

Strategy Runner API

C++ Programming Reference

Variance Futures
www.variancefutures.com

- [Callbacks](#)
 - [onInitParameters](#)
 - [onRegisterBars](#)
 - [onAddIndicators](#)
 - [onRestoreOvernight](#)
 - [onInit](#)
 - [onStart](#)
 - [onEnterPositions](#)
 - [onOpenPositions](#)
 - [onClosePositions](#)
 - [onAlert](#)
 - [onBar](#)
 - [onWaitAfterEntrance](#)
 - [onWaitAfterWin](#)
 - [onWaitAfterLoss](#)
 - [onSaveOvernight](#)
 - [onStop](#)
- [Trading functions](#)
 - [registerBar](#)
 - [addIndicator](#)
 - [startIndicator](#)
 - [sendOrder](#)
 - [sendSmartOrder](#)
 - [modifyOrder](#)
 - [cancelOrder](#)
 - [cancelPackage](#)
 - [sendEnterLimitPositions](#)
 - [sendEnterLimitPositionsPts](#)
 - [sendEnterStopPositions](#)
 - [sendEnterStopPositionsPts](#)
 - [sendClosePositions](#)
 - [sendClosePositionsPts](#)
 - [sendCloseLimitPositions](#)
 - [sendCloseLimitPositionsPts](#)
 - [sendCloseStopPositions](#)
 - [sendCloseStopPositionsPts](#)
 - [closeAllPositions](#)
 - [cancelAllActions](#)
 - [cancelAllOrders](#)
 - [stopTrader](#)

- [Alert functions](#)
 - [setTimeAlert](#)
 - [setTimeOffsetAlert](#)
 - [setTimeTransactionAlert](#)
 - [setPriceAlert](#)
 - [setPnLAlert](#)
 - [setIndicatorAlert](#)
 - [cancelTimeAlert](#)
 - [cancelPriceAlert](#)
 - [cancelPnLAlert](#)
 - [cancelIndicatorAlert](#)
- [Query functions](#)
 - [getStartTime](#)
 - [getEndTime](#)
 - [getLimit](#)
 - [getStop](#)
 - [getOpenPositions](#)
 - [getWaitAfterEntrance](#)
 - [getWaitAfterWin](#)
 - [getWaitAfterLoss](#)
 - [getContractId](#)
 - [getContractCountParam](#)
 - [getMarketDirection](#)
 - [getCurrentPnL](#)
 - [getCurrentPnLPrice](#)
 - [getCurrentTime](#)
 - [getCurrentDate](#)
 - [getContractPointPrice](#)
 - [getContractTick](#)
 - [getDenominator](#)
 - [getFloatValue](#)
 - [getShowValue](#)
 - [round](#)
 - [nextPackageId](#)
 - [isTurnWon](#)
 - [getLastTransaction](#)
 - [getIndicatorValue](#)
 - [getWorkingOrder](#)
 - [getBar](#)
 - [getStrategyBarIndex](#)
- [Messages and tracing](#)
 - [sendMessage](#)
 - [trace](#)

- [Appendix A: Terminology](#)
 - [Start Time](#)
 - [End Time](#)

- [Enter End Time](#)
- [Total Count](#)
- [Win Count](#)
- [Loss Count](#)
- [Package ID](#)
- [Slippage](#)
- [Smart Order](#)
- [Timeout](#)
- [Eye](#)
- [Epsilon](#)
- [Limit](#)
- [Stop](#)
- [Denominator](#)
- [Appendix B: Helper classes](#)
 - [Class Tracer](#)
 - [Class FATS::Time](#)
 - [Class FATS::Date](#)
 - [Class XpathWrapper](#)
 - [Class OvernightDataContainer](#)
 - [Class BarId](#)
 - [Class TransactionDataWrapper](#)
 - [Class BarDataWrapper](#)
 - [Class ActionDataWrapper](#)
 - [Enumeration MARKET_DIRECTION](#)
 - [Class SmartParams](#)
 - [Class family Predicate](#)
- [Appendix C: Strategy XML example](#)

Callbacks

void onInitParameters(XPathWrapper &xpath)

The purpose of this callback is to retrieve the parameters of the strategy from the XML file. It is called once in order to set strategy parameters and constants when Strategy Runner framework initializes the strategy. Its default implementation reads and/or initializes a number of standard parameters. You have to include explicit call of the base class default implementation in your overridden implementation. You should set values of those strategy data members that stay constant throughout the trading day in *onInitParameters*.

Parameters:

- [XPathWrapper &xpath](#) - contains the strategy parameters, defined in the Strategy section of the strategy XML file.

void onRegisterBars(XPathWrapper &xpath)

This callback is called once when Strategy Runner framework initializes the strategy. This function should include [registerBar](#) function calls for each particular bar.

Parameters:

- [XPathWrapper &xpath](#) - contains the bars' parameters, defined in the Bar section of the strategy XML file.

bool onAddIndicators(XPathWrapper &xpath)

This callback is called once when Strategy Runner framework initializes the strategy. Use [addIndicator](#) function to add each particular indicator.

Parameters:

- [XPathWrapper &xpath](#) - contains the indicators' parameters, defined in the Indicator section of the strategy XML file.

Return value: **true** if all indicators were added successfully

bool onRestoreOvernight(XPathWrapper& xpath)

This callback is called when Strategy Runner framework starts the strategy. The callback initializes those strategy data items that were saved by the end of the previous day in order to restore the strategy state on the next day. Strategy Runner framework calls this function only if strategy was defined as overnight in its XML definitions file. **Note:** Special care should be taken in case the strategy is executed for the first time and still has no overnight data.

Parameters:

- [XPathWrapper& xpath](#) - contains the overnight parameters in XML form.

Return value: **boolean** (unused)

bool onInit(const TransactionDataWrapper& data)

This callback is called at the strategy's [Start Time](#) or when the strategy is started manually through the Console. You should set initial values for those strategy variable data members that may change during the trading day in *onInit*.

Parameters:

- [const TransactionDataWrapper& data](#) - start time transaction

Return value: **boolean** (unused)

bool onStart(const TransactionDataWrapper& data)

This callback is called when the [Start Time](#) is reached. Its default implementation consists of the call to [onEnterPositions](#) below. Should it be overridden, it is **not** necessary to explicitly call *onStart* of the base class.

Parameters:

- [const TransactionDataWrapper& data](#) - current transaction.

Return value: **boolean** (unused)

bool onEnterPositions(const TransactionDataWrapper& data)

This callback is called either immediately after the [onInit](#) function or each time the strategy wants to enter a new position (after [onClosePositions](#)). Within this function you should implement the way in which the strategy enters positions (send order, set alerts, etc).

Parameters:

- [const TransactionDataWrapper& data](#) - current transaction.

Return value: **boolean** (unused)

bool onOpenPositions(const ActionDataWrapper& actionWrapper)

This callback is called when an order has been filled, and the strategy has open positions.

Parameters:

- [const ActionDataWrapper& actionWrapper](#) - the filled order.

Return value: **boolean** (unused)

bool onClosePositions(const ActionDataWrapper& actionWrapper)

This function is called when an order has been filled, and the strategy is flat (there is no open positions).

Parameters:

- [const ActionDataWrapper& actionWrapper](#) - the filled order

Return value: **boolean** (unused)

bool onAlert(const ActionDataWrapper& actionWrapper)

This function is called after requested alert is triggered. Available alerts: Time alert, Time Transaction alert, Indicator alert, Price alert.

Parameters:

- [const ActionDataWrapper& actionWrapper](#) - the alert.

Return value: **boolean** (unused)

bool onBar(const BarId &barId, const BarDataWrapper& bar)

This function is called after a bar is closed (end-of-bar event).

Parameters:

- [const BarId &barId](#) - bar id.
- [const BarDataWrapper& bar](#) - the last bar.

Return value: **boolean** (unused)

bool onWaitAfterEntrance(const ActionDataWrapper& actionWrapper)

This callback is called when strategy has changed the number of open positions. This callback is followed by timeout before calling to [onOpenPositions](#).

Parameters:

- [const ActionDataWrapper& actionWrapper](#) - the filled order that has opened the position(s).

Return value: **boolean** (unused)

bool onWaitAfterWin(const ActionDataWrapper& actionWrapper)

This callback is called when strategy has closed its positions, and the last trading cycle was completed with profit. This callback is followed by timeout before calling to [onEnterPositions](#).

Parameters:

- [const ActionDataWrapper& actionWrapper](#) - the order that has closed the position(s).

Return value: **boolean** (unused)

bool onWaitAfterLoss(const ActionDataWrapper& actionWrapper)

This callback is called when strategy has closed its positions, and the last trading cycle was completed with loss. This callback is followed by timeout before calling to [onEnterPositions](#).

Parameters:

- [const ActionDataWrapper& actionWrapper](#) - the order that has closed the position(s).

Return value: **boolean** (unused)

char* onSaveOvernight()

This callback is called at the stop time. Its purpose is to save those of strategy state variables that would be restored on the next day. Strategy Runner framework calls this function only if strategy was defined as overnight in its XML definitions file.

Parameters:

- **char*** - pointer to an [OvernightDataContainer](#) object that contains the saved data.

bool onStop()

This callback is called at the stop time.

Trading Functions

void registerBar(XPathWrapper &xpath, const std::string &name, long contractId, BarId &barId)

This function registers the strategy to receive end-of-bar events for the bar.

Parameters:

- [XPathWrapper &xpath](#) - contains the bar parameters, defined in the Bar section of the strategy XML file.
- **const std::string &name** - bar name.
- **long contractId** - contract id.
- [BarId &barId](#) - id of the new bar (initialized inside the function).

long addIndicator(const char* name, XPathWrapper &xpath)

This function registers calculation of the specified indicator to the strategy.

Note: This function only registers indicator. Call [startIndicator](#) to start indicator calculations.

Parameters:

- **const char* name** - indicator name.
- [XPathWrapper &xpath](#) - contains the indicator parameters, defined in the Indicator section of the strategy XML file.

Return value: **long** - indicator id

void startIndicator(long id)

This function starts continuous indicator calculation as defined by given id. It should be called at exact moment when indicator calculation should commence. Usually, calls to *startIndicator* are performed in [onInit](#) or [onStart](#) which coincide with strategy start time.

long sendOrder(ActionDataWrapper::ACTION_TYPE action_type, double price, double price2, long count, long packageId);

This function sends an order.

Parameters:

- [ActionDataWrapper::ACTION_TYPE](#) - order type
- **double price** - order price (n/a on market orders)
- **double price2** - used only for Stop Limit orders, otherwise equals to **price**
- **long count** - number of lots
- [long packageId](#) - package id

Return value: **long** - sent order id > 0 on success, -1 otherwise

long sendSmartOrder(ActionDataWrapper::ACTION_TYPE action_type, double price, double price2, long count, long packageId, double slippage = 0.0, long timeout = -1)

This function sends a [smart order](#).

Parameters:

- [ActionDataWrapper::ACTION_TYPE](#) - order type
- **double price** - order price (n/a on market orders)
- **double price2** - used only for Stop Limit orders, otherwise equals to **price**
- **long count** - number of lots
- [long packageId](#) - package id
- [double slippage](#) - slippage
- [long timeout](#) - timeout

Return value: **long** - sent order id > 0 on success, -1 otherwise

long modifyOrder (long orderId, double price, double price2, long count)

This function modifies the working order.

Parameters:

- **long orderId** - working order id
- **double price** - new order price
- **double price2** - new order's second price
- **long count** - new count

Return value: **long** - order id > 0 on success, -1 otherwise

long cancelOrder(long orderId)

This function cancels a specific order.

Parameters:

- **long orderId** - order id

Return value: **long** - order id

void cancelPackage(long packageId)

This function cancels all orders and alerts in the package.

Parameters:

- **long packageId** - package id

void sendEnterLimitPositions(double price, double epsilon, const SmartParams smart = NON_SMART_ORDER, long lots = 0, long packageId = -1)

This function sends two limit orders to enter new positions.

Parameters:

- **double price** - start price
- **double epsilon** - epsilon from start price (in percents)
- **const SmartParams smart** - attributes for smart order
- **long lots** - number of lots (if 0, use [default position number](#))
- **long packageId** - package id (if -1, use system package id)

void sendEnterLimitPositionsPts(double price, double buyOffsetPts, double sellOffsetPts, const SmartParams smart = NON_SMART_ORDER, long lots = 0, long packageId = -1)

This function sends two limit orders to enter new positions.

Parameters:

- **double price** - start price
- **double buyOffsetPts** - offset for buy-limit order from start price (in points)
- **double sellOffsetPts** - offset for sell-limit order from start price (in points)
- **const SmartParams smart** - attributes for smart order
- **long lots** - number of lots (if 0, use [default position number](#))
- **long packageId** - package id (if -1, use system package id)

void sendEnterStopPositions(double price, double epsilon, const SmartParams smart = NON_SMART_ORDER, long lots = 0, long packageId = -1)

This function sends two stop orders to enter new positions.

Parameters:

- [double price](#) - start price
- [double epsilon](#) - epsilon from start price (in percents)
- [const SmartParams smart](#) - attributes for smart order
- **long lots** - number of lots (if 0, use [default position number](#))
- [long packageId](#) - package id (if -1, use system package id)

void sendEnterStopPositionsPts(double price, double buyOffsetPts, double sellOffsetPts, const SmartParams smart = NON_SMART_ORDER, long lots = 0, long packageId = -1)

This function sends two stop orders to enter new positions.

Parameters:

- [double price](#) - start price
- **double buyOffsetPts** - offset for buy-stop order from start price (in points)
- **double sellOffsetPts** - offset for sell-stop order from start price (in points)
- [const SmartParams smart](#) - attributes for smart order
- **long lots** - number of lots (if 0, use [default position number](#))
- [long packageId](#) - package id (if -1, use system package id)

void sendClosePositions(double price, double limit, double stop, const SmartParams smart = NON_SMART_ORDER, long lots = 0, long packageId = -1)

This function sends orders to close the already open positions in accordance with stop loss and profit limit values (in percents).

Parameters:

- **double price** - open position price
- [double limit](#) - limit parameter (in percents)
- [double stop](#) - stop parameter (in percents)
- [const SmartParams smart](#) - attributes for smart order
- **long lots** - number of lots (if 0, use open position number)
- [long packageId](#) - package id (if -1, use system package id)

void sendClosePositionsPts(double price, double limitOffsetPts, double stopOffsetPts, const SmartParams smart = NON_SMART_ORDER, long lots = 0, long packageId = -1)

This function sends orders to close the already open positions in accordance with stop loss and profit limit offset values (in points).

Parameters:

- **double price** - open position price
- **double limitOffsetPts** - limit offset (in points)
- **double stopOffsetPts** - stop offset (in points)
- [const SmartParams smart](#) - attributes for smart order
- **long lots** - number of lots (if 0, use open position number)
- [long packageId](#) - package id (if -1, use system package id)

long sendCloseLimitPositions(double price, double limit, const SmartParams smart = NON_SMART_ORDER, long lots = 0, long packageId = -1)

This function sends a limit order to achieve profit target.

Parameters:

- **double price** - open position price
- **double limit** - limit parameter (in percents)
- **const SmartParams smart** - attributes for smart order
- **long lots** - number of lots (if 0, use open position number)
- **long packageId** - package id (if -1, use system package id)

Return value: **long** - limit order id

long sendCloseLimitPositionsPts(double price, double limitOffsetPts, const SmartParams smart = NON_SMART_ORDER, long lots = 0, long packageId = -1)

This function sends a limit order to achieve profit target (in points).

Parameters:

- **double price** - open position price
- **double limitOffsetPts** - limit offset (in points)
- **const SmartParams smart** - attributes for smart order
- **long lots** - number of lots (if 0, use open position number)
- **long packageId** - package id (if -1, use system package id)

Return value: **long** - limit order id

long sendCloseStopPositions(double price, double stop, const SmartParams smart = NON_SMART_ORDER, long lots = 0, long packageId = -1)

This function sends a stop order in accordance with stop loss value.

Parameters:

- **double price** - open position price
- **double stop** - stop parameter (in percents)
- **const SmartParams smart** - attributes for smart order
- **long lots** - number of lots (if 0, use open position number)
- **long packageId** - package id (if -1, use system package id)

Return value: **long** - stop order id

long sendCloseStopPositionsPts(double price, double stopOffsetPts, const SmartParams smart = NON_SMART_ORDER, long lots = 0, long packageId = -1)

This function sends a stop order in accordance with stop loss offset (in points).

Parameters:

- **double price** - open position price
- **double stopOffsetPts** - stop offset (in points)
- **const SmartParams smart** - attributes for smart order
- **long lots** - number of lots (if 0, use open position number)
- **long packageId** - package id (if -1, use system package id)

Return value: **long** - stop order id

long closeAllPositions()

This function closes all strategy open positions.

void cancelAllActions()

This function cancels all alerts and orders.

void cancelAllOrders()

This function cancels all strategy orders.

void stopTrader()

This function closes all positions, cancels all actions, and stops the strategy.

Alert Functions

long setTimeAlert(FATS::Time& time, long& id, long contractId = -1, long packageId = SYSTEM_PACKAGE_ID)

This function sets a time alert. This alert is triggered when the specified time is reached.

Parameters:

- [FATS::Time &time](#) - time
- **long& id** - alert id (assigned inside the function)
- **long contractId** - contract id. If -1, use working contract id.
- [long packageId](#) - package id

Return value: **long** - alert id

long setTimeOffsetAlert(long offset, long& id, long contractId = -1, long packageId = SYSTEM_PACKAGE_ID)

This function sets an offset time alert (in seconds). Alert is triggered after the given time (offset) has passed since the moment when this function was called.

Parameters:

- **long offset** - offset in seconds
- **long& id** - alert id (assigned inside the function)
- **long contractId** - contract id. If -1, use working contract id.
- [long packageId](#) - package id

Return value: **long** - alert id

long setTimeTransactionAlert(FATS::Time &timeAlert, long& id, long contractId = -1, long packageId = SYSTEM_PACKAGE_ID)

This function sets a transaction time alert. This alert is triggered when the system receives the first transaction after the specified time.

Parameters:

- [FATS::Time &timeAlert](#) - transaction time
- **long& id** - alert id (assigned inside the function)
- **long contractId** - contract id. If -1, use working contract id.
- [long packageId](#) - package id

Return value: **long** - alert id

long setPriceAlert(Predicate* p, long contractId = -1, long packageId = SYSTEM_PACKAGE_ID)

This function sets a price alert. When the predicate condition is satisfied, this alert is triggered.

Parameters:

- [Predicate p](#) - price condition
- **long contractId** - contract id. If -1, use working contract id.
- [long packageId](#) - package id

Return value: **long** - alert id

long setPnLAlert(Predicate* p, long userData = 0, long packageId = SYSTEM_PACKAGE_ID)

This function sets a P&L alert. When the predicate condition on P&L is satisfied, this alert is triggered.

Parameters:

- **Predicate p** - P&L condition
- **long userData** - user defined data
- **long packageId** - package id

Return value: **long** - alert id

long setIndicatorAlert(long indicator_id, Predicate* p, long lUserData = 0, long contractId = -1, long packageId = SYSTEM_PACKAGE_ID)

This function sets an alert on value of specified indicator. When the predicate condition is satisfied, this alert is triggered.

Parameters:

- **long indicator_id** - id of indicator of interest
- **Predicate p** - alert condition
- **long contractId** - contract id. If -1, use working contract id.
- **long packageId** - package id

Return value: **long** - alert id

void cancelTimeAlert(long id)

This function cancels specified time alert.

void cancelPriceAlert(long id)

This function cancels specified price alert.

void cancelPnLAlert(long id)

This function cancels specified PnL alert.

long cancelIndicatorAlert(long id_action)

This function cancels specified indicator alert.

Query Functions

FATS::Time getStartTime()

This function returns the starting time of the strategy.

FATS::Time getEndTime()

This function returns the end time of the strategy.

double getLimit()

This function returns the [limit](#) parameter.

double getStop()

This function returns the [stop](#) parameter.

long getOpenPositions()

This function returns the number of currently opened positions. The sign of the number refers either to long (positive) or to short (negative) positions.

long getWaitAfterEntrance()

This function returns strategy timeout (in seconds) after the strategy entered position (the trading cycle's entry order has been filled) and before calling [onOpenPositions](#) callback.

long getWaitAfterWin()

This function returns strategy timeout (in seconds) after the strategy closed position with profit (the trading cycle's profit limit order has been filled) and before calling [onEnterPositions](#) callback.

long getWaitAfterLoss()

This function returns strategy timeout (in seconds) after the strategy closed position with loss (the trading cycle's stop loss order has been filled) and before calling [onEnterPositions](#) callback.

long getContractId()

This function returns the working contract id of the strategy.

long getContractCountParam()

This function returns the default number of lots for order sending. Set the number through the Console.

[MARKET DIRECTION](#) getMarketDirection()

This function returns the market direction (bullish, bearish or none). Set the type through the Console.

double getCurrentPnL()

This function returns the current P&L (profit and loss) in points.

double getCurrentPnLPrice()

This function returns the current P&L in currency.

[FATS::Time](#) getCurrentTime(long contractId = -1)

This function returns the current time for the specified contract. If a contract is not specified, the working contract is used.

FATS::Date getCurrentDate(long contractId = -1)

This function returns the current date for the specified contract. If a contract is not specified, the working contract is used.

double getContractPointPrice(long contractId)

This function returns the price of a one point for the specified contract.

double getContractTick(long contractId)

This function returns the minimal part of point (tick) for the specified contract.

double getDenominator(long contractId)

This function returns the [denominator](#) for the given contract.

double getFloatValue(double showValue, [double denominator](#))

This function converts the price from displayed representation to actual value.

double getShowValue(double floatValue, [double denominator](#))

This function converts the price from actual value to displayed representation.

double round(double price)

This function rounds the price according to the contract tick.

long nextPackageId()

This function returns a unique [package id](#).

bool isTurnWon()

This function returns **true** if last transaction round trip (trading cycle) was with profit, and **false** otherwise.

[TransactionDataWrapper](#) getLastTransaction(long contractId)

This function returns the most recently occurred transaction for specified contract.

double getIndicatorValue(long contractId, long indicatorId, long time)

This function returns the value of the specified indicator at given time.

[ActionDataWrapper](#) getWorkingOrder(long orderId)

This function returns a working order according to its id.

const BarDataWrapper* getBar(const BarId &barId, long since)

This function returns the previous bar.

Parameters:

- [const BarId & barId](#) - bar id
- **long since** - number of bars from the last bar. If since=0 get the last bar.

Return value: [const BarDataWrapper*](#) - pointer to a bar object. If NULL - bar doesn't exist.

int getStrategyBarIndex([const BarId & barId](#))

This function returns the index of the given bar since strategy started.

Messages and tracing

Strategy may send messages to the Console, or write traces and logs to file/database.

`void sendMessage(const char* msg)` sends the given message to the Console.

`BaseStrategyClass::_pTracer` member variable of type [Tracer](#) allows you to insert an arbitrary message into strategy trace file or strategy trace data base.

Appendix A: Terminology

- **Start Time** - time at which the strategy starts executing; defined in the VTrader/Time section of the strategy XML file.
- **End Time** - time at which the strategy stops executing; defined in the VTrader/Time section of the strategy XML file.
- **Enter End Time** - time after which the strategy may not send any new orders; defined in the VTrader/Time section of the strategy XML file.
- **Total Count** - number of transaction round trips (buy-sell or sell-buy transaction pairs) that could happen during the day. Defined in VTrader/Counts section of the strategy XML file.
- **Win Count** - number of transaction round trips with positive outcome (wins) that could take place during the day. Defined in VTrader/Counts section of the strategy XML file.
- **Loss Count** - number of transaction round trips with negative outcome (losses) that are allowed to happen during the day. Defined in VTrader/Counts section of the strategy XML file.
- **Package ID** - A package groups orders and alerts together. Whenever one of the group members is completed other members are canceled automatically. Filling of two or more orders with the same package id causes the strategy to fall into an error state. A package is specified by the package id. If the strategy wants to send an ungrouped order, each order must be sent with a unique package id. A unique package id is acquired by calling [nextPackageId\(\)](#) function.
- **Slippage** is the change in price from the time an order is sent to the time it is executed. Slippage may occur in market orders and stop orders. The desired slippage is defined in [Smart Orders](#).
- **Smart Order** is a special type of order used in Strategy Runner. There are three types of Smart Orders: **smart market**, **smart stop** and **smart limit**.
 - **Smart market** is an order which tries to avoid slippage in the market. Instead of a market order, it sends a limit order at the current price +/- the desired [slippage](#) and waits the desired [timeout](#). If this order is not executed by timeout, it is replaced with a regular market order.
 - **Smart stop** is an order which tries to avoid slippage in the market. Instead of a stop order, it sends a stop limit order. When the price reaches the stop price, the order becomes the limit order at the current price +/- the desired [slippage](#) and waits the desired [timeout](#). If this order is not executed by timeout, it is replaced with a regular market order.
 - **Smart limit** is an order which sends a regular limit order and waits for the limit price to be reached. If the limit price is reached, but the order has not been executed or has been executed partially, the system waits for the desired [timeout](#), and sends a market order to fulfill the unexecuted lots.
- **Timeout** is the waiting time between the completion of the original order and the sending of the market order, which is sent instead of the original order. The desired timeout is defined in [Smart Orders](#).
- **Eye** is the price at which initial calculations are made if necessary. In most cases it is the price at the current time.
- **Epsilon** is the difference between the [eye](#) price and the price at which an order enters a position.
- **Limit** is the difference between the price at which the position is opened and the profit limit price. Strategy Runner uses this parameter to calculate the price of the limit order, which is sent to close the position.
- **Stop** is the difference between the price at which the position is opened and the stop loss price. Strategy Runner uses this parameter to calculate the price of the stop loss order, which is sent to close the position.

- **Denominator** - is the divider, which normalizes the displayed price to its decimal equivalent. Not all contracts use denominator. The example of contract which do use denominator is **TBOND** with denominator 32.

Appendix B: Helper Classes

- **Class Tracer** - strategy tracing

Each strategy that inherits from **BaseStrategyClass** contains a protected member variable **_pTracer**, which is pointer to the object of class **Tracer**. Declaration of class **Tracer** can be found in **CoreStrategies\StrategyFeatures\Tracer.h**. This class is used to store user specified messages (strategy trace information) in file located in **C:\Program Files\StrategyTuner2\bin\UserTrace\<user name>** (or in **C:\Program Files\StrategyTuner2\API\bin_debug\UserTrace\<user name>** if you run the strategy in debug environment).

You must call **BaseStrategyClass::onInitParameters()** in [onInitParameters](#) of your strategy or make sure that it is called down the inheritance tree. Otherwise **_pTracer** would remain undefined becoming a potential cause of strategy failure.

There are two ways to create trace records:

- **void add(TraceLevel l, char *message, ...)** adds to existing trace records or starts a new record if none existed yet. At the end of sequence of calls to **add()** you need to call **void flush()** which causes the record to be written to disk.
- **void trace(TraceLevel l, char *message, ...)** flushes the currently open record to disk if there is any. Creates a new one and immediately flushes it to disk.

TraceLevel is a strategy parameter. It belongs to the Strategy section of parameter file (VTrader->Strategy). Possible values of TraceLevel are "NONE", "LOW", "PARAM", "BAR", "ORDER", "POSITION", "ALWAYS". Additionally, you can switch on or off all traces by means of parameter TraceToFile. It can be found in the same section along with TraceLevel. **TraceLevel** can be one of (from low to high):

- LELEL_NONE
 - no trace is produced.
- LEVEL_LOW
 - low level information, significant only if verbose tracing is necessary.
- LEVEL_PARAM
 - echo output of strategy parameters
- LEVEL_BAR
 - periodic information, such as bar open/close, high/low prices
- LEVEL_ORDER

- order specific data - order type, price, etc
- LEVEL_POSITION
 - positions specific data - time, price, number of lots of position being opened or closed
- LEVEL_ALWAYS
 - important information that has to be present no matter what. For example, error messages of the strategy.

If value of `TraceLevel` parameter is lower or equal (less or equal) than that given in `add()` or `trace()` then the trace information is added to the record. Otherwise, it is omitted.

For sake of simplicity both `add()` and `trace()` have overloaded versions that do not take `TraceLevel` parameter. Their trace information is always added to the record:

- `void add(char *message, ...)`
- `void trace(char *message, ...)`
- **Class FATS::Time** - time representation

Useful functions:

- `long h()` - return current hour (0-23)
- `long m()` - return current minute (0-59)
- `long s()` - return current second (0-59)
- `long seconds()` - return seconds since 00:00:00
- **Class FATS::Date** - date representation

Useful functions:

- `long year()` - return current year (yyyy)
- `long month()` - return current month (1-12)
- `long day()` - return current day (1-32)
- `long days()` - return days since year start (1-366)
- **Class XPathWrapper** - used to retrieve attribute names from strategy XML file in order to set strategy parameters.

Useful functions:

- `const char * asString(const char *expression, char* buffer, long length)` - returns the value of the **expression** in a string format.

Parameters:

- **const char *expression** - XPath expression, for instance `Strategy/@name` will get the value of strategy's attribute `name`
- **char *buffer** - holds the retrieved value
- **long length** - length of the **buffer**

Return value:

- **const char*** - pointer to the beginning of the **buffer**
 - **long asLong(const char *expression)** - returns the value of the **expression** as long
 - **double asDouble(const char *expression)** - returns the value of the **expression** as double
 - **FATS::Time asTime(const char *expression)** - returns the value of the **expression** as time
 - **bool asBool(const char *expression)** - returns the value of the **expression** as boolean
- **Class OvernightDataContainer** - holds the data that is saved by the end of the day and restored on the next day.

Useful functions:

- **bool add(const char* name, T value)** - saves an item of type **T** under the key **name**. Return value is unused.
 - **T get(const char* name, T error)** - returns value of an item of type **T** under the key **name**. In case of a problem, returns the default **error** value.
- **Class BarId** - bar id

Useful functions:

- **BarId::BARTYPE type()** - returns one of following:
 - **ERR** - for error
 - **TIME** - for time bars
 - **VOLUME** - for volume bars
 - **TRANSACTION** - for transaction bars
 - **DAILY** - for daily bars
 - **long period()** - returns bar's period.
- **Class TransactionDataWrapper** contains transaction data and attributes

Useful functions:

- **long contract_id()** - returns contract id
 - **long time()** - returns transaction time
 - **double value(int index)** - returns the transaction data according to the **index**:
 - 0 for price
 - 1 for volume
- **Class BarDataWrapper** contains bar data and attributes

Useful functions:

- **long contract_id()** - returns the contract id
 - **double open()** - returns bar open price
 - **double close()** - returns bar close price
 - **double high()** - returns the highest price within the bar
 - **double low()** - returns the lowest price within the bar
 - **int transNum()** - returns the number of transactions within the bar
 - **int volume()** - returns the volume within the bar
 - **int index()** - returns sequential index of the bar
 - **long openTime()** - returns bar open time
 - **long closeTime()** - returns bar close time
- **Class ActionDataWrapper** contains action data and attributes

Useful functions, members and constants:

- **enum ACTION_TYPE** - action type:
 - **ACTION_NONE**
 - **ALERT_TIME**
 - **ALERT_TIME_TRANS**
 - **BUY_LIMIT**
 - **SELL_LIMIT**
 - **BUY_STOP**
 - **SELL_STOP**
 - **BUY_STOP_LIMIT**
 - **SELL_STOP_LIMIT**
 - **BUY_MARKET**
 - **SELL_MARKET**
- **enum ORDER_STATUS** - order status:
 - **STATUS_QUEUED**
 - **STATUS_SEND**
 - **STATUS_WORKING**
 - **STATUS_CANCELED**
 - **STATUS_REJECTED**
 - **STATUS_FILLED**
 - **STATUS_PARTFILLED**
 - **STATUS_UNKNOWN**
- **bool isOrder()** - returns true if the action is an order, false otherwise
- **bool isLong()** - returns true if long position, false otherwise
- **long _contract_id** - contract id
- **long _id** - action id
- **long _time** - time the action is sent
- **long _fill_time** - time the action is filled (if applicable)
- **double _price** - price of the action
- **double _price2** - price2 of the action (if applicable)
- **double _fill_price** - price at which the action is filled (if applicable)
- **long _count** - number of lots for the action (if applicable)
- **ACTION_TYPE _action_type** - action type
- **ORDER_STATUS _order_status** - action status
- **TransactionDataWrapper _trans** - corresponding transaction
- **Enumeration MARKET_DIRECTION**
 - **MARKET_BULLISH**
 - **MARKET_BEARISH**
 - **MARKET_ANY**
- **Class SmartParams** contains attributes for [smart order](#)

Useful functions and defines:

- **SmartParams()** - set attributes for non-smart order.
- **SmartParams(long timeout)** - set attributes for smart order with given [timeout](#) and zero [slippage](#).
- **SmartParams(long timeout, double slippage)** - set attributes for smart order with given [timeout](#) and [slippage](#).
- **bool smart()** - returns **true** for smart order, **false** otherwise.
- **long timeout()** - returns [timeout](#) for smart order.
- **double slippage()** - returns [slippage](#) for smart order.
- **#define NON_SMART_ORDER (SmartParams())** - non smart order.


```

>
<Bars> : bar section
  <Bar
    name='mybar'      : bar name
    type='time'       : bar type. Options: daily, time, transaction,
volume
    period='600'      : bar period (for all except daily bars)
                        : for time bars - in seconds
                        : for transaction bars - in ticks
                        : for volume bars - in volume

    : previous days sub-section - mandatory for daily bars,
optional for other
    numOfPrevBars='200'

    startTimeOfPrevDays='08:30:00'
    endTimeOfPrevDays='15:15:00' />
  </Bar>
</Bars>

<Indicators> : indicator section
  <Indicator
    name='MyIndicator1'
    packageName='SampleIndicators'
    templateName='MovingAverage'
    period='30'
    show='1' >
  </Indicator>

</Indicators>

</Strategy>

</VTrader>

```