CME MDP 3.0 Handler Documentation

Introduction

This document provides guidance to C++ programmers on using the B2BITS CME MDP 3.0 Handler.
This document assumes the reader is familiar with MDP 3.0 documentation found on the CME web site.

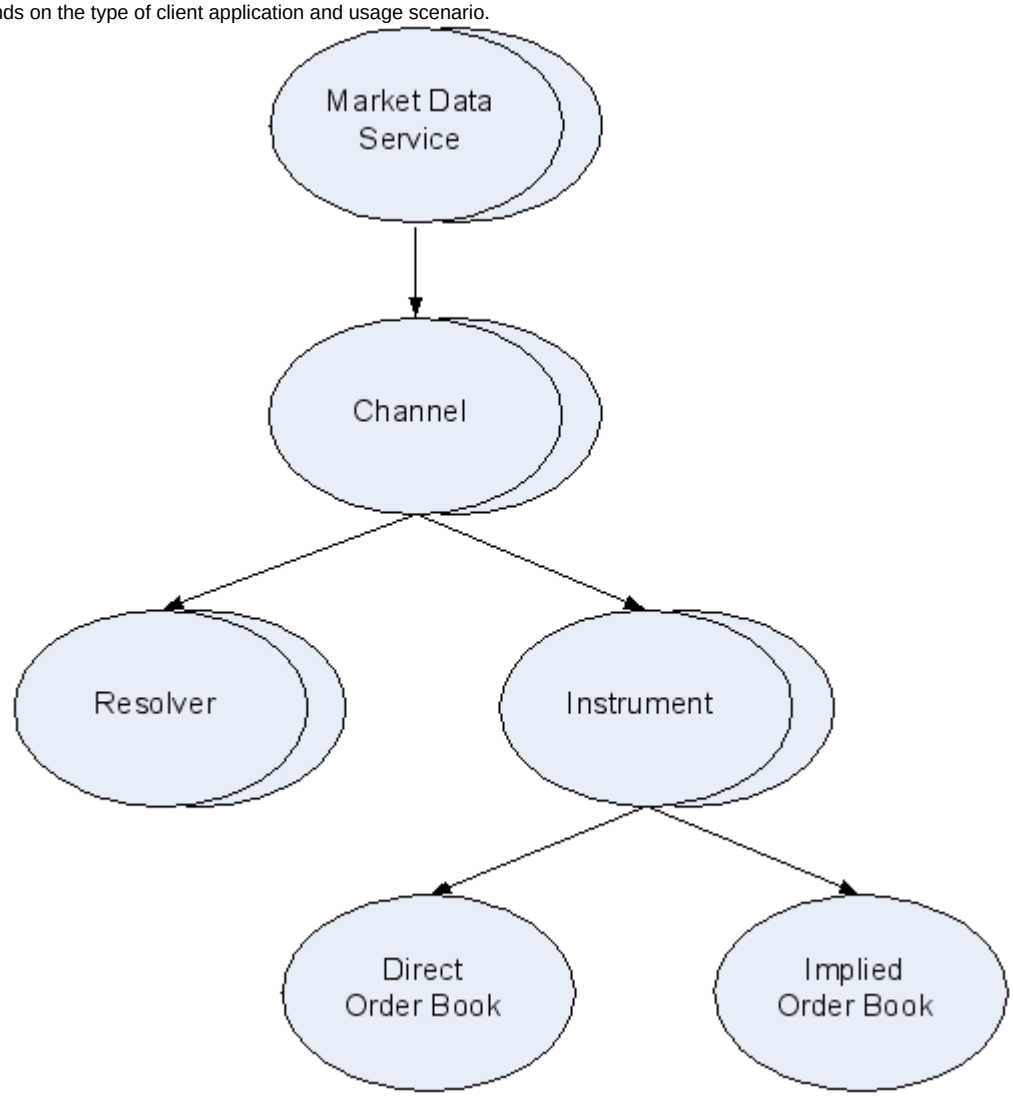
> ⚠ Notes and tips are displayed in yellow.

```
Source code fragments are displayed in code blocks.
```

Complete API reference is available here.

Architecture Overview

Architecture of the CME MDP 3.0 Handler reflects the architecture of the CME Market Data Platform 3. Major entities are Market Data Service, Channel, Resolver, and Instrument, having one-to-many relationships. Each entity is described in detail in subsequent sections. The number of instantiated objects of each entity depends on the type of client application and usage scenario.




```
using namespace Cme;
using namespace Cme::Mdp;
```

Market Data Service

Market Data Service object is a root object to access CME market data. The object maintains a collection of market data channels as well as resources shared across the channels.
Market Data Service object is instantiated by specifying service unique name and options:

```
MarketDataServiceOptions marketDataServiceOptions;
marketDataServiceOptions.channelConfigFile = "config.xml";
// ...
MarketDataService* marketDataService = fixEngine->createCmeMdpMarketDataService("MDS1", marketDataOptions);
```

Market Data Service object is initialized with the open method. Initialization involves loading SBE templates, channel configuration, and resources allocation:

```
marketDataServiceOptions->open();
```

Market Data service object can be finalized with the close method to free resources occupied:

```
marketDataServiceOptions->close();
```

Market Data Service object is destroyed with the destroy method:

```
marketDataServiceOptions->destroy();
```

The object reference is no longer valid.

Market Data Service Options

There is a number of options to control Market Data Service object behavior.
Options are specified separately for incremental and recovery (instrument definition, snapshot) feeds. Options are equally applied to all channels of the Market Data Service object.

> ⚠ Tip: Use different Market Data Service objects to apply different options to groups of channels.

Market Data Service object can be configured to use specific types of instrument recovery on initial join and subsequent recoveries on gaps and sequence reset:

```
marketDataServiceOptions.joinRecoveryType = rtSnapshotRecovery;
marketDataServiceOptions.gapRecoveryType = rtNaturalSnapshotRecovery;
```

Recovery via TCP Historical Replay feed is also available, it requires user name and password to be configured in marketDataServiceOptions:

```
marketDataServiceOptions.username_= "CME";
marketDataServiceOptions.password_= "CME";
```

When credentials are specified and Historical Replay Feed is present in 'config.xml' configuration file - it will be used to subsequent recoveries on small gaps.

Market by Order options

CME Market by Order (MBO) functionality is available in addition to existing Market By Price (MBP) functionality. When enabled - MBO message updates are delivered to OnMessage callbacks and MBO snapshot feeds are used for recovery.

MBO feature does not support per-instrument sequencing, so when a gap is detected in the channel - all subscribed instruments are considered to be out-of-sync and recovery process is started. It is recommended to use TCP Historical Replay feed when using MBO.

> ⚠ MBO support is disabled by default in the handler.

It is configured via *useMBOFeeds_* option:

```
MarketDataServiceOptions marketDataServiceOptions;
marketDataServiceOptions.useMBOFeeds_= true;
```

Market Data Service Sequence Options

Sequence options control sequence processing of the corresponding data feeds.
Market Data Service object can be configured to disable data feed B:

```
marketDataServiceOptions.incrementalSequeneOptions.dataFeedBEnabled = false;
```

Market Data Service Thread Options

Thread options control threads used to process packets of the corresponding data feeds.
Market Data Service object can be configured to create a dedicated thread for each feed or use a limited pool of shared threads across all the corresponding feeds:

```
marketDataServiceOptions.incrementalThreadOptions.threadPoolEnabled = true;
marketDataServiceOptions.incrementalThreadOptions.threadPoolSize = 4;
```

Market Data Service object can be configured to spin threads and avoid idle state to reduce latency at the cost of CPU load and scalability:

```
marketDataServiceOptions.incrementalThreadOptions.spinningEnabled = true;
```

Market Data Service object can be configured to bind threads to a set of CPU cores to reduce latency at the cost of scalability:

```
marketDataServiceOptions.incrementalThreadOptions.affinityMask = 0x3;
```

Market Data Service Socket Options

Socket options control the socket layer used to receive packets of the corresponding data feeds.
Market Data Service object can be configured to use a specific socket type:

```
marketDataServiceOptions.incrementalSocketOptions.socketType = stMyricomDBL;
```

Market Data Service object can be configured to use specific network interfaces for data feeds A and B on a multi-home host:

```
marketDataServiceOptions.incrementalSocketOptions.interfaceAddressA = "172.17.94.7";
marketDataServiceOptions.incrementalSocketOptions.interfaceAddressB = "172.17.94.8";
```

Market Data Service Logging Options

Logging options control logging behavior for testing and troubleshooting purpose.
Market Data Service options can be configured to log all incoming and/or outcoming messages and events:

```
marketDataServiceOptions.loggingOptions.logInMessages = true;
marketDataServiceOptions.loggingOptions.logOutMessages = true;
```

⚠ Note: Logging degrades performance and is not intended for normal operation.

Channels

Channel object is used to receive market data of a particular CME channel and maintains a collection of resolvers and a collection of instruments.
Channel objects for channels defined in the channel configuration file are instantiated automatically by the open method.
Alternatively, Channel object can be instantiated explicitly by specifying channel unique id and options:

```
ChannelOptions channelOptions;
channelOptions.snapshotAddressA = "224.0.31.22";
// ...
Channel* channel = marketDataService->addChannel(310, channelOptions);
```

Channel object can be retrieved by channel id:

```
Channel* channel = marketDataService->getChannel(310);
```

Channel object can be removed to free resources occupied:

```
marketDataService->removeChannel(310);
```

The object reference is no longer valid.

Resolvers

Resolver object is used to receive reference data (instrument definitions) of the channel. The client application may use one or more resolver objects to receive reference data.
Resolver object is instantiated by specifying resolver unique id:

```
Resolver* resolver = channel->addResolver("ES");
```

Resolver object can be retrieved by resolver id:

```
Resolver* resolver = channel->getResolver("ES");
```

Resolver object can be removed to free resources occupied:

```
channel->removeResolver("ES");
```

The object reference is no longer valid.

Instruments

Instrument object is used to receive market data of a particular instrument. The client application may process a single instrument, a subset of instruments or all instruments of the channel. Market data messages of uninterested instruments are filtered out by the handler.
Instrument object is instantiated by specifying instrument unique symbol, unique id and options:

```
InstrumentOptions instrumentOptions;
instrumentOptions.asset = "ES";
// ...
Instrument* instrument = channel->addInstrument("ESZ4", 28095, instrumentOptions, iboAll);
```

Instrument object can be retrieved by instrument symbol or instrument id:

```
Instrument* instrument = channel->getInstrument("ESZ4");
Instrument* instrument = channel->getInstrument(28095);
```

Instrument object can be removed to free resources occupied:

```
channel->removeInstrument("ESZ4");
channel->removeInstrument(28095);
```

The object reference is no longer valid.

Instrument Options

There are a number of options required by the handler to process instrument market data messages properly.
Instrument asset and group options found in the instrument definition are required to process status update messages:

```
instrumentOptions.asset = "ES";
instrumentOptions.group = "ES";
```

Instrument outright and implied market depth options found in the instrument definition are required to build and update instrument order books:

```
instrumentOptions.directMarketDepth = 10;
instrumentOptions.impliedMarketDepth = 2;
```

Instrument Build Options

Instrument build options define what components of the instrument state are maintained by the handler.

```
iboLastTrade | iboElectronicVolume | iboDirectOrderBook;
```


Tip: Specify build options of interest to reduce market data processing overhead.

Processing Reference Data

The client application implements the ResolverListener callback interface to process reference data events and messages:

```
class MyResolverListener : public ResolverListener
{
        public:
        virtual ResolverControlCode onEvent(Resolver* resolver, const ResolverEvent& event) override;

    virtual ResolverControlCode onMessage(Resolver* resolver, const InstrumentFutureMsg& message) override;
    virtual ResolverControlCode onMessage(Resolver* resolver, const InstrumentSpreadMsg& message) override;
    virtual ResolverControlCode onMessage(Resolver* resolver, const InstrumentOptionMsg& message) override;
 };
MyResolverListener myResolverListener;
```

Reference data processing is initiated by starting a Resolver object:

```
resolver->start(&myResolverListener, rsmAll);
```

Once resolver is started, a complete instrument definition cycle is delivered to the resolver listener followed by any intraday instrument definition updates.
Resolver listeners of different resolver objects receive reference data independently of each other.
Reference data processing can be stopped with the stop method:

```
resolver->stop();
```

Resolver Subscription Mask

Resolver subscription mask defines what events are delivered to the resolver listener object:

```
rsmAll & ~rsmMessages;
```


Tip: Specify events of interest to reduce reference data processing overhead.

Processing Reference Data Events

Reference data events are delivered to the client application via the onEvent callback function. It is expected the client application checks the type of a reference data event and processes it appropriately. Certain events have parameters associated with them.
In the case the client application processes both reference and live data it can instantiate instrument objects of interest in the onEvent callback function:

```
ResolverControlCode MyResolverListener::onEvent(Resolver* resolver, const ResolverEvent& event)
{
        if (event.type == reInstrumentAdded)
        {
                if (event.instrumentDefinition->asset == "ES")
                {
                        resolver->getChannel()->addInstrument(event.instrumentDefinition);
                }
        }
        // ...
}
```

The instrument symbol, id and options are deduced from the instrument definition object automatically.

Processing Reference Data Messages

Reference data messages (d) are delivered to the client application via the onMessage callback function. Reference data messages contain all the tags defined for the instrument. It is expected the client application checks the update action of the message and processes it appropriately.

Processing Live Data

The client application implements the InstrumentListener callback interface to process instrument live data events and messages:

```
class MyInstrumentListener : public InstrumentListener
{
        public:
                virtual InstrumentControlCode onEvent(Instrument* instrument, const InstrumentEvent& event,
bool NR) override;

                virtual InstrumentControlCode onMessage(Instrument* instrument, const MBPIncBookMsg& message,
bool NR) override;
        virtual InstrumentControlCode onMessage(Instrument* instrument, const IncDailyStatisticsMsg& message,
bool NR) override;
        virtual InstrumentControlCode onMessage(Instrument* instrument, const IncLimitsBandingMsg& message,
bool NR) override;
        virtual InstrumentControlCode onMessage(Instrument* instrument, const IncSessionStatisticsMsg& message,
bool NR) override;
        virtual InstrumentControlCode onMessage(Instrument* instrument, const IncTradeMsg& message, bool NR)
override;
        virtual InstrumentControlCode onMessage(Instrument* instrument, const IncVolumeMsg& message, bool NR)
override;
        virtual InstrumentControlCode onMessage(Instrument* instrument, const IncTradeSummaryMsg& message, bool
NR) override;

};
MyInstrumentListener myInstrumentListener;
```

Instrument live data processing is initiated by subscribing an Instrument object:

```
instrument->subscribe(&myInstrumentListener, ismAll);
```

Once the instrument is subscribed, the instrument data recovery is started followed by any incremental updates of the instrument live data.
Instrument live data processing can be stopped with the unsubscribe method:

```
instrument->unsubscribe();
```

Instrument Subscription Mask

Instrument subscription mask defines what events are delivered to the instrument listener object:

```
ismUpdateEndEvents | ismResetEvents | ismMessages;
```


Tip: Specify events of interest to reduce live data processing overhead.

Processing Live Data Events

Live data events are delivered to the client application via the onEvent callback function. It is expected the client application checks the type of a live data event and processes it appropriately. Some events have parameters associated with them while some events indicate an update of the corresponding component of the instrument state.

```
InstrumentControlCode MyInstrumentListener::onEvent(Instrument* instrument, const InstrumentEvent& event, bool
NR)
{
        if (event.type == ieUpdateEnd)
        {
                auto indicator = event.updateEndFlags_.meIndicator();

                if (indicator.test(MatchEventIndicator::LastQuoteMsg))
                {
                        // process book update
                         const OrderBook* orderBook = instrument->getDirectOrderBook();
                        // ...
                }
         }
}
```


Note: The client application may access the instrument state in the IntrumentListener callback interface functions only.

The client application should consider transaction (match event indicator) boundaries when processing live data updates. The ieUpdateEnd events indicate the end of updates of the certain type of data of a transaction, match event indicator is passed as a parameter to event handler.

Processing Live Data Messages

Live data messages (W, X, f, R) are delivered to the client application via the onMessage callback function. The client application may use live data messages to implement custom data processing.
Note: Each incremental update group entry is delivered as a separate message object of type X.

```
InstrumentControlCode MyInstrumentListener::onMessage(Instrument* instrument, const MBPIncBookMsg& message,
bool NR)
{
        switch (message.getMDEntryType())
        {
                case MDEntryTypeBook::Bid:
                {
                        // ...
                        break;
                }
                case MDEntryTypeBook::Offer:
                {
                        // ...
                        break;
                }
                case MDEntryTypeBook::ImpliedBid:
                {
                        // ...
                        break;
                }
                case MDEntryTypeBook::ImpliedOffer:
                {
                        // ...
                        break;
                }
        }
        // ...
}
```

Processing Market By Order (MBO) updates

MBO notifications are available as live data messages updates delivered to the OnMessage callback function. The client application may use live data messages to implement MBO data processing.

There are 3 callbacks that receive MBO messages:

```
// MBO snapshot message
virtual InstrumentControlCode onMessage(Instrument* instrument, const MBOSnapshotMsg& message);
// Separate MBO update
virtual InstrumentControlCode onMessage(Instrument* instrument, const MBOIncOrderBookMsg& message) ;
// Combined MBO+MBP update
virtual InstrumentControlCode onMessage(Instrument* instrument, const MBOIncBookMsg& message, const
MBPIncBookMsg& refMsg) ;
```

Low Latency Configuration

To configure the handler for low latency at the cost of other criteria use these settings:

```
marketDataServiceOptions.recoveryThreadOptions.threadPoolEnabled = true;
marketDataServiceOptions.recoveryThreadOptions.spinningEnabled = false;
marketDataServiceOptions.recoveryThreadOptions.affinityMask = 0x0F;
marketDataServiceOptions.recoverySocketOptions.socketType = stAsio;
marketDataServiceOptions.incrementalThreadOptions.threadPoolEnabled = false;
marketDataServiceOptions.incrementalThreadOptions.spinningEnabled = true;
marketDataServiceOptions.incrementalThreadOptions.affinityMask = 0xF0;
marketDataServiceOptions.incrementalSocketOptions.socketType = stSystem;
marketDataServiceOptions.loggingOptions.logInMessages = false;
marketDataServiceOptions.loggingOptions.logOutMessages = false;
```

Client Samples

Two simple interactive console applications found in the samples folder demonstrate how to instantiate the handler, process reference and live data events and messages.

help (h)	Print commands
open (o) channel_id	Open channel channel_id
subscribe (s) channel_id 'symbol'	Subscribe to instrument symbol of channel channel_id
subscribe (s) channel_id	Subscribe to all instruments of channel channel_id
unsubscribe (u) channel_id 'symbol'	Unsubscribe from instrument symbol of channel channel_id
unsubscribe (u) channel_id	Unsubscribe from all instruments of channel channel_id
close (c) channel_id	Close channel channel_id
close (c)	Close all channels
exit (e)	Exit application