this keyword ensures that the sentimentor can only be added as a `Stop` into a study.
2. The function `MarketPosition()` informs about the current position:
 1 = long
 0 = flat
 -1 = short
3. The boolean funtion `IsIntradayEntry()` returns `true` in case the position has just been opened in the current, not yet closed period. Sometimes it is necessary to use a different scheme for calculating the stop price for the initial period, e.g., based only on the entry price. In case the position is not closed within this initial period, the stop price for the next period to come will be calculated again.
4. The function `SetStopPrice()` makes the computed stop price available to *NanoTrader* that will choose the tightest stop among all used stops in the current study.
5. For `Stop` sentimentors there is no `Interpretation Section` and no `Plot Section`

10

Stop/Tactic Example with intraperiod updates

At activation time/fill time the stop is placed `$initialRisk` ticks below the entry price. When the traded price reaches the entry price + `$profitTrigger` ticks the

stop is adjusted to entryPrice + \$initialProfitOffset. From that moment on it starts trailing with a distance of \$trail ticks. If the \$trail parameter is set to 0, not trailing occurs.

<pre> Express Stop BETrailStop vars input \$initialRisk(0, 50, 10); input \$profitTrigger(0, 5, 3); input \$initialProfitOffset(0, 5, 2); input \$trail(0, 10, 5); numeric entryPrice, tickSize; numeric extreme; numeric breakEven, trailStop; calculation if IsFirstBar() then begin SetIntraPeriodUpdate(); entryPrice = EntryPriceOriginal(); tickSize = TickSize(); end if MarketPosition() = 1 then //Long position begin if IsIntradayEntry() then extreme = MaxPriceEntryBar(); else if (BarsSinceEntry() = 0) then//We entered via a study //at the end of the period extreme = close; else extreme = max (extreme, Highest(high, BarsSinceEntry())); breakEven = entryPrice + \$profitTrigger * tickSize; if \$trail = 0 then //trail deactivated? trailStop = -9999; else trailStop = extreme - \$trail * tickSize; if extreme >= breakEven then SetStopPrice (max(entryPrice + \$initialProfitOffset * tickSize, trailStop)); else SetStopPrice(entryPrice - \$initialRisk * tickSize); end else if MarketPosition() = -1 then //Short position begin if IsIntradayEntry() then extreme = MinPriceEntryBar(); else if (BarsSinceEntry() = 0) then extreme = close; else </pre>	<p>1.</p> <p>2.</p> <p>3.</p>
---	-------------------------------

```

    extreme = min (extreme, Lowest (low, BarsSinceEntry()));

breakEven = entryPrice - $profitTrigger * tickSize;
if $trail = 0 then
    trailStop = 999999;
else
    trailStop = extreme + $trail * tickSize;

if extreme <= breakEven then
    SetStopPrice (min(entryPrice - $initialProfitOffset *
        tickSize, trailStop));
else
    SetStopPrice(entryPrice + $initialRisk * tickSize);
end

```

Explanation:

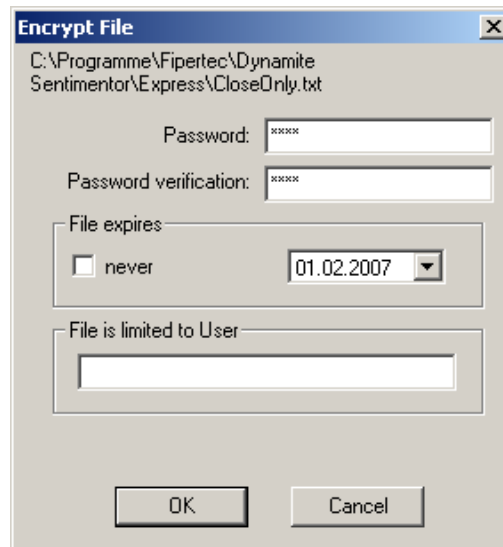
1. The existence of this routine in the source code activates the intraperiod updates.
2. The opening price of the position as booked in the account.
3. The highest price achieved in the opening period after opening the position. Example: 60-minutes periods. Position entry after 30 minutes => the return value is the high of the remaining 30 minutes.

11 Encrypting Express-Sentimentors

Express sentimentors can be encrypted. This allows a broad range of commercial third party applications. An encrypted Express sentimentor can be used exactly as the provided built-in sentimentors. However, the Express code itself cannot be edited or seen by the user.

To encrypt an Express sentimentors, chose from the main menubar Extras|Encrypt Express Sentimentors.

This will bring up a file selection dialog which allows you to select all the Express sentimentors to be encrypted. After finishing the selection the following dialog is shown:



This dialog allows to define the password. Moreover, an optional expiration date can be defined, i.e., you might enforce a user to renew his subscription to you Express sentimentor after a given date.

Also, the Express sentimentor can be licensed to a special user name.

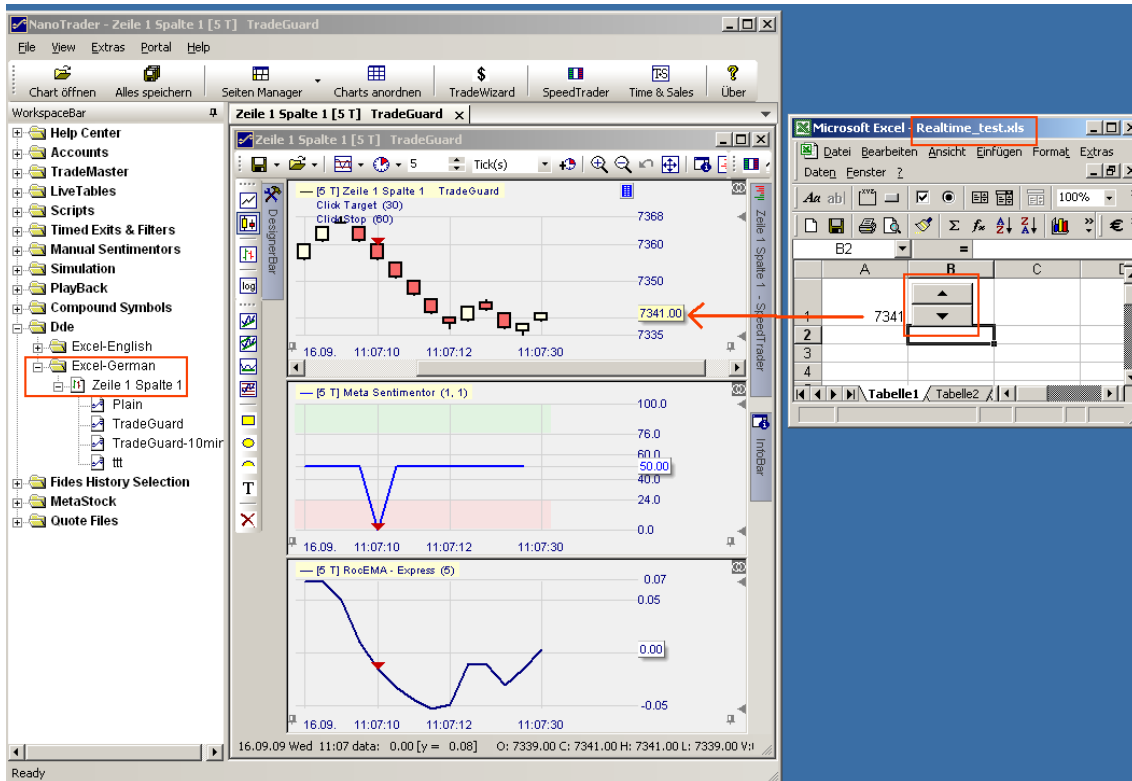
In contrast to a normal Express sentimentor an encrypted Express sentimentor is *referred* to by the study, i.e., the study refers to the encrypted file that needs to be located in the Express subdirectory of the installation directory. Hence, when distributing a complete study that contains an encrypted Express sentimentor, the encrypted Express file has also to be delivered.

12 Tipps for Debugging

The Express environment is primary designed for small developments and thus does not contain a dedicated debugger. However, there are various techniques to help the debugging process:

- plot a series that holds intermediate values
- use Annotate() or Highlight() to plot values and debugging notes directly into the chart
- use ShowTip() to assign values and notes to individual bars

Often specific data constellations need to be tested. To achieve this it is very comfortable to create the data via Excel yourself. To do so activate DDE in Extras|Datasources. In the installation directory of NanoTrader you will find the file Realtime_test.xls. Open this file and add the sentimentor which is to be debugged to a study that is based on Excel. (Depending on you Excel version you might need to use a symbol available in the "Excel-English" folder.)



13 Built-In Functions and Procedures

Express provides a number of built-in functions that can be called from within an Express program. If a built-in function requires parameters, NanoTrader checks if the provided parameters match the *function definition*. A function definition declares how a function is to be called.

The function definition of the `Max()`-function is given as:

```
float Max (float value1, float value2)
```

Hence, the return value of the function `Max` is of type `float`. The function takes two parameters, both of type `float`. Recall that Express automatically converts integer values to float values if needed, so

```
Max (close, 5000)
```

is a valid call, as 5000 would automatically be converted into a `float` value.

Functions returning a value can be used in expressions, as in
`mySeries = Max (open, close) * 2;`

Functions that do not return a value are also called *procedures*. A procedure call cannot be used in an expression. Instead, it forms a complete statement:

```
MovingAverage (mySeries, mySeries, $span);
```

The definition of `MovingAverage` is

```
void MovingAverage (series source, series target, int span)
```

The return value `void` indicates that there is in fact no return value.

The names of the parameters in the definitions are chosen such that they indicate their role for the function – they have no other specific meaning.

Even if a function does not receive parameters, the parenthesis have to be used:

```
index = CurrentBarIndex();
```

The following built-in functions are available:

Definition: float **AbsValue** (float value)

Meaning: Returns the absolute value of `value`.

Example: AbsValue (-3.7) returns 3.7; AbsValue (5) returns 5

Definition: void **Annotate** (string type, string color, float high, float low)
 void **Annotate RGB** (string type, int red, int green, int blue, float high, float low)
 void **AnnotateAt** (int offset, string type, string color, float high, float low)
 void **AnnotateRGBAAt** (int offset, string type, int red, int green, int blue, float high, float low)

Meaning: Adds a note to the current bar with respect to the chosen `type` in the specified `color`.

The following types are supported: (See screenshot)

“ellipse”, “upTriangle”, “downTriangle” as well as “labelLeft”, “labelCenter” and “labelRight”.

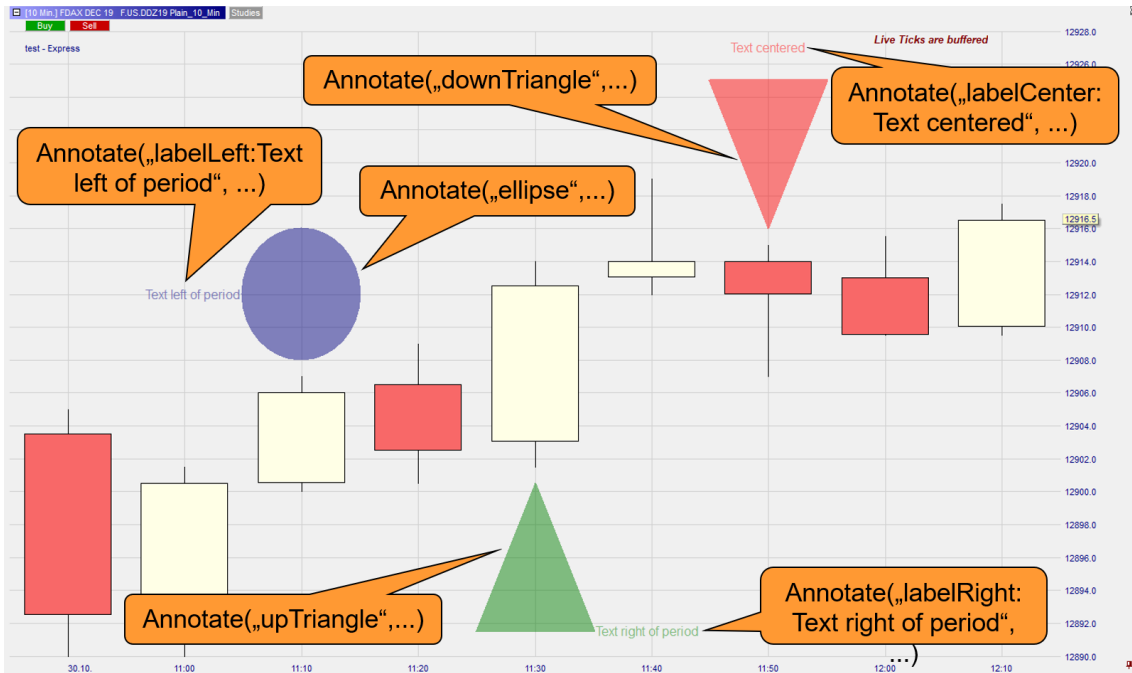
The vertical position and size of the annotation for the types “ellipse”, “upTriangle” and “downTriangle” is determined by the parameters `high` and `low`.

The text to be displayed with “labelLeft”, “labelCenter” and “labelRight” is appended to the type separated by a colon, e.g.:

```
Annotate("labelLeft:This text appears\nleft of the period",  
"black", (open+close)/2, 0);
```

A line break can be enforced by using the character sequence `\n`. Lines are not wrapped automatically.

The vertical position of the text is determined by the parameter `high`.



See function “plot” for a listing of the supported color names.

Multiple notes can be overlaid.

Example:

```
if (volume > 500) then Annotate(“ellipse”, “green”, high, low);
```

```
if (volume > 500) and IsBarCompleted() then
```

```
    Annotate(“ellipse”, “green”, high, low); //no intra bar annotation
```

The “At”-versions allow to annotate the current bar, similar to indexing price data. Ex. use `AnnotateAt(2, "upTriangle", "blue", high, low)` to set a note two periods before the current bar. This makes it easy to annotate price patterns that stretch out over multiple periods for example.

See also `Highlight()`.

Definition: float **ArcTangent** (float value)

Meaning: Returns the arcus tangent of `value`.

Example: `ArcTangent (2.7475)` returns 70.

Definition: float **Atr** (int span)

Meaning: Returns the Average True Range of the MasterChart for the actual and previous `span` bars expressed in percent.

Definition: float **AtrAbs** (int span)

Meaning: Returns the Average True Range of the MasterChart for the actual and previous `span` bars expressed in points.

Definition: void **Bands** (series series, series lower, series upper)

Meaning: Standard interpretation scheme where the sentiments are computed based on an upper and lower series

Example: interpretation Bands (close, myLower, myUpper);

Definition: int **BarsSinceEntry** ()

This function is only available for Stop sentimentors.

Meaning: Since how many periods is the current trade open?

Example: if (MarketPosition() = 1) and (BarsSinceEntry() > 10) then ...

Definition: void **CalculateAtEveryTick** (bool value)

Meaning: Call CalculateAtEveryTick(false) to disable the execution of an Express script for each incoming tick. The script will then be executed only at the end of a period. This helps tremendously to speed up very time consuming or poorly programmed Express sentimentors.

Example: if IsFirstBar() then CalculateAtEveryTick(false);

Definition: float **Ceiling** (float value)

Meaning: Returns the smallest integer greater than `value`.

Example: Ceiling (2.95) returns 3

Definition: float **Cosine** (float value)

Meaning: Returns the cosine of `value` degrees.

Example: Cosine (45) returns 0.7071

Definition: void **CreateFile**(string file, string text)

Meaning: Creates a file with the given text as content. If the file already exists, the given text will be appended. The path and filename is defined by using the parameter file.

Please see function PlaySound() for a description of when a file is created. The same principles apply as for playing sounds.

Example:

if (close > high[1]) and (close > high[2]) then

```
CreateFile("C:\nano-actions.txt", TimeToString(datetime, "%Y-%m-%d
%H:%M:%S") + " New peak at symbol " + SymbolName());
```

Definition: bool **CrossesAbove** (series curve, series trigger)

Meaning: Returns true (if curve[1] <= trigger [1]) and (curve > trigger), false otherwise

Example: if CrossesAbove (mySeries, close) then sentiment = 100;

Definition: bool **CrossesAboveThreshold** (series curve, float threshold)

Meaning: Returns true (if curve[1] <= threshold) and (curve > threshold), false otherwise

Example: if CrosseAboveThreshold (mySeries, 70) then sentiment = 100;

Definition: bool **CrossesBelow** (series curve, series trigger)

Meaning: Returns true (if curve[1] >= trigger[1]) and (curve < trigger), false otherwise

Example: if CrossesBelow (mySeries, close) then sentiment = 0;

Definition: bool **CrossesBelowThreshold** (series curve, float threshold)

Meaning: Returns true (if curve[1] >= threshold]) and (curve < trigger), false otherwise

Example: if CrossesBelowThreshold (mySeries, 30) then sentiment = 0;

Definition: int **CurrentBarIndex** ()

Meaning: The index of the currently processed bar. The first bar has the index 0.

Example: highDiff = CurrentBarIndex() – IndexOfHighest (close, 10);

Definition: float **DateToNumeric** (time value)

Meaning: Converts a date into a numeric value. The date 2013-04-23 is converted into 130423.

This function is ideal whenever the computation should rely on the date.

Example:

```
sentiment = preCalculatedSentiment * (1 – (DateToNumeric(dateopen)-
130000)/1231);
```

Definition: int **Duration** (time start, time end)

Meaning: The duration in seconds from start to end.

Example: d = Duration (timeOpen, time);

d = Duration (dateTime[20], dateTime);

Note: The caller needs to make sure that both parameters are of identical type, i.e., Duration (dateOpen, time) will give an invalid result as “dateOpen” only contains the day-component whereas “time” only the time component.

Definition: int **DayOfWeek** (time time)

Meaning: The index of the day in the week from `time`, where Monday = 1, Tuesday = 2, ...

Example: if DayOfWeek(date) = 5 then sentiment = 50;//no entries on Fridays

Definition: float **EntryPrice** ()

This function is only available for Stop sentimentors.

Meaning: The entry price for the current position. If the Stop is used in a TradeGuard or as a Tactic then the return value is the price that was traded at activation time of the TradeGuard or Tactic. If the TradeGuard was already active while the position was opened then the price equals the fill price.

See also EntryPriceOriginal().

Example: SetStopPrice(EntryPrice() – 5 * TickSize());

Definition: float **EntryPriceOriginal** ()

This function is only available for Stop sentimentors.

Meaning: The actual opening price of the account position.

See also EntryPrice ().

Example: SetStopPrice(EntryPriceOriginal() + TickSize());

Definition: float **Exp** (float value)

Meaning: The exp() function applied on `value`.

Example: val = exp(1.254);

Definition: int **FinalBarIndex** ()

Meaning: Returns the index of the final period in the evaluation range.

Example:

```

if IsFirstBar() then
begin
  ExpMovingAverage(close, ema1, 5);
  ExpMovingAverage(close, ema2, 33);

  for i = 0 to FinalBarIndex()
    helper[-i] = ema1[-i] - ema2[-i];
end
  
```

Definition: **ExpMovingAverage** (series source, series target, int span)

Meaning: Computes the span-bar exponential moving average of `source` and writes the result into `target`. This function should only be used in conjunction with IsFirstBar() or IsFinalBar().

Example:

```

if IsFirstBar () then ExpMovingAverage (close, mySeries, $span);
if IsFinalBar () then ExpMovingAverage (mySeries, mySeries, $span);
  
```

Definition: float **Floor** (float value)

Meaning: Returns the largest integer smaller than `value`.

Example: Floor (2.95) returns 2

Definition: string **GetApplicationLanguage** ()

Meaning: Returns the configured language of NanoTrader. Possible return values are DEU, ENG, FRA, HUN, ITA, NLD and POL.

Example:

```

if GetApplicationLanguage() = "ENG" then
  MessageBox("This message is in English.");
  
```

Definition: int **GetArraySize** (array arr)

Meaning: Returns the number of elements of array `arr`.

Example:

```

sum = 0;
for i = 0 to GetArraySize(arr) - 1
begin
  sum = sum + arr[i];
end
  
```


Definition: int **GetDefaultOrderSize** ()

Meaning: Returns the order size for the instrument the Express script is running on. This function is available in experts mode only (see SetExpertsMode()).

This function is ideal whenever the order size needs to increase/decrease in relation to the value of the sentiment.

```
Example:  if (sentiment = 0) OR (sentiment = 100) then
           SetDefaultOrderSize(GetDefaultOrderSize()+1);
           else
           SetDefaultOrderSize(1);
```

Definition: time **GetExpiration** ()

Meaning: Returns the expiration date and time of the symbol the Express sentimentor is attached to. If the symbol has no expiration, the function results January 1st 1970 at 0:00.

Example:

```
if (date >= GetExpiration()) then
    sentiment = senti_flat;
```

Definition: string **GetPriceFormat** ()

Meaning: Returns the internal format for plotting the price in the MasterChart's y-axis.

Example: SetYscaleFormat (GetPriceFormat());

Display the indicator's y-axis in the same format as the MasterChart is displayed.

Definition: float **GetSpreadSize** ()

Meaning: Returns the current spread of the symbol the Express sentimentor is attached to.

Example: ask = close + GetSpreadSize();

Definition: float **Highest** (series series, int span)

Meaning: Returns the highest value in series for the elements series[0], ... series[span - 1]

Example: tenBarHigh = Highest (close, 10);

Definition: void **Highlight** (string type, string color)
 void **HighlightRGB** (string type, int red, int green, int blue)
 void **HighlightAt** (int offset, string type, string color)
 void **HighlightRGBA** (int offset, string type, int red,int green,int blue)

Meaning: Highlights the current bar with respect to the chosen `type` in the specified `color`.

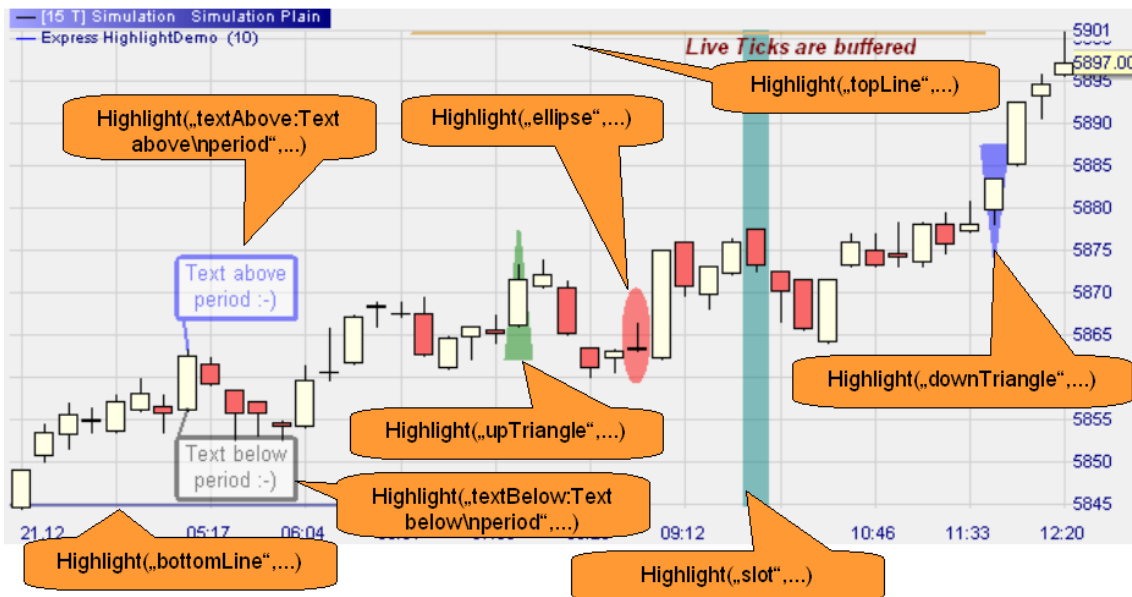
The following types are supported: (See screenshot)

“ellipse”, “upTriangle”, “downTriangle”, “slot”, “bottomLine”, “topLine”, as well as “textAbove”, “textBelow”.

The text to be displayed with “textAbove” and “textBelow” is appended to the type separated by a colon, e.g.:

```
Highlight("textAbove:This text appears\nabove the period", "black");
```

A line break can be enforced by using the charater sequence `\n`. Lines are not wrapped automatically.



See function “plot” for a listing of the supported color names.

Multiple highlights can be overlaid.

Example:

```
if (volume > 500) then Highlight("ellipse", "green");
```

```
if (volume > 500) and IsBarCompleted() then
    Highlight("ellipse", "green"); //no intra bar highlighting
```

The “At”-versions allow to highlight the current bar, similar to indexing price data. Ex. use `HighlightAt(2, "upTriangle", "blue")` to set a highlight two periods

before the current bar. This makes it easy to highlight price patterns that stretch out over multiple periods for example.

Definition: **IndexOfHighest** (series series, int span)

Meaning: Returns the index of the highest value for the elements series[0], ... series[span – 1]

Example: highDiff = CurrentBarIndex() – IndexOfHighest (close, 10);

Definition: **IndexOfLowest** (series series, int span)

Meaning: : Returns the index of the lowest value for the elements series[0], ... series[span – 1]

Example: lowDiff = CurrentBarIndex() – IndexOfLowest (close, 10);

Definition: bool **IsBarCompleted**()

Meaning: Returns true if the period currently worked on is completed.

Example:

if IsBarCompleted() and (volume > 1000) then
 PlaySound("gong");

Definition: bool **IsFinalBar**()

Meaning: Returns true if currently the final bar is processed.

Example: if IsFinalBar () then MovingAverage (mySeries, mySeries, \$span);

Definition: bool **IsFirstBar**()

Meaning: Returns true if currently the first bar is processed.

Example: if IsFirstBar () then MovingAverage (close, mySeries, \$span);

Definition: bool **IsIntradayEntry**()

This function is only available for Stop sentimentors.

Meaning: Returns `true` in case the position has just been opened in the current, not yet closed period. Sometimes it is necessary to use a different scheme for calculating the stop price for the initial period, e.g., based only on the entry price. In case the position is not closed within this initial period, the stop price for the next period to come will be calculated again.

Example: if IsIntradayEntry () then SetStopPrice(EntryPrice() – 0.05);

Definition: bool **IsNewDay**()

Meaning: Returns `true` in case the the current bar is the first bar of a new day of if it is the very first bar of the available data.

Example: `if IsNewDay () then Highlight("slot", "blue");`

Definition: bool **IsNonZero**(float value)

Meaning: Returns true if `value` ≥ 0.001 . Never test with „= 0“, because due to rounding errors this condition is very rarely met.

Example: `if IsNonZero (a * b) then val = sum / (a * b);`

Definition: bool **IsZero**(float value)

Meaning: Returns true if `value` < 0.001 . Never test with „= 0“, because due to rounding errors this condition is very rarely met.

Example: `if Not IsZero (a * b) then val = sum / (a * b);`

Definition: float **Log** (float value)

Meaning: Returns the natural logarithm of `value` or void if `value` ≤ 0 .

Example: `Log (1000)` returns 6.9078

Definition: float **Lowest** (series series, int span)

Meaning: Returns the lowest value in series for the elements `series[0], ... series[span - 1]`

Example: `tenBarLow = Lowest (close, 10);`

Definition: int **MarketPosition** ()

This function is only available for Stop sentimentors.

Meaning: Returns the direction of the current position:

1 = long

0 = flat

-1 = short

Example: `if MarketPosition() = 1 then SetStopPrice (low - 0.01);`

Definition: int **MarketPositionSize** ()

This function is only available for Stop sentimentors.

Meaning: Returns the volume size of the current position:

> 0 = long

0 = flat
 < 0 = short

Stops containing MarketPositionSize() are ignored in backtesting.

Example: if MarketPositionSize() > 3 then
 SetStopPrice (low – 0.03);
 else
 SetStopPrice (low – 0.01);

Definition: float **Max** (float value1, float value2)

Meaning: Returns the maximum value of `value1` and `value2`

Example: Max (3, 7.5) returns 7.5

Definition: float **MaxPriceEntryBar** ()

This function is only available for Stop sentimentors.

Meaning: For Stops with intraperiod updates (see SetIntraPeriodUpdate()). Returns the highest price of the opening period of a trade that was achieved *after* the position was opened.

Example:
 If (MarketPosition() = 1) and IsIntradayEntry() then
 SetStopPrice (MaxPriceEntryBar() - 5 * TickSize());

Definition: float **MinPriceEntryBar** ()

This function is only available for Stop sentimentors.

Meaning: For Stops with intraperiod updates (see SetIntraPeriodUpdate()). Returns the lowest price of the opening period of a trade that was achieved *after* the position was opened.

Example:
 If (MarketPosition() = -1) and IsIntradayEntry() then
 SetStopPrice (MinPriceEntryBar() + 5 * TickSize());

Definition: void **MessageBox**(string message)

Meaning: Displays `message` in a popup window.

Please see function PlaySound() for a description of when a message is displayed. The same principles apply as for playing sounds.

Example:
 if (close > high[1]) and (close > high[2]) then
 MessageBox("New peak at symbol " + SymbolName());

Definition: float **Min** (float value1, float value2)

Meaning: : Returns the minimum value of `value1` and `value2`

Example: Min (3, 7.5) returns 3

Definition: **MovingAverage** (series source, series target, int span)

Meaning: Computes the span-bar MovingAverage of `source` and writes the result into `target`. This function should only be used in conjunction with IsFirstBar() or IsFinalBar().

Example:

if IsFirstBar () then MovingAverage (close, mySeries, \$span);

if IsFinalBar () then MovingAverage (mySeries, mySeries, \$span);

Definition: **NoDrawingOHLCinMC** ()

Meaning: Disables the drawing of the data series open, high, low and close (e.g. bars, renkos, candles) in the MasterChart. This function is available in experts mode only (see SetExpertsMode()).

Example: if IsFirstBar() then NoDrawingOHLCinMC();

Definition: **NormalCDF** (float value)

Meaning: Returns the value of the density function of the standard normal distribution at value `value`.

Example:

cdf = NormalCDF(0.2);

Definition: **NormalIPDF** (float value)

Meaning: Returns the value of the standard normal distribution at value `value`.

Example:

cdf = NormalIPDF(0.2);

Definition: time **NumericToDate** (float value)

Meaning: Converts `value` into a time value whereby `value` is interpreted as YYYYMMDD, e.g., 130423 is converted into the time 2013-04-23.

If the day-part of `value` is larger than the number of days in the specified month it is set to the last day of the month.

If the month-part of `value` is larger than 12 it is set to 12.

The year-part of `value` cannot be larger than 99.

This function is ideal whenever the computation should rely on a date that needs to be adjustable through the DesignerBar and/or the optimizer.

Example:

```
if (date >= NumericToDate($blockStart))
  and (date <= NumericToDate($blockEnd)) then
  sentiment = senti_block;
```

Definition: string **NumericToString** (float value, string format)

Meaning: Formats `value` according to format `format` and returns the result as a string.

`Format` supports all formats as used for the C-function "printf()". The most important formats are:

"%f"	decimal floating point
"%6.2f"	rounds to two decimals
"%g"	discards trailing zeroes
"%e"	scientific notation

In case format is the empty string the function uses the "%g" format.

For formatting a price, see function PriceToString().

Example:

```
ShowTip(NumericToString(val, "%6.4f");
```

Definition: time **NumericToTime** (float value)

Meaning: Converts `value` into a time value whereby `value` is interpreted as HHMM, e.g., 1545 is converted into the time 15:45.

If the hour-part of `value` is larger than 23 it is set to 23.

If the minute-part `value` is larger than 59 it is set to 59.

This function is ideal whenever the computation should rely on a time that needs to be adjustable through the DesignerBar and/or the optimizer.

Example:

```
if (time >= NumericToTime($blockStart))
  and (time <= NumericToTime($blockEnd)) then
  sentiment = senti_block;
```

Definition: void **PlaySound**(string sound)

Meaning: Plays the sound file denoted by `sound`. `sound` may be a complete path name to the .wav file to be played or it may be the so-called *file title* of a .wav file residing in the subdirectory Wav of the installation directory.

E.g. if that directory contained a file named "ringin.wav" then the call PlaySound("ringin") would refer to that file.

A sound is only played once and only if it occurred by receiving live data.

Note that with every incoming tick the Express program is executed, hence for the current period the sound will by default be played as soon as the function

PlaySound() is called and will not wait until the end of the period.
 If a sound should only be played at the end of a period this can be achieved as follows:

```
if IsBarCompleted () and soundCondition then
  PlaySound("gong");
```

Note that per period only that sound is played that was initiated as the first sound.

If 'sound' cannot be resolved to a valid wav-file a beep is played.

Example:

```
if volume > 300 then
  PlaySound("gong"); //intra bar notification of a high volume period
```

```
if IsBarCompleted () and (close > high[1]) then
  PlaySound ("corkpop"); //end of bar notification of a period's close exceeding
  //the previous period's high
```

Definition: void **Plot** (series curve, string color, int penWidth)
 void **Plot** (series curve, int red, int green, int blue, int penWidth)

Meaning: Plots the series 'curve' using in the specified color and pen width.

```
Example: Plot (close, "green", 2);
  Plot (close, 128, 128, 128, 1); //grey
```

Definition: void **PlotBand** (series curve1, string color1, int penWidth1,
 series curve2, string color2, int penWidth2, string fillColor)

Meaning: Plots the series 'curve1' and 'curve2' and fills the interior with color 'fillColor'. An almost infinite number of colors can be selected using the RGB color scheme.

```
Example: PlotBand(upper, "green", 2, lower, "red", 2, "lightgreen");
  PlotBand(upper, 150, 0, 0, 2, lower, 0, 0, 0, 2, 128, 128, 128);
```

Definition: **PlotBars** (series open, series close, series high, series low)
PlotBars (series open, series close, series high, series low,
 string colorBull, string colorBear)
PlotBars (series open, series close, series high, series low,
 series colors)

Meaning: Plots a bar chart using the specified series. If no colors are specified, the ColorManager settings for bullish and bearish bars will be used. An almost infinite number of colors can be selected using the RGB color scheme.

Example: PlotBars (myOpen, myClose, high, low);

PlotBars (myOpen, myClose, high, low, "lightyellow", "lightblue");

PlotBars (myOpen, myClose, high, low, 150, 0, 0, 128, 128, 128);

PlotBars (myOpen, myClose, high, low, myColors);

Definition: **PlotCandles** (series open, series close, series high, series low)

PlotCandles (series open, series close, series high, series low,
string colorBull, string colorBear)

PlotCandles (series open, series close, series high, series low,
series colors)

Meaning: Plots a candle stick chart using the specified series. If no colors are specified, the ColorManager settings for bullish and bearish candles will be used. An almost infinite number of colors can be selected using the RGB color scheme.

Example: PlotCandles (myOpen, myClose, high, low);

PlotCandles (myOpen, myClose, high, low, "lightyellow", "lightblue");

PlotCandles (myOpen, myClose, high, low, 150, 0, 0, 128, 128, 128);

PlotCandles (myOpen, myClose, high, low, myColors);

Definition: void **PlotCrossingLines** (series curve1, string color1, int penWidth1,
series curve2, string color2, int penWidth2, string fillColor1,
string fillColor2)

Meaning: Plots the series `curve1` and `curve2` and fills the interior with color `fillColor1` when `curve1` is above `curve2`. Otherwise the interior is filled with color `fillColor2`. An almost infinite number of colors can be selected using the RGB color scheme.

Example: PlotCrossingLines(upper, "green", 2, lower, "red", 2, "lightgreen",
"lightred");

PlotCrossingLines(upper, 150, 0, 0, 2, lower, 0, 0, 0, 2, 128, 128,
128, 200, 200, 200);

Definition: void **PlotLine** (float value, string color, int penWidth)

Meaning: Plots a horizontal line at level `value` using the specified color and pen width..

Example: PlotLine (\$threshold, "red", 2);

PlotLine (100, 150, 0, 0, 1); //dark red

Definition: float **PointValue** ()

Meaning: Returns the point value of the symbol the Express sentimentor is attached to.

Example: `barValue = (high – low) * PointValue();`

Definition: float **Power** (float value, float exponent)

Meaning: Returns `value` raised to the power `exponent`.

Example: `Power (3, 3)` returns 27

Definition: float **PrevDayHigh/Low/Open/Close/Vol** ()

Meaning: Returns the High/Low/Open/Close/Vol of the previous day or `void` in case the data is not available

Example: `yesterdayMedian = (PrevDayHigh() + PrevDayLow() + PrevDayClose()) / 3`

Definition: string **PriceToString** (float value)

Meaning: Rounds `value` to the nearest price with respect to the defined ticksize and precision of the analyzed symbol and converts it into a string. Takes fractional notations into account.

Example: `ShowTip("TriggerPrice = " + PriceToString(high[1] + 2*TickSize()));`

Definition: int **RGB** (int red, int, green, int blue)

Meaning: Converts the given red, green and blue portions of a color into an integer specifying the same color. This function is helpful when assigning colors to a series as parameter for `PlotBars()` and `PlotCandles()`.

Example: `myColors = RGB (255, 255, 160);`

Definition: float **Round** (float value, int precision)

Meaning: Returns `value` rounded to `precision` decimals.

Example: `Round (2.428, 2)` returns 2.43

Definition: float **RoundMultiple** (float value, float multiple)

Meaning: Returns `value` rounded to the nearest multiple of `multiple`.

Example: `RoundMultiple ((high + low) / 2, TickSize());`

Definition: **RSI** (series source, series target, int span)

Meaning: Computes the span-bar RSI of `source` and writes the result into `target`. This function should only be used in conjunction with `IsFirstBar()` or `IsFinalBar()`.

Example:

```
if IsFirstBar () then RSI (close, mySeries, $span);
```

```
if IsFinalBar () then RSI (mySeries, mySeries, $span);
```

Definition: void **Screenshot**(string filename)

Meaning: Creates a screenshot of the MasterChart and saves it as png file in `filename`. If `filename` does not contain a full path then it is saved in the Screenshots directory.

The screenshot is done exactly as shown on the screen and using the same dimensions.

It is recommend to use the function `ScreenshotEx()` instead as it offers more features.

Please see function `PlaySound()` for a description of when screenshot is done. The same principles apply as for playing sounds.

Example:

```
If IsFinalBar() then
```

```
    Screenshot("epxpress.png");
```

Definition: void **ScreenshotEx**(string filename, int style, int width, int height, int periodsOrDays)

Meaning: Creates a screenshot of the MasterChart or Equity chart and saves it as png file in `filename`. If `filename` does not contain a full path then it is saved in the Screenshots directory.

The `style` determines the elements of the MasterChart or Equity chart to be drawn and defines the default dimensions.

0 = exactly as shown on the screen

1 = less decoration, medium sized

2 = even less decoration, small

3 = show Equity, medium sized

4 = show Equity, small

Use `width` and `height` to define the dimension of the created image. Set to 0 to use the default as defined by the style.

`periodsOrDays` defines the number of periods to be displayed, counting from the last available period. If set to a value greater than 0 then it defines the total number of periods to be shown. If set to 0 then it shows the starting point of the

current zoom in the MasterChart. If set to a negative number then it is interpreted as full days, e.g., -2 means “show today and yesterday”.

Please see function PlaySound() for a description of when screenshot is done. The same principles apply as for playing sounds.

Example:

```
If IsFinalBar() then
  ScreenshotEx("epxpress.png", 0, 600, 400, -2);
```

Definition: void **SendEmail**(string subject, string message)

Meaning: Sends an email using the given subject and message to the email address configured under Extras/Options.

Please see function PlaySound() for a description of when an email is sent. The same principles apply as for playing sounds.

Example:

```
if (close > high[1]) and (close > high[2]) then
  SendEmail("NanoTrader-Notification", "New peak at symbol " +
  SymbolName());
```

Definition: void **SetArraySize** (array arr, int size)

Meaning: Sets the size of array `arr` to `size`, i.e., the elements can be accessed using the indices 0 to (size – 1).

Example: if IsFirstbar() then arr.SetSize(arr, \$arrSize);

Definition: void **SetArrayTo** (array arr, float value)

Meaning: Sets all entries of the array `arr` to `value`.

Example: if IsFirstbar() then SetArrayTo(arr, 500);

Definition: void **SetDefaultOrderSize** (int value)

Meaning: Sets the order size to `value` for the instrument the Express script is running on. As the order size is changed per instrument the modification is applied to all accounts and studies containing the instrument. This function is available in experts mode only (see SetExpertsMode()).

This function is ideal whenever the order size should rely on the value of the sentiment.

```
Example:   if (sentiment = 0) OR (sentiment = 100) then
           SetDefaultOrderSize(2);
           else
           SetDefaultOrderSize(1);
```

Definition: void **SetExpertsMode** ()

Meaning: Call SetExpertsMode() to enable the usage of functions in Express which are requiring expert knowledge (e.g. SetDefaultOrderSize()). By setting the experts mode you confirm that you know what you are doing. The responsibility for any harm caused by expert functions is up to you.

Example: if IsFirstBar() then SetExpertsMode();

Definition: void **SetIntraPeriodUpdate** ()

Meaning: Valid only for Stop sentimentors. The Stop calculation is set to be done with each incoming tick. This is specifically useful when programming tactics.

Stops containing SetIntraPeriodUpdate() are ignored in backtesting.

Note: For internal reasons the existence of the function call in the source code suffices to activate the intra period computation – even if the corresponding statement is never executed, i.e., even with code like
 If false then SetIntraPeriodUpdate(); the intra period computation is activated.

Definition: void **SetLongTrigger** (float value)

Meaning: Defines the confirmation price or limit price for a long signal. To activate the evaluation of this price the Evaluator's policy for „Sentiment Enter Signals“ must be set to „Confirmation price next bar“ or „Limit price next bar“. In case no sentimentor calls this routine the confirmation price is set to the High/Low of the period generating the signal. The limit price will be set to the close of the period generating the signal. In case many sentimentors call this routine the strictest price is taken.

Example: SetLongTrigger ((high + low) / 2);

Definition: void **SetShortTrigger** (float value)

Meaning: Analogously to SetLongTrigger().

Example: SetShortTrigger ((high + low) / 2);

Definition: void **SetStopPrice** (float value)

This function is only available for Stop sentimentors.

Meaning: Defines the stop price for the current period in case the position has just been entered or for the next period in case the position has been entered before this period.

Note: In a given sentimentor there cannot be calls to both SetStopPrice() and SetTargetPrice().

Example: SetStopPrice (low[-1]);

Definition: void **SetTargetPrice** (float value)

This function is only available for Stop sentimentors.

Meaning: Defines the target price for the current period in case the position has just been entered or for the next period in case the position has been entered before this period.

Note: In a given sentimentor there cannot be calls to both SetStopPrice() and SetTargetPrice().

Example: SetTargetPrice (high[-1]);

Definition: void **SetYscaleFormat** (string format)

Meaning: Defines the format of the y-axis in printf like format.

Example: SetYscaleFormat("%.5d"); //use 5 decimal places

SetYscaleFormat("%g"); //use the compactest display

Definition: void **ShowTip** (string message)

Meaning: Anchors `message` to the current bar. The message is displayed in a popup window when the cursor is above the bar. To indicate a new line use the character sequence `\n`.

Example: if CrossesAbove (mySeries, 70) then
 ShowTip („Entered upper zone!\nWait for confirmation.“);

Definition: float **Sign** (float value)

Meaning: Returns the sign of value.

Example: Sign (-3) return -1; Sign (5) returns 1; Sign (0) returns 0

Definition: float **Sine** (float value)

Meaning: Returns the sine of `value` degrees.

Example: Sine (70) returns 0.9397

Definition: float **SquareRoot** (float value)

Meaning: Returns the square root of value or void if `value` < 0.

Example: SquareRoot (4) returns 2

Definition: void **StdDev** (series source, series target, int span)

Meaning: Computes the standard deviation of the values source, source [1], ..., source[span-1] saves the result in target.

This function should only be used in conjunction with IsFirstBar().

Example: StdDev (close, myseries, 10);

Definition: float **Sum** (series series, int span)

Meaning: : Returns the sum of the elements series[0], ... series[span - 1].

Returns void in case one or more required elements are void.

Example: amount = Sum (mySeries, \$span);

Definition: void **Swing** (series series, input spanLeft, input spanRight)

Meaning: Standard interpretation scheme where the sentiments are computed based on swings in series

Example: interpretation Swings (mySeries);

Definition: string **SymbolName**()

Meaning: Returns the name of the symbol this Express script is working on.

Example:

if (close > high[1]) and (close > high[2]) then

 MessageBox("New peak at symbol " + SymbolName());

Definition: float **Tangent** (float value)

Meaning: Returns the tangent of `value` degrees.

Example: Tangent (70) returns 2.7475.

Definition: float **TickSize** ()

Meaning: Returns the ticksize of the symbol the Express sentimentor is attached to.

Example: trigger = high[1] + 3 * TickSize();

Definition: float **TickValue** ()

Meaning: Returns the tick vlaue of the symbol the Express sentimentor is attached to.

Example: profit = \$nbTicks * TickValue();

Definition: float **TimeToNumeric** (time value)

Meaning: Converts a time into a numeric value. The time 15:45 is converted into 1545.

This function is ideal whenever the computation should rely on the time.

Example:

```
sentiment = preCalculatedSentiment * (1 - TimeToNumeric(timeopen)/2400);
```

Definition: string **TimeToString** (time timeVal, string format)

Meaning: Format `timeVal` according to `format` into a string

Format supports the formatting of the C-function strftime(), i.e.:

- %a** Abbreviated weekday name
- %A** Full weekday name
- %b** Abbreviated month name
- %B** Full month name
- %c** Date and time representation appropriate for locale
- %d** Day of month as decimal number (01 – 31)
- %H** Hour in 24-hour format (00 – 23)
- %I** Hour in 12-hour format (01 – 12)
- %j** Day of year as decimal number (001 – 366)
- %m** Month as decimal number (01 – 12)
- %M** Minute as decimal number (00 – 59)
- %p** Current locale's A.M./P.M. indicator for 12-hour clock
- %S** Second as decimal number (00 – 59)
- %U** Week of year as decimal number, with Sunday as first day of week (00 – 53)
- %w** Weekday as decimal number (0 – 6; Sunday is 0)
- %W** Week of year as decimal number, with Monday as first day of week (00 – 53)
- %x** Date representation for current locale
- %X** Time representation for current locale
- %y** Year without century, as decimal number (00 – 99)
- %Y** Year with century, as decimal number
- %z, %Z** Time-zone name or abbreviation; no characters if time zone is unknown
- %%** Percent sign

If `format` is the empty string the time is formatted to display Date and Time.

Example: ShowTip(TimeToString(time, "%H:%M:%S"));

Definition: **TriggerLine** (series curve, series trigger)

Meaning Standard interpretation scheme where the sentiments are computed based on the crossings of `curve` and `series`.

Example: interpretation TriggerLine (close, mySeries);

Definition: void **TwoThresholds** (series series, input upThreshold, input downThreshold)

Meaning: Standard interpretation scheme where the sentiments are computed based on zones defined by two thresholds.

Example: interpretation TwoThresholds (mySeries, \$upperZone, \$lowerZone);

Definition: void **Unaggregate** (series source, series target)

Meaning: If the series `source` was imported from another sentimentor which potentially was aggregated then use Unaggregate() to map it back to the MasterChart aggregation.

Example:

```
series maAgg(MovingAverage.main); //the MovingAverage sentimentor is
                                //aggregated in the study
```

```
series ma;
```

```
...
```

```
if IsFirstBar() then
```

```
    Unaggregate (maAgg, ma);
```

Definition: **WeightedMovingAverage** (series source, series target, int span)

Meaning: Computes the span-bar Weighted MovingAverage of `source` and writes the result into `target`. This function should only be used in conjunction with IsFirstBar() or IsFinalBar().

Example:

```
if IsFirstBar () then WeightedMovingAverage (close, mySeries, $span);
```

```
if IsFinalBar () then WeightedMovingAverage (mySeries, mySeries, $span);
```