

Strategy Runner API

V 2.0

Development Guide

Table of Contents

Chapter 1. Overview	1
Developing with Strategy Runner API ...	Error! Bookmark not defined.
Back-Testing with Strategy Tuner	Error! Bookmark not defined.
Strategy Analysis.....	Error! Bookmark not defined.
Chapter 2. Getting Started.....	2
Installation	2
Verifying Installation of Strategy Runner API	2
Adding a Strategy Project to Strategy Runner API.....	2
Chapter 3. Basic Concepts.....	5
Framework–Strategy Relations	5
Events and Callbacks.....	5
Service Functions	6
Chapter 4. Strategy Development.....	7
Init, Start and Stop	7
Entering and exiting positions.....	8
Alerts	8
Chapter 5. Strategy Example - Trend	10
Coding the strategy.....	11
Strategy Parameters file	13

Chapter 1. Overview

Strategy Runner API is a C++ Strategy-Oriented API, for rapid development of reliable trading strategies with virtually unlimited degree of complexity and flexibility of strategy.

The basic concept of strategy development with Strategy Runner concept is **event-oriented** approach, which allows you to take immediate actions once **trading event** occurs, i.e. notification about order being filled, canceled, position opened, indicators threshold is reached, time expired, etc.

The way the strategy is define and executed follows closely the natural way in which a human trader would go about his or her trading day manually.

The strategy execution model works as following. During trading period, strategy can register for different kind of trading events through Strategy Runner framework. In addition, strategy keeps number of callback functions for each event type. When an event occurs, Strategy Runner triggers call of the corresponding callback, assuring propagation of the strategy execution.

In this document we introduce basic concepts of strategy development with Strategy Runner API, describe how to develop trading strategies with Strategy Runner API, and show how virtually any trading method can be translated to easily readable C++ code with the help of Strategy Runner API.

Chapter 2. Getting Started

Installation

Read Strategy Tuner – Installation Guide document.

Verifying Installation of Strategy Runner API

Open MS Visual Studio with Strategy Runner API project and compile it. Make sure you can run the strategies supplied with the API and that the environment is working properly by the following steps:

1. Go to ***Start*** → ***Programs*** → ***StrategyTuner2***
2. Click on ***Tuner Server***. This action opens command line window with Strategy Tuner Server running.
3. Click on ***Tuner Console*** in order to open the console application. Enter your login and password, which were supplied in the confirmation e-mail you have received.
4. In the Console, add the ***MINI NSDQ*** contract and choose a strategy that belongs to the vendor ***StrategyRunner***. Strategies are grouped according to their vendors. Strategies supplied with this installation belong to vendor ***StrategyRunner***. Then choose strategy running period and run the strategy. See detailed explanation in ***Strategy Tuner User Manual***.

If the execution was successful, we can move on to the next step – creating new project for strategy development.

Adding a Strategy Project to Strategy Runner API

A strategy in Strategy Runner framework is a C++ class derived from BaseStrategyClass. Strategy project may contain any number of strategies, generally from the (same) strategy vendor, deployed in strategy DLL.

To create new strategy project the following should be done:

1. Open the Strategy Runner API workspace.
Go to ***Start*** → ***Programs*** → ***StrategyTuner2*** and click on ***API Project*** or go to

C:\Program Files\StrategyTuner2\API and double click on *StrategyRunnerAPI.dsw*. MS Visual C++ project will be opened.

2. Create a new project for your strategy in the workspace.
Right-click on the root element in workspace window, choose *Add New Project to Workspace ...* in the pop-up menu.
3. Choose *project type* as *Win32 DLL*, choose the location of the project as *C:\Program Files\StrategyTuner2\API\UserStrategies* and name the project (for example *MyStrategies*). The new directory *C:\Program Files\StrategyTuner2\API\UserStrategies\MyStrategies* will be created with the new project file *MyStrategies.dsp*.
4. Prepare the settings of the project:
 - a. Open *Project*→*Dependencies*. Set the project to be dependant on the project *CoreStrategies*
 - b. Open *Project*→*Settings*, choose *C/C++* tab.
 - i. In *Preprocessor* category set additional include directories to be *C:\Program Files\StrategyTuner2\API,C:\Program Files\StrategyTuner2\API\UserStrategies\MyStrategies*.
 - ii. In *Code Generation* category set *Use run-time library* to be *Debug Multithreaded DLL* for debug compilation and *Multithreaded DLL* for release compilation.
 - c. Open *Project*→*Settings*, choose *Link* tab.
 - i. In *Input* category add *XPath_1.lib xerces-c_2.lib XalanSourceTree_1.lib UtilLib.lib* to the *Object/library modules*. Also, set *Additional library path* to be *C:\Program Files\StrategyTuner2\API\Xalan\Release,C:\Program Files\StrategyTuner2\API\xalan\xml-xerces\lib,C:\Program Files\StrategyTuner2\API\bin_debug* for debug compilation and *C:\Program Files\StrategyTuner2\API\Xalan\Release,C:\Program Files\StrategyTuner2\API\xalan\xml-xerces\lib,C:\Program Files\StrategyTuner2\bin* for release compilation.
 - ii. In *General* category set *Output file name* to be *..\..\bin_debug\MyStrategies.dll* for debug compilation and *..\..\bin\MyStrategies.dll* for release compilation.
5. There are provided template files in *C:\Program Files\StrategyTuner2\API\UserStrategies\EmptyProject* directory. Copy all the files from this to your project directory *C:\Program Files\StrategyTuner2\API\UserStrategies\MyStrategies*.
6. In your project directory:
 - a. Rename the file *NewStrategies.DEF*. The base name of this file should be the same as the deployed DLL, for this example *MyStrategies.DEF*.

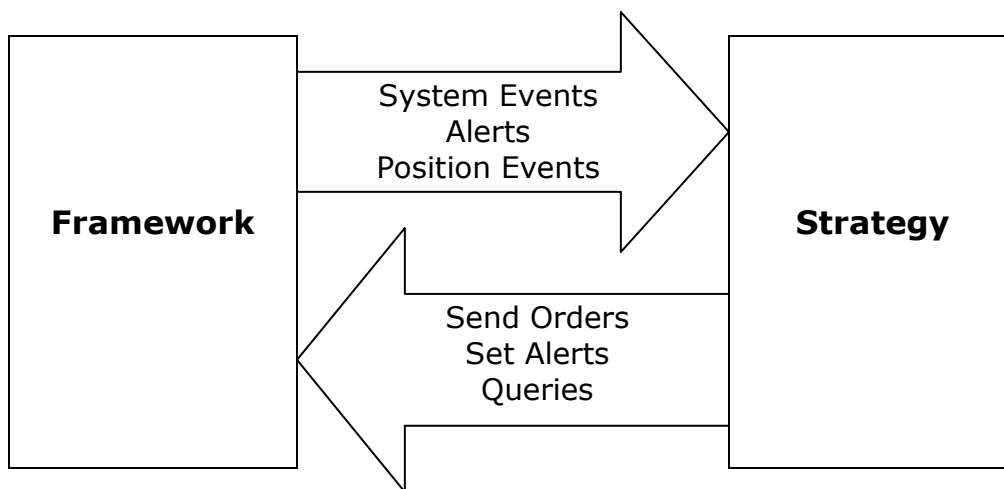
- b. Open text editor with *MyStrategies.DEF*. In the first line replace *LIBRARY* name *NEWSTRATEGIES* with name of deployed DLL, in this example with *MYSTRATEGIES*.
7. Add files *MyStrategies.DEF* and *StrategyManager.cpp* to the project. You can also use *NewClass.cpp* and *NewClass.h* templates for strategy implementation.
8. Write your own strategy as explained below. Add the files to the project according to the structure of your strategy.
9. For each written strategy edit the file *StrategyManager.cpp*:
 - a. Include the header file of the strategy: *#include "NewClass.h"*
 - b. Register strategy in *init()* function: *ADD_STRATEGY(NewClass);*

Chapter 3. Basic Concepts

Framework–Strategy *Relations*

Strategy Runner API consists of two main modules: Framework and Strategy, which calls functions of each other.

While strategy logic is implemented in functions (callbacks) of the Strategy module, Framework actually calls these functions. Framework provides Strategy object with trading facilities such as sending orders and alerts. When an order is filled or alert is triggered, trading event occurs, and Framework immediately notifies Strategy by calling corresponding callback.



Trading events and corresponding callbacks constitute the **event-oriented** approach, which assures propagation of the strategy execution.

Events and Callbacks

Trading event is a central term in strategy execution. When strategy sends order or alert, it is expecting for completion of this action. Framework manages all the strategy actions and notifies strategy upon event occurrence.

There are three types of trading events:

- **System events** are connected to management of the strategy by the Framework and occur for all strategies. For example, when strategy is supposed to be started, Framework calls to sequence of callbacks for initialization and running the strategy.
- **Alerts** are the events, the strategy registers to. Conditions for alerts are defined by strategy using framework function. Framework manages all alerts and notifies strategies when an alert's condition is satisfied. Types of alerts:
 - **Time alert** is triggered when a specified time has been reached, or a desired time interval has been ended.
 - **Price alert** is triggered when price reaches specified value.
 - **P&L alert** is triggered when strategy P&L (profit and loss) reaches specified value.
 - **Indicator alert** is triggered when indicator reaches specified value.
 - **Bar alert** is a special kind of alerts. Bars divide feeding into discrete units. *Bar event* is triggered each time one bar ends and the next starts.
- **Position change events** are connected with order filling. When order sent by the strategy is filled, the number of strategy's open positions is changed. This causes *position change event* to be triggered.

Note: the number of open positions may be either **positive** (strategy trades in long direction), **negative** (strategy trades in short direction) or **zero** (strategy is flat, or has no open positions). The absolute number of open positions means the quantity of lots.

Service Functions

Strategy logic is concentrated in strategy callbacks. In these callbacks strategy may want to access to trading facilities and to perform necessary computations. For that purpose Framework supplies the strategy with service functions of the following four types:

- **Trading functions** allow user to send an order or a group of related orders.
- **Set-Alert functions** are responsible for sending alerts.
- **Query functions** provide the status of various strategy's attributes.
- **Miscellaneous functions** provide some other services, such as tracing the strategy or sending message to the Console application.

Chapter 4. Strategy Development

All the actual strategy logic is divided in the callback functions. Each callback is linked with certain trading event. The order the callbacks are called depends on order the events are triggered.

In this section we describe basic strategy callbacks and auxiliary functions. See **C++ Programming Reference** for full list and documentation.

Init, Start and Stop

When the start time of the strategy is reached, the Framework generates first system event – start of the strategy execution. The Framework calls several strategy callbacks that complete strategy initialization and prepare the strategy for the trading:

1. **onInitParameters** – callback where you should initialize all custom strategy constant parameters.
2. **onRegisterBars** – callback where you should register bars, if used.
3. **onAddIndicators** – callback where you should register tick indicators, if used.
4. **onRestoreOvernight** – callback where you should restore overnight data, if applicable.
5. **onInit** - callback where you should initialize all strategy variables.

After the strategy is initialized, the actual execution starts. The first callback following the initialization, is called is *onStart*. It's called at strategy start time. The purpose of this callback is to perform one-time preparations, such as priming the indicators or calculation of statistics based on previous day's feedeing. If the strategy doesn't need such preparations, the default implementation of *onStart* may be used. It consists of a single call of *onEnterPositions*, explained below, which actually starts the trading cycle.

Normally strategy starts at **start time** and ends at **end time**. However, it is not always the case. Suppose, design of the strategy requires that the strategy should end after first loosing trading cycle. This and other similar setups are achieved by **counts** parameters.

There are three kinds of counts:

- **Total** – total number of trading cycles (transaction round trip) that can take place during one trading day.
- **Wins** – number of winning trading cycles allowed for a single trading day.

- **Losses** – number of losing trading cycles allowed for a single day.

Along with strategy *start time*, *end time*, and *counts* parameters, there is another important time condition of strategy execution. This is defined by **enterEnd** strategy parameter. This parameter means that after reaching *enterEnd* time, strategy is not allowed to enter any new positions, or in other words, to start new trading cycles. If design of the strategy does not include this feature, it is advisable to set *enterEnd* time to be equal to end time.

So, in fact, strategy ends when any of the following conditions becomes true:

- The strategy end time is reached.
- Number of complete trading cycles including the last one becomes equal to the total count above.
- Number of winning trading cycles reaches wins count.
- Number of losing trading cycles equals losses count.
- The strategy has no open positions (currently running trading cycles), or just has closed its position, and *enterEnd* time has been reached.

When the strategy is about to finish, the Framework generates another system event – end of the strategy execution. This causes the following callbacks to be called:

1. **onSaveOvernight** - saves those strategy variables that will be restored on the next day called at strategy stop time.
2. **onStop** – is called when strategy is finished.

Entering and exiting positions

In order to open the position strategy should send order(s). For this purpose we assign *onEnterPositions* callback. This callback is called when strategy starts and when strategy becomes flat (has closed its positions) after completing a trading cycle.

When an order is filled the *position change event* is generated. If the number of open positions differs from zero, the Framework calls to *onOpenPositions* callback. If the number of open positions equals to zero, the Framework calls to *onClosePositions* callback.

The call of *onClosePositions* marks that the current trading cycle has been ended. At this stage the Framework tries to conceive new trading cycle. First it checks stop conditions (*counts* and *enterEnd* time), and if it's allowed, the Framework calls to *onEnterPositions* callback to initiate new trading cycle.

Alerts

Strategy may set different alerts. This action is performed by calling set-alert functions:

- **setTimeAlert** sets a time alert. This alert is triggered when the specified time is reached.
- **setTimeOffsetAlert** sets an offset time alert. This alert is triggered after the given time (offset) has passed since the moment when this function was called.
- **setPriceAlert** sets a price alert. It is triggered when given price condition is satisfied.
- **setPnLAlert** sets a price alert. When given P&L condition is satisfied, this alert is triggered.
- **setIndicatorAlert** sets an alert on value of specified indicator. When given condition on the indicator value is satisfied, this alert is triggered.

When alert is triggered, the Framework calls to *onAlert* callback, which contain logic fragment for this event.

There is special kind of alert – **bar alert**. Bar divides feeding into discrete units. Strategy may register to arbitrary number of bars. Every time bar ends, bar alert is triggered, causing the Framework to call the *onBar* callback.

Each bar has open, close, high, and low prices, total volume, and number of transactions local to that bar. The following are the supported types of bars:

- **Time bars** divides the trading day into constant periods of time. They are also called time charts, bar charts, or time frames. This is the most commonly used type of bar.
- **Transaction bars**, unlike time bars, have variable duration. They divide the trading day into periods with constant number of transactions in them.
- **Volume bars** are the same as transaction bar, except that division is done by actual volume.
- **Daily bars** are the data-only type of bar as its events are never triggered. It provides high, low, open, close, volume, and number of transactions for the whole day. Daily bars are used in order to obtain this information about previous days trading.

It is possible to obtain bar information for any of these bars for the previous days.

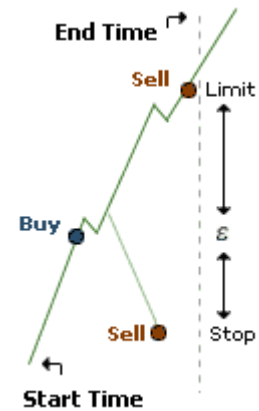
Chapter 5. Strategy Example - Trend

Let us see now a simple trend strategy. This example is supplied with *Strategy Runner API* in *SampleStrategies* project. In order to open the Strategy Runner API go to **Start→Programs→StrategyTuner2** and click on **API Project** or go to **C:\Program Files\StrategyTuner2\API** and double click on **StrategyRunnerAPI.dsw**. MS Visual C++ project will be opened. Open *SampleStrategies* project, then view files *TrendExample.cpp* and *TrendExample.h*.

A trend strategy logic is as follows.

When strategy starts, it memorizes the current price and waits until price changes to epsilon (ϵ) in any direction. If price went up, strategy predicts an up-trend and executes a buy order (open a long position), expecting to sell later at a higher price. If alternatively, the price went down, the strategy predicts a down-trend and executes a sell order (open a short position), expecting to buy later at a lower price.

Once the position has been opened, strategy sends two orders – profit limit and stop loss. The first is to determine desired profit of this trading cycle, the second is to bound potential loss if the price will change its direction.



Now let's translate the logic to sequence of events and corresponding callbacks.

onStart/enterPositions - Assume that current price is **p1 = 1000 points**, and epsilon $\epsilon = 0.5\%$. The strategy sends two stop orders with the same package ID:

- **sell stop** at **p2 = p1 * (1 + ϵ) = 1000 * (1+0.005) = 1005 points**, and
- **buy stop** at **p3 = p1 * (1 - ϵ) = 1000 * (1 - 0.005) = 995 points**.

If the price rises by epsilon, sell stop order will be filled, starting up-trend trading cycle. If the price falls by epsilon, buy stop order will be filled, starting down-trend trading cycle. The same package ID guarantees that once one of these orders is filled, the second is cancelled automatically.

onOpenPositions – Once stop order was filled, strategy send two orders with the same package ID to determine bracket (define desired profit and allowed loss). Assume that **profit limit** is **2%** and **stop loss** is **1%**. Consider long position. Strategy sends two orders:

- **sell limit** at $p4 = p3 * (1 + \text{limit}) = 995 * (1 + 0.02) = 1014.9$ points which will be round to **1015.0 points**, and
- **sell stop** at $p5 = p3 * (1 - \text{stop}) = 995 * (1 - 0.01) = 985.05$ points which will be round to **985 points**.

onClosePositions – When one of the bracket orders is filled, strategy closes its position and finishes trading cycle. Actually there is nothing to do in this callback. The *onEnterPositions* callback will be automatically after this callback to conceive new trading cycle (if none of stop condition hold)

Coding the strategy

Any strategy developing within Strategy Runner API should inherit the *BaseStrategyClass* class, which defines all callbacks and auxiliary functions.

Strategy Header C++ file

```
#include "CoreStrategies/BaseStrategies/BaseStrategyClass.h"

class TrendExample : public BaseStrategyClass
{
public:
    TrendExample(VirtualTrader *pVTrader) : BaseStrategyClass(pVTrader) {}
    virtual ~TrendExample() {}

    //always declare your strategy with the DECLARE_STRATEGY macro
    DECLARE_STRATEGY( TrendExample );

    // callback functions to be implemented
    virtual void onInitParameters(XPathWrapper &xpath);
    virtual bool onInit(const TransactionDataWrapper& data );

    virtual bool onEnterPositions(const TransactionDataWrapper& data);
    virtual bool onOpenPositions(const ActionDataWrapper& actionWrapper );
    virtual bool onClosePositions(const ActionDataWrapper& actionWrapper );

private:
    // epsilon - as a percentage
    double _dEps;
};
```

Strategy Callbacks

The *onInitParameters* callback gets the access to *strategy* block in the strategy parameter file. This function is designed for initialization strategy constants. Note, that you should first call the implementation of the base class.

```

void TrendExample::onInitParameters(XPathWrapper &xpath)
{
    BaseStrategyClass::onInitParameters(xpath);

    _dEps = xpath.asDouble("Conditions/@epsilon") / 100.0;
}

```

The callback *onInit* is called at strategy start time. This function is designed for initialization strategy variables. Note, that you should first call the implementation of the base class. In this example there is nothing to write in this callback.

```

bool TrendExample::onInit(const TransactionDataWrapper& data)
{
    return BaseStrategyClass::onInit(data);
}

```

In this strategy we use the default implementation of *onStart* callback, which calls to *enterPosition* function.

```

bool TrendExample::onEnterPositions(const TransactionDataWrapper& data)
{
    // get current price
    double currPrice = data.value(PRICE);

    // send buy-stop order and sell-stop order to enter new position(s)
    sendEnterStopPositions(currPrice, _dEps);

    return true;
}

```

In *enterPosition* strategy sends two orders. When one of them is filled, the *position change event* is generated and the Framework calls to *onOpenPositions* callback.

As a rule, this callback is called then number of strategy positions has been changed and is **not zero**.

```

bool TrendExample::onOpenPositions(const ActionDataWrapper& actionWrapper)
{
    // send orders to close the already open position(s)
    // in accordance with stop loss and profit limit parameters
    sendClosePositions(actionWrapper._price, getLimit(), getStop());

    return true;
}

```

In *onOpenPositions* strategy sends two orders to determine allowed loss and desired profit. When one of the orders is filled, the *position change event* is generated and the Framework calls to *onClosePositions* callback. As a rule, this callback is called then number of strategy positions has been changed and is **zero**.

In this case there is nothing to do upon closing position.

```
bool TrendExample::onClosePositions(const ActionDataWrapper& actionWrapper )
{
    return true;
}
```

The call of *onClosePositions* marks that the current trading cycle has been ended. At this stage the Framework tries to conceive new trading cycle. If the *counts* conditions allow and *enterEnd* time is not reached, the Framework calls to *enterPosition* callback to initiate new trading cycle.

Strategy Parameters file

Here is an example of the strategy's XML file.

```
<VTrader
  notes='None'
  description='TrendExample Example'
  packageName='MyStrategies'
  templateName='TrendExample'>

  <Counts wins='100' total='100' losses='10' />

  <Time startTime='08:30:00' enterEnd='15:00:00' endTime='15:00:00'/>

  <ClosePosition stop='1.0' limit='2.0' />

  <Wait afterWin='0' afterLoss='0' afterEntrance='0'/>

  <Strategy
    name='MyTrendExample'
    traceToFile='yes'>
    <Conditions
      epsilon='0.5'>
    </Conditions>
  </Strategy>
</VTrader>
```

Save the XML file. Note that file name should be as the strategy name, for this example is *MyTrendExample.xml*. Put the file in

C:\Program Files\StrategyTuner2\metadata\contracts\<contract name>\<strategy vendor>\.

After you have compiled your strategy and placed strategy parameters file, you can run Strategy Tuner and execute the strategy.