Celoxica
pure acceleration

UltraBook UB

User Guide

About This Document

This document describes the UB user 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

Content

Table of Figures

Revisions

Revision	Date	Description of Changes
R2013-1.0	15 JAN 2013	Release R2013-1.0 - Added details to sections Quantity Type and Raw Exchange Specific Data for ARCAOptions
R2012-8.1	20 DEC 2012	Release R2012-8.1 - UB supports PHLX TOPO - Updated Shared Memory section - Updated the Symbology section - Added details to config.markets.market description
R2012-8.0	28 NOV 2012	Release R2012-8.0 - Added details to the trading status and integrity management - Added Shared Memory Allocation section - Added UpdateInfo.exchangeSequenceNumber - Updated CSM market specifics - Updated ARCAOptions market specifics - Updated View class definition - Updated Instrument class definition
R2012-7.1	31 OCT 2012	Release R2012-7.1 - Added ARCAOPTIONS market specifics - Added Control Configuration Settings section - Updated CSM market specifics - Updated Functional Description section - Updated User API section
R2012-ub-beta	17 SEP 2012	Release R2012-ub-beta - OBv2 is renamed UltraBook UB - Added Market Specifics section - Added Usage sub-section in the User API section - Updated Functional Description section - Updated Technical Architecture section - Updated Configuration section - Updated Structures section in the User API section
R2012-5.ob2-alpha	10 AUG 2012	Initial release

1. Functional Description

1.1 Recommended Reading

This document should be read in conjunction with:

- The Celoxica Manager User Guide
- The PCI Express Card Hardware Installation Guide
- The PCI Express Card Software Installation Guide

1.2 Overview

Celoxica’s generic and data-normalized UltraBook UB is an FPGA-based solution providing processed market data stored in a book and high-performance communication with a client software application.

UB normalizes order-driven and book-driven feeds and provides a unified interface for reading orders for different instruments. It supports multiple markets and multiple levels depth.

1.2.1 Symbology

UB provides a flexible symbology, where every instrument can be addressed in multiple ways.

Currently UB needs to determine the mapping between symbol names and indexes from the reference data received directly from the feed. In a future release, a facility for the user to specify a static data referential file will be added.

UB supports several identifier schemes to uniquely identify an instrument. The availability of the schemes per instrument depends on the market.

The list of supported schemes is as follows:

key	
Description	The UB internal unique identifier. This scheme is supported for all feeds. This identifier is provided for all instruments. This identifier is unique per UB daemon and remains valid during the whole UB daemon lifespan.
Usage	key://<keyNumber>
Example	key://421048290333

index	
Description	The hexadecimal form of the instrument index provided by the exchange. This identifier is unique per <marketName> and remains valid until the market changes it.
Usage	index://<index>.<marketName>
Example	index://001e03e4.arca

local	
Description	The instrument name as provided by the exchange. This identifier is unique per <marketName> and remains valid until the market changes it.
Usage	local://<localName>.<marketName>
Example	local://aapl.arca

osi

Description	The OCC Options Symbology Initiative name. This scheme is supported for options only. This identifier is unique per instrument and remains valid during the whole instrument lifespan.
Usage	osi://<osiName>.<marketName>
Example	osi://aapl 120201P00012340.arca

fsi

Description	Celoxica specific naming for Futures. This scheme is supported for futures only. This identifier is unique per instrument and remains valid during the whole instrument lifespan. It consists of the following components: • Future root symbol 6 bytes • Expiry year 2 bytes • Expiry month 2 bytes • Expiry day 2 bytes
Usage	fsi://<fsiName>.<marketName>
Example	fsi://aapl 120201.arca

Note:

<marketName> refers to name of the market used when setting *config.markets.market*.

Other schemes might be supported in future UB releases.

1.2.2 Instrument Static Information

UB provides instrument industry standard static and normalized information:

- The instrument identifiers depending on the available identifier schemes
- The ISO 10962 CFI code
- The instrument type as provided by the exchange

UB also provides raw referential data as provided by the exchange.

UB automatically registers all referential data received from the exchange. The user subscribes to an instrument to get levels, trades, prices and imbalances. UB will only process market data related to a symbol if the user subscribes to that symbol.

1.2.3 Levels

The order book is maintained from level 0 (top of book) to level n where n <=4 and provides the bid and offer quantity per level.

If an exchange provides different quantity types such as customer or all or none quantity type, UB will maintain as many order books as there are quantity types available.

1.2.4 Trades

UB provides the trades received from the exchange when available.

1.2.5 Prices

UB provides the order book OHLC (Open / High / Low / Close) and settlement prices information when available.

1.2.6 Imbalances

UB provides the order book imbalance information when available.

1.2.7 Trading Status

The security and market trading statuses generic forms are provided when available. Note that such statuses often vary significantly between feeds, so it is not always possible to provide a full and generic mapping. UB therefore provides market specific statuses for the user's reference.

UB automatically registers all instrument trading statuses received from the exchange even for non-subscribed instruments; as a consequence, if the user subscribes to an instrument intra-day, he will get the correct instrument trading status.

Note:

For a given instrument, UB registers the trading status only if there was at least one referential message received prior to the trading status message.

1.2.8 Integrity

The instrument and market integrity is also made available to the client application. This integrity status notifies the state of the instrument book and the market:

- **Clean**

  There is no error, no data loss.

- **Best-effort**

  The instrument book / market information is probably not accurate because of gaps and data loss; a recovery may be in progress to update the order book / market integrity.

- **Stalled**

  The instrument book / market information flow has stalled e.g. one of the multicasts on the market has timed out.

This feature is partially supported:

- The instrument integrity is always clean
- The market integrity is set clean at startup, becomes best-effort in case of a sequence gap; however, no recovery process is implemented yet.

1.3 Shared Memory API

UB handles the market data from the exchanges and writes the appropriate market events to shared memory. The shared memory is made accessible to the client application which can therefore read it via a shared memory API.

UB has been designed so that the shared memory order book can be accessed by multiple user readers – multiple threads - as illustrated below.

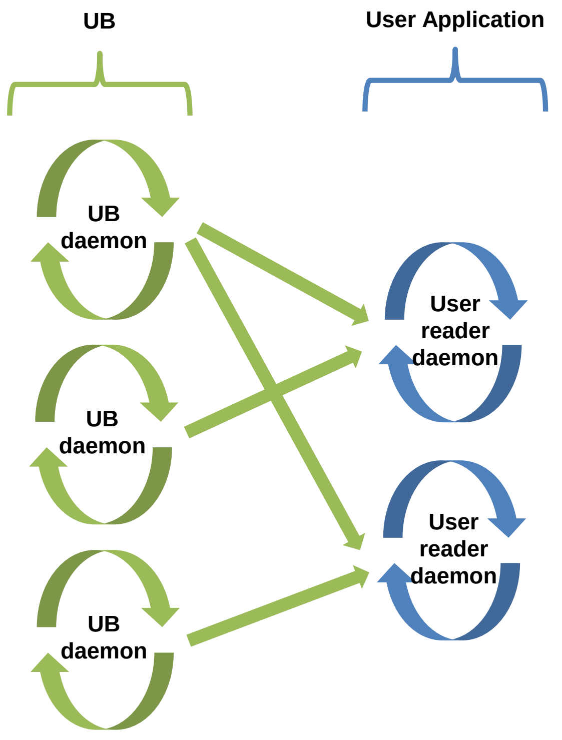


Figure 1 – Multiple user readers

The client application gets access to:

- **Order book tables**

  A complete order book per symbol is made available through shared memory. This API is an easy and quick way for the client application to get access to order books.

  The user gets the most recent live information updates by using this API.

- **Update queue per reader**

  The client application defines a set of reader slots from which it is notified if there was a change in the order book. These notifications do not accumulate and are cleared when the user reads the change list.

  The change list contains flags about what has changed in the book. Change flags can stack up over time in the notification list however the order of the changes is not maintained.

  Several types of changes in the order book are flagged:

  - Top of book (level 0) change per quantity type
  - Levels other than top of book change per quantity type
  - Trades change
  - Prices change
  - Imbalances change
  - Instrument Status change
  - Market Status change
  - Instrument integrity
  - Market integrity

- **Symbol directory**

  The client application reads the symbol directory to get static information of instruments.

  The client application is able to retrieve the most up-to-date exchange specific referential data as well as standard and normalized instrument characteristics.

Order books for regulatory and consolidated feeds have the same client API presentation as primary feeds i.e. a separate order book per venue.

1.4 Filtering

Software filtering is natively implemented in UB for all feeds as UB automatically filters the symbols based on the subscription.

Note:

UB does not support hardware symbol filtering yet.

1.5 Recovery

Recovery is handled by UB automatically. Books are always initiated with best-effort status, so UB detects this and triggers the snapshot recovery process. During the snapshot process, the user is presented with all updates received from the market even if the book is in best-effort status. Once the snapshot has been processed successfully, the book switches to clean status. If a gap is detected the book switches back to best-effort status and the snapshot process is triggered again.

The following types of recovery are or will be available:

- **Natural refresh**

  The order book is built using only live data. This provides a quick and easy way of order book refresh.

  This feature is supported.

- **Snapshot recovery**

  The picture of the order book is refreshed using market snapshot (when available). This is useful for mid-day starts and for non-liquid instruments.

  This feature is not supported yet.

- **Gap skip recovery**

The order book may be crossed when some data are missed. UB is able to cope with such cases by using the uncrossing mechanism: this is a scheme whereby crossing entries are deleted from the book automatically.

This feature is not supported yet.

2. Configuration

This section describes the UB configuration file and all available configuration options.

UB is configured using an XML-style '.cfg' file. When the UB initializes, it loads this configuration file. See section 2.1 for more details.

Example configuration files are shipped with your software. These files were created using reliable, fully tested settings. Celoxica highly recommends creating backup copies of any configuration files before editing.

One configuration file must be defined per UB instance; it allows the user to configure:

- Some general settings; see section 2.2.1 for more details;
- Some logger daemon specific settings; see section 2.2.2 for more details;
- Some UB server process specific settings, see section 2.2.3 for more details;
- Some market specific settings, in particular details for the live multicasts; see section 2.2.4 for more details;
- Market data handlers; see section 2.2.5 for more details.

2.1 Configuration File Schema

The basic structure of an UB configuration file is shown in the following schema. Tags have to be defined in the order as shown. This schema shows tags but not values.

```
<config>
      <ade>
            <cpu>
      <ade>

      <cpu-other>
      <verbosity>
      <daemonize>

      <logger>
            <key>
            <file>
            <size>
      </logger>

      <control>
            <name>
            <client-timeout>
      </control>

      <markets>
            <market>
                  <sessions>
                        <session>
                              <channels>
                                    <!- Live multicasts ->
                                    <channel>
                              </channels>

                              <references>
                                    <!- Security definitions channel ->
                                    <channel>
                              </references>

                              <recovery>
                                    <!- Recovery channel ->
                                    <channel>
                              </recovery>
                              <snapshots>
                                    <!- Snapshots channel ->
                                    <channel>
                              </snapshots>
                        </session>
                  </sessions>
            </market>
      </markets>

      <handlers>
            <handler>
            </handler>
      </handlers>
</config>
```

2.2 Configuration Settings

2.2.1 General Configuration Settings

These configuration settings are general and not market specific.

2.2.1.1 config

Description	This tag initiates a body for UB configurations to be made in a file. This tag is at the top of the file.
Usage	`<config>`

2.2.1.2 config.ade

Description	This tag initiates a body for ADE configurations to be made.
Usage	`<ade>`

2.2.1.3 config.ade.cpu

Description	Sets the CPU number, starting from 0.	
Usage	`<cpu> CPU </cpu>`	
Parameters	`CPU`	Integer
Example	`<cpu> 0 </cpu>`	

2.2.1.4 config.verbosity

Description	Sets the verbosity of UB.	
Usage	`<verbosity> verbosity </verbosity>`	
Parameters	`verbosity`	String. Possible values are: • error • warning • info • debug
Example	`<verbosity> error </verbosity>`	

2.2.1.5 config.cpu-other

Description	Specifies the CPU number on which other non-latency-critical Celoxica threads are allowed to run, as a bitmask. Such threads include the network emulation thread, which enables the accelerator card to exchange non-offloaded traffic with the OS network stack or the UB snapshot management thread.	
Usage	`<cpu-other> CPU </cpu-other>`	
Parameters	`CPU`	Integer
Example	`<cpu-other> 0 </cpu-other>`	

2.2.1.6 config.daemonize

Description	Specifies if UB is daemonized (the process is running on the background) or not. When enabled the executable will make the daemon run on the background and the console will be returned back to the user. The parameter must be enabled to make the UB daemon be managed by the Celoxica manager.
Usage	`<daemonize> daemon </daemonize>`
Parameters	`daemon` Boolean
Example	`<daemonize> true </daemonize>`

2.2.2 Logger Configuration Settings

The logger configuration allows the setting of parameters for UB to use the logger daemon.

2.2.2.1 config.logger

Description	This tag initiates a body for the logger configurations to be made.
Usage	`<logger>`

2.2.2.2 config.logger.key

Description	Sets the logger daemon shared memory key.	
Usage	`<key>Key</key>`	
Parameters	`Key`	integer
Example	`<key>11</key>`	

2.2.2.3 config.logger.file

Description	Sets the default log file if the logger daemon is not present.	
Usage	`<file>File</file>`	
Parameters	`File`	File path. Default is /dev/stderr
Example	`<file>/tmp/log.log</file>`	

2.2.2.4 config.logger.size

Description	Sets the size of the shared memory if not created.	
Usage	`<size>Size</size>`	
Parameters	`Size`	Log size expressed in bytes
Example	`<size>5242880</size>`	

2.2.3 Control Configuration Settings

These configuration settings allow the user to define the UB server process.

2.2.3.1 config.control

Description This tag initiates a body for the UB server process configurations to be made.

Usage `<control>`

2.2.3.2 config.control.name

Description Sets the server process name.

Usage `<name> name </name>`

Parameters	`name`	String

Example `<name> UB_server_1 </name>`

2.2.3.3 config.control.client-timeout

Description Sets the time in seconds to wait before the server disconnects a silent client.

Usage `<client-timeout> timeout </client-timeout>`

Parameters	`timeout`	Integer. Default is 10 seconds. Timeout is deactivated if set to 0.

Example `<client-timeout> 15 </client-timeout>`

2.2.4 Market Configuration Settings

These configuration settings are market specific.

2.2.4.1 config.markets

Description This tag initiates a body for market configurations to be made.

Usage `<markets>`

2.2.4.2 config.markets.market

Description Configures a market with integer id and string name used to identify the market.

Usage `<market> id marketName protocol`

Parameters	`id`	Id of the market. Integer.
	`marketName`	Name of the market. This is the name used in the symbology.
	`protocol`	Protocol to be used. See section 6 for details on the settings to use for each market.

Example	`<market> 0 cboe_strategy CSM`
	`<market> 1 cboe_nonstrategy CSM`

Notes	Most of markets ensure the symbol index uniqueness at the market level. However, some markets guarantee the symbol index uniqueness at different layers, for instance per market / source session or per market / instrument type (strategy / non strategy). In such a case, the markets must be configured to ensure the symbol index is unique per market. Refer to the example configuration files shipped with UB.

2.2.4.3 config.markets.market.sessions

Description This tag initiates a body for session configurations to be made.

Note:

This configuration node is originally intended to separate recovery channels from other channels: this feature is not implemented yet.

A single session per market will therefore have to be configured.

Usage `<sessions>`

2.2.4.4 config.markets.market.sessions.session

Description Configures a session with integer id and string name used to identify the session.

Usage `<session> id name`

Parameters	`id`	Id of the session
	`name`	Name of the session

Example `<session> 0 Test_Market_Data`

2.2.4.5 config.markets.market.sessions.session.channels

Description This tag initiates a body for channels configurations to be made.

Usage `<channels>`

2.2.4.6 config.markets.market.sessions.session.channels.channel

Description Configures the list of live multicast channels.

Usage `<channel> localChanId numQueue srcPlugin interface srcIP:port:type </channel>`

Parameters	`localChanId`	The local id for this channel, must be unique for the market
	`numQueue`	The DMA queue number which this channel is delivered, can be set to 0 only
	`srcPlugin`	The source for this channel, set to `lldt` when using the accelerator card
	`interface`	The physical interface on the accelerator card where the channel is received. If required, a particular interface on a VLAN can be specified.
	`srcIP:port`	The multicast IP address and port for this channel
	`type`	The type of connection for the channel, t for TCP and u

	for UDP

Example `<channel> 0 0 lldt ac0 224.0.62.2:30001:u </channel>`

```
Interface on VLAN 444:
<channel> 0 0 lldt ac0.444 224.0.62.2:30001:u
</channel>
```

2.2.4.7 config.markets.market.sessions.session.references

Description This tag initiates a body for security definition configurations to be made.

Usage `<references>`

2.2.4.8 config.markets.market.sessions.session.references.channel

Description Configures the channel from which security definitions are received.

Usage `<channel> localChanId numQueue srcPlugin interface srcIP:port </channel>`

Parameters

`localChanId`	The local id for this channel, must be unique for the market
`numQueue`	The DMA queue number which this channel is delivered, can be set to 0 only
`srcPlugin`	The source for this channel, set to `lldt` when using the accelerator card
`interface`	The physical interface on the accelerator card where the channel is received. If required, a particular interface on a VLAN can be specified.
`srcIP:port`	The multicast IP address and port for this channel
`type`	The type of connection for the channel, t for TCP and u for UDP

Example `<channel> 0 0 lldt ac0 224.0.62.2:30001:u </channel>`

```
Interface on VLAN 444:
<channel> 0 0 lldt ac0.444 224.0.62.2:30001:u
</channel>
```

2.2.4.9 config.markets.market.sessions.session.recovery

Description This tag initiates a body for recovery (dropped packets recovery on supported markets) configurations to be made.

Gap recovery is not supported yet.

Usage `<recovery>`

2.2.4.10 config.markets.market.sessions.session.recovery.channel

Description Configures the recovery channel.

Usage `<channel> localChanId numQueue srcPlugin interface srcIP:port </channel>`

Parameters

`localChanId`	The local id for this channel, must be unique for the market
`numQueue`	The DMA queue number which this channel is delivered, can be set to 0 only
`srcPlugin`	The source for this channel, set to `lldt` when using the accelerator card
`interface`	The physical interface on the accelerator card where the channel is received. If required, a particular interface on a VLAN can be specified.
`srcIP:port`	The multicast IP address and port for this channel
`type`	The type of connection for the channel, t for TCP and u for UDP

Example

```
<channel> 0 0 lldt ac0 224.0.62.2:30001:u
</channel>
```

```
Interface on VLAN 444:
```

```
<channel> 0 0 lldt ac0.444 224.0.62.2:30001:u
</channel>
```

2.2.4.11 config.markets.market.sessions.session.snapshots

Description This tag initiates a body for snapshots configurations to be made.

Snapshot recovery is not supported yet.

Usage

```
<snapshots>
```

2.2.4.12 config.markets.market.sessions.session.snapshots.channel

Description Configures the snapshot channel.

Usage

```
<channel> localChanId numQueue srcPlugin interface
srcIP:port </channel>
```

Parameters

`localChanId`	The local id for this channel, must be unique for the market
`numQueue`	The DMA queue number which this channel is delivered, can be set to 0 only
`srcPlugin`	The source for this channel, set to `lldt` when using the accelerator card
`interface`	The physical interface on the accelerator card where the channel is received. If required, a particular interface on a VLAN can be specified.
`srcIP:port`	The multicast IP address and port for this channel
`type`	The type of connection for the channel, t for TCP and u for UDP

Example

```
<channel> 0 0 lldt ac0 224.0.62.2:30001:u
</channel>
```

```
Interface on VLAN 444:
```

```
<channel> 0 0 lldt ac0.444 224.0.62.2:30001:u
</channel>
```

2.2.5 Handler Configuration Settings

These configuration settings allow the user to define market data handlers.

2.2.5.1 config.handlers

Description	This tag initiates a body for market data handler configurations to be made.
Usage	`<handlers>`

2.2.5.2 config.handlers.handler

Description Configures a handler with integer id and string name used to identify the handler.

This handler must be specified when the user wants to use the UltraBook User API.

The available market data handlers are:

- “OrderBook” Builds the order books
- “Dump” Prints the UB internal events generated by the market parsers. This handler is intended for trouble-shooting/debugging purposes.

Usage `<handler> id name </handler>`

Parameters

`id`	Id of the handler
`name`	Name of the handler

Example

```
<handler> 0 OrderBook </handler>
<handler> 1 Dump </handler>
```

3. User API

UB ships as a set of dynamic libraries, which the user code can interface to via the single and simple client API it exposes. A “hello world” example is shipped in the UB package, to illustrate the basic usage of the API. This is described in section *Shipped Examples*.

3.1 Class Diagram

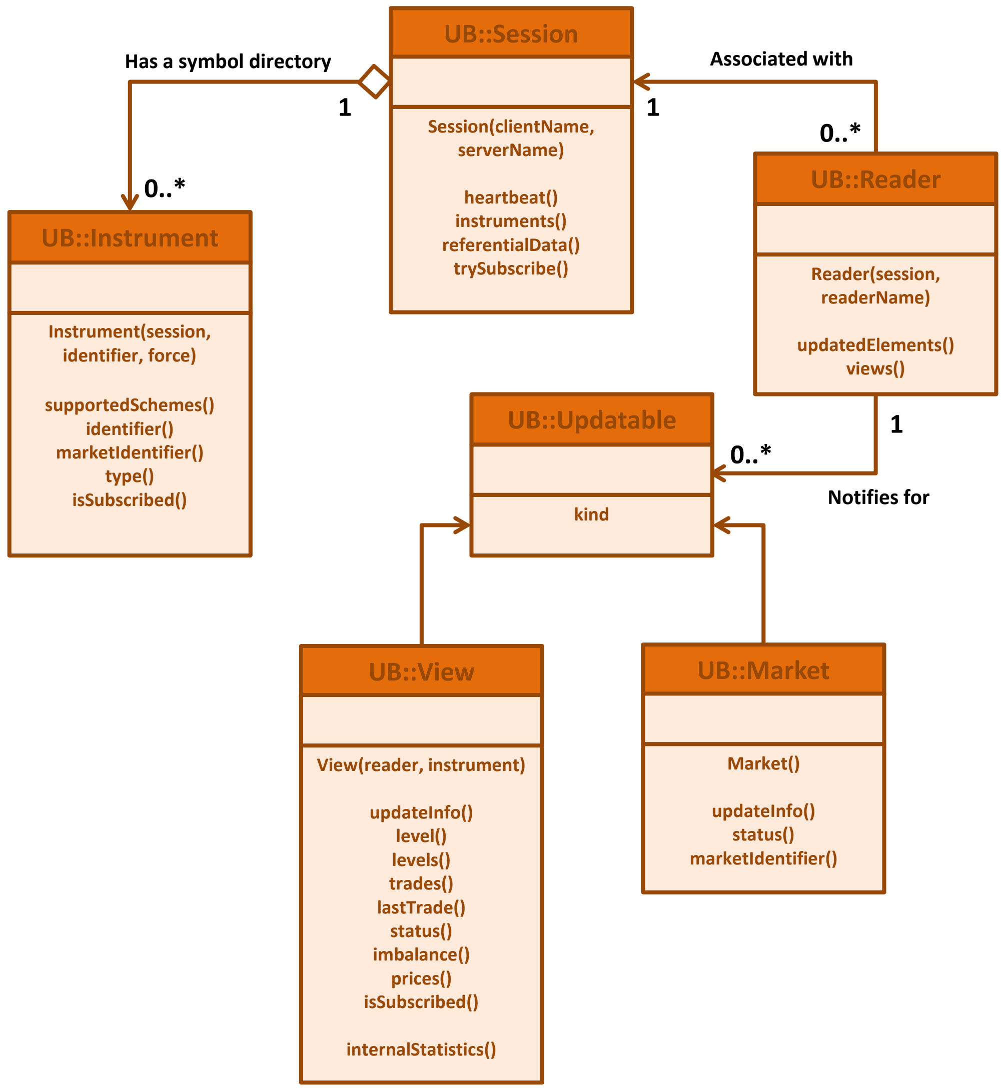


Figure 2 – Class diagram

3.2 Class Definitions

3.2.1 Session

Description An object of class Session type connects the client application to an order book.

The construction of a Session object in the client application does what is necessary to register the client and open the shared memory segments.

The Session object provides a method to access an instrument index table where all the known instruments and their static information are stored. It also provides a method to retrieve the raw referential data as provided by the market feeds.

An exception is thrown if the server cannot be reached or if an error occurred while communicating with the server.

The session implementation class is allocated on the heap when constructed and freed when the last pointer to it is destroyed.

The Session object can be allocated either on the stack or on the heap.

The copy constructor is enabled.

Copying a Session object only implies a copy of the pointer to the session implementation class.

Class UB::Session

Constructor `Session(clientName, serverName)`

`clientName`

client process name

`serverName`

UB daemon name as defined using configuration node *config.control.name.*

Exceptions `ConnectionFailed`

Connection failed with UB daemon.

Operators N/A

Methods `instruments()`

Returns the most up-to-date list of instruments.

`heartbeat()`

Returns a Boolean flagging whether the connection with the server is alive or not. The method is time consuming because of the heartbeat exchange between the client and the server; it should therefore not be called in the performance loop.

`referentialData(instrument)`

Returns the most up-to-date exchange specific data for the given instrument.

See structure *Instrument* for more details.

`trySubscribe()`

Tries to subscribe to any pre-subscribed instruments and corresponding views. This method has to be called regularly as the list of instruments may change with live referential updates provided by the exchange.

See section 3.3.1 for more details.

3.2.2 Reader

Description An object of class Reader type manages order book views and is notified of any order book change.

A Reader object is a thin wrapper around what is needed to be able to notify a given client that an order book has been modified. The Reader object provides the user application with the means to get a set of all modified order books and market trading status since the last time it checked for changes.

The construction of a Reader object requires a session, and will negotiate to get a free notification queue in the shared memory. An exception is raised if the application runs out of free queues.

The reader implementation class is allocated on the heap when constructed and freed when the last pointer to it is destroyed.

The Reader object can be allocated either on the stack or on the heap.

The copy constructor is enabled.

Copying a Reader object only implies a copy of the pointer to the reader implementation class.

Class UB::Reader

Constructor `Reader(session, readerName)`

`session`

Session identifier.

`readerName`

Reader name assigned by the client application.

Exceptions `ReaderUnavailable`

A free notification queue in the shared memory is not available.

Operators N/A

Methods `updatedElement()`

Returns a pointer to the next updated object of class View type or Market type since the last call.

`View::updateInfo()` method has to be called on the returned object of class View type to indicate that the last updates on an order book view have been read; otherwise the View object will not be returned again by `Reader::updatedElement()` even if additional updates are received.

`Market::updateInfo()` method has to be called on the returned object of class Market type to indicate that the last updates on a market have been read; otherwise the Market object will not be returned again by `Reader::updatedElement()` even if additional updates are received.

See sections 3.3.2 and 3.3.3 for more details.

`views()`

Returns all the views currently associated with this reader.

3.2.3 Instrument

Description An object of class Instrument type provides static information for an instrument using methods.

An Instrument object is a thin wrapper around one entry in the instrument index table.

The construction of an Instrument object requires a session and an instrument identifier. The instrument identifier must follow a scheme format as described in section *Symbology.*

The Instrument object provides various methods to get the instrument static information.

The instrument implementation class is allocated on the heap when constructed and freed when the last pointer to it is destroyed.

The Instrument object can be allocated either on the stack or on the heap.

The copy constructor is enabled.

Copying an Instrument object only implies a copy of the pointer to the instrument implementation class.

Class UB::Instrument

Constructor `Instrument(session, identifier, force)`

`session`

Session identifier.

`identifier`

Instrument identifier.

`force`

Optional boolean parameter to force the pre-subscription to an instrument if the instrument is not referenced by UB yet.

A referenced instrument will be subscribed upon its creation whatever the `force` value.

Default is false.

See section 3.3.1 for more details.

Exceptions `UnsupportedScheme`

The scheme is not available for this instrument.

`UnknownBook`

The instrument is not known from the instrument library.

Operators ==

!=

For convenient comparisons between Instrument objects.

Methods `SupportedSchemes()`

Returns the list of schemes for this instrument.

`identifier(scheme)`

Returns the instrument identifier using the scheme format for the specified scheme e.g. osi://aapl 120201P00012340.arca if `scheme` = osi.

The default scheme is ‘key’.

`marketIdentifier()`

Returns the market identifier for this instrument e.g. arcaox (Arca Option) for instrument which identifier is osi://aapl 120201P00012340.arcaox using the scheme osi.

`type()`

Returns the instrument type.

See structure *InstrumentType* for more details.

`isSubscribed()`

Returns a boolean to indicate whether the instrument is subscribed (true) or only pre-subscribed (false).

An instrument may only be pre-subscribed if its creation has been forced.

The method `Session::trySubscribe()` has to be called to try to subscribe to pre-subscribed instruments.

3.2.4 Updatable

Description An object of class Updatable type is an object returned by `Reader::updatedElements()`.

The actual updatable object is provided using `Updatable::kind` field (see enumeration *UpdatableKind* for more details).

Class UB::Updatable

Constructor `Updatable()`

Exceptions N/A

Operators	N/A
Methods	N/A

3.2.5 View

Description

An object of class View type provides the last live information updates on an order book.

A View object is an updatable object of kind UPDATABLE_VIEW (see enumeration *UpdatableKind* for more details).

A View object is a thin wrapper around an order book from a reader point of view. The View object wraps the order book itself and also reader-specific update flags to allow the client application to check what has changed since the last time it has checked the order book on the reader associated with the view.

A View object also provides a set of methods to read the dynamic data from the order book in shared memory. These methods provide the dynamic data that have been changed since the last time the client application checked the order book on the reader associated with the view.

The construction of a View object requires a Reader object to associate to, and an Instrument object to be able to get the corresponding order book.

The view implementation class is allocated on the heap when constructed and freed when the last pointer to it is destroyed.

The View object can be allocated either on the stack or on the heap.

View objects cannot be copied as pointers are returned to the View objects that are associated to a reader: the copy constructor is disabled.

Class UB::View

Constructor `View(reader, instrument)`

`reader`

Reader to be associated to the view.

`instrument`

Instrument to retrieve the order book information.

Exceptions N/A

Operators N/A

Methods `level(level, quantityType)`

Returns the data of an order book for a given level (both sides) and a given quantity type (QUANTITY_TYPE_NORMAL is the default quantity type).

The function ensures that the data being returned have not been modified while being read in shared memory.

See enumeration *QuantityType* for the possible quantity types.

`levels(quantityType)`

Returns data of all levels (both sides) of an order book for a given quantity type (QUANTITY_TYPE_NORMAL is the default quantity type).

The function ensures that the data being returned have not been modified while being read in shared memory.

See enumeration *QuantityType* for the possible quantity types.

`updateInfo()`

Returns the last updates on an order book view. Notifications are cleared when the user reads the change list.

See structure *UpdateInfo* for more details.

`trades()`

Returns the list of as many unread trades as possible.

The function does not guarantee to return all trades if it is not frequently called: the oldest trades are dropped when the trade queue is full. The most recent trades are therefore always made available.

See structure *TradeList* for more details.

`lastTrade()`

Returns the latest received trade.

If no trade has been received, the returned trade has a type TRADE_TYPE_INVALID.

`instrument()`

Returns the instrument static data.

`prices()`

Returns the order book prices. See structure *Prices* for more details.

`status()`

Returns the instrument trading status, market trading status and instrument integrity.

See structure *Status* for more details.

`imbalance()`

Returns the order book imbalance.

See structure *Imbalance* for more details.

`internalStatistics()`

Returns some internal counters. See structure *InternalStatistics* for more details.

`isSubscribed()`

Returns a boolean to indicate whether the view is subscribed (true) or only pre-subscribed (false).

The method `Session::trySubscribe()` has to be called to try to subscribe to pre-subscribed instruments.

3.2.6 Market

Description An object of class Market type provides the last live information updates on a market.

A Market object is an updatable object of kind UPDATABLE_MARKET (see enumeration *UpdatableKind* for more details).

The Market object is automatically created when a view is associated to a reader.

A Market object also provides a set of methods to read the dynamic data in shared memory. These methods provide the dynamic data that have been changed since the last time the client application checked the market trading status on the reader associated with the market.

The market implementation class is allocated on the heap when constructed and freed when the last pointer to it is destroyed.

The Market object can be allocated either on the stack or on the heap.

The copy constructor is enabled.

Copying a Market object only implies a copy of the pointer to the market implementation class.

Class UB::Market

Constructor `Market()`

Exceptions N/A

Operators N/A

Methods `status()`

See structure *MarketStatus* for more details.

`updateInfo()`

Returns the last updates on the market. Notifications are cleared when the user reads the change list.

See structure *UpdateInfo* for more details.

`marketIdentifier()`

Returns the market identifier as configured in the UB server configuration file.

3.3 Usage

UB automatically processes the data coming from the market feeds. It builds order books for instruments for which there is at least one subscription initiated. The use of the API does not affect the UB internal operation.

3.3.1 Instrument Pre-Subscription and Subscription

When an Instrument object is created, the API tries to find it in the instruments already referenced by UB (instruments for which referential data have been received from the exchange) to subscribe to it.

The user has the ability of anticipating an instrument creation by forcing the subscription to that instrument: a forced subscription is a pre-subscription.

The pre-subscription and subscription mechanism works as follows:

1. If the instrument is referenced by UB, the subscription will be activated and the user will be able to create views with that instrument.
2. If the instrument is not referenced by UB yet and the instrument creation is not forced (force parameter set to false), the instrument creation will throw an exception.
3. If the instrument is not referenced by UB yet and the instrument creation is forced (force parameter set to true), the user will pre-subscribe to the instrument. All views created with a pre-subscribed instrument will also be considered as pre-subscribed. As new instruments may be added intra-day, the user is responsible for changing the instruments and views pre-subscriptions into subscriptions by regularly calling `Session::trySubscribe()`.

3.3.2 View Changes Notifications

The API gives the user the ability to access the selected order books. It is possible to iteratively monitor the View objects by using the access methods such as `View::level()`, `View::status()`, `View::trades()` … The output of these methods is a current snapshot.

Order book changes are flagged using the change flags provided by UB. Changes in the View object are monitored using the change flags on a per View object basis. UB notifies the user when the change flags are first set. The user retrieves the notifications using the `Reader::updatedElements()` method. This method returns a pointer to a View object (a View object is an Updatable object of kind UPDATABLE_VIEW) which has been changed.

Then the user can obtain the change flags by calling the `View::updateInfo()` method. This method returns the change flags and then clears them. No further notifications occur until the change flags have been cleared using this method.

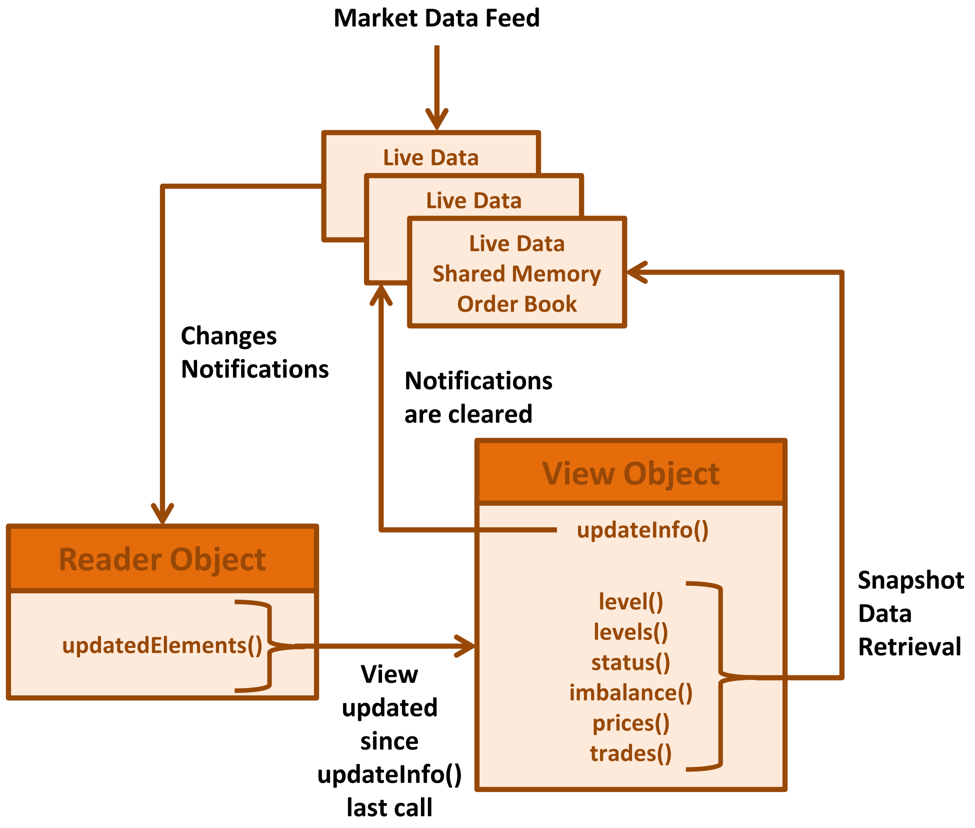
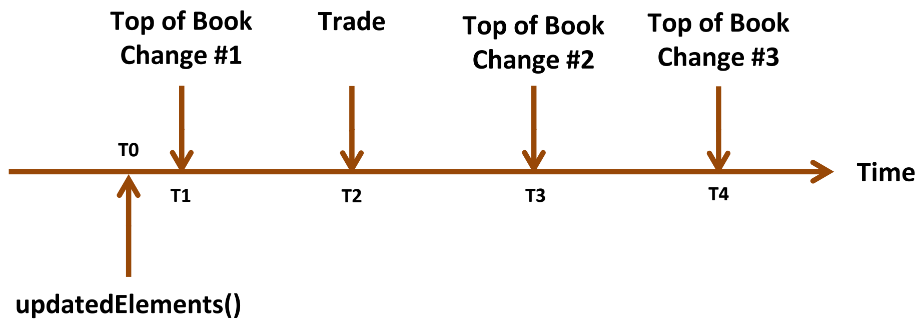


Figure 3 – View changes notifications

The processing of market data is asynchronous to the access methods. The access methods provide a snapshot of the current order book. As a result, the user will always see the same order of changes as received from the market feed. There is however no guarantee that the user will see all the changes; for example it can happen that the top of the book has changed twice, but the user only retrieves the top of book resulting from the two changes.

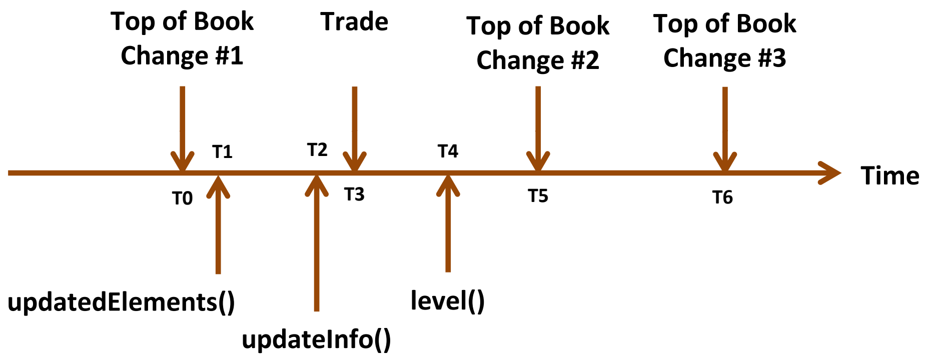
The API usage is illustrated by the following use cases:

1. First use case



At T0 the user calls `Reader::updatedElements()`. No pointer to the View object is returned assuming that the first change occurs at T1 only.

2. Second use case



a) At T0 the top of the book is changed. This change is assumed to be the first change. The top of the book change flag is set.

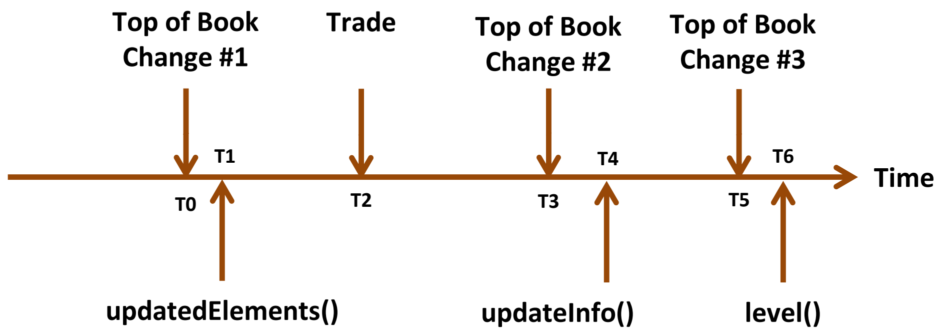
b) At T1 the user calls `Reader::updatedElements()`: the method returns a pointer to the View object.

c) At T2 the user calls `View::updateInfo()`: the method returns that the only change that occurred is a top of the book change. Then this change flag is automatically reset.

d) At T3 a trade is received. The trade change flag is set. The View object is therefore considered as updated again.

e) At T4 the user retrieves the current top of the book by using `View::level()`.

3. Third use case



a) At T0 the top of the book is changed. This change is assumed to be the first change. The top of the book change flag is set.

b) At T1 the user calls `Reader::updatedElements()`: the method returns a pointer to the View object.

c) At T2 a trade is received. The trade change flag is set.

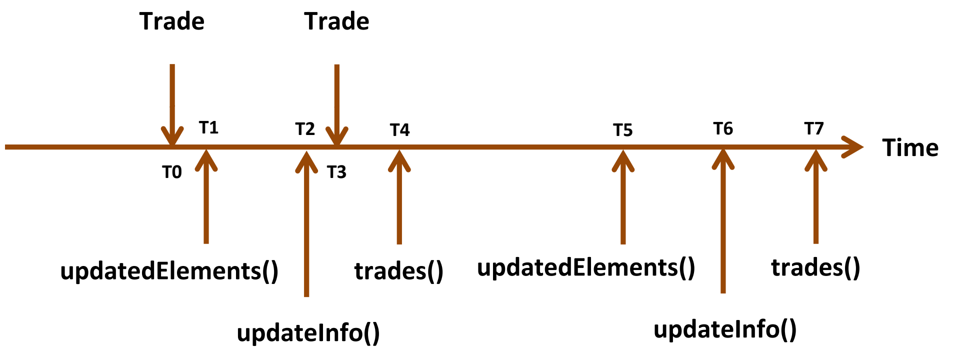
d) At T3 the top of the book is changed again. The top of the book change flag remains set.

e) At T4 the user calls `View::updateInfo()`: the method returns that the changes are of type top of the book change and trades. Then these two change flags are automatically reset.

f) At T5 the top of the book is changed again. The top of the book change flag is set. The View object is therefore considered as updated again.

g) At T6 the user retrieves the current top of the book by using `View::level()`: the retrieved top of the book takes into account the top of the book change that happened at T5.

4. Fourth use case



a) At T0 a trade is received. This change is assumed to be the first change. The trade change flag is set.

b) At T1 the user calls `Reader::updatedElements()`: the method returns a pointer to the View object.

f) At T2 the user calls `View::updateInfo()`: the method returns that the change is of type trade. Then this change flag is automatically reset.

c) At T3 a trade is received. The trade change flag is set.

d) At T4 the user calls `View::trades()`: the method returns two trades.

e) At T5 the user calls `Reader::updatedElements()`: the method returns a pointer to the View object.

f) At T6 the user calls `View::updateInfo()`: the method returns that the change is of type trade. Then this change flag is automatically reset.

g) At T7 the user calls `View::trades()`: the method returns no trade as the received trades have already been read.

3.3.3 Market Changes Notifications

The API gives the user the ability to monitor the Market objects by using the access method `Market::status()`. The output of these methods is a current snapshot.

Market changes are flagged using the change flags provided by UB. Changes in the Market object are monitored using the change flags on a per Market object basis. UB notifies the user when the change flags are first set. The user retrieves the notifications using the `Reader::updatedElements()` method. This method returns a pointer to a Market object (a Market object is an Updatable object of kind UPDATABLE_MARKET) which has been changed.

Then the user can obtain the change flags by calling the `Market::updateInfo()` method. This method returns the change flags and then clears them. No further notifications occur until the change flags have been cleared using this method.

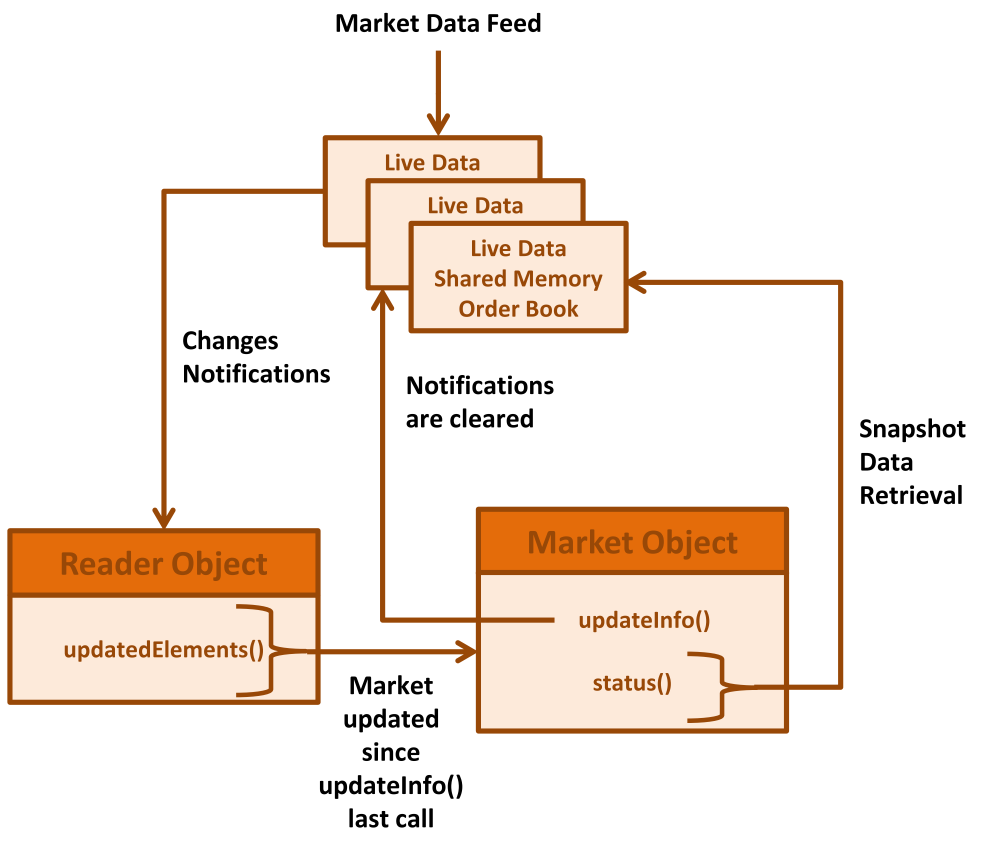


Figure 4 – Market changes notifications

The access method provides a snapshot of the current market trading status. As a result, the user will always see the same order of changes as received from the market feed. There is however no guarantee that the user will see all the changes; for example it can happen that the market trading status has changed twice, but the user only retrieves the last market trading status.

3.4 Structures

See the header files for more details.

3.4.1 InstrumentType

Description Provides the instrument type.

Structure InstrumentType

The structure description is as follows:

Field	Description
cfiCode	Instrument ISO 10962 CFI code
exchangeInstrumentType	Raw instrument type as provided by the exchange

3.4.2 Level

Description Provides the quantity and price on both bid and offer level sides.

Structure Level

The structure description is as follows:

Field	Description
askPrice	Price on the offer side
askQuantity	Quantity on the offer side
bidPrice	Price on the bid side
bidQuantity	Quantity on the bid side

3.4.3 Trade

Description Provides the trade data.

Structure Trade

The structure description is as follows:

Field	Description
type	Type of trade. See enumeration *TradeType* for the possible values.
quantity	Optional trade quantity
price	Optional trade price
exchangeFlags	Trade flags as provided by the exchange if any
exchangeTradeId	Trade identifier as provided by the exchange
exchangeTimestamp	Optional trade timestamp as provided by the exchange
exchangeSequenceNumber	Optional trade sequence number as provided by the exchange

3.4.4 TradeList

Description Provides a list of trades

Structure TradeList

The structure description is as follows:

Field	Description

Field	Description
totalTradedVolume	Cumulative total traded volume computed by UB
totalTradedCapital	Cumulative total traded capital computed by UB
trades	See structure *Trade* for more details
exchangeTotalTradedVolume	Exchange total traded volume as provided by the exchange

3.4.5 Prices

Description Provides the order book prices.

Structure Prices

The structure description is as follows:

Field	Description
openPrice	Optional open price
highPrice	Optional high price
lowPrice	Optional low price
closePrice	Optional closing price
settlementPrice	Optional settlement price
exchangeSettlementDate	Optional settlement date as provided by the exchange

3.4.6 Imbalance

Description Provides the order book imbalances.

Structure Imbalance

The structure description is as follows:

Field	Description
referencePrice	Optional reference price
pairedQuantity	Optional paired quantity at current reference price
askImbalance	Optional not paired offer quantity at current reference price
bidImbalance	Optional not paired bid quantity at current reference price

3.4.7 Status

Description Provides the instrument trading status and the market trading status.

Structure Status

The structure description is as follows:

Field	Description
marketStatus	Market status. See enumeration *TradingStatus* for the possible values.
exchangeMarketStatus	Raw exchange market status as provided by the exchange
marketIntegrity	Market integrity. See enumeration *Integrity* for the possible values.
securityStatus	Instrument trading status. See enumeration *TradingStatus* for the possible values.
exchangeSecurityStatus	Raw exchange instrument status as provided by the exchange
bookIntegrity	Instrument integrity.

Field	Description
	See enumeration *Integrity* for the possible values.
recoveryInProgress	Bit flag to indicate that UB is trying to update the order book whose integrity status is INTEGRITY_BEST_EFFORT

3.4.8 MarketStatus

Description Provides the market trading status.

Structure MarketStatus

The structure description is as follows:

Field	Description
marketStatus	Market status. See enumeration *TradingStatus* for the possible values.
exchangeMarketStatus	Raw exchange market status as provided by the exchange
marketIntegrity	Market integrity. See enumeration *Integrity* for the possible values.

3.4.9 UpdateInfo

Description Provides the event notifications.

Structure UpdateInfo

The structure description is as follows:

Field	Description
timestamp	Timestamp of the last event which updated the current view
exchangeTimestamp	Exchange packet timestamp of the last event which updated the current view
exchangeSequenceNumber	Exchange sequence number of the last message which updated the current view
flags	Bit flags to indicate the kind of information that has been updated since the last time the view has been read. Flags are reset each time the user reads. See enumeration *UpdateFlags* for the possible values.

3.4.10 InternalStatistics

Description Provides counters used to ensure that the data returned to the user have not been modified while being read in shared memory.

Structure InternalStatistics

The structure description is as follows:

Field	Description
readCount	Number of times the levels have been read by the user. This counter is incremented each time methods level() or levels() are called.
readFailedCount	Number of times UB failed to read the shared memory to return consistent data to the user when methods level() or levels() are called: data may indeed be modified while being read before being returned to the user. This counter is incremented each time that UB has to read again the shared memory to get consistent data to be returned to the user. The counter is not incremented if the first read achieves to get consistent data.

3.4.11 Instrument

Description Provides the raw exchange specific instrument data.

Structure Instrument

The structure description is as follows:

Field	Description
instrumentType	Protocol used to receive the instrument data. See enumeration *InstrumentType* for the possible values.
InstrumentData	Exchange specific structure which provides the raw instrument data as provided by the exchange. Refer to the exchange specifications for more details.

3.5 Enumerations

Some descriptions may be market dependent, refer to section *Market Specifics* for more details.

3.5.1 TradingStatus

Description Instrument and market status list.

Enumeration TradingStatus

Values	Description
TRADING_STATUS_OPEN	Open trading status. Orders and / or quotes submission is allowed with matching (regular trading) within market hours.
TRADING_STATUS_EXTENDED	Extended hours trading status. Orders and / or quotes submission is allowed with matching outside the standard market hours.
TRADING_STATUS_QUOTATION	Quotation-only trading status. Orders and / or quotes submission is allowed with no matching.
TRADING_STATUS_HALTED	Suspended or halted trading status. Orders and / or quotes submission may be allowed or not with no matching.
TRADING_STATUS_CLOSED	Closed trading status. Orders and / or quotes submission is not allowed.
TRADING_STATUS_INITIAL	This status is the initial instrument and / or market trading status. It is also the status assigned to an instrument and / or market when a trading status received from the exchange cannot be easily mapped against the supported trading statuses.

The following statuses can be combined with the previous ones:

Values	Description
TRADING_STATUS_SHORT_SELL_RESTRICTION	Short sell restriction trading status.
TRADING_STATUS_GUESSED_FLAG	Flag indicating the status is guessed: the mapping of the trading status provided by the exchange is not straightforward.

3.5.2 TradeType

Description Trade type list.

Enumeration TradeType

Values	Description
TRADE_TYPE_NEW	New trade
TRADE_TYPE_CANCEL	Canceled trade
TRADE_TYPE_UPDATE	Updated trade
TRADE_TYPE_INVALID	Not a valid trade

3.5.3 Integrity

Description Book / market integrity list.

Enumeration Integrity

Values	Description
INTEGRITY_CLEAN	No error, no data loss
INTEGRITY_BEST_EFFORT	The order book information or the market status is probably not accurate because of gaps and data loss
INTEGRITY_STALLED	The order book information or the market status is probably old

3.5.4 UpdateFlags

Description Update flags list.

Enumeration UpdateFlags

Values	Description
UPDATE_FLAGS_TRADE	Trade structure has been updated
UPDATE_FLAGS_TRADING_STATUS	The instrument trading status or the market trading status has been updated
UPDATE_FLAGS_BOOK_PRICES	The instrument prices have been updated
UPDATE_FLAGS_BOOK_IMBALANCE	The instrument imbalance has been updated
UPDATE_FLAGS_INTEGRITY	The instrument integrity or the market integrity has been updated
UPDATE_FLAGS_LEVEL_0_NORMAL	The top of the book for the normal quantity has been updated
UPDATE_FLAGS_LEVEL_1_NORMAL	Level 1 for the normal quantity has been updated
UPDATE_FLAGS_LEVEL_2_NORMAL	Level 2 for the normal quantity has been updated
UPDATE_FLAGS_LEVEL_3_NORMAL	Level 3 for the normal quantity has been updated
UPDATE_FLAGS_LEVEL_4_NORMAL	Level 4 for the normal quantity has been updated
UPDATE_FLAGS_LEVEL_0_IMPLIED	The top of the book for the implied quantity has been updated
UPDATE_FLAGS_LEVEL_1_IMPLIED	Level 1 for the implied quantity has been

Values	Description
	updated
UPDATE_FLAGS_LEVEL_2_IMPLIED	Level 2 for the implied quantity has been updated
UPDATE_FLAGS_LEVEL_3_IMPLIED	Level 3 for the implied quantity has been updated
UPDATE_FLAGS_LEVEL_4_IMPLIED	Level 4 for the implied quantity has been updated
UPDATE_FLAGS_LEVEL_0_CUSTOMER	The top of the book for the customer quantity has been updated
UPDATE_FLAGS_LEVEL_1_CUSTOMER	Level 1 for the customer quantity has been updated
UPDATE_FLAGS_LEVEL_2_CUSTOMER	Level 2 for the customer quantity has been updated
UPDATE_FLAGS_LEVEL_3_CUSTOMER	Level 3 for the customer quantity has been updated
UPDATE_FLAGS_LEVEL_4_CUSTOMER	Level 4 for the customer quantity has been updated
UPDATE_FLAGS_LEVEL_0_ALL_OR_NONE	The top of the book for the all or none quantity has been updated
UPDATE_FLAGS_LEVEL_1_ALL_OR_NONE	Level 1 for the all or none quantity has been updated
UPDATE_FLAGS_LEVEL_2_ALL_OR_NONE	Level 2 for the all or none quantity has been updated
UPDATE_FLAGS_LEVEL_3_ALL_OR_NONE	Level 3 for the all or none quantity has been updated
UPDATE_FLAGS_LEVEL_4_ALL_OR_NONE	Level 4 for the all or none quantity has been updated

3.5.5 QuantityType

Description Quantity type list.

Enumeration QuantityType

Values	Description
QUANTITY_TYPE_NORMAL	Total quantity as provided by the exchange
QUANTITY_TYPE_IMPLIED	Implied quantity as provided by the exchange
QUANTITY_TYPE_CUSTOMER	Customer quantity as provided by the exchange
QUANTITY_TYPE_ALL_OR_NONE	All or none quantity as provided by the exchange

3.5.6 InstrumentType

Description Protocol list.

Enumeration InstrumentType

Values	Description
INSTRUMENT_TYPE_UNKNOWN	Unknown protocol
INSTRUMENT_TYPE_CSM	Instrument received via the CSM protocol
INSTRUMENT_TYPE_ARCA_OPTIONS	Instrument received via the ARCAOPTIONS protocol

3.5.7 UpdatableKind

Description Updated object type list.

Enumeration UpdatableKind

Values	Description
UPDATABLE_UNKNOWN	Unknown object
UPDATABLE_MARKET	The updated object is a Market object
UPDATABLE_VIEW	The updated object is a view objet

4. Technical Architecture

This section should be read in conjunction with the Celoxica Manager User Guide.

4.1 UB Daemon Monitoring

UB is monitored by the Master Daemon: this daemon starts and stops processes, acts as a watchdog and sends alerts in case of issues.

UB uses the Logger Daemon: this daemon reads the log messages written to shared memory by UB, formats them and writes the formatted log messages to files.

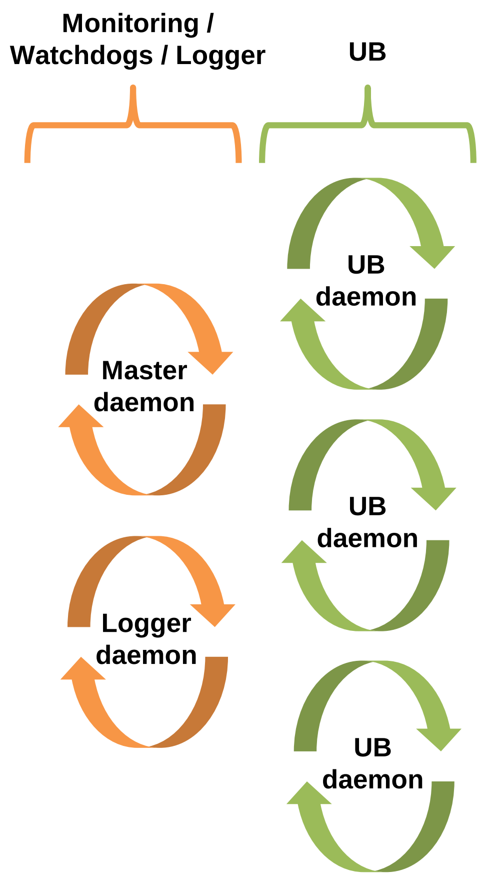


Figure 5 – UB daemon

4.2 Shared Memory Allocation

The shared memory is represented as a file in the /dev/shm filesystem. The size of this filesystem naturally determines the maximum size of the shared memory segment(s) that can be allocated for UB.

The shared memory must be allocated before starting the UB daemon using the "ultrabook-manage-memory" tool, otherwise errors will be raised and logged.

It is recommended that the "ultrabook-manage-memory" tool be monitored by the Master Daemon; see the Master Daemon configuration parameter config.init-appli.

As the "ultrabook-manage-memory" tool uses the Logger Daemon, the logs written by the "ultrabook-manage-memory" tool are accessible using the viewlog command.

The "ultrabook-manage-memory" tool has the following options:

Option	Description
-n or --name <server-name>	Allocates shared memory to server named <server-name>
-m or --market <market-count>	Limits the number of markets to <market-count> If not specified, 8 markets will be created by default.
-s or --subscription <subscription-count>	Limits the number of instrument subscriptions to <subscription-count>
-G <gigabytes>	Limits the allocated memory to <gigabytes> GiB
-M <megabytes>	Limits the allocated memory to <megabytes> MiB
--dry-run	Does not allocate or create anything, but only prints information about what would be done
--remove	Removes the existing shared memory for the given server name
--print	Prints the existing shared memory details for the given server name
-h or --help	Prints the help

The maximum number of readers is automatically set to 32.

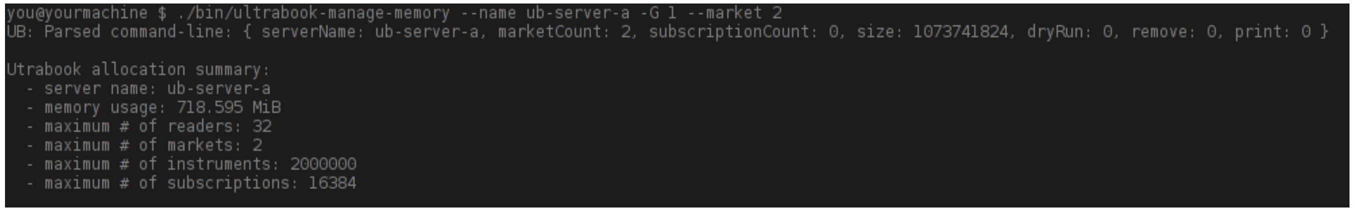
UB supports a maximum of 1,000,000 instruments per market.

The maximum number of instrument subscriptions is computed by the “ultrabook-manage-memory” tool as follows:

1. If the memory amount is provided using –M or –G options and the number of subscriptions is not provided using –s option, the maximum number of subscriptions will be rounded up down to the highest power of two which fits in memory.

   For example:

```
The user requires 1 GiB shared memory for 2 markets.
```



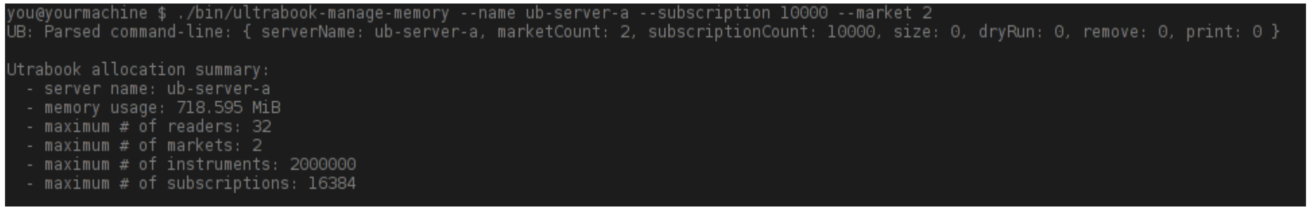
```
you@yourmachine $ ./bin/ultrabook-manage-memory --name ub-server-a -G 1 --market 2
UB: Parsed command-line: { serverName: ub-server-a, marketCount: 2, subscriptionCount: 0, size: 1073741824, dryRun: 0, remove: 0, print: 0 }

Utrabook allocation summary:
  - server name: ub-server-a
  - memory usage: 718.595 MiB
  - maximum # of readers: 32
  - maximum # of markets: 2
  - maximum # of instruments: 2000000
  - maximum # of subscriptions: 16384
```

```
In this example, the computed maximum number of subscriptions is 16384.
```

2. If the number of subscriptions is provided using –s option and the memory amount is not provided using either –M or –G options, the maximum number of subscriptions will be rounded up to the lowest power of two greater than the specified maximum and the allocated shared memory will be deduced.

   For example:

```
The user requires 10000 subscriptions for 2 markets.
```



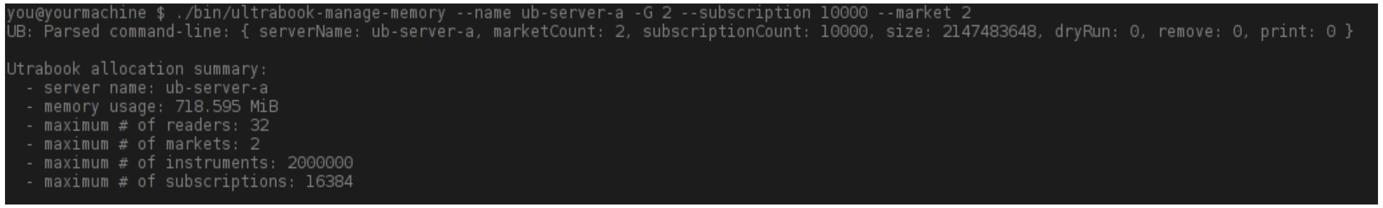
```
you@yourmachine $ ./bin/ultrabook-manage-memory --name ub-server-a --subscription 10000 --market 2
UB: Parsed command-line: { serverName: ub-server-a, marketCount: 2, subscriptionCount: 10000, size: 0, dryRun: 0, remove: 0, print: 0 }

Utrabook allocation summary:
  - server name: ub-server-a
  - memory usage: 718.595 MiB
  - maximum # of readers: 32
  - maximum # of markets: 2
  - maximum # of instruments: 2000000
  - maximum # of subscriptions: 16384
```

```
In this example, the computed maximum number of subscriptions is 16384.
```

3. If both the memory amount and the number of subscriptions are provided, the maximum number of subscriptions will be rounded up to the lowest power of two greater than the specified maximum and the allocated shared memory will be deduced (the specified number of subscriptions prevails over the specified memory amount).

For example:

```
The user requires 2 GiB shared memory and 10000 subscriptions for 2 markets.
```



```
you@yourmachine $ ./bin/ultrabook-manage-memory --name ub-server-a -G 2 --subscription 10000 --market 2
UB: Parsed command-line: { serverName: ub-server-a, marketCount: 2, subscriptionCount: 10000, size: 2147483648, dryRun: 0, remove: 0, print: 0 }

Utrabook allocation summary:
  - server name: ub-server-a
  - memory usage: 718.595 MiB
  - maximum # of readers: 32
  - maximum # of markets: 2
  - maximum # of instruments: 2000000
  - maximum # of subscriptions: 16384
```

```
In this example, the computed maximum number of subscriptions is 16384.
```

A single reader can handle the maximum number of subscriptions as computed when allocating the shared memory with the insurance of no notification loss.

4.3 Shared Memory Usage

4.3.1 Memory Consumption

The shared memory is attached only once on the server side. On the client side, the shared memory is attached once for the session object but also each time for each reader object.

As a consequence, when using the “top” tool, the user will get the following memory information:

- The virtual memory (VIRT counter) comprises several times the allocated shared memory
- The actual memory consumption is obtained by summing the SHR counter (shared memory) and the RES counter (resident memory)

4.3.2 Memory Lock

For performance purpose, Celoxica strongly recommends granting rights to the binaries to lock the memory on both the server and the client sides.

Giving permissions to lock the memory can be achieved:

1. Using the “setcap” command when this utility is available

```
sudo setcap cap_ipc_lock=ep my_binary
```

   This does not work over the NFS (Network File System).

2. Or by modifying /etc/security/limits.conf to increase the memlock counter.

```
#<domain>   <type>      <item>      <value>
    *          -        memlock     unlimited
```

   The documentation is accessible though standard man pages: "man limits.conf".

   Once the limits are changed the user must login again.

5. Shipped Examples

UB is shipped with a “hello world” example showcasing most of UB’s functional aspects and a “reader” example intending as a coding base for a client trading application using UB.

The two examples are intended to be easy to understand and to illustrate the use of the UB user API; they are therefore written with simplicity rather than speed in mind, and contains detailed comments.

To compile the example:

1. cd ~
2. celoxica_copy_examples
3. cd ~/celoxica-examples/ultrabook/hello_world

   or cd ~/celoxica-examples/ultrabook/reader
4. ls –la

   At this stage it is possible to review the files of the example.
5. make

To run the server:

1. cd ~/celoxica-examples/ultrabook/hello_world

   or cd ~/celoxica-examples/ultrabook/reader
2. ub -f ub-hello.cfg

   It is necessary to configure the UB server correctly. By default the ub-hello.cfg is provided. It connects to the CBOE Streaming Market CSM test environment. Note that on this environment there are very few book updates.

   Note that ub is installed in /usr/bin
3. leave your server running (do not terminate it)

To run the example:

1. cd ~/celoxica-examples/ultrabook/hello_world

   or cd ~/celoxica-examples/ultrabook/reader
2. ./hello_world or ./reader

   Note that the examples do not need any parameters.
3. Observe the output
4. Terminate the example

If the example does not terminate properly, the client will therefore be hanging on the server side indefinitely: you will not be able to run the example with the same server again.

6. Market Specifics

This section describes market specifics for the supported protocols.

6.1 ARCAOptions

This section describes the configuration specific settings for the following markets:

Market	Protocol	Description
ARCA OX	ArcaBook for Options	NYSE ARCA Option
AMEX OX	ArcaBook for Options	NYSE AMEX Option

The protocol to be set in *config.markets.market* is **ARCAOptions**.

Note:

Complex options are not supported yet.

6.1.1 Source

Documentation	NYSE ArcaBook for options Client Specification
Version	3.15
Date	September 27, 2012

6.1.2 Symbology

Valid identifier schemes are as follows:

Values	Support	Examples
key	✓	key://421048290333
index	✓	index://001e03e4.arcaox
local	✓	local://aapl.arcaox
osi	✓	osi://aapl 120201P00012340.arcaox
fsi	✗	-

6.1.3 Reference Data

The raw exchange specific instrument data structures are constructed from the following exchange messages:

Exchange Message	Type
Series Index Mapping	‘m’
Underlying Index Mapping	‘n’

6.1.4 Instrument Trading Status

TRADING_STATUS_INITIAL is set as follows:

Values	Yes / No
TRADING_STATUS_OPEN	✓
TRADING_STATUS_EXTENDED	✗
TRADING_STATUS_QUOTATION	✗
TRADING_STATUS_HALTED	✗
TRADING_STATUS_CLOSED	✗
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗
TRADING_STATUS_GUESSED_FLAG	✓

The instrument trading status is initialized with TRADING_STATUS_INITIAL and is then updated as follows:

Values	Support	Comments
TRADING_STATUS_OPEN	✓	Generated from message System Event 'v' where Event Code equals to: • 'N' (open indication dark series) • 'O' (open indication series) • 'o' (open indication) • 'T' (unhalt dark series) • 'U' (unhalt series) • 'u' (unhalt underlying)
TRADING_STATUS_EXTENDED	✗	-
TRADING_STATUS_QUOTATION	✗	-
TRADING_STATUS_HALTED	✓	Generated from message System Event 'v' where Event Code equals to: • 'S' (halt series) • 's' (halt underlying)
TRADING_STATUS_CLOSED	✓	Generated from message System Event 'v' where Event Code equals to: • 'X' (close indication series) • 'x' (close indication underlying)
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗	-
TRADING_STATUS_GUESSED_FLAG	✓	-
TRADING_STATUS_INITIAL	✓	Generated from message System Event 'v' where Event Code equals to: • 'D' (clear underlying ask) • 'E' (clear underlying bid) • 'F' (clear underlying) • 'L' (light up a dark series) • 'A' Clear series ask • 'B' Clear series bid • 'C' Clear series • 'G' Clear subscription ask • 'H' Clear subscription bid • 'I' Clear subscription

6.1.5 Market Trading Status

TRADING_STATUS_INITIAL is set as follows:

Values	Yes / No
TRADING_STATUS_OPEN	✓

Values	Yes / No
TRADING_STATUS_EXTENDED	✗
TRADING_STATUS_QUOTATION	✗
TRADING_STATUS_HALTED	✗
TRADING_STATUS_CLOSED	✗
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗
TRADING_STATUS_GUESSED_FLAG	✓

The market trading status remains TRADING_STATUS_INITIAL as no trading status at the market level is published.

6.1.6 Instrument Integrity

The instrument integrity remains INTEGRITY_CLEAN.

6.1.7 Market Integrity

Values	Support	Comments
INTEGRITY_CLEAN	✓	Initial status
INTEGRITY_BEST_EFFORT	✓	This status applies as soon as there is sequence gap. No recovery process is undertaken (recovery is not implemented yet).
INTEGRITY_STALLED	✗	-

6.1.8 Trade Type

Values	Support	Comments
TRADE_TYPE_NEW	✓	Generated from message Last Sale ‘x’
TRADE_TYPE_CANCEL	✓	Generated from message Trade Bust or Correction ‘u’ where Even Code = ‘B’ (Trade bust)
TRADE_TYPE_UPDATE	✓	Generated from message Trade Bust or Correction ‘u’ where Even Code = ‘C’ (Trade correction)

6.1.9 Update Flags

Values	Support	Comments
UPDATE_FLAGS_TRADE	✓	-
UPDATE_FLAGS_TRADING_STATUS	✓	-
UPDATE_FLAGS_BOOK_PRICES	✗	-
UPDATE_FLAGS_BOOK_IMBALANCE	✓	Generated from message Auction Imbalance ‘i’. Only Imbalance.referencePrice and Imbalance.pairedQuantity are provided.
UPDATE_FLAGS_INTEGRITY	✓	-
UPDATE_FLAGS_LEVEL_0_NORMAL	✓	-
UPDATE_FLAGS_LEVEL_1_NORMAL	✓	-
UPDATE_FLAGS_LEVEL_2_NORMAL	✓	-
UPDATE_FLAGS_LEVEL_3_NORMAL	✓	-
UPDATE_FLAGS_LEVEL_4_NORMAL	✓	-
UPDATE_FLAGS_LEVEL_0_IMPLIED	✗	-

Values	Support	Comments
UPDATE_FLAGS_LEVEL_1_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_2_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_3_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_4_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_0_CUSTOMER	✓	-
UPDATE_FLAGS_LEVEL_1_CUSTOMER	✓	-
UPDATE_FLAGS_LEVEL_2_CUSTOMER	✓	-
UPDATE_FLAGS_LEVEL_3_CUSTOMER	✓	-
UPDATE_FLAGS_LEVEL_4_CUSTOMER	✓	-
UPDATE_FLAGS_LEVEL_0_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_1_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_2_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_3_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_4_ALL_OR_NONE	✗	-

6.1.10 Quantity Type

Values	Support	Comments
QUANTITY_TYPE_NORMAL	✓	Corresponds to the Total Contract / Shares AQ_QUANTITY from message Aggregate Quote ‘q’
QUANTITY_TYPE_IMPLIED	✗	-
QUANTITY_TYPE_CUSTOMER	✓	Corresponds to the Customer Volume AQ_CUSTOMER_VOLUME from message Aggregate Quote ‘q’
QUANTITY_TYPE_ALL_OR_NONE	✗	-

6.1.11 Raw Exchange Specific Data

The raw exchange specific data is populated from the following exchange messages:

Field	UB Structure	Correspondence
exchangeInstrumentType	InstrumentType	SecurityType AQ_SECURITY_TYPE from message Underlying Index Mapping ‘n’
exchangeTotalTradedVolume	TradeList	Not populated
exchangeSettlementDate	Prices	Not populated
exchangeMarketStatus	Status / MarketStatus	Not populated
exchangeSecurityStatus	Status	EventCode AQ_EVENT_CODE from message System Event ‘v’
exchangeFlags	Trade	SaleCondition from message Last Sale ‘x’
exchangeTradeId	Trade	TradeReferenceNumber from message Last Sale ‘x’

6.2 CSM

This section describes the configuration specific settings for the following markets:

Market	Protocol	Description
CBOE	CBOE Streaming Market	Chicago Board Options Exchange
C2	CBOE Streaming Market	CBOE C2
One Chicago	CBOE Streaming Market	CBOE One Chicago
CFE	CBOE Streaming Market	CBOE Futures Exchange
CBSX	CBOE Streaming Market	CBOE Stock Exchange

The protocol to be set in *config.markets.market* is **CSM**.

6.2.1 Source

Documentation	CBOE Streaming Market CSM
Version	1.3.2
Date	March 30, 2012

6.2.2 Symbology

Valid identifier schemes are as follows:

Values	Support	Examples
key	✓	key://421048290333
index	✓	index://001e03e4.cboe
local	✓	local://aapl.cboe
osi	✓	osi://aapl 120201P00012340.cboe
fsi	✓	fsi://VIX 120201.cfe

6.2.3 Reference Data

The raw exchange specific instrument data structures are constructed from the following exchange messages:

Exchange Message	Type
Security Definition	'd'

6.2.4 Instrument Trading Status

TRADING_STATUS_INITIAL is set as follows:

Values	Support
TRADING_STATUS_OPEN	✓
TRADING_STATUS_EXTENDED	✗
TRADING_STATUS_QUOTATION	✗
TRADING_STATUS_HALTED	✗
TRADING_STATUS_CLOSED	✗
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗
TRADING_STATUS_GUESSED_FLAG	✓

The instrument trading status is initialized with TRADING_STATUS_INITIAL and is then updated as follows:

Values	Support	Comments
TRADING_STATUS_OPEN	✓	Generated from messages Current

Values	Support	Comments
		Market Refresh ‘W’ and Current Market Update ‘X’ where Security Trading Status equals to: • 17 (market open) • 23 (fast market) • 25 (strategy market quotes non-firm)
TRADING_STATUS_EXTENDED	✗	-
TRADING_STATUS_QUOTATION	✓	Generated from messages Current Market Refresh ‘W’ and Current Market Update ‘X’ where Security Trading Status equals to: • 21 (pre-open) • 22 (market in opening rotation) • 24 (strategy market in opening rotation)
TRADING_STATUS_HALTED	✓	Generated from messages Current Market Refresh ‘W’ and Current Market Update ‘X’ where Security Trading Status equals to: • 2 (market halted) • 26 (market suspended)
TRADING_STATUS_CLOSED	✓	Generated from messages Current Market Refresh ‘W’ and Current Market Update ‘X’ where Security Trading Status equals to: • 18 (market closed)
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗	-
TRADING_STATUS_GUESSED_FLAG	✓	-
TRADING_STATUS_INITIAL	✓	-

6.2.5 Market Trading Status

TRADING_STATUS_INITIAL is set as follows:

Values	Yes / No
TRADING_STATUS_OPEN	✓
TRADING_STATUS_EXTENDED	✗
TRADING_STATUS_QUOTATION	✗
TRADING_STATUS_HALTED	✗
TRADING_STATUS_CLOSED	✗
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗
TRADING_STATUS_GUESSED_FLAG	✓

The market trading status remains TRADING_STATUS_INITIAL as no trading status at the market level is published.

6.2.6 Instrument Integrity

The instrument integrity remains INTEGRITY_CLEAN.

6.2.7 Market Integrity

Values	Support	Comments
INTEGRITY_CLEAN	✓	Initial status
INTEGRITY_BEST_EFFORT	✓	This status applies as soon as there is sequence gap. No recovery process is undertaken (recovery is not implemented yet).
INTEGRITY_STALLED	✗	-

6.2.8 Trade Type

Values	Support	Comments
TRADE_TYPE_NEW	✓	Generated from message Ticker where Trade Condition <> ‘CANC’ or ‘CNCO’
TRADE_TYPE_CANCEL	✓	Generated from message Ticker where Trade Condition = ‘CANC’ or ‘CNCO’
TRADE_TYPE_UPDATE	✗	-

6.2.9 Update Flags

Values	Support	Comments
UPDATE_FLAGS_TRADE	✓	-
UPDATE_FLAGS_TRADING_STATUS	✓	-
UPDATE_FLAGS_BOOK_PRICES	✗	-
UPDATE_FLAGS_BOOK_IMBALANCE	✓	Generated from message EOP. Only Imbalance.referencePrice and Imbalance.pairedQuantity are provided.
UPDATE_FLAGS_INTEGRITY	✓	-
UPDATE_FLAGS_LEVEL_0_NORMAL	✓	CSM publishes only top of the book
UPDATE_FLAGS_LEVEL_1_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_2_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_3_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_4_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_0_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_1_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_2_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_3_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_4_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_0_CUSTOMER	✓	CSM publishes only top of the book
UPDATE_FLAGS_LEVEL_1_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_2_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_3_CUSTOMER	✗	-

Values	Support	Comments
UPDATE_FLAGS_LEVEL_4_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_0_ALL_OR_NONE	✓	CSM publishes only top of the book
UPDATE_FLAGS_LEVEL_1_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_2_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_3_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_4_ALL_OR_NONE	✗	-

6.2.10 Quantity Type

Values	Support	Comments
QUANTITY_TYPE_NORMAL	✓	Corresponds to the Total Limit Volume from messages Current Market Refresh ‘W’ and Current Market Update ‘X’
QUANTITY_TYPE_IMPLIED	✗	-
QUANTITY_TYPE_CUSTOMER	✓	Corresponds to the Customer Limit Volume from messages Current Market Refresh ‘W’ and Current Market Update ‘X’
QUANTITY_TYPE_ALL_OR_NONE	✓	Corresponds to the Total Contingency Volume from messages Current Market Refresh ‘W’ and Current Market Update ‘X’

6.2.11 Raw Exchange Specific Data

The raw exchange specific data is populated from the following exchange messages:

Field	UB Structure	Correspondence
exchangeInstrumentType	InstrumentType	Security Type from message Security Definition ‘d’
exchangeTotalTradedVolume	TradeList	Not populated
exchangeSettlementDate	Prices	Not populated
exchangeMarketStatus	Status / MarketStatus	Not populated
exchangeSecurityStatus	Status	Security Trading Status from messages Current Market Refresh ‘W’ and Current Market Update ‘X’
exchangeFlags	Trade	Trade Condition from message Ticker ‘X’ (exchangeFlags lower bits are populated)
exchangeTradeId	Trade	Not populated

6.3 OPRA_BINARY

This section describes the configuration specific settings for the following markets:

Market	Protocol	Description
OPRA	Binary	OPRA Binary

The protocol to be set in *config.markets.market* is **OPRA_BINARY**.

This protocol is not supported yet.

6.3.1 Source

Documentation	OPRA Binary Data Recipient Interface Specification
Version	1.0
Date	April 17, 2012

6.3.2 Symbology

Valid identifier schemes are as follows:

Values	Support	Examples
key	✓	key://421048290333
index	✗	-
local	✓	local://aapl 120201P00012340.opra
osi	✓	osi://aapl 120201P00012340.opra
fsi	✗	-

6.3.3 Instrument Trading Status

Instrument trading status is not available.

6.3.4 Market Trading Status

Market trading status is mapped as follows:

Values	Support	Comments
TRADING_STATUS_OPEN	✓	Generated from message of category 'H' where Type equals to: • 'D' (good morning)
TRADING_STATUS_EXTENDED	✗	-
TRADING_STATUS_QUOTATION	✗	-
TRADING_STATUS_HALTED	✗	-
TRADING_STATUS_CLOSED	✓	Generated from message of category 'H' where Type equals to: • 'G' (early market close) • 'H' (end of transaction reporting)
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗	-
TRADING_STATUS_GUESSED_FLAG	✗	-
TRADING_STATUS_INITIAL	✓	-

6.3.5 Trade Type

Values	Support	Comments
TRADE_TYPE_NEW	✓	Generated from message of category 'a' where Type equals to: • ' ' (REGULAR) • 'B' (OSEQ) • 'D' (LATE) • 'F' (OPEN) • 'H' (OPNL) • 'I' (AUTO) • 'J' (REOP) • 'K' (AJST) • 'L' (SPRD) • 'M' (STDL) • 'N' (STPD) • 'P' (BWRT) • 'Q' (CMBO) • 'R' (SPIM) • 'S' (ISOI) • 'T' (BNMT) • 'X' (XMPT)
TRADE_TYPE_CANCEL	✓	Generated from message of category 'a' where Type equals to: • 'A' (CANC) • 'C' (CNCL) • 'E' (CNCO) • 'G' (CNOL) • 'O' (CSTP)
TRADE_TYPE_UPDATE	✗	-

6.3.6 Update Flags

Values	Support	Comments
UPDATE_FLAGS_TRADE	✓	-
UPDATE_FLAGS_TRADING_STATUS	✓	-
UPDATE_FLAGS_BOOK_PRICES	✓	Only open_price, close_price, high_price and low_price are optionally provided. Generated from messages of category 'f'. Generated from messages of category 'a' where Type equals to: • 'H' (OPNL) • 'J' (REOP)
UPDATE_FLAGS_BOOK_IMBALANCE	✗	-
UPDATE_FLAGS_INTEGRITY	✓	-
UPDATE_FLAGS_LEVEL_0_NORMAL	✓	-
UPDATE_FLAGS_LEVEL_1_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_2_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_3_NORMAL	✗	-

Values	Support	Comments
UPDATE_FLAGS_LEVEL_4_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_0_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_1_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_2_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_3_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_4_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_0_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_1_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_2_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_3_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_4_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_0_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_1_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_2_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_3_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_4_ALL_OR_NONE	✗	-

6.3.7 Quantity Type

Values	Support	Comments
QUANTITY_TYPE_NORMAL	✓	-
QUANTITY_TYPE_IMPLIED	✗	-
QUANTITY_TYPE_CUSTOMER	✗	-
QUANTITY_TYPE_ALL_OR_NONE	✗	-

6.4 PHLXTopoV3

This section describes the specifics supported by the following markets:

Market	Protocol	Description
PHLX	TOPO	NASDAQ OMX PHLX Top Of PHLX Options
NOM	BONO	NASDAQ OMX NOM Best Of NASDAQ Options
BX Option	Top	NASDAQ OMX BX Options Top of Market

The protocol to be set in *config.markets.market* is **PHLXTopoV3**.

6.4.1 Source

Documentation	NASDAQ OMX PHLX Market Data Feed Top of PHLX Options
Version	3.00
Date	N/A

6.4.2 Symbology

Valid identifier schemes are as follows:

Values	Support	Examples
key	✓	key://421048290333
index	✓	index://001e03e4.phlx
local	✓	local://aapl.phlx
osi	✓	osi://aapl 120201P00012340.phlx
fsi	✗	-

6.4.3 Reference Data

The raw exchange specific instrument data structures are constructed from the following exchange messages:

Exchange Message	Type
Options Directory	'D'

6.4.4 Instrument Trading Status

TRADING_STATUS_INITIAL is set as follows:

Values	Support
TRADING_STATUS_OPEN	✓
TRADING_STATUS_EXTENDED	✗
TRADING_STATUS_QUOTATION	✗
TRADING_STATUS_HALTED	✗
TRADING_STATUS_CLOSED	✗
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗
TRADING_STATUS_GUESSED_FLAG	✓

The instrument trading status is initialized with TRADING_STATUS_INITIAL and is then updated as follows:

Values	Support	Comments
TRADING_STATUS_OPEN	✓	Generated from message Security Open / Closed 'O' where Open State equals to 'Y' (Open for auto execution) if Current Trading State from message Trading Action

Values	Support	Comments
		‘H’ equals to ‘T’ (Trading resumed)
TRADING_STATUS_EXTENDED	✗	-
TRADING_STATUS_QUOTATION	✗	-
TRADING_STATUS_HALTED	✓	a) Generated from message Trading Action ‘H’ where Current Trading State equals to ‘H’ (Halt in effect) b) Generated from message Security Open / Closed ‘O’ where Open State equals to ‘N’ (Closed for auto execution) if Current Trading State from message Trading Action ‘H’ equals to ‘T’ (Trading resumed)
TRADING_STATUS_CLOSED	✗	-
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗	-
TRADING_STATUS_GUESSED_FLAG	✓	-
TRADING_STATUS_INITIAL	✓	-

6.4.5 Market Trading Status

TRADING_STATUS_INITIAL is set as follows:

Values	Yes / No
TRADING_STATUS_OPEN	✓
TRADING_STATUS_EXTENDED	✗
TRADING_STATUS_QUOTATION	✗
TRADING_STATUS_HALTED	✗
TRADING_STATUS_CLOSED	✗
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗
TRADING_STATUS_GUESSED_FLAG	✓

The instrument trading status is initialized with TRADING_STATUS_INITIAL and is then updated as follows:

Values	Support	Comments
TRADING_STATUS_OPEN	✓	Generated from message System Event ‘S’ where Event Code equals to ‘Q’ (Start of opening process)
TRADING_STATUS_EXTENDED	✓	Generated from message System Event ‘S’ where Event Code equals to ’N’ (End of normal hours processing)
TRADING_STATUS_QUOTATION	✓	Generated from message System Event ‘S’ where Event Code equals to ‘S’ (Start of system hours)
TRADING_STATUS_HALTED	✗	-
TRADING_STATUS_CLOSED	✓	Generated from message System Event ‘S’ where

Values	Support	Comments
		Event Code equals to: • 'L' (End of late hours processing) • 'E' (End of system hours)
TRADING_STATUS_SHORT_SELL_RESTRICTION	✗	-
TRADING_STATUS_GUESSED_FLAG	✓	-
TRADING_STATUS_INITIAL	✓	-

6.4.6 Instrument Integrity

The instrument integrity remains INTEGRITY_CLEAN.

6.4.7 Market Integrity

Values	Support	Comments
INTEGRITY_CLEAN	✓	Initial status
INTEGRITY_BEST_EFFORT	✓	This status applies as soon as there is sequence gap. No recovery process is undertaken (recovery is not implemented yet).
INTEGRITY_STALLED	✗	-

6.4.8 Trade Type

Values	Support	Comments
TRADE_TYPE_NEW	✓	Generated from message Trade Report 'R'
TRADE_TYPE_CANCEL	✓	Generated from message Broken Trade Report 'X'
TRADE_TYPE_UPDATE	✗	-

6.4.9 Update Flags

Values	Support	Comments
UPDATE_FLAGS_TRADE	✓	-
UPDATE_FLAGS_TRADING_STATUS	✓	-
UPDATE_FLAGS_BOOK_PRICES	✗	-
UPDATE_FLAGS_BOOK_IMBALANCE	✗	-
UPDATE_FLAGS_INTEGRITY	✓	-
UPDATE_FLAGS_LEVEL_0_NORMAL	✓	PHLX publishes only top of the book
UPDATE_FLAGS_LEVEL_1_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_2_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_3_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_4_NORMAL	✗	-
UPDATE_FLAGS_LEVEL_0_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_1_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_2_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_3_IMPLIED	✗	-

Values	Support	Comments
UPDATE_FLAGS_LEVEL_4_IMPLIED	✗	-
UPDATE_FLAGS_LEVEL_0_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_1_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_2_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_3_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_4_CUSTOMER	✗	-
UPDATE_FLAGS_LEVEL_0_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_1_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_2_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_3_ALL_OR_NONE	✗	-
UPDATE_FLAGS_LEVEL_4_ALL_OR_NONE	✗	-

6.4.10 Quantity Type

Values	Support	Comments
QUANTITY_TYPE_NORMAL	✓	Corresponds to: • Bid Size or Ask Size from messages: ○ Best Bid AND Ask Update Short Form ‘q’ ○ Best Bid AND Ask Update Long Form ‘Q’ • Size from messages: ○ Best Bid OR Ask Update Short Form ‘a’ / ‘b’ ○ Best Bid AND Ask Update Long Form ‘A’ / ‘B’
QUANTITY_TYPE_IMPLIED	✗	-
QUANTITY_TYPE_CUSTOMER	✗	-
QUANTITY_TYPE_ALL_OR_NONE	✗	-

6.4.11 Raw Exchange Specific Data

The raw exchange specific data is populated from the following exchange messages:

Field	UB Structure	Correspondence
exchangeInstrumentType	InstrumentType	Not populated
exchangeTotalTradedVolume	TradeList	Not populated
exchangeSettlementDate	Prices	Not populated
exchangeMarketStatus	Status / MarketStatus	Event Code from message System Event ‘S’
exchangeSecurityStatus	Status	Not populated
exchangeFlags	Trade	Trade Condition from message Trade Report ‘R’
exchangeTradeId	Trade	Cross ID from message Trade Report ‘R’

7. FAQ

1. The client is hanging on the server side indefinitely

Please check the configuration node *config.control.client-timeout* setting.

Note that the "hello world" example does not use the heartbeats: the default UB server configuration file ub.cfg has the timeout monitoring deactivated (=0).

2. Shared memory allocation

Run the following command to allocate 1024 MB shared memory to the UB server named ub-server:

```
[you@yourmachine]$  /usr/bin/ultrabook-manage-memory  -n  ub-
server -M 2048
```

3. Shared memory removal

Either use the "ultrabook-manage-memory" tool and run the following command to remove the existing shared memory for the UB server named ub-server:

```
[you@yourmachine]$  /usr/bin/ultrabook-manage-memory  -n  ub-
server --remove
```

Or run the following command:

```
rm /dev/shm/ub-server
```