MILLISTREAM

# The MDF C/C++ API

Technical Description and Reference Guide

15 May 2024

**Millistream Market Data AB**
Box 5243, 402 24 Göteborg, Sweden

**www.millistream.com**
Org nr: 556763-4117

# Installation

The MDF API can be downloaded from : **`https://packages.millistream.com`** Binaries are available for Windows, Linux and macOS. Full source code is also available for download for those willing to compile with their own compiler or to support operating systems for which there is no pre-built binaries, the source can be found at https://packages.millistream.com/source/

On Unix/Linux, the source can be compiled with the standard, "./configure && make && [sudo] make install". For Windows we currently only support building with MSYS2.

To build on macOS, Homebrew have to be installed and the packages "pkg-config", "zstd" and "openssl" have to be installed via brew.

MDF links against zlib (http://zlib.net), zstd (https://github.com/facebook/zstd) and OpenSSL (https://openssl.org/). Pre-built binaries of both is available in the Windows binary zip archive, but not for the Linux variants since they should be available through your distribution repository.

Example programs using libmdf can be downloaded from https://packages.millistream.com/source/mdf_examples.zip

# Includes

API function declarations are collected in the **`mdf.h`** include file. Definitions for messages, tags etc are collected in the **`mdf_fields.h`** include file.

### Definition prefixes

| Prefix | Description |
|---|---|
| MDF_M_ | Message References |
| MDF_MC_ | Message Classes |
| MDF_RC_ | Request Classes |
| MDF_RT_ | Request Types |
| MDF_F_ | Fields / Tags |
| MDF_CA_ | Corporate Action Types |
| MDF_DLY_ | Message Delay |
| MDF_TC_ | Trade Codes |
| MDF_OPT_ | API Properties |
| MDF_MSG_OPT_ | Message Properties |
| MDF_ERR_ | API Errors |
| MDF_STATUS_ | API Status Callback Status Codes |

So for example, the Quote message is defined as MDF_M_QUOTE and the Bid Price field is defined as MDF_F_BIDPRICE. A detailed description of the available messages are documented in the *MDF Messages Reference* document, and the fields are documented in the *MDF Fields Reference* document.

# Linking

Applications should link with **`libmdf.so`** on Unix/Linux (**`-lmdf`**), or **`libmdf-0.dll`** on Windows (use **`libmdf.lib`** as the import library for Microsoft Visual C or **`libmdf.dll.a`** for MinGW).

# The API Handle

Each connection to the Millistream system is managed by the API handle, this handle is represented by the opaque pointer **`mdf_t`**.

Handles are created with **`mdf_create()`** and destroyed with **`mdf_destroy()`**. Handles are not thread-safe, so if multiple threads will share access to a single handle, the accesses has to be serialized using a mutex or other forms of locking mechanisms. The API as such is thread-safe so multiple threads can have local handles without the need for locks.

# Connecting to the Millistream System

A API handle can be connected to the Millistream System with **`mdf_connect()`**. A comma separated list of servers can be supplied and the API will try each server in turn until it find one that answers.

After the connect has succeeded, the client must login with the **MDF_M_LOGON** message before the server grants access to the system. How to send messages to the Millistream system is described in a later section, so we have supplied a small example of how to connect and send the login message. Please note that this example lacks all form of error handling. It will connect to the server “sth2.millistream.com” at tcp port '9000'.

```
mdf_t mdf;
mdf_message_t msg;

mdf = mdf_create();
mdf_connect(mdf, “sth2.millistream.com:9000”);

msg = mdf_message_create();
mdf_message_add(msg, 0, MDF_M_LOGON);
mdf_message_add_string(msg, MDF_F_USERNAME, “testuser”);
mdf_message_add_string(msg, MDF_F_PASSWORD, “t0ps3cr1t”);
mdf_message_send(mdf, msg);
mdf_message_destroy(msg);
      ...
mdf_destroy(mdf);
```

The connection can be disconnected with **`mdf_disconnect()`**, any current connection will also automatically be disconnected when the handle is destroyed.

It is possible to register a status callback, which will be called by the API during the various stages of the connect procedure (and when/if the connection is disconnected). See the section of properties for how to register such a callback function.

# Receiving Data

Once the connection is up, the client must consume data sent from the server with **`mdf_consume()`**. The consume function will return when there is messages to decode, if the connection has been disconnected or when the connection has been idle for the number of seconds supplied by the client (which can be immediately if the client supplied '0' as timeout).

Clients wanting to use their own I/O event notification system can fetch the connection socket / file descriptor via the properties, and call the consume function with '0' as timeout when it notices that there is data to be consumed. Clients doing this should make sure to periodically call the consume function even if there is no incoming data so that the API can send heartbeat requests to the server, so a stale connection can be detected properly (the **`MDF_OPT_TIMEOUT`** property can be fetched to know how long to wait before having to call the consume function).

If the consume function returns true, the client must decode the received messages with **`mdf_get_next_message2()`**, it is also possible to register a data callback function that will be called by the consume function when there is messages to decode (the consume function will no longer return true if a data callback function is registered).

When **`mdf_get_next_message2()`** returns true, the individual fields of the message can be retrieved by repeating calls to **`mdf_get_next_field()`** until it returns false. Since messages can be empty (for example if all the fields are null on a reply to an IMAGE request) there is no guarantee that **`mdf_get_next_field()`** will return true after a **`mdf_get_next_message()`**.

# Requesting Data

The Millistream Data Feed is a request based feed, so upon a successful login, there will be no messages sent by the server unless the client requests them (there are a few exceptions such as heartbeats). Requests can be of two types; stream or image.

Stream requests (**`MDF_RT_STREAM`**) are realtime streaming data, I.e when an event happens at a market place, the event is immediately sent to the client by the server in realtime. The first message for an instrument after a stream request is with only the fields that where changed due to the event.

Image requests (**`MDF_RT_IMAGE`**) are requests for the current image of the instruments. There will only be a single reply for each instrument in an image request and it will be a complete message will all the fields that currently contain a value (null values will be sent if they have been deliberately set to null by a stream event). If **`MDF_F_DATE`** and/or **`MDF_F_TIME`** is present in the request, only the fields that have been updated since the time+date specified are sent.

Usually, clients will issue combined requests (I.e stream+image) since they want to know the current image of the instruments and also subscribe to the realtime changes to this image. This is known as a full request (**`MDF_RT_FULL`**).

Currently the request functionality of the Millistream system is somewhat limited (for example there is no possibility to request trades or news messages sent previous to the request, and there is not possibility to request only instruments of a specific type or belonging to a specific market or list).

The filtering possibility in the current version of the system is to request all available instruments or to supply a space separated list of the instruments to subscribe to. For example a client application could request Basic Data for all instruments and then issue requests based upon the received Basic Data, this works quite well since Basic Data is a low messages per second stream.

Issuing requests is done be sending messages to the Millistream system and since it is described separately in a later section we have created a small example on how to issue requests for both Basic Data and Quotes, the request is a full request and for all available instruments.

```
mdf_message_t msg = mdf_message_create();


mdf_message_add(msg, 0, MDF_M_REQUEST);
```

```
mdf_message_add_list(msg, MDF_F_REQUESTCLASS, MDF_RC_BASICDATA " "
MDF_RC_QUOTE);
mdf_message_add_numeric(msg, MDF_F_REQUESTTYPE, MDF_RT_FULL);
mdf_message_add_string(msg, MDF_F_INSREFLIST, "*");

mdf_message_send(mdf, msg);
```

If the client adds a **MDF_F_REQUESTID** field to a request, a **MDF_M_REQUESFINISHED** message will be sent when the image request has been completed in full (and the corresponding request id will also be returned with that message).

When issuing full requests, each instrument will be opened in turn, I.e first the image will be sent and then the stream subscription will be enabled on that instrument. So the first message for every instrument in a full request will always be a complete message, and each “opened” instrument will begin to send it's realtime stream as soon as possible meaning that the client will not have to wait for the whole image part to complete before receiving realtime data.

Trades and news does currently not contain image data, so full requests for these messages will be handled like stream requests only.

By adding **MDF_F_DATE** and **MDF_F_TIME** to an image or full request, the server will only send data that has been updated since that date+time. This can be useful for example on reconnect where a client do not want to perform a complete download of all the messages and fields in order to limit the data needed to be received and processed.

When the client no longer want to subscribe to realtime data it can issue a **MDF_M_UNSUBSCRIBE** message to unsubscribe to specified messages and instruments. It's possible to unsubscribe to a subset of the instruments and messages that the client is subscribing to, i.e the list of instruments and messages to the unsubscribe request does not have to fully match a previous request message.

The **MDF_F_INSREFLIST** field should contain a space separated list of the insrefs that you want to request to or unsubcribe from. If the field instead contains the character "*****" the request or unsubscribe is for all instruments that the account is entitled to.

The **MDF_F_REQUESTCLASS** field should contain a space separated list of the Request Classes that you want to request to or unsubcribe from. If the field instead contains the character "*****" the request or unsubscribe is for all Request Classes that the account is entitled to.

# Properties

It is possible to modify and fetch properties with **mdf_set_property()** and **mdf_get_property()**.

| Property | Type | Description |
|---|---|---|
| MDF_OPT_FD | int | The file descriptor used by the connection. Will be -1 (or INVALID_SOCKET on Windows) if there is no connection. |
| MDF_OPT_ERROR | MDF_ERROR | The current API error code |
| MDF_OPT_RCV_BYTES | uint64_t | The number of bytes received by the server since the handle was created |
| MDF_OPT_SENT_BYTES | uint64_t | The number of bytes sent by the client since the handle was created |
| MDF_OPT_DATA_CALLBACK_FUNCTION | mdf_data_callback | This callback function will be called by the consume function if there is any messages to decode |

| | | |
|---|---|---|
| MDF_OPT_DATA_CALLBACK_USERDATA | void* | Custom userdata that will be available to the data callback function |
| MDF_OPT_STATUS_CALLBACK_FUNCTION | mdf_status_callback | This callback function will be called whenever there is a change of the status of the connection |
| MDF_OPT_STATUS_CALLBACK_USERDATA | void* | Custom userdata that will be available to the status callback function |
| MDF_OPT_CONNECT_TIMEOUT | int | The number of seconds before determining that a connect attempt has timed out. Valid values are '1' to '60'. The default is '5'. |
| MDF_OPT_HEARTBEAT_INTERVAL | int | The number of seconds the connection must be idle before the API sends a heartbeat request to the server. Valid values are '1' to '86400'. The default is '30'. |
| MDF_OPT_HEARTBEAT_MAX_MISSED | | How many outstanding hearbeat requests to allow before the connection is determined to be disconnected. Valid values are '1' to '100'. The default is '2'. |
| MDF_OPT_TCP_NODELAY | int | '1' will disable the Nagle algorithm, '0' will enable it. The default is '0'. |
| MDF_OPT_NO_ENCRYPTION | int | '1' will disable encryption of the traffic from the client to the server (traffic from the server to the client is unaffected by this setting), must be set prior to calling mdf_connect(). The default it '0'. |
| MDF_OPT_TIME_DIFFERENCE | int | The time difference in number of seconds between the client and the server. The value should be added to the current time on the client in order to get the server time. Please not that this value can be negative if the client clock is ahead of the server clock. Updated when the client is receiving either a Heartbeat Response or a Heartbeat Request from the server. |
| MDF_OPT_TIME_DIFFERENCE_NS | int64_t | As MDF_OPT_TIME_DIFFERENCE but the difference is in number of nano seconds. |
| MDF_OPT_BIND_ADDRESS | char * | A numerical address to which the API will bind before attempting to connect to a server in mdf_connect(). If the bind fails then mdf_connect() also fails. The string is copied by the API and a NULL value can be used in order to ‘unset’ the bind address. |
| MDF_OPT_CRYPT_DIGESTS | char * | A comma separated list of the message digests that the client will offer to the server upon connect. The value is copied by the API and a NULL value can be used to set the list back to default. |

| | | |
|---|---|---|
| | | The caller must make a copy of the returned value if (s)he want's to keep it. |
| MDF_OPT_CRYPT_CIPHERS | char * | A comma separated list of the encryption ciphers that the client will offer to the server upon connect. The value is copied by the API and a NULL value can be used to set the list back to default.<br><br>The caller must make a copy of the returned value if (s)he want's to keep it. |
| MDF_OPT_CRYPT_DIGEST | char * | Returns the digest chosen by the server, only available after mdf_connect() returns. |
| MDF_OPT_CRYPT_CIPHER | char * | Returns the cipher chosen by the server, only available after mdf_connect() returns. |
| MDF_OPT_TIMEOUT | int | Returns the number of seconds to wait before having to call mdf_consume(). |
| MDF_OPT_HANDLE_DELAY | int | '1' will enable delay-mode in where the server adds the intended delay to each message sent. This also enables the client to set the intended delay of the messages the client sends to the server. Must be set prior to calling mdf_connect(). The default it '0'. |
| MDF_OPT_RBUF_SIZE | size_t | Returns the number of bytes waiting to be processed in the internal read buffer after a call to mdf_consume(). |
| MDF_OPT_RBUF_MAXSIZE | size_t | Returns or sets the size of the internal read buffer. |

Example:

```
/* this is our custom data callback function */
void our_callback (void *udata, mdf_t handle)
{
      uint16_t mref;
      uint64_t insref;

      while (mdf_get_next_message2(handle, &mref, &insref)) {
            uint32_t tag;
            char *value;

            while (mdf_get_next_field(handle, &tag, &value)) {
                  ...
            }
```

```
        }
}

/* register a data callback and custom userdata */
struct our_userdata *udata = xxx;
mdf_set_property(mdf, MDF_OPT_DATA_CALLBACK_FUNCTION, our_callback);
mdf_set_property(mdf, MDF_OPT_DATA_CALLBACK_USERDATA, our_userdata);

/* fetching the received number of bytes, and setting it to zero */
uint64_t val;
mdf_get_property(mdf, MDF_OPT_RCV_BYTES, &val);

val = 0;
mdf_set_property(mdf, MDF_OPT_RCV_BYTES, &val);
```

# Message Properties

It is possible to set message properties with **`mdf_message_set_property()`**.

| Property | Description |
|---|---|
| MDF_MSG_OPT_UTF8 | Enables (1) or Disables (0) UTF8 validation of the strings given to ***mdf_message_add_string()*** and ***mdf_message_add_string2()***. The default value is Enabled(1). |
| MDF_MSG_OPT_COMPRESSION | Sets the **zlib** compression level for ***mdf_message_add_string()*** and ***mdf_message_add_string2()***. Value must be between ***Z_NO_COMPRESSION(0)*** and ***Z_BEST_COMPRESSION(9)***. The default value is ***Z_BEST_SPEED(1)*** since v1.0.21 and ***Z_BEST_COMPRESSION(9)*** on older versions. |
| MDF_MSG_OPT_DELAY | Sets the intended delay for the current and/or future messages. The setting will remain until changed (a call to ***mdf_message_reset()*** will not reset the setting back to the default). The default value is **MDF_DLY_REALTIME** (Realtime) for sending data and **MDF_DLY_BEST** for requests. |

# Sending Data to the System

Besides simple requests and logins, clients can also contribute data to the Millistream system. This data can then be requested by other clients (if they have the proper permissions). The ability to send data into the system is governed by permissions dictating whether the client is allowed to send any data at all, which data to send, for which instruments and if the client can create new instruments (and how many if so).

Messages to send is handled by the **`mdf_message_t`** handle, which is created with **`mdf_message_create()`** and destroyed with **`mdf_message_destroy()`**. The message handle can carry multiple messages, and a new message is

added to the handle with **`mdf_message_add()`**.

Once a message has been added to the message handle, fields can be added with the following functions:

**`mdf_message_add_numeric()`**

**`mdf_message_add_int(), mdf_message_add_uint()`**

**`mdf_message_add_string(), mdf_message_add_string2()`**

**`mdf_message_add_date(), mdf_message_add_date2()`**

**`mdf_message_add_time(), mdf_message_add_time2(), mdf_message_add_time3()`**

**`mdf_message_add_list()`**

Most functions takes the field value as a UTF-8 string, the reason for the _numeric(), _string() etc is to let the API know what type of data you are adding so it can improve the compression; numeric values can experience a very high rate of compression minimizing the bandwidth needed.

The active messages in a handle can be sent to the server with **`mdf_message_send()`**, and the message handle can be reused with **`mdf_message_reset()`**. If reused, the handle keeps the memory currently allocated in order to increase the performance due to far less calls to **`malloc()`** and **`free()`**.

The current message in the handle can be removed from the handle with **`mdf_message_del()`**, repeated calls of this function until it returns false is equal to **`mdf_message_reset()`**.

The number of active messages can be fetched with **`mdf_message_get_num_active()`** and the total number of messages in the handle (both active and reused) can be fetched with **`mdf_message_get_num()`**.

Fetching the number of active messages can be used to know when to call **`mdf_message_send()`** after processing some files are wherever the client will get their data to send, if there can be situations where the indata won't necessarily produce messages to send. And fetching the total number of messages can be used to know when it is time to destroy the message handle in order to release some memory if the number goes over a set limit.

As of v1.0.29 multiple threads can call **`mdf_message_send()`** simultaneously without locks to the same mdf handle as long as they use thread specific message handles.

# mdf_create()

**SYNOPSIS**

```
mdf_t mdf_create();
```

**DESCRIPTION**

Creates a new API handle, this should normally by the first function to call in your application. The handle is not thread-safe, but different threads can different handles simultaneously as long as they don't access the same handle.

On Unix systems, **`SIGPIPE`** will be disabled and will not be restored when the handle is destroyed. So if your application depends upon this signal, you will have to reset the signal after the call to this function. Note that as of v1.0.26 this only happens on systems without **`MSG_NOSIGNAL`** or **`SO_NOSIGPIPE`** which AFAIK only leaves Solaris.

**RETURN VALUE**

On success, a newly created API handle is returned. On error, **`NULL`** is returned.

**ERRORS**

None

# mdf_destroy()

**SYNOPSIS**

```
void mdf_destroy(mdf_t handle);
```

**DESCRIPTION**

Destroys the API handle, any open connection will be automatically closed. All memory allocated by the handle will be freed.

**RETURN VALUE**

None

**ERRORS**

None

# mdf_consume()

**SYNOPSIS**

```
int mdf_consume(handle, int timeout);
```

**DESCRIPTION**

Consumes data sent from the server. If there currently is no data the function waits for maximum **timeout** number of seconds for new data to arrive from the server. If **timeout** is zero (0) the function will return immediately if there is no data from the server. If **timeout** is negative then the wait period is treated as number of microseconds instead of number of seconds (i.e -1000 will wait a maximum of 1000µs).

**RETURN VALUE**

Returns 1 if data has been consumed that needs to be handled by **mdf_get_next_message()** and no callback function has been registered. The function returns 0 on timeout or if a callback function is registered and there was data. On errors, -1 will be returned (and the connection will be dropped).

**ERRORS**

| | |
|---|---|
| **`MDF_ERR_NOT_CONNECTED`** | There is not connection with a server. |
| **`MDF_ERR_DISCONNECTED`** | We where disconnected. |
| **`MDF_ERR_CONNECTION_IDLE`** | There has been no timely reply to any of our heartbeats so the connection has been disconnected. |
| **`MDF_ERR_MSG_OOB`** | The received message contained lengths that would overflow the message |
| **`MDF_ERR_NO_MEM`** | There was not sufficient available memory to fulfill a call to **`realloc()`** when resizing the consume buffer to accommodate for the incoming message. |

# mdf_get_next_message2()

**SYNOPSIS**

```
int mdf_get_next_message2(handle, uint16_t *mref, uint64_t *insref);
```

**DESCRIPTION**

Fetches a message from the current consumed data if one is present and fills the parameters with values representing the message fetched.

The message reference will be returned in **`mref`**, this is the value to match with the **`MDF_M_`** defines.

**`insref`** is the instrument reference, and is the unique id of the instrument within the Millistream universe. A instrument should never change instrument reference, and instrument references will never be reused. There are a limited number of messages in where **`insref`** will not be used to carry the instrument reference, please consult the *MDF Messages Reference* document for more information.

**RETURN VALUE**

Returns true (1) if a message was returned (and the mref and insref fields will be filled) or

false (0) if there are no more messages in the current consumed data (or an error occured).

**ERRORS**

| | |
|---|---|
| **`MDF_ERR_NO_ERROR`** | No errors occurred when decoding the message |
| **`MDF_ERR_MSG_OOB`** | The received message contained lengths that would overflow the message |
| **`MDF_ERR_AUTHFAIL`** | The authentication of the message failed due to either communications errors or an injection attempt. The connection with the server has been dropped. |

# mdf_get_next_message()

**SYNOPSIS**

```
int mdf_get_next_message(handle, int *mref, int *mclass,
                         uint64_t *insref);
```

**DESCRIPTION**

Fetches a message from the current consumed data if one is present and fills the parameters with values representing the message fetched.

The message reference will be returned in **`mref`**, this is the value to match with the **`MDF_M_`** defines.

The message class is returned in **`mclass`**, which will match the **`MDF_MC_`** defines. The message class is normally only used internally by Millistream and is supplied to the client for completeness and transparency, the client should under most circumstances only use the message reference in order to determine which message it has received.

i**`nsref`** is the instrument reference, and is the unique id of the instrument within the Millistream universe. A instrument should never change instrument reference, and instrument references will never be reused. There are a limited number of messages in where **`insref`** will not be used to carry the instrument reference, please consult the *MDF Messages Reference* document for more information.

New clients should use **mdf_get_next_message2()** instead, this function is now only a wrapper around that function anyway, the reason for this is that the values used for the message classes requires a larger data type than the **int** used here.

**RETURN VALUE**

Returns true (1) if a message was returned (and the mref, mclass and insref fields will be filled) or

false (0) if there are no more messages in the current consumed data (or an error occured).

**ERRORS**

| | |
|---|---|
| **`MDF_ERR_NO_ERROR`** | No errors occurred when decoding the message |
| **`MDF_ERR_MSG_OOB`** | The received message contained lengths that would overflow the message |
| **`MDF_ERR_AUTHFAIL`** | The authentication of the message failed due to either communications errors or an injection attempt. The connection with the server has been dropped. |

# mdf_get_next_field()

**SYNOPSIS**

```
int mdf_get_next_field(handle, uint32_t *tag, char **value);
```

**DESCRIPTION**

Fetches the next field from the current message.

The field tag will be returned in **`tag`** and can be matched with the field definitions **`MDF_F_`** to determine which field that was fetched.

The field value will be returned as a UTF-8 string in the char pointer pointed to by **`value`**. The value can be **`NULL`** if the field value is supposed to be null which is a valid value in the Millistream Data Feed. Please note that the returned string points at a static string allocated within the API handle, and that it will be reused on the next call to this function. So in order to retain the value, a local copy has to be performed.

**RETURN VALUE**

Returns true (1) if a field was returned, or false (0) if there are no more fields in the current message.

**ERRORS**

**`MDF_ERR_NO_ERROR`** No errors occurred when decoding the field

**`MDF_ERR_MSG_OOB`** The current field contained lengths that would overflow the message

**`MDF_ERR_UNKNOWN_TEMPLATE`** The current message contains a message reference for which the API does not know the template, so the message cannot be decoded.

**`MDF_ERR_TEMPLATE_OOB`** The current field is outside the template defined for this message, either the API templates are out of date, or we have received some errenous data.

**`MDF_ERR_NO_MEM`** There was not sufficient available memory to fulfill a call to **`realloc()`** when resizing the value string to accommodate for a string or very long number.

# mdf_get_mclass()

**SYNOPSIS**

```
uint64_t mdf_get_mclass(handle);
```

**DESCRIPTION**

Returns the message class of the current message.

**RETURN VALUE**

Returns the message class or MDF_MC_UNDEF (0) if the message class cannot be determined.

**ERRORS**

None

# mdf_get_delay()

**SYNOPSIS**

```
uint8_t mdf_get_delay(handle);
```

**DESCRIPTION**

Returns the intended delay of the current message. Note that unless delay-mode have been activated, this function will always return a delay of Realtime (0). Also note that this is the intended delay and not the actual delay, aka any network or server latencies are not included in the value.

**RETURN VALUE**

0 for Realtime, 1 for Delay (usually 15 minutes), 2 for EndOfDay, 3 for NextDay (usually 24h delay or after midnight local exchange time) and 4 for T+1.

**ERRORS**

None

# mdf_get_property()

**SYNOPSIS**

```
int mdf_get_property(handle, MDF_OPTION option, ...);
```

**DESCRIPTION**

Return the value of the specified option. The third argument **MUST** be a pointer to the the storage type of the option to fetch.

**RETURN VALUE**

Returns true (1) if a property was fetched, or false (0) if the specified property does not exist or if the specified property does not support to be fetched.

**ERRORS**

None

# mdf_set_property()

**SYNOPSIS**

```
int mdf_set_property(handle, MDF_OPTION option, void *value);
```

**DESCRIPTION**

Modify the value of the specified option. The third argument should be a pointer that MUST be the type specified for the particular property.

**RETURN VALUE**

Returns true (1) if the property was successfully modified, or false (0) if the specified property does not exist, if the specified property cannot be modified or if the given value is out of bounds.

**ERRORS**

None

# mdf_connect()

**SYNOPSIS**

```
int mdf_connect(handle, char *servers);
```

**DESCRIPTION**

Connects to the first server in **`servers`**, which can be a comma separated list of 'host:port' pairs, where 'host' can be a DNS host name or an ip address (IPv6 addressed must be enclosed in brackets). If the server does not respond in time (**`MDF_OPT_CONNECT_TIMEOUT`**), the next server in the list will be tried until the list is empty and the function finally fails.

Upon connect, the API will verify the authenticity of the server using it's public RSA key, and a secure channel will be set up between the client and the server before the function signals success.

If this is the first successful connect on the API handle, or the templates has been updated since the last time the API was connected, the server will send a **`MDF_M_MESSAGESREFERENCE`** message to the client containing the new message templates. So you could receive one message before a successful logon request.

**RETURN VALUE**

Returns true (1) if a connection has been set up or false (0) if a connection attempt failed with every server on the list.

**ERRORS**

| | |
|---|---|
| **`MDF_ERR_ARGUMENT`** | The **`servers`** argument was missing or contained erroneous data |
| **`MDF_ERR_CONNECTED`** | The API handle is already connected, if you want to reconnect, you must first disconnect the current connection. |
| **`MDF_ERR_CONNECT`** | The connection attempt failed with every server on the list |

# mdf_disconnect()

**SYNOPSIS**

```
void mdf_disconnect(handle);
```

**DESCRIPTION**

Disconnect a connected API handle. Safe to call even if the handle is already disconnected.

**RETURN VALUE**

None

**ERRORS**

None

# mdf_extract()

**SYNOPSIS**

```
void *mdf_extract(handle, uint16_t *mref, uint64_t *insref, size_t *len);
```

**DESCRIPTION**

Extracts the next message in the stream so that the caller can later inject it into another handle. The **mref** and **insref** is returned just like in **mdf_get_next_message2()** to let the caller apply logic when deciding to which handle to later inject this message, please note that **MDF_M_MESSAGESREFERENCE** messages must be injected into every handle where you inject messages so that the handles can properly set up the message templates. The number of bytes in the message is written to the variable pointed to by **len**.

**RETURN VALUE**

A pointer to the message in the internal read buffer of the handle. Callers must make a copy of this data and not used it directly, i.e the data pointed to by the returned pointer will be overwritten by other functions of the api. Returns **NULL** when there are no more messages to extract (or an error occured).

**ERRORS**

| | |
|---|---|
| **MDF_ERR_NO_ERROR** | No errors occurred when decoding the message |
| **MDF_ERR_MSG_OOB** | The received message contained lengths that would overflow the message |
| **MDF_ERR_AUTHFAIL** | The authentication of the message failed due to either communications errors or an injection attempt. The connection with the server has been dropped. |

# mdf_inject()

**SYNOPSIS**

```
int mdf_inject(handle, void *ptr, size_t len);
```

**DESCRIPTION**

Injects a (or multiple) message(s) into the handle previously extracted by **mdf_extract()**. Multiple messages can be injected at once by concatenating multiple messages from **mdf_extract()** into a single buffer before injecting. After injecting, the caller must call **mdf_get_next_message() / mdf_get_next_message2()** as usual until it returns false to indicate that all messages have been processed before injecting a new message.

The handle used to process injected messages cannot be used for other purposes, i.e it cannot be connected to a server and **mdf_consume()** should never be called on it.

The data given via **ptr** is copied by the API so the same data can be injected into multiple different handles at the same time, and the caller can free or overwrite the data once **mdf_inject()** have returned without risking any side effects.

**RETURN VALUE**

Returns true (1) if the message was injected or (0) if an error occured.

**ERRORS**

| | |
|---|---|
| **`MDF_ERR_NO_MEM`** | There was not sufficient available memory to fulfill a call to **`realloc()`** when resizing the consume buffer to accommodate for the incoming message. |

# mdf_message_create()

**SYNOPSIS**

```
mdf_message_t mdf_message_create();
```

**DESCRIPTION**

Creates a new message handle. A message handle can contain several messages for efficient sending of multiple messages to the Millistream system.

**RETURN VALUE**

On success, a newly created message handle is returned. On error, **`NULL`** is returned.

**ERRORS**

None

# mdf_message_destroy()

**SYNOPSIS**

```
void mdf_message_destroy(mdf_message_t message);
```

**DESCRIPTION**

Destroys the message handle and frees all allocated memory.

**RETURN VALUE**

None

**ERRORS**

None

# mdf_message_set_property()

**SYNOPSIS**

```
int mdf_message_set_property(handle, MDF_MSG_OPTION option, int value);
```

**DESCRIPTION**

Modify the value of the specified option. Modified options retain their value after a ***mdf_message_reset()***.

**RETURN VALUE**

Returns true (1) if the property was successfully modified, or false (0) if the specified property does not exist, if the specified property cannot be modified or if the given value is out of bounds.

**ERRORS**

None

# mdf_message_reset()

**SYNOPSIS**

```
void mdf_message_reset(mdf_message_t message);
```

**DESCRIPTION**

Resets the message handle (sets the number of active messages to zero) so it can be reused. The memory allocated for the current messages in the handle is retained for performance reasons and will be reused when you add new messages to the handle. Has the exact same end effect as calling **`mdf_message_del()`** until it returns false (0).

Settings such as the ***zlib*** compression level, intended delay and so on are retained and not reset back to their default values.

**RETURN VALUE**

None

**ERRORS**

None

# mdf_message_del()

**SYNOPSIS**

```
int mdf_message_del(mdf_message_t message);
```

**DESCRIPTION**

Removes the current active message from the message handle and all the fields that you have added for this message. Points the current message at the previous message in the message handle if it exists, so repeated calls will reset the whole message handle as if **`mdf_message_reset()`** had been called.

**RETURN VALUE**

Returns true (1) if there are more active messages in the message handle or false (0) if the message handle is now empty.

**ERRORS**

None

# mdf_message_add()

**SYNOPSIS**

```
int mdf_message_add(mdf_message_t message, uint64_t insref, int mref);
```

**DESCRIPTION**

Adds a new message to the message handle. If the current active message is empty it will be reused to carry this new message.

**RETURN VALUE**

Returns true (1) if a new message was added to the message handle (or an empty message was reused) or false (0) if there was an error.

**ERRORS**

None

# mdf_message_add2()

**SYNOPSIS**

```
int mdf_message_add2(mdf_message_t message, uint64_t insref,
                     uint16_t mref, uint8_t delay);
```

**DESCRIPTION**

Adds a new message to the message handle. If the current active message is empty it will be reused to carry this new message. Can be more convenient to use if the delay varies much between messages.

**RETURN VALUE**

Returns true (1) if a new message was added to the message handle (or an empty message was reused) or false (0) if there was an error.

**ERRORS**

None

# mdf_message_add_numeric()

**SYNOPSIS**

```
int mdf_message_add_numeric(mdf_message_t message, uint32_t tag,
                            char *value);
```

**DESCRIPTION**

Adds a numeric field to the current active message.

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied value is not of the type specified).

**ERRORS**

None

# mdf_message_add_int()

**SYNOPSIS**

```
int mdf_message_add_int(mdf_message_t message, uint32_t tag,
                        int64_t value, int decimals);
```

**DESCRIPTION**

Adds a scaled signed integer field to the current active message. **decimals** can be between 0 and 19.
A **value** of 12345 with **decimals** set to 2 will be encoded as “123.45”.

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied value is not of the type specified).

**ERRORS**

None

# mdf_message_add_uint()

**SYNOPSIS**

```
int mdf_message_add_uint(mdf_message_t message, uint32_t tag,
                         uint64_t value, int decimals);
```

**DESCRIPTION**

Adds a scaled unsigned integer field to the current active message. **decimals** can be between 0 and 19. A **value** of 12345 with **decimals** set to 2 will be encoded as “123.45”.

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied value is not of the type specified).

**ERRORS**

None

# mdf_message_add_string()

**SYNOPSIS**

```
int mdf_message_add_string(mdf_message_t message, uint32_t tag,
                           char *value);
```

**DESCRIPTION**

Adds a UTF-8 string field to the current active message. The string is compressed with **zlib** using the compression level as set by ***mdf_message_set_property()*** which is ***Z_BEST_SPEED*** by default.

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied value was not a valid UTF-8 string).

**ERRORS**

None

# mdf_message_add_string2()

**SYNOPSIS**

```
int mdf_message_add_string2(mdf_message_t message, uint32_t tag,
                            char *value, size_t len);
```

**DESCRIPTION**

Adds a UTF-8 string field to the current active message. The string is compressed with zlib using the compression level as set by ***mdf_message_set_property()*** which is ***Z_BEST_SPEED*** by default. Only the first **len** number of bytes of the string will be added, note that the string have to be minimum **len** bytes in length or this function will fail.

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied string was not at least **len** number of bytes long).

**ERRORS**

None

# mdf_message_add_date()

**SYNOPSIS**

```
int mdf_message_add_date(mdf_message_t message, uint32_t tag,
                         char *value);
```

**DESCRIPTION**

Adds a date field to the current active message. Please note that all dates and times in the Millistream system is expressed in UTC. The format of **value** must be one of “YYYY-MM-DD”, “YYYY-MM”, “YYYY-H1”, “YYYY-H2”, “YYYY-T1”, “YYYY-T2”, “YYYY-T3”, “YYYY-Q1”, “YYYY-Q2”, “YYYY-Q3”, “YYYY-Q4” or “YYYY-W[1-52]”.

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied value is not of the type specified).

**ERRORS**

None

# mdf_message_add_date2()

**SYNOPSIS**

```
int mdf_message_add_date2(mdf_message_t message, uint32_t tag,
                          int year, int mon, int day);
```

**DESCRIPTION**

Adds a date field to the current active message. Please note that all dates and times in the Millistream system is expressed in UTC.

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied value is not of the type specified).

**ERRORS**

None

# mdf_message_add_time()

**SYNOPSIS**

```
int mdf_message_add_time(mdf_message_t message, uint32_t tag,
                         char *value);
```

**DESCRIPTION**

Adds a time field to the current active message. Please note that all times and dates in the Millistream system are expressed in UTC. The format of **value** must be “HH:MM:SS”, “HH:MM:SS.mmm” (where mmm is the milliseconds), or "HH:MM_SS.nnnnnnnnn" (where nnnnnnnnn is the nano seconds).

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied value is not of the type specified).

**ERRORS**

None

# mdf_message_add_time2()

**SYNOPSIS**

```
int mdf_message_add_time2(mdf_message_t message, uint32_t tag,
                          int hour, int min, int sec, int msec);
```

**DESCRIPTION**

Adds a time field to the current active message. Please note that all times and dates in the Millistream system are expressed in UTC. If **msec** is set to 0 the timestamp is encoded as “HH:MM:SS”.

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied value is not of the type specified).

**ERRORS**

None

# mdf_message_add_time3()

**SYNOPSIS**

```
int mdf_message_add_time2(mdf_message_t message, uint32_t tag,
                          int hour, int min, int sec, int nsec);
```

**DESCRIPTION**

Adds a time field to the current active message. Please note that all times and dates in the Millistream system are expressed in UTC. If **nsec** is 1 – 999 the timstamp is encoded as “HH:MM:SS.mmm”. If **nsec** is set to 0 the timestamp is encoded as “HH:MM:SS”.

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied value is not of the type specified).

**ERRORS**

None

# mdf_message_add_list()

**SYNOPSIS**

```
int mdf_message_add_list(mdf_message_t message, uint32_t tag,
                         char *value);
```

**DESCRIPTION**

Adds a list field to the current active message. A list field is a space separated list of instrument references. The first position in the value can be:

'+' - the supplied list should be added to the current value

'-' - the supplied list should be removed from the current value

'=' - the supplied list is the current value

If there is no such prefix it is interpreted as if it was prefixed with a '='. There is a current soft limit of 1.000.000 instrument references per list.

**RETURN VALUE**

Returns true (1) if the field was successfully added, or false (0) if the value could not be added (because there was no more memory, the message handle does not contain any messages, or the supplied value is not of the type specified).

**ERRORS**

None

# mdf_message_send()

**SYNOPSIS**

```
int mdf_message_send(mdf_t handle, mdf_message_t message);
```

**DESCRIPTION**

Sends all the active messages in the message handle to the server. The message handle will not be reset, so this has to be performed manually.

**RETURN VALUE**

Returns true (1) if there was no errors detected when sending the data, or false (0) if an error was detected (such as not connected to any server). Due to the nature of TCP/IP, a successful return code does not guarantee that the server has received the messages.

**ERRORS**

| | |
|---|---|
| **MDF_ERR_DISCONNECTED** | The error was such that the connection with the server was disconnected |

# mdf_message_get_num()

**SYNOPSIS**

```
int mdf_message_get_num(mdf_message_t message);
```

**DESCRIPTION**

Returns the total number of messages in the message handle (the number of active + the number of reused messages currently not used for active messages).

**RETURN VALUE**

The total number of messages in the message handle.

**ERRORS**

None

# mdf_message_get_num_active()

**SYNOPSIS**

```
int mdf_message_get_num_active(mdf_message_t message);
```

**DESCRIPTION**

Returns the number of active messages in the message handle, aka the number of messages that would be sent to the server if ***mdf_message_send()*** was called.

**RETURN VALUE**

The number of active messages in the message handle.

**ERRORS**

None

# mdf_message_get_num_fields()

**SYNOPSIS**

```
int mdf_message_get_num_fields(mdf_message_t message);
```

**DESCRIPTION**

Returns the number of fields in the current active message.

**RETURN VALUE**

The number of fields in the current active message.

**ERRORS**

None

# mdf_message_move()

**SYNOPSIS**

```
int mdf_message_move(mdf_message_t src, mdf_message_t dst,
                     uint64_t insref_src, uint64_t insref_dst);
```

**DESCRIPTION**

Moves all messages from **src** with an insref matching **insref_src** to **dst** and changes the insref to **insref_dst**. If **dst** is set to the same message handle as **src** or if **dst** is **NULL**, then the change from **insref_src** to **insref_dst** will be done in-place in **src**.

If both **insref_src** and **insref_dst** is set to **`UINT64_MAX`** then the insrefs in **src** will not be changed and all the messages regardless of insref will be moved to **dst**.

**RETURN VALUE**

Returns true (1) if the operation was successful, or false (0) if it failed.

**ERRORS**

None

# mdf_message_serialize()

**SYNOPSIS**

```
int mdf_message_serialize(mdf_message_t message, char **result);
```

**DESCRIPTION**

Serializes the message chain in the message handle and produces a base64 encoded string to the address pointed to by **result**. It's the responsibility of the caller to free the produced string.

**RETURN VALUE**

Returns true (1) if there existed a message chain and if it was successfully base64 encoded, or false (0) if there existed no message chain or if the base64 encoding failed.

**ERRORS**

None

# mdf_message_deserialize()

**SYNOPSIS**

```
int mdf_message_deserialize(mdf_message_t message, char *data);
```

**DESCRIPTION**

Deserializes a base64 encoded message chain and replaces the existing (if any) message chain in the message handler.

**RETURN VALUE**

Returns true (1) if the message chain was successfully deserialized, or false (0) if the deserialization failed (if so the current message chain in the message handler is left untouched).

**ERRORS**

None

# Changelog

**2024-06-24** *Clarified that **mdf_message_send()** can be called from multiple threads as of v1.0.29 as long as each thread uses individual message handles. Added build info for macOS.*

**2024-05-15** *New function in v1.0.29: **mdf_message_add2()** to set the delay directly when adding a new message to the chain.*

**2024-05-03** *Defined the values for the various delays.*

**2022-12-28** *New options in v1.0.27: MDF_OPT_RBUF_SIZE and MDF_OPT_RBUF_MAXSIZE.*

**2022-12-21** *New function in v1.0.27: **mdf_extract()** to extract the next message from a handle.*
*New function in v1.0.27: **mdf_inject()** to inject a previously extracted message to a handle*

**2022-04-18** *As of v1.0.26 **mdf_create()** no longer disables **SIGPIPE** if the system supports **MSG_NOSIGNAL** or **SO_NOSIGPIPE**.*

**2022-04-08** *New function in v1.0.26: **mdf_message_set_property()** that replaces **mdf_message_set_compression_level()**, **mdf_message_set_utf8_validation()** and **mdf_message_set_delay()** with a generic function.*

**2022-03-31** *New option in v1.0.26: MDF_OPT_HANDLE_DELAY.*
*New function in v1.0.26: **mdf_message_set_delay()** to set the intended delay for the current and future messages.*
*New function in v1.0.26: **mdf_get_delay()** to get the intended delay of a received message.*

**2022-03-30** *New function in v1.0.26: **mdf_get_next_message2()** that works like the old **mdf_get_next_message()** only that the mref pointer is now the proper type (**uint16_t**) and the **mclass** argument is removed (can be fetched via a function instead for users that are interested in it). The old **mdf_get_next_message()** is as of now a wrapper around this new function so there are no real use case for using the old one except for supporting legacy software.*
*New function in v1.0.26: **mdf_get_mclass()** that returns the **mclass** of a received message.*

**2022-03-23** *New function in v1.0.26: **mdf_message_get_num_fields()*** to fetch the number of fields in the current active message.

**2022-01-28** *New option in v1.0.26: MDF_OPT_TIMEOUT.*
*As of v1.0.26 **mdf_message_move()** will not change the insref if both **insref_src** and **insref_dst** is set to **UINT64_MAX**;*

**2021-03-13** *New options in v1.0.25: MDF_OPT_CRYPT_DIGEST and MDF_OPT_CRYPT_CIPHER.*

**2021-03-12** *New options in v1.0.25: MDF_OPT_CRYPT_DIGESTS and MDF_OPT_CRYPT_CIPHERS.*

**2021-01-09** *New option in v1.0.24: MDF_OPT_TIME_DIFFERENCE_NS.*

**2020-06-20** *Clarified that MDF_OPT_TIME_DIFFERENCE is updated by the Heartbeat Request and Heartbeat Response from the server.*

**2019-06-20** *As of 2019-06-22 it's possible to send a "*" character to MDF_F_REQUESTCLASS or MDF_F_INSREFLIST when issuing a request or unsubscription to form "wildcard" requests, aka for all Request Classes and/or Instruments that the account is entitled for.*

**2019-02-15** *New function in v1.0.23: **mdf_message_add_string2()*** to add strings with a specified length.

**2018-12-13** *New function in v1.0.22: **mdf_message_set_compression_level()*** to set the zlib compression level used for the ***mdf_message_add_string()*** function.
As of v1.0.22 the ***mdf_consume()*** function can take a negative value in **timeout** to use a timeout period of microseconds instead of seconds.

**2018-07-30** *New option in v1.0.22: MDF_OPT_BIND_ADDRESS.*

**2017-02-23** *New functions in v1.0.20: **mdf_message_serialize()** and **mdf_message_deserialize()** to*

| | |
|---|---|
| | *serialize and deserialize a message to/from a base64 encoded string.* |
| **2017-02-21** | *New function in v1.0.20:* ***mdf_message_move()*** *to move messages from one message handle to another and/or change insrefs in the process.* |
| **2016-04-28** | *New function in v1.0.19:* ***mdf_message_add_time3()*** *for adding nanosecond timestamps to a mdf message without having to convert it to a string first.* |
| **2015-11-26** | *New function in v1.0.19:* ***mdf_message_add_date2()*** *for adding dates to a mdf message without having to convert it to a string first.* |
| **2014-12-18** | *New function in v1.0.17:* ***mdf_message_add_int()*** *for adding scaled signed integers to a mdf message without having to convert it to a string first.* |
| **2014-06-27** | *New function in v1.0.16:* ***mdf_message_add_time2()*** *for adding milliseconds timestamps to a mdf message without having to convert it to a string first.* |
| **2012-04-27** | *New function in v1.0.13:* ***mdf_message_add_uint()*** *for adding scaled unsigned integers to a mdf message without having to convert it to a string first.* |
| **2011-12-12** | *The Request message now allows multiple request classes in a single request for "condensed requests", and it is now also possible to unsubscribe to the requested realtime messages/instruments.* |
| **2011-02-01** | *The Windows binaries are now distributed as a single zip archive for both supported architectures (x86 and x64) and import library files for MinGW are also included.* |
| **2010-10-14** | *As of v1.0.9* ***mdf_consume()*** *will read data in chunks so the old semantics where the client had to implement a level triggered event handler does no longer apply.*<br>*Added info about the* ***mdf_consume()*** *function.*<br>*Added some new error codes to* ***mdf_message_send()*** *and* ***mdf_get_next_message()****.*<br>*Specified the format of the date and time formats accepted by the* ***mdf_message_add_date()*** *and* ***mdf_message_add_time()*** *functions.* |
| **2010-03-25** | *Image replies from the server have changed semantics. The new model is to no longer send empty messages if all fields are null (if the field has been deliberately set to null due to an update, the field will be added as null to the message however and a message will be sent).* |
| **2010-03-15** | *Requests can now be performed from a specified time and date.* |
| **2009-12-30** | *Win64 binaries are now available.* |
| **2009-04-09** | *Corrected how properties are modified.* |
| **2009-04-02** | *Added MDF_OPT_TCP_NODELAY property.* |
| **2009-03-24** | *Added a note that Windows Platform SDK is needed if using Visual Studio 6 to compile mdf.* |
| **2009-03-23** | *Corrected the references to the Messages and Fields reference documents.* |
| **2009-03-18** | *Initial version.* |