Eurex T7 Engine

Programmer's Guide

Version 2.0.0 (2013-09-26)

Table of Contents

1 Introduction ... 3
2 Engine Construction ... 3
3 Engine Initialization ... 3
3.1 Market Data Services ... 5
3.2 Market Data Products ... 6
3.3 Market Data Instruments ... 6
3.4 Trading Sessions ... 8
4 Connecting ... 8
5 Subscribing Trading Session ... 9
6 Trader Logon ... 9
7 Processing Market Data Events ... 9
8 Processing Market Data Messages ... 9
9 Sending Trading Messages ... 10
10 Processing Trading Messages ... 11
11 Unsubscribing Trading Session ... 11
12 Trader Logout ... 11
13 Disconnecting ... 12
14 Engine Finalization ... 12
15 Engine Destruction ... 12
16 Engine Internals ... 12
16.1 Market Data Recovery ... 12
16.2 Market Data Multi-Instrument Messages ... 13
16.3 Market Data Service Statistics ... 13
16.4 Trading Session Recovery ... 13
16.5 Trading Session Reconnection ... 14
16.6 Trading Session States and Events ... 14
16.7 Trading Session Statistics ... 15
16.8 Logging Services ... 15
17 Client Samples ... 15
18 Appendix A. Configuration File ... 16
19 Appendix B. Interfaces ... 23
20 Change Log ... 25
Contact us ... 27

1 Introduction

This document provides guidance to C++ programmers on using the B2BITS Eurex T7 Engine.

This document assumes the reader is familiar with the following Eurex manuals:

- Eurex Functional Reference
- Eurex Market and Reference Data Interfaces – Manual
- Eurex Enhanced Trading Interface - Manual

Notes and tips are displayed in yellow.

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

2 Engine Construction

```
using namespace Eurex;
using namespace Eurex::MarketData;
using namespace Eurex::Trading;
```

The Eurex T7 Engine can be constructed in two ways:

Construct an instance of the engine by specifying the name of a configuration file:

```
T7Engine* engine = T7Engine::create("Configuration.xml");
```

Construct an instance of the engine by specifying the engine name and options:

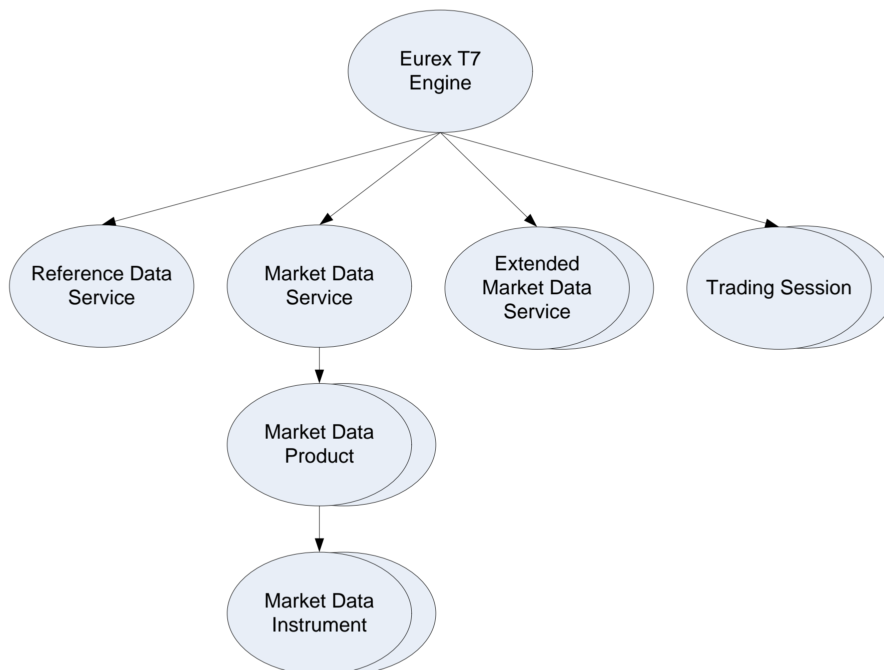
```
T7EngineOptions engineOptions;
engineOptions.threadCount = 16;
// ...

T7Engine* engine = T7Engine::create("Engine", engineOptions);
```

3 Engine Initialization

Engine initialization involves instantiation of trading sessions, market data services, products and instruments of application interest.

Engine static configuration can be defined in a configuration file and applied automatically during the initialization. Engine dynamic configuration can be performed with API calls manually any time after the initialization.



Optionally define an engine listener and attach it to the engine to process engine level events:

```
class MyT7EngineListener : public T7EngineListener
{
public:

    virtual void onMarketDataServiceAdded(T7Engine* engine, Service* service);

    virtual void onMarketDataServiceRemoving(T7Engine* engine, Service* service);

    virtual void onMarketDataProductAdded(T7Engine* engine, Product* product);

    virtual void onMarketDataProductRemoving(T7Engine* engine, Product* product);

    virtual void onMarketDataInstrumentAdded(T7Engine* engine, Instrument* instrument);

    virtual void onMarketDataInsrumentRemoving(T7Engine* engine, Instrument* instrument);

    virtual void onTradingSessionAdded(T7Engine* engine, Session* session);

    virtual void onTradingSessionRemoving(T7Engine* engine, Session* session);
};

MyT7EngineListener myEngineListener;

engine->setListener(&myEngineListener);
```

The engine listener is notified on events from the client thread.

Initialize the Eurex T7 Engine:

```
engine->open();
```

3.1 Market Data Services

Market data services defined in the configuration file are instantiated automatically. A market data service can be accessed by its name specified in the configuration file:

```
ReferenceDataService* referenceDataService = engine->getReferenceDataService("ReferenceData");
```

A market data service can be added manually by specifying the service name, type and options:

```
ServiceOptions serviceOptions;
serviceOptions.snapshotAddressA = "224.0.50.96";
// ...

ReferenceDataService* referenceDataService = engine->addReferenceDataService("ReferenceData",
serviceOptions);
```

The number of market data services instantiated depends on the application.

Note: The configuration files provided with the installation include all the market data services preconfigured for the Eurex production and simulation environments.

Optionally define a market data service listener and attach it to the service to process service level events:

```
class MyReferenceDataServiceEventListener : public ReferenceDataServiceEventListener
{
public:

    virtual void onReferenceDataBegin(Service* service);

    virtual void onReferenceDataEnd(Service* service);

    virtual void onProductDefinition(Service* service, const ProductDefinition*
productDefinition);

    virtual void onInstrumentDefinition(Service* service, const InstrumentDefinition*
instrumentDefinition);

    virtual void onInstrumentAdded(Service* service, const InstrumentDefinition*
instrumentDefinition);

    virtual void onInstrumentDeleted(Service* service, const InstrumentDefinition*
instrumentDefinition);
};

MyReferenceDataServiceEventListener myReferenceDataServiceEventListener;

referenceDataService->setEventListener(&myReferenceDataServiceEventListener);
```

A service listener is notified on events from non-concurrent threads – no synchronization is required. However, if the same service listener is shared across several services it can be notified from concurrent threads and synchronization is required.

3.2 Market Data Products

Products defined in the configuration file are instantiated automatically. A product can be accessed by its id specified in the configuration file:

```
Product* product = marketDataService->getProduct(90);
```

A product can be added manually by specifying the product definition received from the reference data service:

```
Product* product = marketDataService->addProduct(productDefinition);
```

A product can be added manually by specifying the product id and options:

```
ProductOptions productOptions;
productOptions.snapshotAddressA = "224.0.50.99";
// ...

Product* product = marketDataService->addProduct(90, productOptions);
```

The product configuration data is used by the Eurex T7 Engine to connect to the appropriate market data multicast channels, filter out unwanted product messages efficiently on the network, OS and application levels.

Optionally define a product listener and attach it to the product to process product level events:

```
class MyProductMessageListener : public ProductMessageListener
{
public:

    virtual void onReset(Product* product, ResetReason resetReason);

    virtual void onMessage(Product* product, FIXMessage* message, PerformanceIndicator*
performanceIndicator);
};

MyProductMessageListener myProductMessageListener;

product->setMessageListener(&myProductMessageListener);
```

A product listener is notified on events from non-concurrent threads – no synchronization is required. However, if the same product listener is shared across several products it can be notified from concurrent threads and synchronization is required.

3.3 Market Data Instruments

Instruments defined in the configuration file are instantiated automatically. An instrument can be accessed by its id specified in the configuration file:

```
Instrument* instrument = product->getInstrument(15);
```

An instrument can be added manually by specifying the instrument definition received from the reference data service:

```
Instrument* instrument = product->addInstrument(instrumentDefinition);
```

An instrument can be added manually by specifying the instrument id and type:

```
Instrument* instrument = product->addInstrument(15, itFuturesContract);
```

The instrument configuration data is used by the Eurex T7 Engine to filter out unwanted instrument messages on the application level.

Tip: Add instruments of a product while the product is disconnected. You can still add intra-day complex instruments.

Optionally define an instrument listener and attach it to the instrument to process instrument level events:

```
class MyInstrumentListener : public InstrumentEventListener
{
public:

    virtual void onReset(Instrument* instrument);

    virtual void onStateUpdate(Instrument* instrument, InstrumentStatus status,
InstrumentState state);

    virtual void onTopOfOrderBookUpdate(Instrument* instrument, const TopOfOrderBook&
topOfOrderBook);

    virtual void onOrderBookUpdate(Instrument* instrument, const OrderBook* orderBook);

    virtual void onTrade(Instrument* instrument, const Trade* trade);

    virtual void onTradeReversal(Instrument* instrument, const TradeReversal* tradeReversal);

    virtual void onTradeStatisticsUpdate(Instrument* instrument, const TradeStatistics*
tradeStatistics);

    virtual void onQuoteRequest(Instrument* instrument, const QuoteRequest* quoteRequest);

    virtual void onCrossRequest(Instrument* instrument, const CrossRequest* crossRequest);
};

MyInstrumentEventListener myInstrumentEventListener;

instrument->setEventListener(&myInstrumentEventListener, ieAllEvents);
```

An instrument listener is notified on events from non-concurrent threads – no synchronization is required. However, if the same instrument listener is shared across several instruments it can be notified from concurrent threads and synchronization is required.

Tip: Specify instrument events of interest when attaching an instrument listener.

3.4 Trading Sessions

Trading sessions defined in the configuration file are instantiated automatically. A trading session can be accessed by its name specified in the configuration file:

```
Session* orderSession = engine->getTradingSession("OrderSession");
```

A trading session can be added manually by specifying the session name and options:

```
SessionOptions sessionOptions;
sessionOptions.primaryAddress = "193.29.89.65";
// ...

Session* orderSession = engine->addTradingSession("OrderSession", sessionOptions);
```

The number of trading sessions instantiated depends on the application.

Define a trading session listener and attach it to the session to process session level events:

```
class MySessionMessageListener : public SessionMessageListener
{
public:

    virtual void onSessionEvent(Session* session, SessionEvent event, const std::string&
text);

    virtual void onMessage(Session* service, const MessageOut& messageOut);
};

MySessionMessageListener mySessionMessageListener;

orderSession->setMessageListener(&mySessionMessageListener);
```

4 Connecting

Connect a market data service to join UDP multicast groups and start receiving market data messages:

```
service->connect();
```

Once a reference data service (RDI) is connected the application receives a reference data snapshot cycle followed by any reference data incremental messages.
Once a market data service (EMDI / MDI) is connected the application receives market data snapshot messages followed by incremental messages for all the instruments of all the products configured for the service.

Connect a trading session to establish a TCP connection and perform the session logon procedure:

```
session->connect();
```

Once connected the session is active and receives exchange notifications. Nevertheless additional steps are required to setup the session for a particular scenario - order management, order drop copy, trade capture, etc.

5 Subscribing Trading Session

By default every trading session is subscribed to the Session Data and Risk Control notification streams.

A trading session can be selectively subscribed to an arbitrary number of additional notification streams (Listener Broadcast, Trades, News) to start receiving notification messages of the corresponding type:

```
System::u32 subscriptionId = tradeSession->subscribeTrades();
```

The subscription id is used to identify and unsubscribe from the notification stream.

6 Trader Logon

One or more traders can be logged on to a trading session to send request messages:

```
orderSession->logonTrader(traderId, traderPassword);
```

7 Processing Market Data Events

Market data processing is event driven. The application can process events such as order book updates or trades inside the appropriate event handlers of the attached listeners on the instrument, product and service levels.

All the parameters passed to an event handler by value or by reference are only valid inside that handler. The application should copy the appropriate data if required.

Tip: Use the event API for short time to market.

8 Processing Market Data Messages

Market data message processing is event driven. The application can process FIX messages such as MarketDataSnapshotFullRefresh (W) or MarketDataIncrementalRefresh (X) inside the onMessage event handler of the attached message listeners on the product and service levels.

A FIX message passed to the onMessage event handler by reference is only valid inside that handler. The application should copy the appropriate tag values if required.

Tip: Use the message API for custom message processing.

Examine the type of a message with the type member function:

```
if (message->type() == "W")
{
    // ...
}
```

Check the presence of an optional field with the hasValue member function:

```
if (message->hasValue(FIXFields::RefreshIndicator))
{
    // ...
}
```

Iterate through entries of a repeating group:

```
int noEntries = message->getAsInt(FIXFields::NoMDEntries);
FIXGroup* entriesGroup = message->getGroup(FIXFields::NoMDEntries);

for (int entryIndex = 0; entryIndex < noEntries; ++ entryIndex)
{
     TagValye* entry = entriesGroup->getEntry(entryIndex);
     // ...
}
```

9 Sending Trading Messages

Construct a message to be sent to a trading session:

```
NewOrderComplexRequest newOrderComplexRequest;
```

Fill all the required and any optional fields of the message as appropriate:

```
newOrderComplexRequest.setOrdType(OrdType_Limit);
// ...
```

Set the counter field of a repeating group first and then entries of the repeating group:

```
newOrderComplexRequest.setNoLegs(2);
LegOrdGrpComp& orderLeg1 = newOrderComplexRequest.LegOrdGrp(0);
orderLeg1.setLegPositionEffect(LegPositionEffect_Open);
LegOrdGrpComp& orderLeg2 = newOrderComplexRequest.LegOrdGrp(1);
orderLeg2.setLegPositionEffect(LegPositionEffect_Close);
// ...
```

Send the message to the session specifying sending trader id:

```
u32 sequenceNumber = session->sendMessage(userId, newOrderSingleRequest);
```

The message sequence number can be used to correlate request and response messages.

Tip: Reuse message objects when sending messages repeatedly to improve performance.

10 Processing Trading Messages

Trading session message processing is event driven. The application can process response and notification messages inside the onMessage event handler of the attached message listener.

A message passed to the onMessage event handler by reference is only valid inside that handler. The application should copy the appropriate field values if required.

Examine the type of a message with the getTemplateID member function:

```
if (message->MessageHeaderOut().getTemplateID() == OrderExecResponse::TID)
{
    const OrderExeceResponse* orderExecResponse = (const OrderExeceResponse*) message;
    // ...
}
```

Check the presence of an optional field by comparing it to the NoValue constant:

```
if (orderExecResponse->getOrigClOrdID() != UInt64_NoValue)
{
    // ...
}
```

Iterate through entries of a repeating group:

```
for (Counter8 fillIndex = 0; fillIndex < orderExecResponse->getNoFills(); ++ fillIndex)
{
    const FillsGrpComp& fillEntry = orderExecResponse->FillsGrp(fillIndex);
    // ...
}
```

11 Unsubscribing Trading Session

Optionally unsubscribe a trading session from a notification stream to stop receiving notification messages of the corresponding type:

```
tradeSession->unsubscribe(subscriptionId);
```

12 Trader Logout

Optionally logout a trader to prevent from sending request messages:

```
orderSession->logoutTrader(traderId);
```

13 Disconnecting

Optionally disconnect a trading session to perform the session logout procedure and shutdown the TCP connection:

```
session->disconnect();
```

Optionally disconnect a market data service to leave the UDP multicast groups and stop receiving market data messages:

```
referenceDataService->disconnect();
```

14 Engine Finalization

Optionally finalize the instance of the engine to free resources allocated by the instance:

```
engine->close();
```

15 Engine Destruction

Destroy the instance of the engine to free memory occupied by the instance:

```
engine->destroy();
```

The instance pointer is no longer valid.

16 Engine Internals

16.1 Market Data Recovery

Market data recovery is a process of restoring possibly missed market data incremental messages via the snapshot messages disseminated periodically on the market data channels.

The Eurex T7 Engine initiates market data recovery automatically on the following events:

- first message after connect
- start of exchange day
- fail-over on the exchange
- restart of the exchange

- abnormal message sequence
- message sequence gap detected

Once a recovery event is detected the application is notified via the onReset event handler of the attached listeners. The application should invalidate its data until the next update event or snapshot message is received.

Tip: Use the recoveryDelay option to distinguish between lost and delayed messages.

16.2 Market Data Multi-Instrument Messages

For performance reasons repeating group entries of unwanted instruments are not removed from the multi-instrument MarketDataIncrementalRefresh (X) and SecurityMassStatus (CO) FIX messages. The client application should simply ignore the repeating group entries of unwanted instruments in these messages.

16.3 Market Data Service Statistics

At any point market data service statistics can be retrieved:

```
ServiceStatistics serviceStatistics;
service->getStatistics(serviceStatistics);
```

Market data service statistics includes the following counters:

- number of messages transferred to the application
- number of active channels
- number of decoded FAST messages
- number of processed UDP packets
- number of errors

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

16.4 Trading Session Recovery

Trading session recovery is a process of restoring trading messages possibly missed during a period of disconnection via the session retransmission mechanism.

Session Data and Risk Control messages recovery is initiated automatically on the trading session connect. Listener Broadcast, Trades and News messages recovery is initiated automatically on the corresponding subscription request.

Messages are recovered from the start of the business day in the case of a late join or from the last message received in the case of a restart during the business day.

Recovered messages can be identified with the ApplResendFlag (1352) message field.

16.5 Trading Session Reconnection

Once an unexpected session disconnection is detected the application is notified via the onSessionEvent event handler of the attached listener and the process of session reconnection is started.

The Eurex T7 Engine starts session reconnection automatically on the following events:

- communication failure
- ETI gateway failover
- forced session logout

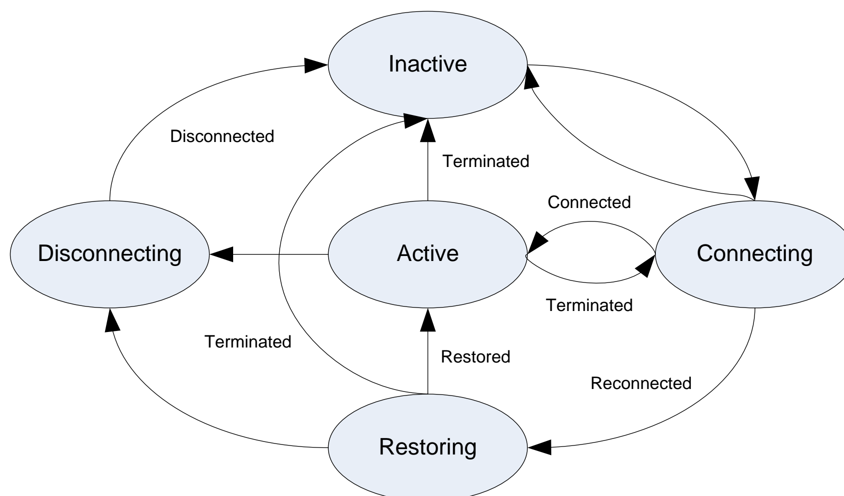
The Eurex T7 Engine restores the following session configuration automatically:

- session subscriptions
- trader logons

Once the session is reconnected the application is notified via the onSessionEvent event handler of the attached listener.

16.6 Trading Session States and Events

Trading session states identify various phases of the session life cycle, while trading session events identify transitions between the phases.



At any point the current state of the session can be retrieved:

```
SessionState sessionState = session->getState();
```

The application is notified on session events via the onSessionEvent event handler of the attached listener.

16.7 Trading Session Statistics

At any point trading session statistics can be retrieved:

```
SessionStatistics sessionStatistics;
session->getStatistics(sessionStatistics);
```

Trading session statistics includes the following counters:

- number of sent messages
- number of send errors
- number of received messages
- number of receive errors
- number of reconnections

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

16.8 Logging Services

The Eurex T7 Engine provides a number of options to control message logging on the market data service and trading session level.

The engine can be configured to log application-level messages sent to and received from the application and/or all the messages sent to and received from sockets to text log files for troubleshooting.

Additionally the FIXAntenna Engine logging options specify whether error, warning and informative text messages are logged to the FIXAntenna Engine log.

Tip: Disable logging to improve performance

17 Client Samples

EurexMarketDataClient is a simple interactive console application that processes reference data, connects to the products of interest and displays product states, instrument states, order books, trade statistics, trades and quote requests.

EurexTradingClient is a simple interactive console application that sets up trading sessions of different types, enters orders and quotes, processes execution reports, order drop copy, trade and news notifications.

The source code of the applications demonstrates how to use the Eurex T7 Engine API and process Eurex market data events and trading session messages.

18 Appendix A. Configuration File

`<engine>`

This root element defines configuration of the Eurex T7 Engine.

`<engine>.[name]`

This required text attribute defines the unique name of the engine.

`<engine>.[localAddress]`

This optional text attribute defines the IP address of the local network interface to use by the engine. Default value is "0.0.0.0".

`<engine>.[threadCount]`

This optional numeric attribute defines the number of concurrent threads used to send and receive data. Default value is "8".

`<engine>.<logging>`

This optional element defines the engine logging options.

`<engine>.<logging>.[directory]`

This optional text attribute defines the path to the log files folder. Default value is "logs".

`<engine>.<marketData>.<service>`

This optional element defines configuration of a market data service.

`<engine>.<marketData>.<service>.[name]`

This required text attribute defines the unique name of the service.

`<engine>.<marketData>.<service>.[type]`

This required text attribute defines the type of the service.

- "RDI" – Reference Data Interface service
- "EMDI" – Enhanced Market Data Interface service
- "MDI" – Market Data Interface service
- "EMDS" – Extended Market Data Service service

`<engine>.<marketData>.<service>.[fixDictionary]`

This required text attribute defines the path to the FIX dictionary file of the service.

```
<engine>.<marketData>.<service>.[fastTemplates]
```

This required text attribute defines the path to the FAST templates file of the service.

```
<engine>.<marketData>.<service>.[cacheSize]
```

This optional numeric attribute defines the maximal number of messages to cache for the service. Default value is "10000".

```
<engine>.<marketData>.<service>.[recoveryDelay]
```

This optional numeric attribute defines the minimal delay in microseconds before the recovery procedure is initiated. Default value is "1000".

```
<engine>.<marketData>.<service>.<snapshot>
```

This optional element defines configuration of the service snapshot channels.

```
<engine>.<marketData>.<service>.<snapshot>.<primary>
```

This optional element defines configuration of the service snapshot channel A.

```
<engine>.<marketData>.<service>.<snapshot>.<primary>.[address]
```

This optional text attribute defines the IP address of the snapshot channel A.

```
<engine>.<marketData>.<service>.<snapshot>.<primary>.[port]
```

This optional numeric attribute defines the port number of the snapshot channel A.

```
<engine>.<marketData>.<service>.<snapshot>.<secondary>
```

This optional element defines configuration of the service snapshot channel B.

```
<engine>.<marketData>.<service>.<snapshot>.<secondary>.[address]
```

This optional text attribute defines the IP address of the snapshot channel B.

```
<engine>.<marketData>.<service>.<snapshot>.<secondary>.[port]
```

This optional numeric attribute defines the port number of the snapshot channel B.

```
<engine>.<marketData>.<service>.<snapshot>.<connection>
```

This optional element defines configuration of the service snapshot connections.

```
<engine>.<marketData>.<service>.<snapshot>.<connection>.[receiveBufferSize]
```

This optional numeric attribute defines the size of the socket receive buffers. 0 indicates the system value. Default values is 4 Mb.

```
<engine>.<marketData>.<service>.<increment>
```

This optional element defines configuration of the service incremental channels.

```
<engine>.<marketData>.<service>.<increment>.<primary>
```

This optional element defines configuration of the service incremental channel A.

```
<engine>.<marketData>.<service>.<increment>.<primary>.[address]
```

This optional text attribute defines the IP address of the incremental channel A.

```
<engine>.<marketData>.<service>.<increment>.<primary>.[port]
```

This optional numeric attribute defines the port number of the incremental channel A.

```
<engine>.<marketData>.<service>.<increment>.<secondary>
```

This optional element defines configuration of the service incremental channel B.

```
<engine>.<marketData>.<service>.<increment>.<secondary>.[address]
```

This optional text attribute defines the IP address of the incremental channel B.

```
<engine>.<marketData>.<service>.<increment>.<secondary>.[port]
```

This optional numeric attribute defines the port number of the incremental channel B.

```
<engine>.<marketData>.<service>.<increment>.<connection>
```

This optional element defines configuration of the service incremental channel connections.

```
<engine>.<marketData>.<service>.<increment>.<connection>.[receiveBufferSize]
```

This optional numeric attribute defines the size of the socket receive buffers. 0 indicates the system value. Default values is 4 Mb.

```
<engine>.<marketData>.<service>.<product>
```

This optional element defines configuration of a product.

```
<engine>.<marketData>.<service>.<product>.[id]
```

This required numeric attribute defines the id of the product.

```
<engine>.<marketData>.<service>.<product>.[marketDepth]
```

This required numeric attribute defines the maximal depth of the order books of the product.

```
<engine>.<marketData>.<service>.<product>.<snapshot>
```

This optional element defines configuration of the product snapshot channels.

```
<engine>.<marketData>.<service>.<product>.<snapshot>.<primary>
```

This optional element defines configuration of the product snapshot channel A.

```
<engine>.<marketData>.<service>.<product>.<snapshot>.<primary>.[address]
```

This optional text attribute defines the IP address of the snapshot channel A.

```
<engine>.<marketData>.<service>.<product>.<snapshot>.<primary>.[port]
```

This optional numeric attribute defines the port number of the snapshot channel A.

```
<engine>.<marketData>.<service>.<product>.<snapshot>.<secondary>
```

This optional element defines configuration of the service snapshot channel B.

```
<engine>.<marketData>.<service>.<product>.<snapshot>.<secondary>.[address]
```

This optional text attribute defines the IP address of the snapshot channel B.

```
<engine>.<marketData>.<service>.<product>.<snapshot>.<secondary>.[port]
```

This optional numeric attribute defines the port number of the snapshot channel B.

```
<engine>.<marketData>.<service>.<product>.<increment>
```

This optional element defines configuration of the product incremental channels.

```
<engine>.<marketData>.<service>.<product>.<increment>.<primary>
```

This optional element defines configuration of the service incremental channel A.

```
<engine>.<marketData>.<service>.<product>.<increment>.<primary>.[address]
```

This optional text attribute defines the IP address of the incremental channel A.

```
<engine>.<marketData>.<service>.<product>.<increment>.<primary>.[port]
```

This optional numeric attribute defines the port number of the incremental channel A.

```
<engine>.<marketData>.<service>.<product>.<increment>.<secondary>
```

This optional element defines configuration of the service incremental channel B.

```
<engine>.<marketData>.<service>.<product>.<increment>.<secondary>.[address]
```

This optional text attribute defines the IP address of the incremental channel B.

```
<engine>.<marketData>.<service>.<product>.<increment>.<secondary>.[port]
```

This optional numeric attribute defines the port number of the incremental channel B.

```
<engine>.<marketData>.<service>.<product>.<instrument>
```

This optional element defines configuration of an instrument.

```
<engine>.<marketData>.<service>.<product>.<instrument>.[id]
```

This required numeric attribute defines the id of the instrument.

```
<engine>.<marketData>.<service>.<product>.<instrument>.[type]
```

This required text attribute defines the type of the instrument.

- "FuturesContract"
- "FuturesSpread"
- "OptionsSeries"
- "OptionsStandardStrategy"
- "OptionsNonStandardStrategy"
- "OptionsVolatilityStrategy"

```
<engine>.<marketData>.<service>.<logging>
```

This optional element defines logging options of the service.

```
<engine>.<marketData>.<service>.<logging>.[logClientMessages]
```

This optional boolean attribute defines whether to log messages received by the client. Default value is false.

```
<engine>.<marketData>.<service>.<logging>.[logAllMessages]
```

This optional boolean attribute defines whether to log all messages received by the service. Default value is false.

```
<engine>.<trading>.<session>
```

This optional element defines configuration of a trading session.

```
<engine>.<trading>.<session>.[name]
```

This required text attribute defines the unique name of the session.

```
<engine>.<trading>.<session>.[id]
```

This required numeric attribute defines the id of the session.

```
<engine>.<trading>.<session>.[password]
```

This required text attribute defines the password of the session.

```
<engine>.<trading>.<session>.<primary>
```

This optional element defines configuration of the primary connection gateway.

```
<engine>.<trading>.<session>.<primary>.[address]
```

This required text attribute defines the primary connection gateway IP address.

```
<engine>.<trading>.<session>.<primary>.[port]
```

This required numeric attribute defines the primary connection gateway port.

```
<engine>.<trading>.<session>.<secondary>
```

This optional element defines configuration of the secondary connection gateway.

```
<engine>.<trading>.<session>.<secondary>.[address]
```

This required text attribute defines the secondary connection gateway IP address.

```
<engine>.<trading>.<session>.<secondary>.[port]
```

This required numeric attribute defines the secondary connection gateway port.

```
<engine>.<trading>.<session>.<application>
```

This required text element defines parameters of the application.

```
<engine>.<trading>.<session>.<application>.[systemName]
```

This required text attribute defines the application system name.

```
<engine>.<trading>.<session>.<application>.[systemVersion]
```

This required text attribute defines the application system version.

```
<engine>.<trading>.<session>.<application>.[systemVendor]
```

This required text attribute defines the application system vendor.

```
<engine>.<trading>.<session>.<application>.[usageOrders]
```

This required text attribute defines the application usage of orders.

```
<engine>.<trading>.<session>.<application>.[usageQuotes]
```

This required text attribute defines the application usage of quotes.

```
<engine>.<trading>.<session>.<application>.[orderRouringIndicator]
```

This required boolean attribute defines the application order routing capabilities.

```
<engine>.<trading>.<session>.<connection>
```

This optional element defines configuration of the session connection.

```
<engine>.<trading>.<session>.<connection>.[connectMaxFailureCount]
```

This optional numeric attribute defines the maximal number of attempts to connect before giving up. Default value is 1.

```
<engine>.<trading>.<session>.<connection>.[reconnectAutomatically]
```

This optional boolean attribute defines whether to start reconnection automatically. Default value is true.

```
<engine>.<trading>.<session>.<connection>.[reconnectMaxFailureCount]
```

This optional numeric attribute defines the maximal number of attempts to reconnect before giving up. 0 indicates no limit. Default value is 0.

```
<engine>.<trading>.<session>.<connection>.[connectInterval]
```

This optional numeric attribute defines the time interval in seconds between subsequent attempts to connect. Default value is 60 s.

```
<engine>.<trading>.<session>.<connection>.[heartbeatInterval]
```

This optional numeric attribute defines the heartbeat interval in milliseconds. Default value is 10000 ms.

```
<engine>.<trading>.<session>.<connection>.[sendNoDelay]
```

This optional boolean attribute defines whether to disable Nagle's algorithm. Default value is true.

```
<engine>.<trading>.<session>.<connection>.[sendBufferSize]
```

This optional numeric attribute defines the size of the socket send buffer. 0 indicates the system value. Default value is 0.

```
<engine>.<trading>.<session>.<connection>.[receiveBufferSize]
```

This optional numeric attribute defines the size of the socket receive buffer. 0 indicates the system value. Default value is 0.

```
<engine>.<trading>.<session>.<logging>
```

This optional element defines logging options of the session.

```
<engine>.<trading>.<session>.<logging>.[logClientMessages]
```

This optional boolean attribute defines whether to log application-level messages sent and received by the application. Default value is false.

```
<engine>.<trading>.<session>.<logging>.[logAllMessages]
```

This optional boolean attribute defines whether to log all messages sent and received by the session. Default value is false.

19 Appendix B. Interfaces

```
Eurex::T7Engine
```

This interface represents an instance of the Eurex T7 Engine.

```
Eurex::T7EngineListener
```

This listener interface is used not notify the application on engine level events.

`Eurex::MarketData::Service`

This interface represents a market data service.

`Eurex::MarketData::ServiceMessageListener`

This listener interface is used to notify the application on FIX messages of the RDI and EMDS services.

`Eurex::MarketData::ReferenceDataService`

This interface represents the RDI market data service.

`Eurex::MarketData::ReferenceDataServiceEventListener`

This listener interface is used to notify the application on reference data events.

`Eurex::MarketData::MarketDataService`

This interface represents the EMDI and MDI market data services.

`Eurex::MarketData::Product`

This interface represents a market data product (e.g. FGBL).

`Eurex::MarketData::ProductDefinition`

This interface represents the definition of a market data product.

`Eurex::MarketData::ProductEventListener`

This listener interface is used to notify the application on product events.

`Eurex::MarketData::ProductMessageListener`

This listener interface is used to notify the application on FIX messages of a product.

`Eurex::MarketData::Instrument`

This interface represents a market data instrument (e.g. FGBL.S.MAR13.JUN13)

`Eurex::MarketData::InstrumentDefinition`

This interface represents the definition of an instrument.

`Eurex::MarketData::InstrumentEventListener`

This listener interface is used to notify the application on instrument events.

```
Eurex::MarketData::OrderBook
```

This interface represents the order book of an instrument.

```
Eurex::MarketData::OrderBookSide
```

This interface represents one of the two sides of an order book (e.g. Bid).

```
Eurex::MarketData::Trade
```

This interface represents a market data trade event.

```
Eurex::MarketData::TradeReversal
```

This interface represents a market data trade reversal event.

```
Eurex::MarketData::TradeStatistics
```

This interface represents the trade statistics of an instrument.

```
Eurex::MarketData::QuoteRequest
```

This interface represents a market data quote request event.

```
Eurex::MarketData::CrossRequest
```

This interface represents a market data cross request event.

```
Eurex::Trading::Session
```

This interface represents a ETI trading session.

```
Eurex::Trading::SessionMessageListener
```

This listener interface is used to notify the application on trading session events.

20 Change Log

Version	Date	Description of change
1.0.0	2012-11-29	Creation
1.0.1	2012-12-10	Added 11 Appendix A. Configuration File Added 12 Appendix B. Interfaces
1.0.2	2012-12-22	Added 9.4 Multi-Instrument Messages Added a tip to 9.1 Recovery

		Updated description of the recoveryDelay option
1.1.0	2013-03-01	Added 5 Processing Market Data Events
		Added 6 Processing Market Data Messages
		Updated to reflect recent API changes
1.1.1	2013-03-06	Added 1 Introduction
1.1.2	2013-06-28	Updated to reflect recent API changes
2.0.0	2013-09-26	Added ETI sections
		Updated to reflect 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