openTMlib User Guide

Introduction

openTMlib is a small, lightweight, test and measurement I/O library. It currently supports message-based I/O through LAN (VXI-11 and direct socket connections), USB and serial interfaces.

Unlike commercial VISA libraries, this library is open-source. As a result, it can be compiled for and should run on all recent versions and flavors of Linux.

openTMlib is based on C++. A C API or bindings for other languages are currently not available.

However, like a commercial VISA, openTMlib offers the benefit of a standardized API, independent of the I/O interface you are using. In other words, the exact same syntax will work for LAN, USB and serial, as long as you are not using unique features of one of these interfaces.

Structure

Here's an overview of the structure of the library.

- **io_session** is the parent class for message-based I/O. It defines a simple interface for message-based I/O, with methods such as write_buffer() and read_buffer. It also includes higher-level methods based on these low-level functions, such as read_string() and write_string.
- For each interface/protocol, a class is derived from io_session which implements basic I/O for its particular interface. The **classes derived from io_session** are:
  - vxi11_session
  - socket_session
  - usbtmc_session
  - serial_session
- **session_factory** is a class which allows you to generate I/O sessions through a unified interface, using VISA resource strings in order to specify the interface, protocol, addresses, and so on. Of course, you can open a session directly by creating, for example, an object of type usbtmc_session. However, since these objects take different parameters, you will need to modify your code if you move from (for example) LAN to USB. The session factory hides these differences.
- **configuration_store** is a class which reads instrument configuration parameters from a store (simple text file). It is used by the session factory to resolve logical instrument names to a VISA resource string which can then be processed by the session factory. It is also used to keep configuration details (such as timeout settings) out of your source if desired.
- **usbtmc.c** is a kernel driver for access to USB-based instruments (compatible with the USBTMC standard).

Installation

Run the shell script named build_and_install to build the files. The script will also build and install the USBTMC kernel driver. For this reason, the script needs to be run with root privilege:

```
sudo ./build_and_install
```

The makefile for openTMlib currently build a simple executable named demo_opentmlib. See

demo_opentmlib.cpp for details. Replace this file with your test application and tune the makefile accordingly. In a future version, the makefile will be modified to generate shared libraries.

Basic Usage

Usage of the library typically starts with opening the session factory:

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

In the above example, no configuration store is used. If you want to use a configuration store, use its path/filename as a parameter to the session_factory constructor, as shown in the following example:

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

Next, the session factory is used to open an I/O session to a particular instrument. An example would be:

```
io_session *my_session;
my_session = factory->open_session("TCPIP0::169.254.2.20::inst0::INSTR", false, 5);
```

In the above example, no configuration store is used: the instrument address string is specified directly. When using a configuration store, you can replace the address string with the instrument's symbolic name, such as:

```
io_session *my_session;
my_session = factory->open_session("func_gen", false, 5);
```

The configuration store is then used to resolve the symbolic name to a VISA address string which the session factory can process. The configuration store can also include additional configuration settings which are applied automatically after resolving the symbolic name.

The next step is using the session object in order to communicate with the instrument. For example:

```
my_session->write_string("*IDN?", true);
my_session->read_string(response);
```

When done, discard the objects via:

```
delete my_session;
delete factory;
```

Error Handling

openTMlib errors are handled by throwing an error message. The error codes are in the 0x8000 range. See opentmlib.h for details.

Note that not all numbers thrown will be in the above mentioned range as the drivers will also throw numbers returned from library functions such as open() and read().

opentmlib.cpp includes a routine which returns an error string when given an error code. Example:

```
string error_message;
error_message.resize(120);
opentmlib_error(error_code, error_message);
cout  << "Error: " << error_message << endl;
```

Attributes

The drivers allow you to set/read configuration parameters. Example:

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

Same attribute, read from the driver:

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

The following general attributes are defined:

- OPENTMLIB_ATTRIBUTE_TIMEOUT: timeout value (s). Default: 5.
- OPENTMLIB_ATTRIBUTE_TERM_CHAR_ENABLE: defines if a read transaction terminates automatically when the termination character is encountered in the input stream. Default: 1 (ON).
- OPENTMLIB_ATTRIBUTE_TERM_CHARACTER: termination character. Default: \n.
- OPENTMLIB_ATTRIBUTE_STATUS_BYTE: instrument status byte (read-only).
- OPENTMLIB_ATTRIBUTE_SET_END_INDICATOR: defines if the end indicator is set with the last byte of an output transaction. Supported by VXI-11, only.
- OPENTMLIB_ATTRIBUTE_EOL_CHAR: character used to signal the end of a command string. Default: \n.
- OPENTMLIB_ATTRIBUTE_WAIT_LOCK: defines the behavior if the instrument can't be locked immediately. 1 = wait for lock, 0 = return immediately with an error. Supported by VXI-11, only.

In addition to these general attributes, there are additional attributes in order to support the specific features of a given interface. See opentmlib.h for details.

I/O Operations

In addition to “normal” instrument I/O, special I/O operation can be performed using, for example:

```
session->io_operation(OPENTMLIB_OPERATION_CLEAR, 0);
```

io_session also defined a number of convenience functions which wrap these calls into more readable methods such as:

```
session->clear();
```

The following I/O operations are defined:

- USBTMLIB_OPERATION_TRIGGER: Triggers the instrument.
- USBTMLIB_OPERATION_CLEAR: Clears the instrument's I/O buffers.

- USBTMLIB_OPERATION_REMOTE: Sets the instrument to remote state (front panel disabled).
- USBTMLIB_OPERATION_LOCAL: Sets the instrument to local state (front panel active).
- USBTMLIB_OPERATION_LOCK: Tries to lock the instrument (for exclusive access).
- USBTMLIB_OPERATION_UNLOCK: Returns the lock.
- USBTMLIB_OPERATION_ENABLE_SRQ: Enables service requests.
- USBTMLIB_OPERATION_ABORT: Aborts a pending operation.
- USBTMLIB_OPERATION_INDICATOR_PULSE: Asks the instrument to pulse the activity indicator (for identification purposes).

Additional I/O operations are defined in order to exercise functions specific to a certain type of I/O interface. See opentmlib.h for details.

Note that the general operations listed above are the more common ones, but they are not supported by each and every type of interface. For exampe, a simple TCP/socket connection has no provision for 488.2-like operations such as SRQs. If the operation is not supported, the drivers will return error code OPENTMLIB_ERROR_BAD_OPERATION.

This is it, folks! This documentation is kind of short, so when in doubt, consult the source code!