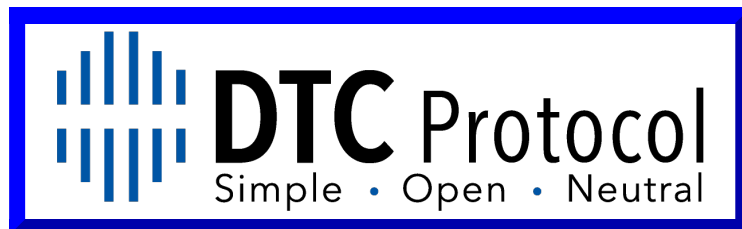


DTC Messages and Procedures

- [Introduction](#)
- [DTC Protocol Procedures](#)
 - [Client and Server Support of DTC Messages](#)
 - [Message Structure](#)
 - [Get and Set Functions](#)
 - [Versioning](#)
 - [Binary Encoding with Fixed Length Strings](#)
 - [Binary Encoding with Variable Length Strings](#)
 - [Protocol Buffer Encoding](#)
 - [Breaking Changes](#)
 - [Header File Namespaces](#)
 - [Symbols](#)
 - [Multiple Connections](#)
 - [Basic Server Procedures](#)
 - [Connection and Logon Sequence](#)
 - [Encoding Request Sequence](#)
 - [Logging Off](#)
 - [Market Data](#)
 - [Market Data Price Format](#)
 - [Trade Order Procedures](#)
 - [Bracket Order Procedures](#)
 - [Integer Trading Messages](#)
 - [Automatic Trading Data Updates](#)
 - [Indication of Unset Message Fields](#)
 - [Historical Price Data](#)
 - [Symbol ID and Request ID Rules](#)
 - [Nonstandard Messages](#)
- DTC Messages
 - [All DTC Messages](#)
 - [Authentication and Connection Monitoring Messages](#)
 - [Market Data Messages](#)
 - [Order Entry and Modification Messages](#)
 - [Trading Related Messages](#)
 - [Account List Messages](#)
 - [Symbol Discovery and Security Definitions Messages](#)
 - [Account Balance Data Messages](#)
 - [Logging Messages](#)
 - [Historical Price Data Messages](#)
- [DTC Message Field Type Descriptions](#)



- [\[int32\] 32-bit Integer](#)
- [\[unsigned int16\] Unsigned 16-bit Integer](#)
- [\[unsigned int32\] Unsigned 32-bit Integer](#)
- [\[EncodingEnum\] Encoding Enumeration](#)
- [\[char\] Character](#)
- [\[unsigned int8\] Byte](#)
- [\[double\] 64-bit Floating Point Value](#)
- [\[float\] 32 bit Floating Point Value](#)
- [\[AtBidOrAskEnum\] At Bid Or Ask Enumeration](#)
- [\[LogonStatusEnum\] Logon Status Enumeration](#)
- [\[t_DateTime\] Date Time](#)
- [\[t_DateTimeWithMilliseconds\] Date Time With Milliseconds](#)
- [\[t_DateTimeWithMicrosecondsInt\] Date Time With Microseconds](#)
- [\[t_DateTime4Byte\] 4 Byte UNIX Date-Time](#)
- [\[MarketDataFeedStatusEnum\] Market Data Feed Status Enumeration](#)
- [\[BuySellEnum\] Buy Sell Enumeration](#)
- [\[RequestActionEnum\] Request Action Enumeration](#)
- [\[MarketDepthUpdateTypeEnum\] Market Depth Update Type Enumeration](#)
- [\[PriceDisplayFormatEnum\] Price Display Format Enumeration](#)
- [\[PutCallEnum\] Put Call Enumeration](#)
- [\[SecurityTypeEnum\] Security Type Enumeration](#)
- [\[SearchTypeEnum\] Search Type Enumeration](#)
- [\[OrderTypeEnum\] Order Type Enumeration](#)
- [\[TimeInForceEnum\] Time In Force Enumeration](#)
- [\[OrderStatusEnum\] Order Status Enumeration](#)
- [\[HistoricalDataIntervalEnum\] Historical Data Interval Enumeration](#)
- [\[OrderUpdateReasonEnum\] Order Update Reason Enumeration](#)
- [\[OpenCloseTradeEnum\] Open Close Trade Enumeration](#)
- [\[HistoricalPriceDataRejectReasonCodeEnum\] Historical Price Data Reject Reason Code Enumeration](#)
- [\[PartialFillHandlingEnum\] Partial Fill Handling Enumeration](#)
- [\[FinalUpdateInBatchEnum\] Final Update In Batch Enumeration](#)

Introduction

This page documents the **Data and Trading Communications (DTC) Protocol** procedures for working with DTC messages, the DTC messages themselves, and the fields of those messages.

For an overview of the DTC Protocol, refer to the [Data and Trading Communications Protocol](#) page.

For the DTC Protocol header files, refer to the [DTC Files](#) section. The files to use depend upon the particular [encoding](#) to use. In the case of JSON encoding, there are no header files to use.

If you will be adopting DTC, either as a Client or Server, then you can take a copy of this documentation and modify it to meet your requirements. It has no copyright.

Client: Any application program supporting DTC. This can be a manual trading, automated/algorithmic trading, charting, or market analysis program, a combination of these, or whatever.

Server: Any service/program that provides Market Data and/or Trading functionality that follows this DTC protocol.

DTC Protocol Procedures

This section documents the basic procedures of using DTC Protocol messages by the Client and Server.

The DTC protocol is nothing more than a messaging protocol with messages containing various fields of information.

It is the exchange of messages between a Client and Server over a network socket. Various [Message Encodings](#) are supported.

Client and Server Support of DTC Messages

The Client and the Server will only support the messages that they need to. Not all of them are required to be supported with DTC.

If a Server does not support a particular message or class of messages, it will indicate this through various 1 byte integer fields in the [LOGON_RESPONSE](#) message. A byte value of 1 signifies the field is TRUE. A byte value of 0 signifies the field is FALSE.

For example, if the Server does not support Security Definitions or Trading, it will set these **LOGON_RESPONSE** fields to zero: **s_LogonResponse::SecurityDefinitionsSupported** and **s_LogonResponse::TradingIsSupported**. The default value for these flag fields is zero.

Message Structure

In the case of simple binary encoding, a DTC message consists of a 2 byte Size and 2 byte message Type at the beginning of every message structure. These 2 fields are automatically set by the message constructor in the standard header file

Get and Set Functions

In the case of the DTC encoding method which uses fixed length structures and strings, each message structure has **Get*** and **Set*** functions for each text string field of the structure.

Each fixed length structure has a **Get*** function for all other fields of the structure except for the **Type** field.

The DTC protocol structures do not use **Set*** functions for non-text strings fields.

There are some reasons for the **Get*** and **Set*** functions. In the case of text strings, one reason is to simplify use of text string fields. You can call a simple function to get and set a text string safely. Another reason is to guard against access violations when accessing string and other fields which might not exist when the remote side is using an older protocol version (this will be a less likely case).

The Get and Set functions follow a standard naming method. They are named **Get[field name]** and **Set[field name]**.

If you add new messages to the DTC Protocol or add new fields to existing data structures, then it is a good idea to add Get and Set functions following the above requirements.

There are 2 special functions on each message structure which are described below:

- **GetMessageSize()** : This returns the size of the message structure in bytes. In the case where this function is called on a message structure received over the network, this gets the size of the message received over the network. Otherwise, it is the size of the message structure according to the version of the header file that you have.
- **CopyFrom(void * p_SourceData)**: This function is used to copy data into a message structure instance, from a memory block/buffer that contains the message received over the network. The beginning of the message from a memory block/buffer needs to be pointed to by the **p_SourceData**. This function internally ensures that the smaller size of either the message structure instance being copied into or the message pointed to by p_SourceData, is used for the copy. The purpose of this does to guard against any access violations due to protocol differences between the Client and the Server.

Versioning

In the DTC Protocol, the version of messages is an important consideration. How a Client and Server handle messages from the other side with differing DTC Protocol versions, depends upon the encoding method.

There are many special considerations in the case of Binary Encoding with Fixed Length Strings and Binary Encoding with Variable Length Strings.

In the case of JSON encoding, being the data is encoded as text, the only potential issue is that a field may be missing in the case of when reading a message when the other side (Client or Server) is using a lower DTC protocol version.

When there are new messages added to the DTC Protocol or new fields added to a structure, then the protocol version number in the [LOGON_REQUEST](#) and [LOGON_RESPONSE](#) messages will be incremented.

Binary Encoding with Fixed Length Strings

In the case of binary encoding, existing defined structure fields will not change. New fields can be added to an existing message at the end of the structure.

Get functions are provided for accessing the structure fields safely. These functions will do structure size checks to ensure there are no memory access violations. Set functions are supported for text strings.

What this means is as follows:

Whether you are the Client or the Server, if you see a newer protocol version from the other side, then there will be no issues. In this case you cannot do anything with unknown message types, however you will read them from the socket and discard them. Every message has a standard header with its size. You will not have access to newer structure fields, but you will know the existing structure fields you have knowledge of, will be filled in.

Whether you are the Client or the Server, if you see an older protocol version from the other side, the newer message types that you have knowledge of, that the other side with the older version does not, are not supported, but it is not harmful to send them.

In the case of newer structure fields on existing message types, the Get functions will safeguard against any memory access violations. When getting a value from a newer unsupported field, the Get function will return the default value.

The way that protocol safety is accomplished is through the use of Get functions. These functions look at the Size of the message sent by the sender, look at the base address of the structure object, look at the address of the field they are accessing and the size of the field. If it is determined that the structure field does not exist, it is not accessed in the default value is returned instead.

Handling a Newer Protocol Version: Normally there is nothing special that needs to be done when communicating with the other side that has a newer protocol version. However, there is one special consideration when reading messages. Make certain that when reading messages from the network socket, that you only read the number of bytes for a message according to the **Size** in its header. This should be obvious.

However, there is another important consideration where there could be a problem. If you read a message, that has a larger size than the corresponding message structure in the header file version you have, then when you make a copy of that message into a destination structure, you must only copy the number of bytes according to the size of the structure in the header file that you have. The **CopyFrom** member function on all message structures supports this.

Binary Encoding with Variable Length Strings

In the case of Binary Encoding with Variable Length Strings, the handling of new fields in a message and new messages is identical to Binary Encoding with Fixed Length Strings (described above).

Protocol Buffer Encoding

In the case of Google Protocol Buffer encoding, there is no issue, with additional fields added. However, changes to existing fields, potentially can be a problem. It depends upon what the specific change is and how the protocol handles this.

Breaking Changes

The DTC Protocol does support breaking changes, where a new protocol version is not back compatible with older protocol versions. This is not a common occurrence. And will be well documented when it occurs.

Header File Namespaces

The header file for [DTC Binary Encoding with Fixed Length Strings](#) has the data structures located in the **DTC** namespace.

The header file for [DTC Binary Encoding with Variable Length Strings](#), has the data structures located within the **DTC_VLS** namespace.

There is no header file for JSON encoding. Therefore, there is no namespace for JSON encoding.

Example:

```
DTC_VLS::s_LogonRequest
```

Symbols

The DTC Protocol supports any Symbol format with an optional Exchange specifier.

DTC requires the use of the actual Symbol for a particular market/security in all messages where a Symbol is required. The Symbol can be anything that the Server defines.

The symbol can define any kind of market/security including futures, stocks, spreads, strategies, options, indices, funds, bitcoins, or whatever. Multi-legged securities like spreads, also can be defined through a unique text a string identifier.

In the case of a futures contract, futures spread, or option symbol, it is necessary for the particular contract month and year and other parameters to be part of the symbol text string.

The combination of the Symbol and optional Exchange are the complete identification for a particular market/security and will either express or imply the particular Security Type. While the Security Type is supported in security definition responses, it is not necessary to ever specify the Security Type when identifying a particular market/security.

There shall be no need for a Client to convert some type of numeric identifier to a Symbol or to an Exchange. The Client will work directly with Symbol and Exchange identifiers. If the Server requires a Symbol to internal identifier mapping, it is the responsibility of the Server to perform that on the Server side.

When subscribing to market data, the DTC Protocol uses an integer identifier in the market data request messages (SymbolID) which is sent to the Server along with the Symbol and Exchange identifiers.

The Server uses that same integer identifier in all of the market data response messages to

uniquely associate the response with a particular Symbol and Exchange. This is done to reduce the space in the market data response messages which improves performance over a network.

Multiple Connections

The DTC Protocol can use multiple connections. For example, one connection could be used for streaming market data, another connection for trading, and another connection for historical price data.

Each of these connections will use the identical protocol and only use the message types relevant to the particular purpose of the connection.

Each of these connections can use a different encoding.

Basic Server Procedures

This section explains the basic top-level procedures for a DTC Protocol Server.

1. Assuming the DTC Server is connected to a TCP/IP network, which is usually the public Internet, the Server will listen for network connections on a particular Internet port number.
2. When an incoming connection is received, the network connection will be established by the underlying TCP protocol.
3. The Server will then wait for an [ENCODING_REQUEST](#) message. The server must implement support for this message. The server should expect this message to be received using [Binary Encoding](#), unless the the Client and Server have each decided upon another encoding to use in which case, the client will skip the step of sending an **ENCODING_REQUEST**.
4. Once the encoding has been established between the Client and Server, the next step is that the [Connection and Logon Sequence](#) needs to be followed.
5. After a successful Logon by the Client, the Client will send request messages to the Server and the Server will send the response messages.
6. The Server can either reject or ignore request messages until there has been a successful Logon by the Client.

Connection and Logon Sequence

The Client will connect to the Server at the specified IP address and port number.

Dynamically reconnecting to another specified Server is supported and is explained in the steps below.

The Client can optionally support launching an executable Server program upon connection. Therefore, Data or Trading service providers can create an executable program that acts as a TCP/IP socket Server if they want their Server to run on the user's computer. The Client will pass the port number on the command line when running this program (Port number parameter format: **/Port:[port number]**). This is the socket port the local Server executable will need to listen on. The Client will verify that the port passed to the Server executable program can be listened on before it starts the Server executable and passes that port number. The Client will always pass

an available port number.

The logon sequence from the Client to the Server is as follows:

1. A DTC Client establishes a TCP/IP connection to the Server. This can be either TLS or non-TLS. It is highly recommended the Server enables [TCP No Delay](#) when the Server accepts the connection.
2. It is important to understand that the DTC protocol uses different encodings for messages. For more information refer to [Message Format and Encoding](#). Messages must be exchanged between the Client and Server using the desired encoding.

If the Client does not know the default encoding used by the Server or the encoding needs to be changed, the Client must send an [Encoding Request Sequence](#) to request the DTC encoding to use before sending any other messages. If the Client knows the default encoding used by the Server, then it does not need to send an ENCODING_REQUEST message to the Server but it is preferred.

3. The Client fills out a [LOGON_REQUEST](#) message structure and sends it to the Server over the network socket. Only the message fields which are relevant for the particular Server being connected to, need to be set.

The [LOGON_REQUEST::HeartbeatIntervalInSeconds](#) must be set by the Client.

4. The Client will wait for a [LOGON_RESPONSE](#) message structure from the Server.

The Server will always respond with an [LOGON_RESPONSE](#) message. It must set the **s_LogonResponse::Result** field. The following are the supported constants for the Result field: **LOGON_SUCCESS**, **LOGON_ERROR**, **LOGON_ERROR_NO_RECONNECT**, **LOGON_RECONNECT_NEW_ADDRESS**.

It is recommended the Server set the **s_LogonResponse::ResultText** string field to free-form text indicating more descriptive text for the logon result whether the connection was successful or unsuccessful.

5. If the **Result** field is set to **LOGON_SUCCESS**, then this indicates to the Client a successful logon. At this time, the Client can interpret the other fields in the [LOGON_RESPONSE](#) message and send other messages to the Server.
6. If the **Result** field is set to either of **LOGON_ERROR** or **LOGON_ERROR_NO_RECONNECT**, then this is an unsuccessful logon and the Server will close the network connection. If the **Result** is **LOGON_ERROR_NO_RECONNECT**, then the Client should not try to reconnect to the Server again.
7. In the [LOGON_RESPONSE](#) message if the **s_LogonResponse::Result** member is set to **LOGON_RECONNECT_NEW_ADDRESS**, then this signals to the Client reconnect to the address given in

s_LogonResponse::ReconnectAddress.

8. The [LOGON_REQUEST](#) indicates a Heartbeat interval that each side needs to follow. Each side will send a [HEARTBEAT](#) message structure at the given interval in seconds.
9. At this point, the Client can make requests to the Server by sending any of the other supported [DTC Messages](#). If a particular class of messages is not supported by the Server, the Client will become aware of this by examining the related variables in the [LOGON_RESPONSE](#) message.

Encoding Request Sequence

The [ENCODING_REQUEST](#) and [ENCODING_RESPONSE](#) messages were added in version 6 of the DTC Protocol. These control the particular message encoding to be used during the connection.

Servers/Clients can support one or more encodings and the Encoding Request Sequence is used to discover the encoding to be used during the connection. The Server can accept or reject the requested message encoding from the Client.

The following sequence is used to request a particular message encoding between the Server and the Client.

1. It is supported in the DTC Protocol for the Server and the Client to agree on a particular encoding that they will use between each other and bypass the [Encoding Request Sequence](#) completely. If this is the case, the remaining steps should be skipped by the Client and the Server. However, this is not considered the DTC standard for Clients and Servers and is an exception to allow for more simple use of the DTC Protocol when Clients and Servers know they are only going to work with a particular message encoding.
2. In version 6 or higher of the DTC Protocol, all Servers must respond to the [ENCODING_REQUEST](#) message, even if it is only to reject a requested change and set the encoding to the current Server supported encoding value.
3. During the initial exchange of the [ENCODING_REQUEST](#) and [ENCODING_RESPONSE](#) messages at the beginning of the network connection, these must be both sent using [Binary Encoding](#).

However, there is an exception to this. If the Server has been preset to a particular encoding at the start of a connection and this has been disclosed to the Client by some other non-technical means, then the Client must send the [ENCODING_REQUEST](#) to the Server using the encoding the Server is currently set to and not necessarily Binary Encoding.

4. For a Client using JSON Encoding to be able to connect to a Server, the Client needs to form and process the initial Encoding Request and Encoding Response messages using the standard Binary Encoding [s_EncodingRequest message structure](#).
5. The Client sends an [ENCODING_REQUEST](#) message to the Server with the

- requested encoding. This message must be sent using [Binary Encoding](#) if it is the first message sent in the data stream during the connection. The Server must be able to respond to this message using Binary Encoding as well.
6. The Client waits for the [ENCODING_RESPONSE](#) from the Server. If the Client has not received a response within a reasonable time, like 20 seconds, the Client should close the network connection because it cannot reliably communicate with the Server until it knows the message encoding.
 7. The Server receives the [ENCODING_REQUEST](#) message, and responds with an [ENCODING_RESPONSE](#) message.

If the Server accepts the requested encoding, the **Encoding** field in the [ENCODING_RESPONSE](#) must be set to the value from the [ENCODING_REQUEST](#) message.

If the Server rejects the requested encoding, the **Encoding** field in the [ENCODING_RESPONSE](#) message must be set to the current encoding value used by the Server.

Servers that only support one encoding must still respond to an [ENCODING_REQUEST](#) by sending an [ENCODING_RESPONSE](#) message with the **Encoding** field set to the particular DTC encoding that the Server supports.

In either case, the [ENCODING_RESPONSE](#) is sent using the encoding the Server is currently using, before the requested change, which usually will be Binary Encoding if it is the first message at the beginning of the network connection. All subsequent messages are sent using the new encoding if it was accepted by the Server.

8. The Client receives the [ENCODING_RESPONSE](#) message and processes it using the original encoding which is Binary Encoding. The Client then changes encoding specified by the **Encoding** field in the [ENCODING_RESPONSE](#) message. All subsequent messages must be sent and received using the potential new encoding.
9. While the encoding request sequence is typically performed before sending the [LOGON_REQUEST](#) message, the sequence can technically be performed at anytime. The only restriction is that the Client should stop sending new messages and continue processing received messages with the current encoding until the encoding is changed with an [ENCODING_RESPONSE](#). This also includes processing the [ENCODING_RESPONSE](#) message with the current encoding. After that message the Client will use the new potential encoding.

Logging Off

A logoff can be performed by either side by sending a [LOGOFF](#) message to the other side and gracefully closing the socket with a linger.

The Server and the Client will always write complete messages into the socket. When either side

wants to logoff and disconnect it will send a [LOGOFF](#) message to the other side and gracefully close the socket.

The only exception to this is when the Server sends a [LOGON_RESPONSE](#) message to the Client with the **Result** field set to a value other than **LOGON_SUCCESS**, then it can gracefully close the network connection after without sending a [LOGOFF](#) message to the Client.

Market Data

A Client will subscribe and unsubscribe to market data for a symbol by using the [MARKET_DATA_REQUEST](#) message. When subscribing to market data for a symbol, the request is identified by a unique [SymbolID](#) field.

All messages from the Server relating to that market data request, will use this same **SymbolID** to associate the particular data message with the symbol. None of the market data messages from the Server use a symbol or exchange string.

After a Client subscribes to market data, the Server will respond with any of the **Server >> Client** [Market Data Messages](#) currently defined in the DTC Protocol.

The Server will implement and send what messages are appropriate for its implementation of the DTC Protocol for market data. The Client will implement support for all market data messages from the Server. Although the Client does not need to implement any of the Integer market data messages.

This section explains the procedures for the functioning of market data between the Server and Client for a particular symbol.

1. A Client will subscribe to market data by using the [MARKET_DATA_REQUEST](#) message.
2. If the Client has subscribed to an invalid symbol or there is some problem the Server finds with the [MARKET_DATA_REQUEST](#) message the Server will respond with a [MARKET_DATA_REJECT](#) message. At this point, there is nothing further the Server needs to respond with.
3. If the Client has subscribed to a valid symbol, the Server will respond immediately with a [MARKET_DATA_SNAPSHOT](#) message, providing as many of the message fields as possible.
4. When there is a change with the Bid or Ask prices or quantities for the symbol subscribed to, the Server will send a [MARKET_DATA_UPDATE_BID_ASK](#) message to the Client. If the Symbol only provides Bid and Ask updates, then this message will be used. **MARKET_DATA_UPDATE_TRADE** messages will not be sent in this case.
5. When a new trade occurs for the symbol subscribed to, the Server will send a [MARKET_DATA_UPDATE_TRADE](#) message to the Client.
6. When the session trade volume for the symbol changes, the Server can send a [MARKET_DATA_UPDATE_SESSION_VOLUME](#) message. However, for the complete details of when the session trade volume message should be sent the Client, refer to the documentation for

MARKET_DATA_UPDATE_SESSION_VOLUME.

7. When a new session High or Low occurs for the symbol, the Server will send a MARKET_DATA_UPDATE_SESSION_HIGH or MARKET_DATA_UPDATE_SESSION_LOW message.
8. When there is a change of the session Settlement price, session Open price or the Open Interest value for the symbol, the Server will send a MARKET_DATA_UPDATE_SESSION_SETTLEMENT, MARKET_DATA_UPDATE_SESSION_OPEN, or MARKET_DATA_UPDATE_OPEN_INTEREST message respectively.
9. If any of the other fields in a MARKET_DATA_SNAPSHOT message change and there is no corresponding **MARKET_DATA_UPDATE_*** message available for that field, then the Server will send a new MARKET_DATA_SNAPSHOT message. This message is intended to be sent infrequently.
10. If the Client requires market depth data beyond the best Bid and Ask level, then it will request it with a MARKET_DEPTH_REQUEST message. The Client shall only request market depth data if **s_LogonResponse::MarketDepthIsSupported** is set to 1.

If the market depth request is rejected for any reason, the Server will send a MARKET_DEPTH_REJECT message to the Client.

11. If the market depth request is accepted by the Server, then it will respond immediately with a series of MARKET_DEPTH_SNAPSHOT_LEVEL messages which the Client uses to build its initial market depth book.
12. In order to conserve bandwidth, the Server can optionally set **s_LogonResponse::MarketDepthUpdatesBestBidAndAsk** to 1 and when beginning to send market depth data in response to a MARKET_DEPTH_REQUEST, it should then stop best Bid and Ask updates through a MARKET_DATA_UPDATE_BID_ASK message.

The Server is not under an obligation to do this in the DTC protocol. In the case where the Server is sending market depth messages only, the Client will then use the updates at market depth level 1 to update the best Bid and Ask prices and it will know to do that because

s_LogonResponse::MarketDepthUpdatesBestBidAndAsk has been set to 1.

13. As the market depth data changes, the Server will send to the Client MARKET_DEPTH_UPDATE_LEVEL messages to update the market depth book. Each message updates one level of depth on one side.
14. If a Client makes a request for Security Definition data with the SECURITY_DEFINITION_FOR_SYMBOL_REQUEST message, the Server will respond to this request with a SECURITY_DEFINITION_RESPONSE and provide as many of the message fields as possible.
15. It is recommended the Client always makes a Security Definition For Symbol request to the Server at the time of making a market data request, to obtain the available Security Definition data.

Market Data Price Format

In the **MARKET_DATA_*** and **MARKET_DEPTH_*** messages, price values can be stored in two formats. These are double precision floating point and single precision floating point formats.

The standard format is double precision floating point which is consistent with the FIX protocol.

As of August 2020, integer market data messages which use integers to hold price values, are discontinued.

If a market data/depth message ends with **_COMPACT**, the price values are stored in single precision floating-point format.

In the case when the server sends market data messages using a double or float value for the price related fields, it may require that a multiplier be applied. The multiplier is the **DisplayPriceMultiplier** field in the [SECURITY_DEFINITION_RESPONSE](#) message which is requested by the Client with the [SECURITY_DEFINITION_FOR_SYMBOL_REQUEST](#) message.

A Client should implement support for all market/depth data messages sent from the Server, to be compatible with any Server no matter what messages the Server chooses to use.

Trade Order Procedures

The DTC Protocol provides full high-performance support for financial market trading.

There are messages for submitting orders, providing Open Orders and Order Updates, reporting Trade Positions, and reporting Historical Order Fills.

With the DTC Protocol, the general model is that the Server maintains a list of Open Orders, the current Trade Positions, and can optionally maintain Historical Order Fills.

When the Client connects to the Server, it will make a request for the list of [Trading Accounts](#), [Open Orders](#), [Trade Positions](#), and [Historical Order Fills](#). The Client will then be up-to-date in regards to trading once it receives this information.

In DTC there is a single message, [ORDER_UPDATE](#), which is similar to the FIX Execution Report which is used to communicate from the Server to the Client, the details of an order, the status of an order, the most recent action on the order, and Client order actions which results in a rejection.

The reason there is a single unified message is to ensure easier integration on the Client-side, to ensure that the status of an order, and the reason for the sending of the [ORDER_UPDATE](#) is always clear without any confusion. This has proven to be a very good model to follow. It is the model used by FIX. The primary difference between the DTC [ORDER_UPDATE](#) message and the FIX Execution Report, is order Cancel and order Cancel/Replace rejects also use the same [ORDER_UPDATE](#) message to communicate those rejections. The **OrderUpdateReason** field in the [ORDER_UPDATE](#) message indicates if the order update is a rejection and the specific reason for the rejection. There is no room for confusion.

The main identifying fields of the [ORDER_UPDATE](#) message are the **ClientOrderID**, **ServerOrderID**, **OrderUpdateReason**, and the **OrderStatus**.

The other fields of the **ORDER_UPDATE** message need to be set by a Server when sending the [ORDER_UPDATE](#) message, unless they are identified as optional by the DTC Protocol and are not relevant for the particular order.

The only exception to this is when the **OrderUpdateReason** is a ***REJECT**. In this case, the only fields that need to be set in the [ORDER_UPDATE](#) message are **TotalNumMessages**, **MessageNumber**, **Symbol**, **ClientOrderID**, **OrderStatus**, **OrderUpdateReason**.

When an order fills, an **ORDER_UPDATE** message needs to be sent from the Server to the Client, and a [POSITION_UPDATE](#) needs to be sent to the Client updating the Trade Position data for the Symbol and Trade Account.

Bracket Order Procedures

A bracket order consists of a parent order and 2 child orders in an OCO Group. One of the child orders will be a target order. Usually this is a Limit order. The other child order will be a Stop order. The below steps explain how the Client should submit a bracket order in the DTC protocol and how the Server should handle this order.

Once submitted, all three of the orders are independent orders which can be modified and canceled. However, they have an association with each other.

1. The client will submit the Parent order using the [SUBMIT_NEW_SINGLE_ORDER](#) message. In the parent order message it is necessary for the Client to set the **s_SubmitNewSingleOrder::IsParentOrder** field to 1. This signifies to the Server that this is a Parent order of a bracket order.

The Server will use **IsParentOrder** as a flag to know that this message is a parent order. The Server must hold onto this order until it receives the subsequent [SUBMIT_NEW_OCO_ORDER](#) message and then process all of the orders as one complete set. If the next message the Server receives is not a [SUBMIT_NEW_OCO_ORDER](#) message, then it should reject the [SUBMIT_NEW_SINGLE_ORDER](#) message with an [ORDER_UPDATE](#) message.

2. The client will submit the 2 child orders using the [SUBMIT_NEW_OCO_ORDER](#) message.
3. In the [SUBMIT_NEW_OCO_ORDER](#) message, the Client needs to set the **s_SubmitNewOCOOrder::ParentTriggerClientOrderID** field. This field needs to be set to the **ClientOrderID** used in the previous [SUBMIT_NEW_SINGLE_ORDER](#) message which is the parent order for a bracket order. This signifies that the two orders in the OCO order are child orders of the specified parent.

When submitting the parent order, it is necessary for the Client to set **IsParentOrder** on that order. Otherwise, the Server must ignore **ParentTriggerClientOrderID** in this message if it is referencing an order that

did not set **IsParentOrder**.

4. The combination of the single parent order and the OCO order make up a complete bracket order which the Server needs to process.
5. The Server needs to acknowledge the three orders which have been sent by sending a separate [ORDER_UPDATE](#) for each of the three orders. The **OrderUpdateReason** needs to be either **NEW_ORDER_ACCEPTED** or **NEW_ORDER_REJECTED**. If any of the orders are rejected, they all should be rejected by the Server. Further updates to the status of the orders needs to be communicated by the Server to the Client by sending a [ORDER_UPDATE](#) message to the Client individually for each order.
6. The OCO order pair does not become active until the parent is filled or partially filled. Each of the orders in the OCO pair orders need to have an Order Status of **ORDER_STATUS_PENDING_CHILD** set by the Server. When the parent is partially filled, then the quantities of the OCO orders, only increase to the filled quantity of the parent.
7. Each of the three orders can be independently canceled and modified by using the [CANCEL_REPLACE_ORDER](#) or [CANCEL_ORDER](#) messages. When the parent order is canceled, the children need to be canceled as well by Server.
8. It is recommended, although not required by DTC, that when the parent order is modified by the Client before it is filled, that the child orders are also modified by the Server to maintain the same price offsets that they had to the parent order before the parent order was modified.

Integer Trading Messages

The following integer order messages are no longer supported as of August 2020:

- [SUBMIT_NEW_SINGLE_ORDER_INT](#)
- [SUBMIT_NEW_OCO_ORDER_INT](#)
- [CANCEL_REPLACE_ORDER_INT](#)

Automatic Trading Data Updates

Each side assumes that all changes to Open Orders, Trade Positions, and Account Balances are automatically sent from the Server to the Client with the corresponding messages.

So it is implied that these are subscribed to for all accounts associated with the logged in Username.

If the Username supports a very large number of accounts and this is impractical, then this is an area where DTC can be further expanded upon by a service provider.

Indication of Unset Message Fields

In the DTC Protocol, the default value for numeric, date-time, or enumeration fields is zero unless otherwise specified in the field description.

The default for text strings is an empty string unless specified otherwise.

In the case of DTC Binary encoding for messages, where message fields are optional, which are not set by either a Client or Server before sending a message, where that field is a numeric type, and where the default value of zero can be interpreted as a valid value, then DTC requires the field be set to its maximum value. The field description will specify this if it applies to that particular field. Otherwise, it does not.

In the case of a double, this will be **DBL_MAX**.

In the case of an integer, this will be **INT_MAX**.

In the case of an unsigned integer, this will be **UINT_MAX**.

In the case of a 64 bit integer this will be **INT64_MAX**.

This way either the Client or Server can compare the field with these values to determine if the field is actually unset and should be ignored.

In the case of JSON encoding and Google Protocol Buffers encoding, an unset value can be indicated by the field not being present in the JSON object or Google Protocol Buffers object. Therefore, with these protocols, an unset field needs to be indicated by the field not being present in the message object.

With the [Order Entry and Modification Messages](#), Price and Quantity fields are required to be set so they will never use unset values.

However, in the case of the Order Cancel and Replace messages, the Price fields can be optionally set but there are separate fields indicating whether they are set. So the Price fields will never use unset values if they are not set.

Historical Price Data

Historical price data is fully supported with DTC. It is a highly efficient and straightforward protocol.

It optionally supports ZLib compression.

The general sequence for historical price data requests and responses is as follows:

1. Historical data should use a completely separate network connection. The Client and the Server will follow the [Connection and Logon Sequence](#) for this connection.

There is not a strict requirement to use a separate network connection for historical data and implementers of the DTC Protocol can deviate from this. If a separate connection is not used, then when sending the historical price data records in response to a request and they are ZLib compressed, there can be no other message types interleaved with a response until the historical data is fully served for the request. This includes Heartbeat messages.

2. If a separate network connection is used, it is recommended that heartbeat messages not be used. When compression is used, this will corrupt the

transmission of the compressed data. With compression they definitively cannot be used. Therefore, it is best practice to avoid the use of heartbeat messages and close the network socket connection on the Server side, after 30 seconds or so of inactivity.

The Client should not depend upon receiving heartbeat messages and instead should rely upon the receiving of historical data records to indicate the connection is active.

3. The Server can optionally set the **OneHistoricalPriceDataRequestPerConnection** field to 1 in the [LOGON_RESPONSE](#) message to indicate that it only will accept one historical price data request per network connection. After the first request is served or rejected, the network connection will be gracefully closed at the appropriate time by the Server. The Server should close the socket when the Client has read all of the data. This method simplifies the serving of historical price data on the Server side and the implementation on the Client side when data compression is used.
4. The Client will send to the Server a [HISTORICAL_PRICE_DATA_REQUEST](#) message with the details of the historical data to be returned by the Server.
5. If the Server rejects the historical data request, it will send a [HISTORICAL_PRICE_DATA_REJECT](#) message. This is never ZLib compressed. At this point, the communication is complete for the particular historical data request.
6. If the Server accepts the historical data request, it will send a [HISTORICAL_PRICE_DATA_RESPONSE_HEADER](#) message. This is never ZLib compressed.
7. In the case of a request other than for Tick data, the Server will respond with a series of [HISTORICAL_PRICE_DATA_RECORD_RESPONSE](#) messages until the historical data request is completely fulfilled.

These can be compressed with ZLib compression if the Client requested it and the Server supports this compression.

DTC strictly defines the ordering of records returned, from oldest to newest.

When the Server writes these records into the network socket, it should write them in batches for efficiency. The batch size will be determined by the Server.

8. In the case of a request for Tick data, the Server will respond with a series of [HISTORICAL_PRICE_DATA_TICK_RECORD_RESPONSE](#) messages until the historical data request is completely fulfilled. If the Server does not support Tick data or does not have Tick data for the specified time period it can respond with [HISTORICAL_PRICE_DATA_RECORD_RESPONSE](#) messages.

In either case these messages can be compressed with ZLib if the Client requested it and the Server supports this compression.

DTC strictly defines the ordering of records returned, from oldest to newest.

When the Server writes these records into the network socket, it should write them in batches for efficiency. The batch size will be determined by the Server.

9. The final historical data record message ([HISTORICAL_PRICE_DATA_RECORD_RESPONSE](#), [HISTORICAL_PRICE_DATA_TICK_RECORD_RESPONSE](#)), needs to set the **IsFinalRecord** structure field to 1 to flag that it is the final record.

This record does not need to contain any price data. If the record is an empty record other than having the flag set, then the **t_DateTime** structure field needs to be set at the default of 0 to indicate to the Client that the record contains no valid price data.

10. The Server must hold the connection open so long as the Server has data to send in the network socket buffer. Furthermore, the socket should not be closed until either there is several minutes of inactivity or the Client-side has closed the network socket.
11. When the final historical data record has been received, the historical data request has been rejected, or a **HISTORICAL_PRICE_DATA_RESPONSE_TRAILER** message is received, then the Client can close the connection if no further historical data requests are to be made, or the server only supports one historical data request per connection ([LOGON_RESPONSE](#)::OneHistoricalPriceDataRequestPerConnection = 1).
12. If for whatever reason, the connection is lost, the Client will make a fresh connection for historical price data when needed. So there is no need for the connection to be persistent.
13. Since historical price data is not the type of data the Client will be making frequent requests for, the Server can close the socket connection after a certain period of inactivity determined by the Server (example: 30 seconds). If the Server has set the field **OneHistoricalPriceDataRequestPerConnection** to 1 in the [LOGON_RESPONSE](#) message, it will only respond to one request from the Client and gracefully closed a network socket when done. The Client should not reestablish this connection until it needs historical price data again.
14. For further details, refer to the relevant header file which includes all of the associated data structures and member fields of those structures.

Symbol ID and Request ID Rules

RequestID message field values need to be unique per request, message type and per connection session. A connection session is defined as the time from a successful logon up until the disconnection.

RequestID values can conflict between message types and connection sessions. **RequestID** message field values can also conflict with **SymbolID** field values.

SymbolID message field values for market data request messages need to be unique per Symbol and Exchange combination only. The **SymbolID** value cannot change when market data is subscribed to. When requesting snapshot data or after unsubscribing and re-subscribing, the

SymbolID value can change for a Symbol and Exchange combination.

SymbolIDs can conflict with RequestIDs used in other message types.

Nonstandard Messages

The DTC protocol provides for nonstandard, custom or private messages that can be shared between Client and Server developers.

The Client or the Server will need to develop their own messages and fields and agree upon them. The **Type** field of the message needs to have a number in the range of 10000 and higher. This is the only requirement.

Even if different Clients and Servers who do not know each other and do not communicate with each other, use **Type** numbers which are conflicting, this will not cause a problem because these messages are considered private.

DTC Messages

Below are the links to each message which includes the documentation for the message and the fields within the message.

The below documentation is not meant to be specific to any particular encoding, whether binary, JSON, or Google protocol buffers.

DTC Message Field Type Descriptions

[int32] 32-bit Integer

The int32 type is a 32-bit integer.

[unsigned int16] Unsigned 16-bit Integer

This is a 16-bit unsigned integer value.

[unsigned int32] Unsigned 32-bit Integer

This is a 32 bit unsigned integer value.

[EncodingEnum] Encoding Enumeration

This enumeration indicates the encoding method. It can be one of the following values.

- **BINARY_ENCODING** = 0
- **BINARY_WITH_VARIABLE_LENGTH_STRINGS** = 1
- **JSON_ENCODING** = 2
- **JSON_COMPACT_ENCODING** = 3
- **PROTOCOL_BUFFERS** = 4

[char] Character

This is a character string. The final byte is always a null terminator for binary encoding with and without a variable length strings.

[unsigned int8] Byte

This is a single byte. Usually it represents a TRUE or FALSE state. Where a numeric value of 1 is TRUE and 0 is FALSE. 1 and 0 are not character values. They are integer values.

[double] 64-bit Floating Point Value

This is a 64-bit floating-point value.

[float] 32 bit Floating Point Value

This is a 32-bit floating-point value.

[AtBidOrAskEnum] At Bid Or Ask Enumeration

- **BID_ASK_UNSET** = 0
- **AT_BID** = 1: Use AT_BID if the aggressor of the trade was the seller.
- **AT_ASK** = 2: Use AT_ASK if the aggressor of the trade was the buyer.

[LogonStatusEnum] Logon Status Enumeration

- **LOGON_SUCCESS** = 1
- **LOGON_ERROR** = 2
- **LOGON_ERROR_NO_RECONNECT** = 3
- **LOGON_RECONNECT_NEW_ADDRESS** = 4

[t_DateTime] Date Time

This is a 64-bit integer UNIX time value.

This is the number of seconds since the UNIX epoch (January 1, 1970, 00:00:00 UTC).

With the DTC Protocol, the time zone is always UTC for Date-Time values.

In the case of Google Protocol Buffer encoding the equivalent data type used is **sfixed64**.

[t_DateTimeWithMilliseconds] Date Time With Milliseconds

This is a 64-bit floating-point UNIX time value.

The integer portion is the number of seconds since the UNIX epoch (January 1, 1970, 00:00:00 UTC).

With the DTC Protocol, the time zone is always UTC for Date-Time values.

The portion of this value to the right of the decimal point is the optional number of milliseconds. Where one millisecond equals .001.

In the case of Google Protocol Buffer encoding, the equivalent data type used is **double**.

[t_DateTimeWithMicrosecondsInt] Date Time With Microseconds

This is a 64-bit integer UNIX time value.

The integer portion is the number of microseconds since the UNIX epoch (January 1, 1970, 00:00:00 UTC).

With the DTC Protocol, the time zone is always UTC for Date-Time values.

[t_DateTime4Byte] 4 Byte UNIX Date-Time

This is a 32 bit integer UNIX time value.

This is the number of seconds since the UNIX epoch (January 1, 1970, 00:00:00 UTC).

With the DTC Protocol, the time zone is always UTC for Date-Time values.

In the case of Google Protocol Buffer encoding, the equivalent data type used is **sfixed32**.

[MarketDataFeedStatusEnum] Market Data Feed Status Enumeration

This indicates if the market data feed is available in its entirety or for an individual symbol. It can be one of the following values.

- **MARKET_DATA_FEED_STATUS_UNSET = 0**
- **MARKET_DATA_FEED_UNAVAILABLE = 1**
- **MARKET_DATA_FEED_AVAILABLE = 2**

[BuySellEnum] Buy Sell Enumeration

This indicates buy or sell. It can be one of the following values.

- **BUY_SELL_UNSET = 0**
- **BUY = 1**
- **SELL = 2**

[RequestActionEnum] Request Action Enumeration

This indicates the particular request action for market data and market depth requests. It can be one of the following values.

- **SUBSCRIBE = 1**
- **UNSUBSCRIBE = 2**
- **SNAPSHOT = 3**

[MarketDepthUpdateTypeEnum] Market Depth Update Type Enumeration

This indicates the particular market depth update type. It can be one of the following values.

- **DEPTH_UNSET** = 0
- **MARKET_DEPTH_INSERT_UPDATE_LEVEL** = 1
- **MARKET_DEPTH_DELETE_LEVEL** = 2

[PriceDisplayFormatEnum] Price Display Format Enumeration

This indicates the price display format for market data prices. It can be one of the following values.

- **PRICE_DISPLAY_FORMAT_UNSET** = -1
- **PRICE_DISPLAY_FORMAT_DECIMAL_0** = 0
- **PRICE_DISPLAY_FORMAT_DECIMAL_1** = 1
- **PRICE_DISPLAY_FORMAT_DECIMAL_2** = 2
- **PRICE_DISPLAY_FORMAT_DECIMAL_3** = 3
- **PRICE_DISPLAY_FORMAT_DECIMAL_4** = 4
- **PRICE_DISPLAY_FORMAT_DECIMAL_5** = 5
- **PRICE_DISPLAY_FORMAT_DECIMAL_6** = 6
- **PRICE_DISPLAY_FORMAT_DECIMAL_7** = 7
- **PRICE_DISPLAY_FORMAT_DECIMAL_8** = 8
- **PRICE_DISPLAY_FORMAT_DECIMAL_9** = 9
- **PRICE_DISPLAY_FORMAT_DENOMINATOR_256** = 356
- **PRICE_DISPLAY_FORMAT_DENOMINATOR_128** = 228
- **PRICE_DISPLAY_FORMAT_DENOMINATOR_64** = 164
- **PRICE_DISPLAY_FORMAT_DENOMINATOR_32_QUARTERS** = 136
- **RICE_DISPLAY_FORMAT_DENOMINATOR_32_HALVES** = 134
- **PRICE_DISPLAY_FORMAT_DENOMINATOR_32** = 132
- **PRICE_DISPLAY_FORMAT_DENOMINATOR_16** = 116
- **PRICE_DISPLAY_FORMAT_DENOMINATOR_8** = 108
- **PRICE_DISPLAY_FORMAT_DENOMINATOR_4** = 104
- **PRICE_DISPLAY_FORMAT_DENOMINATOR_2** = 102

[PutCallEnum] Put Call Enumeration

This indicates if the option is a put or call. It can be one of the following values.

- **PC_UNSET** = 0
- **PC_CALL** = 1
- **PC_PUT** = 2

[SecurityTypeEnum] Security Type Enumeration

This indicates the Security Type. It can be one of the following values.

- **SECURITY_TYPE_UNSET** = 0
- **SECURITY_TYPE_FUTURES** = 1
- **SECURITY_TYPE_STOCK** = 2
- **SECURITY_TYPE_FOREX** = 3 (Also applies to Bitcoins)
- **SECURITY_TYPE_INDEX** = 4
- **SECURITY_TYPE_FUTURES_STRATEGY** = 5
- **SECURITY_TYPE_FUTURES_OPTION** = 7
- **SECURITY_TYPE_STOCK_OPTION** = 6
- **SECURITY_TYPE_INDEX_OPTION** = 8
- **SECURITY_TYPE_BOND** = 9
- **SECURITY_TYPE_MUTUAL_FUND** = 10

[SearchTypeEnum] Search Type Enumeration

This indicates the Search Type. It can be one of the following values.

- **SEARCH_TYPE_UNSET** = 0
- **SEARCH_TYPE_BY_SYMBOL** = 1
- **SEARCH_TYPE_BY_DESCRIPTION** = 2

[OrderTypeEnum] Order Type Enumeration

This indicates the order type. It can be one of the following values.

- **ORDER_TYPE_UNSET** = 0
- **ORDER_TYPE_MARKET** = 1
- **ORDER_TYPE_LIMIT** = 2
- **ORDER_TYPE_STOP** = 3
- **ORDER_TYPE_STOP_LIMIT** = 4
- **ORDER_TYPE_MARKET_IF_TOUCHED** = 5

[TimeInForceEnum] Time In Force Enumeration

This enumeration indicates the Time in Force for orders. It can be one of the following values.

- **TIF_UNSET** = 0
- **TIF_DAY** = 1
- **TIF_GOOD_TILL_CANCELED** = 2
- **TIF_GOOD_TILL_DATE_TIME** = 3
- **TIF_IMMEDIATE_OR_CANCEL** = 4
- **TIF_ALL_OR_NONE** = 5
- **TIF_FILL_OR_KILL** = 6

The Client needs to understand that not all of these will be supported for the particular Symbol being traded or by the Server. In which case, the order can be rejected.

[OrderStatusEnum] Order Status Enumeration

This enumeration indicates the Order Status. It can be one of the following values.

- **ORDER_STATUS_UNSPECIFIED** = 0
- **ORDER_STATUS_ORDER_SENT** = 1
- **ORDER_STATUS_PENDING_OPEN** = 2
- **ORDER_STATUS_PENDING_CHILD** = 3
- **ORDER_STATUS_OPEN** = 4
- **ORDER_STATUS_PENDING_CANCEL_REPLACE** = 5
- **ORDER_STATUS_PENDING_CANCEL** = 6
- **ORDER_STATUS_FILLED** = 7
- **ORDER_STATUS_CANCELED** = 8
- **ORDER_STATUS_REJECTED** = 9
- **ORDER_STATUS_PARTIALLY_FILLED** = 10

[HistoricalDataIntervalEnum] Historical Data Interval Enumeration

This indicates the time interval for historical price data records. It can be one of the following values.

- **INTERVAL_TICK** = 0
- **INTERVAL_1_SECOND** = 1
- **INTERVAL_2_SECONDS** = 2
- **INTERVAL_4_SECONDS** = 4
- **INTERVAL_5_SECONDS** = 5
- **INTERVAL_10_SECONDS** = 10
- **INTERVAL_30_SECONDS** = 30
- **INTERVAL_1_MINUTE** = 60
- **INTERVAL_1_DAY** = 86400
- **INTERVAL_1_WEEK** = 604800

[OrderUpdateReasonEnum] Order Update Reason Enumeration

This indicates the reason for sending an Order Update message. It can be one of the following values.

- **ORDER_UPDATE_REASON_UNSET** = 0
- **OPEN_ORDERS_REQUEST_RESPONSE** = 1
- **NEW_ORDER_ACCEPTED** = 2
- **GENERAL_ORDER_UPDATE** = 3
- **ORDER_FILLED** = 4
- **ORDER_FILLED_PARTIALLY** = 5
- **ORDER_CANCELED** = 6
- **ORDER_CANCEL_REPLACE_COMPLETE** = 7
- **NEW_ORDER_REJECTED** = 8
- **ORDER_CANCEL_REJECTED** = 9
- **ORDER_CANCEL_REPLACE_REJECTED** = 10

[OpenCloseTradeEnum] Open Close Trade Enumeration

For orders this field specifies whether the order opens a new Position or increases an existing Position, or closes an existing Position or decreases an existing Position.

For order fills this field specifies whether the fill opened a new Position or increased an existing Position, or closed an existing Position or decreased an existing Position.

The use of this field for new orders depends upon the particular market/security as specified by the Symbol and Exchange fields and whether the Server requires it. Clients should always try to set this field for new orders, however the Server may not use it. This field is not used for futures.

It can be one of the following values.

- **TRADE_UNSET** = 0
- **TRADE_OPEN** = 1
- **TRADE_CLOSE** = 2

[HistoricalPriceDataRejectReasonCodeEnum] Historical Price Data Reject Reason Code Enumeration

The following enumerations are for the [Historical Price Data Reject](#) message.

- **HPDR_UNSET** = 0. The historical price data reject code is unset.
- **HPDR_UNABLE_TO_SERVE_DATA_RETRY_IN_SPECIFIED_SECONDS** = 1.
The server is unable to serve the historical data request and the request should be retried in the specified number of seconds. A properly implemented high-performance server should never utilize this reject code.
- **HPDR_UNABLE_TO_SERVE_DATA_DO_NOT_RETRY** = 2. The server is unable to serve the historical data request and there should be no retry.
- **HPDR_DATA_REQUEST_OUTSIDE_BOUNDS_OF_AVAILABLE_DATA** = 3.
The Date-Time range of historical data requested is outside the bounds of the available data.
- **HPDR_GENERAL_REJECT_ERROR** = 4. There is another undocumented reason the server cannot accept the historical data request and it has been rejected.

[PartialFillHandlingEnum] Partial Fill Handling Enumeration

This enumeration is for the OCO order messages.

- **PARTIAL_FILL_UNSET** = 0. Indicates no special partial fill handling.
- **PARTIAL_FILL_HANDLING_REDUCE_QUANTITY** = 1. This specifies that when there is a partial fill of one of the orders in the OCO order set, that the quantity of the other order needs to be reduced by the quantity of the order fill.
- **PARTIAL_FILL_HANDLING_IMMEDIATE_CANCEL** = 2. This specifies that when there is a partial fill of one of the orders of the OCO order set, that the other order needs to be immediately canceled.

[FinalUpdateInBatchEnum] Final Update In Batch Enumeration

This enumeration is for market depth updates.

- **FINAL_UPDATE_UNSET** = 0. Indicates the value is unset.
- **FINAL_UPDATE_TRUE** = 1. Indicates the market depth update message is the final message in the batch.
- **FINAL_UPDATE_FALSE** = 2. Indicates the market depth update message is not the final update in the batch.
- **FINAL_UPDATE_BEGIN_BATCH** = 3. Indicates the market depth update message is the first update in the batch.

*Last modified Tuesday, 22nd November, 2022.