B2BITS CME MDP Handler

Programmer's Guide

Version 1.0.6 (2015-06-09)

Table of Contents

1 Introduction

This document provides guidance to C++ programmers on using the B2BITS CME MDP Handler.

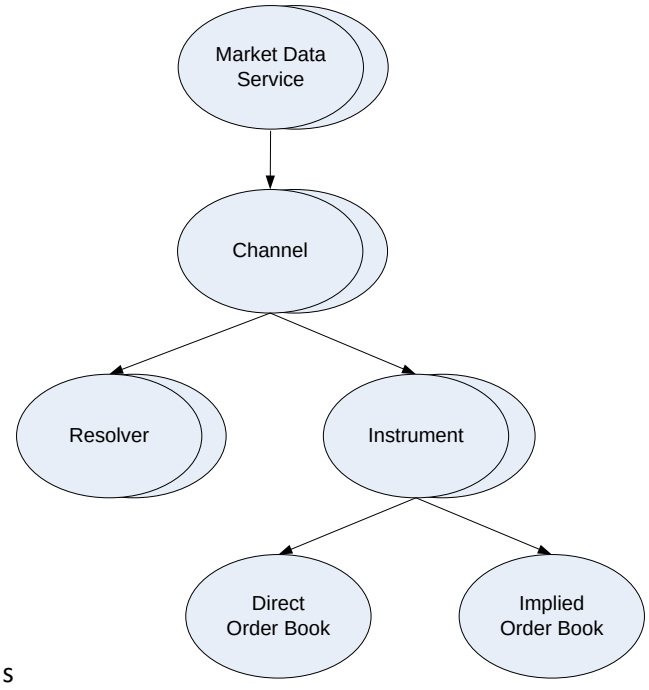
This document assumes the reader is familiar with MDP 3 pages found on the CME web site.

Notes and tips are displayed in yellow.

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

2 Architecture Overview

Architecture of the Market Data Handler reflects 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.



s

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

3 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 = createMarketDataService("MDS",
marketDataServiceOptions);
```

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

```
marketDataService->open();
```

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

```
marketDataService->close();
```

Market Data Service object is destroyed with the destroy method:

```
marketDataService->destroy();
```

The object reference is no longer valid.

3.1 Market Data Service Options

There are 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 resets:

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

3.1.1 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.incrementalSequenceOptions.dataFeedBEnabled = false;
```

Market Data Service object can be configured to ignore packet gaps:

```
marketDataServiceOptions.recoverySequenceOptions.ignorePacketGaps = true;
```

Note: Data integrity is not guaranteed if packet gaps are ignored.

3.1.2 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;
```

3.1.3 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";
```

3.1.4 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.

4 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.

5 Resolver

Resolver object is used to receive reference data (instrument definitions) of the channel.

Resolver object can be retrieved with the getResolver method:

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

6 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.

6.1 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;
```

6.2 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.

7 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);

    virtual ResolverControlCode onMessage(Resolver* resolver, const Message& message);
};

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.

Note: Due to possible loss of UDP packets the client application should be ready to receive several reCycleBegin events before receiving reCycleEnd event.

Reference data processing can be stopped with the stop method:

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

7.1 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.

7.2 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.

7.3 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.

8 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);

    virtual InstrumentControlCode onMessage(Instrument* instrument, const Message& message);

    virtual InstrumentControlCode onNaturalRecoveryEvent(Instrument* instrument, const
InstrumentEvent& event);

    virtual InstrumentControlCode onNaturalRecoveryMessage(Instrument* instrument, const
Message& message);

};

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();
```

8.1 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.

8.2 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)
{
    if (event.type == iePriceUpdateEnd)
    {
        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 (matching event) boundaries when processing live data updates. The ieXXXUpdateEnd events indicate the end of updates of the corresponding type of data of a transaction while the ieTransactionEnd event indicates the end of all updates of a transaction.

8.3 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. It is expected the application checks the type of a message and processes it accordingly.

Note: Each incremental update group entry is delivered as a separate message object of type X.

```
InstrumentControlCode MyInstrumentListener::onMessage(Instrument* instrument, const Message&
message)
{
    char type = message.getType();

    if (type == 'X')
    {
        char entryType = message.getAsChar(Fields::MDEntryType);

        switch (entryType)
        {
            case MDEntryType::Bid:
            {
                // ...

                break;
            }
            case MDEntryType::Offer:
            {
                // ...

                break;
            }
            case MDEntryType::ImpliedBid:
            {
                // ...

                break;
            }
            case MDEntryType::ImpliedOffer:
            {
                // ...

                break;
            }
            case MDEntryType::Trade:
            {
```

```
                // ...

                break;
            }
            case MDEntryType::ElectronicVolume:
            {
                // ...

                break;
            }
        }
    }

    // ...
}
```

Processing Raw Data

The client application implements ChannelDataFeedListener callback interface to process channel UDP packets as they arrive from the socket layer:

```
class MyChannelDataFeedListener : public ChannelDataFeedListener
{
public:

    virtual ChannelDataFeedControlCode onEvent(Channel* channel, DataFeed dataFeed, const
ChannelDataFeedEvent& event);

    virtual ChannelDataFeedControlCode onPacket(Channel* channel, DataFeed dataFeed, const
Packet& packet);

};

MyChannelDataFeedListener myChannelDataFeedListener;
```

Raw data processing is initiated by connecting the channel object to data feed:

```
channel->connectDataFeed(dfIncremental, &myChannelDataFeedListener);
```

Raw data processing is stopped by disconnecting the channel object from the data feed:

```
channel->disconnectDataFeed(dfIncremental);
```

9.1 Processing Raw Data Packets

Raw data UDP packets are delivered to the client application via the onPacket callback function. It is expected the client application checks packet sequence number, iterates through messages of the packet and process them accordingly.

Note: The packets delivered to the channel data listener object are not sequenced, the client application has to handle packet duplicates, out-of-order packets and packet gaps, if required.

10 Natural Recovery

Natural recovery may reduce recovery time of frequently updated instruments. Natural recovery is not guaranteed and should be used in conjunction with snapshot recovery:

```
marketDataServiceOptions.gapRecoveryType = rtNaturalSnapshotRecovery;
```

Once recovery process is started for an instrument the handler starts building instrument natural recovery state and delivering notifications on natural recovery state updates to the instrument listener object via the onNaturalRecoveryEvent callback function. It also delivers incremental messages to the instrument listener object via the onNaturalRecoveryMessage callback function.

The client application may examine the natural recovery data or process natural recovery messages and consider the instrument recovered. In this case the client application returns the iccStopRecovery control code from the onNaturalRecoveryEvent or onNaturalRecoveryMessage callback functions to apply the instrument natural recovery state and finish the recovery process.

If no iccStopRecovery control code is returned by the client application the recovery process finishes automatically when the handler considers the instrument naturally recovered or a snapshot message is received for the instrument, whichever happens earlier.

11 Channel Statistics

At any point channel statistics can be retrieved:

```
ChannelStatistics channelStatistics;
channel->getStatistics(channelStatistics);
```

Channel statistics includes the following counters:

- number of messages delivered to the client application
- number of gaps detected by the handler
- number of UDP packets received by the handler

Tip: By polling channel statistics with the desired time interval rate parameters can be calculated.

12 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;
```

13 Socket Buffer Max Size

On a Linux operating system it may be needed to increase the maximal size of socket input buffer which is set too low by default:

```
sudo sysctl -w net.core.rmem_max=16777216
```

14 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
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
```

15 Change Log

Version	Date	Description of change
1.0.0	2014-12-23	Creation
1.0.1	2014-12-29	Added 9 Natural Recovery
1.0.2	2014-12-29	Added 10 Channel Statistics
1.0.3	2015-01-27	Added 12 Socket Buffer Max Size
1.0.4	2015-03-11	Added 9 Processing Raw Data
1.0.5	2015-03-20	Updated to recent API changes
1.0.6	2015-06-09	Updated to recent API changes

Contact us

sales@btobits.com
Phone: +1-888-378-0666

Global Headquarters
US Client Support and Delivery Center
EPAM Systems, Inc
41 University Drive
Suite 202
Newtown, PA 18940
Phone: +1-267-759-9000
Fax: +1-267-759-8989