openTMlib
User Guide

Table of Contents

1Introduction....5
 1.1Protocol Support....5
 1.2Value Proposition....5
 1.3Limitations....5
2Installation....6
3Overview....7
4Instrument I/O API....8
 4.1Creating and Destroying I/O Sessions (Session Factory)....8
  4.1.1Creating The Session Factory....8
  4.1.2io_session *open_session(string resource, bool lock, unsigned int timeout);....8
  4.1.3void close_session(io_session *session_ptr);....9
 4.2High-level Message-based I/O....9
  4.2.1int write_string(string message, bool eol = true);....9
  4.2.2int read_string(string & message);....9
  4.2.3int write_int(int value, bool eol = true);....10
  4.2.4int read_int(int & value);....10
  4.2.5int write_binblock(char *buffer, int count);....10
  4.2.6int read_binblock(char * buffer, int max);....10
  4.2.7int query_string(string query, string & response);....10
  4.2.8int query_int(string query, int & value);....11
 4.3Low-level (Binary) I/O....11
  4.3.1int write_buffer(char *buffer, int count);....11
  4.3.2int read_buffer(char * buffer, int max);....11
 4.4Special I/O Operations....11
  4.4.1void io_operation(unsigned int operation, unsigned int value = 0);....11
  4.4.2clear()....12
  4.4.3abort()....13
  4.4.4trigger()....13
  4.4.5remote()....13
  4.4.6local()....13
  4.4.7lock()....13
  4.4.8unlock()....13
5Error Handling....14
6Session Configuration / Attributes....15
 6.1API....15
  6.1.1void set_attribute(unsigned int attribute, unsigned int value = 0)....15
  6.1.2unsigned int get_attribute(unsigned int attribute)....15
 6.2General Attributes....15
 6.3Attributes Specific to Serial Instruments....16
 6.4Attributes Specific to USB Instruments....17
 6.5Attributes Specific to VXI-11 LAN-Based Instruments....17
 6.6Attributes Specific to Direct TCP Instruments....18
7Using the Configuration Store....19
 7.1Benefits....19
 7.2Configuration Store File Structure....19

7.3Keys used by the session factory....19
7.4API....19
7.4.1load()....20
7.4.2save()....20
7.4.3string lookup(string section, string option)....20
7.4.4update(string section, string option, string value)....20
7.4.5remove(string section, string option)....20
7.5Errors....20

1 Introduction

openTMlib is a light-weight test and measurement I/O library for Linux. It facilitates the programmatic control of test instruments.

1.1 Protocol Support

openTMlib supports message-based I/O using the following protocols:

- VXI-11 and TCP (LAN-based instruments)
- USBTMC (USB instruments)
- Serial instruments

1.2 Value Proposition

The main differentiator between openTMlib and commercial alternatives is that openTMlib is free and open-source (based on the GPL). It should compile and run unchanged on most current distributions and flavors of Linux. In contrast, commercial solutions are delivered in binary format and are supported on a limited number of distributions and versions, only.

OpenTMlib offers the following features:

- A common API, independent of the I/O interface you are using. The same syntax will work for all interfaces supported.
- Centralized session management (through the session factory).
- Centralized management of instrument configuration data (through the configuration store).
- Integrated (optional) monitoring of I/O traffic for debugging purposes (through the I/O monitor).

1.3 Limitations

Limitations include the following:

- GPIB is currently not supported unless you use a LAN-to-GPIB converter which uses VXI-11 (such as the Agilent E5810A).
- Register-based I/O (e. g. for the control of PXI cards) is not supported.
- Currently, the API is C++, only.

Enhancements coming up next will likely include:

- Support for LAN-based instruments using the HISLIP protocol.
- ANSI-C API, probably in the form of a VISA subset.

2 Installation

Download the files from github (https://github.com/stefankopp/openTMlib).

After unpacking the files to a directory of your choice, run the shell script `build_and_install` to build the files. The script will also install the USBTMC kernel driver and the openTMlib shared library. For this reason, the script needs to be run with root privilege:

```
sudo ./build_and_install
```

If you use USBTMC instruments, you will need to run the script every time you reboot your machine, as the USBTMC kernel driver is installed dynamically using *insmod*. Alternatively, you can run *insmod* separately (either manually or from a start-up script).

The makefile for openTMlib currently builds, in addition to the openTMlib library itself, a simple executable, `demo_opentmlib`. See *demo_opentmlib.cpp* for details. openTMlib is statically linked to this executable for testing and debugging purposes.

In most situations, however, it will be more convenient to use the openTMlib shared library instead of static linking. To use the shared library, use `-lopentmlib` when linking your application:

```
g++ -c -o my_application.o my_application.cpp
g++ -o my_application my_application.o -lopentmlib
```

The installation script copies the openTMlib shared library to */usr/local/lib*. If you don't find the library when building your application, make sure *ldconfig* is configured to scan */usr/local/lib* (or modify the script to copy the library to */usr/lib*, instead).

3 Overview

Below is an overview of openTMlib's main components (classes).

- `io_session` is the (virtual) parent class for message-based I/O drivers. It defines a simple interface for message-based I/O, with low-level methods such as `write_buffer()` and `read_buffer()`. It also implements higher-level methods based on these low-level methods, such as `read_string()`, `write_string()`, `read_binblock()`, `read_int()` etc..
- For each supported interface/protocol, a class is derived from `io_session` which implements basic I/O for its protocol. The derived classes for message-based I/O are:
  - `vxi11_session`
  - `socket_session`
  - `usbtmc_session`
  - `serial_session`
- The session factory (`session_factory` class) allows you to create/initialize I/O sessions through a unified interface, using VISA resource strings to specify the interface, protocol and addresses. Alternatively, you can open a session directly by creating an object of one of the derived classes listed above. However, since the class constructors take different parameters (depending on the protocol's addressing scheme), you will need to modify your code if you move from one protocol to another. Using the session factory allows you to hide these differences behind a unified interface.
- The configuration store (`configuration_store` class) manages instrument configuration data. It allows you to refer to your instruments using symbolic names. The session factory, when creating a session, retrieves the instrument's resource string, as well as session configuration settings, from the configuration store. This mechanism further increases portability by keeping session configuration out of your application code.
- The I/O monitor (`io_monitor` class) is a debugging tool which facilitates the tracing/logging of I/O messages for debugging or documentation purposes.
- *usbtmc.c* is a kernel driver for access to USB-based instruments. It is required by the `usbtmc_session` class.

Typically, your starting point to the above structure is the `session_factory` class. The session factory will automatically create both the `configuration_store` and `io_monitor` objects, as well as the instrument sessions.

4 Instrument I/O API

4.1 Creating and Destroying I/O Sessions (Session Factory)

I/O sessions are typically created and destroyed using the session factory (`session factory` class). While you can create session objects directly, the session factory offers a uniform API for all session types.

4.1.1 Creating The Session Factory

The first step is creating the session factory object itself:

```
session_factory *factory;
factory = new session_factory();
```

In the above example, the default configuration store file is used, */usr/local/etc/opentmlib.store*. In order to use a different file, use its path/filename as a parameter to the session factory constructor, as shown below:

```
session_factory *factory;
factory = new session_factory(“my_store_file”);
```

Once the session factory object has been created, use its methods (described below) to open and close individual instrument sessions.

When no longer needed, discard the session factory object using:

```
delete(factory);
```

4.1.2 io_session *open_session(string resource, bool lock, unsigned int timeout);

Use `open_session()` to create (initialize) individual instrument session. Example:

```
io_session *dmm;
dmm = factory->open_session(“TCPIP0::192.168.1.2::5025::SOCKET”, false, 5);
```

resource specifies the interface and protocol used to control the instrument, as well as any addressing information required. The format is that of a standard VISA address string.

openTMlib accepts the following types of resource strings:

- Socket communication: `TCPIP0::`*IP Address*`::`*Port*`::SOCKET` (where *Port* is typically set to `5025`).
- VXI-11: `TCPIP0::`*IP Address*`::`*Logical Name*`::INSTR` (where *Logical Name* is typically set to `inst0`).
- USBTMC: `USB0::`*Manufacturer Code*`::`*Product Code*`::`*Serial Number*`::INSTR` (where *Manufacturer Code* and *Product Code* are given in hex notation). *Manufacturer Code* and *Product Code* are often hard to find. The USBTMC kernel driver logs these fields in the kernel log when a USBTMC device is attached. Use *dmesg* to inspect the kernel log after attaching the instrument.
- Serial: `ASRL`*n*`::INSTR` (where *n* indicates the COM port to be used).

resource can also be set to an instrument's symbolic name when using the configuration store to resolve that name to the actual address string.

lock specifies if the session factory should attempt to get an exclusive lock for the instrument. If set to *true*, `open_session()` will throw an appropriate error if the instrument can't be locked within the timeout periode.

timeout specifies the maximum waiting time if the instrument can't be locked immediately.

4.1.3 void close_session(io_session *session_ptr);

Use close_session() to tear down an instrument session which is no longer needed. Example:

```
factory->close_session(dmm);
```

session_ptr is a pointer to the I/O session to be closed (returned by `open_session()`).

4.2 High-level Message-based I/O

The methods described below are implemented by the `io_session` parent class. They are based on the low-level methods implemented by the individual I/O session classes (such as `socket_session`).

All of the methods return an integer value which indicates the number of characters read or written.

4.2.1 int write_string(string message, bool eol = true);

Use `write_string()` to send an instrument command which is held in a C++ string data type. Example:

```
string command = “*RST”;
session->write_string(command); // EOL character will be added by default
```

`write_string()` is often more convenient than the low-level `write_buffer()` method because the number of characters to be sent does not need to be specified explicitly – the actual string length is used.

The optional *eol* parameter specifies if the EOL character (typically new-line) is appended to the string. Set *eol* to true (or omit the parameter) if this transaction ends the command string. Set *eol* to *false* if this transaction is followed by other transactions which completes the instrument command.

4.2.2 int read_string(string & message);

Use `read_string()` to read an instrument response into a C++ string variable. Example:

```
sesion->write_string(“*IDN?”); // EOL character added by default
string instrument_id;
session->read_string(instrument_id);
```

`read_string()` is often more convenient than the low-level `read_buffer()` method because the maximum number of characters to be read is determined automatically.

The maximum number of characters is determined as follows:

- If the current size of *message* is smaller than the standard string size (see attribute *OPENTMLIB_ATTRIBUTE_STRING_SIZE*), *message* is resized to the standard string size. In most situations, this allows you to use string variables without checking their size and/or resizing the strings yourself.

- If the current size of *message* is equal to or larger than the standard string size, the current size will be used. This allows you to read large instrument response strings by resizing *message* before calling `read_string()`.

4.2.3 int write_int(int value, bool eol = true);

Use `write_int()` to send an integer value to the instrument (converted to decimal text form). Example:

```
int frequency = 45000;
session->write_string("CONF:FREQ ", false);  // No EOL after this piece
session->write_int(frequency); // Now add EOL character
```

The optional *eol* parameter specifies if the EOL character (typically new-line) is appended to the message. Set *eol* to *true* (or omit the parameter) if this transaction ends the command string. Set *eol* to *false* if this transaction if followed by other transactions which completes the instrument command.

4.2.4 int read_int(int & value);

Use `read_int()` to read an instrument response (in decimal text form) and convert it to an integer value. Example:

```
int frequency;
session->write_string("CONF:FREQ?"); // EOL character appended by default
session->read_int(frequency);
cout << "Frequency is " << frequency << endl;
```

4.2.5 int write_binblock(char *buffer, int count);

Use `write_binblock()` to write binary data to the instrument in the form of an IEEE488 arbitrary length binary block (binblock).

`write_binblock()` adds (prepends) the binblock header to the data sent. *buffer* should point to the raw data (without header).

4.2.6 int read_binblock(char * buffer, int max);

Use `read_binblock()` to read an IEEE488 arbitrary length binary block (binblock) from the instrument.

`read_binblock()` reads the complete binblock, the length of which is indicated in the binblock header information. The raw data (without header) is copied to *buffer*. The method's return value indicates the number of bytes copied.

If the binblock's actual size is larger than the buffer provided (as indicated by *max*), the method will throw an error with *code* set to *-OPENTMLIB_ERROR_BINBLOCK_SIZE*.

4.2.7 int query_string(string query, string & response);

`query_string()` is a convenience method which combines `write_string()` and `read_string()` in a single method. Note that the output string is sent with *eol* set to true (see above), i. e. an end-of-line character (typically new-line) will be added. Example:

```
string id_string;
session->query_string("*IDN?", id_string);
cout << "ID is " << id_string << endl;
```

4.2.8 int query_int(string query, int & value);

`query_int()` is a convenience method which combines `write_string()` and `read_int()` in a single method. Note that the output string is sent with *eol* set to true (see above), i. e. an end-of-line character (typically new-line) will be added. Example:

```
int frequency;
session->query_int("CONF:FREQ?", frequency);
cout << "Frequency is " << frequency << endl;
```

4.3 Low-level (Binary) I/O

The methods described below are defined by the `io_session` parent class but implemented by the various derived classes (such as `socket_session`).

For text commands, using the high-level methods (see above) is typically easier and more convenient. Use these methods for sending and reading binary data.

4.3.1 int write_buffer(char *buffer, int count);

Use `write_buffer()` to write binary data to the instrument. *count* bytes are written from *buffer*.

4.3.2 int read_buffer(char * buffer, int max);

Use `read_buffer()` to read binary data from the instrument. Up to *max* bytes are read to *buffer*. The method returns the actual number of bytes read.

4.4 Special I/O Operations

The methods described below are available for special I/O operations such as *Device Clear*.

Not all operations are supported by all drivers. If you attempt to perform an unsupported operation, the driver session will throw an `opentmlib_exception` error object with *code* set to -*OPENTMLIB_ERROR_BAD_OPERATION*.

4.4.1 void io_operation(unsigned int operation, unsigned int value = 0);

Use `io_operation()` to perform any of the special I/O operations supported by the driver and session type. For example, the below line performs a *Device Clear* operation:

```
session->io_operation(OPENTMLIB_OPERATION_CLEAR);
```

Most I/O operations do not use the *value* parameter, so you can omit it (i. e. use its default value of 0).

The following general I/O operations are defined:

- *USBTMLIB_OPERATION_TRIGGER*: Triggers the instrument. Supported with IEEE 488-like protocols, such as VXI-11.

- *USBTMLIB_OPERATION_CLEAR*: Clears the instrument's I/O buffers. Supported with most protocols.
- *USBTMLIB_OPERATION_REMOTE*: Sets the instrument to remote state (front panel disabled). Supported with IEEE 488-like protocols, such as VXI-11.
- *USBTMLIB_OPERATION_LOCAL*: Sets the instrument to local state (front panel active). Supported with IEEE 488-like protocols, such as VXI-11.
- *USBTMLIB_OPERATION_LOCK*: Tries to lock the instrument (for exclusive access). Supported with IEEE 488-like protocols, such as VXI-11.
- *USBTMLIB_OPERATION_UNLOCK*: Returns the lock. Supported with IEEE 488-like protocols, such as VXI-11.
- *USBTMLIB_OPERATION_ENABLE_SRQ*: Enables service requests. Supported with IEEE 488-like protocols, such as VXI-11.
- *USBTMLIB_OPERATION_ABORT*: Aborts a pending operation. Supported with most protocols.
- *USBTMLIB_OPERATION_INDICATOR_PULSE*: Asks the instrument to pulse the activity indicator (for identification purposes). Supported with IEEE 488-like protocols, such as VXI-11.

The following USBTMC-specific operations are defined:

- *USBTMLIB_OPERATION_USBTMC_ABORT_WRITE*: Aborts the last write transaction.
- *USBTMLIB_OPERATION_USBTMC_ABORT_READ*: Aborts the last read transaction.
- *USBTMLIB_OPERATION_USBTMC_CLEAR_OUT_HALT*: Clears a halt state on the USB BULK OUT end point.
- *USBTMLIB_OPERATION_USBTMC_CLEAR_IN_HALT*: Clears a halt state on the USB BULK IN end point.
- *USBTMLIB_OPERATION_USBTMC_RESET*: Resets the USB configuration (not the instrument itself).
- *USBTMLIB_OPERATION_USBTMC_REN_CONTROL*: Sets remote enable state (see USBTMC specification for details).
- *USBTMLIB_OPERATION_USBTMC_GO_TO_LOCAL*: Sends *GO_TO_LOCAL* request (see USBTMC specification for details).
- *USBTMLIB_OPERATION_USBTMC_LOCAL_LOCKOUT*: Sends *LOCAL_LOCKOUT* request (see USBTMC specification for details).

4.4.2 clear()

Convenience method equivalent to `io_operation(OPENTMLIB_OPERATION_CLEAR)`. See above.

4.4.3 abort()

Convenience method equivalent to `io_operation(USBTMLIB_OPERATION_ABORT)`. See above.

4.4.4 trigger()

Convenience method equivalent to `io_operation(USBTMLIB_OPERATION_TRIGGER)`. See above.

4.4.5 remote()

Convenience method equivalent to `io_operation(USBTMLIB_OPERATION_REMOTE)`. See above.

4.4.6 local()

Convenience method equivalent to `io_operation(USBTMLIB_OPERATION_LOCAL)`. See above.

4.4.7 lock()

Convenience method equivalent to `io_operation(USBTMLIB_OPERATION_LOCK)`. See above.

4.4.8 unlock()

Convenience method equivalent to `io_operation(USBTMLIB_OPERATION_UNLOCK)`. See above.

5 Error Handling

openTMlib errors are handled by throwing an `opentmlib_exception` error object. `opentmlib_exception` is derived from `std::runtime_error`. In addition to the `what()` member function which returns the error message, the class also features a `code` member variable which holds an error code.

openTMlib error codes are in the -0x8000 range. See *opentmlib.h* for details. Note, however, that not all error codes will be in the mentioned range as the drivers will also create error objects based on error codes (*errno*) returned from library functions such as `open()` and `read()`.

Below is a basic example demonstrating how to access error information in a catch block:

```
try
{
   dmm_session = factory->open_session("dmm");
}
catch (opentmlib_exception & e)
{
   cout << "Error message: " << e.what() << endl;
   cout << "Error code: " << e.code << endl;
}
```

If you don't handle (catch) the errors yourself, the system will print the error message, as shown below:

```
terminate called after throwing an instance of 'opentmlib_exception'
  what():  Bad instrument resource (address) string
Aborted
```

6 Session Configuration / Attributes

openTMlib's I/O sessions (all classes deriving from `io_session`) allow you to set/read driver configuration parameters through a unified interface, using attributes.

6.1 API

The `io_session` parent class defines the following methods for access to driver attributes.

6.1.1 void set_attribute(unsigned int attribute, unsigned int value = 0)

Use `set_attribute()` to set an attribute to the given value. For example, the below line sets the timeout value for an instrument session to 10 seconds:

```
session->set_attribute(OPENTMLIB_ATTRIBUTE_TIMEOUT, 10);
```

Not all attributes are known to and supported by all drivers. If you attempt to read or set an attribute which is not supported, an error object will be thrown with an error code of *-OPENTMLIB_ERROR_BAD_ATTRIBUTE*.

If you attempt set set an attribute to an illegal value, an error object will be thrown with an error code of *-OPENTMLIB_ERROR_BAD_ATTRIBUTE_VALUE*.

6.1.2 unsigned int get_attribute(unsigned int attribute)

Use `get_attribute()` to read the current state of an attribute. For example, the below lines read the current timeout value:

```
unsigned int timeout;
timeout = session->get_attribute(OPENTMLIB_ATTRIBUTE_TIMEOUT);
```

6.2 General Attributes

The general attributes listed below are used by several or all drivers:

- *OPENTMLIB_ATTRIBUTE_TIMEOUT*
  Timeout value in seconds.
  Corresponding configuration store key: `timeout`.
  Default value applied by session factory: 5.
- *OPENTMLIB_ATTRIBUTE_TERM_CHAR_ENABLE*
  Specifies if a read transaction terminates automatically when the termination character (see below) is encountered in the input stream.
  Corresponding configuration store key: `term_char_enable`.
  Default value applied by session factory: 1 (ON).
- *OPENTMLIB_ATTRIBUTE_TERM_CHARACTER*
  Termination character (ASCII code). If *OPENTMLIB_ATTRIBUTE_TERM_CHAR_ENABLE* is set to 1, a read transaction terminates automatically when this characters is encountered in the input stream.
  Corresponding configuration store key: `term_char`.
  Default value applied by session factory: 10 (new-line character).

- *OPENTMLIB_ATTRIBUTE_EOL_CHAR*
  End-of-line character. This character is appended to outgoing instrument commands when using `write_string()` or other higher-level methods based on `write_string()`. It is used to signal the end of the command message.
  Corresponding configuration store key: `eol_char`.
  Default value applied by session factory: 10 (newline character).
- *OPENTMLIB_ATTRIBUTE_STATUS_BYTE*
  Returns the instrument's actual status byte value (read-only). Supported by IEEE 488-like protocols such as VXI-11.
- *OPENTMLIB_ATTRIBUTE_SET_END_INDICATOR*
  Defines if the end indicator is set with the last byte of an output transaction. Supported by IEEE 488-like protocols such as VXI-11.
  Corresponding configuration store key: `set_end_indicator`.
  Default value applied by session factory: 0 (OFF).
- *OPENTMLIB_ATTRIBUTE_WAIT_LOCK*
  Defines the driver behavior when trying to acquire a lock, if the instrument can't be locked immediately. When set to 1, the driver will wait for the lock (until a timeout occurs). When set to 0, the driver will return immediately with an error. Supported by IEEE 488-like protocols such as VXI-11.
  Default value applied by session factory: None (use `set_attribute()`).
- *OPENTMLIB_ATTRIBUTE_STRING_SIZE*
  Defines the minimum string size (in characters) used for `read_string()`. If the size of the string passed to `read_string()` is smaller, the string is resized to this value. This mechanism allows you to call `read_string()` without taking care of resizing strings yourself.
  Default value applied during session creation: 200.
- *OPENTMLIB_ATTRIBUTE_ERROR_ON_SCPI_ERROR*
  This attribute is relevant for `scpi_check_errors()`. `scpi_check_errors()` reads errors from the instrument error queue using the `SYSTEM:ERROR?` command. The attribute specifies if an error is thrown if the instrument returns an error. If set to 0 (OFF), `scpi_check_errors()` only throws an error if it is unable to clear the error queue with the number of queries specified.
  Default value applied during session creation: 1 (ON).
- *OPENTMLIB_ATTRIBUTE_TRACING*
  This attribute is relevant for tracing. If set to 1 (ON), communication for this instrument is sent to the I/O monitor for tracing. The I/O monitor dumps messages to a tracing file for later analysis for debugging purposes.
  Corresponding configuration store key: `tracing`.
  Default value applied by session factory: 0 (OFF).

6.3 Attributes Specific to Serial Instruments

The attributes listed below are specific to serial instruments:

- *OPENTMLIB_ATTRIBUTE_SERIAL_BAUDRATE*

Baud rate (bits per second) used by the instrument (50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600 or 115200).
Default value applied by session factory: None (use `set_attribute()`).

- *OPENTMLIB_ATTRIBUTE_SERIAL_SIZE*
  Character size (bits) used (5, 6, 7 or 8).
  Default value applied by session factory: None (use `set_attribute()`).
- *OPENTMLIB_ATTRIBUTE_SERIAL_PARITY*
  Specifies if parity bit is used. Choices are:
  OPENTMLIB_SERIAL_PARITY_NONE: no parity,
  OPENTMLIB_SERIAL_PARITY_EVEN: even parity,
  OPENTMLIB_SERIAL_PARITY_ODD: odd parity.
  Default value applied by session factory: None (use `set_attribute()`).
- *OPENTMLIB_ATTRIBUTE_SERIAL_STOPBITS*
  Specifies how many stop bits are used (1 or 2).
  Default value applied by session factory: None (use `set_attribute()`).
- *OPENTMLIB_ATTRIBUTE_SERIAL_RTSCTS*
  Specifies if RTS/CTS (hardware) flow control is used (1 = ON, 0 = OFF).
  Default value applied by session factory: None (use `set_attribute()`).
- *OPENTMLIB_ATTRIBUTE_SERIAL_XONXOFF*
  Specifies if XON/XOFF (software) flow control is used (1 = ON, 0 = OFF).
  Default value applied by session factory: None (use `set_attribute()`).

6.4 Attributes Specific to USB Instruments

The attributes listed below are specific to USBTMC instruments:

- *OPENTMLIB_ATTRIBUTE_USBTMC_INTERFACE_CAPS*
  Returns the USBTMC interface capabilities value.
  See section 4.2.1.8 of the USBTMC specification for details.
- *OPENTMLIB_ATTRIBUTE_USBTMC_DEVICE_CAPS*
  Returns the USBTMC device capabilities value.
  See section 4.2.1.8 of the USBTMC specification for details.
- *OPENTMLIB_ATTRIBUTE_USBTMC_488_INTERFACE_CAPS*
  Returns the USBTMC-USB488 interface capabilities value. Only supported for devices confirming to the USB488 sub class.
  See section 4.2.2 of the USBTMC-USB488 specification for details.
- *OPENTMLIB_ATTRIBUTE_USBTMC_488_DEVICE_CAPS*
  Returns the USBTMC-USB488 device capabilities value. Only supported for devices confirming to the USB488 sub class.
  See section 4.2.2 of the USBTMC-USB488 specification for details.

6.5 Attributes Specific to VXI-11 LAN-Based Instruments

The attributes listed below are specific to VXI-11 instruments:

- *OPENTMLIB_ATTRIBUTE_VXI11_MAXRECVSIZE*
  Returns the maximum command message size the instrument is prepared to receive. This value is returned by the instrument when creating the VXI-11 link. It is for information, only – the driver will split larger transactions, automatically.
- *OPENTMLIB_ATTRIBUTE_VXI11_LAST_ERROR*
  Returns the error code of the most recent failed VXI-11 transaction. This value is for information and/or debugging purposes, only. The driver converts VXI-11 error codes to corresponding openTMlib errors. However, the value might be of interest in rare cases where VXI-11 returns an unlisted error code in which case openTMlib will throw a generic I/O error.

6.6 Attributes Specific to Direct TCP Instruments

The attributes listed below are specific to instruments using basic TCP socket communication:

- *OPENTMLIB_ATTRIBUTE_SOCKET_BUFFER_SIZE*
  Returns the size of the local buffer allocated by the driver. When using termination character handling, the driver will buffer incoming data until the termination character is received. Transaction size is therefore limited by the size of the local buffer. If you have a specific requirement with regards to buffer size, you can verify proper driver configuration by reading the attribute. The attribute is read-only – the buffer size is currently set statically through a define in `socket_session`.

7 Using the Configuration Store

The configuration store (`configuration_store` class) is used to store/retrieve instrument configuration data. It is mainly used by the session factory in order to a) resolve symbolic instrument names to VISA resource strings, and b) configure the I/O session (timeout value etc.).

7.1 Benefits

Using the configuration store provides the following benefits:

- Symbolic names for your instruments allow you to hide the I/O interface and addressing details from the application.
- When a session is created by the session factory, it is automatically configured using the settings found for the instrument (identified by its symbolic name). This allows you (when used properly) to immediately communicate with the instrument without bothering about basic settings.

7.2 Configuration Store File Structure

The configuration store's structure is that of a classic INI file:

- Configuration settings are grouped in sections. The section name corresponds to the instrument's symbolic name.
- Each section includes an arbitrary number of configuration settings. Each setting is a key/value pair.
- The file format used is very simple, with section names enclosed in brackets, and key/value pairs separated by a space character. You can edit the file manually, but it is easier and safer (in case of changes to the structure) to use the configuration store methods to add information.
- The store's default location is */usr/local/etc/opentmlib.store*.

7.3 Keys used by the session factory

When opening a session using a symbolic instrument name, the session factory will look for a number of standard key/value pairs in the section corresponding to the instrument. The keys looked for basically correspond to a number of basic attributes which the session factory applies when creating the session.

See the section *Session Configuration Using Attributes* for details.

You are free to store and retrieve other key/value pairs (in addition to those used by the session factory) to track additional settings.

7.4 API

In most cases, you will not use the configuration_store class directly. It will be used by the session factory when creating instrument sessions. However, if you do want to use the class directly, you can a) create your own configuration store object or b) get the object pointer from the session factory.

The below lines shows how to get access to the configuration store through the object created by the session factory:

```
configuration_store *store;
store = factory->get_store();
```

7.4.1 load()

Use `load()` to reload the contents of the configuration store file. The content is cached in the `configuration_store` object, so a (manual) change to the store file will go unnoticed unless you reload the contents.

7.4.2 save()

Use `save()` to write the current contents of the configuration store to the store file. Changes done to the internal cache will not be saved until you call `save()`.

7.4.3 string lookup(string section, string option)

Use `lookup()` to retrieve individual configuration settings from the store. The method will look for the given option (key) in the given section (which is usually the symbolic name of the instrument). It returns the value found or an empty string of the section or key was not found.

7.4.4 update(string section, string option, string value)

Use `update()` to update an existing option or add a new section/option.

7.4.5 remove(string section, string option)

Use `remove()` to remove either a complete section (when called with option being an empty string) or an individ`ual op`tion in an existing section (when called with option being set to the name of the option to be removed).

7.5 Errors

In case of errors, a `usbtmc_exception` error object will be thrown with `code` set to one of the below errors.

When referring to an alias which does not exist:

-OPENTMLIB_ERROR_CSTORE_BAD_ALIAS

When specifying an illegal value string:

-OPENTMLIB_ERROR_CSTORE_BAD_VALUE

When loading a configuration store file which exceeds the maximum file size:

-OPENTMLIB_ERROR_CSTORE_FILE_SIZE

When referring to a section name which does not exist:

-OPENTMLIB_ERROR_CSTORE_BAD_SECTION

When referring to an option name which does not exist:

-OPENTMLIB_ERROR_CSTORE_BAD_OPTION