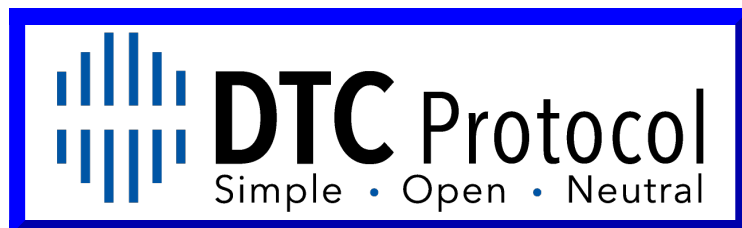


Data and Trading Communications Protocol

DTC Protocol

- [Introduction to DTC Protocol](#)
- [Why DTC?](#)
- [Why not FIX?](#)
- [Is Standardization Possible?](#)
- [Why Should We As a Client or Server Adopt DTC?](#)
- [Name and Logo](#)
- [Working Group](#)
- [Adaptable](#)
- [Tight Protocol With Support for New Message Types](#)
- [Message Format and Encoding](#)
 - [Binary Encoding](#)
 - [Binary With Variable Length Strings Encoding](#)
 - [JSON Encoding](#)
 - [Compact JSON Encoding](#)
 - [Google Protocol Buffers \(GPB\)](#)
- [Underlying Transport Layer and Security](#)
- [Server Can Be Local or Remote](#)
- [DTC Projects](#)
- [Open Specification](#)
- [DTC Protocol Test Client](#) (Opens a new page)
- [DTC Messages and Procedures](#) (Opens a new page)
- [DTC Files](#)
 - [Standard Binary Encoding with Fixed Length Strings C++ Header File](#) Updated on 18th May, 2023
 - [Standard Binary Encoding with Fixed Length Strings C++ File](#) Updated on 05th April, 2023
 - [Binary Encoding with Variable Length Strings C++ Header File](#) Updated on 13th April, 2023
 - [Binary Encoding with Variable Length Strings C++ File](#) Updated on 13th April, 2023
 - [Google Protocol Buffer Encoding](#) Updated on 23rd May, 2023



Introduction to DTC Protocol

The Data and Trading Communications Protocol (abbreviated **DTC Protocol**) is an open communications protocol for the financial market trading community.

The DTC Protocol is an initiative to establish an open communications protocol for financial market data, trading, historical market data and historical price data.

The DTC Protocol defines a common set of messages and fields within those messages, and has clear procedures for working with these messages and fields in order to create reliable and plug-and-play interoperability between Clients and Servers working with financial market data and trading functions.

The DTC Protocol is based upon the FIX protocol as much as possible. Trading messages and fields are very closely related to the FIX protocol.

Market data in the DTC Protocol follows a model used by well designed and efficient data feeds.

The DTC Protocol allows any manual trading, automated/algorithmic trading, charting, market analysis, or a similar type of program (the **Client**), and market Data and Trading services (the **Server**), to interact with each other reliably and efficiently.

This protocol is for any type of user whether it is an individual writing a program for their own use, a commercial provider of trading and charting software, an institutional high-frequency trader, or a Data or Trading services provider.

The definition of a "Data" service is a service that provides financial market data. The definition of a "Trading" service is a service that may provide financial market data, provides connectivity to exchanges or operates their own exchange, and they may also handle customer transactions.

This is a protocol designed to work directly on a network communication link and can be used over both SSL/TLS and non-SSL/TLS network connections.

The DTC protocol is a simple messaging protocol. In the DTC Protocol, the Client and Server, exchange Messages between each other with well-defined fields and procedures.

The term Message, refers to a message instance sent over a secure or nonsecure network communications link which typically is a TCP/IP connection over the Internet.

There are different methods of [encoding](#) which are supported by the DTC Protocol.

Complete documentation for the Messages and Message fields of the DTC Protocol can be found on the [DTC Messages and Procedures](#) page.

Why DTC?

It is common sense that it is better for Data and Trading service providers, and the users who program to those services, to follow a standard communications protocol.

Rather than a Client writing to the protocol or API used by the Server, which is generally the case, or the Server writing to the protocol or API used by the Client, less often the case, there is now a single, neutral and open specification protocol that each side will write to. Therefore, neither the Server or the Client is performing integration work which is specific to a particular Client or Server.

The proper working model is that neither side is integrating to the others service/product and each side uses a common standardized protocol which they focus on and ensure 100% reliability and compatibility with.

DTC is completely neutral. What this means is that it does not favor the Client or the Server.

Think if a web browser had to be designed to work with a different protocol from every single website out there. Would that make sense? The obvious answer is that this does not make sense and this is why establishing a common open specification protocol is needed. The DTC protocol is the solution.

Standards are very common in [electronic data communications](#) and physical electronic networks. The Internet would not even be possible without standards. There is the TCP/IP standard. Hardware layers are highly standardized. There is a standard for Ethernet. There is no way that physical networks can interoperate without standardization.

Where would the Internet be without communication standards? The explosive growth seen on the Internet which began in the 1990s, throughout the first decade of the 21st century, and continues to this day, would not have taken place without standards.

When front-end client trading or charting program supports many external services, working with various protocols or APIs, becomes difficult and problematic especially if in-process API components are used.

It is the view that in process API components are very problematic and whether they are open source or black box, they force the developer using them to follow a certain input-output model which may not match the Client or Server which is using them.

An application program, whether it supports manual trading, algorithmic trading, charting, market analysis, or whatever, if it supports the DTC Protocol, it will be automatically compatible with any Server that also supports DTC.

It is far more productive for both developers of Client and Server systems to be focusing on maintaining and integrating to a single open specification protocol rather than numerous protocol/APIs which exist in this industry.

In the case of the Client, there is much more time available for the development of the clients software rather than integrating to Servers and dealing with the various associated issues.

Both the Client and the Server will work hard at following the protocol strictly, so there is automatic compatibility. This is a requirement of the DTC Protocol.

Why not FIX?

FIX is regarded as a high overhead protocol for market data. FAST is a solution to this, but it technically is complex to implement.

For some users FIX is not easily implemented and used. Data and Trading service providers know that many of their customers will have difficulty using FIX.

Over an authenticated TCP/IP connection, FIX has unnecessary fields like the Sender Comp ID, Target Comp ID and checksum.

FIX is not a strict standard. Although a Client and the Server may implement FIX, they are not automatically compatible with each other unless they follow a specific FIX implementation they agree upon.

FIX does not have messages for historical price data. And also FIX is not efficient with historical data transport. Whereas DTC, has full support for historical price data in the most efficient way possible.

The DTC Protocol is very efficient and requires minimal network bandwidth when using binary encoding.

Is Standardization Possible?

Is it possible to standardize financial market data and trading?

To answer this question, requires us to look at what exactly is being communicated between Clients and Servers.

For market data, this is the communication from a Server to a Client in response to requests from the Client, of pricing, index or indicator data. This type of data consists of snapshot data representing Settlement, Open, High and Low values and various other fields during the trading session, and real-time updates to those fields. All of this is very easily standardized. Historical activity of this pricing data is also easily standardized.

For trading, a Client will need to submit an order to buy or sell a particular security to the Server and receive updates on the order status, and the positions for the account or multiple accounts. Order entry and positions also have common fields in financial market trading. Therefore, standardization is possible.

The very fact that the FIX protocol exists, also proves that standardization is possible. Although the FIX protocol is not a highly standardized plug-and-play protocol. The DTC Protocol is.

Why Should We As a Client or Server Adopt DTC?

Whether you develop a Client-side market analysis and/or trading program, or a Server-side Data or Trading service, adopting the DTC Protocol greatly increases your connectivity choices with other Clients and Servers adopting the DTC Protocol.

In the long run, it reduces development expenses related to integration with other Clients or Servers because a common standardized protocol is being used.

Users of the DTC Protocol follow a common protocol which facilitates nearly immediate plug-and-play compatibility between each other.

Connections between Clients and Servers is greatly simplified and easier to maintain because a standard protocol is being followed.

Whether you are the Client or the Server, or both, it is the spirit of the DTC Protocol that each side will strive to follow this protocol so there is complete compatibility and reliability.

It is not the purpose of the DTC protocol to replace any proprietary protocols or integration methods that a Client or Server already implements or would like to implement.

What we currently have in this industry is disorganized with so many different types of connectivity methods and protocols. This is really compounded when these proprietary in-process APIs are used.

It makes most sense for all of us to focus on developing our own products and services, rather than having someone who is interested in our product or service, to then have to go through the time, effort and expense to integrate to it using a proprietary method.

If the proprietary integration method involves an API component, then these have proven to be confusing and have ongoing difficulties and force the other side to follow in input-output model created by someone outside of their development team. In practice, this kind of model is highly impractical and problematic. The industry should shun the use of proprietary API components.

The DTC Protocol is a plug-and-play protocol. So Clients and Servers that implement this protocol can immediately begin to interoperate with each other with no effort. There will be some specialized cases, where this may not quite be the case. However, in general this is the objective of the DTC Protocol.

For services that provide Trading services, by adopting the DTC Protocol you can ensure that the handling of orders on the Client-side will be done properly.

The DTC Protocol uses the [ORDER_UPDATE](#) message to update a trading order and provide the reason for the update and the current status of the order through the **OrderUpdateReason** (equivalent to the FIX **ExecutionType** field) and the **OrderStatus** message fields.

The [ORDER_UPDATE](#) message is a single unified message for the Server to provide any kind of feedback relating to an order for trading. By using this message there is no room for a Client to misinterpret an orders state and to mishandle an order.

Whereas, with proprietary APIs and protocols, it generally takes a long time through API documentation reading, testing, and user feedback that trading order handling has been properly implemented for a Trading service. This is not a desirable outcome because it provides a negative user experience. The DTC Protocol solves this problem.

A complete implementation of the [DTC Protocol headers](#), not applicable to JSON encoding, are defined and can be freely used.

Complete and formal [documentation](#) is available for this protocol. You are welcome to take a copy of this documentation. There are no copyrights here.

The only way in which easy interoperability between Clients and Servers can be achieved is by embracing and supporting a common protocol. We will not solve the current problem that exists in the industry by doing nothing. We all must move forward with this.

You need to ask yourself, can you survive long-term without following well-designed standards? Are you going to be isolated by following your own proprietary protocols?

If you are a Data or Trading services provider, you may convince yourself that your existing users who have integrated to your protocols and APIs are satisfied with what you offer. How do you know they will

not be even more satisfied with the DTC Protocol? Especially being this is a very basic communication protocol which can serve as the foundation for higher level interfacing methods which use the DTC Protocol at their core.

Name and Logo

The name of this protocol is the DTC Protocol. This stands for Data and Trading Communications Protocol.

The logo for it is below.



[Other Logo Images](#)

The DTC logo images have been donated into the community for free use. The only conditions are that they are used in association with the Data and Trading Communications Protocol and that you use them only if you implement the DTC protocol either on the Client and/or the Server side.

Working Group

To be part of the DTC Protocol working group, contact us at **[Java Script Is Required. To View The Email, enable Java Script]**.

The working group responsibilities are to implement and maintain the DTC Protocol according to the intentions described on this page.

Adaptable

This protocol is adaptable. New messages can be added and fields can be added to existing messages.

The [LOGON_RESPONSE](#) message has flags indicating what messages the Server supports.

The Client will support the messages that it needs to based upon the requirements of the Client.

To add message types, refer to [Tight Protocol With Support for New Message Types](#).

For information about how new fields are handled in existing messages, refer to [Versioning](#).

Tight Protocol With Support for New Message Types

The FIX protocol has variations in the tags used in specific FIX messages, some variations with the specific meaning of the different message tags, no strict standard for Order ID handling, and uses

custom tags.

With the DTC Protocol, it is designed to be a tightly defined protocol with a clear set of messages and fields which have specific meanings. The protocol is also expandable.

Anyone using this protocol, should be able to connect to the other side, and expect everything to work properly. This is the purpose of the protocol.

To request new message types and add fields to an existing message, contact **[Java Script Is Required. To View The Email, enable Java Script]** and request the particular messages and/or fields to be added. The major working group members will review and evaluate the request and discuss it with you.

In the case of binary encoding, new fields must always be added onto the end of a structure. Existing structure fields must not change in order to maintain protocol safety.

New message types which are considered very proprietary need to have a Type number that is in the 10000 range.

The DTC Protocol recognizes that some messages will be unsupported by a Server. If a Server does not support a particular message category, indicate so through the corresponding flags in the [LOGON_RESPONSE](#) message.

Message Format and Encoding

In the current implementation of DTC, the following encodings are defined.

In the case of when separate network connections are used for market data and trading, the market data connection could use Binary Encoding, and the trading connection could use JSON Encoding. Therefore, there is complete flexibility to use multiple encoding methods and the most appropriate encoding method depending upon whether the connection is used for market data or for trading.

The DTC Protocol supports specifying the encoding to be used over the network connection. Refer to the [Encoding Request Sequence](#) for the procedure for specifying the encoding.

Binary Encoding

Binary Encoding is the most basic method of encoding. In binary encoding, messages are encoded using fixed size binary data structures with embedded fixed length strings. This allows for very easy implementation in code, compactness of messages (in the case of market data messages) and very fast performance.

All of data structures for binary encoding are defined in the [Standard Binary Encoding with Fixed Length Strings C++ Header File](#).

With binary encoding, the beginning of each structure begins with a 2 byte size and a 2 byte type indicating the message type and the total size of the message including the size and type fields.

New messages can be defined when needed and additional fields can be added to the end of existing message structures. The data held in the message structures is directly usable in

program code without any translation or parsing.

Binary With Variable Length Strings Encoding

The Binary With Variable Length Strings Encoding is very similar to the standard Binary Encoding except the fixed length strings from the binary encoded DTC messages are replaced with variable length strings. This enables shorter messages to be sent, and also allows strings that are larger than the defined fixed length strings to be sent.

There is a separate header file, [DTCProtocolVLS.h](#), that redefines messages that contain strings.

Any DTC messages that do not contain strings continue to use the exact same messages as Binary Encoding, so this header should be used along with the main [DTC Binary Encoding header](#) file.

DTC messages that contain strings are re-defined by replacing the fixed length string field in the DTC message with a 16-bit Offset and a 16-bit Length field in the data structure. These two fields are contained in the `s_VariableLengthStringField` structure.

Variable Length String Structure

```
struct s_VariableLengthStringField
{
    uint16_t Offset;
    uint16_t Length;
};
```

The actual variable length strings are appended to the end of the fixed size portion of the DTC message in the stream, and the message **Size** field includes these bytes. The Offset field specifies the offset from the base on the DTC message where the variable length string starts. The Offset is relative to an index of 0 at the base. The Length field specifies how long the string is, including the null terminator. Variable length strings should always include a null terminating char (zero). If the string is not present, an Offset and Length of zero must be specified which is the default.

In the [Binary Encoding with Variable Length Strings C++ Header File](#), there are functions that show how a string is added to a message and parsed from a message:

AddVariableLengthStringField() and **GetVariableLengthStringField()**. When sending a message with variable length strings, it is imperative that the strings are sent in the same order that they were added to the base structure.

Each DTC message that supports variable length strings has **Add*(uint16_t StringLength)** and **Get*()** functions for adding a string and getting a string respectively in the DTC message.

Similar to the binary encoding, the messages for the Binary with Variable Length Strings Encoding each begin with a 2 byte message size and a 2 byte type indicating the message type.

JSON Encoding

The JSON Encoding sends each DTC message as a JSON object with **name:value** pairs. Inherently JSON supports variable length strings and allows the very easy adding and removing of fields. Therefore, it is a flexible encoding method for the DTC Protocol.

Messages using the JSON encoding, do not contain a size field like the binary encoding, and so each DTC JSON message object is separated with a null (8 bit numeric 0) terminator in the network data stream.

Each JSON message needs to be encoded using 8-bit characters. Unicode is not supported.

The name:value pair names match the DTC message field names, and the value portion matches the type of the DTC field. The JSON field **name** is case sensitive.

The values are either a number, a string, an array, or a "null". A null value indicates that the field is not set. String values are only present for text string fields. An array is only present for several DTC messages that contain depth arrays.

An unset field can be left out of JSON object.

The following is a JSON Encoding example of a s_MarketDataUpdateTrade message:

```
{"Type":107, "SymbolID":1, "AtBidOrAsk":0, "Price":2058.75, "Volume":12, "DateTime":1456736458}
```

The **Type** field should be first. Otherwise, the general field order does not matter.

To determine the number for the **Type** field for a particular DTC message, look up the message type constant in the [DTCProtocol.h](#) file and use that message number. Example:

LOGON_REQUEST = 1.

Compact JSON Encoding

The Compact JSON Encoding sends high frequency DTC messages in a more compact form with no field names, while sending the remaining messages in the standard JSON Encoding.

This encoding maintains many of the strengths of the standard JSON encoding while vastly improving the efficiency of high frequency messages.

Messages using the compact JSON encoding, do not contain a size field like the binary encoding, and so each DTC JSON message object is separated with a null terminator in the network data stream. The Type name:value pair is maintained to allow compact JSON messages to be initially parsed in a similar manner to the standard JSON encoding.

All of the remaining type:value pairs are replaced with a single JSON pair named "**F**", and the value contains all of the DTC message fields represented as an array of values in the same order as they exist in the standard [DTC Binary Encoding](#) message. Each value matches the type of the corresponding DTC Protocol field.

The values are either a number, a string, an array, or a "null". A null value indicates that the field is not set. String values are only present for string fields. An array is only present for several DTC

messages that contain market depth data arrays.

The following is a Compact JSON Encoding example of a s_MarketDataUpdateTrade message:

```
{"Type":107, "F":[1, 0, 2058.75, 12, 1456736458]}
```

The **Type** field must be first. For this encoding, the field order must match the order of the corresponding DTC Protocol Binary Encoding message.

The following types use the compact JSON format:

- **HEARTBEAT**
- **MARKET_DATA_SNAPSHOT**
- **MARKET_DATA_SNAPSHOT_INT**
- **MARKET_DATA_UPDATE_TRADE**
- **MARKET_DATA_UPDATE_TRADE_COMPACT**
- **MARKET_DATA_UPDATE_TRADE_INT**
- **MARKET_DATA_UPDATE_LAST_TRADE_SNAPSHOT**
- **MARKET_DATA_UPDATE_BID_ASK**
- **MARKET_DATA_UPDATE_BID_ASK_COMPACT**
- **MARKET_DATA_UPDATE_BID_ASK_INT**
- **MARKET_DATA_UPDATE_SESSION_OPEN**
- **MARKET_DATA_UPDATE_SESSION_OPEN_INT**
- **MARKET_DATA_UPDATE_SESSION_HIGH**
- **MARKET_DATA_UPDATE_SESSION_HIGH_INT**
- **MARKET_DATA_UPDATE_SESSION_LOW**
- **MARKET_DATA_UPDATE_SESSION_LOW_INT**
- **MARKET_DATA_UPDATE_SESSION_VOLUME**
- **MARKET_DATA_UPDATE_OPEN_INTEREST**
- **MARKET_DATA_UPDATE_SESSION_SETTLEMENT**
- **MARKET_DATA_UPDATE_SESSION_SETTLEMENT_INT**
- **MARKET_DEPTH_SNAPSHOT_LEVEL**
- **MARKET_DEPTH_SNAPSHOT_LEVEL_INT**
- **MARKET_DEPTH_UPDATE_LEVEL**
- **MARKET_DEPTH_UPDATE_LEVEL_COMPACT**
- **MARKET_DEPTH_UPDATE_LEVEL_INT**
- **MARKET_DEPTH_FULL_UPDATE_10**
- **MARKET_DEPTH_FULL_UPDATE_20**
- **HISTORICAL_PRICE_DATA_RECORD_RESPONSE**
- **HISTORICAL_PRICE_DATA_TICK_RECORD_RESPONSE**
- **HISTORICAL_PRICE_DATA_RECORD_RESPONSE_INT**
- **HISTORICAL_PRICE_DATA_TICK_RECORD_RESPONSE_INT**

Google Protocol Buffers (GPB)

The DTC Protocol supports Google Protocol Buffer encoding. All DTC messages and constants

are defined in the [Google Protocol Buffer Encoding](#) file. This file requires version 3 of the Protocol Buffer compiler and libraries.

The proto file is then used with the protocol buffer compiler (protoc) to generate a language specific API like for C++ or whatever language. The generated API is then used to build, encode, decode, and extract messages and message fields.

DTC messages are built using the generated API message object/methods. All DTC messages are then serialized into encoded blocks of data via the `Serialize*()` methods in the generated API.

These encoded messages are then sent over the network. Since the protocol buffer wire format is not self-delimiting, a small header is used to delimit and describe the encoded messages. The header is a 4-byte header containing a 2-byte unsigned integer size and a 2-byte unsigned integer message type. Each field is sent in little endian format. The data structure is defined below.

```
struct s_DTCMessageHeader
{
    uint16_t Size;
    uint16_t Type;
};
```

The header **Size** field is set to the length of the encoded message plus the size of the header. The **Size** field allows the receiver to know when an entire message has been received. The **Type** field contains the DTC Message type, which are all defined in the **DTCMessageType** enum in the proto file.

The **Type** field is then used by the receiver to parse the encoded message into the associated object with the Google Protocol Buffers API `Parse()` methods. The generated API is then used to access the message fields.

For more information about Google Protocol Buffers, refer to the [Google Protocol Buffers Documentation](#).

Underlying Transport Layer and Security

The DTC Protocol does not define what the underlying transport layer is for the DTC messages.

For communication over the public Internet, the underlying transport is going to be a TCP/IP network connection.

However, the websocket protocol could also be used. Although we see no advantage of using the websocket protocol in the case of when using [TLS](#) over a TCP/IP network connection.

In general, the use of the websocket protocol as the underlying transport layer for the DTC Protocol over the public Internet should be unnecessary unless there is a very specific reason why it is needed.

The underlying connection must use [TLS](#) for security when the connection is used for trading. This protects both the Username and Password from being intercepted. And if the security certificate is

checked by the Client to ensure that it is from a trusted authority, also verifies that the Client is connecting to a trusted Server.

Server Can Be Local or Remote

The DTC Protocol supports a Server which can exist remotely on the network, or locally on a client computer. For Data or Trading service providers that do not want to support connectivity straight to their backend, for whatever reason, can still support the DTC Protocol through a local Server program.

For more information about using a local server executable program, refer to [Proposal for Using a Local Server Executable Program](#).

DTC Projects

The DTC Working Group will provide simple C++ projects which use the DTC Protocol. These will serve as examples.

These will be posted here as they become available.

The full source code will be provided.

Open Specification

The specifications for the DTC Protocol has no license and is in the public domain.

The greater the adoption of the protocol by Clients and Servers, the greater the success.

The DTC Protocol is designed to be a practical and straightforward protocol that is easy to implement.

Complete and open documentation for the DTC Protocol is found on the [DTC Messages and Procedures](#) page. It has no copyrights.

These are the basic objectives of the DTC protocol:

1. It is the purpose of the DTC protocol to define a common set of messages and fields within those messages, and establish clear procedures for working with these messages and fields in order to create reliable and plug-and-play interoperability between Clients and Servers for the communication of financial market data and trading messages.
2. The header of each message contains the message type, the same as FIX messages.
3. The message types and fields for trading related messages should be the same or similar to FIX where possible and appropriate.
4. The DTC Protocol needs to be simple and should not get out of control with excessive number of messages and fields. Although it will grow as needed. It is for this reason, that managed order types like trailing stops should not be adopted by the standard DTC Protocol implementation.

Clients and Servers can define their own custom messages between each other where

the DTC Protocol does not meet their requirements.

5. The content of each message is based upon the purpose of the message.
6. The DTC Protocol supports any kind of connectivity method. Typically this will be the Internet but can work over any reliable communications network. It can use an encrypted or not encrypted connection. The protocol is simply a layer on top of the transport and encryption layers.
7. The DTC Protocol supports Client and Server identification with authentication. Just because the Client and the Server support the same protocol, does not mean that each side has to talk to each other. For example, if the server requires a license key from the Client, the server can deny the connection if the license key is not valid.
8. The DTC Protocol supports dynamic connections where the Server to connect to can change dynamically. Multiple connections are supported.

A Client or Server that follows the DTC Protocol, will be able to reliably establish connectivity to any other side supporting the DTC Protocol.

Patent/License: This protocol has no patent or license. The protocol is in the public domain. The protocol is based upon very straightforward, common sense, and established programming methods. For example, using the beginning of a data block to identify the size and the type of the data block. Therefore, this does not even remotely come close to meet the non-obvious patent test.

*Last modified Tuesday, 22nd November, 2022.