Celoxica
pure acceleration

Order Book OB

API Reference Guide

About This Document

This document describes the OB API reference guide.

Copyright Information

Sales	sales@celoxica.com
Customer Support	support@celoxica.com
Website	http://www.celoxica.com

UK Head Office
Celoxica Limited
34 Porchester Road
London
W2 6ES, UK
Phone: +44 (0) 20 7313 3180

US Head Office
Celoxica Inc.
275 Madison Avenue, Suite 404
New York, NY
10016, USA
Phone: +1 (0) 212 880 2075

US Chicago Office
Celoxica Inc.
141 W Jackson Blvd, Suite 2350
Chicago, IL
60604, USA
Phone: +1 (0) 312 893 1204

Celoxica
pure acceleration

Content

Revisions

Revision	Date	Description of Changes
R2012-6.2	11 SEP 2012	Description update for OBGroupRemoveSubscription() and OBGroupRemoveSubscriptionEx()
R2012-6.0	17 AUG 2012	New template
<= 2.20		Older versions

1. Order Book API

1.1 Description

1.1.1 Overview

Celoxica's Order Book provides the most efficient way to process data from market feeds. The accelerator card streams data to multiple DMA queues and the OB library creates a thread to process each of these queues.

Workload is automatically balanced between all the cores processing the feeds. Users can also use multiple threads to read book updates.

1.1.2 Usage

The Order Book application is initiated by the client API function call `OBOpen()`.

The Order Book application is closed using `OBClose()`.

1.2 Functions

1.2.1 OBOpen()

Description Opens the book, and sets the OB pointer. Initializes market dependent structures, connects to the market(s).

This function is in charge of:

- Get the configuration files to the Order Book handling module
- Initiate the generic GMAC modules using the configuration for all the markets in the configuration file
- Wait and receive the information on the data referential structure so as to be able to read it
- Read the data referential
- Create the local order books for all the corresponding instruments
- Listen to the market data information and update the relevant order book accordingly

Prototype `OBStatus OBOpen(OBHandle **OB, Config *OBConfigPtr)`

Arguments

`OB`	OB handle
`OBConfigPtr`	Pointer to the config structure used for initialization

1.2.2 OBOpenEx()

Description Extended version of `OBOpen()`, accepting optional parameters

Prototype

```
OBStatus OBOpenEx(OBHandle **OB, Config
*OBConfigPtr, OBOpenExtraParams *ExtraParams)
```

Arguments

`OB`	OB handle
`OBConfigPtr`	Pointer to the configuration structure used for initialization
`ExtraParams`	Extra parameters to use for OB creation

1.2.3 OBClose()

Description Closes the book and frees up used resources

Prototype `void OBClose(OBHandle *OB)`

Arguments

`OB`	OB handle

1.2.4 OBStatusToString()

Description Transforms the OB status to a string representation

Prototype `char* OBStatusToString(OBStatus St)`

Arguments

`St`	Status code

Returns String representation of the status

1.2.5 OBGetMarket()

Description Reconciles Market Identification with Market Name.

Provides mapping from market identification to the market name.

Prototype
```
OBStatus OBGetMarket (OBHandle *OB, char
**MarketName, MarketID M)
```

Arguments

`OB`	OB handle
`MarketName`	Pointer to where pointer to the name is stored
`M`	Market ID

1.2.6 OBSymbolListCreate()

Description Retrieve data referential information. Returns a list of all the currently registered symbols in the book.

This function applies a lock on the data referential block, hence its use has a significantly impact on performance. It should therefore only be used for debugging purposes.

Data should be freed using `OBSymbolListDestroy()`.

See the GMAC API Reference Guide for details on GMAC EISIN, which is a Celoxica-specific "Extended ISIN" code.

Prototype
```
OBStatus OBSymbolListCreate (OBHandle *OB,
GMACEISIN **ArrayPtr, int *Size)
```

Arguments

`OB`	OB handle
`ArrayPtr`	Pointer where data array is returned
`Size`	Size of the array

1.2.7 OBSetGMACReferentialReceivedcallback()

Description Sets the OB callback to call when GMAC data referential are received

Prototype
```
OBStatus
OBSetGMACReferentialReceivedCallback(OBHandle *OB,
OBOnGMACReferentialReceivedCallback_t Callback,
void *Private)
```

Arguments

`Callback`	The callback function to call whenever a GMAC Referential message is received
`Private`	Private user data that will be provided to the callback

1.2.8 OBSymbolListDestroy()

Description Frees symbol list.

Frees data returned by `OBSymbolListCreate()`.

See the GMAC API Reference Guide for details on GMAC EISIN, which is a Celoxica-specific "Extended ISIN" code.

Prototype `void OBSymbolListDestroy (GMACEISIN *Array)`

Arguments

`Array`	GMAC handle

1.2.9 OBRegisterErrorHandler()

Description Registers Error Handler.

This function allows to register specific error handler.

Prototype `void OBRegisterErrorHandler(OBHandle *OB, OBErrorHandler Handler)`

Arguments

`OB`	OB handle
`Handler`	Error handler function pointer

1.2.10 OBRegisterErrorHandlerForGMAC()

Description Registers an Error Handler for GMAC. This function registers a specific error handler for GMAC.

This error handler is called before the status codes are returned and allows the user to change the error code being returned, as well as react on it.

Prototype `void OBRegisterErrorHandlerForGMAC(OBHandle *OB, GMACErrorHandler Handler)`

Arguments

`OB`	OB handle
`Handler`	Error handler function pointer

1.2.11 OBRegisterTimeoutHandler()

Description Registers Timeout Handler.

This function allows to register specific timeout handler. It will be called when a timeout is detected on a multicast.

Prototype `void OBRegisterTimeoutHandler(`**OBHandle** `*OB,` **OBTimeoutHandler** `Handler, void *Private)`

Arguments

`OB`	OB handle
`Handler`	Timeout handler function pointer
`Private`	An user defined pointer that will be passed as parameter to the function

1.2.12 OBGetMarketNames()

Description Gets market names.

This function returns a list of market names

Prototype `OBStatus OBGetMarketNames (OBHandle *OB, OBMarketNames MarketNames)`

Arguments	`OB`	OB handle
	`MarketNames`	List of market names

1.2.13 OBGetMulticastMappingTable()

Description Returns the multicast mapping table for a specified market. This function returns a mapping between Multicast IDs and IP addresses/ports. It can be called after `OBOpen()`, which ensures that the OB configuration has been read and processed. The function allocates the memory for the mapping table, for which the address is returned via `MulticastMappingPtr`. The memory is allocated using libc malloc(). It is the user's responsibility to free this memory using a compatible method.

Prototype

```
OBStatus OBGetMulticastMappingTable(OBHandle *OB,
unsigned MarketID, GMACMulticastMapping
**MulticastMappingPtr)
```

Arguments	`OB`	OB handle
	`MarketID`	The Market ID of interest for which the multicast mapping table should be populated
	`MulticastMappingPtr`	Address of the multicast mapping table.

2. Order Book Processing Updates API

2.1 Description

2.1.1 Overview

The Order Book framework process all data received on the market in the internal book; the user doesn't have access to these data. To avoid locking the user is provided with his own copy of the book at the desired time, and subsequent consecutive updates.

The book updates are designed in a way that allows the application to get access to the user's copy of the book in constant time.

2.1.2 Usage

After the Order Book framework was properly initialized, the user can create Subscription groups, using `OBGroupOpen()`. Each group comes with a buffer for book updates the user can read.

When the group is created the user can subscribe to multiple instruments within the group. Copy of the book is generated just after the subscription, and subsequent updates are stored in the group buffer.

The user should read and apply the updates using `OBGroupRead()` function, or register a callback function using `OBGroupWait()`.

User can inspect the update and/or apply it to his own copy. The user can write the code applying the updates, or use the provided function `OBProcessDeepBookUpdate()` or `OBProcessConsolidatedBookUpdate()`.

If the `OBGroupRead()` function is used, the user must free data when processing is done using `OBGroupRelease()`.

If an error occurs, e.g. the user is not keeping up and the updates buffer is full, the OB library will generate a subscription failure notification. OB notifies the user that the updates for the local book are no longer being generated. An update with action 'F' (see Order Book Update Header description for the full list of update actions) is inserted to the group buffer. The user then has to call `OBGroupRemoveSubscription()` to clean-up all used resources related to the subscribed instrument, and then subscribe again.

To close the group, user can call `OBGroupClose()` knowing that `OBClose()` will close all groups automatically as well as `OBGroupClose()` will close all the subscriptions.

2.1.2.1 Multithreading

The user can use more threads when reading the updates; the sole requirement is to call all functions related to the same group from the same thread. It is recommended to have a thread for each subscription group.

2.1.2.2 Call Backs

The Order Book framework gives the user the possibility to use two different call back mechanisms, which have very different behavior.

The user can register call back function by calling `OBGroupOpen()`, this is called an 'instant callback'. Passed callback function is called when any of the subscribed instruments are updated in the internal book. The callback functions is therefore called from the thread (CPU) processing the feed. In general, that means that it may be invoked from multiple CPUs at the same time. Doing any extensive computation will slow down feed processing, and may cause performance issues.

It is not recommended to use this feature for normal trading.

2.1.2.3 Reading the Deep Book

Copied (user local) data are stored in **OBDeepBookData** structure. The pointer can be acquired using OB_GET_DEEPBOOK macro called on subscription handle pointer. The pointer is provided when subscribing, and as well, within book updates.

The structure contains trading information like OHLC, trading status ... and pointer to Order Buffer, where all the book information is stored.

Book Orders **OBDeepBookOrder** structure are linked together and ordered by price, and time of arrival.

There are access macros to read book orders:

- OB_ORDER_BEST_ASK_OFFSET
- OB_ORDER_BEST_BID_OFFSET
- OB_ORDER_BEST_ASK
- OB_ORDER_BEST_BID

- OB_ORDER_NEXT

2.1.2.4 Reading the Consolidated Book

Copied (user local) data are stored in the **OBConsolidatedBookData** structure. The pointer can be acquired using OB_GET_CONSOLIDATEDBOOK macro called on subscription handle pointer. The pointer is provided when subscribing, and as well, within book updates.

2.2 Functions

2.2.1 OBGetPriceDivisor()

Description Returns fixed point price correction.

All prices are returned as 64bit number with fixed point precision.

To compute a correct price: (float) Price / `OBGetPriceDivisor()`

Prototype
```
unsigned int OBGetPriceDivisor(void)
```

Returns Price correcting divisor

2.2.2 OBReadUpdates()

Description Reads order updates from the book.

The function never blocks. Returns a NULL pointer when no data are available.

Prototype
```
OBStatus OBReadUpdates(OBSubscriptionHandle *OBSub,
OBBookUpdate **Update)
```

Arguments

`OBSub`	Subscription handle
`Update`	Book update

2.2.3 OBReleaseUpdates()

Description Releases a book update.

Function must be called for every update received by `OBReadUpdates()`.

`OBGroupRemoveSubscription()` call frees all Updates as well.

Calling this function after the `OBGroupRemoveSubscription()` call for the instrument will cause errors.

Prototype
```
void OBReleaseUpdates(OBBookUpdate *Update)
```

Arguments

`Update`	Book update

2.2.4 OBGroupOpen()

Description Creates Subscription Group.

OB API allows grouping listing on multiple instruments.

It is possible to set up listing for multiple instruments within one group and then listen for updates on the group.

The most common use is to have one group per trading strategy, or per thread/CPU.

Instruments can be added to the group using `OBGroupAddSubscription()` call.

`InstantCallback` parameter allows user to perform computation before data are stored in the group queue. It is not recommended to use this callback for trading computations. The function can be called multiple times in parallel from different CPUs. Making extensive computation will slow down

processing of the whole Order Book. It is meant to be used for redistribution (tcp/udp writes), and/or possible filtering.

It is not safe to work with the same subscription group from multiple-threads.

Prototype

```
OBStatus OBGroupOpen(OBHandle *OB,
OBSubscriptionGroupHandle **SubscriptionGroupPtr,
int BusyWait, OBInstantCallback_t InstantCallback,
void *Private)
```

Arguments

`OB`	OB handle
`SubscriptionGroupPtr`	Pointer where the subscription group handle is returned
`BusyWait`	If set to '1', the reading from the group will be busy wait, otherwise the process will sleep. Note: Must be set to '1', as sleeping is disabled
`InstantCallBack`	Pointer to callback function, the callback is called before the data are stored in the group queue from the processor executing the update on local structures. Use NULL for no instant callback
`Private`	Pointer to be passed to the callback function

2.2.5 OBGroupClose()

Description Closes Subscription Group.

Frees up all resources of subscriptions belonging to the group and the group itself.

Prototype

```
OBStatus OBGroupClose(OBSubscriptionGroupHandle
*SubscriptionGroup)
```

Arguments

`SubscriptionGroup`	Subscription group handle

2.2.6 OBGroupRead()

Description Reads Data from a group of subscriptions.

The function reads order book updates for all subscriptions belonging to the specified group. It returns a pointer to the head of the linked list that is accessible through the `UpdateHead` parameter. The queue is not guaranteed to be emptied by a single call to `OBGroupRead()`; multiple calls to the function may be necessary to retrieve all the updates in the queue.

The user can specify whether the function should block if no data are available: the blocking behaviour is selected when creating the subscription group via `OBGroupOpen()`.

`OBGroupRelease()` must be called once processing of the updates is complete.

Prototype

```
OBStatus OBGroupRead(OBSubscriptionGroupHandle
*SubscriptionGroup, OBBookUpdate *UpdateHead,
unsigned int *CountPtr, int Block)
```

Arguments

`SubscriptionGroup`	Group to read data from
`UpdateHead`	Head of the list of updates
`CountPtr`	Number of updates in the linked list returned
`Block`	The function will return immediately if there is no data and `Block` is set to '0'

2.2.7 OBGroupRelease()

Description Releases updates.

Releases data returned by `OBGroupRead()`.

Prototype

```
void OBGroupRelease(OBBookUpdate UpdateHead)
```

Arguments

`UpdateHead`	Head of the linked list to release

2.2.8 OBGroupAddSubscription()

Description Subscribes to a specific instrument.

Prototype

```
OBStatus
OBGroupAddSubscription(OBSubscriptionGroupHandle
*SubscriptionGroup, OBSubscriptionHandle **OBSub,
int MarketID, char *EISIN, int Aggregated, int
MaxDepth)
```

Arguments

`SubscriptionGroup`	Subscription group handle
`OBSub`	Pointer where subscription handle will be created
`MarketID`	Market ID
`EISIN`	EISIN code of the instrument we are subscribing to. See the GMAC API Reference Guide for details on GMAC EISIN, which is a Celoxica-specific "Extended ISIN" code.
`Aggregated`	Specifies if the output should be aggregated (quantities per level summed)
`MaxDepth`	Specifies what is the maximal book depth to go to, when creating user data (0 for unlimited)

2.2.9 OBGroupAddSubscriptionEx()

Description Subscribes to a specific instrument.

Prototype

```
OBStatus
OBGroupAddSubscriptionEx(OBSubscriptionGroupHandle
*SubscriptionGroup, OBSubscriptionHandle **OBSub,
int MarketID, char *EISIN, int Aggregated, int
MaxDepth, int Force)
```

Arguments

`SubscriptionGroup`	Subscription group handle
`OBSub`	Pointer where subscription handle will be created
`MarketID`	Market ID
`EISIN`	EISIN code of the instrument we are subscribing to. See the GMAC API Reference Guide for details on GMAC EISIN, which is a Celoxica-specific "Extended ISIN" code.
`Aggregated`	Specifies if the output should be aggregated (quantities per level summed)
`MaxDepth`	Specifies what is the maximal book depth to go to, when creating user data (0 for unlimited)
`Force`	Set to 1 to force the subscription. It will subscribe without previous instrument registration.

2.2.10 OBGroupAddSubscriptionExWithPrivate()

Description Subscribes to a specific instrument.

Prototype

```
OBStatus
OBGroupAddSubscriptionExWithPrivate(OBSubscriptionGroupHandle *SubscriptionGroup, OBSubscriptionHandle **OBSub, int MarketID, char *EISIN, int Aggregated, int MaxDepth, int Force, void *PrivatePtr)
```

Arguments

`SubscriptionGroup`	Subscription group handle
`OBSub`	Pointer where subscription handle will be created
`MarketID`	Market ID
`EISIN`	EISIN code of the instrument we are subscribing to. See the GMAC API Reference Guide for details on GMAC EISIN, which is a Celoxica-specific "Extended ISIN" code.
`Aggregated`	Specifies if the output should be aggregated (quantities per level summed)
`MaxDepth`	Specifies what is the maximal book depth to go to, when creating user data (0 for unlimited)
`Force`	Set to 1 to force the subscription. It will subscribe without previous instrument registration.
`PrivatePtr`	Pointer to the private data used by the subscription

2.2.11 OBGroupRemoveSubscription()

Description Stops listening for updates on specific instruments; all resources consumed by the subscription will be freed.

The call is synchronous and waits until the book building CPU has freed the subscriptions.

The function times out after 0.5 second, in which case a status OB_STATUS_SUBSCRIPTION_FAILED is returned. The subscription may still be successfully freed later as the work was scheduled.

For an asynchronous version, see `OBGroupRemoveSubscriptionEx()`.

Prototype

```
OBStatus
OBGroupRemoveSubscription(OBSubscriptionGroupHandle *SubscriptionGroup, OBSubscriptionHandle *OBSub)
```

Arguments

`SubscriptionGroup`	Subscription group handle
`OBSub`	Subscription handle

2.2.12 OBGroupRemoveSubscriptionEx()

Description Stops listening for updates on specific instruments; all resources consumed by the subscription will be freed.

The call is asynchronous and the subscription will be freed in a separate thread. This version includes a callback function to notify the caller when the subscription is freed.

For a synchronous version, see `OBGroupRemoveSubscription()`.

Prototype

```
OBStatus
OBGroupRemoveSubscriptionEx(OBSubscriptionGroupHandle *SubscriptionGroup, OBSubscriptionHandle *OBSub, OBSubscriptionCallback_t Callback, void *CallbackPrivate)
```

Arguments

`SubscriptionGroup`	Subscription group handle
`OBSub`	Subscription handle
`Callback`	Optional callback function to be called when the subscription is freed
`CallbackPrivate`	Private data for the callback

2.2.13 OBGroupWait()

Description Waits on the Group of Subscriptions.

This function allows user to use callbacks instead reading the group queue in cycle using `OBGroupRead()`.

When data arrives supplied callback function will be called.

The main difference between this callback mechanism and the callback function supplied in `OBGroupOpen()` is that the updates goes through group queue and callback are always called from the thread calling `OBGroupWait()`.

Prototype

```
OBStatus OBGroupWait(OBSubscriptionGroupHandle *SubscriptionGroup, OBUpdateCallback_t CallBack, void *Private)
```

Arguments

`SubscriptionGroup`	Subscription group handle
`Callback`	Processing callback function
`Private`	Pointer to be passed to the callback function

2.2.14 OBGroupStop()

Description Allows to unblock a blocking call `OBGroupRead()` at any time. The stop flag is persistent.

Prototype

```
void OBGroupStop(OBSubscriptionGroupHandle *SubscriptionGroup, int StopFlag)
```

Arguments

`SubscriptionGroup`	Subscription group handle
`StopFlag`	When set the OBGroupRead() will unblock and return OB_STATUS_STOP code. The stop flag is persistent. To continue reading user needs to call this function again and set the `StopFlag` to zero

2.2.15 OBSubscriptionGetPrivatePtr()

Description Gets the private pointer bound to a subscription set by `OBSubscriptionSetPrivatePtr()`.

Prototype

```
void *OBSubscriptionGetPrivatePtr(OBSubscriptionHandle *OBSub)
```

Arguments

`OBSub`	Subscription handle

2.2.16 OBSubscriptionSetPrivatePtr()

Description Sets the private pointer for a subscription.

This function associates a private user pointer with a valid subscription. User can retrieve this pointer anytime by `OBSubscriptionGetPrivatePtr()`.

Prototype

```
void OBSubscriptionSetPrivatePtr(OBSubscriptionHandle *OBSub, void *PrivatePtr)
```

Arguments

`OBSub`	Subscription handle
`PrivatePtr`	Pointer to the private data used by the subscription

2.2.17 OBSubscriptionGetEISIN()

Description Gets the EISIN (Celoxica-specific "Extended ISIN") for a subscription.

This function returns a string representation of the EISIN for a subscription.

See the GMAC API Reference Guide for details on GMAC EISIN, which is a Celoxica-specific "Extended ISIN" code.

Prototype

```
char* OBSubscriptionGetEISIN(OBSubscriptionHandle *OBSub)
```

Arguments

`OBSub`	Subscription handle

Returns String representation of the EISIN

2.2.18 OBTranslateMarketToEISINEx()

Description Translates the market local name to EISIN if this feature is supported for the specific market.

Prototype

```
OBStatus OBTranslateMarketToEISINEx(OBHandle *OB, int MarketID, GMACLocalName Src, GMACEISIN Dst)
```

Arguments

`OBH`	OB handle
`MarketID`	Market ID
`Src`	Market local name
`Dst`	Pointer to where the ISIN will be copied (must be allocated by caller). See the GMAC API Reference Guide for details on GMAC EISIN, which is a Celoxica-specific "Extended ISIN" code.

2.2.19 PrintTimestamp()

Description Prints a timestamp.

Prototype

```
void PrintTimestamp(FILE *f, uint64_t Timestamp)
```

Arguments

`f`	Stream to print to
`Timestamp`	Timestamp to be printed

2.2.20 OBPrintDeepBook()

Description Prints the user's book.

Prototype `void OBPrintDeepBook(FILE *f, OBDeepBookUpdate *BD, int Limit)`

Arguments		
	`f`	Stream to print to
	`BD`	Book to be printed
	`Limit`	Book depth to be printed

2.2.21 OBPrintDeepBookUpdate()

Description Prints the update content.

Prototype `void OBPrintDeepBookUpdate(FILE *f, OBBookUpdate Update)`

Arguments		
	`f`	Stream to print to
	`Update`	Update to print

2.2.22 OBProcessDeepBookUpdate()

Description Processes updates for the user into his book.

Prototype `OBStatus OBProcessDeepBookUpdate(OBBookUpdate Update)`

Arguments		
	`Update`	Pointer of an update received by `OBReadUpdates()`

2.3 Convenience Macros

2.3.1 OB_UPDATE_NEXT

Description Provides the next update.

Updates returned by `OBGroupRead()` are linked together in linked list.

This macro allows traversing the list.

Prototype `OB_UPDATE_NEXT (PTR)`

2.3.2 OB_TRADING_STATUS

Description Provides the cumulative trading status of an instrument.

Computed from trading statuses of Market, Channel, Segment and Instrument, if available.

Prototype `OB_TRADING_STATUS (Status)`

2.3.3 OB_TRADING_STATUS_INSTRUMENT

Description Provides the trading status of an instrument.

This trading status refers just to instrument; it may still not be possible to trade if the whole market trading is disabled. If unset, it will be set to

OB_TRADING_CLOSED when the book state is CLEAN, otherwise it will be set to OB_TRADING_UNKNOWN.

Prototype `OB_TRADING_STATUS_INSTRUMENT (Status)`

2.3.4 OB_TRADING_STATUS_CHANNEL

Description Provides the trading status of a channel.

Some feeds/markets provide trading statuses for channels. The channel may be disabled by OB framework as well, if recovery mechanism fails to retrieve missed data.

It is set to OB_TRADING_OPEN by default.

Prototype `OB_TRADING_STATUS_CHANNEL (Status)`

2.3.5 OB_TRADING_STATUS_SEGMENT

Description Provides the trading status of a segment.

Some feeds/markets provide trading statuses for segments.

It is set to OB_TRADING_OPEN by default.

Prototype `OB_TRADING_STATUS_SEGMENT (Status)`

2.3.6 OB_TRADING_STATUS_MARKET

Description Provides the trading status of a market.

Prototype `OB_TRADING_STATUS_MARKET (Status)`

2.3.7 OB_TRADING_STATUS_SECTOR

Description Provides the trading status of a sector.

Some feeds/markets provide trading statuses for sectors.

It is set to OB_TRADING_OPEN by default.

Prototype `OB_TRADING_STATUS_SECTOR (Status)`

2.3.8 OB_ORDER_NEXT

Description Provides the next order.

Prototype `OB_ORDER_NEXT (OB_BOOK_DATA, POS)`

2.3.9 OB_ORDER_BEST_ASK_OFFSET

Description Provides the offset of the best ask.

Prototype `OB_ORDER_BEST_ASK_OFFSET (OB_BOOK_DATA)`

2.3.10 OB_ORDER_BEST_BID_OFFSET

Description Provides the offset of the best bid.

Prototype `OB_ORDER_BEST_BID_OFFSET (OB_BOOK_DATA)`

2.3.11 OB_ORDER_BEST_ASK

Description Provides the best ask order.

Prototype `OB_ORDER_BEST_ASK (OB_BOOK_DATA)`

2.3.12 OB_ORDER_BEST_BID

Description Provides the best bid order.

Prototype `OB_ORDER_BEST_BID (OB_BOOK_DATA)`

2.3.13 OB_GET_DEEPBOOK

Description Provides the deep book.

Prototype `OB_GET_DEEPBOOK (OB_BOOK_DATA)`

3. Super Book API

3.1 Description

3.1.1 Overview

The super book building is implemented on top of the standard Celoxica order book API.

In order to reduce the CPU usage, super book is not built for the whole market, but just for instruments of interest.

The API also gives the user the ability to build the super book from any instruments specified. They don't need to be on different markets. The features provided by underlying deep book API, like limiting the depth of the book, or consolidating on the prices, are preserved.

3.1.2 Usage

The user has to call `OBSuperBookCreate()` to create a super book.

It is then possible to add already active (subscribed) subscriptions using `OBSuperBookAddSubscription()`: this will allocate the additional buffer for super book links and merge the new orders into the already existing super book. This should be called repetitively to add all instruments which should be part of the super book.

The usual call `OBProcessDeepBookUpdate()` will update the super book linking if the subscription is part of the super book.

Data in the original user's copy of the book are preserved and can be accessed even if the subscription is added in the super book.

3.2 Functions

3.2.1 OBSuperBookCreate()

Description Creates Super Book structure.

Allocates the necessary resources for the Super Book.

Prototype `OBStatus OBSuperBookCreate(OBHandle *H, OBSuperBookData **SB)`

Arguments

`H`	OB handle
`SB`	Pointer to a **OBSuperBookData** structure where data should be written

3.2.2 OBSuperBookAddSubscription()

Description Adds instrument to the Super Book.

Once the subscription is created by `OBGroupAddSubscription()` call, the subscription can be made part of the super book.

This call will merge the orders representing the subscription into the orders in the super book. Each `OBProcessDeepBookUpdate()` on updates for this subscription will now maintain the changes to the super book as well.

The instrument (subscription) added first will have index 0, the second one, index 1, etc ... It is possible to add subscriptions from the same market or from different markets as well. It is possible to use different settings for each added subscription, in terms of price aggregation and/or limiting the depth, see `OBGroupAddSubscription()` for more details.

Prototype `OBStatus OBSuperBookAddSubscription(OBSuperBookData *SB, OBSubscriptionHandle *Sub)`

Arguments

`SB`	Super Book handle
`Sub`	Subscription handle

3.2.3 OBSuperBookRemoveSubscription()

Description Removes an instrument from the Super Book.

This call will remove a specific subscription/instrument form the super book. Internally, the subscription will remain active and updates for it will continue to be processed.

The Super Book Index used by the instrument in question will be reused by the last Super Book Index present in the super book.

After removing a subscription using this method, it is possible to re-subscribe using `OBSuperBookAddSubscription()`.

Prototype
```
OBStatus
OBSuperBookRemoveSubscription(OBSuperBookData *SB,
OBSubscriptionHandle *Sub)
```

Arguments

`SB`	Super Book handle
`Sub`	Subscription handle

3.2.4 OBSuperBookDestroy()

Description Destroys the super book structure.

This call frees up the resources taken by the super book, and removes all the subscriptions which were part of it from the super book.

Prototype
```
OBStatus OBSuperBookDestroy(OBSuperBookData *SB)
```

Arguments

`SB`	Super Book handle

3.2.5 OBPrintSuperBook()

Description Prints the super book.

Prototype
```
void OBPrintSuperBook(FILE *f, OBSuperBookData *SB,
int Limit)
```

Arguments

`f`	Stream to print to
`SB`	Super Book handle
`Limit`	How many lines to print, use -1 for unlimited

3.3 Convenience Macros

3.3.1 OB_GET_SUPERBOOK

Description Gets user copy of the super book from the subscription handle.

Prototype
```
OB_GET_SUPERBOOK(Subscription)
```

4. Consolidated Book API

4.1 Functions

4.1.1 OBPrintConsolidatedBookOrder()

Description Prints a consolidated book order.

Prototype
```
void OBPrintConsolidatedBookOrder(FILE *f,
OBConsolidatedBookOrder *O)
```

Arguments

`f`	Stream to print to
`O`	Book order to be printed

Returns Printed consolidated book order

4.1.2 OBPrintConsolidatedBook()

Description Prints the user's book.

Prototype
```
void OBPrintConsolidatedBook(FILE *f,
OBConsolidatedBookData *BD)
```

Arguments

`f`	Stream to print to
BD	Book data to be printed

Returns Printed consolidated book

4.1.3 OBPrintConsolidatedBookUpdate()

Description Prints the update content.

Prototype
```
void OBPrintConsolidatedBookUpdate(FILE *f,
OBBookUpdate Update)
```

Arguments

`f`	Stream to print to
`Update`	Update to be printed

Returns Printed consolidated book update

4.1.4 OBProcessConsolidatedBookUpdate()

Description Processes updates for the user into his book.

Prototype
```
OBStatus
OBProcessConsolidatedBookUpdate(OBBookUpdate
Update)
```

Arguments

`Update`	Update received by `OBGroupRead()`

4.2 Convenience Macros

4.2.1 OB_GET_CONSOLIDATEDBOOK

Description Gets user copy of the book from the subscription handle.

Prototype `OB_GET_CONSOLIDATEDBOOK(Subscription)`

5. Hardware Filtering API

5.1 Functions

5.1.1 OBHWFilteringisEnabled()

Description Queries the state of the hardware filtering on a market.

This function is used to check the status of hardware filtering on a given channel.

Prototype `int OBHWFilteringIsEnabled(OBHandle *OB, int MarketID)`

Arguments

`OB`	OB handle
`MarketID`	A valid market ID for the market to enable filtering on

Returns -1 on error, else non-zero if filtering is enabled.

5.1.2 OBHWFilteringForce()

Description Forces hardware filtering on the market.

This function is used to force filtering on the market that use indexes when the OB is still waiting for reference data. This can occur if a symbol in the filter list is for whatever reason missing from the reference data.

It should be used with a timeout in conjunction with `OBHWFilteringisEnabled()` - e.g. after five minutes have passed - or with a user signal. It can safely be called against a market where filtering is already enabled.

Note:

This should be used with extreme caution as because any symbols where the reference data has not been processed will be filtered out until the reference data is received. This may result in an incomplete or invalid book for such instruments.

This function will not return an error status if it is called against a valid market ID that does not support filtering. However, an error message will be displayed in such cases.

This is an asynchronous call. Querying the hardware filtering state immediately after this function may indicate filtering is not enabled even if it is.

Prototype `OBStatus OBHWFilteringForce(OBHandle *OB, int MarketID)`

Arguments

`OB`	OB handle
`MarketID`	A valid market ID for the market to enable filtering on

5.1.3 OBHWFilteringQuery()

Description Queries hardware filtered symbols.

This function is used to determine what symbols will be filtered by the hardware if/when hardware filtering is enabled on the market and which symbols were configured to be filtered but will not because no reference data has been received for the symbol.

Note:

This function requires you to provide sufficient memory to store all filtered symbols on the given market. It will return

OB_STATUS_MALLOC_ERROR if there is insufficient space in the list.

Prototype

```
OBStatus OBHWFilteringQuery(OBHandle *OB, int
MarketID, GMACEISIN *List, size_t Space, int
*Count, unsigned short PresenceFlag)
```

Arguments

`OB`	OB handle
`MarketID`	A valid market ID for the market to enable filtering on
`List`	A block of memory large enough to hold at least one GMAC EISIN
`Space`	The size of List in bytes
`Count`	A pointer to an integer where the number of symbols will be written if this call is successful
`PresenceFlag`	If 0 (zero) this call will attempt to return a list of symbols that will be added to the hardware filter when reference data has arrived. If non-zero this function will return a list of symbols already in the hardware filter

6. Statistics API

6.1 Functions

6.1.1 OBStatsChannel()

Description Gets pointer to live statistics.

Function sums stats of all open and already closed channels.

Prototype

```
OBStatus OBStatsChannel(OBHandle *OB, int MarketID,
int LocalChannelID, GMACStats **Stats)
```

Arguments

`OB`	OB handle
`MarketID`	Market ID
`LocalChannelID`	Channel ID local to the market, the ID from the configuration file
`Stats`	Pointer to the stats structure

6.1.2 OBStatsGetQueues()

Description Fills in Statistics.

Gets the number of threads the OB is running on.

Prototype

```
int OBStatsGetQueues(OBHandle *OB)
```

Arguments

`OB`	OB handle

Returns The total number of enabled queues

6.1.3 OBStatsMulticast()

Description Returns pointer to live statistics of a particular channel.

Prototype

```
OBStatus OBStatsMulticast(OBHandle *OB,
GMACMulticastID MulticastID, GMACStats  **Stats)
```

Arguments

`H`	OB handle
`MulticastID`	Multicast ID
`Stats`	Pointer to the stats structure

6.1.4 OBStatsQueueGlobal()

Description Returns the 'live' global statistics i.e. the current statistics for all the multicast channels in use. The function accesses registers on the accelerator card, which is an expensive operation. It should therefore not be called excessively, and it should not be called from performance-sensitive code.

Prototype

```
OBStatus OBStatsQueueGlobal(OBHandle *OB, int
Queue, GMACStats **Stats, uint64_t *PacketsMissed,
uint64_t *CRCErrors)
```

Arguments

`H`	OB handle
`Queue`	Queue number
`Stats`	Pointer to the stats structure

	`PacketsMissed`	Packets dropped by the accelerator card, because software wasn't keeping up
	`CRCErrors`	Packets dropped by the accelerator card, because of underlying protocol errors

6.1.5 OBStatsMulticastCount()

Description Returns number of open channels

Prototype
```
void OBStatsMulticastCount(OBHandle *OB, int *Count)
```

Arguments		
	`Ob`	OB handle
	`Count`	Channel count

Returns Number of open channels

6.1.6 OBStatsMarketMulticastCount()

Description Returns number of open multicasts for specific market

Prototype
```
OBStatus OBStatsMarketMulticastCount(OBHandle *OB, int MarketID, int *Count)
```

Arguments		
	`OB`	OB handle
	`MarketID`	Market ID
	`Count`	Channel count

6.1.7 OBStatsMarketChannelCount()

Description Returns number of open channels for specific market

Prototype
```
OBStatus OBStatsMarketChannelCount(OBHandle *OB, int MarketID, int *Count)
```

Arguments		
	`OB`	OB handle
	`MarketID`	Market ID
	`Count`	Channel count

6.1.8 OBStatsMarketCount()

Description Returns number of markets

Prototype
```
void OBStatsMarketCount(OBHandle *OB, int *Count)
```

Arguments		
	`OB`	OB handle
	`Count`	Market count

Returns Number of markets

6.1.9 OBGroupGetStats()

Description This function allows user code to query the OBGroup buffer usage in terms of numbers of updates buffered. It doesn't take locks in order to make it performance non-intrusive. It is safe to run this function from any thread, as

long as the OBGroup exists.

Please note that one internal buffer is created for each book building thread, but `UpdatesBuffered` and `MaxUpdatesBuffered` are reported as sums across all underlying buffers. Therefore, under special circumstances, subscription can fail on 50% buffer usage. For instance, consider a scenario with 2 threads using 2 buffers, where total size (as reported by `MaxUpdatesBuffered`) is 500. If one thread has already pushed 250 updates, it will not be able to push any more updates as it will have already consumed all the space allocated to it.

Prototype

```
OBStatus OBGroupGetStats(OBSubscriptionGroupHandle
*SubscriptionGroup, uint32_t *UpdatesBuffered,
uint32_t *MaxUpdatesBuffered)
```

Arguments

`SubscriptionGroup`	Subscription group handle
`UpdatesBuffered`	Number of updates currently queue in the OBGroup buffer
`MaxUpdatesBuffered`	Maximum number of updates the buffer can hold Configurable using the updates bufferpool configuration option.

7. Latency API

7.1 Description

The Order Book framework uses advanced hardware assisted latency measurement techniques to give users the possibility to monitor latency through the day and to tune the system settings on the host machine for the best possible performance.

Each packet received is time stamped by the accelerator card: a 32 bit 125 MHz counter (giving 8 ns resolution) value is appended to each packet. The timestamp is propagated to the user with each book update. The user has the option to send the timestamp value back to the accelerator card via `OBLatencyCheckPoint()` at any point in his processing, and thus to measure the latency from the wire to the point where said method was called. The card calculates the difference between the timestamp and the current counter value. The actual and average latencies are reported by the card periodically. The overhead of sending the timestamp back to the card is minimal.

Additionally, each update has a CPU timestamp, which is appended when data enter the book building block. Users can thus track the amount of time spent inserting data into the book and buffering the update between CPUs. The supplied CPU timestamp library is used to provide these CPU timestamps.

Both of these timers are used in all the examples provided with Order Book package to track latency, for illustrative purposes.

Certain OS properties can influence the performance of the Order Book application.

Some simple rules:

- CPUs that need to exchange data should be on the same die:
  - A die usually has memory banks attached to it, memory access is fastest there
  - The CPU processing the book and the user thread listening for updates communicate through a buffer
  - Some dies may have empty memory banks, CPUs on these dies will exhibit inferior performance because of slow memory access
- Set interrupt mask properly:
  - Some OSs (e.g. Linux) allow for setting which CPUs are responsible for interrupt handling
  - Use CPUs that are not used by Order Book application

7.2 Functions

7.2.1 OBLatencyCheckPoint()

Description Reports the timestamp from an **OBBookUpdate** back to the accelerator card. The accelerator card can use this to provide latency metrics, see `OBLatencyRealtime()` and `OBLatencyTotal()`.

Prototype

```
void OBLatencyCheckPoint(OBBookUpdate U)
```

Arguments

`U`	Update

7.2.2 OBLatencySetAutoCheckPoint()

Description Automatically reports a timestamp back to the accelerator card, from one of the pre-set sampling points available via **OBAutoCheckPoint**. The accelerator card can use this to provide latency metrics, see `OBLatencyRealtime()` and `OBLatencyTotal()`.

Prototype

```
void OBLatencySetAutoCheckPoint(OBHandle *H,
OBAutoCheckPoint P)
```

Arguments

`H`	OB handle
`P`	Latency measurement point

7.2.3 OBLatencyRealtime()

Description Returns the latency metrics, as reported by the hardware to the OB software-side every 0.5 seconds. The values it returns are for the most recent measured 0.5-second window. Calling the function multiple times within a 0.5-second window will therefore result in the same values being returned. This function is fast to return, so can be called in performance-sensitive code.

Note:

> The metrics returned depend on the user reporting latency ticks back to the hardware using `OBLatencyCheckPoint()` or `OBLatencySetAutoCheckPoint()`.

Prototype

```
OBStatus OBLatencyRealtime(OBHandle *H, int
ThreadNumber, uint64_t *LatencyNSPtr)
```

Arguments

`H`	OB handle
`ThreadNumber`	Thread ID
`LatencyNSPtr`	Latency in nanoseconds

7.2.4 OBLatencyTotal()

Description Returns the latency metrics for an entire OB session: i.e. since the last reset or since the application start. The function accesses registers on the accelerator card, which is an expensive operation. It should therefore not be called excessively, and it should not be called from performance-sensitive code. The function can be called safely from any thread, but the caller should ensure that OB does not close during the call (again because it accesses registers on the card).

Note:

> The metrics returned depend on the user reporting latency ticks back to the hardware using `OBLatencyCheckPoint()` or `OBLatencySetAutoCheckPoint()`.

Prototype

```
OBStatus OBLatencyTotal(OBHandle *H, int
ThreadNumber, uint64_t *SumNSPtr, uint64_t
*SamplesPtr)
```

Arguments

`H`	OB handle
`ThreadNumber`	Thread ID
`SumNSPtr`	Total sum of latencies for all samples
`SamplesPtr`	Total number of samples

8. Format

See the header files for more details.

8.1 Variables

The following table provides the values of some OB variables:

Variable	Value	Description
OB_UNDEFINED_VALUE	**~((unit64_t)0)**	Undefined value
OB_MAX_INSTRUMENTS	**100**	Maximum number of instruments in super book
MAX_REGULATORY_FEEDS	**2**	Maximum number of regulatory feeds
GMAC_LOCAL_NAME_LENGTH	**32**	Market local instrument name maximum length

8.2 Enumerations

The following table provides the possible values for some OB lists:

Enumeration	Values	Description
OBStatus	**OB_STATUS_OK**	Success
	OB_STATUS_MALLOC_ERROR	Memory allocation error
	OB_STATUS_INIT_FAIL	Initialization failed
	OB_STATUS_PARTIALLY_INITIALIZED	Partially initialized
	OB_STATUS_NOT_FOUND	Instrument not found
	OB_STATUS_NOLINK	Line failure
	OB_STATUS_MISSING_DATAREF_MESSAGE	Missing data referential message
	OB_STATUS_SUBSCRIPTION_FAILED	Subscription failed
	OB_STATUS_STOP	Reading from the group is stopped

	OB_STATUS_PARAM_ERROR	Parameter error
	OB_STATUS_TOO_BUSY	API is too busy to complete the call
	OB_STATUS_CANCELLED	The request was cancelled
	OB_STATUS_DUPLICATE_SUBSCRIPTION	Duplicate subscription is detected
OBTradingStatus	**OB_TRADING_OPEN**	Trading is open
	OB_TRADING_EXTENDED	Extended market hours
	OB_TRADING_QUOTATION	Quotation only
	OB_TRADING_HALTED	Suspension / halt
	OB_TRADING_CLOSED	Trading is closed
	OB_TRADING_UNKNOWN	Unknown
	OB_TRADING_ERROR	Error
	OB_TRADING_ENUM_SIZE	Size of this enumeration list
OBAutoCheckPoint	**OB_AUTOCHECKPOINT_OFF**	Automatic checkpoint is switched off
	OB_AUTOCHECKPOINT_INPUT	Automatic checkpoint is at the input of the Order Book (when message arrives from GMAC)
	OB_AUTOCHECKPOINT_OUTPUT	Automatic checkpoint is at the output of the Order Book (when sending user update)

8.3 Data Structures

8.3.1 Order Book Update Header

Description Order Book update header.

The update header handles the action that should be taken when processing the update.

The Deep Book Actions are:

- **N** New order
- **U** Order update/delete/trade. There are 3 cases of use:
  - Trade only (uncrossing volume is passed as a trade) : `OrderOffset` is set to zero, `Price` and `Quantity` (negative) are set
  - Order update only: `OrderOffset` is set, `Price` is set to zero
  - Order update and trade: `OrderOffset` is set, `Price` and `Quantity` (negative) are set
- **E** Space expand
- **T** Trading status
- **O** OHLC
- **C** Computed OHLC
- **I** Imbalance
- **Y** Currency
- **S** Book status
- **F** There was an error when producing updates (usually out of buffer, user is not keeping up). There will be no more updates for this subscription, the user should call `OBGroupRemoveSubscription()` to let OB free the resources, and then subscribe to the instrument again using `OBGroupAddSubscription()`.

The Consolidated Book Actions are:

- **X** Change
- **B** BBO change

Structure **OBUpdateHeader**

The structure description is as follows:

Field	Type	Description
Link		LinkOBUpdateHeader
Action	char	Action to be taken.
***Subscription**	OBSubscriptionHandle	Order Book instrument subscription handle. Order Book subscription handle created by `OBGroupAddSubscription()` call
OriginalSequence	uint64_t	Sequence number of the message causing this update, ~0 if invalid
OriginalTimestamp	uint64_t	Timestamp reported by exchange, normalized to nanoseconds since UTC midnight
HWTimestamp	uint32_t	Hardware timestamp to allow latency measurements.
CPUTimetstamp	uint64_t	CPU Timestamp to allow software latency measurements
CPUTimetstampUpdateSent	uint64_t	The time the update was sent
CPUTimetstampIn	uint64_t	The time the message was read from the HW queue
CPUTimetstampOut	uint64_t	The time GMAC finished processing the message
BookThread	int	At which book thread was this update received
MulticastID	GMACMulticastID	GMAC multicast ID this updated is originated from

8.3.2 Order Book Update

Description Order Book update.

This structure is:

- Either a book update header structure of type **OBUpdateHeader**
- Or a deep book update structure of type **OBDeepBookUpdate**

- Or a consolidated book update structure of type **OBConsolidatedBookUpdate**

Structure **OBBookUpdate**

8.3.3 Deep Book Update

Description Deep book update.

Structure **OBDeepBookUpdate**

The structure description depends on the action, and is as follows:

`Action` = **N**

Field	Type	Description
Header	OBUpdateHeader	Update header
OrderOffset	offset	Order offset within Order Buffer. Offset Pointer for user's updates. Used in **OBDeepBookData** to specify position of an order **OBDeepBookOrder**. The position of an order can be computed as Order Buffer pointer + Offset.
PredecessorOffset	offset	Offset of the in order predecessor. Offset Pointer for user's updates. Used in **OBDeepBookData** to specify position of an order **OBDeepBookOrder**. The position of an order can be computed as Order Buffer pointer + Offset.
Quantity	uint32_t	Order quantity
Price	uint64_t	Order price, corrected by the price factor
Side	uint8_t	Order side Possible values: 1 Bid -1 Ask

`Action` = **U**

Field	Type	Description
Header	OBUpdateHeader	Update header
OrderOffset	offset	Order offset within Order Buffer. Offset Pointer for user's updates. Used in **OBDeepBookData** to specify position of an order **OBDeepBookOrder**. The position of an order can be computed as Order Buffer pointer + Offset.
UncrossFlag	uint8_t	When set to 1, the update is artificially uncrossing the book
Side	uint8_t	Order side Possible values: 1 Bid -1 Ask
DeltaQuantity	uint32_t	Quantity difference (trade quantity / uncrossing volume)
Price	uint64_t	Price (order price / trade price / uncrossing price)
TotalTradedVolume	uint64_t	Total traded volume for the day
UpdateFollowsFlag	uint8_t	Flag to indicate if the update has resulted in a temporary empty book

`Action` = **E**

Field	Type	Description
Header	OBUpdateHeader	Update header
MaxOrders	offset	New size of the order buffer. The OB framework have more orders than the present Order Buffer size, this message notifies user that the buffer size needs to be increased, reallocation call is necessary

`Action` = **T**

Field	Type	Description
Header	OBUpdateHeader	Update header
TradingStatus	uint32_t	New trading status

`Action` = **O**

Field	Type	Description
Header	OBUpdateHeader	Update header
Open	uint64_t	Opening price as reported by the exchange
High	uint64_t	Highest price as reported by the exchange
Low	uint64_t	Lowest price as reported by the exchange
Close	uint64_t	Closing price as reported by the exchange
TotalTradedVolume	uint64_t	Total traded volume for the day

`Action` = **Y**

Field	Type	Description
Header	OBUpdateHeader	Update header
Currency	char [4]	Currency update

`Action` = **C**

Field	Type	Description
Header	OBUpdateHeader	Update header
Open	uint64_t	Opening price, first encountered trade
High	uint64_t	Highest price encountered
Low	uint64_t	Lowest price encountered

`Action` = **I**

Field	Type	Description
Header	OBUpdateHeader	Update header

Field	Type	Description
CrossType	char	Cross type Possible values: 'O' Opening 'C' Closing 'H' Halted
Direction	char	Market side of the order imbalance Possible values: 'B' Buy 'S' Sell 'N' No imbalance 'O' Insufficient orders
Variation	char	Price variation code
Reference	uint64_t	Reference price
Far	uint64_t	Far price
Near	uint64_t	Near price
Paired	uint64_t	Volume for paired number of shares for the current reference price
Imbalance	uint64_t	Volume for not paired shared at current reference price

`Action` = **S**

Field	Type	Description
Header	OBUpdateHeader	Update header
BookStatus	char	Book status Possible values: 'C' Clean book: There are no gaps and the instrument book is synchronized with the live feed 'R' Refreshing book: The order book is going through a refresh cycle to synchronize the instrument book with the live feed 'S' Stalled book: The book is delayed (not up to date, but will soon be) due to a recovery task initiated by GMAC 'B' Best effort book: The instrument book has lost synchronization with the live feed (due to gaps or intra-day start). This approximation is the order

Field	Type	Description
		books' best effort for the instrument

`Action` = **F**

Field	Type	Description
Header	OBUpdateHeader	Update header
Reason	char	Failure reason code. Possible values: 'F' Buffer full 'I' Index mapping collision 'C' Symbol cleared by market

8.3.4 Deep Book Order

Description Single order.

Structure **OBDeepBookOrder**

The structure description is as follows:

Field	Type	Description
Quantity	uint32_t	Order quantity
Price	uint64_t	Order price, corrected by price factor
Next	offset	Offset of the (in-order) successor order. Offset Pointer for user's updates. Used in **OBDeepBookData** to specify position of an order **OBDeepBookOrder**. The position of an order can be computed as Order Buffer pointer + Offset.
Previous	offset	Offset of the (in-order) predecessor order. Offset Pointer for user's updates. Used in **OBDeepBookData** to specify position of an order **OBDeepBookOrder**. The position of an order can be computed as Order Buffer pointer + Offset.

8.3.5 Deep Book Data

Description Structure in which book data are given to the user.

Structure **OBDeepBookData**

The structure description is as follows:

Field	Type	Description
OrderBuffer	OBDeepBookOrder	Pointer to the Order Buffer
OrderBufferSize	int	Order buffer size
SuperBook	OBSuperBookData	Pointer to the Super Book
SuperBookIndex	int	
***Subscription**	OBSubscriptionHandle	Order Book instrument subscription handle
MarketID	int	Market ID, as specified in the configuration file
TradingStatus	uint32_t	Trading status code
LastTradePrice	uint64_t	Last trade price
LastTradeQuantity	uint32_t	Last trade quantity
LastOriginalTimestamp	uint64_t	Exchange timestamp of the last update on the book
LastOriginalSequence	uint64_t	Message sequence number of the last update on the book
TotalTradedVolume	uint64_t	Total traded volume for the day
PriceOpen	uint64_t	Opening price as reported by the exchange
PriceHigh	uint64_t	Highest price as reported by the exchange
PriceLow	uint64_t	Lowest price as reported by the exchange
PriceClose	uint64_t	Closing price as reported by the exchange

Field	Type	Description
ComputedPriceOpen	uint64_t	Price of the first encountered trade
ComputedPriceHigh	uint64_t	Highest seen price
ComputedPriceLow	uint64_t	Lowest seen price
Currency	char [4]	Instrument currency (according to ISO4217 + USX + GBX)
BookStatus	char	Book status Possible values: 'C' Clean book: There are no gaps and the instrument book is synchronized with the live feed 'R' Refreshing book: The order book is going through a refresh cycle to synchronize the instrument book with the live feed 'F' failure to refresh

8.3.6 Consolidated Book Update

Description Consolidated book update

Structure **OBConsolidatedBookUpdate**

The structure description depends on the action, and is as follows:

`Action` = **X**

Field	Type	Description
Header	OBUpdateHeader	Update header
Side	GMACSide	Order side. Possible values: 1 Bid -1 Ask
Index	OBConsolidatedBookIndex	Index into the quotes array in structure **OBConsolidatedBookData**
Order	OBConsolidatedBookOrder	Consolidated book order

`Action` = **B**

Field	Type	Description
Header	OBUpdateHeader	Update header
BBOIndex	OBConsolidatedBookIndex	Index into the quotes array in structure **OBConsolidatedBookData**
BBO	OBConsolidatedBBO	Consolidated BBO

8.3.7 Consolidated Book Order

Description Consolidated book order.

Structure **OBConsolidatedBookOrder**

The structure description is as follows:

Field	Type	Description
Quantity	uint64_t	Order quantity
Price	int64_t	Order price, corrected by price factor

8.3.8 Consolidated BBO

Description Best Indicator.

Consolidated feeds present information, from multiple markets, about which Bid and Ask is currently the national best, and/or FINRA best, etc ...

Structure **OBConsolidatedBBO**

The structure description is as follows:

Field	Type	Description
Index [2]	OBConsolidatedBookIndex	Venue ID with the best Bid(0), Ask(1)

8.3.9 Consolidated Book Data

Description User's copy of the book.

Contains Top of the Book information from various venues on the same feed.

Structure **OBConsolidatedBookData**

The structure description is as follows:

Field	Type	Description
void *BackgroundInfo		Pointer to feed dependent background info.
Venue	OBConsolidatedBookQuote	An array of TOB info [GMAC_MAX_VENUES]
BBO	OBConsolidatedBBO	List of Best Bid and Offer Indicators [GMAC_MAX_BESTS]

8.3.10 Consolidated Book Quote

Description Book quote.

Structure **OBConsolidatedBookQuote**

The structure description is as follows:

Field	Type	Description
Order [2]	OBConsolidatedBookOrder	Top of the book orders Bid(0), Ask (1)

8.3.11 Super Book Data

Description Structure in which super book data are given to the user.

Structure **OBSuperBookData**

The structure description is as follows:

Field	Type	Description
***OB**	OBHandle	Pointer to the OB handle
***DeepBooks**	OBDeepBookData	Pointer to the deep book handle [OB_MAX_INSTRUMENTS]
***SuperBookLink**	OBSuperBookLink	Pointer to additional super book linking buffers [OB_MAX_INSTRUMENTS]
Instruments	int	Number of instruments in this super book
Asks	OBSuperBookLink	Dummy super book linking structure Asks->Next* points to the best Ask in the super book
Bids	OBSuperBookLink	Dummy super book linking structure Bids->Next* points to the best Bid in the super book

8.3.12 Super Book Link

Description Provides links to previous and next order in the super book.

Structure **OBSuperBookLink**

The structure description is as follows:

Field	Type	Description
PrevIndex	uint8_t	Super Book index of the previous order in the super book

Field	Type	Description
PrevOffset	offset	Super Book offset of the previous order in the super book. Offset Pointer for user's updates. Used in **OBDeepBookData** to specify position of an order **OBDeepBookOrder**. The position of an order can be computed as Order Buffer pointer + Offset.
NextIndex	uint8_t	Super Book index of the next order in the super book
NextOffset	offset	Super Book offset of the next order in the super book. Offset Pointer for user's updates. Used in **OBDeepBookData** to specify position of an order **OBDeepBookOrder**. The position of an order can be computed as Order Buffer pointer + Offset.

8.3.13 Extra Parameters

Description Optional parameters to provide to `OBOpenEx`() enabling additional operations when opening OB.

Structure **OBOpenExtraParams**

The structure description is as follows:

Field	Type	Description
GMACReferentialReceivedCallback	OBOnGMACReferentialReceivedCallback_t	
Void *GMACReferentialReceivedCallbackPrivate		

8.4 Argument Types

The following table provides the argument types description:

Enumeration	Description
GMACPrice	uint64_t GMAC price format
GMACQuantity	uint64_t

	GMAC quantity format
GMACHandle	Opaque structure
	see GMAC API Reference Guide
OBHandle	Opaque structure
OBSecurityName	Type GMACLocalName
	see GMAC API Reference Guide
GMACLocalName	see GMAC API Reference Guide
GMACMulticastID	see GMAC API Reference Guide
GMACStats	see GMAC API Reference Guide
Config	See the Configuration Parser API Reference Guide and the GMAC Configuration Guide.
OBMarketNames	Type GMACMarketNames
	see GMAC API Reference Guide
GMACEISIN	See GMAC API Reference Guide
OBErrorHandler	`OBStatus (*OBErrorHandler)(OBHandle *OB, OBStatus StatusCode, const char *Function, char *Msg)`
GMACErrorHandler	See GMAC API Reference Guide
OBTimeoutHandler	`OBStatus (*OBTimeoutHandler)(OBHandle *OB, void *Private, int MarketID, int MulticastID, uint32_t IP, uint16_t Port, char *Msg)`
GMACMulticastMapping	See GMAC API Reference Guide
OBSubscriptionHandle	Opaque structure
	Order book subscription handle created by `OBGroupAddSubscription()`
OBSubscriptionGroupHandle	Opaque structure
	Order book group subscription handle created by `OBGroupOpen`()
OBInstantCallback_t	The callback can be registered when calling `OBGroupOpen`().
	The function will be called by one of the threads processing the book. The function may be called simultaneously.
	Any extensive computation in this function may cause packet drops.

It is not recommended to use this callback for trading algorithm, use OBUpdateCallback_t instead.

Possible use of this function can be UDP distribution.

It is not necessary to call `OBGroupRelease`().

If the function returns zero, data will not be passed to the subscription group queue, and it will not be possible to read it using `OBGroupRead`(). If the function returns non-zero value, data will be passed forward, as if the callback wasn't there.

```
int (*OBInstantCallback_t)(OBBookUpdate Update, void *Private)
```

OBUpdateCallback_t

This mechanism allows user to use callback interface for receiving book updates.

It is given that this function will be executed only from thread calling `OBGroupWait()`, and therefore there are not race-condition issues.

It is not necessary to call `OBGroupRelease`().

If the function returns non-zero value, the function `OBGroupWait()` will return.

```
int (*OBUpdateCallback_t) (OBSubscriptionGroupHandle*Group, OBBookUpdate UpdateList, unsigned int
Count, void *Private)
```

OBSubscriptionCallback_t

```
void (*OBSubscriptionCallback_t)(void *Private)
```

GMACSide

See GMAC API Reference Guide

OBConsolidatedBookIndex

uint8_t

index into the quotes array in **OBConsolidatedBookData**

OBOnGMACReferentialReceivedCallback_t

Sets the OB callback to call when GMAC data referential are received.

```
int (*OBOnGMACReferentialReceivedCallback_t)( OBHandle *OB, GMACHandle *GMAC, GMACMulticastID
MulticastID, const char* const Data, size_t const DataLength, void *Private)
```

Arguments	`OB`	OB handle
	`GMAC`	GMAC handle corresponding to the GMAC instance where the message has been received

	`MulticastID`	GMAC multicast ID indicating on which GMAC channel the message has been received
	`Data`	GMAC referential message as a byte array, starting with a GMACMessageHeader
	`DataLength`	Total number of bytes in the message, including the GMACMessageHeader and the extensions
	`Private`	Private pointer that was registered with the callback
Returns	0 if the referential message is to be processed by OB	
	1 if the referential message is to be ignored by OB	

9. Terminology

Channel A logically separated stream of data within GMAC.

A channel contains one or more multicasts from one or more markets which are reordered, normalized and directed to a DMA queue.

Number of market logical channels is fixed to **256**.

Consolidated Book Regulatory consolidated order book.

Deep Book Order book.

DMA Queue A group of channels delivered from GMAC directly to a single CPU for immediate processing in a user's application.

Channels may have data from several different markets.

Currently GMAC supports **two** DMA queues.

Exchange A stock exchange; transmits data and receives order transactions on one or more markets.

Feed Market data transmitted on a single physical line (Ethernet connection). Feeds consist of one or more multicasts.

GMAC Generic Market capture ACcelerator, combined hardware and software solution which receives market data, reorders and normalizes it and delivers it directly to the CPU for processing in the user's application.

Hardware The hardware refers to the Celoxica's FPGA-based Accelerator Card.

Hardware Filtering Instrument-filtering market data so that only the required market data is transmitted to the GMAC software-side.

Market Transmits data on one or more duplicated feeds and receives order transactions via TCP connections.

Multicast Market data transmitted from a single source IP address and port.

OB Order Book.

Recovery Process of recovering of dropped packets.

Super Book Different order books are linked together to form a super book.