MEMOIR Common For Crypto
Version 2.0

This specification is being provided to you strictly for informational purposes solely for the purpose of developing or operating systems that interact with EDX Markets, or EDXM. All proprietary rights and interest in this specification and the information contained herein shall be vested in EDXM and all other rights including, but without limitation, patent, registered design, copyright, trademark, service mark, connected with this publication shall also be vested in EDXM. No part of this specification may be redistributed or reproduced in any form or by any means or used to make any derivative work (such as translation, transformation, or adaptation) without written permission from EDXM. EDXM reserves the right to withdraw, modify, or replace the specification at any time, without notice. No obligation is made by EDXM regarding the level, scope, or timing of EDXM's implementation of the functions or features discussed in this specification.

THE SPECIFICATION IS PROVIDED "AS IS", "WITH ALL FAULTS" AND EDXM MAKES NO WARRANTIES AND DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, OR STATUTORY RELATED TO THE SPECIFICATIONS. EDXM IS NOT LIABLE FOR ANY INCOMPLETENESS OR INACCURACIES IN THE SPECIFICATIONS. EDXM IS NOT LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES RELATING TO THE SPECIFICATIONS OR THEIR USE.

Table of Contents

< THIS PAGE INTENTIONALLY LEFT BLANK>

1 Overview

Defines the MEMOIR messages that are common to the MEMOIR Depth For Crypto and MEMOIR Top For Crypto feeds.

The MEMOIR Common For Crypto message set consists of the following:

- **Instrument Directory -** Supplies a mapping between symbology identifiers and token identifiers, which are used for order entry, and market data.
- **Instrument Trading Status** - Provides status messages to inform participants of market events for an instrument.
- **Trading Session Status** - Provides status messages to inform participants of trading session status events.
- **Snapshot Complete** - Used on Snapshot ports - Indicates that the entire snapshot has been sent and provides the point to continue reading the live data (see Snapshot under Message State/Recovery Methods)

Application messages are implemented using a binary protocol based on SBE (Simple Binary Encoding).

2 Encoding

The MEMOIR feeds use the FIX Trading Community's Simple Binary Encoding (SBE) to specify message encoding. More information about SBE can be found at the FIX SBE XML Primer.

The feed is always encoded in Big Endian byte order.

2.1 Data Types

All encoding and decoding for SBE is centered around a set of basic primitive types.

For more information on primitive type encoding, see the SBE specification.

Type	Length (bytes)	Description	Value Range	Null Value	Null Value (Hex)
CHAR	1	ASCII Character	`0` (`NUL`) to `127` (`DEL`)	`0`	`0x00`
INT8	1	Signed Integer	`-127` to `127`	`-128`	`0x80`
INT16	2	Signed Integer	`-32767` to `32767`	`-32768`	`0x8000`
INT32	4	Signed Integer	`-2^31 + 1` to `2^31 - 1`	`-2^31` (`-2147483648`)	`0x80000000`
INT64	8	Signed Integer	`-2^63 + 1` to `2^63 - 1`	`-2^63`	`0x8000000000000000`
UINT8	1	Unsigned Integer	`0` to `254`	`255`	`0xFF`
UINT16	2	Unsigned Integer	`0` to `65534`	`65535`	`0xFFFF`
UINT32	4	Unsigned Integer	`0` to `2^32-2`	`2^32 - 1` (`4294967295`)	`0xFFFFFFFF`

The MEMOIR specification does not use the SBE floating point data types.

2.1.1 Boolean

The SBE specification does not define a primitive data type for booleans. MEMOIR defines one as a single UINT8 value, set to 1 for “true”, and 0 for “false”.

2.1.2 FixedPointDecimal

Prices and some other values are encoded as a fixed-point scaled decimal, consisting of a signed long (int64) mantissa and a constant exponent of -8.

Type Name	Length	Type	Description
Mantissa	8	INT64	The fixed-point decimal representation of the number.
Exponent	N/A	INT8	MEMOIR uses a constant exponent of -8 for all fixed-point integers. This field is constant, and as such will not be transferred on the wire.

For example, a mantissa of `123456789` with the constant exponent of `-8` represents the decimal number `1.23456789`, and would appear encoded on the wire as the hex value `00000000075BCD15`.

2.1.3 String

Strings are fixed length ASCII based character sequences. Lengths are defined in the schema. Variable length fields are not supported.

2.1.4 Currency

Currencies are expressed as (up to) 3-character strings, right-padded with NULL (`0x0`) bytes, if required. Currency code values will align with the ISO 4217 standard, where possible. For currencies which do not have a matching ISO value, a custom, non-conflicting value will be supplied.

2.1.5 BooleanType

Boolean values are not defined specifically in SBE. The schema defines a BooleanType which represents a numeric value with True = `1` and False = `0`.

2.1.6 IDType

A 16-byte globally unique identifier representing an entity, expressed for convenience as two 8-byte integers. This identifier will be unique across session and day boundaries.

Note that this identifier may not conform to a UUID standard. Readers should not attribute any independent significance to the numeric value of the higher or lower bit fields in isolation.

Type Name	Length	Type	Description
UpperBits	8	INT64	The most significant bits of the ID
LowerBits	8	INT64	The least significant bits of the ID

2.1.7 UTC Timestamp Nanos

Fields with the `UTCTimestampNanos` type represents a timestamp in Coordinated Universal Time (UTC) which begins at the UNIX epoch (January 1, 1970 at 00:00:00 UTC).

UTCTimestampNanos has the time unit of nanoseconds and is encoded as follows:

Type Name	Length	Type	Description
Time	8	INT64	UTC timestamp since unix epoch with nanosecond precision.
Unit	N/A	UINT8	Unit of time. UTCTimestampNanos are represented in nanoseconds. This field is constant (value=`9`) and as such will not be transferred on the wire.

2.2 Header

SBE includes a header for each message. The SBE header is followed by the SBE body for the message.

The common SBE message header contains the fields that allows the decoder to identify what codec should be used as the template for a message.

The MEMOIR SBE header appears on the wire as:

Field	Offset	Length	Type	Description
BlockLength	0	2	UINT16	The number of bytes in the message body (does not include the header bytes). Note that MEMOIR

Field	Offset	Length	Type	Description
				messages do not use repeating groups or variable-length fields.
TemplateID	2	1	UINT8	Identifier of the message template (ie. the message type).
SchemaID	3	1	UINT8	The identifier of a message schema. NOTE: SchemaID=`6` for MEMOIR Depth For Crypto. SchemaID=`7` for MEMOIR Top For Crypto.
Version	4	2	UINT16	The version number of the message schema that was used to encode a message. Two pieces of information are packed into the (UINT16) version field: a major version, and a minor version. For example, a version of `258` (hex `0102`) indicates major version `1`, minor version `2`.

3 Message Field Types

All messages are composed of fields. Each field has a type.

This section defines the field types, their underlying wire type, acceptable values and a description of the field.

3.1 SideType

SideType describes the side the order is on: either bid (buy) or offer (sell).

SideType is a 1-byte CHAR value.

Value	Name
B	Buy
S	Sell

3.2 InstrumentTradingStatusType

InstrumentTradingStatusType represents the current trading state of an instrument on the exchange.

InstrumentTradingStatusType is a 1-byte CHAR value.

Value	Name
H	Halted
Q	Quoting
L	Limit-Only Trading
T	Trading (All Order Types)

3.3 InstrumentTradingStatusReasonType

InstrumentTradingStatusReasonType represents the reason for this instrument trading status.

InstrumentTradingStatusReasonType is a 1-byte CHAR value.

Value	Name	Description
X	None	The instrument trading status does not apply (e.g the instrument is trading normally).
A	Administrative	The instrument trading status change originated from the exchange

3.4 TradingSessionType

TradingSessionType represents the trading session for all symbols on the market.

TradingSessionType is a 1-byte CHAR value.

Value	Name	Description
1	Trading	Market session
2	Closed	Market closed, or undergoing temporary maintenance

4 Messages

This section defines the messages that make up the protocol. For each message, it lists the fields in the message, as well as each field's position and length in the message, its underlying type, and a description of its purpose.

4.1 Instrument Directory

At the start of a trading session, an instrument directory message will be sent for all tradable token pairs on the exchange. The `TokenID` field uniquely specifies a particular token trading pair, and is used to identify all subsequent market data events relevant to this instrument, as well as for order entry.

Field	Offset	Length	Type	Meaning
SBE Header	0	6	N/A	SBE Header with SchemaID set as specified in Header definition above, TemplateID=1, BlockLength=33
Timestamp	6	8	UTCTimestampNanos	The timestamp when the event occurred.
TokenID	14	8	CHAR	A unique code identifying the instrument. Generally, this will take the form `<Base Currency>/<Quote Currency>`
BaseCurrency	22	3	Currency	The base currency of the trading pair.
QuoteCurrency	25	3	Currency	The quote currency of the trading pair. (The currency that quotations and pricing data are denominated in)
UnitMultiplier	28	2	INT16	The amount of 1 unit of order quantity, expressed in the base currency. UnitMultiplier is the multiplier that is applied to all order quantities for a given token to calculate quantity of the token being referenced. For example, for BTC/USD with UnitMultiplier = -8, an order for BTC/USD with order quantity = 350, will be multiplied by 10^{-8}, resulting in a quantity of 0.00000350 BTC. All quantities referenced in MEMO messages are specified as a whole numbers of units.
IsTestSymbol	30	1	BooleanType	Determines if this security is a test instrument.
MPV	31	8	FixedPointDecimal	The minimum price variation for an instrument, used as the smallest quoting price increment.

4.2 Instrument Trading Status

This message will inform the client of the current trading status of an instrument on the exchange, and is sent for all instruments after a corresponding `Instrument Directory` message is sent.

The exchange may internally halt trading for an instrument for administrative or operational reasons at any time.

This message can and will be sent out throughout the trading session to indicate realtime changes in the instrument state.

If an `InstrumentTradingStatus` message is not received for an instrument, it should be assumed that the status is `Halted`.

Field	Offset	Length	Type	Meaning
SBE Header	0	6	N/A	SBE Header with SchemaID set as specified in Header definition above, TemplateID=2, BlockLength=18
Timestamp	6	8	UTCTimestampNanos	The timestamp when the event occurred.
TokenID	14	8	CHAR	A code uniquely identifying the instrument from the Instrument Directory.
InstrumentTradingStatus	22	1	InstrumentTradingStatusType	The current trading state.
InstrumentTradingStatusReason	23	1	InstrumentTradingStatusReasonType	The source of the trading status change.

4.3 Trading Session Status

The trading system has entered a new trading session.

Field	Offset	Length	Type	Meaning
SBE Header	0	6	N/A	SBE Header with SchemaID set as specified in Header definition above, TemplateID=3, BlockLength=9

Field	Offset	Length	Type	Meaning
Timestamp	6	8	UTCTimestampNanos	The timestamp when the event occurred.
TradingSession	14	1	TradingSessionType	The trading session which was entered.

4.4 Snapshot Complete

End of the snapshot message, all messages have been sent. Used only in Snapshot recovery (see Message/State Recovery Methods below)

Upon receipt of this message, clients should disconnect from the MEMOIR Snapshot feed and follow the recovery procedure to reconcile their state against the incremental MEMOIR feed.

Field	Offset	Length	Type	Meaning
SBE Header	0	6	N/A	SBE Header with SchemaID set as specified in Header definition above, TemplateID=4, BlockLength=16
Timestamp	6	8	UTCTimestampNanos	The timestamp when the event occurred.
AsOfSequenceNumber	14	8	INT64	The sequence number on the real-time multicast channel that this snapshot is based upon. All messages buffered from the real-time feed that are less than or equal to this number should be discarded. All messages buffered after this number should be applied. (with appropriate gaps filled normally).

5 Message/State Recovery Methods

When MEMOIR is disseminated over the MEMX-UDP transport protocol, consumers of the feed may periodically miss messages due to the unreliable nature of the UDP protocol. Two independent Multicast groups will be provided for standard A/B arbitration and quick recovery of messages dropped due to packet loss on a single channel. However, there are cases, such as a catastrophic system issue on the client's end, or the loss of the same message from both channels, where the client may need to request data to be re-sent from the exchange. For these situations, two modes of recovery are available for users.

The first is a "gap fill" mechanism, used to recover small numbers of messages, and is typically used for small episodes of packet loss that affect the same messages on both Multicast channels.

The second is a "snapshot" mechanism, where the current business state of the feed is sent, followed by a "live" sequence number for the client to begin processing the real-time MEMX-UDP channels.

5.1 Gap Fill

A client detects missing messages over the MEMX-UDP transport for MEMOIR by using the sequence number in the MEMX-UDP datagram header to determine if a gap in sequence numbers has occurred. In order to recover these messages, the client may use a connection to a Gap Fill Server via the MEMX-TCP Replay mode to request the missing messages.

NOTE: The client may initiate and maintain a connection to the MEMX-TCP Gap Fill Server early (to verify connectivity) and stay connected (while periodically sending heartbeats) until the functionality is needed.

The process by which a client recovers missing messages via the Gap Fill Server is as follows:

1. Issue a MEMX-TCP `ReplayRequest` to the MEMOIR Gap Fill Server, containing the `SessionID` received over the MEMX-UDP channel(s), the `NextSequenceNumber` of the first missing message, and the `Count` of messages required (including the sequence provided)
2. While the above request is being processed, the client should buffer any received MEMX-UDP multicast messages and not apply them until the missing sequence numbers have been recovered.
3. After the receipt of the MEMX-TCP Replay Request, the Gap Fill Server will send the following:
    a. A MEMX-TCP `ReplayBegin` message, with `NextSequenceNumber` set to the requested Sequence number, and a `PendingMessageCount` set to the number of messages to follow in the replay. (This may be less than the requested count - see NOTE below)
    b. The requested number of **MEMOIR** messages (or less given the constraints listed above on request size), starting at the requested sequence number
    c. A MEMX-TCP `ReplayComplete` message
4. After the `ReplayComplete` message is received, the client is free to send another request (if this request did not satisfy the entire gap)
5. Once the client has received all of the missing messages from one or more replay requests to the MEMX-TCP Replay server, the client may then process the messages buffered from the MEMX-UDP channel, discarding duplicate sequence numbers and applying future sequence numbers in order.

NOTE: the replay provided may consist of fewer messages than requested due to some constraints on the size of the response. The number of messages in the replay response will be the minimum of:

- the requested count
- the configured maximum messages per request for the service
- the number of messages remaining from the requested start sequence to the highest published sequence

5.2 Snapshot

If a catastrophic failure occurs on the client side that requires a full state reset on the client, and the client does not wish to manually replay the entire day via multiple MEMX-TCP Replay requests, the client may opt to use the MEMX-TCP snapshot mechanism to recover state. Once the snapshot has completed, the client should use normal Gap Fill processing (described above) to recover any further gaps in data that may occur during that trading session, as Snapshots can be quite large compared to a gap of one or two datagrams.

NOTE: The client may initiate and maintain a connection to the MEMX-TCP snapshot server early (to verify connectivity) and stay connected (while periodically sending heartbeats) until the functionality is needed.

The process by which a client recovers via snapshot is as follows:

1. Join the real-time MEMX-UDP multicast group(s) for the data feed, and begin buffering all received messages.
2. Issue a MEMX-TCP `ReplayAllRequest` to the MEMOIR snapshot server, containing the `SessionID` received over the MEMX-UDP channel(s)
3. After the receipt of the MEMX-TCP ReplayAll request, the Snapshot Server will send the following:
    a. A MEMX-TCP `ReplayBegin` message, with `NextSequenceNumber` set to 1 and `PendingMessageCount` set to the number of MEMOIR Business Level messages contained in the snapshot
    b. MEMOIR Business Level `InstrumentDirectory`messages containing an instrument description and associated TokenID to be used throughout the session.
    c. MEMOIR Business Level `InstrumentTradingStatus` messages containing the current trading status for each instrument
    d. MEMOIR Business Level `TradingSessionStatus` message containing the current trading session status.
    e. For MEMOIR Depth For Crypto: MEMOIR Business Level `AddOrder` messages for each instrument. For MEMOIR Top For Crypto: Business Level 1 or 2-sided book updates `BestBidOffer`, `BestBid`, `BestOffer` messages for each instrument.
    f. A single MEMOIR Business Level `SnapshotComplete` event containing an `AsOfSequenceNumber` set to the sequence number on the MEMX-UDP multicast channel that this snapshot was taken (clients should resume processing at this number + 1 on the multicast feed after applying the snapshot)
    g. A MEMX-TCP `ReplayComplete` message
4. Discard all buffered events from the MEMX-UDP real-time feed at or before the `AsOfSequenceNumber` in the received Snapshot Complete event.

5. Apply all buffered events from the MEMX-UDP real-time feed after the `AsOfSequenceNumber` in the received Snapshot Complete event, in sequence order.
6. Continue to process the MEMX-UDP real-time multicast feed normally from this point onward, using the **Gap Fill** mechanism above to recover dropped messages.