# OpenMAMDA API Developer's Guide

30 September 2012

# Document Conventions

| **Information Type** | **Example** |
|---|---|
| Feed name | *OpenMAMA Source* |
| NYSE Technologies product | ***OpenMAMA*** |
| Configuration file content or source code | `<Parameter>`<br>`      <Name>PublishFullOrderBook</Name>`<br>`      <Value>false</Value>`<br>`</Parameter>` |
| Property names | *`store size`* |
| Property values in text | "true" |
| File names | `wombat.xml` |
| Command-line commands/instruction | `$ uname -rm` |
| Command names | **setMode** |
| Environment variables | `WOMBAT PATH` |
| User-replaceable text | Required: `<yourvalue>`<br>Optional: `[yourvalue]` |
| Command-line prompt | **`root@host#`** |
| Command line output | `2.6.9-55.EL x86_64` |
| Keyboard keys | **[Tab]** |

## Document Revision History

| Date | Version | Description |
|---|---|---|
| 30 Apr 2012 | 1.0 | Initial release. |
| 30 Sep 2012 | 1.1 | Updated to include C# support. |

# Table of Contents

# 1 About OpenMAMDA

Traditional application programming interfaces (APIs) used for market data distribution, such as the ***OpenMAMA API***, provide anonymous field and/or record based interfaces. These APIs require applications to iterate over fields and provide internal logic to determine what type of message the collection of fields represents. For some applications, such as GUIs, the application need not understand the context of the collection of fields, they only need to display each field on a particular part of the screen. Other applications, like program trading, tick capture, analytical, and smart order routing applications need a deeper understanding of the meaning and context of each field in the message, as well as the type of message the update represents.

Classifying market data messages extends beyond determining the messages' high level types such as quotes, trades, order book updates, security status updates, and fundamental data. Applications must further categorize these messages into subtypes: regional vs consolidated quotes, regional vs consolidated trades, trade cancels/errors/corrections, out of order trades etc. The field based approach to accessing market data demands a great deal of complex logic to correctly classify incoming data. ***OpenMAMDA (Middleware Agnostic Market Data) API*** augments ***OpenMAMA*** with a rich set of market data related data structures that address the shortcomings of field-based APIs. The object oriented ***OpenMAMDA*** C++, Java and C# APIs decipher field based data from ***OpenMAMA*** into various types of messages, and provides convenient interfaces for applications to process this data.

***OpenMAMDA*** provides for the following functionality:

- Callback based API that separates incoming data into appropriate classes and subclasses of event.
    - Trades
    - Quotes
    - Security Status
    - Fundamental data
    - Order imbalance information
    - Order Books
    - Options chains
    - News
- Maintains an internal cache and updates it as data arrives.
- Flexible listener based API for simple application development.

**Table 1: Document Terminology**

| Term | Use |
|---|---|
| Object | Refers to C structures and C++/Java/C# object types interchangeably. |
| Symbol | Interchangeable with the concept of a 'topic', a term more commonly used in publish/subscribe messaging. |
| NYSE Technologies Market Data Platform | Refers to both **NYSE Technologies** feed handlers, and feed handlers built using the ***NYSE Technologies FH SDK***. |

## 1.1 The OpenMAMDA API

This C++, Java and C# implementations of the ***OpenMAMDA API*** expose the same top-level objects and interfaces with minor language dependant interface differences. This document describes the C++ API in detail, and notes major differences in the Java and C# APIs where applicable. All examples are illustrated using C++. Specific Java and C# examples are provided where clarity on any differences between the two implementations is required. In general, however, all three language APIs provide an identical interface.

Several ***OpenMAMDA*** objects require a two step initialization process, where the caller first allocates the object and then initializes it by invoking a `create()` method. This is required where memory allocation and creation are two logically different steps, and where properties on an object can be set/ changed prior to creation.

### Example 1: Initialization process

```
MamdaSubscription subsc  = new MamdaSubscription();

... /*Set properties on the subscription*/

subsc->create(...);
```

All ***OpenMAMDA*** applications require three core classes:

- a subscription
- one or more listeners
- one or more handlers

Applications use the pattern to process quotes, trades, order books, news, etc. Subsequent sections of this document address this pattern and its application in detail. All functions/methods in the API across all language implementations and transports exhibit the same behavior, unless otherwise stated.

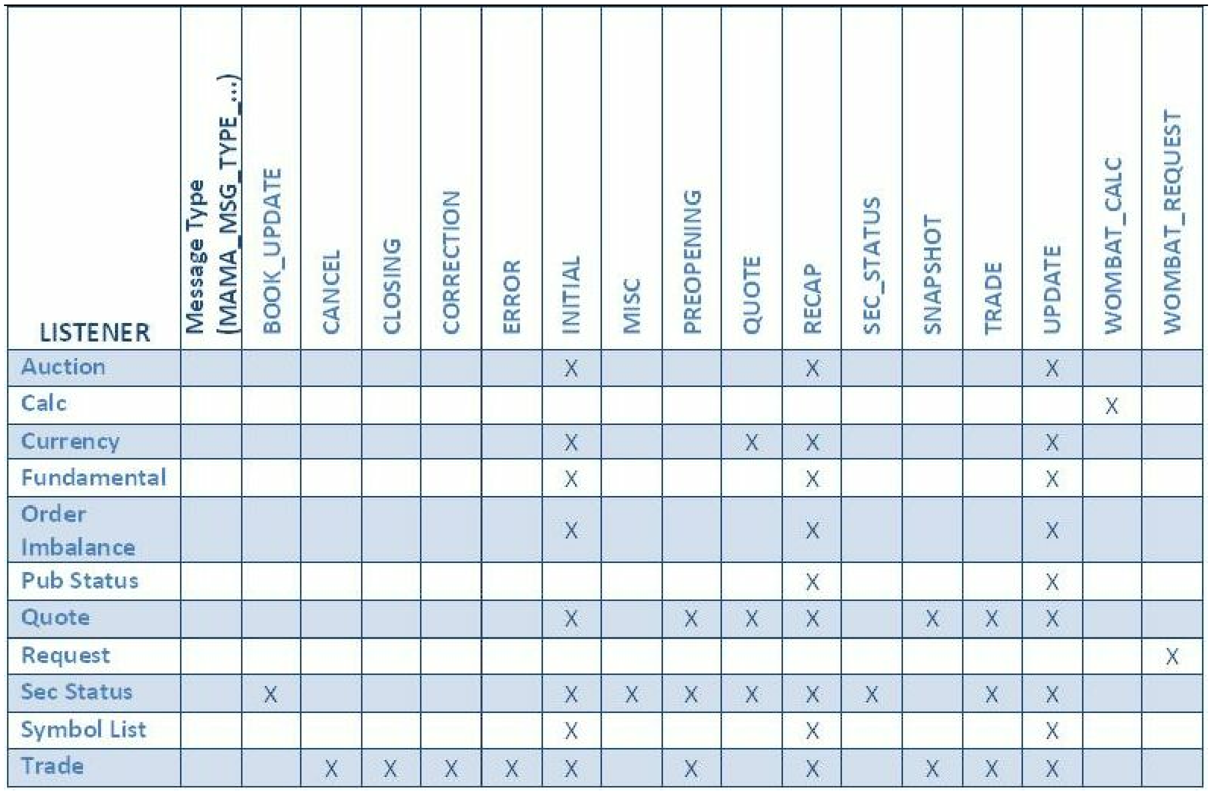
## 1.2 The Core Classes

An ***OpenMAMDA*** application uses the four core divisions of classes detailed in **Table 2: OpenMAMDA Core Classes**.

### Table 2: OpenMAMDA Core Classes

| Class | Description |
|---|---|
| `MamdaSubscription` | This class is used to register interest in an instrument. The MamdaSubscription replaces the need for the MamaSubscription when subscribing to market data on the ***NYSE Technologies Market Data Platform***. To process data returned via a MamdaSubscription instance, one or more MamdaMsgListeners (and usually handlers) must be registered. |
| `Mamda<type>Listener` | The Mamda<type>Listener classes are responsible for identifying the type of data represented by an incoming MamaMsg. Each of the Listener classes, with the exception of the MamdaMsgListener, then identifies the event type for the message type and invokes appropriate event callbacks on any registered Mamda<type>Handler class instances. **Figure 1** details the ***OpenMAMDA*** listener classes and the message types that invoke a callback.<br><br>The Listener objects also maintain an up-to-date cache of all updates |

| | |
|---|---|
| | received so far. This provides the application with ready access to the latest state for each instrument without having to contact the feeds. As a result, listener instances must not be reused across multiple subscriptions.<br><br>Each of the Mamda<type>Listeners derives from the MamdaMsgListener super class, which can be used in its own right for the most basic of applications.<br><br>Examples of MamdaMsgListener sub classes are:<br><br>▪ MamdaQuoteListener<br>▪ MamdaTradeListener<br>▪ MamdaOrderImbalanceListener<br>▪ MamdaSecStatusListener<br>▪ MamdaFundamentalListener<br>▪ MamdaOrderBookListener |
| `Mamda<type>Handler` | The Mamda<type>Handler callback classes are used to propagate event notifications related to the type of messages processed by the parent Mamda<type>Listener class. Concrete implementations of these interfaces exist for each of the market data types supported by the API. |
| `Mamda<type>Fields` | To identify data from messages, ***OpenMAMDA*** uses multiple caches of data dictionary field descriptors, one per market data type, and a single common set of fields. It is the responsibility of an application built using the API to obtain the data dictionary from the platform and to initialize each of the field caches required, depending on the market data type being subscribed to. Each of these caches need only be initialized once for an application. Failure to initialize the appropriate cache for a market data listener can result in unexpected behavior. The MamdaCommonFields class exists in addition to the market data object specific class, which contains field descriptors shared across objects within the API. This cache should be initialized for all applications regardless of the market data being looked at.<br><br>**Note:** The Mamda<Type>Fields classes simply maintain pointers/ references to the field descriptors in the data dictionary. As such, it is necessary to keep a reference to the data dictionary for the lifetime of the ***OpenMAMDA*** based application.<br><br>There is also a reset method for the Mamda<Type>Fields classes. Fields then need to be initialized once more. This allows the fields to be reset during the lifetime of the ***OpenMAMDA*** application without restarting the application. |



| LISTENER | Message Type (MAMA_MSG_TYPE_...) | BOOK_UPDATE | CANCEL | CLOSING | CORRECTION | ERROR | INITIAL | MISC | PREOPENING | QUOTE | RECAP | SEC_STATUS | SNAPSHOT | TRADE | UPDATE | WOMBAT_CALC | WOMBAT_REQUEST |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Auction | | | | | | | X | | | | X | | | | X | | |
| Calc | | | | | | | | | | | | | | | | X | |
| Currency | | | | | | | X | | | X | X | | | | X | | |
| Fundamental | | | | | | | X | | | | X | | | | X | | |
| Order Imbalance | | | | | | | X | | | | X | | | | X | | |
| Pub Status | | | | | | | | | | | X | | | | X | | |
| Quote | | | | | | | X | | X | X | X | | X | X | X | | |
| Request | | | | | | | | | | | | | | | | | X |
| Sec Status | | X | | | | | X | X | X | X | X | X | | X | X | | |
| Symbol List | | | | | | | X | | | | X | | | | X | | |
| Trade | | | X | X | X | X | X | | X | | X | | X | X | X | | |

**Figure 1: OpenMAMDA listener classes and the OpenMAMA message types that invoke a callback**

## 1.3 The Libraries

Each of the key functional areas of the ***OpenMAMDA API*** are bundled in a separate library, as listed in **Table 3: OpenMAMDA API Libraries**.

**Table 3: OpenMAMDA API Libraries**

| Library | Description |
|---|---|
| `libmamda.[a][so]/libmamda.jar` | The core foundation classes of the API, including all record based listeners and event/data objects. |
| `libmamdabook.[a][so]/ mamda_book.jar` | All classes required for the ***OpenMAMDA*** Order Book processing. Includes atomic books for the C++ API. Depends on libmamda. |
| `libmamdaoptions.[a][so]/ mamda_options.jar` | All classes required for the ***OpenMAMDA*** options processing. Depends on libmamda. |
| `libmamdanews.[a][so]` | All classes required for the ***OpenMAMDA*** news processing. Depends on libmamda. (C++ only) |

## 1.4 Exception Handling

All language ***OpenMAMDA APIs*** propagate errors through a combination of exceptions and callbacks. ***OpenMAMDA*** uses callbacks in cases where catching exceptions is not practical, such as message loops. The C++ API throws ***OpenMAMDA*** specific exceptions that derive from the **C++ Standard Library** exception classes such as `invalid_argument`. The Java API throws exceptions derived from `java.lang.RuntimeException` and `com.wombat.mama.MamaException`. The C# api throws exceptions derived from `System.Exception`.

**Table 4: Java API Exception Handling**

| Exception | Description |
|---|---|
| `MamdaDataException` | Throws a runtime exception with the specified cause and a detail message. Both the cause and message may contain NULL values. |
| `MamdaOrderBookException` | Throws an exception due to inconsistency in the Order Book. This may be due to a number of reasons, such as the feed sending inconsistent data, undetected missed data, or the program manipulating the Order Book independently of the MamdaOrderBookListener. |
| `MamdaOrderBookInvalidEntryException` | Throws an exception due to an attempt to try update or delete an entry that does not exist. This will also throw an exception when attempting to access the price on an entry which does not have an associated price level. |
| `MamdaOrderBookMissingEntryException` | Throws an exception when there is an attempt to access an entry in the MamdaOrderBookEntryManager that does not exist. |
| `MamdaOrderBookDuplicateEntryException` | Throws an exception when there is an attempt to add an entry in the MamdaOrderBookEntryManager that already exists. |

## 1.5 Example Programs

Source code and sample build files for example programs that illustrate how to use the ***API*** are available in the `examples/mamda` directory of the ***OpenMAMDA API*** distribution.

**Table 5: OpenMAMDA API Example Programs**

| Example | Description |
|---|---|
| mamdalisten/MamdaListen | The most basic ***OpenMAMDA*** application that uses only the MamdaMsgListener. Prints the contents of each message received to the console window. For example: `mamdalisten -S NASDAQ -s MSFT -tport tport name` |
| quoteticker/MamdaQuoteTicker | Illustrates using the MamdaQuoteListener and related classes. Prints bid/ask size and price for each quote update received. For example: `quoteticker -S NASDAQ -s MSFT -tport tport name` |
| tradeticker/MamdaTradeTicker | Illustrates using the MamdaTradeListener and related classes. Prints trade related information for each trade update received. For example: `tradeticker -S NASDAQ -s MSFT -tport tport name` |
| comboticker/MamdaComboTicker | Illustrates the use of multiple listeners with a single |

| Example | Description |
|---|---|
| | MamdaSubscription. In this case, the application is using both the MamdaQuoteListener and the MamdaTradeListener. For example: `comboticker -S NASDAQ -s MSFT -tport tport_name` |
| multiparticker/MamdaMultiPartTicker | Illustrates the use of the MamdaMultiParticipantManager. The example registers a quote and trade listener/handler with each separate participant symbol in the group subscription. For example: `multiparticker -S NASDAQ -s MSFT.GRP -tport tport_name` |
| multisecurityticker/ MamdaMultiSecurityTicker | Illustrates the use of the MamdaMultiSecurityManager. The example registers a quote and trade listener/handler with each separate symbol in the group subscription. For example: `multisecurityticker -S NASDAQ -s NASDAQ_ALL -tport tport_name` |
| optionchainer/ MamdaOptionChainExample | Illustrates the use of the ***OpenMAMDA*** Option chaining API. The example creates a quote and trade listener/handler for each option contract as they are added to the chain. For example: `optionchainer -S NASDAQ -OS OPRA -s MSFT -tport tport_name` Where OS is the [O]ption [S]ymbol Namespace if different than the underlying namespace. |
| optionview/ MamdaOptionChainViewExample | Illustrates the ***OpenMAMDA*** option chain "view" processing. |
| secstatuslisten/MamdaSecStatusTicker | Illustrates the use of the MamdaSecStatusListener and related classes. |
| bookticker/MamdaBookTicker | Illustrates the use of the ***OpenMAMDA*** Order Book API. Demonstrates how to access Order Book price levels and entries in the handler callbacks. For example: `bookticker -S ARCA -s bMSFT.ARCA -tport tport_name` |
| bookviewer (C++ only) | As with bookticker, but uses ncurses to provide a graphical representation of the Order Book. |
| atomicbookticker | Illustrates the use of the 'Atomic' Order Books in the API. For example: `atomicbookticker -S ARCA -s bMSFT.ARCA -tport tport_name` |

# 2 Creating an OpenMAMDA Application

This section describes the steps required to create a simple application using the ***OpenMAMDA API***. It highlights how the main objects within the API interact, and illustrates the basic steps required to write any ***OpenMAMDA*** based application, regardless of which market data objects from the API are being used.

## 2.1 The Role of OpenMAMA in an OpenMAMDA Application

The ***OpenMAMDA API*** adds to the data processing aspect of the ***OpenMAMA API***. As such, the ***OpenMAMDA API*** cannot be used in isolation. Any ***OpenMAMDA*** application must use the core ***OpenMAMA*** functionality to create transports, control event dispatching, add interval-based timer activities, or add IO-based activities, effectively putting together the building blocks required to access data on the ***NYSE Technologies Market Data Platform***.

As with an ***OpenMAMA*** application, ***OpenMAMDA*** applications are free to use multiple event queues and threads to distribute the processing of data. See the *OpenMAMA Developers Guide* for more detail on event queues and event dispatching. For example, an ***OpenMAMDA*** application is responsible for obtaining the MamaDataDictionary and using it to populate the Mamda<type>Fields field descriptor caches.

## 2.2 The Building Blocks of an OpenMAMDA Application

The following code example illustrates the basic steps required to build a simple ***OpenMAMDA*** application. In this case, the code is simply registering a concrete instance of the MamdaMsgListener super class with a single MamdaSubscription to a single instrument. The code sample has no practical purpose as it provides no benefit over subscribing to data using the ***OpenMAMA API***, where sub classes of the MamdaMsgListener provide the data identification, caching and event propagation. The code serves to illustrate the relationship between the MamdaSubscription and the message listeners, and demonstrates how all ***OpenMAMDA*** based applications are initially constructed.

**Example 2: Building a simple OpenMAMDA application**

```
...
class ListenerCallback : public MamdaMsgListener 1.
{
    void onMsg (
        MamdaSubscription*   subscription,
        const MamaMsg&       msg,
        short                msgType)
    {
        /*Process message*/
    }
}
...

Mama::open() 2.

ListenerCallback ticker;

MamdaSubscription* aSubscription = new MamdaSubscription ();

aSubscription->addMsgListener   (&ticker); 3.

aSubscription->create (queue, source, "MSFT"); 4.
```

```
...

Mama::start (bridge); 5.
```

1. Create a sub-class of the MamdaMsgListener super class that will receive callbacks whenever a message arrives for the associated subscription with which it is registered. In this case the same listener instance could be shared across multiple subscriptions as the class does not cache data. In all other cases a unique listener instance must be registered with each subscription created. Within the market data specific implementations of the MamdaMsgListener, message processing occurs within the `onMsg()` callback, ultimately resulting in specific handler event callbacks being invoked.
2. As with the ***OpenMAMA API***, the `Mama.open()` must be the first method called, initializing the internal state of the API.
3. Listener instances are registered with a MamdaSubscription. Multiple listeners can be registered with a single subscription **Note**: Market data listeners cannot be shared across subscriptions.
4. The NULL parameter indicates the use of the internal default dispatch queue (see *Section 6: Events and Queues* in the *OpenMAMA Developers Guide* for detail on event queues in ***OpenMAMA***). Creation of the MamaSource is not illustrated here. If multi-threading is being used to distribute processing load within an application multiple queues/threads can be used.
5. As with the ***OpenMAMA API***, the `Mama::start()` method starts dispatching on the default internal ***OpenMAMA*** event queue. The creation of the MamaBridge is outside the scope of this example.

This code sample does not show the use of the Mamda<data type>Handler callbacks. These will be discussed in detail in their relevant sections.

# 3 Simple Market Data Classes

This section describes the non-structured, record based, simple market data objects supported within the API. In each case the data is presented on a subtype event basis via callbacks on the relevant handler classes. The event classes provide a suite of accessors for gaining access to the data describing the event.

Each data type supports an update and a recap event object. Other event data types are provided depending on the market data object in question.

**Note** All callbacks provide access to the message that resulted in the callback being invoked. This provides developers access to additional fields that may not be available through the ***OpenMAMDA*** data event objects.

## 3.1 Quotes

The MamdaQuoteListener and related classes are provided to facilitate the processing of Quote related updates on the ***NYSE Technologies Market Data Platform***. Developers provide their own implementation of the MamdaQuoteHandler interface, and will be delivered notifications for quotes and quote closing prices.

The MamdaQuoteListener class caches quote-related field values. One of the benefits of this feature is that caching of these fields makes it possible to provide complete quote-related callbacks, even when the publisher (e.g., feed handler) is only publishing deltas containing modified fields.

The fields that are available completely describe a quote. Non-standard, exchange-specific fields can be obtained using MamaMsg.

**Example 3: Processing quotes** illustrates the steps required for processing quotes within an ***OpenMAMDA*** application.

### Example 3: Processing quotes

```
...
class QuoteTicker : public MamdaQuoteHandler 1.
{
public:
    virtual ~QuoteTicker () {}

    void onQuoteRecap (
        MamdaSubscription*      subscription,
        MamdaQuoteListener&     listener,
        const MamaMsg&          msg,
        const MamdaQuoteRecap&  recap)
    {
         //process the quote recap
    }

    void onQuoteUpdate (
        MamdaSubscription*      subscription,
        MamdaQuoteListener&     listener,
        const MamaMsg&          msg,
        const MamdaQuoteUpdate& quote,
        const MamdaQuoteRecap&  recap)
    {
         //process the quote update
    }

    void onQuoteGap (
        MamdaSubscription*      subscription,
        MamdaQuoteListener&     listener,
        const MamaMsg&          msg,
        const MamdaQuoteGap&    event,
        const MamdaQuoteRecap&  recap)
    {
         //respond to notification of gap in quote messages.
    }

    void onQuoteClosing (
        MamdaSubscription*        subscription,
        MamdaQuoteListener&       listener,
        const MamaMsg&            msg,
        const MamdaQuoteClosing&  event,
        const MamdaQuoteRecap&    recap)
    {
         //process the closing quote.
    }

    void onQuoteOutOfSequence (
        MamdaSubscription*              subscription,
        MamdaQuoteListener&             listener,
        const MamaMsg&                  msg,
        const MamdaQuoteOutOfSequence&  event,
        const MamdaQuoteRecap&          recap)
    {
        //process an out-of-sequence quote
    }

    void onQuotePossiblyDuplicate (
        MamdaSubscription*                  subscription,
        MamdaQuoteListener&                 listener,
        const MamaMsg&                      msg,
        const MamdaQuotePossiblyDuplicate&  event,
        const MamdaQuoteRecap&              recap)
    {
        //process a possibly duplicate quote
    }
}
...

Mama::open()

// Obtain the data dictionary from the platform

MamdaCommonFields::setDictionary (dictionary);
MamdaQuoteFields::setDictionary (dictionary); 2.

MamdaQuoteListener quoteListner;
```

```
QuoteTicker quoteTicker;

MamdaSubscription* aSubscription = new MamdaSubscription ();

quoteListener.addHandler (&quoteTicker);

aSubscription->addMsgListener   (&quoteListner); 3.

//Create a MamaSource for the data feed

aSubscription->create (queue, source, "MSFT"); 4.

...

Mama::start (bridge); 5.
```

1. Provide a sub-class of the MamdaQuoteHandler interface. This interface provides callbacks for quote-related events and access to all the cached quote data.
2. Unlike the basic example using the MamdaMsgListener, all market data listener implementations require the use of a cache of MamaFieldDescriptors. These are obtained from a valid MamaDictionary. The MamdaCommonFields and MamdaQuoteFields classes must be initialized with the data dictionary.

**Note** The obtaining of the data dictionary is not illustrated in **Example 3: Processing quotes**.

3. The quote listener is registered with the subscription as with the MamdaMsgListener instance in **Example 2: Building a simple OpenMAMDA application**.
4. Create the subscription. In this case the subscription is being created on the default event queue.

**Note** The creation of the MamaSource and the MamaQueue is not illustrated in **Example 3: Processing quotes**.

5. Start dispatching on the default event queue. At this point subscriptions will be throttled and data dispatching will commence. The creation of the MamaBridge is not included in **Example 3: Processing quotes**.

### 3.1.1 Event Notifications

**Table 6: Event Notification Conditions - Quotes** describes the circumstances under which each of the MamdaQuoteHandler event callbacks are invoked.

**Table 6: Event Notification Conditions - Quotes**

| Callback | Condition |
|---|---|
| `onQuoteRecap` | Invoked in response to an initial value upon subscription creation. Also invoked upon receipt of an ***OpenMAMA*** level recap during a data quality event. Invocation of the `onQuoteRecap()` callback indicates that the state of the cache has been refreshed with the latest snapshot available from the platform. |
| `onQuoteUpdate` | Invoked in response to a 'delta' tick update being received for the subscribed instrument. As the API maintains a 'latest value' cache, all fields describing the quote are available. It is not currently possible to identify only the fields that have changed as a result of the update. |
| `onQuoteGap` | Invoked in response to a gap being detected in the quote count. This is separate to the message sequence number checking at the ***OpenMAMA*** level. The ***OpenMAMA*** level sequence number checking will also include non-quote updates for the same symbol, such as Trades. |
| `onQuoteClosing` | Invoked when the closing quote for the data is received by the API. |
| `onQuoteOutOfSequence` | Invoked for a message marked as out of sequence. The Listener must be configured to check the MsgQualifier, i.e., call `setControlProcessingByMsgQual()` on the listener, passing a value of "true". This feature must also be enabled on the feed handlers in order to calculate this information and to send it along with messages. Typically applicable after a fault tolerant takeover. |
| `onQuotePossiblyDuplicate` | Invoked for a message that is marked as possibly duplicate. The Listener must be configured to check the MsgQualifier, i.e., call `setControlProcessingByMsgQual()` on the listener, passing a value of "true". This feature must also be enabled on the feed handlers in order to calculate this information and to send it along with messages. Typically applicable after a fault tolerant takeover. |

### 3.1.2 Accessing Quote Data

The event objects passed to the application in the MamdaQuoteHandler callbacks provide access to the underlying quote related data.

**Table 7: Event Objects for Quotes**

| Event Object | Description |
|---|---|
| `MamdaQuoteUpdate` | Provides access to quote related fields that can change during a trading day, or which are sent as part of a delta from a publisher on the platform. Updates are available as ticks that arrive intraday from a publisher. |
| `MamdaQuoteRecap` | Provides access to fields that do not change during a trading day, or are derived from data in the update, and that are additional to the fields available from the `MamdaQuoteUpdate`, via the following methods:<br><br>`getQuoteCount` `getAskLow` `getAskPrevCloseDate`<br>`getAskCloseDate` `getAskClosePrice` `getBidLow`<br>`getBidHigh` `getBidPrevCloseDate` `getBidPrevClosePrice`<br>`getBidCloseDate` `getBidClosePrice`<br><br>It also provides access to the same fields as `MamdaQuoteUpdate`. |
| `MamdaQuoteGap` | Provides access to the start and end gap sequence numbers related to the quotes for the instrument. |
| `MamdaQuoteClosing` | Provides access to fields describing the closing quote received by the API. |

## 3.2 Trades

The MamdaTradeListener and related classes are provided to facilitate the processing of Trade related updates on the ***NYSE Technologies Market Data Platform***. Developers provide their own implementation of the MamdaTradeHandler interface, and will be delivered notifications for trade reports.

The MamdaTradeListener class caches trade-related field values. One of the benefits of this feature is that caching of these fields makes it possible to provide complete trade-related callbacks, even when the publisher (e.g., feed handler) is only publishing deltas containing modified fields.

The fields that are available completely describe a trade. Non-standard, exchange-specific fields can be obtained using MamaMsg.

**Example 4: Processing trades** illustrates the steps required for processing trades within an ***OpenMAMDA*** application.

### Example 4: Processing trades

```
...
class TradeTicker : public MamdaTradeHandler 1.
{
public:
    virtual ~TradeTicker () {}

    void onTradeRecap (
        MamdaSubscription*      subscription,
        MamdaTradeListener&     listener,
        const MamaMsg&          msg,
        const MamdaTradeRecap&  recap)
    {
        //process the trade recap
    }

    void onTradeReport (
        MamdaSubscription*      subscription,
        MamdaTradeListener&     listener,
        const MamaMsg&          msg,
        const MamdaTradeReport& event,
        const MamdaTradeRecap&  recap)
    {
        //process the trade report
    }

   void onTradeCancelOrError (
        MamdaSubscription*              subscription,
        MamdaTradeListener&             listener,
        const MamaMsg&                  msg,
        const MamdaTradeCancelOrError&  event,
        const MamdaTradeRecap&          recap)
    {
        //process the trade cancel or error
    }

    void onTradeCorrection (
        MamdaSubscription*              subscription,
        MamdaTradeListener&             listener,
        const MamaMsg&                  msg,
        const MamdaTradeCorrection&     event,
        const MamdaTradeRecap&          recap)
    {
        //process the trade correction
    }

    void onTradeClosing (
        MamdaSubscription*              subscription,
        MamdaTradeListener&             listener,
        const MamaMsg&                  msg,
        const MamdaTradeClosing&        event,
        const MamdaTradeRecap&          recap)
    {
        //process the closing trade
    }

    void onTradeOutOfSequence (
        MamdaSubscription*              subscription,
        MamdaTradeListener&             listener,
        const MamaMsg&                  msg,
        const MamdaTradeOutOfSequence&  event,
        const MamdaTradeRecap&          recap)
    {
        //process the out-of-sequence trade
    }

    void onTradePossiblyDuplicate (
        MamdaSubscription*                  subscription,
        MamdaTradeListener&                 listener,
        const MamaMsg&                      msg,
        const MamdaTradePossiblyDuplicate&  event,
        const MamdaTradeRecap&              recap)
    {
        //process the possibly duplicate trade
    }
}
...

Mama::open()

// Obtain the data dictionary from the platform

MamdaCommonFields::setDictionary (dictionary);
MamdaTradeFields::setDictionary (dictionary); 2.

MamdaTradeListener tradeListner;
TradeTicker        tradeTicker;

MamdaSubscription* aSubscription = new MamdaSubscription ();

tradeListener.addHandler (&tradeTicker);

aSubscription->addMsgListener   (&tradeListner); 3.

//Create a MamaSource for the data feed

aSubscription->create (queue, source, "MSFT"); 4.
```

```
...

Mama::start (bridge); 5:
```

1. Provide a sub-class of the MamdaTradeHandler interface. This interface provides callbacks for trade-related events and access to all the cached quote data.
2. Unlike the basic example using the MamdaMsgListener, all market data listener implementations require the use of a cache of MamaFieldDescriptors. These are obtained from a valid MamaDictionary.

The MamdaCommonFields and MamdaTradeFields classes must be initialized with the data dictionary.

**Note** The obtaining of the data dictionary is not illustrated in **Example 4: Processing trades**.

3. The trade listener is registered with the subscription.
4. Create the subscription. In this case the subscription is being created on the default event queue.

**Note** The creation of the MamaSource is not illustrated in **Example 4: Processing trades**.

5. Start dispatching on the default event queue. At this point subscriptions will be throttled and data dispatching will commence.

### 3.2.1 Event Notifications

**Table 8: Event Notification Conditions - Trades** describes the circumstances under which each of the MamdaTradeHandler event callbacks are invoked.

**Table 8: Event Notification Conditions - Trades**

| Callback | Condition |
|---|---|
| `onTradeRecap` | Invoked in response to an initial value upon subscription creation. Also invoked upon receipt of an ***OpenMAMA*** level recap during a data quality event. Invocation of the `onTradeRecap()` callback indicates that the state of the cache has been refreshed with the latest snapshot available from the platform. |
| `onTradeReport` | Invoked in response to a trade update being received from the platform |
| `onTradeCancelOrError` | Invoked in response to a trade cancel or trade error being reported. |
| `onTradeCorrection` | Invoked when a trade correction is reported. |
| `onTradeClosing` | Invoked in response to the closing report being received. |
| `onTradeGap` | Invoked when a gap in trade reports is discovered. |
| `onTradeOutOfSequence` | Invoked for a message marked as out of sequence. The Listener must be configured to check the MsgQualifier, i.e., call `setControlProcessingByMsgQual()` on the listener, passing a value of "true". This feature must also be enabled on the feed handlers in order to calculate this information and to send it along with messages. Typically applicable after a fault tolerant takeover. |

| | |
|---|---|
| `onTradePossiblyDuplicate` | Invoked for a message that is marked as possibly duplicate. The Listener must be configured to check the MsgQualifier, i.e., call `setControlProcessingByMsgQual()` on the listener, passing a value of "true". This feature must also be enabled on the feed handlers in order to calculate this information and to send it along with messages. Typically applicable after a fault tolerant takeover. |

### 3.2.2 Accessing Trade Data

The event objects passed to the application in the MamdaTradeHandler callbacks provide access to the underlying trade-related data.

**Table 9: Event Objects for Trades**

| Event Object | Description |
|---|---|
| `MamdaTradeReport` | Provides access to trade-related fields that can change during a trading day, or which are sent as part of a delta from a publisher on the platform. Both regular and irregular trades are reported as a trade report. An irregular trade will not have updated the official last price or the intra-day high/low values within the cache. Whether the trade is regular or irregular is identified via the `getIsIrregular()` method. Updates are available as ticks arrive intraday from a publisher. |
| `MamdaTradeRecap` | Provides access to fields that do not change during a trading day, or are derived from data in the update, and that are additional to the fields available from the `MamdaTradeReport`, via the following methods:<br><br>`getQuoteCount` `getAskLow` `getAskPrevCloseDate`<br>`getAskCloseDate` `getAskClosePrice` `getBidLow`<br>`getBidHigh` `getBidPrevCloseDate` `getBidPrevClosePrice`<br>`getBidCloseDate` `getBidClosePrice`<br><br>It also provides access to the same fields as `MamdaTradeReport`.<br><br>The `MamdaTradeRecap` also provides access to the following data: |

| Method | Data |
|---|---|
| `getLastPrice()` | Last regular price |
| `getLastTime()` | Last regular time |
| `getLastPartId()` | Last regular participant ID |
| `getLastVolume()` | Last regular volume |
| `getIrregPrice()` | Last irregular price |
| `getIrregTime()` | Last irregular time |
| `getIrregPartId()` | Last irregular participant ID |
| `getIrregVolume()` | Last irregular volume |

These sets of methods are distinct from the `getTradePrice()`, `getTradeTime()`, `getTradePartId()`, and `getTradeVolume()` methods in that the cached fields they provide access to are only updated for regular or irregular trades respectively. The `getTradeX()` methods can refer to either regular or irregular values, with the regularity of the trade being established by the value of the `getIsIrregular()` method, as noted above.

| | |
|---|---|
| `MamdaTradeCancelOrError` | Provides access to data describing a trade cancellation or error. Cancels are distinguished from errors via the `getIsCancel()` method. |
| `MamdaTradeCorrection` | Provides access to fields describing a correction to a previous trade report. |
| `MamdaTradeClosing` | Provides access to the closing price for the days trading. The object also indicates whether the closing is indicative or not. |

## 3.3 Order Imbalance

MamdaOrderImbalanceListener specializes in handling imbalance order updates. An imbalance order occurs when there are too many orders of a particular type, either buy, sell or limit, for listed securities, and not enough matching orders are received by an exchange.

## 3.4 Security Status

MamdaSecurityStatusListener is a class that specializes in handling security status updates.

## 3.5 Fundamental Data

MamdaFundamentalListener is a class that specializes in handling fundamental equity pricing/analysis attributes, indicators and ratios.

# 4 Structured Market Data Classes

Structured data support is a core feature of the ***OpenMAMDA API***. Structured data support allows ***OpenMAMDA*** to provide depth to the market data and event types by adding industry standard structure. The ***API*** does this by providing powerful, flexible, and simple to use object-oriented views of the market data.

## 4.1 Order Books

This section describes how the ***OpenMAMDA API*** manages and propagates order books on the ***NYSE Technologies Market Data Platform***. It discusses the various formats that ***OpenMAMDA*** employs to describe order book data, and describes how the ***API*** represents and exposes order books.

### 4.1.1 The NYSE Technologies Order Book

**NYSE Technologies** manages and publishes normalized order book data from numerous disparate data sources, such as Exchanges, ECN, and order book aggregators. Using the **NYSE Technologies** fully structured order book format, the ***OpenMAMDA API*** delivers both the full depth of book and aggregated book (at the price level) for processing in a similar manner. The **NYSE Technologies** order book provides customers with easy access to full market depth for any given security.

This section provides an overview of how the ***OpenMAMDA API*** logically stores order book data, and terminology for describing that data.

#### Types of Order Book

**NYSE Technologies** uses the terms 'single-participant' and 'multi-participant' to categorize two possible views of order/quote data provided by order books.

Single participant books provide details in the book for a single participant, such as an Exchange, ECN, or Market Maker. These single participant books typically contain information for each individual order in the book. For example, NYSE ARCA sends a single-participant book with all orders for each security.

Multi-participant order books aggregate order book information from a number of participants trading the same security. For these books the lowest level of granularity available is the individual participants' rolled-up position at a given price, i.e. the total number of orders and the cumulative size of those orders at a price, as opposed to details on each of the individual orders on the participants books for that price. The NASDAQ Totalview feed is an example of a multi-participant order book. The ***NYSE Technologies SuperBook*** product provides an aggregated order book across any number of individual source feeds that trade the same instrument.

**NYSE Technologies** also introduces the 'pseudo order book' concept whereby the data from regular quote-based feeds can be represented as an order book.

The **NYSE Technologies** order book comprises a collection of price levels for each side of the book (bid/ask-offer), each price level optionally comprising a collection of order entries.

#### Order Book Entry

The basic unit of a **NYSE Technologies** order book is the 'Order Book Entry'. This term is used rather than the term 'Order' as, depending on the source of the order book information, details on individual orders within a book may not be available.

In the case of single-participant order books, the entry represents an individual buy or sell order within the book, and is generally identified by a unique order ID.

For multi-participant order books the entry represents an individual participant's consolidated position for a specific price on either the buy or sell side of the book. Individual orders at a price are not available. The entry for multi-participant order books is generally identified by a participant ID, such as the Market Maker ID, or the Exchange ID.

For the pseudo order books, the order entry represents either a bid or ask quote instead of an individual order. This order book view of quote data can be useful for monitoring market depth.

Attributes describing an entry include the following: entry ID (Order ID or Market Participant ID), entry size, entry time stamp.

The order book entry is not always available for **NYSE Technologies** order books. Certain exchanges, such as CME, do not provide order/entry information. In this case order books comprise bid and ask price levels without any depth.

### Order Book Price Level

A price level represents a collection of order entries at the same price on a particular side (bid or ask) of an order book. All order entries at the same price for a particular side are associated with the level for that price. In addition, the price level entity provides summary information on the entries it contains, including the number of entries at that level and the total number of shares/lots.

Attributes describing a price level include: total size, the number of entries at the price level, price level time stamp and details of individual entries at each price level.

### Order Book Representations

The **NYSE Technologies** platform represents order book data in the following formats:

- Summary Book (s) record: A record-based message containing the Top-N price levels from an order book, configured at the book publisher. This record does not contain any entry information.
- Full Book (f) record: A record-based message containing the Top-N order book entries from an order book.

Both of these records are windows onto, and are derived from, the underlying structured order book, Order Book Format, maintained by the feed handlers. The Order Book Format publishes the full depth of an order book, and is the only format supported by the ***OpenMAMDA API***. The default structured book symbology includes the instrument's identifying symbol prefixed with a "b", for example. for the Microsoft order book on NYSE ARCA the default subscription symbol is bMSFT.ARCA.

### 4.1.2 Order Books in OpenMAMDA

The ***OpenMAMDA API*** provides two sets of classes by which order book data can be obtained and processed:

- MamdaOrderBookListener and related classes: Provide full order book maintenance and caching, access to order book deltas as they arrive, and access to the full book with the deltas applied.
- MamdaBookAtomicListener and related classes: Provide a simple interface for accessing order book deltas only, without caching a local full book.

**Note** Avis does not support vector messages, which are required in the payload for order books.

#### Order Book Limit and Market Orders

Both Limit and Market order data can be processed in the MamdaOrderBookListener and MamdaBookAtomicListener classes. The processing of Market orders is disabled by default in the MamdaOrderBookListener class, while it is on by default in the MamdaBookAtomicListener class. Details of functionality specific to processing Market Orders is given in the following sections.

#### Cached Order Books (MamdaOrderBookListener)

The MamdaOrderBookListener class handles order book updates sent using the **NYSE Technologies** Order Book Format. Developers provide a concrete subclass implementation of the MamdaOrderBookHandler interface, which receives notifications for order book recaps and deltas. Notifications for order book deltas include the delta itself, as well as the full order book with the applied deltas. An obvious application for this ***OpenMAMDA*** class is any kind of program trading application that looks at depth of book.

The MamdaOrderBookListener class also caches the full order book. Caching of these fields allows the ***OpenMAMDA API*** to provide full-book related callbacks, even when the publisher, such as a feed handler, only publishes deltas containing modified fields.

***OpenMAMDA*** physically represents order books with a structure similar to the logical order book structure described previously within this section. The MamdaOrderBook class represents the order book itself, and provides access to constituent price levels and, ultimately, if supported, order book entries. The ***OpenMAMDA API*** describes order book price levels using the MamdaOrderBookPriceLevel class and entries using the MamdaOrderBookEntry class.

##### *The Order Book Classes*

The MamdaOrderBook class represents the full order book. The table below describes the most commonly used methods on the order book class. Many of the methods on the order book are for use internally by the MamdaOrderBookListener and are not described here. For details on all order book methods, see the API documentation.

**Table 10: MamdaOrderBook**

| Method | Description |
|---|---|
| `getSymbol()` | Get the subscription symbol for the book. |
| `getSource()` | Get the subscription source for the book. |
| `copy()` | Create a deep copy of the order book. |

| Method | Description |
|---|---|
| | **Note**: This method should not be invoked on a per event basis, because copying may add a considerable amount of additional CPU overhead, reducing an applications ability to process events in a timely fashion. |
| `getTotalNumLevels()` | Get the total number of levels within the order book, which includes the levels from both sides of the book - bid and ask. |
| `getNumBidLevels()` | Get the number of price levels on the bid side of the book. |
| `getNumAskLevels()` | Get the number of price levels on the ask side of the book. |
| `getBookTime()` | The time of the last update to the order book. |
| `getLevelAtPrice()` | Get the price level for a specified price and side of the book. |
| `getLevelAtPosition()` | Get the price level at the specified indexed position within the book. |
| `getEntryAtPosition()` | Get the order book entry at the specified indexed position within the book. |
| `getBidMarketOrders()` | Get the order book market level for the bid side. |
| `getAskMarketOrders()` | Get the order book market level for the ask side. |
| `getMarketOrdersSide()` | Get the market orders for the specified side. Will return NULL if no market orders exist in the book. |
| `getOrCreateMarketOrdersSide()` | Get the market orders for the specified side. Will create an empty level if none exist. |
| `applyMarketOrder()` | Apply a market order delta to this book, for both simple and complex deltas. |
| `dump()` | Dump the contents of the order book to the specified stream. |

The following methods provide the ability to iterate over all price levels and entries within a book using STL style iterators. In C++ each method also has an overloaded version that returns a const object. In the Java API the methods are named `bidIterator()`, `bidEntryIterator()`, etc, and they return a Java Iterator.

**Table 11: MamdaOrderBook Iterator Methods**

| Method | Description |
|---|---|
| `bidBegin()` | Start iterator for all bid price levels within the book. |
| `bidEnd()` | End iterator for all bid price levels within the book. |
| `askBegin()` | Start iterator for all ask price levels within the book. |
| `askEnd()` | End iterator for all ask price levels within the book. |
| `bidEntryBegin()` | Start iterator for all bid order entries within the book. |
| `bidEntryEnd()` | End iterator for all bid order entries within the book. |
| `askEntryBegin()` | Start iterator for all ask order entries within the book. |
| `askEntryEnd()` | End iterator for all ask order entries within the book. |

The MamdaOrderBookPriceLevel class provides aggregate details on all entries within the level, and also provides access to all of the entries at that level, if supported. As with the MamdaOrderBook, only methods that are intended for use directly by a subscribing application are detailed. Details for all other methods can be found in the API documentation.

**Table 12: MamdaOrderBookPriceLevel**

| Method/Enum | Description |
|---|---|
| enum Action | If part of a delta, this enum indicates:<br>▪ how the delta should be applied to an order book<br>▪ how this delta was applied to the cached order book |

| Method/Enum | Description |
|---|---|
| | Valid values are:<br>MAMDA_BOOK_ACTION_ADD<br>MAMDA_BOOK_ACTION_UPDATE<br>MAMDA_BOOK_ACTION_DELETE<br>MAMDA_BOOK_ACTION_UNKNOWN |
| enum Side | The side of the book to which the price level belongs.<br><br>Valid values are:<br>MAMDA_BOOK_SIDE_BID<br>MAMDA_BOOK_SIDE_ASK<br>MAMDA_BOOK_SIDE_UNKNOWN |
| enum Reason | The reason for the update to the book. This information is sent from the feeds if available.<br><br>Valid values are:<br>MAMDA_BOOK_REASON_MODIFY<br>MAMDA_BOOK_REASON_CANCEL<br>MAMDA_BOOK_REASON_TRADE<br>MAMDA_BOOK_REASON_CLOSE<br>MAMDA_BOOK_REASON_DROP<br>MAMDA_BOOK_REASON_MISC<br>MAMDA_BOOK_REASON_UNKNOWN |
| enum OrderType | The order type for level. This is either a limit order or market order.<br><br>Valid values are:<br>MAMDA_BOOK_LEVEL_LIMIT<br>MAMDA_BOOK_LEVEL_MARKET<br>MAMDA_BOOK_LEVEL_UNKNOWN |
| `copy()` | Get a deep copy of the price level. |
| `getPrice()` | Get the price for this level. |
| `getSize()` | Get the total size, across all entries, at this level. |
| `getSizeChange()` | Get the size change for this level. This attribute only applies to levels obtained as part of a delta. For full order books this field will be equal to the size of the price level. |
| `getNumEntries()` | Get the number of entries at this level. The number of entries returned here may not equal the number of entries that are available from the level if the feed does not provide entries, or the number of entries is restricted or the price level has been obtained from a delta update. |
| `empty()` | Returns "true" if the level contains no entries. |
| `getSize()` | Get the size of the book to which the level belongs. |
| `getAction()` | The action which should be applied for this delta if maintaining an external order book. If the level is from a full order book this will return MAMDA_BOOK_ACTION_ADD. |
| `getTime()` | The time at which the level was last updated. For a delta level, it represents the event time. |
| `getOrderBook()` | The order book to which this level belongs. NULL if the level does not belong to a book. |
| `getSymbol()` | Return the subscribed symbol for the book. NULL if the level does not belong to a book. |
| `findEntry()` | Find an entry in this level with the specified ID. NULL if a matching entry is not found. |

| Method/Enum | Description |
|---|---|
| `getEntryAtPosition()` | Get the entry at a specified position in the level. NULL if an entry at the specified position is not found. |
| `findEntryAfter()` | Return the entry after the one with the specified ID. NULL if no entry is found. |
| `begin()` | Start iterator over all entries within the price level. In the Java API the `entryIterator()` returns an iterator for the entries. |
| `end()` | End iterator over all entries within the price level. |

The MamdaOrderBookEntry class provides detail on individual orders, or quotes/aggregate participant position, within a price level.

## Table 13: MamdaOrderBookEntry

| Method/Enum | Description |
|---|---|
| enum Action | If part of a delta, this enum indicates:<br>▪ how the delta should be applied to an order book<br>▪ how this delta was applied to the cached order book<br><br>Valid values are:<br>MAMDA_BOOK_ACTION_ADD<br>MAMDA_BOOK_ACTION_UPDATE<br>MAMDA_BOOK_ACTION_DELETE<br>MAMDA_BOOK_ACTION_UNKNOWN |
| `getId()` | Returns the identifier for the entry. If the entry represents an individual order within a book, this will be the order ID. For multi-participant books, this will be the market maker ID or participant ID. For the multi-participant books, the ID will be unique within a level, but may be duplicated within the book, i.e.the participant may have positions at multiple prices for a single instrument. |
| `getUniqueId()` | If supported, returns the order book entry unique ID (order ID, participant ID, etc.). This ID should be unique throughout the order book. If no explicit unique ID has been set, then it assumed that the basic ID is unique - in this instance, the basic ID is returned.<br><br>If set, the unique ID for an entry will be the id+price+side, for example, the ARCA participant on the bid side of the book at a price of 23.45:<br><br>`ARCA23.45B` |
| `getSize()` | The size of the order entry. This will be the number of lots of the security in the order. |
| `getAction()` | If the entry is part of a delta, the action indicates how it should be applied to a book external to the API. If part of the cached order book, the value will be MAMDA_BOOK_ACTION_ADD. |
| `getTime()` | The time the entry was last updated within the book. If part of a delta, this is the event time. |
| `getPrice()` | The price for the entry. |
| `getSide()` | The side of the order book (bid or ask) the entry belongs to. |
| `getPosition()` | The position in the order book for this entry. If maxPos is not zero, then the method will return a result no greater than maxPos. This is to prevent searching the entire book when only a limited search is necessary.<br><br>**Note**: The logic used in the positional search is to use the number of entries that `mamdaOrderBookPriceLevel::getNumEntries()` returns for price levels above the entry's price level. -1 is returned if the entry is in the book but not currently "visible" (i.e., it is being omitted because the ***OpenMAMA*** source is |

| Method/Enum | Description |
|---|---|
| | turned off). A MamdaOrderBookInvalidEntry is thrown if the entry is not found in the book. |
| `equalId()` | Returns "true" if the two entry IDs being compared are equal. |
| `getPriceLevel()` | The price level to which this entry is attached. |
| `getOrderBook()` | The order book to which this entry is attached. |
| `getManager()` | The MamdaOrderBookEntryManager instance to which the entry belongs. |
| `getSymbol()` | The symbol for the entry, if possible. This can only be done if the entry is part of a price level and the price level is part of an order book. NULL is returned if no symbol can be found. |

### *Creating a Caching Order Book Application*

Complete the following steps to create an application that processes cached order books using the ***OpenMAMDA API***. Each step is illustrated in **Example 5: Subclassing the MamdaOrderBookHandler interface** and **Example 6: Creating a caching order book application**.

1. Subclass the MamdaOrderBookHandler interface. The order book handler interface provides callbacks that are invoked in response to order book related events within the API. Applications gain access to the full order book, or delta updates to the book, via these callbacks. Book clear and gap events are also propagated via callbacks to the MamdaOrderBookHandler. Each callback method is detailed in **Table 14: MamdaOrderBookHandler**.

#### Example 5: Subclassing the MamdaOrderBookHandler interface

```
class BookTicker : public MamdaOrderBookHandler 1.
{
    void onBookRecap (
        MamdaSubscription*                 subscription,
        MamdaOrderBookListener&            listener,
        const MamaMsg*                     msg,
        const MamdaOrderBookComplexDelta*  delta,
        const MamdaOrderBookRecap&         recap,
        const MamdaOrderBook&              book) {/*Process recap*/}

    void onBookDelta (
        MamdaSubscription*                 subscription,
        MamdaOrderBookListener&            listener,
        const MamaMsg*                     msg,
        const MamdaOrderBookSimpleDelta&   delta,
        const MamdaOrderBook&              book) {/*Process simple delta*/}

    void onBookComplexDelta (
        MamdaSubscription*                 subscription,
        MamdaOrderBookListener&            listener,
        const MamaMsg*                     msg,
        const MamdaOrderBookComplexDelta&  delta,
        const MamdaOrderBook&              book) {/*Process complex delta*/}

    void onBookClear (
        MamdaSubscription*          subscription,
        MamdaOrderBookListener&     listener,
        const MamaMsg*              msg,
        const MamdaOrderBookClear&  clear,
        const MamdaOrderBook&       book) {/*Process book clear*/}
    void onBookGap (
        MamdaSubscription*          subscription,
        MamdaOrderBookListener&     listener,
        const MamaMsg*              msg,
        const MamdaOrderBookGap&    event,
        const MamdaOrderBook&       book) {/*Process book gap*/}
}
```

2. Call `open()` first to initialize the underlying ***OpenMAMA API***.
3. Obtain the data dictionary via ***OpenMAMA*** using the DictRequester utility class supplied with the example programs.
4. A single instance of a MamdaOrderBookHandler subclass can be used to process data from any number of subscriptions. Invocation of the callbacks on this class by the parent listener instance is the mechanism by which developers gain access to the Order Book (and related information).
5. Initialize the market data specific field descriptor cache. Failing to do so will result in a non-determined failure. The common fields should be initialized for all market data objects.
6. Create a MamdaSubscription to register interest in a book instrument. The default symbology on the ***NYSE Technologies Market Data Platform*** uses a 'b' prefix on all order book format symbols.
7. A new instance of the MamdaOrderBookListener must be used for each subscription. This instance is responsible for caching the order book and maintaining the integrity of the cached book (applying updates, responding to data quality events etc.).
8. The MamdaOrderBookHandler instance is registered with the listener instance. It is via callbacks defined on this interface that you gain access to the updates to the order book.
9. Set the listener to "false" to not process entries within the book (the listener processes entries by default). This can reduce processing overhead if the application is only interested in data aggregated at the price level.
10. Turn on the processing of Market Orders in the MamdaOrderBookListener class. This will maintain both market order bid and ask levels.
11. Once configured, the listener instance is registered with the subscription. At this point the subscription has not been created and no callbacks on the handler will be invoked.
12. The subscription type must be set appropriately to indicate to the feeds that order book data is being subscribed to.
13. The market data type will eventually replace the subscription type as the identifier used by the feeds to determine the nature of the subscription request received.
14. Create the subscription. As with all subscription creation, the actual subscription request will be sent from the throttle queue once dispatching on the internal default event queue has started. The earliest point at which the handler callbacks can be invoked is after the subscription request has been sent and dispatching on the queue associated with the subscription creation has started. In this example, both are on the default ***OpenMAMA*** event queue.
15. Start dispatching on the default ***OpenMAMA*** event queue. At some point after this call the subscription request is sent.

### Example 6: Creating a caching order book application

```
...

Mama::open() 2.

...

DictRequester    dictRequester; 3.

BookTicker       ticker = new BookTicker(); 4.
```

```
...

MamdaOrderCommonFields::setDictionary (dictionary);
MamdaOrderBookFields::setDictionary (dictionary); 5.

for (each symbol being subscribed)
{
    MamdaSubscription*      aSubscription = new MamdaSubscription; 6.
    MamdaOrderBookListener* aBookListener = new MamdaOrderBookListener; 7.

    aBookListener->addHandler (ticker); 8.
    aBookListener->setProcessEntries (true); 9.
    aBookListener->setShowMarketorders (true); 10.


    aSubscription->addMsgListener (aBookListener); 11.

    aSubscription->setType (MAMA_SUBSC_TYPE_BOOK); 12.
    aSubscription->setMdDataType (MAMA_MD_DATA_TYPE_ORDER_BOOK); 13.
    aSubscription->create (queue, "NASDAQ", symbol); 14.
}

Mama::start (bridge); 15.

...
```

### *Processing Data in the MamdaOrderBookHandler Callbacks*

MamdaOrderBookHandler is an interface providing an easy way to receive and process updates to an order book. The interface defines callback methods for different types of order book related events, such as order book recaps, updates, book clears and gaps.

**Table 14: MamdaOrderBookHandler**

| Method/Enum | Description |
|---|---|
| `onBookRecap()` | Invoked when a full refresh of the order book is available. The method may be called for any of the following reasons:<br>▪ Initial Image - The full book received upon subscription creation. This is the initial state of the book at a point in time and before any updates are applied on the client.<br>▪ Start-of-day book state - Received after the feed resets its state, having received a Start of Day message. This will be the state of the book before trading for the day commences.<br>▪ ***OpenMAMA*** recap - An unsolicited recap can be sent from the feeds if a fault tolerant takeover occurs. The ***OpenMAMA API*** can also solicit a recap if a data quality event occurs. |
| `onBookDelta()` | Invoked when a simple delta is received by the API. A simple delta is a delta that effects a single entry in a single level for the book, or a single level if entries are not being processed. Both limit and market orders can be processed through this callback. |
| `onBookComplexDelta()` | Invoked when a complex delta is received by the API. A complex delta is one that effects more than a single entry or level within the order book. It is a list of simple deltas. Both limit and market orders can be processed through this callback. |
| `onBookClear()` | Invoked when a clear order book message is received from the feeds. |
| `onBookGap()` | Invoked if a gap is detected in the symbol level sequence number sent |

| Method/Enum | Description |
|---|---|
| | with all updates. The MamdaOrderBookGap class can be used to see exactly what messages were missed. When a gap occurs, ***OpenMAMA*** automatically requests a recap, and applications should assume the book is stale/suspect until they receive the recap. |

The MamdaOrderBookHandler presents data to API users in terms of the full book, represented as an instance of the MamdaOrderBook class. ***OpenMAMDA*** invokes the callbacks after applying the deltas to the book. In addition to the updated book, the API provides the deltas/updates to the book as either a MamdaOrderBookSimpleDelta or a MamdaOrderBookComplexDelta.

The MamdaOrderBookSimpleDelta represents a delta that updates only a single entry within a single level on one side of the book. The class is a subclass of the MamdaOrderBookBasicDelta interface and simply exposes all functionality available there. The MamdaOrderBookComplexDelta is a list of simple deltas.

In each of the callbacks, applications have access to the full order book and can iterate over all levels and entries within the book.

The following code sample illustrates how to iterate over all levels and entries within an order book callback, and accesses all pertinent information for each level and entry. The Java version of this method would use the `Iterator.next()` and `Iterator.hasNext()` methods to iterate over the price levels.

## Example 7: Processing data in using the onBookRecap callback

```
/*From the MamdaOrderBookHandler interface*/
void onBookRecap (
        MamdaSubscription*                 subscription,
        MamdaOrderBookListener&            listener,
        const MamaMsg*                     msg,
        const MamdaOrderBookComplexDelta*  delta,
        const MamdaOrderBookRecap&         recap,
        const MamdaOrderBook&              book)
    {
        MamdaOrderBook::constBidIterator bidIter = book.bidBegin (); 1.
        MamdaOrderBook::constBidIterator bidEnd  = book.bidEnd ();
        char timeStr[32];
        while (bidIter != bidEnd) 2.
        {
            const MamdaOrderBookPriceLevel* bidLevel = *bidIter;
            bidLevel->getTime().getAsFormattedString (timeStr, 32, "%T%:");
            printf ("  Bid  %4d       %12s %7g %7.2f\n", 3.
                    bidLevel->getNumEntries (),
                    timeStr,
                    bidLevel->getSize (),
                    bidLevel->getPrice ());

            MamdaOrderBookPriceLevel::const_iterator end = bidLevel->end (); 4.
            MamdaOrderBookPriceLevel::const_iterator i   = bidLevel->begin ();
            while (i != end)
            {
                const MamdaOrderBookEntry* entry = *i;

                const char*      id    = entry->getId (); 5.
                mama_quantity_t  size  = entry->getSize ();
                double           price = bidLevel->getPrice ();
                entry->getTime().getAsFormattedString (timeStr, 32, "%T%:");
                printf ("  %14s  %12s %7g %7.2f\n",
                        id, timeStr, size, price);
                ++i;
            }
            ++bidIter;
        }
    }
```

1. Obtain the start and end iterators for the bid side of the book from the full order book passed as an argument to the `onRecap()` callback. **Note:** The full order book is available from each of the handler callbacks.
2. Loop over each of the price levels returned within the iterator.
3. Get the event time, number of entries at that price level, the cumulative size of all entries at that level, the price and log to stdout.
4. Obtain the iterator for the entries within the level.
5. Get the entry id, size, event time and price and log to stdout.

Applications interested only in what has changed within an order book as a result of an update to the book can simply access the order book data via the MamdaOrderBookBasicDelta class, whether accessed from the `onBookSimpleUpdate()` or the `onBookComplexUpdate()` callbacks.

The following code sample illustrates obtaining data from within a complex update. The same code can be used to obtain data in the `onBookSimpleUpdate()` callback, as all delta information is presented in terms of the MamdaOrderBookBasicDelta.

### Example 8: Processing data in using the onBookComplexDelta callback

```
void onBookComplexDelta (
        MamdaSubscription*                 subscription,
        MamdaOrderBookListener&            listener,
        const MamaMsg*                     msg,
        const MamdaOrderBookComplexDelta&  delta,
        const MamdaOrderBook&              book)
{
    MamdaOrderBookComplexDelta::iterator end = delta.end();
    MamdaOrderBookComplexDelta::iterator i   = delta.begin();
    printf ("Complex Delta for side=%d, delta_count=%d.\n",
        delta.getModifiedSides (), delta.getSize ());
    for (; i != end; ++i)
    {
        MamdaOrderBookBasicDelta*  basicDelta = *i;
        printf ("Basic Delta: size=%d, level_action=%c, entry_action=%c\n",
            basicDelta->getPlDeltaSize (), basicDelta->getPlDeltaAction (),
            basicDelta->getEntryDeltaAction ());

        // See onBookRecap () implementation above for printing levels
        // and entries.
        printPriceLevel (basicDelta->getPriceLevel ());
        printEntry (basicDelta->getEntry ());
    }
}
...
```

**Table 15: MamdaOrderBookBasicDelta**

| Method/Enum | Description |
|---|---|
| getPriceLevel() | Get the MamdaOrderBookPriceLevel to which this delta applies. |
| getEntry() | Get the MamdaOrderBookEntry to which this delta applies. Will return NULL if no entry is associated with the delta (applies to feeds that do not supply entry information.) |
| getPlDeltaSize() | The difference in size for the price level. |
| getPlDeltaAction() | The delta action w.r.t the price level, i.e. whether to ADD, UPDATE or DELETE the level from the book. |
| getEntryDeltaAction() | The delta action w.r.t the entry, i.e. whether to ADD, UPDATE or DELETE the entry from the level. |
| getOrderBook() | Get the MamdaOrderBook instance to which this delta applies. |
| The MamdaOrderBookBasicDelta inherits the following methods from MamdaBasicEvent: | |
| getSrcTime() | Get the exchange generated time stamp. |
| getActivityTime() | Feed handler generated time stamp indicating when the last update occurred. |
| getEventSeqNum() | The exchange generated sequence number |
| getEventTime() | The time that the event actually occurred. For many feeds this is the same time as the "source time". |

The MamdaOrderBookComplexDelta represents a delta that updates more than one entry and/or level within a book. This can be within a single level, across multiple levels or even across both sides of the book. The complex delta is presented as a collection of simple delta instances and is a subclass of the MamdaOrderBookBasicDeltaList class.

**Table 16: MamdaOrderBookComplexDelta**

| Method/Enum | Description |
|---|---|
| enum ModifiedSides | Use to specify the side(s) of the book that are modified by this complex delta. Valid values are:<br>MOD_SIDES_NONE<br>MOD_SIDES_BID<br>MOD_SIDES_ASK<br>MOD_SIDES_BID_AND_ASK |
| getModifiedSides() | The side(s) of the book that were modified by this delta. |
| getOrderBook() | The order book that was updated as a result of this delta. |
| getSize() | The number of simple deltas contained within this complex delta. |
| dump() | Dump the complex delta to the specified stream. |
| begin() | Start iterator for the basic deltas within the complex delta. |
| end() | End iterator for the basic deltas within the complex delta. |
| The MamdaOrderBookComplexDelta inherits the following methods from MamdaBasicEvent: | |
| getSrcTime() | Get the exchange generated time stamp. |
| getActivityTime() | Feed handler generated time stamp indicating when the last update occurred. |
| getEventSeqNum() | The exchange generated sequence number |
| getEventTime() | The time that the event actually occurred. For many feeds this is the same time as the "source time". |

### MamdaBookAtomicListener

The MamdaBookAtomicListener class specializes in handling order book updates. Unlike the MamdaOrderBookListener, no actual order book is built or maintained. The sole purpose of this is to provide clients direct access to the order book updates without the overhead of maintaining a book. Developers provide their own implementation of either or both the MamdaBookAtomicLevelHandler and the MamdaBookAtomicLevelEntryHandler interfaces, and will be delivered notifications for order book recaps and deltas. While the MamdaBookAtomicLevelHandler handles recaps and deltas at a Price Level granularity, the MamdaBookAtomicLevelEntryHandler handles recaps and deltas at a Price Level Entry level (both level and entry data). Notifications for order book deltas include only the delta. Unlike, the MamdaOrderBookListener class, the MamdaBookAtomicListener class always processes market orders, and these can be identified based on the order type, obtained from the level.

An obvious application for this ***OpenMAMDA*** class is any kind of program trading application that needs to build its own order book, or an application that needs to archive order book data.

If the only handler added to this listener is an MamdaBookAtomicLevelHandler, then only updates and deltas are processed to Price Level granularity. Entry Level data is ignored, saving on processing time.

***Creating an Atomic Order Book Application***

The following example provides the code snippets required to create an application that processes order book data using the ***OpenMAMDA API***.

## Example 9: Processing order book data in an atomic application

```
class AtomicBookTicker : public MamdaBookAtomicBookHandler,
                         public MamdaBookAtomicLevelHandler,
                         public MamdaBookAtomicLevelEntryHandler
{
  void onBookAtomicLevelRecap (
    MamdaSubscription*                  subscription,
    MamdaBookAtomicListener&         listener,
    const MamaMsg&                   msg,
    const MamdaBookAtomicLevel&      level){/*Process Level Recap*/}

 void onBookAtomicLevelDelta (
    MamdaSubscription*                  subscription,
    MamdaBookAtomicListener&         listener,
    const MamaMsg&                   msg,
    const MamdaBookAtomicLevel&      level){/*Process Level Delta*/}

 void onBookAtomicLevelEntryRecap (
    MamdaSubscription*                  subscription,
    MamdaBookAtomicListener&         listener,
    const MamaMsg&                   msg,
    const MamdaBookAtomicLevelEntry& levelEntry){/*Process Entry Level Recap*/}

 void onBookAtomicLevelEntryDelta (
    MamdaSubscription*                  subscription,
    MamdaBookAtomicListener&         listener,
    const MamaMsg&                   msg,
    const MamdaBookAtomicLevelEntry& levelEntry){/*Process Entry Level Delta*/}

 void onBookAtomicClear (
    MamdaSubscription*                  subscription,
    MamdaBookAtomicListener&         listener,
    const MamaMsg&                   msg){/*Process book clear*/}

 void onBookAtomicGap (
    MamdaSubscription*                  subscription,
    MamdaBookAtomicListener&         listener,
    const MamaMsg&                   msg,
    const MamdaBookAtomicGap&        event){/*Process book gap*/}
}
...

Mama::open()

...
DictRequester       dictRequester;
AtomicBookTicker    ticker = new AtomicBookTicker();
...

MamdaOrderCommonFields::setDictionary (dictionary);
MamdaOrderBookFields::setDictionary (dictionary);

for (each symbol being subscribed)
{
    MamdaSubscription*      aSubscription = new MamdaSubscription;
    MamdaAtomicBookListener* aBookListener = new MamdaAtomicBookListener;

    aBookListener->addBookHandler (aTicker);
    aBookListener->addLevelHandler (aTicker);

    //Entries
    if (addEntryHandler)
    {
        aBookListener->addLevelEntryHandler (aTicker);
    }

    aSubscription->addMsgListener (aBookListener);
    aSubscription->setType (MAMA_SUBSC_TYPE_BOOK);
    aSubscription->setMdDataType (MAMA_MD_DATA_TYPE_ORDER_BOOK);
    aSubscription->create (queues->getNextQueue(), source, symbol);

}

Mama::start (bridge);

...
```

To create an atomic order book application:

- Create an order book application (see *Section 4.1.2.2: Creating a Caching Order Book Application* for details)
- Add the code snippets from **Example 9: Processing order book data** above

Similar to the MamdaOrderBook example, the atomic book handler interface provides callbacks that are invoked in response to order book related events within the API. Applications gain access to the delta updates to the book via these callbacks, and both limit and market order updates are available on these callbacks. Atomic book clear and gap events are also propagated via callbacks to the MamdaAtomicBookHandler.

The table below describes the available Price Level methods on the MamdaBookAtomicLevel interface.

**Table 17: MamdaBookAtomicLevel**

| Method | Description |
|---|---|
| `getPriceLevelNumLevels()` | Number of price levels in the order book update. |
| `getPriceLevelNum()` | The position of this level in the update received. |

| Method | Description |
|---|---|
| `getPriceLevelPrice()` | Price for this price level. |
| `getPriceLevelSize()` | The number of entries in this price level. |
| `getPriceLevelSizeChange()` | The aggregate size at the current price level. |
| `getPriceLevelAction()` | The price level action. |
| `getPriceLevelSide()` | The price level side. |
| `getPriceLevelTime()` | Time of order book price level. |
| `getPriceLevelNumEntries()` | The number of entries at the current price level. |
| `getOrderType()` | The order type for the level (limit or market) |

The table below describes the available Price Level Entry methods with the MamdaBookAtomicLevelEntry interface. The Price Level methods from the above table are also available using this interface.

**Table 18: MamdaBookAtomicLevelEntry**

| Method | Description |
|---|---|
| `getPriceLevelEntryAction()` | The order book entry action. |
| `getPriceLevelEntryId()` | The order book entry ID. |
| `getPriceLevelEntrySize()` | order book entry size. |
| `getPriceLevelEntryTime()` | Time of order book entry update. |

***Using Atomic Order Book with NYSE Technologies V5 Platform***

The ***NYSE Technologies V5 platform*** reduces latency and the CPU footprint by minimizing value add and unnecessary processing. One of the key differences with ***V5*** is that entry based feeds maintain unstructured entry books that have no price level information. For example, price level size change is not monitored from one change to the next.

Normally, this information can be calculated using the MamdaOrderBookListener interface, as it constructs price levels as part of the MamdaOrderBook. However, the MamdaBookAtomicListener interface does not maintain a MamdaOrderBook, and so price level information is no longer available.

## 4.2 Order Book Publishing

Publishing of order book data as ***OpenMAMA*** messages is a feature of the MamdaOrderBook object. It is possible to publish full order book and/or delta messages using MamdaOrderBook along with ***OpenMAMA*** publishing (see *Section 13: Publishing* in the *OpenMAMA Developer's Guide*). Integration with the ***OpenMAMA*** publishing framework will permit advanced publishing with control of data quality, handling of recap requests, new topic requests, refreshes and fault tolerance implementation. Levels and/or entries can be added to the book, and the publishing mechanism will maintain a history of these changes for later publishing. Changes to the price level objects within the book object, for example, adding entries, updating entries, are also recorded by the publishing functionality and published.

**Table 19: MamdaOrderBook - Publishing**

| Method/Enum | Description |
|---|---|
| `generateDeltas()` | Enables order book publishing. |
| `populateRecap (mamaMsg)` | Populates a MamaMsg with the full book information. |
| `populateDelta (MamaMsg)` | Populates a MamaMsg with the changes applied to the MamdaOrderBook from an initial or previous state. Returns "true" when a message is successfully populated, clearing the associated delta list, and "false" when no population occurs, i.e. when no saved deltas exist. |

Code snippets on the use of the publishing functionality are shown below.

```
/*Using the MAMDA Order Book publishing functionality*/

//create new Book and set-up publishing

MamdaOrderBook* aBook = new MamdaOrderBook();

aBook->generateDeltas(true);


//Edit book object by adding levels and/or entries

// Obtain an OpenMAMA message representation of the full book
MamaMsg msg;
aBook->populateRecap(msg);


//Edit book again

//Obtain an OpenMAMA message with all the changes made to the book including the
initial set-up of the book
bool msgPopulated = aBook->populateDelta(msg);

//Edit book again

//Obtain an OpenMAMA message with all the changes made to the book since the
last getDeltaMsg() call
bool msgPopulated = aBook->populateDelta();
```

### 4.2.1 Creating an Order Book Publisher

The code snippets shown above are limited to the ***OpenMAMDA*** publishing functionality. However, ***OpenMAMDA*** publishing is designed to be used in parallel with the ***OpenMAMA*** publishing framework. A full example using both the ***OpenMAMA*** and ***OpenMAMDA*** publishing frameworks is available with your software release as part of the examples programs, bookpublisher.cpp.

The bookpublisher example program integrates advanced ***OpenMAMA*** publishing and ***OpenMAMDA*** order book publishing functionality to enable publishing of MamdaOrderBook data to clients. The example application has two main components:

- the publisher manager/publisher
- the order book publishing functionality

The subscription handler, MamaDQPublisherManagerCallback, is implemented in the `BookPublisher()` class. It is tied to an ***OpenMAMA*** subscription and it handles the following:

- the subscription level requests for the underlying symbol from potential clients
- refresh requests
- errors

The publisher manager, MamaDQPublishManager, creates the subscription used for listening to client requests on a source and acts as a store for all the publishers publishing different symbols on that source. The MamaDQPublisher adds certain fields to the message, such as the message type, before publishing the message. In the example application, a MamdaOrderBook object is created. it is populated with levels and/or entries during runtime, depending on the configuration parameters. Data for the example program is artificially created as an array of book order data, and this drives the example program. Using a MamaTimer object, orders are processed every second during the `MamaTimer::onTimer()` callback, and any changes published using the publishing functionality. Clients that connect to the bookpublisher application, subscribing to a symbol that is being published, will firstly receive a book Inital message followed by the book updates every second. Recap requests from the clients are handled by the bookpublisher, and clients will receive book recap messages when required.

```
/* Integrating OpenMAMA and Order Book publishing */

BookPublisher bookpublisher = new BookPublisher();

//parseCommandLine and set-up dict

//Set-up OpenMAMA publishing functionality
createPublisherTransport();
creatPublisherManager();
creatPublisherAndTimer();

//Set-up book for publishing
createBook();

Mama::start();

//during the bookEditTimer::onTimer() callback, an order is processed,
and if a a message is populated, a delta is published showing all
changes to the book. After the orderArray has been processed,
the book is cleared and we loop over the book order array again
processing the same orders.

onTimer()
{
    bool publish = false;
    if (orderCount ==10)
    {
         //populate clear msg
         publish = true;
    }

    else
    {
        processOrder();
        publish = book.populateDelta();
    }

    if (publish) publishMessage();
}
```

The bookpublisher.cpp example application also implements a locking mechanism that prevents changes to the book when book initials/recaps are published to clients. `MamdaLock()` enables the use of multiple threads for publishing and editing the book, passing in the command line parameter "-threads no. of threads".

The example application also shows how a publisher should handle book initial/recap requests. To maintain data integrity between the publisher and across all clients, upon an initial/recap request, the publisher should firstly publish any changes to the book that may have been executed and stored, before then publishing the initial/recap. This way, all client books will match the publisher book.

## 4.3 Options Chains

The Option Chaining API within ***OpenMAMDA*** provides a suite of classes for managing chains of options contracts. Handler callbacks are invoked in response to state changes, such as the addition and removal of option contracts to and from the chain.

Quote and Trade listeners/handlers can be registered with individual option contracts within a chain to obtain a further level of granular detail.

Advanced features of the Option Chain API include:

- Determine the underlying price.
- Get the set of contracts within a % of the underlying price.
- Get the set of contracts within a fixed size range surrounding the underlying price.
- Determine whether a strike price is within a specific % of the 'at the money' price.
- Obtain a moving window view of strike prices based on the above.

Using the Option Chaining API requires the use of group subscriptions on the OPRA feed handler and regular subscriptions to the feeds for the underlying equities (NASDAQ UTP and CTA (CQS and CTS)).

By default users create a group subscription to the OPRA feeds (the default group symbol is the same as the underlying BBO symbol. E.g. For Microsoft the group option symbol is MSFT). The API also enables users to create an additional regular subscription to the underlying equity and associate this with the option chain listener. Quote and trade listeners for the underlying can also be associated with the MamdaOptionChain to provide most in-depth functionality (E.g. Determining 'in the money' contracts).

Implementing the basic functionality of the ***OpenMAMDA*** Option Chaining API requires creating a MamdaOptionChain and MamdaOptionChainListener object for each underlying symbol that requires an option chain. **Example 10: Creating an option chain** and the following steps illustrate how to implement an option chain.

### Example 10: Creating an option chain

```
class OptionChainHandler : public MamdaOptionChainHandler 1.
{
public:
    void onOptionChainRecap (
        MamdaSubscription*              subscription,
        MamdaOptionChainListener&       listener,
        const MamaMsg&                  msg,
        MamdaOptionChain&               chain)
    {
    }

    void onOptionContractCreate (
        MamdaSubscription*              subscription,
        MamdaOptionChainListener&       listener,
        const MamaMsg&                  msg,
        MamdaOptionContract&            contract,
        MamdaOptionChain&               chain)

    {
    }
    void onOptionSeriesUpdate (
        MamdaSubscription*              subscription,
        MamdaOptionChainListener&       listener,
        const MamaMsg&                  msg,
        const MamdaOptionSeriesUpdate&  event,
        MamdaOptionChain&               chain)
    {
    }

    void onOptionChainGap (
        MamdaSubscription*              subscription,
        MamdaOptionChainListener&       listener,
        const MamaMsg&                  msg,
        MamdaOptionChain&               chain)
    {
    }

...
Mama::open();

MamdaOptionFields::setDictionary (dictionary); 2.

MamdaTradeListener* underlyingTradeListener = new MamdaTradeListener; 3.
MamdaQuoteListener* underlyingQuoteListener = new MamdaQuoteListener;

MamdaOptionChain*   anOptionChain = new MamdaOptionChain ("MSFT"); 4.

anOptionChain->setUnderlyingQuoteListener (underlyingQuoteListener);
anOptionChain->setUnderlyingTradeListener (underlyingTradeListener); 5.

MamdaOptionChainListener*  anOptionListener
    = new MamdaOptionChainListener (anOptionChain);
anOptionListener.addHandler (new OptionChainHandler); 6.

MamdaSubscription*  aBaseSubscription    = new MamdaSubscription; 7.
aBaseSubscription->addMsgListener (underlyingQuoteListener);
aBaseSubscription->addMsgListener (underlyingTradeListener);
anOptionSubscription->create (queue, "NASDAQ", "MSFT");


MamdaSubscription*  anOptionSubscription = new MamdaSubscription; 8.
anOptionSubscription->addMsgListener (anOptionListener);
anOptionSubscription->setType (MAMA_SUBSC_TYPE_GROUP);
anOptionSubscription->create (queue, "OPRA", "MSFT");

}

Mama::start (bridge);
```

1. Subclass the MamdaOptionChainHandler interface. The MamdaOptionChainHandler provides callbacks highlighting activity on the underlying option chain. Each callback method is detailed in **Table 20: MamdaOptionChainHandler**.

**Note** This interface only provides information at the chain level, i.e. additions and removals to and from the chain. Quote and trade handlers can be added to individual contracts within the chain, to provide a further level of detail.

**Table 20: MamdaOptionChainHandler**

| Method/Enum | Description |
|---|---|
| `onOptionChainRecap()` | This callback is invoked when all options for the chain have been received. At this point all initial values for each individual option contract have been received.<br><br>**Note**: Although all option contracts have been received by the API, it is still possible for new contracts to be added to the chain intra-day, useful for contracts that may not start trading until after market open. |
| `onOptionContractCreate()` | Invoked when a new contract is added to the chain. Options may be added to the chain on receipt of an initial value, or from an intra-day update. In either case, this method is invoked. If more detail is required on individual contracts, such as quote and trade information, instances of the required handlers can be registered with the contract object. |
| `onOptionSeriesUpdate()` | Invoked in response to intra-day changes in the chain. Typically invoked when options are added to the chain intra-day, or when options expire and are removed from the chain. |
| `onOptionChainGap()` | Invoked when a gap is detected in the processing of options within the chain. |

2. Initialize the cached dictionary fields for options processing.

**Note** If registering quote and trade handlers with individual option contracts, the appropriate MamdaXXXFields classes must also be initialized.

3. (Optional) Create a trade and quote listener to track activity on the underlying equity.
4. Create the MamdaOptionChain object.
5. Register the listeners for the underlying with the option chain. The information provided from the underlying equity enables the API to determine the 'in the money' contracts, amongst other things.
6. Create the options listener, registering the actual chain object. The handler that receives notifications on changes to the chain is registered with the listener.
7. Create the subscription for the underlying, registering the quote and trade listeners. This subscription is required to obtain data on the underlying instrument.
8. Create a subscription for the option chain. Register the option chain listener with the subscription.

**Note** This is a group subscription (MAMA_SUBSC_TYPE_GROUP). A user subscribes to the group symbol for the chain and receives initials and updates for all option contracts derived from the associated underlying. Typically this subscription symbol is the same as the BBO symbol for the underlying instrument.

### 4.3.1 Accessing the Chain and Contract Information

The MamdaOptionChain class provides users with access to all option contracts within an option chain. The chain is logically represented as lists of put and call options. Users can obtain iterators for both the put and call contract lists and iterate over all contracts within each.

```
MamdaOptionChain::const_iterator callIter = chain.callIterator ();
MamdaOptionChain::const_iterator putIter  = chain.putIterator ();

while (callIter.hasNext())
{
    const MamdaOptionContract* contract = callIter.next ();
}
```

The MamdaOptionContract object provides all information describing a single option contract, such as strike price and expiration date.

```
const char*       symbol       = contract.getSymbol ();
const char*       exchange     = contract.getExchange ();
const char*       expireDate   = contract.getExpireDateStr ();
double            strikePrice  = contract.getStrikePrice ();
long              openInterest = contract.getOpenInterest ();
```

If a quote and trade listener for the underlying instrument has been provided for the option listener, a quote and trade recap can be obtained for the contract.

```
const MamdaTradeRecap& tradeRecap     = contract.getTradeInfo ();
const MamdaQuoteRecap& quoteRecap     = contract.getQuoteInfo ();

const MamaPrice&  lastPrice    = tradeRecap.getLastPrice ();
mama_quantity_t   accVolume    = tradeRecap.getAccVolume ();
const MamaPrice&  bidPrice     = quoteRecap.getBidPrice ();
const MamaPrice&  askPrice     = quoteRecap.getAskPrice ();
```

Users can obtain calculated information from an option contract. This includes:

- At the money price: This can be calculated based on the mid quote (bid+ask/2), ask, bid or last trade price.

```
double theMoneyPrice
= contract.getAtTheMoney (MAMDA_AT_THE_MONEY_COMPARE_LAST_TRADE);
```

- All contracts with a strike price within a percentage of the money. The strikes returned may vary depending on which money compare type is in use to determine the underlying price.

```
StrikeSet strikeSet;
contract.getStrikesWithinPercent (strikeSet,
                                  12.0,
                                  MAMDA_AT_THE_MONEY_COMPARE_LAST_TRADE);
```

- All contracts with a strike price within a range of the money. The strikes returned may vary depending on which money compare type is in use to determine the underlying price.

```
StrikeSet strikeSet;
contract.getStrikesWithinRange (strikeSet,
                                5,
                                MAMDA_AT_THE_MONEY_COMPARE_LAST_TRADE);
```

- Whether a price is within a percentage of the 'money'. The returned value may be different depending on which price is being used for the money calculation.

```
bool isWithinMoney contract.getIsPriceWithinPercentOfMoney (
                                26.70,
                                3.0,
                                MAMDA_AT_THE_MONEY_COMPARE_LAST_TRADE);
```

- Get a list of all expiration dates. Traverse all contracts by expiration date.

The following classes are used throughout the API that provide different groupings of contracts:

- MamdaOptionContract - an individual option contract.
- MamdaOptionContractSet - a set of option contracts at a particular strike price.
- MamdaOptionStrikeSet - both the put and call contract sets for a given strike price. Comprises a MamdaOptionContractSet for the put and the call set of contracts.
- MamdaOptionExpirationStrikes - a set of strike prices at a particular expiration date. Comprises multiple MamdaOptionStrikeSets for each strike price at that expiration date.
- MamdaOptionExpirationDateSet - a set of expiration dates for the chain. Comprises multiple MamdaOptionExpirationStrikes for each expiration date in the chain.

### 4.3.2 Sliding Window Option Chain Views

***OpenMAMDA*** provides the MamdaOptionChainView class, which is used to maintain and provide a sliding window view of an associated option chain.

MamdaOptionChainView is a class that represents a "view" of a subset of an option chain. The view can be restricted to a percentage or number of strike prices around "the money", as well as to a maximum number of days into the future. The view will be adjusted to include strike prices within the range as the underlying price moves. This means that the range of strike prices will change over time. In order to avoid a "jitter" in the range of strike prices when the underlying price hovers right on the edge of a range boundary, the class also provides a "jitter margin" as some percentage of the underlying price (default is 0.5%).

The view class is provided to enable users of the API to easily keep track of only the contracts within a chain they are interested in.

A view instance is created, configured and subsequently registered with the MamdaOptionChainListener instance as a handler.

```
MamdaOptionChainView* anOptionView = new MamdaOptionChainView
(*anOptionChain);
anOptionView->setNumberOfExpirations (2);
anOptionView->setStrikeRangeNumber (4);
```

```
anOptionView->setJitterMargin (1.0);

...

 anOptionListener->addHandler (anOptionView);
```

Use the view in any of the chain related (or quote/trade handler) callbacks. The MamdaOptionExpirationDateSet can be obtained from the view. Drilling down into this structure provides access to all contracts available in the view.

## 4.4 News

***OpenMAMDA*** News provides a generic API for handling financial news. The API consists of three functional components, namely, MamdaNewsManager, MamdaNewsHeadlineHandler, MamdaNewsStoryHandler, and a few data-centric components, MamdaNewsHeadline, MamdaNewsStory.

The MamdaNewsManager, as the main stem of ***OpenMAMDA*** News API, maintains the underlying subscriptions and all headline and story handlers.

**Example 11: MamdaNewsManager**

```
void onNewsHeadline (
      MamdaNewsManager&          manager,
      const MamaMsg&             msg,
      const MamdaNewsHeadline&   headline,
      void*                      closure)

void onNewsStory (
      MamdaNewsManager&          manager,
      const MamaMsg&             msg,
      const MamdaNewsStory&      story,
      void*                      closure)
```

MamdaNewsHeadline and MamdaNewsStory store the actual headline and story body respectively, as well as other associated attributes.

### 4.4.1 Headline Attributes

- The text of the headline for the story.
- The data source ID of the news story.
- The original data source ID of the news story (e.g., if the story was provided by a news aggregator).
- The ANSI language ID of the news story.
- An array of native meta-data codes associated with this news story.
- An array of native feed symbol codes associated with this news story.
- An array of normalized industry codes associated with this news story.
- An array of normalized market sector codes associated with this news story.
- An array of normalized region codes associated with this news story.
- An array of ISO country codes associated with this news story.
- An array of normalized topic (or "subject") codes associated with this news story.
- An array of normalized product codes associated with this news story.
- An array of normalized miscellaneous codes associated with this news story.
- Miscellaneous codes are those not categorized as industry, market sector, region, country or product codes.
- An array of normalized symbol codes associated with this news story.
- Whether the feed provider has designated this story as with normal priority or "hot" (important)

priority.

- The revision number. Returns zero if the data source does not provide revision numbers.
- The original publish time of the news story.

### 4.4.2 Story Attributes

- The text of the story.
- The unique story ID, for the data source, for this news story.
- The revision number. Returns zero if the data source does not provide revision numbers.
- The story status (see **Table 21: Story Status** for possible values).
- The latest story update time (e.g. Time of correction).
- The original story publish time.
- All of the headline IDs associated with this news story.

**Table 21: Story Status**

| Status | Description |
|---|---|
| NO_STORY | There is currently no story for the headline. This may occur for feeds that provide "alert" headlines, either as the only headline or as a precursor to a full story. |
| FULL_STORY | The complete story text is being provided in the current callback. |
| FETCHING_STORY | The story is currently being fetched by the publisher. This is a temporary status. An additional callback will automatically be invoked when the full story is available. |
| DELAYED_STORY | The story is not currently available but is expected at some time in the future. No additional callback will be automatically invoked. |
| NOT_FOUND | The publisher does not currently have a story for this headline and cannot determine whether a story will arrive for the headline. No additional callback will be automatically invoked. |
| UNKNOWN | Indicates an unknown condition (should not happen). |

The following procedure and code example illustrate the initialization of the News API in ***OpenMAMDA***:

1. Create the handler class for processing both headlines and stories.
2. Set the news dictionary fields.
3. Register the handler with the manager.
4. Add headline sources.
5. Add the source for stories.

All headlines and stories will be passed to registered handlers.

```
class NewsTicker : public MamdaNewsHeadlineHandler 1.
               , public MamdaNewsStoryHandler
{
  void onNewsHeadline (
      MamdaNewsManager&          manager,
      const MamaMsg&             msg,
      const MamdaNewsHeadline&   headline,
      void*                      closure)
  {
   //process the news headline
  }

  void onNewsStory (
      MamdaNewsManager&          manager,
      const MamaMsg&            msg,
      const MamdaNewsStory&     story,
      void*                     closure)
  {
   // process the news story
  }
};

....

MamdaNewsManager*   aNewsManager   = new MamdaNewsManager;
NewsTicker*         aTicker        = new NewsTicker;

MamdaCommonFields::setDictionary (dictionary); 2.
MamdaNewsFields::setDictionary (dictionary);

aNewsManager->addBroadcastHeadlineHandler (aTicker); 3.
aNewsManager->addBroadcastStoryHandler (aTicker);

for (vector<const char*>::const_iterator i
   = symbolList.begin();i != symbolList.end();++i) 4.
{
    const char* symbol = *i;
    aNewsManager->addBroadcastHeadlineSource (
        queues->getNextQueue(), source, symbol, NULL);
}

aNewsManager->addBroadcastStorySource (
   queues->getNextQueue(), source, "STORIES", NULL
 ); 5.
```

# 5 Non Market Data Utility Classes

***OpenMAMDA*** also provides utility classes that, in isolation, are not market data aware. These classes are intended to be used in conjunction with the market data classes described in previous sections to expand on the functionality of the API.

Currently, the API provides the MamdaMultiParticipantManager and the MamdaMultiSecurityManager, both of which promote the ease of use of the API with group subscriptions.

## 5.1 MamdaMultiParticipantManager

The MamdaMultiParticipantManager provides a mechanism by which ***OpenMAMDA*** can be used to more easily process participant group subscriptions. These are group subscriptions for an instrument that provides updates from all participants currently quoting that instrument. Participant group subscriptions are available on NASDAQ Level 1 UTP and other feeds where instruments are multiply listed.

Using the participant group subscription facility means you do not need to know the individual symbology for each participant quoting the security, or what participants are currently providing quotes. The group subscription and the MamdaMultiParticipantManager will also provide notification to users of the API of new participants as they start quoting intra-day.

Users of the API create a subscription for a participant group symbol, for example, MSFT.GRP on NASDAQ UTP, and register a multi-participant manager instance with the subscription. The manager identifies the individual participants within the group, including the BBO or consolidated record if present, and invokes callbacks on the MamdaMultiParticipantHandler indicating when new participants (or consolidated) have been added to the manager, usually when initial values for each participant arrive.

At this point users of the API can register any of the other ***OpenMAMDA*** listener classes with the individual participants to obtain more fine grained information on all or a subset of participants within the group.

**Note** If registering interest in both participants and the BBO (consolidated), trades and quotes may be reported twice.

The following code snippets illustrate the use of the MamdaMultiParticipantManager and registers a MamdaQuoteListener for each participant and for the BBO.

```
...

class MultiParticipantExample : public MamdaMultiParticipantHandler 1.
{
public:
    void onConsolidatedCreate (
        MamdaSubscription*              subscription,
        MamdaMultiParticipantManager&   manager)
    {
        // Create a quote listener for the consolidated member. 2.
        MamdaQuoteListener* aQuoteListener = new MamdaQuoteListener;
        QuoteTicker*        aTicker = new QuoteTicker (*aQuoteListener);
        aQuoteListener->addHandler (aTicker);

        manager.addConsolidatedListener (aQuoteListener);
    }

    void onParticipantCreate (
        MamdaSubscription*             subscription,
        MamdaMultiParticipantManager&  manager,
        const char*                    partId,
        bool                           isPrimary)
    {
        // Create a quote listener for the regional member. 3.
        MamdaQuoteListener* aQuoteListener = new MamdaQuoteListener;

        QuoteTicker*        aTicker = new QuoteTicker (*aQuoteListener);

        aQuoteListener->addHandler (aTicker);

        manager.addParticipantListener (aQuoteListener, partId);
    }
};
...

Mama::open()

// Obtain the data dictionary from the platform

MamdaCommonFields::setDictionary (dictionary);
MamdaQuoteFields::setDictionary (dictionary); 4.

MamdaSubscription* aSubscription = new MamdaSubscription ();

MultiParticipantExample*  multipartHandler =
                new MultiParticipantExample;
MamdaMultiParticipantManager*  multipartManager =
                new MamdaMultiParticipantManager ("MSFT.GRP");

multipartManager->addHandler (multipartHandler);
aSubscription->addMsgListener   (multipartManager); 5.

//Create a MamaSource for the data feed

aSubscription->setType (MAMA_SUBSC_TYPE_GROUP);
aSubscription->create (queue, source, "MSFT.GRP");

...

Mama::start (bridge); 6.
```

1. Subclass the MamdaMultiParticipantHandler interface. Callbacks on a subclass of the MamdaMultiParticipantHandler interface provides an indication when new participants or the consolidated is added to the manager.
2. (Optional) Register a quote listener for the symbol when informed of the consolidated record being added to the manager.

**Note** This listener can be registered prior to creating the underlying subscription. In that case is not necessary to respond to the `onConsolidatedCreate()` callback.

3. (Optional) Register a quote listener for the symbol when informed of a new participant record being added to the manager. Users can register interest in a subset of participants if required. All others will be silently ignored by the manager.

**Note** This listener can be registered prior to creating the underlying subscription. In that case is not necessary to respond to the `onParticipantCreate()` callback.

4. Initialize the cached dictionary fields for quotes processing, required as the example creates a quote listener for each participant and the consolidated population of the MamdaQuoteFields cache.
5. Add the MamdaMultiParticipantManager as a listener to the MamdaSubscription, in the same fashion as all other listener instances.

**Note** The symbol used has a .GRP suffix. This is a special type of group subscription and requires a subscription type of MAMA_SUBSC_TYPE_GROUP.

6. Start dispatching on the default event queue.

## 5.2 MamdaMultiSecurityManager

The MamdaMultiSecurityManager provides a mechanism by which ***OpenMAMDA*** can be used in conjunction with the GROUP_ALL subscription facility of the ***NYSE Technologies Market Data Platform***. This is a special case group subscription where all instruments from a feed are available on a single group subscription.

**Note** The GROUP_ALL facility should be used with caution. It is recommended only for low volume feeds, and where there is lightweight processing on the client. When using this feature, all processing must be carried out on a single thread within the client application. Horizontally scaling the processing across threads/queues is not possible, as all updates arrive from a single subscription within ***OpenMAMA***/***OpenMAMDA***.

```
...

class MultiSecurityExample : public MamdaMultiSecurityHandler 1.
{
```

```
public:
    void onSecurityCreate (
        MamdaSubscription*             subscription,
        MamdaMultiSecurityManager&     manager,
        const char*                    securitySymbol,
        bool                           isPrimary)
    {
........// Create a trade listener. 2.
        MamdaTradeListener* aTradeListener = new MamdaTradeListener;

        TradeTicker* aTradeTicker = new TradeTicker (*aTradeListener);

        aQuoteListener->addHandler (aTradeTicker);

        manager.addSecurityListener (aTradeListener, securitySymbol);
    }
};
...

Mama::open()

// Obtain the data dictionary from the platform

MamdaCommonFields::setDictionary (dictionary);
MamdaTradeFields::setDictionary (dictionary); 3.

MamdaSubscription* aSubscription = new MamdaSubscription ();

MultiSecurityExample*  multisecurityHandler =
                new MultiSecurityExample;
MamdaMultiSecurityManager*  multisecurityManager =
                new MamdaMultiSecurityManager ("NASDAQ_ALL");

multisecurityManager->addHandler (multisecurityHandler);
aSubscription->addMsgListener   (multisecurityManager); 4.

//Create a MamaSource for the NASDAQ UTP data feed
//Create MamaQueue or obtain default queue from MamaBridge

aSubscription->setType (MAMA_SUBSC_TYPE_GROUP);
aSubscription->create (queue, source, "NASDAQ_ALL");

...

Mama::start (bridge); 5.
```

1. Subclass the MamdaMultiSecurityHandler interface. Callbacks on a subclass of the MamdaMultiSecurityHandler interface provides an indication when new members (securities) are added to the multi security manager.

**Note** The isPrimary parameter to the `onSecurityCreate()` callback method is not currently used and is reserved for future use.

2. (Optional) Register a trade listener for the symbol when informed of a new instrument/security being added to the manager.

> **Note** This listener can be registered prior to creating the underlying subscription. In that case is not necessary to respond to the `onSecurityCreate()` callback.

3. Initialize the cached dictionary fields for trade processing, required as the example creates a trade listener for each security in the group.
4. Add the MamdaMultiSecurityManager as a listener to the MamdaSubscription, in the same fashion as all other listener instances.

> **Note** The symbol used is NASDAQ_ALL. This is a special type of group subscription and requires a subscription type of MAMA_SUBSC_TYPE_GROUP. This particular symbol is the NASDAQ UTP group symbol for all securities for that feed handler.

5. Start dispatching on the default event queue.