© Lime Brokerage LLC 2011-2017

Last Updated: 10/9/2017

Strategy Studio

Version 2.18.0.0

Getting Started

Welcome to Strategy Studio, a high performance C++ multi asset class strategy development platform, designed for efficient implementation, testing, and deployment of algorithmic trading strategies. Strategy Studio provides a *Strategy Development API* which is agnostic to data feed and execution providers, allowing you to focus on the logic of your strategies. Strategy Studio also includes a set of *Strategy Servers*, which are responsible for loading and running your trading strategies, along with the *Strategy Manager*, a performance monitoring and risk management user interface that you can use to centralize your research and production trading.

This document provides basic information on installing and configuring the various components of the Strategy Studio software, as well as an overview of how to create and deploy a trading strategy.

More detailed *Strategy Development API* documentation may be found in the 'html' directory of Strategy Studio's 'docs' folder. More detailed *Strategy Manager* documentation is available from within the user interface.

We are happy to answer any questions you have. Please email us at StrategyStudio@limebrokerage.com.

Contents

Server Configuration

Our service representatives will assist you with configuring the Strategy Studio Server components.

The first part of the process will be generating a license file. You or one of our customer service representatives will need to run LicenseInfoCollector.exe on the physical server(s) where you will be installing Strategy Studio. This program will gather the information we need to create a license, outputting a file license-info.txt. We will use this file to generate your license.

The server software and updates thereof may be downloaded from the Strategy Studio FTP site.

The software configuration should rarely need updating or changing and will generally be handled by our service representatives

Server config.txt settings

While under most circumstances you should not need to manually change your configuration settings, the Strategy Server's config file includes support for the following variables:

NUM_PROCESSORS: The number independent logical event processors to assign strategies to. Only relevant for the live case. When Backtesting, Strategy Studio assigns one event processor per back testing experiment.

ACCEPTOR_PORT: The TCP/IP port used by the server to communicate with the Strategy Manager application

PNL_UPDATE_INTERVAL_SEC: How often in seconds to send pnl snapshots to the Strategy Manager. Defaults to 1 on a live server and 15 on a back testing server.

CUSTOM_GRAPH_THROTTLING_SEC: Minimum time in seconds between data points in a custom strategy graph. Defaults to 1 on a live server and 15 on a back testing server.

ENABLE_RECOVERY: If set to true on a Live Simulation or Production Trading Server, will instruct Strategy Studio to remember positions data overnight and between restarts of the server.

PROCESSOR_LOG_LEVEL, NETWORK_LOG_LEVEL: Can be used to control threshold for log file output, defaults to INFO. Valid values are DEBUG, INFO, WARN, ERROR, FATAL.

Strategy Manager Configuration

The Strategy Manager user interface is available in 32bit and 64bit Windows builds, and requires .Net version 4 to be installed on the workstation. The application is typically located in

C:\Lime\StrategyStudio\StrategyManager\StrategyManager.exe

and has a configuration file ClientConfig.xml located in the same directory. This file contains the locations of the Strategy Server(s) the Strategy Manager should connect to, and whether those servers are enabled (meaning whether the Strategy Manager should actively try to establish connectivity to them). Example entries look like:

```
<?xml version="1.0" encoding="utf-8" ?>
<Config>
  <Connections>
    <Server Name ="SimulationServer1" Mode="Live Simulation" IP="127.0.0.1"
Port="13000" Enabled="true"/>
    <Server Name ="ProductionServer1" Mode="Production" IP="127.0.0.1"
Port="13002" Enabled="true"/>
    <Server Name ="BacktestServer1" Mode="Backtesting" IP="127.0.0.1"
Port="13001" Enabled="true"/>
  </Connections>
</Config>
```

This file can be edited manually, or maintained via the Strategy Manager's *Server>Connections* dialog.

Strategy Studio: Hardware Guidelines

Strategy Studio was designed to run on commodity hardware. We prefer to avoid detailed hardware recommendations as the requirements will depend to a large extent on the nature of your strategies and the scope of your back testing. However, the following recommendations can be used as a starting point.

Servers:

- 64bit OS required
- 8 Core Nehalem
- 32GB RAM
- 300GB available disk space
- Gigabit Ethernet or better

Strategy Manager Workstations:

- 64bit OS strongly encouraged
- 4 Core Xeon
- 6GB RAM
- Gigabit Ethernet or better
- Video card with at least 256MB memory

Operating Systems and Compilers

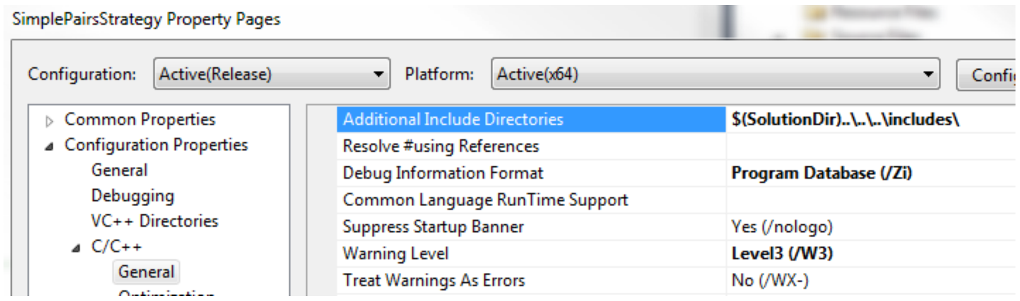
The Strategy Studio Server components are written in native C++ with maximal portability in mind. So far we have built the server for:

- Windows Server 2003/2008/2008R2/ 2012/2012R2 64 bit.
  - Microsoft Visual Studio 2010 compiler
  - Microsoft Visual Studio 2013 compiler
  - Microsoft Visual Studio 2015 compiler
- 64bit Linux
  - GCC 4.4.6, 4.6.3, & 4.8.2
  - CentOS, RHEL, Ubuntu

Creating a New Strategy: Project Configuration

The easiest way to create a new project is to copy it from one of the existing example Strategies in the StrategyStudio/examples/strategies folder on your computer. After copying the folder, rename the folder and the project files according to your new strategy's name, and within the copied files, find/replace the example strategy's name with your strategy's new name.

If you copied the example strategy to a different root directory, you will need to modify the paths of the include directories and library locations. In Visual Studio these settings are found by right clicking on your strategy project and selecting "Properties"


SimplePairsStrategy Property Pages

Configuration: Active(Release)   Platform: Active(x64)

Additional Include Directories	$(SolutionDir)..\..\..\includes\
Resolve #using References	
Debug Information Format	Program Database (/Zi)
Common Language RunTime Support	
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level3 (/W3)
Treat Warnings As Errors	No (/WX-)

Change the "Additional Include Directories" settings to an absolute path corresponding to your installation directory, such as C:\Lime\StrategyStudio\includes. Then look for the Linker section and change the "Additional Library Dependencies" directory to something such as C:\Lime\StrategyStudio\libs\x64.

If you are building your project in Linux, you may find the analogous LIBPATH and INCLUDEPATH settings in your project's Makefile.

In the header file for your strategy, please note the functions in the *extern* "c" section at the bottom of the file. These functions *GetType(), GetAuthor(),* and *GetAuthorGroup()* are used by the Strategy Studio servers to identify the strategy's name, its author, and the trading group that authored it. The author and author group must correspond to an existing users and trading group configured via Strategy Manager's administration interface, and are used to establish default permissioning for the new strategy.

Deploying Your New Strategy

Once you have successfully built your new strategy deploying it is simple:

- Copy the strategy .dll file output by the build process to the strategies_dlls directory in the location where your server is installed.
- Strategy Studio will register the strategy once you click "Refresh Strategies" in the Strategy Manager user interface, or the next time you restart the Strategy Server process.

Core API Overview

Strategy Studio's Strategy Development API is divided into several core namespaces, each of which serve a central purpose in the process of strategy development.

LimeBrokerage::StrategyStudio

The header files which fall under the StrategyStudio namespace are located in the root "includes" directory of the StrategyStudio installation path. The classes within this namespace are the core Strategy development interface, including the Strategy class from which your strategies will derive, the event messages which Strategies process, the event registration infrastructure, and the objects which track your Strategies' behavior and state, including classes representing your portfolio, your order history, and the history of your fills and trades.

LimeBrokerage::StrategyStudio::MarketModels

The header files which fall under the MarketModels namespace are located in the MarketModels subdirectory of the root "includes" directory. MarketModels contains abstractions Strategy Studio uses to track market center (eg exchange) activity, and the associated market data types: trades, quotes, order books, financial instruments, fees, and other market reference data.

LimeBrokerage::StrategyStudio::Analytics

The header files which fall under the Analytics namespace are located in the Analytics subdirectory of the root "includes" directory. Analytics contains Strategy Studio's library of tools for numerical analysis of market data, including routines for regression analysis, containers for handling time series data, rolling windows, and common technical indicators for use within your strategies.

Versioning Convention

Strategy Studio uses the following convention for version numbers:

MAJOR_VERSION.MINOR_VERSION.MAJOR_REVISION.MINOR_REVISION

When the server loads strategy and adapter shared libraries, it validates that they were built with a library version compatible with the current server version. We maintain binary compatibility within MINOR_VERSION releases. Additionally the Strategy Manager and Server will remain compatible at minimum within minor version releases.

Other Libraries In Your 'includes' Directory

For your convenience, Strategy Studio includes a couple 3rd party software libraries as part of its distribution package. These include:

- Portions of the *boost* C++ libraries. Boost contains several useful extensions to the C++ language, many of which ultimately find their way into the standard. Among other components of Boost, Strategy Studio uses Boost's DateTime library as the foundation of its timestamps.

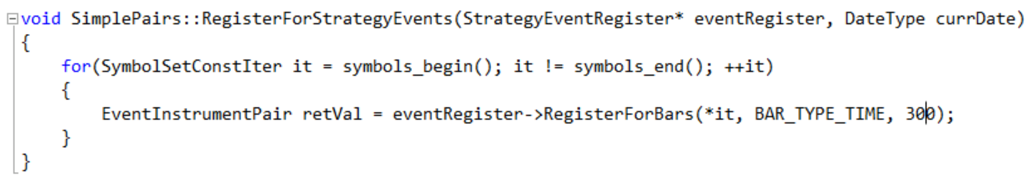
- *Eigen*, a C++ linear algebra library which provides very fast matrix math and is used by some of Strategy Studio's numerical analysis components. Eigen uses expression templates and explicit vectorization to provide both speed and an elegant matrix math syntax. We selected Eigen after extensive performance testing and feature comparison analysis.

The Strategy Class

Your strategies will derive from the *StrategyStudio::Strategy* class. This class provides the event hooks such as *OnTrade, OnQuote,* and *OnDepth* that you will use to receive market data, instructions from the Client, and information on your own orders. The class also provides accessors such as *portfolio()*, *orders()*, and *fills()* where you can find information about the state of your positions, your trading activity, and your PnL. It also provides *trade_actions(),* your starting point for sending out orders.

Registering For Market Data

After the server initializes your strategy, it will attempt to register for market data. By default, Strategy Studio will automatically register to trades/quotes/depth data for the instruments you have requested via the "Add Strategy Instance" dialog in the Strategy Manager, and for any instruments in which your strategy has a non-zero overnight position. If your strategy needs additional market data, such as subscriptions to reference instruments your strategy always needs, or subscriptions to bar or news data, you request those data items by implementing the function RegisterForStrategyEvents(StrategyEventRegister* eventRegister, DateType currDate), which will be called at the beginning of each trading day. As an example, the following implementation requests 5 minute bars for all the symbols requested by the Strategy Manager:



```
void SimplePairs::RegisterForStrategyEvents(StrategyEventRegister* eventRegister, DateType currDate)
{
    for(SymbolSetConstIter it = symbols_begin(); it != symbols_end(); ++it)
    {
        EventInstrumentPair retVal = eventRegister->RegisterForBars(*it, BAR_TYPE_TIME, 300);
    }
}
```

Note that you can use the Strategy class's symbols_begin() and symbols_end() to iterate over the symbols the Strategy Manager requested, or similarly you can use instrument_begin() and instrument_end() to iterate over the Instruments with active subscriptions. For dynamic event registrations, the Strategy class also has a *runtime_event_register()*; currently dynamic registration is restricted to scheduled events and message passing between strategies.

Resetting Your Strategy's State

When back testing, Strategy Studio allows you to rerun an instance of the strategy after tweaking strategy parameters. In order to allow you to reset your strategy's internal variables when this reset occurs, the API provides an *OnResetStrategyState()* event hook. This hook is also triggered if you use the Reset command in the Strategy Manager to reset a Live Simulation or Production strategy instance. As a best practice, we recommend that your implementation of this function restores all class variables within your strategy to the values you initialized them to in your strategy's constructor.

Establishing Your Strategy's Parameters

Strategy Studio allows you to manage many trading strategies, each with their own list of parameters that you can modify or view while running your strategy. Strategy Studio supports several types of parameters: *RUNTIME* parameters which you can modify from the Strategy Manager while your strategy is running, *STARTUP* parameters for settings which are only relevant during the initialization of a strategy instance, and *READONLY* parameters which you can use to output values to the GUI.

The Strategy class has a *params()* accessor which allows you to view and set the values of your parameters via *params().GetValue(...)* and *params().SetValue(...)*.. Updates to these values will automatically propagate to the Strategy Manager user interface. To define your strategy's set of parameters, implement the Strategy class' function *DefineStrategyParams*(), calling *params().CreateParam(...)* for each parameter you wish to add. The Strategy class also provides hooks *OnParamValidate()* and *OnParamChanged(...)* which will fire whenever a parameter's value is modified, regardless of whether the change initiated from the Strategy Manager or from programmatic *SetValue(...)*calls. They also fire when your parameter's default values are set and when any persisted values are loaded from during strategy initialization, and after OnResetStrategyState to remind your strategy logic of the most recently set values. This reinitialization of strategy parameters happens before the initial call to RegisterForStrategyEvents so that the remembered parameter values can be used to influence event subscriptions. *OnParamValidate* allows you to return 'false' to implement customized parameter validation; *OnParamChanged* fires when a new parameter value passes any bounds checking and custom validation.

Establishing Customized Strategy Commands

In addition to configuring parameters for your strategy, you may define customized commands as well. These commands give you a way to issue instructions to your strategy, such as 'Liquidate', 'CancelAllOrders', etc. Additionally, the StrategyCommands allow you to associate an input field with the command.

Similarly to the interface for parameters, the Strategy class provides a function *DefineStrategyCommands()* from which you can call *commands().AddCommand(...)*. The Strategy class will notify you when these commands trigger via an event hook *OnStratetegyCommand(...)*. In the Strategy Manager, you can access the commands you have defined in the right-click context menu when you click on a strategy instance or strategy type node in the Strategy tree.

Defining Custom Strategy Graphs

The Strategy Manager has built in graphing for core portfolio metrics (PnL, notional exposures, shares traded), but you may also define custom time series graphs for your strategy instances, and view them using the View->Custom Strategy Graphs dialog in the GUI.

The API's Strategy class has a hook *DefineStrategyGraphs()* which provides an ideal spot to add graphing series, via calls like *graphs().series().add("ExampleSeriesName")*. From subsequent event callbacks you can add new points to the series via calls like *graphs().series()["ExampleSeriesName"]->push_back(msg.event_time(), 10.0)*. By default, the server throttles submissions to avoid overloading the network and Strategy Manager with data; see Page 2 for information on configuring this throttling interval.

Core Strategy Control

Strategy Studio exposes a core strategy control mechanism for all instances of your strategies: you can Stop, Pause, or Play your strategies by right clicking in the Strategy Manager's strategy tree, by using the corresponding buttons in the Strategy Manager's toolbar, or by using the corresponding command line instructions. Pausing a strategy prevents the strategy from sending new orders, but allows the strategy to continue receiving market data. Stopping a strategy prevents new orders, cancels any existing orders, and filters market data and bars from reaching the strategy.

The strategy class has an *OnStrategyControl(…)* event hook so you can listen to information about the running state of your strategy. You can also use these events to perform custom handling for state transitions: for example you may wish to add 'cancel all' behavior when a strategy pauses. The strategy class also allows for programmatic calls to *Stop() and Pause()*.

Using Event Messages

The event messages in the *StrategyStudio::Strategy* class are your primary means of receiving the information needed to drive your trading strategies. This section explains some core considerations you should keep in mind.

Thread Safety

Strategy Studio is a multithreaded trading engine. That said, to ensure that your strategy logic is viewing a consistent set of information when making its decisions, Strategy Studio guarantees that while you are in one of our event messages, the state of your strategy is thread safe. This means if you check your *orders()* and then your *portfolio()* a few lines of code later, the contents of your portfolio will still be consistent with the information you just retrieved from your orders. This makes your trading logic easier to verify, and adds some determinism to your view of your strategy's state.

Market Data Event Messages

Strategy Studio is intrinsically a multi asset and multi asset class trading API. All market data event messages such as *OnTrade* come through with an *instrument()* accessor which tells you which instrument the data pertains to. The Instrument class itself will tell you what type of Instrument it is, what it's characteristics such as tick size and multiplier are, and the current state of its market activity, such as what market centers you have data for, and what its quotes and order books look like on each market center.

In addition to identifying the instrument for which there is new information, the market data event messages will identify the new piece of information being propagated. For example, the event message in *OnTrade* has a *trade()* accessor.

All event messages in Strategy Studio contain an *event_time()* to provide the Strategy Server's timestamp for the event. Market data items such as trades and quotes contain additional accessors to pass along the data source's own timestamp, as well as your feedhandler's timestamp, when available/applicable.

OnMarketState

Strategy Studio uses *OnMarketState* to give strategies information about whether the market is open, closed, etcetera. It is important to understand that this event gets triggered not just when the state of the market changes, but also to deliver snapshot notifications of the state of the market when a Strategy initializes or when a new day of back testing begins. The event message passed on by the *OnMarketState* callback contains an accessor *IsSnapshot()* to differentiate the two cases.

Scheduled Events

Strategy Studio allows subscription to scheduled events, which can be used for instance to receive alerts at pre-specified times during the trading day. Registration looks like:

```
eventRegister->RegisterForSingleScheduledEvent("15AfterOpen", USEquityOpenUTCTime(currDate) +
                                                    boost::posix_time::minutes(15));
```

These events are delivered via OnScheduledEvent(...). Recurring events may also be defined via eventRegister->RegisterForRecurringScheduledEvents(...) and will repeat according to a user specified interval.

Bars

Strategy Studio can generate Bars which summarize periods of market activity based on the underlying tick data. Bars may be specified in terms of time intervals (with a minimum interval of 1 millisecond), number of trade ticks, number of shares/contracts traded, realized variance, or high-low range. Page 6 of this document has an example of bar registration.

Strategy Studio aligns time based bars to the market open time. Realized variance bars are based on the returns of the top_quote()'s mid price, sampled every trade tick. The formula we use in calculating realized variance is Sum(|logReturn|^p) where p may be specified via the optional p_value parameter of the RegisterForBars(...) function.

Bar's are delivered via the strategy's OnBar(...). If you have subscribed to more than one bar interval for an instrument, they may be differentiated using the BarEventMsg's type(), interval(), and p_value() accessors.

If a time bar and a scheduled event are due at the same time, bars will fire first. This allows you to write code that does per instrument calculations with time bars, and cross sectional post processing using RecurringTimedEvents of equal interval.

Sending Messages to Another Strategy

Strategy Studio allows message passing from one strategy instance to another (limited for now to instances running on the same server). These messages are sent via the SendMessageToStrategy(...) functions available in either the eventRegister parameter of RegisterForStrategyEvents or the Strategy's runtime_event_register().

Messages are delivered via the Strategy class's OnStrategyMessage(...) event hook.

Additionally, the API allows for topic based pub/sub style messaging. Strategies may register for a named topic using eventRegister->RegisterForTopicSubscription(...), send messages to a topic via runtime_event_register()->SendMessageToTopic(…), and receive the messages using the Strategy class's OnTopicMessage(...) event hook.

Market Data

Every market data event message references an *Instrument*. This class exposes both descriptive data about the instrument, along with state information Strategy Studio maintains based on the incoming market data events.

The Strategy Studio API handles several asset classes. The instrument's *type()* accessor identifies whether the instrument is an equity, option, future, or currency pair; based on this type, the Instrument may be cast to one of the corresponding derived classes (eg *Option*, *Future*, *CurrencyPair*) to retrieve additional properties of the instrument specific to the security type.

When subscribing to derivatives, Strategy Studio automatically associates derivatives to their underliers. For example, upon a market data event for an option, instrument().underlier() can be used to quickly access the underlier; from a tick on the underlier, *instrument().children_begin()/instrument().children_end()* can be used to iterate through the associated options.

The Instrument's *markets()* accessor provides data specific to each market center associated with the instrument.

Order Books

Strategy Studio maintains an *instrument().aggregate_order_book()* which combines liquidity information contributed from the order book or quotes of each market center for which the server has data. This object exposes many summary statistics about how much liquidity is available at each price level in the market. Each price level in the *aggregate_order_book* also exposes *participating_markets*, to quickly check which market centers are contributing size to the price level.

For each market center for which the server has an order book feed, Strategy Studio also maintains an exchange-specific order book, exposed via *instrument().markets()[market_center_id].order_book()*. This object exposes a similar set of summary statistics as the aggregate_order_book, with the additional ability to iterate through each individual order on the exchange's book.

Quotes and Trades

The Instrument's *last_trade()* and *top_quote()* accessor provide easy access to the market-wide last trade print, and to a summary quote representing the global best bid and offer, with sizes aggregated from all market centers present at the aggregated top of book.

Additionally, each instrument().markets()[market_center_id] has *last_trade()* and *quote()* accessors to check the latest trade and current BBO quote for the particular MarketCenter. When servers subscribe to overlapping sources of data, these accessors take an optional parameter to check for the trade or quote as it appears from a specific type of data source.

Daily Trade Stats

The instrument's *trade_stats()* exposes accessors for prior day's close (and close date), most recent open price, day's volume, and day high/low prices, when available. The server endeavors to initialize these stats upon connecting to a live data source, though this may not be possible with all data adapters.

The server increments and updates high, low, and volume from the size and price of incoming trade ticks, filtered by the Trade object's *DoesTradeUpdateVolume*/*DoesTradeUpdateConsolidatedHighLows* helper functions . The server updates open/close fields based on the trade tick's trade conditions; note with many of the data adapters, the necessary flags to properly identify these prints will only be available from US Consolidated data feeds.

Feed Types and Managing Overlapping Data Sources

In the US Equity markets, it is common for the server to connect to several data feeds which can contain overlapping data for an instrument. For example, subscription to the consolidated tape plus a depth-of-book feed such as Nasdaq TotalView will result in duplicate trade prints and top of book liquidity for Nasdaq.

To manage these overlapping data scenarios and avoid double counting, Strategy Studio classifies market data ticks into three source types: CONSOLIDATED (eg SIAC/UTP/OPRA), DIRECT (exchange direct trade and BBO feeds such as NSYE Quotes), and DEPTH_OF_BOOK (eg Nasdaq TotalView or NYSE Open Book Ultra).

When multiple feed types are present for a market center, the server will use only one feed type to contribute data to the *aggregate_order_book* and *top_quote*, and will use only one feed type to update Bars, daily high/low/volume stats, and simulated fills. By default the server chooses the best available feed type from the data adapter's feed configuration, where Depth is considered better than Direct, which is considered better than Consolidated. Each Trade and Quote event message will identify what type of feed the tick comes from.

The server allows manual configuration for the preferred feed type for each market center (separately for quotes and trades), via an optional "preferred_feeds.csv" file in the server executable's directory. An example file is located in the SDK's docs folder. The format is

MARKET_CENTER_ID,PREFERRED_QUOTE_SOURCE,PREFERRED_TRADE_SOURCE

where valid source type specifiers are: BEST, CONSOLIDATED, DIRECT, DEPTH, NONE. *MARKET_CENTER_ID* names should be specified based on the API's MarketCenterID enum names, omitting the enum prefix (for example: "NYSE"). Any market center not listed in this file defaults to BEST.

In the event of a live data problem with the active preferred feed type, server supports hot swapping a different feed type via command line instruction:

set_preferred_feed_type <market_center> <quote_feed_type> <trade_feed_type>

For live configuration changes, supported values are: DEPTH, DIRECT, CONSOLIDATED, NONE. Changes made via this command will persist to the "preferred_feeds.csv" file.

Strategies can check programmatically the current contributor to the aggregate_order_book by checking for example

instrument->markets()[MARKET_CENTER_ID_NYSE].preferred_quote_source()

This is useful in identifying whether a quote() or the price level of the order_book() is the source of a market center's liquidity in the aggregate_order_book().

Order Management

In the Strategy Development API, your strategy can send orders by initializing an OrderParams object and passing it to *strategy->trade_actions().SendNewOrder(params)*. The *trade_actions* object also exposes functions for sending Cancels, CancelReplaces, and batch Cancels.

The return value of the *trade_actions().SendNewOrder* will communicate inline whether the Strategy Server was able to initiate the requested action. Reasons for failure include internal filtering (for example, the strategy is not permission to send trades after hours), or inability to communicate with an execution handler.

Once an order has been placed, it is moved into a pending state, added to an order tracking container, and Strategy Studio starts delivering updates to that order (including a notification of the pending state) via the strategy's *OnOrderUpdate* callback. Strategy Studio sends these updates to the Strategy Manager's Orders grid as well.

<u>Order Tracking</u>

Strategy Studio maintains an *IOrderTracker* object accessible via the Strategy's *orders()* accessor. This object contains collections for both the strategy's working orders, organized by instrument to facilitate quick within-instrument iteration, and all its tracked orders. By default Strategy Studio keeps all orders placed during the current trading day, as well as any orders left open from a prior trading day, in this in-memory collection. Orders can be looked up using *orders().find(OrderID)*. *IOrderTracker* also provides various order iterators and summary statistics on your strategies' order activity via the *orders().stats()* and *orders().stats(instrument)* accessors.

The *IOrderTracker::stats(...)* accessors return an *OrderStatistics* object for either strategy level or within-instrument statistics, including information on how many orders have been placed, along with working order risk measurements such as *net_working_order_size()* and *net_working_order_notional()*. Strategy Studio computes these metrics in a conservative way when there are actions like CancelReplaces pending. In other words, if you're trying to CancelReplace to a higher absolute notional size, we assume the larger size is working unless/until we're rejected on the replace, but if you're trying to CancelReplace to a smaller absolute notional size, we assume the larger size is still working until we get an acknowledgement for the replace.

<u>Order Updates and State Management</u>

The *OnOrderUpdate* callback contains an *OrderUpdateEventMsg*, which has an *order_id()* and an *update_type()* accessor to identify the type of update delivered. The message also contains an *order()* accessor from which you can retrieve the order's parameters and status information, including filled quantities and an *order_state()* that describes the overall state of an order. Additionally, if there is a fill associated with the order update, the associated information will be available from the *fill()* pointer, and *fill_occurred()* will return true.

All orders initialize to state NEW_ORDER and transition to OPEN_PENDING when successfully sent. The market center will then either acknowledge or reject the order which will move the state to OPEN or REJECTED respectively.

As updates arrive, only certain state transitions are allowed, which helps enforce a consistent state and maintains state priorities. As a result, not all order updates will change the order state. For example, a PARTIAL_FILL while an order is in CANCEL_PENDING state will trigger an OnOrderUpdate with *update_type()* == ORDER_UPDATE_TYPE_PARTIAL_FILL but will not change the order_state() as the CANCEL_PENDING state takes priority.

There are several order states the system considers Complete, meaning the order will not undergo any further state changes and will be removed from the working orders collection. For example, consider a cancel request which crosses paths with an incoming fill that completes the order. In this case, the fill will move the order into the final state FILLED, and the order will no longer be considered 'working.' The function *bool OrderIsComplete(OrderState)* can be used to identify these states. In normal circumstances, for every successfully sent order the Strategy should receive: one *OnOrderUpdate* with state OPEN_PENDING; one order update which an update_type() indicating an ORDER_UPDATE_TYPE_OPEN or ORDER_UPDATE_TYPE_REJECT; and one update that transitions an order to a Complete state. Once an order reaches a complete state, the only time the strategy should see subsequent updates is if an ORDER_UPDATE_TYPE_BUST occurs, in which case there will be a fill object representing the fill adjustment. To avoid any possible ambiguity as to when the system removes an order from IOrderTracker's list of working orders, the *OrderUpdateEventMsg* also has a flag "*bool completes_order ()*."

Order State	Complete
ORDER_STATE_UNKNOWN	No
ORDER_STATE_NEW_ORDER	No
ORDER_STATE_OPEN_PENDING	No
ORDER_STATE_OPEN	No
ORDER_STATE_REPLACE_PENDING	No
ORDER_STATE_REPLACED	No
ORDER_STATE_CANCEL_PENDING	No
ORDER_STATE_CANCELLED	Yes
ORDER_STATE_PARTIALLY_FILLED	No
ORDER_STATE_FILLED	Yes
ORDER_STATE_REJECTED	Yes

An order in any non-Complete state can transition to a Complete state, after which the order won't transition again. In addition to Complete states, the allowed transitions from Working states are as follows:

Current Order State	New Order State
OPEN	CANCEL_PENDING, REPLACE_PENDING, PARTIALLY_FILLED
REPLACE_PENDING	OPEN, REPLACED, PARTIALLY_FILLED
REPLACED	CANCEL_PENDING, REPLACE_PENDING, PARTIALLY_FILLED
CANCEL_PENDING	OPEN, PARTIALLY_FILLED
PARTIALLY_FILLED	CANCEL_PENDING, REPLACE_PENDING

To minimize the memory utilization of back test servers, an optional server config variable, DELETE_COMPLETED_ORDERS can be enabled. If set to TRUE, the server will delete order objects and remove them from IOrderTracker's tracked_orders collection immediately after a call to OnOrderUpdate in which msg.completes_order() is true. This setting is not recommended for Production servers.

Handling Timestamps

All timestamps in Strategy Studio are typed as *TimeType*, defined in *StrategyStudio::Utilities*. *TimeType* is a typedef wrapping boost::posix_time::ptime from Boost's DateTime library, which provides extensive functionality for dealing with date and time information. Here are some things to keep in mind:

- All timestamps in the strategy servers are presented in UTC
- Strategy Studio's *TimeType* is a datetime, meaning timestamps include both the calendar date and time of day. You can isolate the date portion of a timestamp by calling the timestamp's *date()* accessor.
- Strategy Studio's timestamps will usually have microsecond granularity. However, *TimeType* is capable of storing nanosecond data, so if your datasources provide nanosecond timestamps and the nanoseconds are important to you, Strategy Studio can be configured to expose this information.
- For your convenience Strategy Studio has provided several basic functions for converting timezones and converting *TimeType* to string representations. These functions are located in Utilities/TimeType.h
- If you need date or time manipulation functionality beyond that included in our TimeType.h, please refer to Boost DateTime's documentation, located at:
    - http://www.boost.org/doc/libs/1_61_0/doc/html/date_time.html
- Market data such as trades and quotes will have timestamps for source time and feedhandler time, in addition to Strategy Studio's internal time. Strategy Studio's data adapters will differentiate these timestamps as best as possible given your available data. Note that in back testing mode, Strategy Studio's internal time will correspond to one of the other timestamps. If you are back testing from bulk historical data purchased from an exchange, Strategy Studio will only have the source time to base its timestamps from. If you use a tick database to collect and store data, or if you capture data from your feedhandlers and timestamp them manually, Strategy Studio will use the collector/feedhandler timestamp as its internal time.

Market Hours

Configuring Server's Primary Trading Hours

By default, Strategy Studio will filter market data and filter sending orders outside of normal trading hours. To toggle these filters, the Strategy class exposes the ability to set the following properties:

```
set_enabled_pre_open_data_flag(true);
set_enabled_pre_open_trade_flag(true);
set_enabled_post_close_data_flag(true);
set_enabled_post_close_trade_flag(true);
```

To send orders for extended-hours trading sessions, it's important to also set the order's TIF appropriately. This generally means using ORDER_TIF_GTX for post-close orders.

The server's main config file allows the following variables for customizing the default times at which *OnMarketState* events occur:

```
PRE_OPEN_MARKET_TIME=4:00
OPEN_MARKET_TIME=9:30
CLOSE_MARKET_TIME=16:00
```

The default time zone for these values is EST. This may be changed by setting one of these two config file options:

```
TIME_ZONE=PST-8PDT,M3.2.0,M11.1.0
```
This must be specified in POSIX time zone format

-or-

```
TIME_ZONE_REGION=America/Los_Angeles
```
The value must be a region ID from a boost::date_time timezone database file, an example of which is included in the SDK's docs folder

```
TIME_ZONE_DB_FILE=path
```
Specifies location of the boost timezone database file, defaults to ./date_time_zonespec.csv

By default, each day in a backtest begins with the clock set to midnight EST. To configure this daily start time, set the following server config variable:

```
DAY_BOUNDARY_TIME=-5:00
```

The time zone for this value defaults to EST, but can be overridden with DAY_BOUNDARY_TIME_ZONE and DAY_BOUNDARY_TIME_ZONE_REGION following the same conventions as for market times above.

Configuring Holidays and Half Days

At server startup Strategy Studio will look for a holiday & half day file named "*holidays.txt*" in the server's main directory. An example holiday file is located in the SDK's docs folder. The times in this file will be interpreted according to the time zone settings established in the server's main config file. The entries in this file should be formatted as:

```
YYYYMMDD,<holiday_type h|f>[,<market_open>,<market_close>[,<market_pre_open>]]
```

On half-days, the file's adjusted open/close times will override the default primary trading hours for firing Strategy's *OnMarketState*. Entries will also adjust the return values of the functions in MarketModels/MarketCenterHours.h such as *USEquityOpenUTCTime(...)*.

Backtest servers will skip querying data on days labeled as full holidays.

Real-Time PnL Measurement

The Strategy Studio servers calculate and expose real-time profit and loss calculations to the Strategy API and to the Strategy Manager user interface. The Strategy class's *portfolio()* object offers *total_pnl()*, *realized_pnl()*, and *unrealized_pnl()* both for the strategy's overall portfolio, and for any instrument within the portfolio. By default Strategy Studio marks positions to market, but this may be configured via a server config file variable, MARKING_MODE. Valid values are

- MARKING_MODE=MARK_TO_MARKET
  - Marks long positions to bid if a *top_quote* bid exists, else to last trade price, else to fill price. Marks short positions to offer if a *top_quote* offer exists, else to last trade price, else to fill price. (Default setting)
- MARKING_MODE=MARK_TO_MID
  - Marks positions to *top_quote*'s mid price if both a bid and an offer exist, else to last trade price, else to fill price.
- MARKING_MODE=MARK_TO_LAST
  - Marks positions to the *last_trade* price if there have been valid trade ticks, otherwise to fill price.

Strategy Studio also keeps track of a daily PnL track record. By default, daily profit is sampled at market close (or at server shutdown time if the server is shutdown before the market closes), but this may be configured by setting the server config file variable:

- EOD_MARKING_TIME=18:00
  - Interpreted according to DAY_BOUNDARY_TIME_ZONE.

The marking mode used to sample daily profit defaults to the server's marking mode, but this can be configured by setting:

- EOD_MARKING_MODE={MARK_TO_MARKET, MARK_TO_MID, MARK_TO_LAST, MARK_TO_CLOSE}
  - MARK_TO_CLOSE uses the official close prices if available, and defaults EOD_MARKING_TIME to DAY_BOUNDARY_TIME.

The *portfolio()*'s day_pnl accessors are calculated by taking total_pnl - previous_end_of_day_pnl. Note that while the daily PnL is sampled at market close, the newly sampled end of day marks do not become the previous_end_of_day_pnl benchmark for the sake of 'day_pnl' until the day boundary passes.

Corporate Actions

Strategy Studio will automatically adjust positions and PnLs for splits and dividends if the server is configured to use a Corporate Action Adapter. When a corporate action occurs, the Strategy will also receive a notification via the *OnCorporateAction(...)* event. Dividends and spinoffs (treated as cash distributions) will accrue to the *portfolio()*'s cash_balance, as well as to the individual PositionRecord's cash_flow metric.

The TextCorporateActionAdapter enables the server to process symbol changes and other corporate actions specified in CSV files. To configure this adapter, the following lines must be added to the server config file:

- CORPORATE_ACTION_LOADER=TextCorporateActionAdapter
- CORPORATE_ACTION_SOURCE_DIRECTORY=<path to directory containing input files>

Symbol changes are specified in a file named symbol_changes.csv with the format:

- Date(YYYYMMDD),OldSymbol,NewSymbol

Other corporate actions are specified in a file named corporate_actions.csv with the format:

- ExDate(YYYYMMDD),Symbol,Type,RateType(A/M),Rate
- Type is from the set { CASH_DIVIDEND, STOCK_DIVIDEND, STOCK_SPLIT, SPINOFF }

Multi-Currency Portfolios

Strategy Studio supports trading instruments priced in multiple quote currencies and converting PnL measurements back to a domestic portfolio currency. This currency is specified by the server config file variable HOME_CURRENCY, taking an ISO currency code value, defaulting to USD. To ensure consistent performance data aggregation, the Strategy Manager must connect to a set of servers sharing a common home currency; the Strategy Manager's Tools - > System Settings dialog allows selection of this currency.

Strategy Studio's reference database is responsible for tracking the quote currency for an Instrument, which is exposed to the API via *instrument.currency()*. Reference data loader utilities are available for various data vendors. For spot currency pairs, the Instrument may be cast to a CurrencyPair object to retrieve *ccy1()* and *ccy2()* accessors -- *ccy2()* and *currency()* will return the same values.

Configuring Currency Rate Data

When CurrencyPair tick data is available, Strategy Studio supports real-time rate adjustments to PnL calculations. When registering for an Instrument denominated in a currency that does not match the HOME_CURRENCY, the PnL engine will scan the list of available currency pairs for the rate (or combination of rates) required to translate PnL from the quote currency to the home currency, and automatically subscribe to this data behind the scenes.

When CurrencyPair tick data is not available, the server may load a static file containing a time series of daily currency rates. The server config file variable CURRENCY_RATES_FILE specifies the location of this file, which should have lines in the following format:

- #SYMBOL,DATE,RATE
    - *Symbol* should be a currency pair such as EUR/USD
    - *Date* should be YYYY-MM-DD

At the start of each day of strategy processing, the server will look for the most recent day's rate and apply it to PnL calculations. The server will also chain together available rates and rate inversions if needed to find an appropriate conversion rate for an instrument. If the server cannot find an appropriate conversion rate, it will use a default rate of 1.

Conventions for Raw vs Currency Adjusted Prices, and Currency Funding Flows

Strategy Studio will present market data and fill prices in the raw quote currency of the instrument, but present PnL, notional exposures, account balances, and cash flows in the HOME_CURRENCY. Additionally, dollar trading fees are assumed to be specified in units of HOME_CURRENCY.

When transacting an instrument denominated in a non-home quote currency, Strategy Studio currently makes the assumption that there is an ongoing balance in the quote currency until the position is closed. For example, to buy a European equity denominated in EUR within a USD account, Strategy Studio assumes the need to borrow the required quantity of EUR to make the exchange, and that a short EUR exposure is in effect until the position is liquidated. Realized earnings in the quote currency are automatically converted to the home currency at the prevailing conversion rate.

This applies to positions in currency pairs as well, eg CCY1/CCY2 where ccy2 does not match the home currency. When closing positions in CCY1, realized balances in CCY2 will be converted to home currency at the prevailing conversion rate.

The outstanding balance in the position's quote currency is available via IPositionRecord's *currency_position()* accessor.

Additionally Strategy Studio maintains the aggregated currency exposure for each Strategy's portfolio, exposed via *portfolio().currency_exposure(CurrencyType currency)*. These aggregates are also viewable via the Strategy Manager's Currency Exposure grid.

The portfolio tracker's *home_currency*() accessor may be used to retrieve the server's home currency.

Server Types

Strategy Studio offers four types of servers to which strategy libraries may be loaded:

- **Production Servers**
  - Server's executable is named *StrategyServerProduction.exe*
  - Connects strategies to live market data and a production execution adapter
- **Lime Simulation Servers**
  - Server's executable is named *StrategyServerLiveSim.exe*
  - Connects strategies to live market data and a simulated execution adapter
- **Backtesting Servers**
  - Server's executable is named *StrategyServerBacktesting.exe*
  - Connects strategies to a historical tick data reader and a simulated execution adapter
- **Local Development Server**
  - This is a special limited server type included in the SDK, located at:
    - *StrategyStudio/LocalDevServer/ StrategyServerLocalDev.exe*
  - Can be used to test strategy code locally before deploying to a full server with real data credentials
  - Similar to a Live Simulation server, except connects only to a Simulated Market Data adapter
    - This adapter currently simulates the case of order book feeds coming from multiple exchanges. It does not yet simulate consolidated quote feeds.
  - Does not require a licensing file to run
  - Unlike the full servers, does not perform Author and AuthorGroup validation checks before loading a strategy DLL; assumes everything falls under TEST_USER, TEST_GROUP, TEST_ACCOUNT
  - Documentation on the Simulate Market Adapter's price process is available in the StrategyStudio/docs/adapters folder of the SDK.

Server Command Line

Most often you will use the Strategy Manager user interface to control and monitor your strategies. However, the Strategy Server command line does offer some control functionality. Command convention is: <command> <mandatory_args> [<optional_args>]. Mandatory arguments are positional. Supported commands include:

- **create_instance** < strategy_instance_name > <strategy_type> <group> <account> <trader> <initial_cash> [-server <server_name>] [-symbols <symbol_list>] [-params <param_list>] [-subscribe_to_options <0|1>] [-processor <processor_id>]
  - <symbol_list> format is <symbol1>|<symbol2>|<symbolN>
  - <param_list> format is <parameter1>=<value>|<parameter2>=<value>|<parameterN>=<value>
- **export_cra_file** <cra_file_name> [<output_directory>] [<full_order_history_boolean>]
  - Exports csv files for orders, fills, and intraday strategy pnl history from a strategy result cra file
- **external_fill** <strategy_instance_name> <symbol> <side> <quantity> <price> <market_center> [-server <server_name>]
- **help**
- **load_commands** <filename>
  - Loads a text file with one command per line for batch command processing
- **param** <param_list> {-name <strategy_instance_name>}|{-type <strategy_type> [-group <group>] [-trader <trader>] [-account <account>] [-server <server_name>]}
  - Updates the strategy parameters based on the supplied list for the given strategy instance or type
  - <param_list> format is <parameter1>=<value>|<parameter2>=<value>|<parameterN>=<value>
- **pause|start|stop|terminate** <strategy_instance_name>|-all|{[-type <strategy_type>] [-group <group>] [-trader <trader>] [-account <account>]}
- **quit**
- **recheck_strategies**
  - Scans the strategies_dlls directory for newly added trading strategy types and registers them with the server. This ability is also available in the server management dialog in the Strategy Manager GUI.
- **set_default_fees** [apply_to_existing_strategies_bool] -fee_type [fee_value] ...
  - Updates default account fees based on the supplied values.
  - Valid fee types are: broker_clr_fees_per_order, broker_clr_fees_per_contract, fees_percentage_of_notional, sales_fees_percentage_of_notional, sales_fees_per_share, and broker_passes_through_exchange_fees.
- **set_preferred_feed_type** <market_center> <quote_feed_type> <trade_feed_type>
  - Supported feed type values are DEPTH,DIRECT,CONSOLIDATED,NONE
- **set_simulator_params** -param_name [param_value] ...
  - Updates simulator parameters based on the supplied values.
  - Parameter names are: latency_to_exchange (ms), latency_from_exchange (ms), excess_slippage (bps), liquidity_multiplier, ignore_trade_ticks, and wait_if_locked_crossed.
- **start_backtest** <start_date> <end_date> <strategy_instance_name|...> <query type>
  - Starts a backtest experiment
  - Format for <start_date> and <end_date> is YYYY-mm-dd
  - <query_type> is 0 for quotes and trades, 1 for trades only.
- **strategy_cmd** <command_id> [-cmd_input command_input] {-name <strategy_instance_name>}|{-type <strategy_type> [-group <group>] [-trader <trader>] [-account <account>] [-server <server_name>]}
- **strategy_instance_list**
  - Outputs the names of all of the strategy instances registered on the server
- **update_strategy_fees** {-name <strategy_name>}|-all|{[-type <strategy_type>] [-group <group>] [-trader <trader>] [-account <account>]} -fee_type [fee_value] ...
  - Updates the instance's strategy fees based on the supplied fee parameters.

Command Line Utility

For an alternative to interacting with the server via its command prompt or via the Strategy Manager, Strategy Studio includes a StrategyCommandLine utility executable in the 'utilities' subdirectory of the server folder. The StrategyCommandLine utility allows you to issue commands to a remote Strategy Server instance. Depending on the arguments issued when running the utility, it will be placed into one of three modes:

- Interactive mode
  - Arguments: none
  - Accepts commands in a command prompt loop and sends them to the server for execution
- File mode
  - Arguments: <filename>
  - Reads commands from supplied file and sends each to the server for execution
  - Stops processing as soon as server indicates one of commands failed
- Single Command mode
  - Arguments: cmd <command and its params>
  - Sends the single command to the server for execution

Executable Return Values (File mode and Single Command mode, only):

0	SUCCESSFUL
1	FAILED – server response indicated a failure
2	FAILED – unable to process command line arguments
3	FAILED – unable to process configuration file
4	FAILED – unable to connect to server
5	FAILED – server response timed out
6	FAILED – unexpected network error or disconnect
7	FAILED – unexpected exception (see log file)

In order for StrategyCommandLine to run, a configuration file named cmd_config.txt must be placed in the same directory as the utility executable. The format of the file is:

- USERNAME=username
- PASSWORD=password
- SERVER=StrategyServerHostname
- PORT=StrategyServerPort
- RUN_MODE={1,2,3,4}
- TIMEOUT=number of milliseconds to wait for each command. *Optional;, default value of 0 means wait synchronously for server response.*

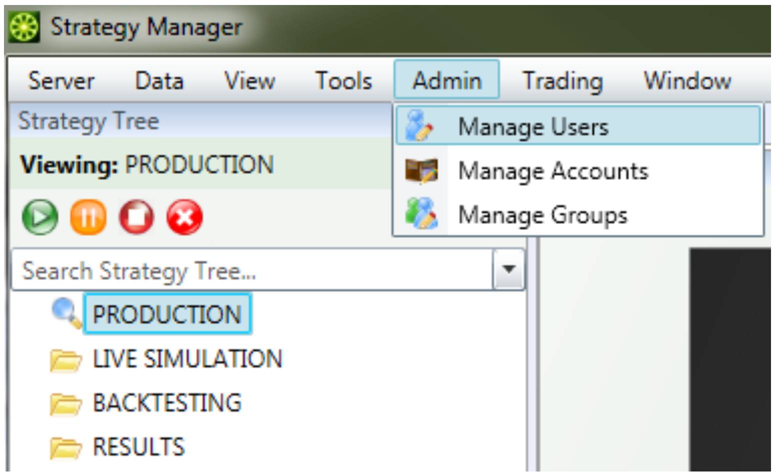
The run modes are, sequentially from 1 to 4, Backtesting, Live Simulation, Production, and Local Development. StrategyCommandLine accepts the same commands as the server's integrated command line. However, note that when using the utility in Interactive mode, 'quit' will quit the utility rather than the server.

The server process's built-in command line can be disabled by adding the line DISABLE_COMMAND_LINE=TRUE to the server's config file. With the command line disabled, the server can be run as a background process in Linux, and you can shut down gracefully by sending a signal to it (eg 'Ctrl-C'), or by using the StrategyCommandLine utility to issue the 'quit' command in file mode or in single command mode.

Users/Groups/Accounts

Strategy Studio performs portfolio aggregation by trader, by trading group, and by account. We also aggregate performance to the firm level. To ensure that the PnLs sum in a meaningful way, Strategy Studio enforces that each instance of a strategy is 'owned' by a particular user, group , and account. Strategy Studio assigns this ownership when you create an instance of the strategy via the Strategy Manager.

When our service representatives deploy our software at your site, we will make sure you have at least one administrative user, one trading group, and one account properly configured. You may add as many additional users, groups and accounts as you wish via the *Admin* menu in our Strategy Manager interface:



Frequently Asked Questions

- When looking at US consolidated equity data, why do I sometimes see trades that are far from the bid/ask prices?
    - *Trades in the consolidated equity feeds are associated with one of many Trade Conditions, which can identify trades that are for example reported late or out of sequence, or trades that reported with a price calculated as a function of older prices (for instance VWAP executions). Strategy Studio exposes the Trade Condition (and the Quote Condition) in the Trade (and Quote) class in MarketModels. MarketModels also contains ConditionCodes.h, which contains several helpful functions that can help you filter your trade ticks appropriately. Errors in your data feed are a less likely possibility.*
- Does Strategy Studio filter trade ticks when constructing Bars?
    - *Yes, if Strategy Studio has access to Trade Conditions, Strategy Studio only uses a trade tick's price for calculating the high, low, open, and close values of a bar if the trade's condition is appropriate.*
- Does Strategy Studio filter quote ticks when constructing top_quote?
    - *Yes, if Strategy Studio has access to Quote Conditions, it will only include a market center's quote in its top quote calculation if, in the case of US consolidated equity data, the quote is considered eligible to participate in the SIP's NBBO calculation, or, in the general case, if the quote is a firm quote.*