OpenMAMA Developer's Guide for C++

23 November 2012

Document Conventions

Information Type	Example
Feed name	*OpenMAMA Source*

Configuration file content or source code	`<Parameter>`
	`  <Name>PublishFullOrderBook</Name>`
	`  <Value>false</Value>`
	`</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> 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
13 Oct 2011	1.0	Initial release.
30 Apr 2012	1.1	Updated to include support for: ▪ Windows operating system ▪ 29West LBM middleware ▪ C++ and JNI/Java languages
30 Sep 2012	1.2	Updated to include support for C#.
23 Nov 2012	1.3	Updated the following: ▪ **Table 2: Middlewares and Identifiers** ▪ *Section 5.2: Setting Transport Properties*

Table of Contents

1 Introduction and Architecture

The ***MAMA (Middleware Agnostic Messaging) API*** is a subscription based messaging API with publish/subscribe semantics, which provides a lightweight abstraction on top of a variety of underlying messaging middlewares. The ***OpenMAMA API*** provides developers with a common interface to the underlying messaging API, allowing migration from one messaging API to another without any code changes to applications built using the API.

The API provides an asynchronous, event-driven programming model. You use the API to provide callbacks where required. Data is propagated back to the registered application via these callbacks in response to dispatching of events from event queues.

Market data semantics are added through the use of the ***OpenMAMDA API***. The ***OpenMAMA API*** also provides extra features when used on the ***NYSE Technologies Market Data Infrastructure***, such as:

- Subscription throttling
- Entitlement enforcement
- Initial values/recaps
- Data quality
- Refresh messages
- Group subscriptions

The document details the major objects within the API and their most common usage.

1.1 Operating Systems

The following operating systems are currently supported:

- Linux
- Windows

1.2 Middlewares

The following middlewares are currently supported:

- Avis
- NYSE Technologies Data Fabric (as a separate plug in for Data Fabric customers only)
- 29West LBM (as a separate plug in for NYSE Technologies customers)

1.3 API Language Implementations

The various language implementations for the ***OpenMAMA API*** expose the same top level objects with broadly the same programming interface. Unless otherwise specified, allocation of ***OpenMAMA*** objects and creation of ***OpenMAMA*** objects are two separate steps. The first step allocates the memory and the second initializes the object. Each of the API implementations is thread safe and thread aware. All functions/methods in the API, across all language implementations and transports, exhibit the same behavior, unless otherwise stated.

The following language implementations are currently supported:

- C
- C++
- C#
- JNI

The C API naming convention is used to identify objects and the functions that operate on those objects. Each object has its own header file containing definitions for all operations supported by that object. The naming convention is as follows:

- Objects: `mama<Type>`
- Functions: `mama<Type>_operation`

All structures are defined as opaque types. As such they cannot be allocated directly by users of the API. All memory management for a ***OpenMAMA*** object in C is controlled via calls to the appropriate functions, for example:

- `mama<Type>_allocate()`
- `mama<Type>_destroy()`
- `mama<Type>_deallocate()`

Sample applications written using the API are located in the `examples` directory provided with the distribution. Pre-compiled versions of these example programs are located in the `bin` directory. Example programs are detailed in *Chapter 15: Example Programs*. For clarity, the sample snippets of code do not check the return values from function/method calls. In a production application it is recommended that all return codes are checked for success before proceeding.

Use of the API is not restricted to the ***NYSE Technologies Market Data Platform***. We provide the concept of 'Basic' subscribers and publishers which allow users of the API to publish and subscribe to topics outside of the Market Data Platform (details provided in relevant sections).

All callback functions/methods provide access to closure data in order to allow applications to associate contextual information in their own applications with callbacks invoked from within the ***OpenMAMA API***.

1.4 Using the API

The following is a high level overview of the steps required to implement the most common use of the ***OpenMAMA API***: to write a market data subscribing application. Each step and their associated features are detailed in the following chapters.

1. Load the bridge(s).

   At least one bridge object must be created before ***OpenMAMA*** is opened. Bridges can only be loaded at this time.

2. Initialize the API.

   Before creating any ***OpenMAMA*** objects it is necessary to initialize the API. Open ***OpenMAMA*** by calling the `Mama::open()` function. This call sets up a number of internal processes within the API that are required for successful use of the API.

   Note The call to `Mama::open()` is reference counted and each call must have a corresponding call to `Mama::close()`.

3. Create event queue(s) (optional).

   Users of the API can use the default internal event queue for a bridge. If multi-threading is required, or more control over the dispatching of events, separate event queues can be created using `MamaQueue()`. See *Section 6: Events and Queues* for details.

   Note It is not necessary for an application to create its own event queues.

4. Create transport(s).

   An application requires that at least one transport object has been created. The `MamaTransport` object is used to define the properties for the underlying middleware's communication protocol. There must be an entry in the `mama.properties` file for each transport created. See *Section 5: Transports* for details.

5. Fetch data dictionary.

   A data dictionary is typically required in order to obtain complete information (name, fid and data type) regarding fields within messages. See *Section 10: OpenMAMA Dictionary* for details.

6. Create subscriptions.

   Create a subscription object, `MamaSubscription`, for each symbol known to the application at startup. See *Section 7: Subscriptions* for details.

7. Start dispatching.

   Once all subscriptions have been created, start dispatching on the default event queue for a bridge,

and any other queues that have been created. You can continue to create further intra-day subscriptions.

8. Shut down.

   Objects must be destroyed in the following order when shutting down the application:

   1. Stop dispatching on any event queues (also call `Mama::stop()`)
   2. Destroy event object (Subscriptions, timers, io objects)
   3. Destroy all event queues
   4. Destroy all transports
   5. Call `Mama::close()`

   Event objects can be created and destroyed at runtime, however, queues must not be destroyed before all event objects that use those queues are destroyed. It is recommended that ***OpenMAMA*** objects are destroyed in this order, as it will result in the same behavior on all middlewares currently supported.

> **Note** This applies only to shutdown. Event objects can be destroyed at any point during the life of an application.

1.5 Object Summary

Table 1: OpenMAMA Objects lists the major objects/types that are available for use within an ***OpenMAMA*** based application.

Table 1: OpenMAMA Objects

MAMA Object	Description
Bridge	Used by ***OpenMAMA*** to communicate with a middleware.
Transport	Communication protocol properties
Subscription	Register interest in a symbol (topic) and receive callback updates on that symbol (topic). Receive point-to-point requests.
Message	Access to the data delivered through program listing subscription callback events. Used to create structured data for sending when publishing via the API.
Queue	Representation of the underlying event queue for dispatching events (data, timer, io etc).
Timer	Recurring timer implementation. Receive a callback at a recurring interval.
Io	Register interest in events associated with file descriptors.
Publisher	Publish data to a specific symbol (topic) onto the messaging backbone. Send point-to-point requests.
Inbox	Receive responses to point-to-point requests.
Dictionary	Access to the definition of fields (name, fid and type) being used on the Market Data Platform.
Source	Details of how to obtain data when creating a subscription.

2 Installation

2.1 Linux

2.1.1 Installation

OpenMAMA for Linux is distributed as a gzipped tar file. File name format:

```
wombat_products_<mw>_<os>_<compiler>_<glibc>_ent.tgz
```

where:

`<mw>`	- indicates the middleware this version of ***OpenMAMA*** is for
`<os>`	- indicates the operating system
`<complier>`	- indicates the compiler
`<glibc>`	- the glibc version

To install ***OpenMAMA*** on Linux, complete the following steps:

1. Unzip the file to a convenient directory. For illustration, we use the directory `/var/userspace/mama`. The following directories are created:

Directory	Description
`bin`	Executable files.
`doc`	***OpenMAMA*** documentation.
`examples`	Source code for the example programs, and an example `mama.properties` file .
`include`	Header files.
`lib`	Library files.
`RELEASE_NOTES`	Release notes.

2. Create a `config` directory:

   ```
   $ mkdir /var/userspace/mama/config
   ```

3. Copy `mama.properties` from `examples/mama` to the `config` directory. This file should have transport settings.

4. Set the `WOMBAT_PATH` variable to include `config`:

   ```
   $ export WOMBAT_PATH=/var/userspace/mama/config
   ```

5. Set the `LD_LIBRARY_PATH` variable to include `lib`:

   ```
   $ export LD_LIBRARY_PATH=/var/userspace/mama/lib:$LD_LIBRARY_PATH
   ```

2.1.2 Running Example Programs

When you have extracted the API and set the environment variables as described above, you can run the example programs. For the example programs to receive any data there must be a publisher somewhere on the network.

To run the **mamalistenc** program, change to the `bin` directory and run the following command:

```
$ ./mamalistenc -S SOURCENAME -tport transportname -s SYMBOL
```

where:

SOURCENAME	- the source name
transportname	- the transport name
SYMBOL	- the symbol you wish to subscribe to

The transport should be defined in both `mama.properties` and the publisher configuration. The source should be defined in the publisher configuration.

The other example programs use similar options to **mamalistenc**. For a full list of options for a particular program, use the **-?** option. For example, to see all the options for bookviewer, run the following command:

```
$ mamalistenc -?
```

2.1.3 Compiling Example Programs

The C++ examples are found in `examples/mama`. They can be compiled using `make` and `Makefile.sample`. The variable `API_HOME` has to be set to the directory where the API was installed. For example, to build **mamalistenc**, run the following command:

```
$ make -f Makefile.sample API_HOME=/var/userspace/mama mamalistenc
```

where:

`API_HOME`	- the full path to the directory where the API was installed

To build all the example programs, run `make` without a target:

```
$ make -f Makefile.sample API_HOME=/var/userspace/mama
```

Possible values for the target are `mamalistenc`, `mamalistencpp`, `mamapublisherc`, `mamapublishercpp`, `mamasubscriberc`, `mamasubscribercpp`, `mamainboxc`, `mamainboxcpp`, `mamasymbollistsubscriberc`, and `mamasymbollistsubscribercpp`.

2.2 Windows

2.2.1 Installation Steps

OpenMAMA for Windows is distributed as a zip file. The name of the zip file has the following format:

```
<d>_<t>_wombat_products_<mw>_win32_<VS>_with_entitle_<branch>.zip
```

where:

`<d>`	- a date stamp
`<t>`	- a time stamp
`<mw>`	- indicates the middleware this version of ***OpenMAMA*** is for
`<VS>`	- the version of the Visual Studio compiler used
`<branch>`	- the branch number

To install ***OpenMAMA*** on Windows complete the following steps:

1. Unzip the file to a convenient directory. For illustration we use the directory `C:\mama`. The following directories are created:

Directory	Description
`bin`	Executable files.
`doc`	***OpenMAMA*** documentation.
`examples`	Source code for the example programs.
`include`	Header files.
`lib`	Library files for use with .dll files.
`pdb`	Debug files.
`RELEASE_NOTES`	Release notes.

2. Create a `config` directory:

```
$ mkdir c:\mama\config
```

3. Create a `mama.properties` file in the `config` directory. This file should have transport settings.
4. Set the `WOMBAT_PATH` variable to include `config`:

```
$ set WOMBAT_PATH=c:\mama\config;%WOMBAT_PATH%
```

5. Set `API_HOME` to the directory where you extracted MAMA:

```
$ set API_HOME=c:\mama
```

6. Set the `PATH` variable to include `bin\dynamic` and `bin\dynamic\debug`:

```
$ set PATH=%PATH%;c:\mama\bin\dynamic;c:\mama\bin\dynamic-debug
```

2.2.2 Compiler Discrepancies: Building with Visual Studio 2010

Static Example Programs

In order to correctly build and run static example programs, the environment variable `PLATFORM_LIB` must be defined to include the appropriate middleware libraries. This is a list of libraries separated by spaces:

```
$ set PLATFORM_LIB_DEBUG=libmamawmwimplmtd.lib libwombatmwmtd.lib
$ set PLATFORM_LIB=libmamawmwimplmt.lib libwombatmwmt.lib
```

Note When using a debug build this should be `PLATFORM_LIB_DEBUG`.

This list must also include the OEA entitlement, Wirecache and FAST libraries, if appropriate.

Note When the static example programs are built inside the Visual Studio 2010 environment, the library names must be separated by semi-colons.

```
$ set PLATFORM_LIB_DEBUG=libmamawmwimplmtd.lib;libwombatmwmtd.lib
$ set PLATFORM_LIB=libmamawmwimplmt.lib;libwombatmwmt.lib
```

This does not apply when building using the `Makefile.example.vcc`.

3 Bridges

3.1 Middleware Bridges

OpenMAMA supports the different middlewares through the use of bridge objects. Multiple bridges can be loaded at any one time, one for each middleware, meaning that a single ***OpenMAMA*** application can support more than one middleware concurrently.

Bridge objects must be created at startup, before `mama_open()` is called. How you create the bridge depends on how the bridge libraries are linked into the ***OpenMAMA*** application, see *Section 3.1.1: Using Linked Bridge Libraries* and *Section 3.1.2: Loading Bridge Libraries at Runtime* for details. Once initialized, bridge objects are passed as parameters when creating transport, queue and queue group objects. Any further objects (such as subscriptions or timers) that use a transport or queue automatically use the same middleware and bridge. `mama_start()` and `mama_stop()` also take a bridge as a parameter to start dispatching on the default event queue for that bridge.

The following table lists the identifier strings to use to represent the different middleware bridges, and which vendor supports them.

Table 2: Middlewares and Identifiers

Middleware	Identifier	Supported By
Avis	avis	Open source
29West LBM	lbm	Informatica
NYSE Technologies Data Fabric	wmw	NYSE Technologies
Tick 42 BLP	tick42blp	Tick 42
RAI	rai	Rai Technologies
QPID	qpid	Open source
Exegy	exegy	Exegy Inc.

The bridge implementation libraries shipped with ***OpenMAMA*** contain all the middleware specific functions. These bridge libraries can either be linked into the application at link time, or dynamically loaded at runtime (all languages), as detailed in the following sections.

3.1.1 Using Linked Bridge Libraries

If the bridge implementation libraries are linked into the ***OpenMAMA*** application, either statically or dynamically, then a bridge can be loaded using one of the following methods:

- Using one of the middleware specific load functions.
- Through a macro that takes the middleware identifier as a parameter.

An application will fail to compile if it tries to use a middleware bridge that has not been linked in.

3.1.2 Loading Bridge Libraries at Runtime

Bridge libraries can be dynamically loaded at runtime. This is different from dynamic linking as the bridge libraries are not actually linked into the application. The bridge to be used is decided at runtime. Using dynamic loading offers the greatest flexibility as it means that applications do not have to be recompiled to use different middlewares. This is the method that the example applications use.

If the application tries to load a bridge library that is not available, or if the middleware libraries for that bridge are not available, then the load will fail. A path to the bridge libraries must be available from

LD_LIBRARY_PATH, for Unix systems, or PATH, for Windows systems.

Example 1: Loading bridge libraries at runtime

```
mamaBridge bridge = NULL;
bridge = Mama::loadBridge ("avis");
```

3.1.3 Loading Bridge Libraries at Runtime from a Specified Location

OpenMAMA also allows the user to load the bridge libraries from a specified path. If no path is cited, loadBridge functionality defaults to that of *Section 3.2.1: Loading Bridge Libraries at Runtime*. The specified path must use the appropriate path separator for the OS i.e. "/" for Unix systems, "\" for Windows systems.

Example 2: Loading bridge libraries at runtime with path

```
mamaBridge bridge = NULL;
bridge = Mama::loadBridge ("avis", "/home/usr/wombat/mama/lib");
```

3.1.4 Creating Bridges

A utility macro is provided with C++ that creates a bridge object using a linked bridge library if available, or dynamically loads a bridge library if it has not been linked in. This means that the application can use the same call independent of the method it uses to load the bridges.

3.2 Payload Bridges

OpenMAMA supports different payloads through the use of bridge objects. Multiple bridges can be loaded at any one time, one for each payload, meaning that a single ***OpenMAMA*** application can support more than one payload concurrently.

Payload bridge objects follow the same rules for loading as middleware bridges (see *Section 3.1: Middleware Bridges*). **Table 3: Payloads and Identifiers** lists the identifier strings to use to represent the different middleware bridges.

Table 3: Payloads and Identifiers

Middleware	Identifier	Shared object name
Avis	A	libmamaavismsgimpl
NYSE Technologies proprietary format	W	libmamawmsgimpl

3.2.1 Loading Payload Bridges at Runtime

Payload bridge libraries can be dynamically loaded at runtime in the same way as a middleware bridge.

Example 3: Loading payload bridge libraries at runtime

```
mamaPayloadBridge payload = NULL;
payload = Mama::loadPayloadBridge ("avis");
```

3.2.2 Default Payloads

A middleware bridge may specify which payload to use, which is identified during the middleware bridge loading process. When specified, ***OpenMAMA*** tries to load the requested payload bridge. If the payload is not available the middleware bridge continues to load, and the payload load failure is logged.

The first payload bridge that is successfully loaded is marked as the default payload. This can be overridden programmatically.

Example 4: Set default payload

```
mama_setDefaultPayloadBridge ('A');
```

3.2.3 Using Payload

The default payload is used when no payload bridge is explicitly stated.

There are three options available for specifying payload creation:

- Using the default (see **Example 5: Implicit msg payload**)
- Using the payload ID (see **Example 6: Explicit msg payload using ID**)
- Using the bridge structure (see **Example 7: Explicit msg payload bridge**)

The following examples show the payload creation options

Example 5: Implicit msg payload

```
mamaMsg*   msg = NULL;
mamaMag_create(&msg);
```

Example 6: Explicit msg payload using ID

```
mamaMsg*   msg = NULL;
mamaMag_createForPayload(&msg, 'A');
```

Example 7: Explicit msg payload bridge

```
mamaMsg*   msg = NULL;
mamaMag_createForPayloadBridge(&msg, payload);
```

4 Properties

There are a number of mechanisms in place by which properties for the ***OpenMAMA API***, and the underlying messaging middleware specific APIs, can be specified.

The default behavior for the API (for all languages) is to look for a file named `mama.properties` in the directory specified by the `WOMBAT_PATH` environment variable. If present, the specified properties file is loaded on application startup when `Mama::open()` is called. ***OpenMAMA*** accesses the properties file and looks for all the possible properties (see *Section 18: Configuration Reference* for details). Misspellings and omissions are not highlighted on startup. Please look at the documentation for the messaging middleware being used for more detail on, and an explanation of, transport level properties.

Alternatively, users of the API can override this behavior and can specify a file name and location using `Mama.openWithProperties()`. The fully qualified path to the directory is required.

4.1 Setting Properties at Runtime

Properties can also be specified at runtime. This approach can be used to override existing properties specified in `mama.properties`, or to add new properties to the API. All properties should be specified prior to creating any transport objects.

Note Calling `mama_open` or `mama_openWithProperties` overrides any properties set prior to these calls.

Properties are typically interpreted when objects such as transports or subscriptions are created, therefore changing these at runtime has little or no effect.

The following example illustrates the setting of the router location for the Avismessaging middleware. Properties are always specified as strings in all API versions.

Example 8: Setting properties at runtime

```
mama_setProperty ("mama.avis.transport.avis_tport.url",
                  "elvin://127.0.0.1:5555");
```

Regardless of the mechanism used to specify the properties, or the specific language implementation used, the properties must always have the same format. An example of a `mama.properties` file is provided in the `examples` directory as part of the release structure, which highlights and describes the most commonly specified properties for each supported middleware.

5 Transports

The underpinning object in any ***OpenMAMA*** application is the MamaTransport object. The MamaTransport object defines the network protocol-level parameters over which ***OpenMAMA*** distributes data. Transports effectively provide scope for data identifying the underlying protocols and their values for data delivery. This object is an isolated communication channel for data traffic. Transports specify the communication channels to use for making subscriptions and publishing data through the API. The transport properties that need to be set for each middleware are as follows:

- 29West LBM: topic resolution and immediate messaging properties.
- ***NYSE Technologies Data Fabric***: machine, port, and other ***Data Fabric*** properties.
- Avis: location of the router

A single ***OpenMAMA*** application can define multiple transport objects if more than one set of physical transport channels exist on the network.

When a transport object is created it is given a name identifier, which is used internally to obtain any configuration parameters that have been specified on application startup. These properties can be specified in a variety of ways, the most common being in `mama.properties` (see *Section 4: Properties*). If the name is not found or null, then the defaults specified by ***OpenMAMA*** for that middleware are used. The defaults can be overridden by creating transport entries in the properties file with the name default.

5.1 Creating a Transport

The following example code illustrates how to create a transport object using the identifier "avis_tport" to locate relevant properties from the list specified to the API upon initialization.

Example 9: Creating a transport

```
MamaTransport* transport = new MamaTransport();

//set runtime properties
transport->create ("avis_tport", bridge);
```

5.2 Setting Transport Properties

Properties are typically set on a per transport basis within the API. Transport properties always follow this naming convention:

```
mama.<middleware>.transport.<transport name>.<property name>=<value>
```

where:

<middleware>	-	for details on the supported middlewares and their identifiers, see *Section 3.1: Middleware Bridges*
<transport name>	-	the string identifier used when creating a MamaTransport object
<property name>	-	the name of the property at the messaging middleware level

The property name and property value can be separated by either a space (' ') or an equals sign ('=').

Example 10: Avis example property

```
mama.avis.transport.avis_sub.url=elvin://localhost:7777
```

5.3 Transport Runtime Attributes

There are several transport runtime attributes that can be set after allocation.

The API provides throttling of subscription creation and the rate at which recaps can be sent from the API. The default values are 500 per second for initial values (subscription creation) and 250 per second for recap requests. Subscription requests are throttled on the default throttle and recaps on the recap throttle.

The throttles are created and can be configured on a per transport basis. The values for the throttle rates can be changed at runtime via calls to `MamaTransport.setOutboundThrottle()`.

Example 11: Setting throttles

```
/*Change the value of the default throttle to 1000 msg/sec*/
transport.setOutboundThrottle (1000.0, MAMA_THROTTLE_DEFAULT);

/*Change the value of the recap throttle to 500 msg/sec*/
transport.setOutboundThrottle (500.0, MAMA_THROTTLE_RECAP);
```

A callback can also be supplied to register for any transport level events that occur. This is dependent on the underlying messaging middleware being used, as not all provide this type of information.

Example 12: Setting callbacks

```
MamaTransport.setTransportCallback (callback);
```

5.4 Load Balancing Transports

A transport represents an isolated communication channel for data traffic. However, it is also possible to create a load-balanced transport, which makes available a set of channels. Different channels may be selected to balance the amount of traffic transmitted across this set. The selection could be made at random, or in a round-robin fashion, or in some other way according to user requirements. Load balancing is only available on ***NYSE Technologies Data Fabric***.

Two types of load balancing scheme are possible:

- *Client Load Balancing*: In this case, a client selects one of the channels for use. Different clients may then use different channels.
- *Subscription Load Balancing*: In this case, a client may select one of the channels for each symbol it is registering an interest in. Different symbols may then use different channels.

5.4.1 Properties for Load Balancing

A number of properties may be used for load balancing:

lb<n>, lb_scheme, lb_shared_object

There are a number of different load balancing schemes available. Each scheme requires a set of transports to be created under the same name. This set of transports is then available for making different selections. The set of transports is defined by adding "*lb*" terms after the name of the transport. These terms need to be numbered consecutively from 0, without gaps in the numbering, as illustrated in the following example.

Example 13: Defining the transport set for load balancing

```
mama.wmw.transport.sub.lb0.subscribe_address_0=cache0
mama.wmw.transport.sub.lb0.subscribe_port_0=1457
mama.wmw.transport.sub.lb1.subscribe_address_0=cache1
mama.wmw.transport.sub.lb1.subscribe_port_0=1457
```

Once the transport set is defined, one of two load balancing schemes may be chosen: either select a transport from the set of transports and use this transport to create all subscriptions (client load balancing), or select a transport from the set of transports each time a subscription is created (subscription load balancing).

Client Load Balancing

With client load balancing, a transport is selected at random and used for all subscriptions. Create a callback, such as the following, to override this random selection and instead make the selection from a shared object (dynamic link library) or directly in code:

```
void mamaTransportLbInitialCB (int  numTransports, int* transportIndex);
```

This must pass back an index (0 <= transportIndex < numTransports). In code, this callback is set with the following function:

```
mama_status
mamaTransport_setLbInitialCallback (mamaTransport            transport,
                                    mamaTransportLbInitialCB callback);
```

Subscription Load Balancing

The default behavior for the subscription load balancing scheme is to begin from the first transport for the first subscription, then select transports in round-robin for each subsequent subscription created. This initialization and round-robin selection may be overridden by using the mamaTransportLbInitialCB callback described above to choose the initial transport, and by creating a mamaTransportLbCB callback to make the selection in a shared object, dynamic link library, or directly in code, as follows:

```
void mamaTransportLbCB (int         curTransportIndex,
                        int         numTransports,
                        const char* source,
                        const char* symbol,
                        int*        nextTransportIndex);
```

This must pass back an index (0 <= transportIndex < numTransports). The source and symbol name are passed to assist the decision process, for example, splitting the subscriptions alphabetically. In code, this callback is set with the function:

```
mama_status
mamaTransport_setLbCallback (mamaTransport     transport,
                             mamaTransportLbCB callback);
```

Summary of Properties

The load balancing schemes described above are controlled by a set of four properties:

round_robin: This will round robin the transport used for each subscription.
static: This will select a transport at random to use for all subscriptions.

round_robin: This will round robin the transport used for each subscription.
api: This will allow the use of `mamaTransport_setLbInitialCallback` and `mamaTransport setLbCallback` to set callbacks to override default behaviour.
library: This will allow the use of a shared object (dynamic link library) with loadBalanceInitial and loadBalance entry points to override default behaviour.

Example 14: Use the alpha shared object containing implementations of loadBalanceInitial and loadBalance

```
mama.wmw.transport.sub.lb_scheme = library
mama.wmw.transport.sub.lb_shared_object = alpha.so
```

Example 15: Round robin the transports used to create each subscription

```
mama.wmw.transport.sub.lb_scheme = round_robin
```

Restrictions on Load Balancing

Load balancing is currently only available under the ***NYSE Technologies Data Fabric***. In the event that one of the transports within a load-balanced group is down, the subscriptions will not automatically fail over to another transport within the group.

6 Events and Queues

Events and queues are the core functionality of the ***OpenMAMA API***, as they enable asynchronous, event-driven data processing. ***OpenMAMA*** applications register interest in events, from data arriving on a socket for a subscription or being informed of an elapsed timer, and execute application code through callbacks in response to them.

Each event created needs a corresponding callback function created that is invoked by the API once an event of the specific type occurs. A closure can also be specified for the event when it is being created. A closure is an arbitrary piece of data which is returned to the user in the callback. It provides a mechanism by which a user can associate context between the code in the callback and the application environment. Closures are specified with the argument type `void*`.

Events, when they occur within the API, are placed onto an event queue. An event queue is the mechanism that controls the dispatching of events within the API. These events result in the invocation of a callback for that event type once the event has been dispatched from the event queue.

Dispatching an event involves removing the next available event from the event queue, identifying the event type, and invoking the corresponding callback registered for that event type.

Queues are represented by the MamaQueue object.

The API maintains a default internal data queue for each bridge which it uses for internal timers and controls such as the throttling of subscriptions. You can use the default event queue for a bridge when creating subscriptions, timers, and so forth. Dispatching on this queue starts once `Mama.start()` is called. In this case the API is essentially being used in single threaded mode. The default event queue should never be destroyed. The call to `Mama.start()` is reference counted and each call must have a corresponding call to `Mama.Stop()`. Dispatching stops when `Mama.Stop()` is called for the final time. The `Mama.start()` and `Mama.Stop()` calls are also thread safe.

When a queue is being actively dispatched, events on that queue may only be dispatched by the dispatching thread. If dispatching is stopped then an event object can be destroyed from any thread. This is true for both the default queue and user-created queues.

In ***OpenMAMA*** all data is propagated in response to events being dispatched from one or more event queues. Callbacks registered with the API are invoked on the threads dispatching on particular queues whenever an event is available. Internally, the middleware adds data onto the event queue and it is the responsibility of the application code to dispatch events from these queues in a timely fashion.

6.1 Accessing the Internal Event Queue

Example 16: Accessing the internal event queue

```
MamaQueue* queue = NULL;
queue = Mama::getDefaultEventQueue (bridge);
```

6.2 Creating Queues

For multi-threaded dispatching and for more control over the de-queuing of events in the API you can create your own queues from which events can be manually dispatched, as illustrated in the following example.

Example 17: Creating queues

```
MamaQueue* queue = new MamaQueue;
queue->create (bridge);
```

6.3 Destroying Queues

An ***OpenMAMA*** queue can only be destroyed if all the objects using it have been destroyed first (such as timers, inboxes and subscriptions).

Objects like these are destroyed asynchronously which means that there is a time delay between calling the 'destroy' function and the object actually being destroyed. The object is deemed to be destroyed whenever it is impossible for further events to be placed onto the queue on its behalf.

Each time one of these objects is created, a lock count is incremented on the associated queue, which is only decremented when that object is fully destroyed.

6.3.1 How to Destroy a Queue

An ***OpenMAMA*** queue can be destroyed using the `mamaQueue_destroy()` function shown in the following example. If there are any open objects on the queue then an error code is returned.

Example 18: mamaQueue_destroy()

```
/* Attempt to destroy the queue. */
mama_status status = mamaQueue_destroy(queue);
if(status == MAMA_STATUS_QUEUE_OPEN_OBJECTS)
{
    printf("Can't destroy queue as there are open objects.\n");
}
```

If the objects using the queue have had their 'destroy' functions called but have not been fully destroyed, then the `mamaQueue_destroyWait()` function can be used. This blocks and processes messages from the queue until all objects have been fully destroyed:

Example 19: mamaQueue_destroyWait()

```
/* Block until all objects have been destroyed. */
mamaQueue_destroyWait(queue);
```

The `mamaQueue_destroyTimedWait()` function behaves in the same manner as `mamaQueue_destroyWait()` but only blocks for the supplied timer period. Once this period elapses a timeout error is returned if all open objects have not been destroyed.

Example 20: mamaQueue_destroyTimedWait()

```
/* Process messages for 6 seconds. */
mama_status status == mamaQueue_destroyTimedWait(queue, 6000);
if(status == MAMA_STATUS_TIMEOUT)
{
          printf("Timed out waiting for queue to be destroyed.\n");
}
```

Additionally, the `mamaQueue_canDestroy()` function indicates if all objects using the queue have been destroyed:

Example 21: mamaQueue_canDestroy()

```
/* Check if queue can be destroyed. */
mama_status status == mamaQueue_canDestroy(queue);
if(status == MAMA_STATUS_QUEUE_OPEN_OBJECTS)
{
    printf("Queue cannot be destroyed as it is still in use.\n");
}
else if(status == MAMA_STATUS_OK)
{
    printf("Queue can be safely destroyed.\n");
}
```

6.3.2 Destroying the Default Queue

Objects can be created on the default queue that are only destroyed when the `mama_close` function is called. The `mamaQueue_destroyTimedWait()` function is called internally by ***OpenMAMA*** and blocks for two seconds, after which the timeout error is returned.

The two second time period can be increased by use of the following entry in the `mama.properties` file, expressed in milliseconds:

```
# Increase the native queue wait to 5 seconds
mama.defaultqueue.timeout = 5000
```

6.3.3 Object Destroy Notifications

Ideally, applications will not attempt to destroy the queue until all the objects using it have been destroyed first. ***OpenMAMA*** can provide notification that an object has been fully destroyed by invoking a call-back function. The method of registration depends on the language being used.

The following example code snippet shows how to register for the callback. The example application mamaInbox2 provides a full working demonstration.

Example 22: Creating a timer

```
class TimerCallbackEx : public MamaTimerCallbackEx
{
    virtual void onDestroy(MamaTimer *timer)
    {
       cout << "Timer has been destroyed." << endl;
    }

    virtual void onTimer(MamaTimer* timer)
    {
cout << "The timer has ticked." << endl;
```

```
    }
}

// Create a 2 second timer
MamaTimer timer;
TimerCallbackEx timerCallbackEx;
timer.create(queue, timerCallbackEx, 2);
```

6.3.4 Debugging Queue Destroy

If a programmer creates an object but forgets to destroy it, then the `mamaQueue_destroyWait` function will block forever. Tracking down the offending object can be difficult for a large application and so ***OpenMAMA*** provides assistance via the object lock tracking property, which can be enabled in the `mama.properties` file.

Example 23: Enabling queue tracking

```
# Turn on queue tracking
mama.queue.object_lock_tracking = 1
```

With this property turned on, ***OpenMAMA*** writes a log message every time the queue lock count is increased or decreased. The following output is an example of this message written at 'NORMAL' level.

Example 24: Queue lock count log messages

```
2011-04-26 13:04:15:296: mamaQueue_incrementObjectCount(): queue 0x0092E600, owner
0x0092D480, new count 1.
2011-04-26 13:04:15:296: mamaQueue_incrementObjectCount(): queue 0x0092E600, owner
0x00E65A20, new count 2.
2011-04-26 13:04:15:296: mamaQueue_incrementObjectCount(): queue 0x0092E600, owner
0x00E65D30, new count 3.
2011-04-26 13:04:15:296: mamaQueue_incrementObjectCount(): queue 0x0092E600, owner
0x00E6F488, new count 4.
2011-04-26 13:04:15:296: mamaQueue_incrementObjectCount(): queue 0x0092E600, owner
0x00E65F00, new count 5.
```

Additionally, a block of memory is allocated at this point, which is freed whenever the lock count is decremented. This allows a memory leak detection tool to display the stack trace and pinpoint the location where the object was created. The following output is from valgrind running on linux and shows that a subscription was created in the `subscribeToSymbols` function that was not destroyed.

Example 25: valgrind output

```
==15766== 8 bytes in 1 blocks are possibly lost in loss record 26 of 245
==15766==    at 0x4906795: calloc (vg_replace_malloc.c:418)
==15766==    by 0x4A72BED: mamaQueue_incrementObjectCount (queue.c:551)
==15766==    by 0x4A7A99A: mamaSubscription_setupBasic (subscription.c:649)
==15766==    by 0x4A7E562: mamaSubscription_setup (subscription.c:3238)
==15766==    by 0x4A7A119: mamaSubscription_create_ (subscription.c:349)
==15766==    by 0x4A7AC09: mamaSubscription_create (subscription.c:732)
==15766==    by 0x403BB8: subscribeToSymbols (in /var/userspace/gclarke/work/mama-50-
dev/install/examples/mama/mamalistenc)
==15766==    by 0x40379A: main (in /var/userspace/gclarke/work/mama-50-dev/install/
examples/mama/mamalistenc)
```

Note The object lock tracking is turned off by default as it may cause degradation in application performance.

6.4 Dispatching

OpenMAMA provides the following mechanisms by which events can be dispatched from a queue:

1. Blocking Dispatching

   In this case a call to `MamaQueue.dispatch()` simply blocks, constantly dispatching events from the queue until `MamaQueue.stopDispatch()` is called. `stopDispatch()` can be called from within a callback function/method from that queue or from another thread (which may be dispatching events from a different queue).

2. Timed Dispatching

   OpenMAMA provides a `MamaQueue.timedDispatch()` function, the behaviour of which is dependent on the underlying middleware being used. When using TIBCO Rendezvous the function behaves as `tibrvQueue_timedDispatch()` whereby the function blocks until an event has been dispatched or the specified timeout interval has elapsed. When using ***29West LBM*** or the ***NYSE Technologies Data Fabric***, the function only unblocks after the interval has elapsed regardless of events being dispatched or not.

3. Dispatch a single event

   The `MamaQueue.dispatchEvent()` function dispatches a single event from the queue and returns. If there are no events on the queue the function returns immediately.

Note Timed dispatching or single event dispatching are only available on queues created externally to the API. They are not available on the internal default event queue.

For full control over dispatching and threading in an application, you can create your own threads from which to dispatch on each queue created. Alternatively a `MamaDispatcher` can be used, which simply creates a new thread and starts dispatching (using `MamaQueue.dispatch()`) on the specified queue.

6.5 Queue Monitoring

The API provides the ability to receive notifications when certain conditions on an event queue occur. For instance, many application developers find it useful to know if the event queue is backing up, an early indication of a slow consuming client application.

The API currently supports registering for callbacks to be invoked when a high watermark for the queue is reached, that is, the number of outstanding events on the queue reaches a specified threshold, and for when the number of events returns to a low watermark.

Callbacks can be registered with a MamaEventQueue. These callbacks are invoked when certain conditions on the event queue are met. Which callbacks can be called and under what conditions they

are called is middleware-dependent. Details on middleware-specific functionality within the API can be found in the *Section 18: Configuration Reference*.

In all object orientated languages the callbacks are defined as concrete implementations of the `MamaQueueMonitorCallback`.

The high and low watermarks for an event queue are set via calls to `MamaQueue.setHighWatermark()` and `MamaQueue.setLowWatermark()` respectively. Callbacks for monitoring will not be called unless the watermark values have been set for a queue.

It is recommended that users specify a name for each queue being monitored. This will aid in logging and debugging the application.

If a monitoring feature is not available on a particular middleware an error will be returned, or a `MamaStatus` exception thrown.

Example 26: Queue monitoring

```
//Callback class for receiving queue monitor events
class QueueMonitorCallback : public MamaQueueMonitorCallback
{
public:
     QueueMonitorCallback          () {}
     virtual ~QueueMonitorCallback () {}

     virtual void onHighWatermarkExceeded (MamaQueue* queue, size_t size, void*
closure)
     {
         //Process the high watermark exceeded event
     }


     virtual void onLowWatermark (MamaQueue* queue, size_t size, void *closure)
     {
         //Process the low watermark event
     }
}

...

MamaQueue* queue =  new MamaQueue;

queue->create (bridge);

queue->setQueueName ("MyEventQueue");

queue->setQueueMonitorCallback (new QueueMonitorCallback (), "Closure);

queue->setHighWatermark (10000);
queue->setLowWatermark (900);
```

6.5.1 Event queue size

An application can find out the number of events currently on the event queue at any point, independent from the queue monitoring callbacks. This is available as `MamaQueue.getEventCount()`.

6.6 Queue groups

The C++ APIs provide a further level of abstraction in the form of the MamaQueueGroup. The MamaQueueGroup, upon construction, creates the specified number of MamaQueue and MamaDispatcher objects and starts dispatching on the queues. The threading and dispatching is hidden from the application developers. The `MamaQueueGroup.getNextQueue()` returns the next available queue using round-robin, ensuring an even distribution of event sources across queues. For more control over dispatching of events within an application we recommend that you manipulate and manage queues directly by the application.

6.7 Developer Tips

1. Integrate an external event loop.

   The combined use of `MamaQueue.dispatchEvent()` and `MamaQueue.setEnqueueCallback()` can be used to integrate an ***OpenMAMA*** event loop into the event loop of another application when multi threading is being avoided. The `MamaQueue.setEnqueueCallback()` allows an application to be notified of enqueue events on a specified queue without having those events being dispatched. In the `mamaQueueEnqueueCB()` callback function the application can post an event to its own event queue. When this event is dispatched the application then calls `dispatchEvent()`. This strategy is sometimes employed in GUIs.

2. One to one relationship between queues and threads.

   For use of the API within the **MAMA Advanced Publisher** you have to maintain a one to one relationship between queues and dispatching threads. This is due to the internal use of message level sequence numbers used for data quality purposes. If dispatching from a single event queue coming from across multiple threads, the API can no longer enforce data quality correctly as messages cannot be guaranteed to be dispatched in the order they were received.

3. Minimize event queue growth.

   It is recommended that as little time as possible is spent executing code in the callback functions/ methods. Failure to do so can result in the event queue growing, ultimately running out of memory and/or message loss.

4. Using multiple queues/threads.

   There are several reasons why you may choose to use multiple threads/queues within an application built using the ***OpenMAMA API***:

   - The application already uses an event loop to control processing. In this case the application cannot use the main thread to call `Mama::start()`. The alternative here is to run `Mama::startBackground()` which spawns a background thread on which to start dispatching on the default event queue or to integrate the two event queues (see point 1 above).

   - The application requires fine grained control over how events are dispatched. When using the default event queue developers have no control over how dispatching occurs. In order to leverage the further control provided by the queue object, developers must create their own event queues and dispatch from these on a separate thread. When this level of control is required it is recommended that the MamaQueueGroup is not utilized.

   - The nature of the application dictates that separate threads are used for processing data events.

Consider the following scenario:

One event source provides message updates which require a short processing time per message. A second event source provides message updates which require considerably more CPU cycles to process each message. To avoid this second source negatively effecting the processing of the first source of messages it may be beneficial to distribute the processing across two threads; one for each message source. Note: On a single CPU machine this approach is unlikely to provide much additional benefit.

When using multiple queues/dispatchers or the MamaQueueGroup utility class, developers need to be aware that the callback functions/methods can be called from multiple threads.

7 Subscriptions

Subscriptions in ***OpenMAMA*** provide the ability to register interest in a source of data for a specific symbol. The subscription interface hides the underlying middleware specific subscription concept (e.g. Listener when using the TIBCO Rendezvous middleware and Receiver when using the 29West LBM middleware). The ***OpenMAMA API*** is used to subscribe to market data from a market data source, e.g. the **MAMA Advanced Publisher**. The API also supports basic subscriptions, which are used to subscribe to data published using the***OpenMAMA*** publishing functionality (see *Section 13: Publishing* for details). All sections of this chapter, other than *Section 7.6: Basic Subscriptions*, refer to market data subscriptions.

Note Feed handlers are used throughout this section to illustrate subscribing to market data. A **MAMA Advanced Publisher** can also be used.

7.1 Life Cycle of the MAMA Subscription

The ***OpenMAMA*** subscription moves through a number of different states during its lifetime, as defined in the following table.

Table 4: OpenMAMA Subscription States

State	Description
Unknown	The state of the subscription is unknown.
Allocated	The subscription has been allocated in memory.
Setup	Initial setup work has been done, `mamaSubscription_activate` must still be called. Note that this state is only valid for market data subscriptions.
Activating	The subscription is now on the throttle queue waiting to be fully activated.
Activated	The subscription is now fully activated and is processing messages.
Deactivating	The subscription is being de-activated, it will not be fully deactivated until the `onDestroy` callback is received.
Deactivated	The subscription has been de-activated. Messages are no longer being processed.
Destroying	The subscription is being destroyed, it will not be fully destroyed until the `onDestroy` callback is received.
Destroyed	The subscription has been fully destroyed.
De-allocating	The subscription is in the process of being de-allocated, this state is only valid if the `mamaSubscription_deallocate` function is called while the subscription is being destroyed.
De-allocated	The subscription has been de-allocated. This state is only temporary and exists until such point as the subscription's memory is freed. It is provided so that a log entry will be written out.

The transitions between the various states are shown in the following state machine diagrams. The transitions also show the function calls that are required to change state.

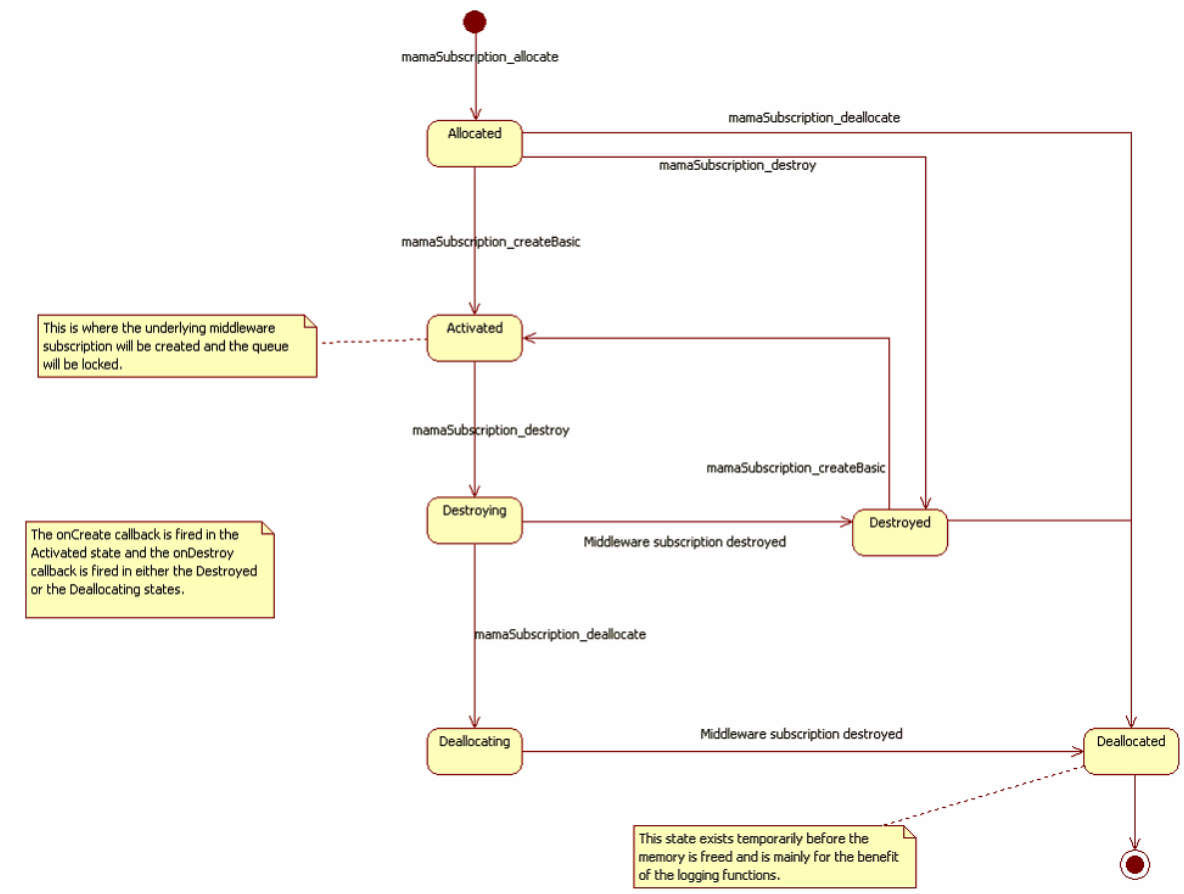


Figure 1: State transitions for a Basic Subscription

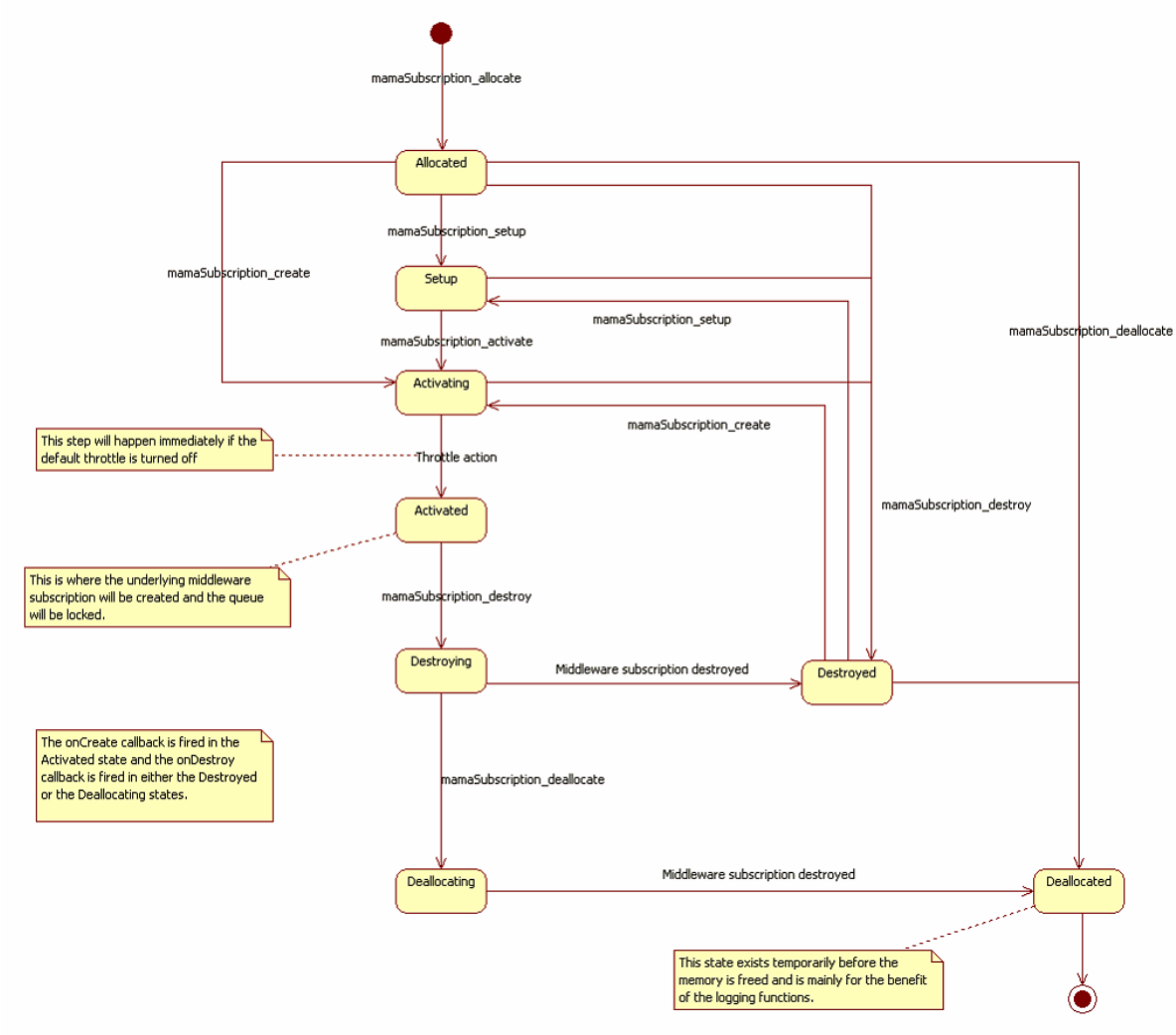


Figure 2: State transitions for a Market Data Subscription

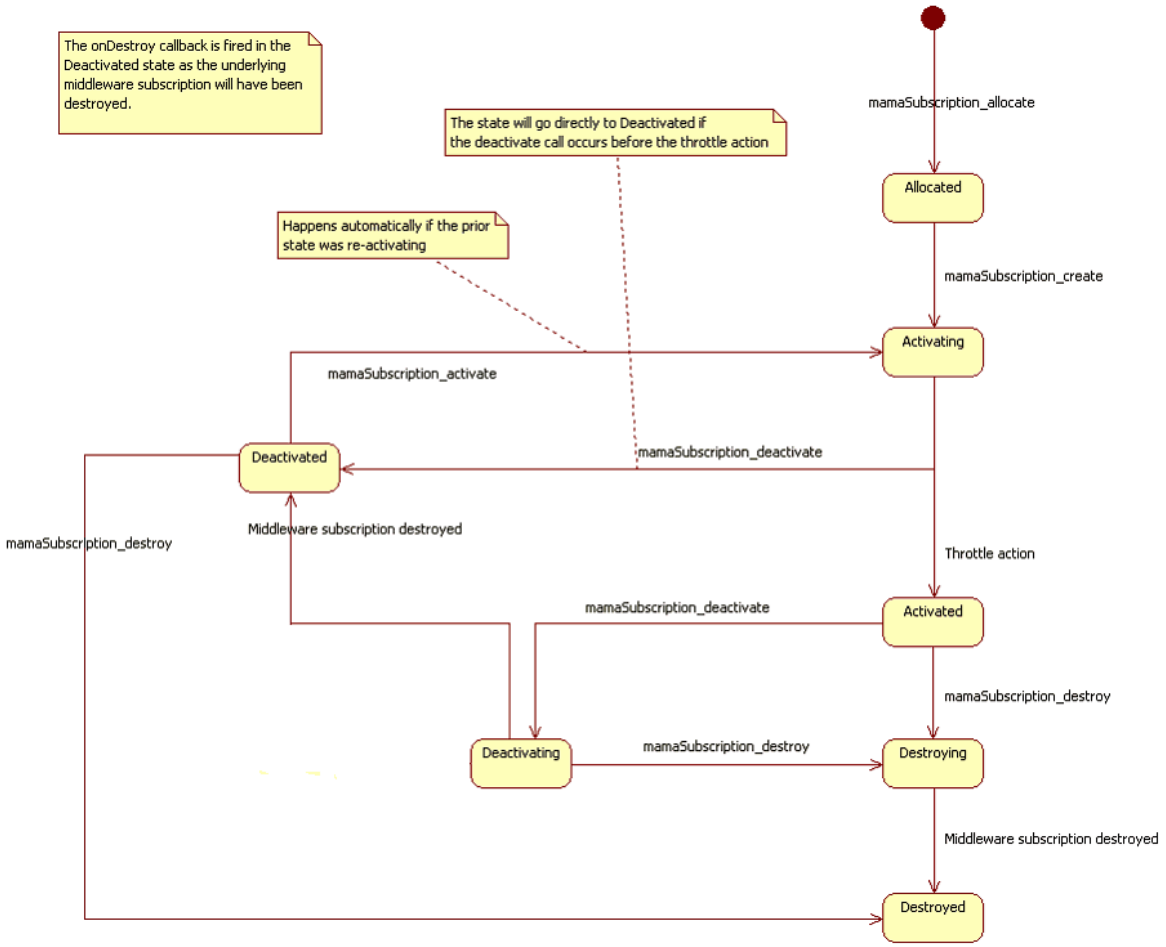


Figure 3: Activating and de-activating the Market Data Subscription

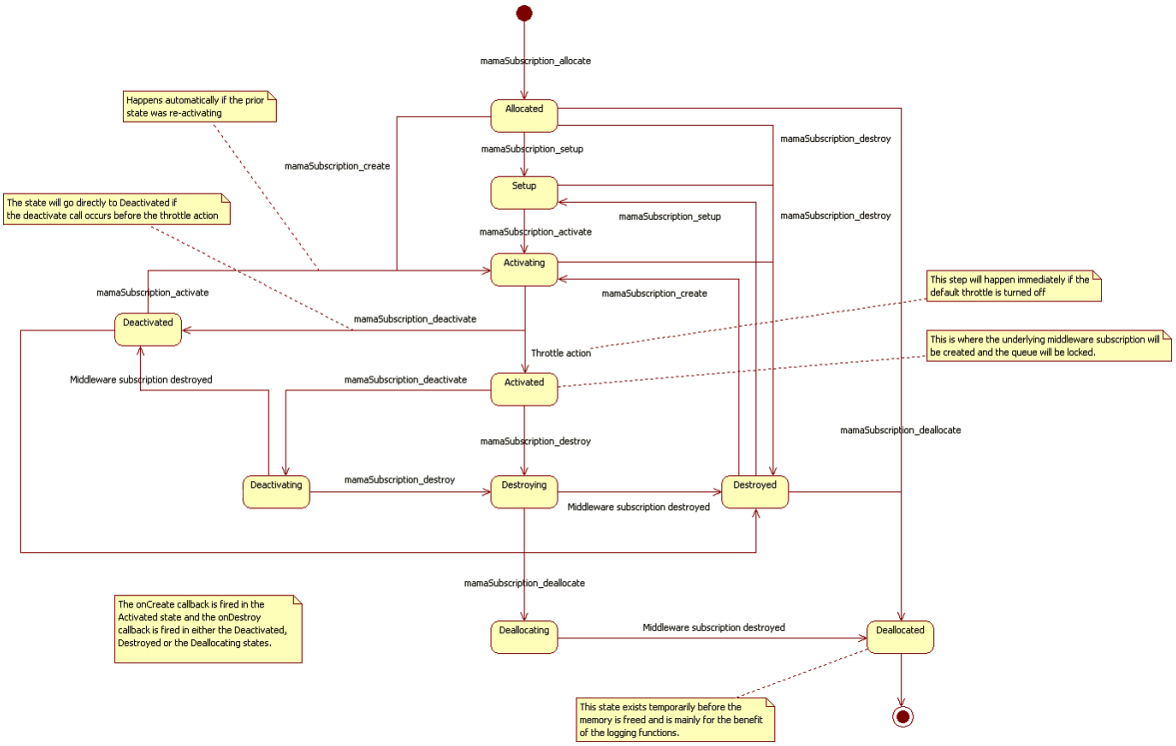


Figure 4: The complete state machine for a Market Data Subscription, including the main flow and the activation states

This lifecycle is reflected in the `mamaSubscriptionState` enumeration. The current state of the subscription can be obtained using the `mamaSubscription_getState` function. This replaces the legacy `mamaSubscription_isActive` and `mamaSubscription_isValid` function calls which are now deprecated. These functions can still be used and will return 'True' if the subscription is in the Activated state.

Additional Notes:

1. If a function such as `mamaSubscription_activate` is called and the subscription is not able to make an appropriate transition, (as described by the state machine), then an error code of MAMA_STATUS_SUBSCRIPTION_INVALID_STATE will be returned.
2. Calling `mamaSubscription_deactivate` on a subscription in the allocated, setup, deactivated or destroyed state will have no effect and the function will simply return MAMA_STATUS_OK.
3. The unknown state indicates an error condition and should not normally occur.

7.2 Common Regular Subscription Behaviour

OpenMAMA provides a number of subscription types that are described in *Section 7.5: Subscription Types*. All subscriptions share the following common concepts.

7.2.1 SymbolNamespace & Symbol

When a subscription is created, the user specifies a mandatory symbol namespace (e.g. "NASDAQ") and a mandatory symbol (e.g. "MSFT").

The symbol is the instrument identifier and should be unique when combined with the source. The ***NYSE Technologies Market Data Platform Feed Handler Suite*** refers to this as the issue symbol.

The symbol namespace is a logical feed handler group identifier as configured by the feed handler administrators. It can be thought of as a namespace qualifier for market data and is useful in separating two publishers of data with the same symbology on the same transport settings.

For example, with a NASDAQ UTP feed handler configured with a namespace of "UTP" and the symbol "MSFT", using the default UTP symbology on the feed handler, represents the identifier for the NBBO (National Best Bid and Offer) for Microsoft equities from NASDAQ.

7.2.2 Callbacks

Callbacks are registered with subscriptions when they are created. The callbacks are described the following table.

Table 5: Callbacks

Callback	Invoked
`onMsg()`	When data arrives from the network for the subscription, an event is created for the data and placed on the specified event queue. When the event is dispatched from the queue the `onMsg()` callback is invoked, passing the message details to the application for processing.
`onCreate()`	Invoked when a subscription creation is completed. For basic subscriptions, this will be immediately. For regular subscriptions, this will be when the subscription creation has been executed from the creation throttle.
`onQuality()`	Invoked when the quality of a subscription changes
`onError()`	Invoked when an error is encountered during the subscription creation process and subsequent data processing.
`onGap()`	(Optional)Invoked when a gap occurs in a market data subscription, or when a recap request is made.

Regular Subscriptions

The sending of subscription requests to the feed handler, and the registering of interests with the underlying middleware are throttled when a regular subscription is created. This throttle is always controlled by the internal default queue in ***OpenMAMA***, therefore, all subscription requests are sent by the thread calling `Mama::start()`.

When subscriptions are created using the default queue, `onCreate()` is guaranteed to be called before `onMsg()` and always from the thread calling `mama_start()`. However, if subscriptions are created on a user-created mamaQueue there is a possibility that `onMsg()` could get invoked before `onCreate()`. This is because `onCreate()` is called on the thread invoking `Mama::start()` whereas the `onMsg()` is called on the thread dispatching from the queue that was associated with the subscription upon

creation.

Essentially, `onCreate()` is called as soon as the subscription request has been issued and the interest registered with the middleware. It is at this stage that the subscription creation has been picked off the throttle. For a user-created queue, because the data for the subscription is being dispatched on a separate thread, the possibility exists for the data to return from the publisher and be dispatched before the `onCreate()` callback gets scheduled to execute by the operating system thread scheduler.

This race condition exists as it was decided to keep the data callbacks lock-free for performance reasons. Regardless of which order the `onCreate()` and `onMsg()` callbacks are invoked, at the time of invocation the subscription can be treated as if it has been fully created and is considered valid in both cases.

Basic Subscriptions

Basic subscriptions are created immediately rather than being throttled, therefore all callbacks are invoked on the thread dispatching from the queue associated with the subscription.

7.2.3 Initial Images

By default, **MAMA Advanced Publishers** do not send all data for a symbol with every update. They only send modified data (deltas) or modified data along with additional specific static data. This saves on processing in the construction of messages and bandwidth when sending data over the network. Either option is more efficient than sending all data with every update.

While this is a very efficient mechanism, subscribing applications may have to wait an arbitrary length of time to obtain the latest value of all available published fields. To address this problem the ***OpenMAMA API*** provides the concept of initial images.

An initial image is a special message type that contains all the available fields in the market data publishers cache that are configured to be published. It is effectively a snapshot in time of the net effect of all updates on the instrument up to the point of subscription. As initial images contain all the fields in the publisher cache, the message size tends to be significantly larger than subsequent updates.

Initial images are identified as such through use of the utility function, `MamaMsgType.typeForMsg()`. The MdMsgType reserved field stores this information within a message.

If this initial snapshot of data is not required, for example, for tick capture systems where only individual updates are of interest, the receipt of initial images on subscription can be disabled via a call to `MamaSubscription.setRequiresInitial(false)`.

In this case, the creation of a subscription simply informs the **MAMA Advanced Publisher** of a new subscriber and to start publishing for that instrument if not already doing so.

The rate at which initial images are sent can be controlled at the transport level.

7.2.4 Recaps

A recap is an initial image that is sent to the API to provide the client with the latest snapshot for a symbol in response to a data quality event on the infrastructure.

Recaps can be solicited, in that they are requested from the API in response to a sequence number gap being detected in the inbound messages for a subscription. See *Section 12: Data Quality* for details on sequence number checking in the API.

Conversely, recaps can be unsolicited, in that they are sent by the feed handlers under certain circumstances, for example, in the event of a cancel or correction being received by the **MAMA Advanced Publisher**.

The rate at which recaps are sent from the API can be controlled at the transport level.

7.2.5 Timeout/Retries

A **MAMA Advanced Publisher** may not respond to a subscription or initial image request in a timely fashion for a variety of reasons, such as network problems or being overloaded with requests. To further ensure receipt of an initial image within the API a subscription supports the concept of timeout intervals and a number of subscription retries before giving up and reporting a timeout error via the registered callbacks.

The number of retries, set via a call to `MamaSubscription.setRetries()`, specifies the number of attempts made to obtain an initial image for a subscription, with an interval defined by the timeout as described above.

The defaults here are three retries with a 10 second timeout between each retry.

7.2.6 Refreshes

A **MAMA Advanced Publisher** has no mechanism by which it can detect a subscribing client shutting down (i.e. In the case of an uncontrolled shutdown). Instead, there is the concept of refresh messages sent from the client to the **MAMA Advanced Publisher** to indicate that there is still some interest in the data being published. Refresh messages are sent once an hour, distributed over the hour, for each symbol a client has subscribed to. The **MAMA Advanced Publisher** stops publishing data for a symbol if it has not received a refresh message for that symbol during a defined, configurable, period of time (default 60 minutes).

To reduce the possibility of all clients sending refresh messages for the same symbols the **MAMA Advanced Publisher** sends a response to a refresh message to all clients. Upon receipt of this message each client puts the refreshed symbol on the end of its refresh list. Using this mechanism a **MAMA Advanced Publisher** will not be flooded with refresh messages when one is sufficient.

7.2.7 Throttling of Subscription Creation

By default, when ***OpenMAMA*** subscriptions are created, the subscription request is not sent immediately from the API. Instead, the request is placed on the default throttle queue to be sent at a later stage. Sending of subscription requests does not start until dispatching on the default event queue has commenced, i.e. by calling `start()`. This behaviour protects the messaging backbone from a storm of subscription requests and the **MAMA Advanced Publisher** from becoming overwhelmed with such requests.

The default throttling rate within the API, if none is specified, is 500 msg/sec. It is recommended that this value is significantly lowered in the following cases

- When using subscription types that result in larger initial values, e.g. Group and order book subscriptions.
- If there are a large number of subscribing applications that start up at the same time.

The throttling is controlled at the transport level and applies to all subscriptions created on that transport (see *Section 5: Transports*, for details).

7.2.8 Caching of Updates Prior to Initial

It is possible, on subscription creation, due to updates and initial values arriving along different communication paths, that the updates and initial values arrive out of sync. Consider the following scenario:

1. The **MAMA Advanced Publisher** is already publishing updates for a particular symbol (the client has subscribed after market open).

2. The client creates a subscription request.

3. The **MAMA Advanced Publisher** creates an initial image message from the current state of the cache for the subscribed symbol.

4. An update arrives to the **MAMA Advanced Publisher** for that symbol immediately after the initial value message has been created. The effects of this update on the **MAMA Advanced Publisher** symbol cache are published as normal.

5. The update from the previous step arrives at the client before the previously created initial value message. This could happen as the initial may be being published on TCP and the update on multicast.

6. The next update received at the client is not the next expected sequence number for that symbol. The client detects a gap and issues a recap request.

This scenario can lead to a number of recaps being send from a client resulting in an arbitrary amount of time before the client recovers. For instance if the scenario outlined occurred for an instrument that was not very liquid it may be some time before the feed handler receives and sends another update to the client.

To reduce the chance of this occurring the API can be configured to cache updates that arrive prior to receiving an initial image for a subscription. If the update subsequent to the initial value results in a gap being detected the API checks its message cache for the missing update before resorting to issuing a recap request.

The size of the cache used can be controlled on a per subscription basis using `subscription.setPreInitialCacheSize()`. The default is a cache of 10 messages to be stored prior to receiving the initial image.

7.3 Creating and Destroying Subscriptions

Subscriptions require that a transport, source and symbol are specified upon creation.

Subscriptions must be destroyed from within the subscriptions' own callbacks or from other event callbacks on the same queue as the subscription callbacks are being dispatched.

Example 27: Creating and destroying a subscription

```
class DisplayCallback : public MamaSubscriptionCallback
{
public:
    DisplayCallback () {}
    virtual ~DisplayCallback (void) {}

    virtual void onCreate  (MamaSubscription* subscription)
```

```
    {
        //The subscription has been created.
    }

    virtual void onError   (MamaSubscription * subscription,
                            const MamaStatus&  status,
                            const char*        symbol)
    {
        //An error was encountered during the processing of the subscription
    }

    virtual void onQuality (MamaSubscription* subscription,
                            mamaQuality       quality,
                            const char*       symbol)
    {
        //A data quality event has occurred. Likely to be a stale message
        //due to a sequence number gap.
    }

    virtual void onMsg     (MamaSubscription* subscription,
                            const MamaMsg&    msg)
    {
        //process the message
    }

    virtual void onGap (MamaSubscription*  subscription)
    {
        // A gap has been detected for the subscription
    }

    virtual void onRecapRequest (MamaSubscription*  subscription)
    {
        // A recap request has been sent for this subscription
    }
}

....

DisplayCallback*  callback     = new DisplayCallback;
MamaSubscription* subscription = new MamaSubscrption ();

.... //set MamaSubscription properties if required

/*Create the source for the data - contains symbolnamespace and trasport*/
MamaSource* source = new MamaSource ("SourceId", transport, "NASDAQ");

subscription->create (queue,
                      callback,
                      source,
                      "MSFT",
                      "My Closure");

....

//Destroy the subscription
subscription->destroy ();
delete subscription;
```

Note In the above example the call to `create()` is the equivalent of calling `setup()` followed by `activate()` on a subscription.

7.4 Reusing Subscriptions

It is possible to reuse already created subscriptions within the C++ version of the API. A subscription can be destroyed and subsequently recreated, or it can be deactivated and subsequently reactivated.

When a subscription is deactivated, all state objects created in the call to `setup()`, and associated with the subscribed symbol, are retained. The act of deactivating simply deregisters interest in the symbol with the underlying middleware. No more updates for the subscription are received after deactivate is called. Reactivating a subscription results in an initial value request being sent, if the subscription is using initial values, and interest in the symbol being registered with the underlying middleware. In this case the same symbol, symbolnamespace, transport and queue are used when the subscription is activated.

If reuse of the subscription for a different symbol is required the subscription can be destroyed. This tears down all state objects within the subscription that associate it with a particular symbol. The subscription can be recreated, using `setup()` or `activate()`, with completely new attributes enabling it to subscribe to a separate symbol.

7.5 Subscription Types

There are a number of different types of subscriptions that can be created using ***OpenMAMA***, depending on the nature of the data being subscribed to. The type of subscription being created and its behaviour are controlled via two properties of the subscription that can be specified at creation time. These are:

- SubscriptionType:
  - MAMA_SUBSC_TYPE_NORMAL - Regular market data subscription
  - MAMA_SUBSC_TYPE_BOOK - Order Book Subscription
  - MAMA_SUBSC_TYPE_GROUP - Group subscription
  - MAMA_SUBSC_TYPE_BASIC - Basic Subscription
  - MAMA_SUBSC_TYPE_DICTIONARY - Dictionary Subscription
  - MAMA_SUBSC_TYPE_SYMBOL_LIST - Symbol List Subscription
  - MAMA_SUBSC_TYPE_SYMBOL_LIST_BOOK - Book Symbol List Subscription
- ServiceLevel: The additional behavior for the specific type of subscription. Currently only two of the values are supported:
  - MAMA_SERVICE_LEVEL_REAL_TIME - A real time subscription receives updates when they occur.
  - MAMA_SERVICE_LEVEL_SNAPSHOT - A snapshot subscription receives only an initial value and no subsequent updates.

The default values for these, if not explicitly specified upon subscription creation, are MAMA_SUBSC_TYPE_NORMAL and MAMA_SERVICE_LEVEL_REAL_TIME resulting in a real time market data subscription.

The behaviour of each of the subscription types below can be further qualified through the use of the service level.

Table 6: Subscription Types

Subscription Type	Value	Description
Normal (Market Data)	MAMA_SUBSC_TYPE_NORMAL	A regular market data subscription, used to subscribe to record-based instruments from the **MAMA Advanced Publisher**.
Order Book	MAMA_SUBSC_TYPE_BOOK	Use to subscribe to structured order books from **MAMA Advanced Publishers** that support the format. A structured order book comprises multiple price levels, each containing multiple entries (orders). The details of this structure differ depending on the underling message format used and any optimization configuration enabled on the **MAMA Advanced Publisher**. ***OpenMAMDA*** is required to leverage the power of structured order books.
Group	MAMA_SUBSC_TYPE_GROUP	A group subscription is an ***OpenMAMA*** concept, whereby a client can subscribe to a single 'group' symbol and receive initial values and updates on an arbitrary number of symbols associated with this symbol in the feed handlers.
Basic	MAMA_SUBSC_TYPE_BASIC	Allows subscription to non-market data that is being published via a ***OpenMAMA*** based publisher. As initial values are a concept specific to the **MAMA Advanced Publisher**, the only value of service level that is supported for basic subscriptions is MAMA_SERVICE_LEVEL_REAL_TIME (the default value when creating a subscription). A basic subscription does not provide initial values, recaps, data quality or refreshes. Throttling of subscription requests, however, is supported.
Symbol List	MAMA_SUBSC_TYPE_SYMBOL_LIST	Returns the full list of symbols from the **MAMA Advanced Publisher**. The actual symbol supplied when creating the subscription is irrelevant and may be NULL; only the subscription type matters. The symbols are returned as a field in a message callback, and are not stored anywhere else. It is up to the user to parse the MamaSymbolList field and store the symbols, or make individual subscriptions to each symbol. When the symbol list subscription is made, a series of initial messages will be received. Each of these messages will contain a field "MamaSymbolList" (FID 81). This field will contain a subsection of the full symbol list from the feed handler. By default, each initial will contain 500 symbols (this number is configurable in the feed handler). This field will be in the form of a comma separated list of strings. After one minute, a message of type MAMA_MSG_TYPE_END_OF_INITIALS will be received. At this point, no further initials will be received for the

Subscription Type	Value	Description
		symbol list subscription and it may be assumed that all symbols currently being published by the feed handler have been received. **Note**: It is not possible to determine how many initial messages will be received.
Book Symbol List	MAMA_SUBSC_TYPE_SYMBOL_LIST_BOOK	Has the same functionality as the symbol list subscription, but only returns order book symbols.

7.6 Basic Subscriptions

A basic subscription allows users of the ***OpenMAMA API*** to subscribe to non-market data that is being published via an ***OpenMAMA*** based publisher. A basic subscription is created by specifying a subscription type of MAMA_SUBSC_TYPE_BASIC. As initial values are a concept specific to the **MAMA Advanced Publisher**, the only value of service level which is supported for basic subscriptions is MAMA_SERVICE_LEVEL_REAL_TIME (the default value when creating a subscription). A basic subscription does not provide initial values, recaps, data quality or refreshes. Throttling of subscription requests, however, is supported.

8 Entitlements

When subscribing to market data, and when that data is configured to include entitle codes, the API enforces entitlements as defined in the entitlements server. Entitlements are not enforced for 'basic' subscriptions.

Note If developing against non-entitled APIs, please be aware of the the client responsibilities outlined in the licensing file, "Entitlements Check Disclaimer".

The ***OpenMAMA API*** obtains its user based entitlements on application startup. The entitlements for the logged on application user are obtained from a HTTP server. The API obtains its list of available entitlements servers from the `mama.properties` file. This occurs on the initial call to `Mama.open()`. The following example shows how to specify the location of entitlement servers using a comma separated list.

Example 28: Specifying the location of entitlement servers

```
entitlement.servers=server1:9090,server2:8080
```

The API checks each server, using round-robin, to locate the entitlements for the current user. If no entitlements servers are present, or no entitlements exist for the current user, the call to `Mama.open()` fails, with a status of ENTITLE_NO_SERVERS_SPECIFIED or ENTITLE_NO_USER, respectively.

Entitlements are enforced at two points during the subscription process:

1. At the point of subscription creation.

   When a subscription is being created the API checks the entitlement rules for the user to determine whether there are sufficient privileges to allow the user to create the subscription. This check is based on the symbol string being subscribed to. If the user is not entitled to subscribe to the symbol in question, the call to `create()` returns a value of MAMA_STATUS_NOT_ENTITLED.

2. On receipt of the first update to the client.

   The user must also be entitled to view data with a specific injected entitle code. When the API gets the first message for a subscription it checks if the user is entitled to view data with the specific entitle code. If the user is not entitled to this code the `onError()` subscription callback is invoked with an error code of MAMA_STATUS_NOT_ENTITLED.

Note The entitlements are requested via HTTP GET to the entitlements server. If this is not possible please contact your system administrator.

Note If a message with an injected entitle code is received by a basic subscription it will be rejected, the assumption being that this is a market data message and should not be available to basic data subscribers.

9 Threading

This section identifies the threads on which the various callbacks within the API are invoked. In the majority of cases the threading model used is the same across language variations and middlewares. Where the API deviates from this, details, and an explanation, are provided.

Note The "Default Queue" is the thread that invokes `Mama::start()`, and therefore the thread that is dispatching on the internal default event queue.

The "Dispatch Queue" is the thread currently dispatching on the associated event queue. This can be the default queue or a user created event queue (MamaEventQueue).

The following callbacks are those that are associated with a MamaSubscription.

Table 7: MamaSubscription Callbacks

Callback	Invoking thread	Language deviation	Middleware deviation
`onMsg()`	Dispatch Queue	none	none
`onError()`	Dispatch Queue	none	none
`onCreate()`	Default Queue[a]		none
`onQuality()`	Dispatch Queue[b]	See middleware specific variations.	NYSE Technologies Data Fabric: `onQuality()` events due to transport callbacks will be invoked from the transport thread. 29West LBM: `onQuality()` events are always on the dispatch thread.
`onGap()`	Dispatch Queue	none	none
`onRecapRequest()`	Dispatch Queue	none	none

Notes:

- The `onCreate()` callback is always called from the thread dispatching on the default queue. This is because subscriptions are centrally throttled on a per transport basis. The effect is when creating subscriptions using an ***OpenMAMA*** queue other than the default queue, it is possible that the `onMsg()` callback may get invoked prior to the invocation of the `onCreate()` callback. This behavior is being maintained as it is preferable to the overhead and latency that would be incurred in trying to synchronize the two.
- The majority of invocations of `onQuality()` will be called from the thread dispatching on the subscriptions associated event queue. When a subscription state is set to stale as a result of a detected gap, the `onQuality()` callback is always invoked from the dispatch queue.

Table 8: MamaTimer Callbacks

Callback	Invoking thread	Language deviation	Middleware deviation
`onTimer()`	Dispatch Queue	None	None

Table 9: Mamalo Callbacks

Callback	Invoking thread	Language deviation	Middleware deviation
`onIo()`	Dispatch Queue	None	None

Table 10: MamaTransport Callbacks

Callback	Invoking thread	Language deviation	Middleware deviation
All callbacks	See middleware specifics	See middleware specifics	NYSE Technologies Data Fabric: Transport level events (connect/ disconnect etc) are invoked from the IO thread internal to wombat middleware. 29West LBM: N/A

Table 11: Event Queue Callbacks

Callback	Invoking thread	Language deviation	Middleware deviation
`onHighWatermarkExceeded()`	Middleware specific	None	NYSE Technologies Data Fabric: Invoked from the thread dispatching on the monitored queue. 29West LBM: Invoked from the LBM context thread. This is the thread responsible for enqueing the event queue. Stalling in this callback can result in data loss as this thread is also responsible for draining incoming sockets.
`onLowWatermark()`	Middleware specific	None	As with `onHighWaterMarkExceeded()`
`onEvent()`	Dispatch Queue	None	29West LBM: LBM does not expose a mechanism to directly enqueue events onto an LBM queue. In ***OpenMAMA***, this functionality is implemented using zero-length timers. However, due to the nature of the timer implementation in LBM, under high load it is not possible to guarantee that the `onEvent()` callbacks for each event enqueued will fire in the same order as the events were enqueued.

Table 12: MamaDqPublisher Callbacks

Callback	Invoking thread	Language deviation	Middleware deviation
`onCreate ()`	Default Queue	None	None
`onNewRequest ()`	Dispatch Queue	None	None
`onRequest ()`	Dispatch Queue	None	None
`onRefresh ()`	Dispatch Queue	None	None
`onError ()`	Dispatch Queue	None	None

10 OpenMAMA Dictionary

The *Data Dictionary* is a data structure, obtained from the advanced publisher, which provides a mapping between field identifiers (FID's) and field names for a superset of all fields that can be sent on the platform. It also provides data type information for each of the fields.

Creating a data dictionary in ***OpenMAMA*** is similar to creating a subscription, as the data dictionary request is a specialized form of subscription.

10.1 Creating the Data Dictionary (from platform)

Example 29: Creating the Data Dictionary

```
class DictionaryCallback : public MamaDictionaryCallback
{
public:
    DictionaryCallback () {}
    virtual ~DictionaryCallback () {}

    virtual void onTimeout (void)
    {
       //Could not get dictionary. The timeout interval of 60 seconds has elapsed.
    }

    virtual void onError (const char* errMsg)
    {
        //An error was encountered fetching the dictionary.
    }

    virtual void onComplete (void)
    {
        ////The dictionary has been successfully fetched.
    }
}

....

DictionaryCallback callback;
MamaDictionary*    dict = new MamaDictionary;

dict->create (transport,
              queue,
              &callback,
              "WOMBAT");
```

The dictionary request has successfully completed and the mamaDictionary is available for use once the `onComplete()` callback function/method has been invoked.

If there has been no response to the data dictionary request, the `onTimeout()` callback function/method is invoked after 60 seconds has elapsed.

If any other error was encountered during the processing of the request, the `onError()` callback function/method is invoked passing back the appropriate error status.

The data dictionary can be obtained from a single **MAMA Advanced Publisher** instance. However, it is

more typical that a dedicated mamadict process is running on the network that provides a superset of all the fields being used across all the market data publishers on the market data backbone.

The default source value for data dictionary retrieval is "WOMBAT". This value is configurable for the dedicated dictionary publisher.

10.2 Using the Data Dictionary

The ***OpenMAMA*** representation of the *Data Dictionary* comprises a number of MamaFieldDescriptor objects, one for each field in the dictionary. As market data messages on the ***OpenMAMA*** platform generally only contain the FID, the dictionary can be interrogated at run time to find a full description of a message field. Not sending the field name is an optimization when sending messages, saving CPU processing time in the construction of the messages, and bandwidth by reducing the size of the messages.

The MamaFieldDescriptor can be obtained in three ways from the dictionary: by FID, by name, or through iteration across all field descriptors. For example, if the FID for a field is available the field descriptor for the field can be obtained using `MamaDictionary.getFieldByFid()`.

Once the MamaFieldDescriptor has been obtained, the field details can be accessed using the functions:

```
MamaFieldDescriptor.getName()
MamaFieldDescriptor.getFid()
MamaFieldDescriptor.getType()
MamaFieldDescriptor.getTypeName()
```

10.3 Developer Tips

The data dictionary can be serialized to and from a mamaMsg (the dictionary is received from the advanced publisher as a MamaMsg). The dictionary supports the following two operations to facilitate this: `mamaDictionary_getDictionaryMessage()` and `mamaDictionary_buildDictionaryFromMessage()`.

Note A new message is allocated for each invocation of `getDictionaryMessage()`. It is the responsibility of the caller to destroy the message(s) when no longer required.

Once the underlying mamaMsg for a dictionary has been obtained, the message bytes can be written to file (see *Section 11: Messages*). The message can then be later reconstructed from the bytes in the file and from this message the dictionary can be recreated. Memory for the new dictionary is allocated through the `mamaDictionary_create()` function.

Example 30: Serializing the data dictionary

```
MamaDictionary* dictionary = new MamaDictionary;
dictionary->buildDictionaryFromMessage (message);
```

11 Messages

The mamaMsg abstracted interface provides a wrapper for the underlying wire message formats supported on a particular messaging middleware.

11.1 Accessing Data

The mamaMsg object supports direct field access through a suite of strongly typed accessor functions/ methods. For example, `mamaMsg_getI8() -> mamaMsg_getF64()`.

A scalar field can be obtained through an accessor for a type larger than the one being accessed, when the larger type can hold the smaller without loss of precision. For instance, `mamaMsg_getI32()` can be used to get fields of type U16, I16, U8 etc.

In the C++ API, each of the accessor methods are overloaded with a version that accepts a MamaFieldDescriptor instead of the name/FID combination. It is recommended that applications use this variation of the accessor when accessing field data. A representation of the field data as a string can also be obtained via `mamaMsg_getFieldAsString()`. However, this is less efficient than using the correct type accessor for the field.

When accessing string fields using the `getString` method `MamaMsg.getString`, the result points to the copy of the string held internally in the mamaMsg object. This memory is owned by the object and does not need to be explicitly freed.

If one of the strongly typed accessors is called on a mamaMsg and the field is not found, the function returns a mamaStatus value of MAMA_STATUS_NOT_FOUND, or an exception is thrown (C++/Java).

11.2 Message Creation

Applications need to create their own mamaMsgs when using the publishing capability of the API.

A mamaMsg can be created in a number of ways. The default `MamaMsg.create()` creates a message with the default payload bridge `MamaMsg.createForPayload()` creates a message using the specified payload. A message can be recreated from a byte buffer using `MamaMsg.createFromByteBuffer()`. See *Section 11.5: Developer Tips* for more information.

Note The ***OpenMAMA API*** has an internal list of reserved fields used for passing message header information and other data. It is strongly recommended that users of the API do not use FIDs of 100 or lower, or the field 496 if using entitled APIs, as these are used to describe the internal reserved ***OpenMAMA*** fields.

Note The FID uniquely identifies a field within a message, not the FID/name combination. This is an important distinction as the name is only used to search for fields when a field with the specified FID is not found.

Example 31: Message creation

```
MamaMsg msg;
msg.create ();
msg.createForPayload (MAMA_PAYLOAD_AVIS);
```

Fields can be added to messages using the individually typed mutator functions available. When adding fields to messages both the field name and the FID can be specified.

11.3 Field Iteration

OpenMAMA enables iteration through all fields in a message. An application can pass a callback to `MamaMsg.iterateFields()` which is invoked for each field in the message. The callback function/ method provides access to a MamaMsgField object. The field type can be obtained from the MamaMsgField object and the appropriate accessor can be invoked. All typed accessor functions or methods for the MamaMsg are also available for the MamaMsgField object. The field also supports obtaining the field data as a string using `MamaMsgField.getAsString()`. This approach is less efficient than strong typed access, in the same way as when obtaining stringified data directly from a message.

Example 32: Field iteration

```
class MsgIteratorCallback : public MamaMsgFieldIterator
{
public:
    MsgIteratorCallback () {}
    virtual ~MsgIteratorCallback (void) {}

    virtual void onField (const MamaMsg&      msg,
                          const MamaMsgField& field,
                          void*               closure)
    {
        //process the data in the MamaMsgField
    }
}

....

MsgIteratorCallback callback;

msg->iterateFields (callback,
                    dictionary,
                    "My Closure");
```

There is also a separate iterator implementation as an alternative to the callback method. Using this the iterator points to a particular MamaMsgField object, and can be incremented to the next field by the user. A NULL is returned after the last field has been returned. It is possible to reset the iterator to the start of the message at any time.

Example 33: Field iteration without callback

```
MamaMsgIterator iterator (myMamaListen->getMamaDictionary ());
msg.begin(msgIterator);
while (*msgIterator != NULL)
{
    displayMsgField (msg, *msgIterator);
    ++msgIterator;
}
```

Each message can only have one iterator associated with it, though the same iterator can be used for more than one message.

11.4 Special Data Types

Some data types supported by a mamaMsg are specific to the ***OpenMAMA API***.

11.4.1 MamaDateTime

A date/time representation with additional hints for precision, advanced output formatting and support for time zone conversion (using the MamaTimeZone type).

Hints include:

- Whether the time stamp contains a date part, a time part, or both.
- The level of accuracy (if known) of the time part e.g., minutes, seconds, milliseconds, etc.

The output format strings are similar to that available for the `strftime()` function, plus:

- %; adds an optional (non-zero) fractional second to the string
- %: adds fractional seconds based on the accuracy hint (including trailing zeros, if the accuracy hint indicates it should).

The following table provides examples of output.

Table 13: MamaDateTime Examples

Actual Time	Output of "%T%;"	Output of "%T%:"
01:23:45 and 678 milliseconds	01:23:45.678	01:23:45.678
01:23:45 and 0 milliseconds	01:23:45	01:23:45.000

11.4.2 MamaPrice

MamaPrice is a special data type for representing floating point numbers that often require special formatting for display purposes, such as prices. MamaPrice contains the 64-bit (double precision) floating point value and an optional display hint.

A MamaPrice may be marked as valid or invalid. A valid MamaPrice is one that contains a currently valid value. This can be used to differentiate between a zero (valid) value and an absence of value for example. A MamaPrice is set to valid by default when it is created with a value, or a value is explicitly set.

The set of display hints includes hints for:

- the number of decimal places
- the fractional denominator, to a power of two
- special denominators used in the finance industry e.g., halves of 32nds

11.5 Developer Tips

When using the WombatMsg wire format and C++, it is more efficient to use field iteration rather than direct random field access. This results in significant performance improvements in the C++ ***OpenMAMDA API***.

The underlying message can be written to a byte buffer, which can be serialized to file. A message can be created from an existing byte buffer. The mamaMsg determines the underlying message format from the provided byte buffer and constructs the message accordingly.

Note The buffer returned is not a copy and therefore should not be altered once obtained. To do so can corrupt the message.

Example 34: Using a byte buffer

```
const char* buffer = NULL;
mama_size_t bufferSize = 0;
MamaMsg newMessage = new MamaMsg;
msg.getByteBuffer (&buffer,&bufferSize)
//write the buffer to file...
//read the buffer from file...
msg.createFromBuffer(buffer, bufferSize);
```

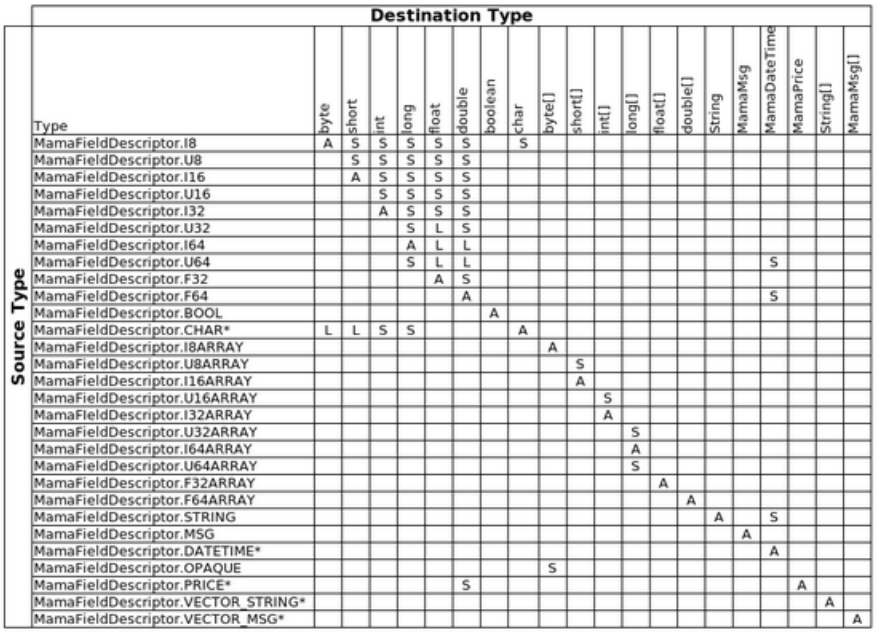
Obtain your MamaFieldDescriptor based on the field name from the data dictionary on application startup. Cache the MamaFieldDescriptors and use them for field access later on, removing the need to know the actual FIDs for individual fields in messages.

Do not delete messages received in subscription callbacks. The API reuses message instances for performance reasons. If a message is required to live beyond the scope of a callback use `MamaMsg.copy()` to create a deep copy of the message. Alternatively `MamaMsg.detach()` transfers ownership of the message from the API to the caller of the function. In this case it is the responsibility of the calling application to destroy the message when it is no longer needed. Similarly, when extracting sub-messages/arrays of sub-messages from a message, the memory for these is freed/deleted when the parent message goes out of scope in a callback or is deleted if owned by the application using the API.

11.6 MamaMsg Wire Format Conversion Matrix

Figure 5: MamaMsg Wire Format Conversion Matrix for Java shows the recommended possible data type conversions from MamaMsg wire format types when extracting data from a MamaMsg field that are supported across *all* middlewares and message types.

Note Other conversions may be possible depending on the middleware and message type being used.



Source Type \ Destination Type	byte	short	int	long	float	double	boolean	char	byte[]	short[]	int[]	long[]	float[]	double[]	String	MamaMsg	MamaDateTime	MamaPrice	String[]	MamaMsg[]
MamaFieldDescriptor.I8	A	S	S	S	S	S		S												
MamaFieldDescriptor.U8		S	S	S	S	S														
MamaFieldDescriptor.I16		A	S	S	S	S														
MamaFieldDescriptor.U16			S	S	S	S														
MamaFieldDescriptor.I32			A	S	S	S														
MamaFieldDescriptor.U32				S	L	S														
MamaFieldDescriptor.I64				A	L	L														
MamaFieldDescriptor.U64				S	L	L											S			
MamaFieldDescriptor.F32					A	S														
MamaFieldDescriptor.F64						A											S			
MamaFieldDescriptor.BOOL							A													
MamaFieldDescriptor.CHAR*	L	L	S	S				A												
MamaFieldDescriptor.I8ARRAY									A											
MamaFieldDescriptor.U8ARRAY										S										
MamaFieldDescriptor.I16ARRAY										A										
MamaFieldDescriptor.U16ARRAY											S									
MamaFieldDescriptor.I32ARRAY											A									
MamaFieldDescriptor.U32ARRAY												S								
MamaFieldDescriptor.I64ARRAY												A								
MamaFieldDescriptor.U64ARRAY												S								
MamaFieldDescriptor.F32ARRAY													A							
MamaFieldDescriptor.F64ARRAY														A						
MamaFieldDescriptor.STRING															A		S			
MamaFieldDescriptor.MSG																A				
MamaFieldDescriptor.DATETIME*																	A			
MamaFieldDescriptor.OPAQUE									S											
MamaFieldDescriptor.PRICE*						S												A		
MamaFieldDescriptor.VECTOR_STRING*																			A	
MamaFieldDescriptor.VECTOR_MSG*																				A

Key	
A	Always supported – same type
S	Supported conversion
L	Conversion with potential loss of information
	Unsupported conversion

*Only available for Wombat Message

Figure 5: MamaMsg Wire Format Conversion Matrix for Java

12 Data Quality

The ***OpenMAMA API*** provides a number of features to ensure the integrity of the inbound data when subscribing to market data.

Note This does not apply to 'basic', non market data, subscriptions.

Messages are sent with an injected sequence number field (MdSeqNum, FID 10) which contains a sequence number for each symbol (each individual symbol within a group subscription has its own sequence number). When a gap is detected in this sequence number ***OpenMAMA*** marks the message as being stale (STATUS_STALE) before passing the message to the client application. The API requests a recap image for the symbol in question. Once the recap has been received the API resets the internal expected sequence number and marks all subsequent messages as being OK (STATUS_OK).

Recaps are generally solicited from a client when a sequence number gap is detected. However, after a fault tolerant takeover the new primary may send unsolicited recaps for all instruments that have changed during a configurable interval. In this case the receipt of the recap message is not preceded with a gap callback. Recaps are sent using broadcast from the **MAMA Advanced Publisher**. Therefore, a client may receive an unsolicited recap as a result of another client's request. Again, this recap is not preceded with a gap callback notification. In each of these cases, all clients subscribed to the recapped instrument on that transport receive the recap.

The sending of recap requests is controlled in the same manner as initial value requests. A configurable number of recap requests can be sent across a specified interval. If a recap, after the specified number of attempts have been made, is not received, the `onError()` callback is invoked with a status of MAMA_STATUS_TIMEOUT. Under normal stable operating conditions a timeout for a recap should not occur. If encountered it is generally an indication of problems with the client and/or environment. The recommended action is to recreate the subscription (the abnormal condition possibly being transient) and alert administrator(s) to a possible application/environment issue. This course of action applies equally to recap timeouts for group subscriptions.

In the API, when a sequence number gap is detected, the subscription `onQuality()` callback is invoked with a value of MAMA_QUALITY_STALE. Once the condition has been resolved (a recap received), the `onQuality()` callback is invoked once again with a value of MAMA_QUALITY_OK. For C++, see the mamaQuality enum in `mama/subscription.h`.

Subscription data quality events can be captured by implementing the `onGap()` and `onRecapRequest()` subscription level callbacks.

The following describes the series of events that occur within the API when a gap is detected:

1. ***OpenMAMA*** detects a sequence number gap for an instrument.

2. The `onGap()` callback is invoked.

3. The subscription `onRecapRequest()` callback is invoked. A recap is requested if the subscription does not currently have a status of STALE and is not already waiting for the response to a recap request.

4. As this is a quality change from OK to STALE the `onQuality()` callback for the subscription is

invoked with a status of STALE.

5. The message is passed up to the application with a status of STALE.

6. If any updates are received prior to the recap being received, the `onGap()` callback is invoked and the message is passed to the application with a status of STALE. This does not result in `onQuality()` being invoked, as this gap does not result in a quality state change, or a recap being requested, as the API has already sent a request and is waiting for a response.

7. The requested recap arrives.

8. The `onQuality()` callback is invoked with a status of OK.

9. The message is passed to the application.

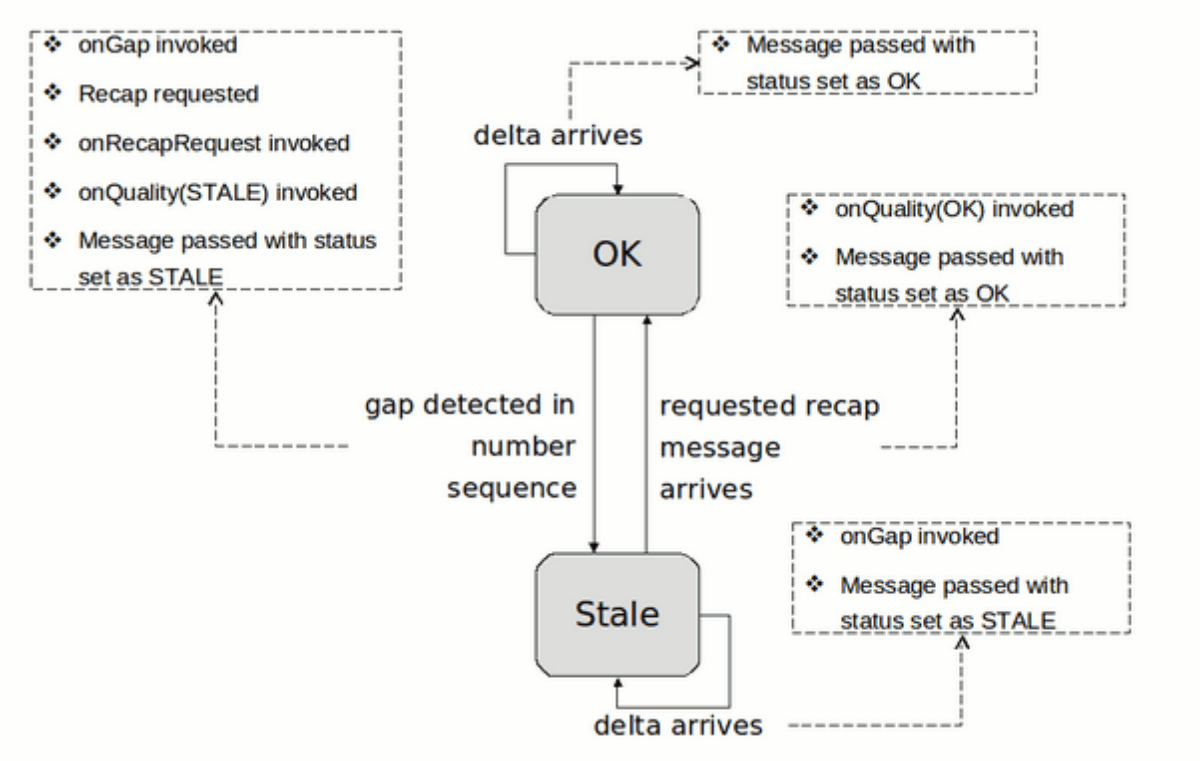


Figure 6: Data Quality 1

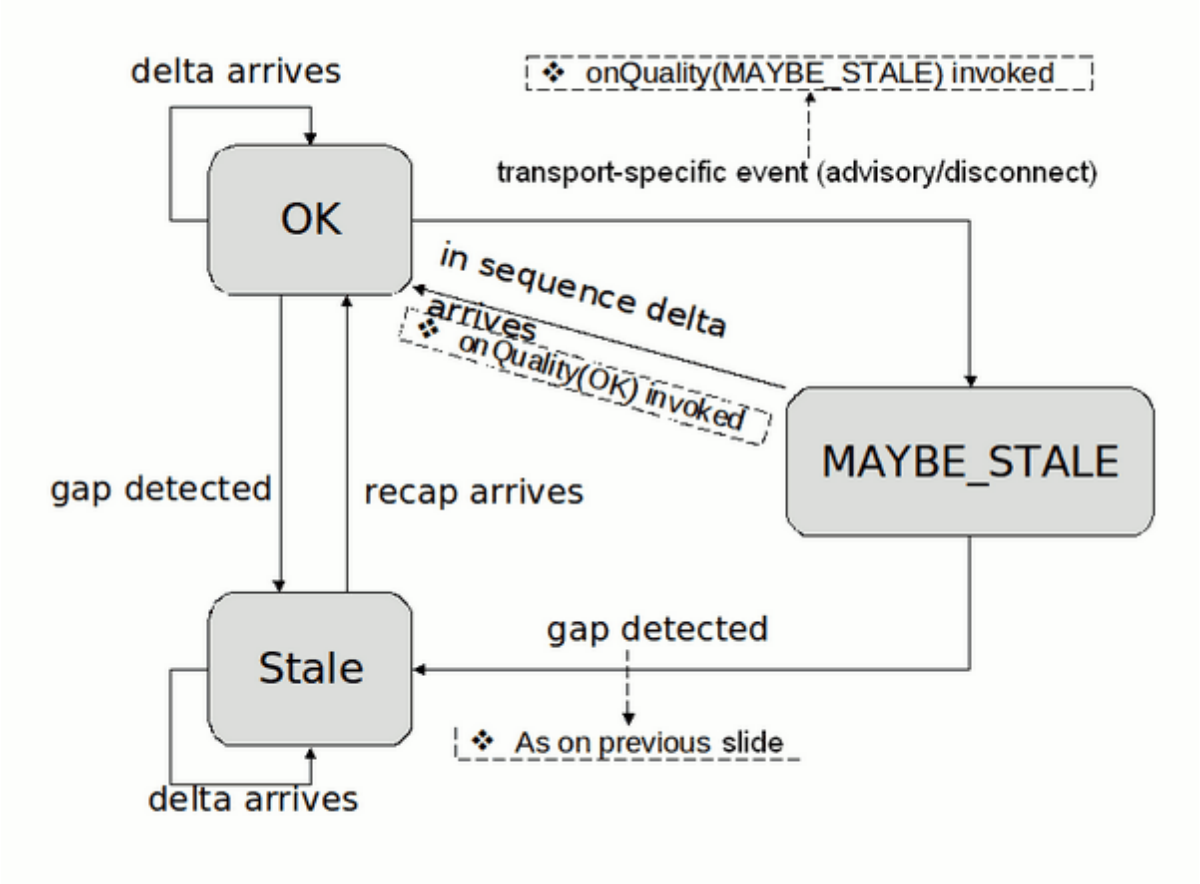


Figure 7: Data Quality 2

Sequence number checking can be disabled in the API if required. Calling `MamaSubscription.setRecoverGaps()` informs the API to no longer check the injected sequence number for gaps.

Sequence number gaps can be indicative of a number of possible problems:

- Data loss on the inbound OS socket buffers (client machine - multicast).

  This can result from the client application monopolizing the CPU for a prolonged period of time thereby being unable to consume data from the inbound socket buffer in a timely fashion.

  Increasing the OS inbound socket buffer size can help to address this problem. The increased buffer size helps deal with short lived events that consume all CPU cycles. However, if the client continues to consume all CPU cycles, increasing the socket buffer sizes only delays the inevitable.

  This problem typically occurs with multicast data:
  - For TIBCO Rendezvous based clients, it is the CPU of the Rendezvous Daemon (RVD) that is of concern.
  - For 29West LBM based clients, it is the CPU of the application using the 29West LBM libraries that is of concern.

- Data loss on the outbound socket buffers (sending machine - TCP).

  This can result from a client application being unable to consume inbound data as quickly as it is being sent. The likely cause of this is the client application using all CPU cycles for a period of time.

Unlike multicast, which sends as fast as possible regardless of the ability of the client to keep up, TCP enforces flow control if a client cannot process messages in a timely fashion and buffers data until such time as the client can process again. If this condition persists the sender buffers will ultimately overflow and lose data.

This occurs with 29west LBM TCP and communication between an RVD and the TIBCO Rendezvous API.

The ability for a client application to consume data from an incoming socket buffer can be impacted by a number of sources of resource contention. For example:
- Running out of physical memory resulting in significant paging.
- High levels of context switching on an overloaded machine - e.g. very high load average on Linux.
- Significant IO blocking.
- Lack of available CPU cycles.

- Data loss - network.

  Various network conditions can result in lost data. Such as:
  - unreliable cabling
  - mis-configured switches
  - mis-configured NIC's
  - router buffer overflow
  - congested network

The ***OpenMAMA API*** also attempts to monitor the health of the underlying messaging infrastructure. Upon detecting a problem like a failed transport, an overflowing queue or a slow consumer, the API marks all inbound messages as 'possibly stale' (MAMA_QUALITY_MAYBE_STALE) while the condition persists. ***OpenMAMA*** continues to mark messages as STATUS_STALE while the condition persists and does not request a recap for the subscriptions until the problem subsides.

Note Messaging infrastructure level checking is only currently available on the TIBCO Rendezvous platform

Note The behavior of data quality in ***OpenMAMA*** does not vary with middleware.

12.1 Data Quality for Group Subscriptions

Items within a group subscription have their own individually tracked sequence number. As such, data quality for groups is tracked on a per item basis. On detecting a gap for an item within a group subscription a recap is requested for that item only.

12.2 Data Quality and Fault Tolerant Takeovers

The **MAMA Advanced Publishers** can be configured to run as primary/secondary pairs in hot/hot fault tolerant mode. Any any point in time only the primary feed is publishing data. When a fault tolerant event occurs, a secondary publisher assumes the role of primary and all instruments that have ticked during a configurable period longer than the fault tolerant takeover interval are recapped by the new primary. It is then assumed that data for all remaining instruments are in sync with the last message sent by the previous primary.

The ***OpenMAMA API*** tracks the senderId, a 64 bit field that uniquely identifies a publisher on the platform (MamaReservedFieldSenderId, FID 20), if present. A change in the sender Id is taken to represent a fault tolerant event at the publisher level. The current behaviour is to assume that an update with a different senderId is the next expected update, and to reset the internal data quality state with the senderId and sequence number of the message, marking the subscription quality as OK if required. Unsolicited recaps will have been sent from the new primary feed on any symbols for which all updates may not have been received.

13 Publishing

Although the ***OpenMAMA API*** is primarily used within subscribing applications, it can also be used for publishing. Data can be published on a particular symbol/topic, in either a request response paradigm or the normal publish-subscribe environment. Two types of publishing are available: basic and advanced.

Basic publishing is the counterpart to basic subscriptions. Basic subscriptions do not require initials, seqnums, refreshes and so on, and basic publishing does not cater for these either. It allows the sending of simple messages of any fields with or without field names or FIDs. Basic publishing is recommended in a generic messaging environment or for admin and control messages between market data applications.

Advanced publishing is a much stricter framework that is designed to provide the necessary tools for building a full market data publishing source for fully interacting with ***OpenMAMA*** clients. The advanced publishing classes build on the basic publisher but still use the same underlying concepts.

13.1 Basic Publishing

Data is sent from a `MamaPublisher` object. On creation of a publisher, a transport and an outbound topic are specified. The outbound topic ("MY_PUB_TOPIC") is the symbol or topic for which a subscribing application must create a basic subscription for.

Example 35: MamaPublisher

```
MamaPublisher publisher = new MamaPublisher;
publisher->create (transport, "MY_PUB_TOPIC");
```

Sending a message from a publisher is straightforward: use the `send()` method, passing in the MamaMsg object to be sent. A message published in this fashion is sent from ***OpenMAMA*** immediately. However, depending on the middleware configuration, there may be a short delay before its actually sent. The message is normally sent from the thread calling `send()`, but there are some exceptions to this depending on middleware configuration.

Example 36: send

```
publisher->send (msg);
```

If there is a need to control the publish rate of messages, use `sendWithThrottle()`. This places the message or action on the internal throttle queue to be sent at some time in the future. The throttle rate of the transport, on which the publisher was created, is used. The default rate is 500 per second. When the message is sent, a complete callback will be called which is passed into the `sendWithThrottle()` method. It is the responsibility of the application to manage the life cycle of messages sent in this fashion. Throttling is enabled if the throttle value is greater than "0". Throttling occurs on the default threads and the request will be sent from there. Setting the throttle rate to "0" will disable throttling and the request will be sent from the thread creating the subscription.

Example 37: sendWithThrottle

```
class SendComplete : public MamaSendCompleteCallback
{
  public:
    void onSendComplete (MamaPublisher& publisher,
                         MamaMsg* msg,
                         MamaStatus& status,
```

```
                        void* closure)
  {
    //Check the status of the call.
    //Destroy the message or reuse.
  }
}
MamaSendCompleteCallback* cb = new MamaSendCompleteCallback;
//To throttle sending of message
publisher->sendWithThrottle (msg, cb, closure);
```

13.1.1 Request/Response

Request/response communication involves a requester of data issuing a request on a topic to responder (s) of data for that topic on a particular transport. A requester can receive multiple responses to a request. The response is always sent over unicast. Request/response communication in ***OpenMAMA*** is achieved through the use of `MamaPublisher` and `MamaInbox` objects.

To make a request in ***OpenMAMA***, a message is sent from a publisher and is associated with an instance of an inbox. The inbox is the destination recipient for any responses to the issued request. A callback is registered with the inbox upon creation and is invoked whenever any responses to the request are received by the API. Inbox requests can be throttled in the same way as non point-to-point messages and are affected by the same threading conditions.

Request messages arrive to the `onMsg()` callback for basic subscriptions on particular topics. A message is identified as being a request via a call to the `MamaMsg.isFromInbox()` method. The `MamaPublisher.sendReplyToInbox()` method is used to actually send the response message to the request. Both the request message and the new response message are passed to the method when it is invoked.

13.2 Advanced Publishing

For advanced, or data quality publishing, the DQPublisherManager is the central class for advanced publishing. It is responsible for the namespace subscription, listening for subscription requests, and handles the DQPublishers, created to respond to those requests. There are also mechanisms for handling refresh messages and synch requests.

Creating a publisher manager requires a transport, a queue, a symbol namespace, the necessary callbacks and an optional root. The root is an identifier specific to the platform. The default is _MD, which is what the client applications subscribe to. For publishing a dictionary the default is _MDDD.

Example 38: DQPublisherManager

```
mamaDQPublisherManager_allocate(&pubManager );
mamaDQPublisherManager_create (pubManager,
                               transport,
                               queue,
                               callbacks,
                               sourcename,
                               root,
                               NULL);
```

Upon creation, the publisher manager creates a subscription to listen for requests on the given namespace. When a new request is received by the publisher manager it calls an onNewRequest callback. The callback supplies the symbol from the request as well as the type and request type. The message, containing the IP address of the requesting client machine, is also supplied.

Example 39: onNewRequestCb

```
onNewRequestCb (mamaDQPublisherManager pubManager,
                const char* symbol,
                short subType,
                short msgType,
                mamaMsg msg);
```

When the request has been received, and if the publishing application can supply data for the requested symbol, then the publishing application should create a publisher and add it to the manager.

Example 40: mamaDQPublisherManager_createPublisher

```
mamaDQPublisherManager_createPublisher (pubManager, symbol, closureData, &symbolPub);
```

If the symbol should not be published, then the request should be ignored and the subscription will timeout on the client side.

A caching structure may also be added at this point. Neither the publisher or the publishing manager interact with the caching structure directly, and it is up to the client application to use a structure which best suits that application. The example applications can use either a simple message or a full field cache.

When creating the publisher, it can be configured in a number of ways. Setting the initial seqnum to "0" means that seqnum is not sent and this effectively turns off recapping. Initials and updates are handled as normal, but if a gap is detected the client will be unable to recover. By default a seqnum is added to every message. Sender ID is also added to every message, unless it is set to "0" explicitly. The msgstatus, also present in every message, is set to whatever the status on the publisher is at the time of send. Default status is MAMA_STATUS_OK, but this can be changed to STALE or MAYBE_STALE as required.

Example 41: Configuring the publisher

```
mamaDQPublisherManager_setStatus (pubManager, status);
mamaDQPublisherManager_setSenderId (pubManager,senderid);
mamaDQPublisherManager_setSeqNum (pubManager, num);
```

The first message published in response to a new request should be of type INITIAL or RECAP. This is a full image of the caching structure and is sent using the send reply method.

Example 42: Obtaining the first message

```
mamaMsg_updateU8(initialMsg, NULL, MamaFieldMsgType.mFid, MAMA_MSG_TYPE_INITIAL);
mamaDQPublisher_sendReply(symbolPub, msg, initialMsg);
```

Once the first message is sent, the client application listens for updates published from that publisher. The recommended approach is for updates to be applied to the cache then the delta to be sent.

When a request for a symbol that is already being published and is familiar to the publisher manager is made, the OnRequest callback is called. As well as the parameters in the OnNewRequest callback, you are also given pointers to the publisher currently publishing this symbol and the closure structure for that symbol.

Example 43: onRequestCb

```
onRequestCb (mamaDQPublisherManager pubManager,
             mamaPublishTopic* publishTopicInfo,
             short subType,
             short msgType, mamaMsg msg)
```

If the publishing application wishes to publish to that client, then it must first send an initial. However, as the cache is being used to publish updates, it is important to make sure that you get a full image in a thread-safe way. The difference between OnNewRequest and OnRequest is that OnRequest has a publisher created and data, whereas OnNewRequest does not, so your data initialization should occur in OnNewRequest. Once the initial is published, the client processes the updates as normal.

Recap requests are passed up in the same way as requests for symbols already being published, through the OnRequest callback. This is because they are handled in the same way. The only difference between the two is that you set the type on the image message published as RECAP rather than INITIAL. You may also want to send the recap as multicast rather than as a reply.

13.2.1 Stop publishing (Refresh Mechanism)

To be in accordance with the normal publishing rules within the platform, publishers should cease to publish one hour after the last request or refresh. Refresh messages are sent by client applications approximately every 55 minutes, and the onRefresh callback will be fired.

Example 44: onRefreshCb

```
onRefreshCb (mamaDQPublisherManager pubManager,
             mamaPublishTopic* publishTopicInfo,
             short subType,
             short msgType,
             mamaMsg msg)
```

There is an acknowledgement sent to all clients for that symbol that refreshing has occurred, which is handled internally. The publishing application should record when the last refresh or request for a symbol was received, and if it was more than one hour ago it may stop publishing that symbol and remove it from the publisher manager.

If the symbol is requested again, an OnNewRequest callback is generated.

If a refresh is received for a symbol that is not currently being published, a OnNewRequest callback is received rather than an OnRefresh. This is useful when bringing up a secondary source while not putting extra strain on the client applications to respond to synchronization request messages.

Synchronization requests can be sent by the publisher manager during start-up to determine which clients are currently subscribed to it. This is generally done when a source has been restarted. A request is sent by the MamaDQPublisher Manager and all clients subscribed respond with a list of the symbols they are interested in. The synchronization request message contains some values to help separate the reply messages if there are a lot of subscriptions across multiple clients.

The reply is parsed by the publisher manager and will generate a number of OnNewRequest and OnRequest callbacks.

It is important that synchronization requests are only sent when the publisher is in a position to handle all the requests generated, and that they are sent after the transport is fully connected. For example, on a TCP connection you would wait for the transport connected callback.

The ***OpenMAMA API*** provides a fault-tolerant module which can be used to heartbeat between two applications and tell which is primary and which is secondary (see *Section 12.2: Data quality and fault tolerant takeovers*). It is important that when attempting a fault-tolerant takeover that at a minimum you recap everything that has ticked since the last heartbeat. This is due to the fact that ***OpenMAMA*** clients will not request a recap when there is a change in seqnum that coincides with a change in sender id. It is the responsibility of the publisher to maintain data quality through a takeover event.

14 Value Add

14.1 Timers

OpenMAMA supports user events triggered by recurring timers. These are represented by the `MamaTimer` object. An event queue must be specified when a timer is created.

A timer is created with a specified interval in seconds, using a double-precision floating point number to give the resolution in fractions of a second if required. The timer callback is invoked repeatedly at an interval no shorter than that specified. A number of factors can cause the timer interval to be inexact or the callback not to be immediately invoked on the firing of the timer.

The timer itself can be inaccurate due to the frequency of the OS interrupts. For instance, on a 100 Hz OS (Linux 2.4), if a 10 milliseconds timer is created one millisecond after the last interrupt fired it will take 19 milliseconds for the timer to fire. This is because only 9 milliseconds will have elapsed when the next scheduled interrupt occurs. Essentially, the best timer resolution possible here is 10 milliseconds and worst 19.9999 milliseconds.

The precision of timers is determined by the implementation in the underlying messaging platform and the interrupt frequency of the operating system.

For example, the Linux 2.4 kernel with an interrupt frequency of 100 Hz is capable of 10 milliseconds resolution at best. A Linux 2.6 (up to 2.6.12) kernel with a 1000 Hz interrupt frequency can provide timers with a best resolution of 1 millisecond. (The default interrupt frequency on Linux 2.6.13 and up is 250 Hz but is now configurable on i386 architectures).

The invoking of a callback in response to a timer firing can depend on the activity on the event queue which was specified on timer creation.

A timer can be destroyed from within a timer callback or any other callback on the queue. This function must be called from the same thread dispatching on the associated event queue unless both the default queue and dispatch queue are not actively dispatching.

14.1.1 Creating a Timer

The following example show the creation of a recurring timer which fires every 500 milliseconds. In each case the actionCallback function/object receives a callback once the interval has elapsed.

Example 45: Creating a timer

```
class TimerCallback : public MamaTimerCallback
{
public:
     TimerCallback () {}
     virtual ~TimerCallback () {}
     virtual void onTimer (MamaTimer* timer)
     {
       MyClosureType* myClosure = dynamic_cast<MyClosureType*>(timer->getClosure ()
       //Perform recurring task.
     }
}
....
MamaTimer*         timer             = new MamaTimer;
TimerCallback*     timerCallback     = new TimerCallback;
```

```
timer->create (queue,
               timerCallback,
               0.5,
               closure);
```

14.2 IO

The ***OpenMAMA API*** provides an abstract mechanism by which a client registers interest for various events on file descriptors. A callback, provided to the API, is invoked whenever an event, of the type specified when registering interest, occurs. The Mamalo API facilitates asynchronous IO operations.

The following event types are supported in the API:

- MAMA_IO_READ
- MAMA_IO_WRITE
- MAMA_IO_CONNECT
- MAMA_IO_ACCEPT
- MAMA_IO_CLOSE
- MAMA_IO_ERROR
- MAMA_IO_EXCEPT

14.2.1 Registering for IO Events

Not all underlying messaging middlewares support all of the event types provided by the ***OpenMAMA API***. In the case of a particular event type not being supported, the call to `mamaIo_create()` fails with a return code of MAMA_STATUS_UNSUPPORTED_IO_TYPE.

Example 46: Registering for IO events

```
class IoCallback : public MamaIoCallback
{
    virtual void onIo (MamaIo* io, mamaIoType ioType)
    {
        //The inboundFileDescriptor is now available for reading.
    }
}

MamaIo     mamaIo               = new MamaIo;
IoCallback ioCallback           = new IoCallback;
int        inboundFileDescriptor = 0;//File descriptor for socket we want to read from
....
    //Get the file descriptor for an inbound accept on a socket
....

mamaIo->create (queue,
                ioCallback,
                inboundFileDescriptor,
                MAMA_IO_READ,
                "My Closure");
```

14.3 User Events

User events can be added to ***OpenMAMA*** queues irrespective of the middleware using `MamaQueue.enqueueEvent()`. This allows user code to be executed on any of the dispatching threads.

The `MamaQueueEventCallback.onEventd()` method will be invoked whenever the event fires. The callback event is added to the back of the queue, so if there are a large number of events on the queue it may be some time before the callback is invoked.

14.4 Logging

To aid with debugging, the ***OpenMAMA API*** provides various levels of verbose logging to give developers more detail during event processing within the API.By default, logging is disabled within the API and needs to be manually enabled along with specifying the level at which to log information.

14.4.1 Logging Via Properties

There are several values that can be set in `mama.properties` to control logging. They are listed in **Table 14: Logging Properties**. Logs can be set to roll over. When set to roll over, the following happens:

1. The log file is allowed to grow until it reaches the maximum size.
2. When it reaches the maximum size, the logfile has .1 appended to its name and a new log file is begun.
3. If a .1 file already exists, this older file is renamed to have a .2 instead of .1, and so on.
4. If there is a maximum number of log files specified, then the total number of log files will never be more than that. The oldest file will then be deleted when the log file rolls over

If using an UNBOUNDED policy, the behaviour is undefined if the log file reaches the maximum size allowed by the file system.

Note: MamaLogFile in C++ is now deprecated.

Table 14: Logging Properties

Property	Description
mama.logging.file.name	The filename for the log file.
mama.logging.level	Sets the log level. Possible values are: ▪ off ▪ severe ▪ error ▪ warn ▪ normal ▪ fine ▪ finest
mama.logging.file.policy	Sets the policy used when logging to a file. Possible values are: ▪ UNBOUNDED (default) - the log file will grow indefinitely ▪ ROLL - the log file will roll over when it reaches a certain specified size ▪ OVERWRITE - when a specified file size is reached, writing continues at the start of the file ▪ USER - when the maximum log file size is reached, the onLogSizeExceeded callback is triggered (see setLogSizeCb (callback) in the next table).
mama.logging.file. maxsize	Sets the maximum size of the logfile, in bytes. This applies to the ROLL, OVERWRITE and USER policies only. The default is 500MB.
mama.logging.file. maxroll	Sets the maximum number of log files to keep.
mama.logging.file.append	When set to "true", an existing log file will be appended to when

Property	Description
	logging is enabled. When set to "false", it will be overwritten.

There are several methods used to control logging. They are listed in the following table.

Table 15: Logging Methods

Method	Description
`logToFile(filename, level)`	Enables logging to `filename` at log level "level". See *`mama.logging.level`* in the table above for values of level.
`disableLogging()`	Disables logging, disables the log size exceeded callback, and sets the log level to "OFF".
`setLogLevel(level)`	Sets the log level to "level". See *`mama.logging.level`* in the table above for values.
`getLogLevel()`	Returns the current log level.
`setLogFilePolicy(policy)`	Sets the log file policy. See *`mama.logging.file.policy`* in the table above for values.
`setLogSize(size)`	Sets the maximum log file size, in bytes. Applies when the policy is ROLL, OVERWRITE or USER.
`setNumLogFiles(numFiles, level)`	Sets the maximum number of log files. Applies when the policy is ROLL only.
`setAppendToLogFile(bool)`	When true, an existing log file will be appended to when logging is enabled. When false, it will be overwritten.
`loggingToFile(void)`	Returns true if logging to a file.
`setLogSizeCb(callback)`	Sets the onLogSizeExceeded callback which is called when the max file size is reached. Applies only to USER policy. Not available in Java

14.5 Conflation

14.5.1 Conflation from the OpenMAMA Client Perspective

Conflation is the process employed by an advanced publisher to merge messages on the write queue to reduce memory use, to reduce bandwidth, and to deal with slow consumers. Conflation is available for the ***NYSE Technologies Data Fabric***.

The advanced publisher installs and un-installs a conflater for individual clients as and when they require conflation. There are two circumstances that require conflation:

1. If the number of messages increases above a high water mark (the advanced publisher write queue is becoming too large).
2. The client requests that conflation is turned on. A client may monitor its own queue sizes to determine when it requires conflated data.

From a ***OpenMAMA*** client perspective, the most important aspect is the method by which a client requests conflation. There is a ***OpenMAMA*** function available for this.

Example 47: Requesting conflation

```
transport.requestConflation();
```

There are two reasons why the conflater may no longer be needed:

1. When the number of messages drops below a low water mark.

2. When the client requests that conflation is no longer needed.

There is an ***OpenMAMA*** function available to request the end of conflation.

Example 48: Ending conflation

```
transport.requestEndConflation();
```

Although clients can request conflation or request that the advanced publisher stops conflating messages at any time, the advanced publisher may not be able to honor the request. For example, some advanced publishers may not be configured for conflation while others may be unable to end conflation at a client's request because the write queues are too large.

When a client receives a conflated message it contains three additional fields: wConflateCount, wConflateTradeCount, and wConflateQuoteCount. These represent the total number of messages conflated into a single message, the number of trades and the number of quotes respectively. wConflateCount == wConflateTradeCount + wConflateQuoteCount. ***OpenMAMA*** and ***OpenMAMDA*** use these fields internally to perform sequence number checking and maintain data quality.

14.6 Statistics

OpenMAMA provides statistics to monitor the ***OpenMAMA*** client. When enabled, ***OpenMAMA*** will log, via ***OpenMAMA*** logging, and/or publish details of the numbers of subscription requests received, number of initial messages received, queue size and various other statistics. 29West LBM also provides various low-level statistics on a per context basis which are exposed via this functionality. Statistics are generated and logged and/or published at a configurable interval. When ***OpenMAMA*** statistics are logged, the value of the statistic for the interval is logged, along with the maximum and running total for that statistic since statistics logging was enabled. When ***OpenMAMA*** statistics are published, the interval value for each statistic is published along with a timestamp indicating the time the message was generated and information identifying the client such as the IP Address, user name, and ***OpenMAMA*** application name. When used in conjunction with the Stats Logger, customers can produce DSV files containing client monitoring information.

Note that the size of the published message can be large. When publishing over 29West LBM, it is likely that such a message will be too large to publish using immediate messaging. Depending on the 29West LBM version being used, this will result in either the message not being published at all, or the message being published in a truncated form and thus missing some or all of the stats vector. For this reason, we recommend disabling the use of immediate messaging for publishing over 29West LBM using the property mama.lbm.transport.TRANSPORT_NAME.use_im_for_publishing= false.

Statistics are provided at various levels: globally (all transports and queues across the application), per-transport, and per-queue. The level at which to generate statistics is configurable, and any combination of global, transport and queue statistics can be used. 29West LBM statistics are enabled separately but are considered transport-level statistics.

Note that when logged to the ***OpenMAMA*** log, only statistics with a value greater than "0" will appear in the log.

Table 16: General OpenMAMA Statistics

Statistic	Description
Initials	Number of initial messages received.
Recaps	Number of recaps received.
Messages	Total number of messages received. This includes initial and recap messages.
FT Takeovers	Number of fault tolerant takeovers. Note that this is only provided globally and per transport, never per queue.
Queue Size	Size of the queue at the time the stats message was generated (i.e., at the end of the interval). Since this is a snapshot of the queue size rather than a cumulative value like the other statistics, no total value is provided when logging via ***OpenMAMA*** logging. This is only ever provided on a per queue basis, never globally or per transport. Note that, when using LBM, the property `mama.lbm.eventqueuemonitor.queue_size_warning` must be set for correct values to be returned.
Subscriptions	Number of subscriptions created. This includes all subscription types, including the dictionary subscription.
Timeouts	Number of subscription timeouts.

14.6.1 Statistics Logging Configuration

The following parameters controls statistics logging. They can be set in `mama.properties`.

Table 17: Statistics Logging Configuration Parameters

Parameter	Value	Description	Default
mama.statslogging.enable	yes or no	Enable/disable stats logging. Whether logging, publishing, or both, this must be set to "yes" for any stats logging functionality to be used.	
mama.statslogging.middleware	lbm, wmw or tibrv	The middleware to use for publishing stats messages. This only needs to be set if at least one of the publishing parameters is enabled.	wmw
mama.statslogging.interval	integer	The interval, in seconds, between published stats logging reports.	60
mama.statslogging.transport	transport name	The ***OpenMAMA*** transport used for publishing the stats reports.	statslogger
mama.statslogging.global.logging	yes or no	Whether or not to log global stats to the ***OpenMAMA*** log.	yes
mama.statslogging.global.publishing	yes or no	Whether or not to publish report messages for global stats.	no
mama.statslogging.transport.logging	yes or no	Whether or not to log transport stats (not including middleware-specific stats) to the ***OpenMAMA*** log.	yes
mama.statslogging.transport.publishing	yes or no	Whether or not to publish report messages for transport stats.	no
mama.statslogging.queue.logging	yes or no	Whether or not to log queue stats to the ***OpenMAMA*** log.	yes
mama.statslogging.queue.publishing	yes or no	Whether or not to publish report messages for queue stats.	no

14.6.2 Published Message Format

Stats reports can be published by enabling one or more of the publishing parameters on the middleware, specified by *mama.statslogging.middleware*, and transport, specified by *mama.statslogging.transport*. These can be published to the Stats Logger product to automatically generate DSV files containing the stats.

These stats reports are simply ***OpenMAMA*** messages published on the topic STATS_TOPIC, and so it is possible to write a ***OpenMAMA*** application to subscribe to stats messages and manually process them. The format of the published messages is described in the following table.

Table 18: Message Format

Field Name	FID	Type	Description
InterfaceVersion	69	U8	Stats Logger interface version
MdSubscSourceUser	65	STRING	The username of the user running the MAMA client publishing the stats messages.
MdSubscSourceHost	63	STRING	Hostname running the client which is publishing the stats messages.
MdSubscSourceApp	64	STRING	MAMA application name of the client publishing the stats message. Defaults to MamaApplication if not set.
MdSubscSourceAppClass	68	STRING	Application class of the MAMA client publishing the stats message. Defaults to MamamApplications if not set.
MdSubscSourceIp	67	STRING	IP Address of the MAMA client publishing the stats message.
MamaUlIntervalStartTime	101	TIME	The start time of the interval in which this set of stats were monitored.
MamaUlIntervalEndTime	102	TIME	The end time of the interval in which this set of stats were monitored.
MamaStatEvents	103	VECTOR_MSG	A vector message containing the actual statistics. Its contents depend on which stats publishing parameters are enabled. If global stats publishing is enabled, it will include a single message containing the global statistics, as well as any other messages. If transport stats are enabled, it will include a message per transport, with each message including statistics for a single transport, as well as any other messages. If queue stats are enabled, it will include a message per queue, with each message including statistics for a single queue, as well as any other messages. The format of these messages is included below.

The following table describes the messages included in MamaStatEvents. All the fields described in the table are available from the MamaStatFields header. Names and FIDs for individual fields can my accessed using FIELD.mName and FIELD.mFid.

Table 19: Sub Message Format

Field Name	FID	Type	Description
Time	101	TIME	Time the stats message was added to the array.
Name	102	STRING	Name of the statistic.
Type	103	STRING	Type of object the statistcs were measured for. Will be either "Transport", "Queue", or "Global".
Middleware	104	STRING	Middleware of the object the statistics were monitored for. Except in the case of global stats, this will be the name of the middleware. Global stats will always log "----" as multiple middlewares may be being used across the application.
Initials	104	U32	See *General OpenMAMA Statistics*.
Recaps	106	U32	
Messages	107	U32	
FT Takeovers	108	U32	See *General OpenMAMA Statistics*. Will never be present if the Type field is "Queue".
Queue Size	109	U32	See *General OpenMAMA Statistics*. Will only ever be present if the Type field is "Queue".
Subscriptions	110	U32	See *General OpenMAMA Statistics*.
Timeouts	111	U32	

15 Example Programs

This section gives a brief explanation of what each of the ***OpenMAMA*** example programs is designed to show. Each example is provided as a prebuilt binary and as source code within the release. Each example accepts a list of command line options. These can be examined by passing `-h` on the command line, or viewing the source code file.

Note The examples should be available across all languages, however, some may have a slight name change, such as mamalistenc, mamalistencpp, mamalistencs, and mamalistenjava. The operation and functionality across the languages should be consistent.

Table 20: Example Programs

Example Program	Description
MamaListen	This is a simple ***OpenMAMA*** application that creates a configurable number of subscriptions to a single source on a single transport. Received messages for these subscriptions are printed to screen with name, fid, type, and value. This application shows the basic operation of a market data application.
MamaPublisher	This is a simple publishing application that uses basic publishing to send messages with a few fields on a well-known topic. Its purpose is to show the use of basic publishing for non-market data.
MamaSubscriber	This application uses basic subscriptions to listen for a basic publisher. It works in conjunction with MamaPublisher. This application shows the other side of non-market data communication.
MamaInbox	This application sends an inbox request to a source and waits for a reply. When used in conjunction with MamaPublisher, the MamaPublisher will listen for the request and respond with a simple message. This illustrates the request/reply mechanism as used with both market data and non-market data situations.
MamaIO	This application shows how to use ***OpenMAMA*** to monitor a file descriptor for input.
MamaMultiSubscriber	This application demonstrates how to use multiple bridges within a single application to receive data from two middlewares. The received messages are processed and displayed in the same manner.
MamaProxy	This application is similar to MamaListenCached, with the added functionality that the messages are republished using the market data publishing component (DQPublisher). This allows a further MamaListen client to receive the data via this path, rather than directly from the source.
MamaSymbolListSubscriber	This application uses a symbol list subscription to get a complete list of all symbols available from the source, and then makes market data subscriptions to these symbols, illustrating how to listen to the "world" in topic terms.
MamaFtMember	This application demonstrates use of ***OpenMAMA*** fault tolerance capability. Each instance of MamaFtMember can be assigned to a group. Each instance within the group has a fault tolerance weight. Whenever all members in a group are active, the highest weighted member will report its status as ACTIVE, the others will be STANDBY. If the highest weighted member is killed the next highest weighted member will become ACTIVE.

16 Performance Programming

OpenMAMA and ***OpenMAMDA*** are commonly used within applications where performance is important. ***OpenMAMA*** itself is very lightweight and an application that does significant work will use much more CPU cycles that the API itself. This section describes a number of techniques and tips that can be used to optimize the use of ***OpenMAMA***, ***OpenMAMDA*** and applications built on top of them.

16.1 Monitoring Performance

There are a number of ways to monitor the performance of a ***OpenMAMA*** or ***OpenMAMDA*** application:

- A good indicator of performance is the queue depth: if the queue is growing then it is a sign that the application is not able to keep up with the amount of incoming data. This adds latency. From within the API, either the depth of queues can be queried or watermarks can be set for a particular queue depth value.
- MamaStats provides a lot of useful information including the queue depth, message rates, number of recaps as well as middleware-specific statistics.
- Recap requests, shown via the MamaStats output or thought the onRecapRequest() callback, are also an indicator that the application is not able to keep up with the data. Recaps are not a good indicator of performance because, by the time a recap is requested, permanent data loss has already occurred.
- The application's CPU use should be monitored. If any of the cores used by the application approach or reach 100% use then issues such as queue growth and data loss will occur.

16.2 Storing Per-Symbol State

It is usual to need to store application state on a per symbol basis e.g. calculated values, pointer/ references to other application objects etc.

Note It is very inefficient to do this by obtaining the symbol from each received message and then using a map to retrieve the state for that symbol.

OpenMAMA and ***OpenMAMDA*** provide an alternative mechanism for storing state though closures. A closure is a user-defined void pointer and is associated with a particular symbol.

Closures can be set on a subscription, created and then retrieved using the accessors on the subscription objects. Single and group subscriptions can be used to receive data from many different symbols, therefore a separate “item closure” is available for the group members. Calling `setItemClosure()` in a group subscription callback will set the closure for that group member.

Another method available is to store the per-symbol state within the objects which implement the subscription callback interfaces e.g. the abstract base class `MamaSubscriptionCallback` in C++. To do this, a separate instance of the callback object should be created for each subscription and its state should be stored in its member variables.

16.3 Message Access

If accessing all, or most, of the data within a message, then iteration is generally faster than direct field access. There are two methods: user driven and callback driven. User driven has been shown to be faster.

If accessing only a few fields within a message, then direct access to those particular fields will generally be faster. The decision of which method to choose will depend on two factors: the particular application use case and the deployment, which will determine the number of fields and where those fields are within a message, so experimentation is encouraged.

16.4 Memory Allocation

Memory allocation and deallocation are major sources of performance issues. It is best to allocate all required memory at start up, and avoid allocating memory while processing data, particularly in on a per-message basis. There are various ways to do this, which include: reusing the same specific objects from one callback to another; or, if the required lifetime of data stored within an object is longer than the scope of a callback, then a pool of reusable objects can be used.

16.5 Threading

The recommended way to scale a ***OpenMAMA*** or ***OpenMAMDA*** application across using multiple CPU cores is to use separate threads on mamaQueues, with one thread/queue per CPU core allocated to ***OpenMAMA***. For example, on an eight core machine with a non-MAMA application having two “hot” threads, the ***OpenMAMA*** application having two “hot” threads but not directly using ***OpenMAMA***, then three mamaQueues should be used. The definition of a “hot” thread is one that continuously uses a significant amount of CPU. The eighth core in the example above is left unassigned because it is recommended to reserve a core for the operating system and other threads that don’t use a significant amount of CPU.

Again, different applications will have behave differently so some experimentation may be required to find the optimal number of queues. The example applications supplied with ***OpenMAMA*** demonstrate an easy way to do this how by having the number of queues configurable number of queues at runtime using the MamaQueueGroup object.

An alternative method is to use a separate thread to do the message processing instead of processing the data in the callbacks. This is not generally recommended as it just shifts the bottleneck from one thread to another, and causes additional overhead by having to copy the message or fields from the message and place onto another queue.

17 Running Multiple Instances of OpenMAMA

Only one instance of ***OpenMAMA*** is supported for each process. However, multiple ***OpenMAMA*** processes can be run successfully.

By default, when the `mama_open` function is called, the `mama.properties` file is obtained from the location specified by the `WOMBAT_PATH` environment variable. This results in all ***OpenMAMA*** processes being configured using the same file. See *Section 4: Properties* for further details.

17.1 Running with a Single Properties File

This is the default approach taken when running multiple instances of the ***OpenMAMA*** example programs, such as **mamalistenc**.

OpenMAMA still runs, but with the following restrictions:

1. Only a single log file can be specified.

   The `mama.logging.file.name` entry in `mama.properties` can be used to specify the log file. All ***OpenMAMA*** processes attempt to open and write to the log file at the same time. To prevent this occuring:

   - Do not specify a log file. If a log file is not specified ***OpenMAMA*** writes log entries to stderr, the output can be directed on a per-process basis.
   - Change the location of the log file in code using the mama_logToFile function.
   - Install a log callback function in code to catch all logging events using the mama_setLogCallback function.

2. Publishing features may not work properly.

   OpenMAMA provides several features that create transports as part of their initialization. Transport names and settings are read from `mama.properties` when mama_open is called.

   This is particularly a problem publishing data when using a point to point middleware such as WMW TCP. It results in an error being returned from mama_open as the publishing port is blocked by a prior ***OpenMAMA*** instance.

   The following features publish data on the transport. If any of these are to be used at the same time across multiple ***OpenMAMA*** instances, then consideration should be given to maintaining multiple `mama.properties` files or configuration of the API in code.
   - Stats publishing
   - Usage logging
   - WAM

   The following features create subscriptions to request data. In these cases a multi-cast middleware could be used that allows multiple connections to be open at the same time.
   - Property server connection
   - Template server connection

17.2 Managing Multiple Properties Files

Due to the restrictions highlighted in *Section 17.1: Running with a Single Properties File,* it may be desirable to configure each process independently by supporting multiple properties files. ***OpenMAMA*** provides two methods of doing this:

- As explained in the previous section, the default behaviour of `mama_open` can be overridden using the `mama_openWithProperties()` function. This function takes a file name and location of the properties file.
- Alternatively, each process can define a different location in the `WOMBAT_PATH` environment variable.

18 Configuration Reference

OpenMAMA properties that apply to all middlewares are listed in the following table.

Table 21: OpenMAMA Properties for All Middlewares

Property	Description	Default
entitlement.servers	A comma-separated list of site server connection specifications. The format is *ip-address*:*port* or *host*:*port*. For example, entitlement.servers = host1:8095, host2:8095	
mama.%s.transport.%s.groupsizehint	This gives a hint as to the expected number of symbols within a group when using group subscriptions. Increasing this may improve performance when using larger groups	100
mama.catchcallbackexceptions.enable	Enable try catch on C++ callbacks to avoid exceptions being propagated back to C code. This is off by default as may have performance implications.	off
mama.entitlement.altuserid	Holds a user ID that OEA will pass to Site Server, in addition to the OS user ID. When processing requests from OEA, Site Server will use the alternative user ID if it is configured to use the alternative user ID. Otherwise, Site Server will use the OS user ID, even if the alternative user ID is provided by OEA. Example: mama.entitlement.altuserid=user1.	
mama.entitlement.effective_ip_address	Holds an IP address to be used for counting concurrent connections. It is also recorded as the IP address in usage logging records. Typically used when the OEA client does not reside in the subscribing application but in a OEA server process. Example: mama.entitlement.effectiveipaddress = 192.168.2.20.	
mama.entitlement.porthigh	Holds the maximum TCP/IP port on which OEA can listen for requests from Site Server. Example: mama.entitlement.porthigh = 10010.	8001
mama.entitlement.portlow	Holds the minimum TCP/IP port on which OEA can listen for requests from Site Server. Example: mama.entitlement.portlow = 10000.	8000
mama.entitlement.site	Holds the name of the site whose entitlements set is queried by Site Server when processing a request from OEA. If this is not specified Site Server uses its default site when processing requests from OEA. Example: mama.entitlement.site = BELFAST.	
mama.logging.file.append	See *Section 14.6.2: Logging Via Properties*	
mama.logging.file.maxroll		
mama.logging.file.maxsize		
mama.logging.file.name		
mama.logging.file.policy		
mama.logging.level		
mama.logging.milliseconds		
mama.maybestale.recap.timeout	Turns on an inactivity check after Tibco Rendezvous advisories. Time to wait is specified in seconds.	

Property	Description	Default
mama.multicast.transport.ft.interface	Specify the interface to use for multicast fault-tolerant setup.	
mama.multicast.transport.ft.iowindow	Specify the IO window size for the fault-tolerant communication.	
mama.multicast.transport.ft.network	Specify the multicast group.	
mama.multicast.transport.ft.service	Specify the multicast port.	
mama.multicast.transport.ft.ttl	Specify the time to live for the multicast messages.	
mama.statslogging.enable	See *Section 14.8.1: Statistics Logging Configuration*.	
mama.statslogging.global.logging		
mama.statslogging.global.publishing		
mama.statslogging.interval		
mama.statslogging.lbm.logging		
mama.statslogging.lbm.publishing		
mama.statslogging.middleware		
mama.statslogging.middleware		
mama.statslogging.middleware		
mama.statslogging.queue.logging		
mama.statslogging.queue.publishing		
mama.statslogging.transport		
mama.statslogging.transport.logging		
mama.statslogging.transport.publishing		
mama.subscription.preinitialcachesize	Controls the size of the preinitial cache. Takes an integer value.	10
mama.throttle.interval	Sets the interval of the throttle timer for subscription and recap requests	
mama.transport.%s.preinitialstrategy	Controls when the updates stored in preinitialcache are passed up. "initial" passes any cached updates immediately after initial "gap" passes the update, only if a gap was detected.	ongap
mama.wirecache.templates	Specifies the directory and file name where the templates are defined, for example, `c:\properties\wirecache templates.xml`.	

18.1 Avis Properties

Setting Avis properties

The following properties can be set for the Avis middleware via ***OpenMAMA*** properties:

URL:
Specify the URL to connect to the Avis server.

```
mama.avis.transport.<tport_name>.url=elvin://localhost
```

If not specified, the default is "`elvin://localhost`".

Transport

The initial release supports a single transport connected to a single Avis server. An attempt to create a second or subsequent transport will return an error (MAMA_STATUS_PLATFORM).

Payload Limitations

The Avis payload has a restricted set of types compared to ***OpenMAMA***. The following table shows how these are mapped in the Avis payload bridge.

Table 22: Type Mapping

OpenMAMA	Avis	Truncation
Char	String	No
Bool	I32	No
I8	I32	No
U8	I32	No
I16	I32	No
U16	I32	No
I32	I32	No
U32	I64	No
I64	I64	No
U64	I64	Yes
F32	F64	No
F64	F64	No
String	String	No
Price	F64	Precision information lost
Datetime	U64	No

Vector types are not supported in the Avis payload, and trying to add these will result in an error (MAMA_STATUS_NOT_IMPLEMENTED).

Opaque types are supported in the Avis payload, but due to a bug in the underlying Avis client library, they cannot be copied. Since ***OpenMAMA*** requires that messages can be copied you should not use the Opaque type.

Fault Tolerance

Native fault tolerance is not supported using the Avis Bridge.

Entitlements

Entitlements are not currently supported using subscriptions on the Avis middleware.

18.2 OpenMAMA Status Codes

This section lists the status and error codes. These are defined in `mama/status.h`.

Table 23: OpenMAMA Status Codes

Numeric Value	Enum Name	Description
0	MAMA_STATUS_OK	Everything okay
1	MAMA_STATUS_NOMEM	No memory
2	MAMA_STATUS_PLATFORM	Messaging platform specific error
3	MAMA_STATUS_SYSTEM_ERROR	General system error
4	MAMA_STATUS_INVALID_ARG	Invalid argument
5	MAMA_STATUS_NULL_ARG	Null argument
6	MAMA_STATUS_NOT_FOUND	Not found
7	MAMA_STATUS_TIMER_FAILURE	Timer failure
8	MAMA_STATUS_IP_NOT_FOUND	IP address not found
9	MAMA_STATUS_TIMEOUT	Timeout (e.g. when subscribing to a symbol)
10	MAMA_STATUS_NOT_ENTITLED	Not entitled to the symbol being subscribed to
11	MAMA_STATUS_PROPERTY_TOO_LONG	Property too long
12	MAMA_STATUS_MD_NOT_OPENED	MD not opened
13	MAMA_STATUS_PUB_SUB_NOT_OPENED	Publish/subscribe not opened
14	MAMA_STATUS_ENTITLEMENTS_NOT_ENABLED	Entitlements not enabled
15	MAMA_STATUS_BAD_TRANSPORT_TYPE	Bad transport type
16	MAMA_STATUS_UNSUPPORTED_IO_TYPE	Using unsupported I/O type
17	MAMA_STATUS_TOO_MANY_DISPATCHERS	Too many dispatchers
18	MAMA_STATUS_NOT_IMPLEMENTED	Not implemented
19	MAMA_STATUS_WRONG_FIELD_TYPE	Wrong field type
20	MAMA_STATUS_BAD_SYMBOL	Bad symbol
21	MAMA_STATUS_IO_ERROR	I/O error
22	MAMA_STATUS_NOT_INSTALLED	Not installed
23	MAMA_STATUS_CONFLATE_ERROR	Conflation error
24	MAMA_STATUS_QUEUE_FULL	Event dispatch queue full
25	MAMA_STATUS_QUEUE_END	End of event queue reached
26	MAMA_STATUS_NO_BRIDGE_IMPL	No bridge
27	MAMA_STATUS_INVALID_QUEUE	Invalid queue
9001	MAMA_ENTITLE_STATUS_NOMEM	No memory
9002	MAMA_ENTITLE_STATUS_BAD_PARAM	Invalid parameter
9003	MAMA_ENTITLE_STATUS_BAD_DATA	The XML returned from entitlement server was invalid
9004	MAMA_ENTITLE_STATUS_URL_ERROR	Invalid URL
9005	MAMA_ENTITLE_STATUS_OS_LOGIN_?ID_UNAVAILABLE	Unable to determine OS ID of account process is running under
9006	MAMA_ENTITLE_STATUS_ALREADY_ENTITLED	An attempt is made to get entitlements after an already successful attempt
9007	MAMA_ENTITLE_STATUS_CAC_LIMIT_EXCEEDED	A user has exceeded concurrent access limit
9008	MAMA_ENTITLE_STATUS_OEP_LISTENER_?CREATION_FAILURE	Failed to create OEP listener that processes inbound messages from site server. Required for concurrent access control and/or dynamic entitlements update
9009	MAMA_ENTITLE_STATUS_HTTP_BASE	N/A

Numeric Value	Enum Name	Description
9010	MAMA_ENTITLE_HTTP_ERRHOST	No such host
9011	MAMA_ENTITLE_HTTP_ERRHOST	Cannot create socket
9012	MAMA_ENTITLE_HTTP_ERRCONN	Cannot connect to host
9013	MAMA_ENTITLE_HTTP_ERRWRHD	Write error on socket while writing to header
9014	MAMA_ENTITLE_HTTP_ERRWRDT	Write error on socket while writing data
9015	MAMA_ENTITLE_HTTP_ERRRDHD	Read error on socket while reading result
9016	MAMA_ENTITLE_HTTP_ERRPAHD	Invalid answer from data server
9017	MAMA_ENTITLE_HTTP_ERRNULL	Null data pointer
9018	MAMA_ENTITLE_HTTP_ERRNOLG	No/bad length in header
9019	MAMA_ENTITLE_HTTP_ERRMEM	Can't allocate memory
9020	MAMA_ENTITLE_HTTP_ERRRDDT	Read error while reading data
9021	MAMA_ENTITLE_HTTP_ERRURLH	Invalid URL (must start with 'http:://')
9022	MAMA_ENTITLE_HTTP_ERRURLP	Invalid port in URL
9023	MAMA_ENTITLE_HTTP_BAD_QUERY	Invalid query - HTTP result 400
9024	MAMA_ENTITLE_HTTP_FORBIDDEN	Forbidden
9025	MAMA_ENTITLE_HTTP_TIMEOUT	Request timeout - HTTP result 403
9026	MAMA_ENTITLE_HTTP_SERVER_ERR	Server error - HTTP result 500
9027	MAMA_ENTITLE_HTTP_NO_IMPL	Not implemented - HTTP result 501
9028	MAMA_ENTITLE_HTTP_OVERLOAD	Overloaded - HTTP result 503
9029	MAMA_ENTITLE_NO_USER	No User
9030	MAMA_ENTITLE_NO_SERVERS_SPECIFIED	No servers specified

19 Glossary

Term	Definition
Data Dictionary	***OpenMAMA*** object containing meta data for fields published on the **NYSE Technologies Market Data Platform**. (See Dictionary for details.)
Entitlements	Authorization/Authentication for market data subscriptions on the **NYSE Technologies Market Data Platform**.
FID	Field Identifier. Integer identifier used to uniquely identify a field within a MamaMsg.
Field Descriptor	***OpenMAMA*** object, obtained from the data dictionary, which contains meta information for a specific field. (See Dictionary for details.)
Initial Image	Point in time snapshot for all published data for a subscribed symbol on the NYSE Technologies Market Data Infrastructure. Typically the first data received for a new subscription. (See Subscriptions for Details)
IO	Input/Output. ***OpenMAMA*** object used to register interest in OS level file descriptor events (Only in C/C++/C#). (See IO for Details)
LBM	Latency Busters Messaging. Messaging middleware provided by 29West. (http://www.29west.com)
LBTRM	Latency Buster Transport - Reliable Multicast. This is the reliable multicast protocol implementation in use within the LBM middleware.
MAMA	Middleware Agnostic Messaging API
MAMDA	Middleware Agnostic Market Data API
OPRA	Options Pricing Regulation Authority http://www.opradata.com/
Queue	***OpenMAMA*** object used to control the dispatching of events within the API.
Recap	***OpenMAMA***, intraday, snapshot update of data for a particular symbol. (See Subscriptions for Details)
Subscription	***OpenMAMA*** object used to register interest in data for a particular symbol. (See Subscriptions for Details)
Symbol	***OpenMAMA*** terminology for the messaging concept of a Topic of information. (See Subscriptions for Details)
Tibrv	TIBCO Rendezvous. Messaging middleware provided by TIBCO. (http://www.tibco.com)
Timer	***OpenMAMA*** object used to trigger recurring event callbacks at a specified interval. (See Timers for details)
Topic	Messaging middleware name for an item of interest when subscribing to data. (See Subscriptions for Details)
Transport	***OpenMAMA*** object used to specify and configure communication protocols for subscriptions and publishing data via the API. (See Transports for details)
UTP	Nasdaq, Unlisted Trading Privileges, data feed. (http://www.nasdaqtrader.com/trader/mds/utpfeeds/utpfeeds.stm)
WombatMsg	**NYSE Technologies**, binary, wire data format. Used to propagate data on the **NYSE Technologies Market Data Platform**.