LibSerial Documentation

Release 1.0.0

LibSerial Development Team

Nov 10, 2018

Contents:

1 Feature Summary **3**

2 Description **5**

3 Download **7**

4 Install **9**

5 Tutorial **11**
- 5.1 Opening a Serial Port I/O Stream . . . 11
- 5.2 Setting the Baud Rate . . . 12
- 5.3 Setting the Character Size . . . 12
- 5.4 Setting the Flow-Control Type . . . 12
- 5.5 Setting the Parity Type . . . 12
- 5.6 Setting the Number of Stop Bits . . . 12
- 5.7 Reading Characters . . . 12
- 5.8 Writing Characters . . . 13
- 5.9 Reading Blocks of Data . . . 13
- 5.10 Writing Blocks of Data . . . 14
- 5.11 Closing the Serial Port . . . 14

6 API Documentation **15**

7 Design Documentation **17**
- 7.1 LibSerial's Coding standards . . . 17
- 7.2 Naming Convention . . . 18
- 7.3 Indentation . . . 18
- 7.4 Include Headers . . . 19

8 Links **21**

9 Indices and tables **23**

Contents:

CHAPTER 1

Feature Summary

- Simplified serial port programming in C++ under POSIX operating systems.
- Support for USB-serial converters.
- Access serial ports from scripting languages such as PHP, Python, Perl, Ruby, and Java.

CHAPTER 2

Description

LibSerial was created to simplify serial port programming on POSIX systems through a collection of object oriented C++ classes.

The *SerialPort* class allows simplified access to serial port settings and usage through a convenient set of methods. This class is useful for embedded systems where a complete C++ STL may not be available.

The *SerialStream* class allows access to serial ports in the same manner as standard C++ iostream objects.

Methods are provided for setting serial port parameters such as baud rate, character size, flow control, etc.

Here is short example using libserial:

```
#include <libserial/SerialPort.h>
#include <libserial/SerialStream.h>

using namespace LibSerial;

int main()
{
   // Instantiate a Serial Port and a Serial Stream object.
   SerialPort serial_port;
   SerialStream serial_stream;

   // Open the hardware serial ports.
   serial_port.Open( "/dev/ttyUSB0" );
   serial_stream.Open( "/dev/ttyUSB1" );

   // Set the baud rates.
   serial_port.SetBaudRate( BaudRate::BAUD_115200 );
   serial_stream.SetBaudRate( BaudRate::BAUD_115200 );

   char write_byte_1 = 'a';
   char write_byte_2 = 'b';

   char read_byte_1 = 'A';
   char read_byte_2 = 'B';
```

(continues on next page)

(continued from previous page)

```
    // Write a character.
    serial_port.Write(&write_byte_1, 1);
    serial_stream << write_byte_2;

    // Read a character.
    serial_port.Read(read_byte_1, 1);
    serial_stream >> read_byte_2;

    std::cout << "serial_port read:   " << read_byte_1 << std::endl;
    std::cout << "serial_stream read: " << read_byte_2 << std::endl;

    // Close the Serial Port and Serial Stream.
    serial_port.Close();
    serial_stream.Close();
}
```

In addition to the C++ programming languge, LibSerial releases after version 0.6.0 also provide bindings to several scripting languages such as Python, Perl, PHP, Java, and Ruby. This provides developers a wide range languages to select when writing applications that need access to serial ports on POSIX compatible operating systems. LibSerial has received the most extensive testing on (Debian) Linux operating systems.

CHAPTER 3

Download

The latest version of LibSerial is 1.0.0. You can find the source code for LibSerial-1.0.0 here. Older versions of LibSerial may also be found at the above site.

CHAPTER 4

Install

To install LibSerial the current release package on many Linux distributions you may simply use the package manager associated with your distribution:

For Debian distrbutions:

```
sudo apt install libserial-dev
```

For Arch Linux distributions:

```
sudo pacman -S libserial-dev
```

To install LibSerial from source, first clone the repository at https://github.com/crayzeewulf/libserial

Using https:

```
git clone https://github.com/crayzeewulf/libserial.git
```

Using ssh:

```
git clone git@github.com:crayzeewulf/libserial.git
```

Next, using make, execute the following commands from your libserial directory:

```
make -F Makefile.dist
./configure
make
```

To install the build to your /usr/local/ directory your may simply:

```
sudo make install
```

To install to another directory, simply use the *prefix* argument in the configure step above:

```
./configure --prefix=<DIRECTORY_NAME>
```

The code is also easily built using *CMake* via a bash script:

```
./compile.sh
```

To install, change directories to the build directory and proceed as with make:

```
cd build/
sudo make install
```

CHAPTER 5

Tutorial

5.1 Opening a Serial Port I/O Stream

A serial port instance, SerialPort, or an I/O stream instance, SerialStream, can be created and opened by providing the name of the serial port device to the constructor:

```
#include <SerialPort.h>
#include <SerialStream.h>

using namespace LibSerial ;

// Create and open the serial port for communication.
SerialPort   my_serial_port( "/dev/ttyS0" );
SerialStream my_serial_stream( "/dev/ttyUSB0" ) ;
```

In certain applications, the name of the serial port device may not be known when the SerialStream instance is created. In such cases, the same effect as above can be achieved as follows:

```
// Create a object instance.
SerialPort   my_serial_port;
SerialStream my_serial_stream;

// Obtain the serial port name from user input.
std::cout << "Please enter the name of the serial device, (e.g. /dev/ttyUSB0): " <<␣
↪std::flush;
std::string serial_port_name;
std::cin >> serial_port_name;

// Open the serial port for communication.
my_serial_port.Open( serial_port_name );
my_serial_stream.Open( serial_port_name );
```

5.2 Setting the Baud Rate

The baud rate for the SerialStream can be set using the SerialStream::SetBaudRate() member function.

```
// Set the desired baud rate using a SetBaudRate() method call.
// Available baud rate values are defined in SerialStreamConstants.h.

my_serial_port.SetBaudRate( BAUD_115200 );
my_serial_stream.SetBaudRate( BAUD_115200 );
```

5.3 Setting the Character Size

```
// Set the desired character size using a SetCharacterSize() method call.
// Available character size values are defined in SerialStreamConstants.h.
my_serial_port.SetCharacterSize( CHAR_SIZE_8 );
my_serial_stream.SetCharacterSize( CHAR_SIZE_8 );
```

5.4 Setting the Flow-Control Type

```
// Set the desired flow control type using a SetFlowControl() method call.
// Available flow control types are defined in SerialStreamConstants.h.
my_serial_port.SetFlowControl( FLOW_CONTROL_HARD );
my_serial_stream.SetFlowControl( FLOW_CONTROL_HARD );
```

5.5 Setting the Parity Type

```
// Set the desired parity type using a SetParity() method call.
// Available parity types are defined in SerialStreamConstants.h.
my_serial_port.SetParity( PARITY_ODD );
my_serial_stream.SetParity( PARITY_ODD );
```

5.6 Setting the Number of Stop Bits

```
// Set the number of stop bits using a SetNumOfStopBits() method call.
// Available stop bit values are defined in SerialStreamConstants.h.
my_serial_port.SetNumOfStopBits( STOP_BITS_1 ) ;
my_serial_stream.SetNumOfStopBits( STOP_BITS_1 ) ;
```

5.7 Reading Characters

Characters can be read from serial port instances using Read(), ReadByte(), and Readline() methods. For example:

```
// Read one character from the serial port within the timeout allowed.
int timeout_ms = 25; // timeout value in milliseconds
char next_char;      // variable to store the read result

my_serial_port.ReadByte( next_char, timeout_ms );
my_serial_stream.read( next_char );
```

Characters can be read from serial streams using standard iostream operators. For example:

```
// Read one character from the serial port.
char next_char;
my_serial_stream >> next_char;

// You can also read other types of values from the serial port in a similar fashion.
int data_size;
my_serial_stream >> data_size;
```

Other methods of standard C++ iostream objects could be used as well. For example, one can read characters from the serial stream using the get() method:

```
// Read one byte from the serial port.
char next_byte;
my_serial_stream.get( next_byte );
```

5.8 Writing Characters

```
// Write a single character to the serial port.
my_serial_port.WriteByte( 'U' );
my_serial_stream << 'U' ;

// You can easily write strings.
std::string my_string = "Hello, Serial Port."

my_serial_port.Write( my_string );
my_serial_stream << my_string << std::endl ;

// And, with serial stream objects, you can easily write any type
// of object that is supported by a "<<" operator.
double radius = 2.0 ;
double area = M_PI * 2.0 * 2.0 ;

my_serial_stream << area << std::endl ;
```

5.9 Reading Blocks of Data

```
// Read a whole array of data from the serial port.
const int BUFFER_SIZE = 256;
char input_buffer[BUFFER_SIZE];

my_serial_port.Read( input_buffer, BUFFER_SIZE );
my_serial_stream.read( input_buffer, BUFFER_SIZE );
```

5.10 Writing Blocks of Data

```
// Write an array of data from the serial port.
const int BUFFER_SIZE = 256;
char output_buffer[BUFFER_SIZE];

for( int i=0; i<BUFFER_SIZE; ++i )
{
    output_buffer[i] = i;
}

my_serial_port.Write( output_buffer, BUFFER_SIZE );
my_serial_stream.write( output_buffer, BUFFER_SIZE );
```

5.11 Closing the Serial Port

```
my_serial_port.Close();
my_serial_stream.Close();
```

CHAPTER 6

API Documentation

The API documentation generated by doxygen is available here.

To generate a new set of docs using Sphinx, simply run:

```
sphinx-build -b html docs/user_manual/ docs/html/
```

The Sphinx output will be located in the */libserial/docs/html/* directory.

To generate a new set of doxygen documentation, you can run the compile.sh script or simply invoke doxygen directly:

```
doxygen doxygen.conf.in
```

The doxygen output will be located in the */libserial/build/docs/html/* directory.

CHAPTER 7

Design Documentation

7.1 LibSerial's Coding standards

Try to utilize these guidelines if you are contributing the LibSerial as a developer. Although we attempt to maintain these standards wherever practical, on occasion you might still discover a few deviations.

Please familiarize yourselves with C++ Core Guidelines and try to follow these guidelines.

LibSerial uses ISO standard C++ based on the C++14 standard.

Use Doxygen style comments (with @ commands) for every:

- Class
- Data Member
- Function
  - @brief command for every function
  - @param command (if not void) for every parameter
  - @return command (if not void)
- File
  - @file command, (except @example files)
  - @copyright command

Allman (BSD) indentation style

Classes/Namespace/Structure/Enumeration names: CamelCase Class methods: CamelCase Class members: mCamelCase

Arguments to methods/functions: camelCase (lower case first word)

7.2 Naming Convention

Use CamelCase for Files, Classes, Namespace, Structures, Enumerations, Functions, Procedures, and Member Variables.

Filenames are the name of the class or namespace within – one class per file.

Classes, Namespaces, Structures, Enumerations, and Functions start with a capitalized letter and are nouns: (e.g. SerialPort, SerialStream, etc.). Inhertied functions may be exceptions.

Function names are a description of the return value, and Procedure names are a strong verb followed by an object. (See Code Complete 2 §7.6 for the difference between a function and a procedure verb.)

Function arguments start with a lowercase letter and are nouns; (e.g. numberOfBytes, etc.)

Member Variables start with a lowercase letter "m" and are nouns; (e.g. mFileDescriptor, etc.).

Use underscores for non-member functions and local variables, lower case with an underscore to separate words; (e.g. lower_case, short_names).

Constants and Globals are named identically to variables.

Do not use abbreviations and be as precise and descriptive with naming as possible.

7.3 Indentation

Indentation shall be 4 space characters, not tabs.

Braces shall begin and end on the indentation level.

Namespaces are NOT indented.

Case statements are NOT indented.

Class visibility statements are NOT indented (public, protected, private).

One statement per line – this includes variable declarations.

Do not put short if() . . . ; statements on one line.

If the constructor initializers don't fit on a single line, put constructor initializer list items one per line, starting with the comma and aligned with the colon separator. For example:

```
Class::Class()
    : var1(1)
    , var2(2)
    , var3(3)
{
...
}
```

The purpose of this indentation policy, which can feel "incorrect" at times is to ensure that changes are isolated to the minimum number of lines. Our tools, (compilers, editors, diff viewers, and source code repository), all operate on a line-by-line basis. When someone makes a change that affects a portion anywhere in the line, the tools consider the entire line changed. This can lead to nasty issues like complex merge conflicts, or, worse, obscure the developer activity.

7.4 Include Headers

A good practice is to include headers in the order most local to least local and alphabetize your lists to avoid duplications. The purpose for this is to ensure that a proper dependency chain is maintained. As the project grows larger, these compilation failures sometimes can be difficult to identify and resolve.

This means that header files have includes alphabetized in the order:

- project includes
- project dependency includes
- system includes

Source files have the includes in the order:

- definition includes
- project includes
- project dependency includes
- system includes

CHAPTER 8

Links

LibSerial-1.0.0rc1

Documentation

CHAPTER 9

Indices and tables

- genindex
- modindex
- search