constant robotics

Communication C++ library v6.0

programmer's manual

clib

www.constantrobotics.com

CONTENTS

DOCUMENT VERSIONS

Table 1 – Document versions.

Version	Release date	What's new
3.0	13.07.2020	Programmer's manual for the Communication lib information exchange software C++ library version 3.0.
4.0	28.03.2021	Programmer's manual for the Communication lib information exchange software C++ library version 4.0.
5.0	28.05.2021	Programmer's manual for the Communication lib information exchange software C++ library version 5.0.
6.0	02.08.2021	Programmer's manual for the Communication lib information exchange software C++ library version 6.0.

LIBRARY VERSIONS

Table 2 – Library versions.

Version	Release date	What's new
3.0	13.07.2020	1. Changed the data exchange principle from requesting lost data to confirming received data. 2. Different protocols have been developed for transmitting data over a network and over serial ports. 3. The software code is completely rewritten. 4. Increased performance.
4.0	28.03.2021	1. A new version of Network Transport Protocol version 4.0 is included.
5.0	28.05.2021	1. Changed the data waiting logic implemented in the Get_Data(...) method of the UDPChannel C++ class for more reliable data delivery over the network using UDP packets. 2. Added TCPPort and PureTCPChannel C++ classes to work with TCP connections. 3. Test applications and examples have been changed.
6.0	02.08.2021	1. New version of Network Transport Protocol version 5.0 implemented. 2. The names of the C++ classes have been changed. 3. Added internal error tracer. 4. Small data waiting error fixed. 5. Added mechanism for waiting of the data reception confirmation. 6. Changed software interface. 7. Added methods to retrieve the current version of the library's software modules.

DESCRIPTION

The **Communication lib** version **6.0** (hereinafter referred to as the library) is intended for use in C++ projects for reliable information exchange between two devices over a network or via serial ports. The library can also be used to exchange data between applications within the same computer. The **Network Transport Protocol** version **5.0**, an layer over UDP protocol, is used to communicate over the network. It is also possible to communicate via a TCP connection without any protocol add-on. **Serial Transport Protocol** version **3.0** is used to communicate over serial ports and provides a transport layer.

The library provides a simple programming interface. Developer does not need to worry about initializing serial ports, UDP ports and TCP ports. The library does all the initialization and communication work, providing simple methods for sending and receiving data. The library is cross-platform and compatible with all Windows and Linux operating systems that support the C++ compiler (standard C++11). The library also enables the secure information exchange over a single serial port

from multiple compute threads. The library provides TCP port communication, including the ability to connect multiple clients on a single port. The library provides synchronization of port calls from multiple threads. The library is delivered as source code. Figure 1 shows how the library software interface is organized.

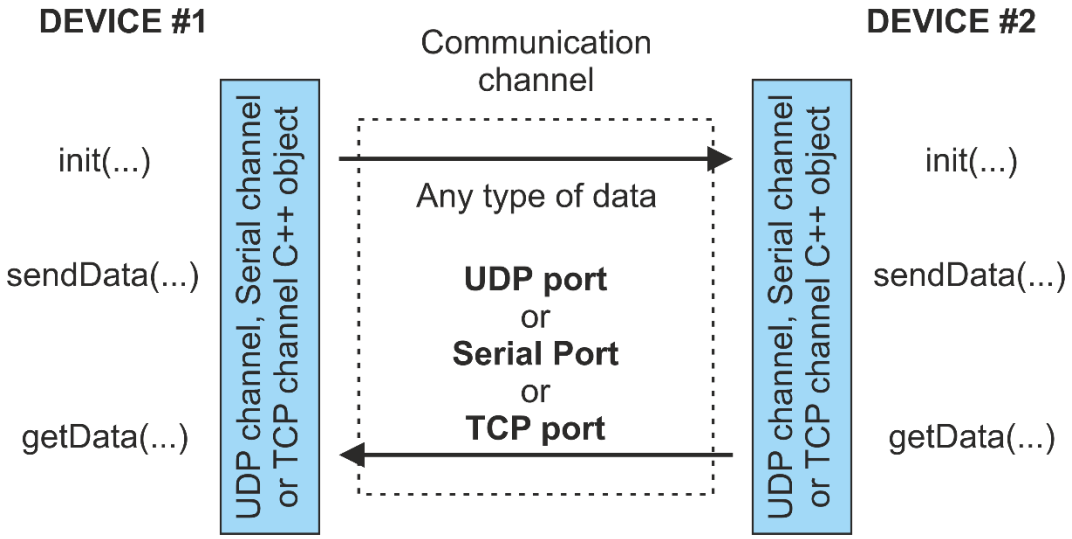

Figure 1 – The library's programming interface principle.

LIBRARY FILES AND PROGRAM CLASSES

Table 3 – Library files.

File	Description
Channel.h	A header file describing the **Channel** interface class that defines the library's programming interface and is parent to the **SerialChannel**, **UdpChannel**, **PureSerialChannel**, **PureUdpChannel** and **PureTcpChannel** classes.
SerialChannel.h, SerialChannel.cpp, SerialChannelVersion.h	**SerialChannel** class header and implementation files for communication via serial ports using **Serial Transport Protocol** version **3.0**.
UdpChannel.h, UdpChannel.cpp, UdpChannelVersion.h	Header files and implementation files for the **UdpChannel** class, designed for communication over the network using **Network Transport Protocol** version **5.0**.
PureSerialChannel.h, PureSerialChannel.cpp, PureSerialChannelVersion.h	Header and implementation files for the **PureSerialChannel** class, which is designed for communication via serial ports without any protocol.
PureUdpChannel.h, PureUdpChannel.cpp, PureUdpChannelVersion.h	Header and implementation files for the **PureUdpChannel** class, designed for communication over the network without any protocol over **UDP**.
PureTcpChannel.h, PureTcpChannel.cpp, PureTcpChannelVersion.h	The header and implementation files of the **PureTcpChannel** class, which is designed for communication over a **TCP** connection.
SerialPort.h, SerialPort.cpp, SerialPortVersion.h	Header files and implementation files of **SerialPort** class for serial port operation (opening, closing, reading and writing). Can be used separately.
UdpSocket.h UdpSocket.cpp, UdpSocketVersion.h	Header and implementation files for **UdpSocket** class to work with **UDP** socket (open, close, read and write). Can be used separately.
TcpSocket.h, TcpSocket.cpp, TcpSocketVersion.h	Header and implementation files for the **TcpSocket** class to work with **TCP** socket (open, connect, close, read and write). Can be used separately.

File	Description
NtpParser.h, NtpParser.cpp, NtpParserVersion.h	Header and implementation files for the **NtpParser** class designed to encode, decode packets and implement **Network Transport Protocol** version **5.0** operation logic.
StpParser.h, StpParser.cpp, StpParserVersion.h	Header files and implementation files of **StpParser** class designed to encode, decode packets and implement **Serial Transport Protocol** version **5.0** operation logic.

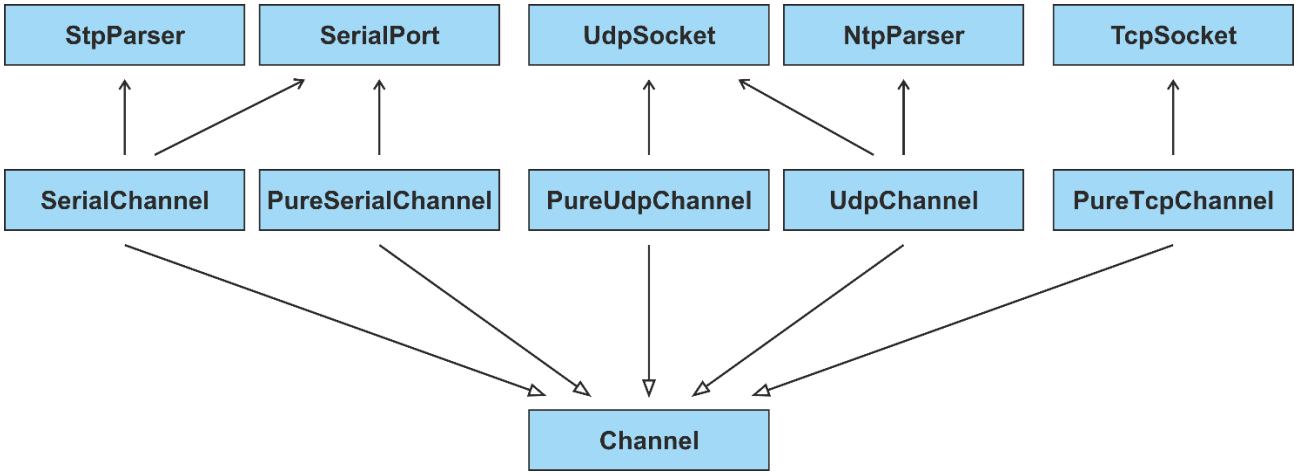

Figure 2 – UML diagram of library classes of the library.

LIBRARY USAGE PRINCIPLE

The library is supplied as source code files in form of CMake project. To use the library, the developer must include the library files in their project (the files are listed in Table 3). You can also pre-build a static or dynamic version of the library (building a CMake project in the standard way, using any CMake-supported environment). The library doesn't have any third-party dependencies. The developer can only include the necessary files from the library for using any interface. The procedure for using the library is follow:

1. Create object of desired class: **UdpChannel**, **SerialChannel**, **PureUdpChannel**, **PureTcpChannel** or **PureSerialChannel**. You can create a pointer to an object of the **Channel** virtual interface class and initialize it with the desired object.
2. Initialize a UDP channel, TCP connection or serial port via the **init(...)** method. For more details about initializing different library classes, see the description of the **init(...)** method.
3. To send data, call **sendData(...)** method passing data pointer, data size and logical port number for **UdpChannel** and **SerialChannel** classes as parameters.
4. Use **getData(...)** method to retrieve received data when available or to wait for data to arrive. For more details about method parameters see description of **getData(...)** method.
5. To exchange data from multiple threads over the same serial port using **SerialChannel** or **PureSerialChannel** class you need to create an instance of **SerialChannel** or **PureSerialChannel** class in each thread with the same initialization parameters (serial port name and baudrate).

DESCRIPTION OF INTERFACE METHODS

CHANNEL CLASS DESCRIPTION

The **Channel** class is an interface class for **UdpChannel**, **SerialChannel**, **PureUdpChannel**, **PureTcpChannel** and **PureSerialChannel** classes. The **Channel** class methods define the libary interface. The Channel class is declared in the **Channel.h** file. The interface is shown below.

```
namespace cr {
namespace clib {

	class Channel
	{
	public:

		static std::string getVersion();

		bool init(std::string config);

		bool sendData(
			uint8_t* data,
			uint32_t dataSize,
			int32_t logicPort = 0,
			uint32_t resendsCount = 0,
			float bandwidthMbps = 0,
			uint32_t waitConfirmedMs = 300);

		bool getData(
			uint8_t* dataBuff,
			uint32_t dataBuffSize,
			uint32_t& inputSize,
			int32_t& logicPort,
			int32_t timeoutMs = 0);

		virtual ~Channel() {};
	};
}
}
```

Table 4 – Channel class methods.

Method	Description
std::string getVersion(…)	A method to get the current version string of a library modules. Versions may differ for each library module (UdpChannel, SerialChannel, PureUdpChannel, PureTcpChannel or PureSerialChannel).
bool init(…)	Initialization method of Channel class object (exchange initialization). Initialization method is different for UdpChannel, SerialChannel, PureUdpChannel, PureSerialChannel and PureTcpChannel classes.
bool sendData(…)	Method to send data.
bool getData(…)	The method to get received data or waiting for data to arrive.
~Channel()	Virtual destructor.

In the following, methods of the **Channel** class will be described with specific of their implementation in **UDPChannel**, **SerialChannel**, **PureUdpChannel**, **PureTcpChannel** and **PureSerialChannel** classes.

INIT(…) METHOD

The init(...) method is used to initialize UDP channel, TCP connection and also to initialize serial port in UdpChannel, SerialChannel, PureUdpChannel, PureTcpChannel and PureSerialChannel classes. The method declaration is given below.

```
bool init(std::string config)
```

Method parameters:
The method has only one parameter **config** – the initialization string. The format of the initialization string differs depending on the class:

1. For **UdpChannel** class the initialization string should have the following format: "destinationIpAddress;inputUdpPort;outputUdpPort", where: destinationIpAddress – IP address string of remote device, inputUdpPort – UDP port number to receive data, outputUdpPort – UDP port number to send data. The UdpChannel class always uses two UDP ports to exchange data. For example, the initialization string for communication with the device having IP address 192.168.2.130 via UDP ports 50010 (input port) and 50011 (output port) will look like: «192.168.2.130;50010;50011». So, the other device needs to initialize the UDP ports mirrored: 50011 (input port) and 50010 (output port) for communication. Suppose our device has IP address 192.168.2.131, then the initialization string for UdpChannel class object on the remote device should have the following format: "192.168.2.131;50011;50010". The UdpChannel class allows exchanges between two applications on the local network on the same computer. In this case, for the UDP port numbers discussed above, the initialization strings for the first and second application respectively will be as follows: "127.0.0.1;50010;50011" и "127.0.0.1;50011;50010".
2. For **SerialChannel** class, initialization string should have following format: "serialPortName;serialPortBaudrate", where: serialPortName – full name of serial port in system (for example: "dev/ttyS1" – for Linux and "\\\\. \\COM1" – for Windows), serialPortBaudrate - exchange rate (available rates: 110, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 128000, 256000, 500000, 1000000 bits per second).
3. For **PureUdpChannel** class, the initialization string is the same as for **UdpChannel** class.
4. For **PureSerialChannel** class the initialization string is the same as for **SerialChannel** class.
5. For the **PureTcpChannel** class, the initialization string has a format that depends on whether the application is a client or a server. If the application is a server the initialization string should have the following format: "destinationIpAddress;tcpPort;0", where: destinationIpAddress – destination IP address of the remote device, tcpPort – TCP port for communication. If the application is a client the initialization string should have the following format: "destinationIpAddress;0;tcpPort", where: destinationIpAddress – destination IP address of the remote device, tcpPort – TCP port for communication. For example, if for communication we use TCP port 50000, the server has IP address "192.168.1.10" and the client "192.168.1.11", then the initialization string for the server will be "192.168.1.11;50000;0" and for the client "192.168.1.10;0;50000".

Return value:
The method returns **TRUE** when initialization was successful or **FALSE** when initialization failed.

SENDDATA(…) METHOD

The sendData(...) method is used to send data. The working principle of the method differs from the class. For UdpChannel and SerialChannel classes, the method puts the data to be sent in a buffer from which it is read by the sending thread. For PureUdpChannel, PureTcpChannel and PureSerialChannel classes the method sends data immediately to UDP port, TCP port or serial port respectively. Method declaration is shown below.

```
bool sendData(
       uint8_t* data,
       uint32_t dataSize,
       int32_t logicPort = 0,
       uint32_t resendsCount = 0,
       float bandwidthMbps = 0,
       uint32_t waitConfirmedMs = 300)
```

Method parameters:

data	Pointer to the data to be sent.
dataSize	Size of data to be sent. When using **UdpChannel** class, maximum size is **4 169 727** bytes. If SrialChannel class is used, maximum size of transmitted data is **4 153 343** bytes. When using **PureUdpChannel** class, it is recommended to transmit data not exceeding maximum size of data included in one UDP packet for a particular case. When using **PureTcpChannel** class, it is recommended to transfer data no larger than maximum allowed for TCP connection. When using **PureSerialChannel** class, maximum size of transmitted data is set on a case-by-case basis. Minimum size of transmitted data is **1 byte**.
logicPort	Logical port number for data transfer according to **Serial Transport Protocol** version **3.0** (see protocol specification) and **Network transport Protocol** version **5.0** (see protocol specification). Can have values from **0** to **15** for **UdpChannel** class and values from **0** to **255** for **SerialChannel** class. Only used in **UdpChannel** and **SerialChannel** classes. Allows to separate data transmission into separate logical streams.
resendsCount	Number of repetitions to send data according to **Serial Transport Protocol** version **3.0** (see protocol specification) and **Network transport Protocol** version **5.0** (see protocol specification). Only used in **UdpChannel** and **SerialChannel** classes. Data resend provides reliable data transmission in case of packet loss in communication channels.
bandwidthMbps	Current data channel bandwidth (megabits per second). Only used in **UdpChannel** class. For **SerialChannel** class serial port bandwidth is defined at initialization stage. This parameter is not used in **PureUdpChannel**, **PureTcpChannel** and **PureSerialChannel** classes.
waitConfirmedMs	Timeout time in milliseconds for waiting data acknowledgement from receiver. If parameter value **>0**, method will wait for acknowledgement of data reception by receiver for specified time (waitConfirmedMs).

Return value:
Method returns **TRUE** if data was successfully sent or successfully added to send buffer. The method returns **FALSE** in case of any errors.

GETDATA(…) METHOD

The getData(...) method is designed to get received data or to wait for data to be received. The principles of the method differ from class to class. For UdpChannel and SerialChannel classes the method checks or waits data according to Network Transport Protocol version 5.0 and Serial Transport Protocol version 3.0 respectively. For PureUdpChannel, PureTcpChannel and PureSerialChannel classes the method waits for any data arrival. Method declaration is shown below.

```
bool getData(
       uint8_t* dataBuff,
       uint32_t dataBuffSize,
       uint32_t& inputSize,
       int32_t& logicPort,
       int32_t timeoutMs = 0)
```

Method parameters:

dataBuff	Pointer to buffer to copy received data.
dataBuffSize	Data buffer size. If the data buffer size is smaller than the received data, the method will return FALSE.
inputSize	Return value of received data size.
logicPort	The number of the logical port for which data is to be received or expected. Only used for UdpChannel and SerialChannel classes.

timeoutMs	The waiting time for data reception. Can have the following values: **<0** – check for new data and wait indefinitely for data to arrive, if not. **0** – only check for new received data for specified logical port. In case of PureUdpChannel, PureTcpChannel and PureSerialChannel classes, method will wait for data for 100 milliseconds. **>0** – check for new data and wait for specified time until it arrives, if no data.

Return value:
The method returns **TRUE** if there is new incoming data. The method returns **FALSE** if:

1. if the UDP ports or serial port is not initialized.
2. if there is no new data (**if the data was read last time, it cannot be retrieved again**) and/or it has not arrived in the time specified.

GETVERSION() METHOD

The getVersion() method is designed to get the string of current version of corresponding module (UdpChannel, SerialChannel, PureTcpChannel, PureSerialChannel and PureUdpChannel). Method declaration is given below:

```
static std::string getVersion()
```

Return value:
The method returns a version string in the following format "Major.Minor.Patch", where Major – major version of the module, Minor – minor version of the module and Patch – patch version. For example, the return value might look like this: "6.0.1".

EXAMPLES OF WORKING WITH THE LIBRARY

EXAMPLE OF WORKING WITH PUREUDPCHANNEL, PURETCPCHANNEL AND PURESERIALCHANNEL CLASSES

Example of send data application

Below is a code of a simple send data application which uses PureUdpChannel, PureTcpChannel and PureSerialChannel classes. Communication parameters (port number, IP addresses and serial port names) are initialized in the source code. The application sends the data at the time interval specified in the source code.

```
#include <iostream>
#include <cstring>
#include <thread>
#include "PureTcpChannel.h"
#include "PureUdpChannel.h"
#include "PureSerialChannel.h"

using namespace cr::clib;

int main(void)
{
```

```
    // Prepare init string for PureUdpChannel.
    uint16_t inputUdpPort = 50001;
    uint16_t outputUdpPort = 50000;
    std::string initStringUdp = "127.0.0.1;" + std::to_string(inputUdpPort) + ";" +
std::to_string(outputUdpPort);

    // Prepare init string for PureTcpChannel.
    uint16_t tcpPort = 50000;
    std::string initStringTcp = "127.0.0.1;0;" + std::to_string(tcpPort);

    // Prepare init string for PureSerialChannel.
    uint32_t baudrate = 115200;
    std::string serialPortName = "";
#if defined(linux) || defined(__linux) || defined(__linux__)|| defined(__FreeBSD__)
    serialPortName = "/dev/ttyS2";
#elif defined(_WIN32) || defined(__WIN32__) || defined(WIN32)
    serialPortName = "\\\\.\\COM2";
#endif
    std::string initStringSerial = serialPortName + ";" + std::to_string(baudrate);

    // Dialog to choose type of channel.
    std::cout << "Enter type of channel: 1 - PureUdpChannel, 2 - PureTcpChannel, 3 -
PureSerialChannel : ";
    int typeOfChannel = 0;
    std::cin >> typeOfChannel;

    // Init channel.
    Channel* channel = nullptr;
    if (typeOfChannel == 1)
    {
        channel = new PureUdpChannel();
        if (!channel->init(initStringUdp))
        {
            std::cout << "ERROR: Pure UDP channel not init. Exit..." << std::endl;
            return -1;
        }
    }
    else if (typeOfChannel == 2)
    {
        channel = new PureTcpChannel();
        if (!channel->init(initStringUdp))
        {
            std::cout << "ERROR: Pure TCP channel not init. Exit..." << std::endl;
            return -1;
        }
    }
    else
    {
        channel = new PureSerialChannel();
        if (!channel->init(initStringSerial))
        {
            std::cout << "ERROR: Pure serial channel not init. Exit..." << std::endl;
            return -1;
        }
    }

    // Init variables.
    uint32_t outputDataSize = 512; // Any value.
    uint8_t* outputData = new uint8_t[outputDataSize];
    uint32_t cycleTimeMs = 500; // Eack 500 ms we send data.

    // Main loop.
    while (true)
       {
        // Prepare data to send.
```

```
        for (uint32_t i = 0; i < outputDataSize; ++i)
            outputData[i] = (uint8_t)(rand() % 255);

        // Send data.
        channel->sendData(outputData, outputDataSize);
        std::cout << "Send " << outputDataSize << " bytes " << std::endl;

        std::this_thread::sleep_for(std::chrono::milliseconds(cycleTimeMs));
      }

      return 1;
}
```

Explanation of the send data application

First the communication parameters are initialized for PureUdpChannel class. For PureUdpChannel class it is necessary to define IP address of data receiver, UDP port number of data reception (data reception port is always specified) and UDP port number of data sending. Initialization string is formed based on these parameters. In given example exchange is performed in local network ("127.0.0.1").

```
uint16_t inputUdpPort = 50001;
uint16_t outputUdpPort = 50000;
std::string initStringUdp = "127.0.0.1;" + std::to_string(inputUdpPort) + ";" +
std::to_string(outputUdpPort);
```

After that communication parameters are initialized for PureTcpChannel class. For PureTcpChannel class it is necessary to define IP address of data receiver and TCP port. The sender in this case is a client (can also be a server). Initialization string is formed based on these parameters. In given example data exchange is performing in local network ("127.0.0.1").

```
uint16_t tcpPort = 50000;
std::string initStringTcp = "127.0.0.1;0;" + std::to_string(tcpPort);
```

For PureSerialChannel class, you need to define the name of serial port in the system (name format differs in Windows and Linux operating systems), as well as exchange rate. Based on these parameters, an initialization string for PureSerialChannel class is formed (serial port number 2 is used for sending data).

```
uint32_t baudrate = 115200;
std::string serialPortName = "";
#if defined(linux) || defined(__linux) || defined(__linux__)|| defined(__FreeBSD__)
serialPortName = "/dev/ttyS2";
#elif defined(_WIN32) || defined(__WIN32__) || defined(WIN32)
serialPortName = "\\\\.\\COM2";
#endif
std::string initStringSerial = serialPortName + ";" + std::to_string(baudrate);
```

Once the initialization strings for the different exchange classes (PureUdpChannel, PureTcpChannel and PureSerialChannel) have been created, the application prompts the user to select the communication channel type (PureUdpChannel, PureTcpChannel or PureSerialChannel).

```
std::cout << "Enter type of channel: 1 - PureUdpChannel, 2 - PureTcpChannel, 3 -
PureSerialChannel : ";
int typeOfChannel = 0;
std::cin >> typeOfChannel;
```

After that, initialize the pointer to the Channel interface class with the desired object (according to the user's choice).

```
Channel* channel = nullptr;
if (typeOfChannel == 1)
{
    channel = new PureUdpChannel();
    if (!channel->init(initStringUdp))
    {
        std::cout << "ERROR: Pure UDP channel not init. Exit..." << std::endl;
        return -1;
    }
}
else if (typeOfChannel == 2)
{
    channel = new PureTcpChannel();
    if (!channel->init(initStringUdp))
    {
        std::cout << "ERROR: Pure TCP channel not init. Exit..." << std::endl;
        return -1;
    }
}
else
{
    channel = new PureSerialChannel();
    if (!channel->init(initStringSerial))
    {
        std::cout << "ERROR: Pure serial channel not init. Exit..." << std::endl;
        return -1;
    }
}
```

Once the channel is initialized, the variables required for exchange (data buffers and variables) are initialized.

```
uint32_t outputDataSize = 512; // Any value.
uint8_t* outputData = new uint8_t[outputDataSize];
uint32_t cycleTimeMs = 500; // Each 500 ms we send data.
```

Once all necessary variables have been initialized, the application will send data in a loop using UDP packets or TCP connection or serial port (depending on user choice). The data is sent according to the specified time interval.

```
while (true)
{
    // Prepare data to send.
    for (uint32_t i = 0; i < outputDataSize; ++i)
        outputData[i] = (uint8_t)(rand() % 255);

    // Send data.
    channel->sendData(outputData, outputDataSize);
    std::cout << "Send " << outputDataSize << " bytes " << std::endl;
```

```
    std::this_thread::sleep_for(std::chrono::milliseconds(cycleTimeMs));
}
```

Example of the receiving data application

Below is a code of a simple receiving data application which uses PureUdpChannel, PureTcpChannel and PureSerialChannel classes. Communication parameters (port number, IP addresses and serial port names) are initialized in the source code. The application waits data for the time specified in the source code.

```
#include <iostream>
#include <cstring>
#include <thread>
#include "PureTcpChannel.h"
#include "PureUdpChannel.h"
#include "PureSerialChannel.h"

using namespace cr::clib;

int main(void)
{
    // Prepare init string for PureUdpChannel.
    uint16_t inputUdpPort = 50000;
    uint16_t outputUdpPort = 50001;
    std::string initStringUdp = "127.0.0.1;" + std::to_string(inputUdpPort) + ";" +
std::to_string(outputUdpPort);

    // Prepare init string for PureTcpChannel.
    uint16_t tcpPort = 50000;
    std::string initStringTcp = "127.0.0.1;" + std::to_string(tcpPort) + ";0";

    // Prepare init string for PureSerialChannel.
    uint32_t baudrate = 115200;
    std::string serialPortName = "";
#if defined(linux) || defined(__linux) || defined(__linux__)|| defined(__FreeBSD__)
    serialPortName = "/dev/ttyS1";
#elif defined(_WIN32) || defined(__WIN32__) || defined(WIN32)
    serialPortName = "\\\\.\\COM1";
#endif
    std::string initStringSerial = serialPortName + ";" + std::to_string(baudrate);

    // Dialog to choose type of channel.
    std::cout << "Enter type of channel: 1 - PureUdpChannel, 2 - PureTcpChannel, 3 -
PureSerialChannel : ";
    int typeOfChannel = 0;
    std::cin >> typeOfChannel;

    // Init channel.
    Channel* channel = nullptr;
    if (typeOfChannel == 1)
    {
        channel = new PureUdpChannel();
        if (!channel->init(initStringUdp))
        {
            std::cout << "ERROR: Pure UDP channel not init. Exit..." << std::endl;
            return -1;
        }
    }
    else if (typeOfChannel == 2)
    {
```

```
        channel = new PureTcpChannel();
        if (!channel->init(initStringTcp))
        {
            std::cout << "ERROR: Pure TCP channel not init. Exit..." << std::endl;
            return -1;
        }
    }
    else
    {
        channel = new PureSerialChannel();
        if (!channel->init(initStringSerial))
        {
            std::cout << "ERROR: Pure serial channel not init. Exit..." << std::endl;
            return -1;
        }
    }

    // Init variables.
    uint32_t dataBufferSize = 8192; // Any big value.
    uint8_t* dataBuffer = new uint8_t[dataBufferSize];
    uint32_t inputDataSize = 0;
    uint32_t waitDataTimeoutMs = 1000; // 1000 ms to wait data.

       // Main loop.
       while (true)
       {
              // Wait 1000 ms for new data.
        int32_t conn_id = 0;
        if (channel->getData(dataBuffer, dataBufferSize, inputDataSize, conn_id,
waitDataTimeoutMs))
             {
            std::cout << "Get " << inputDataSize << " bytes" << std::endl;
             }
        else
        {
            std::cout << "No data" << std::endl;
        }
       }

       return 1;
}
```

Explanation of the data receiving application

First the communication parameters are initialized for PureUdpChannel class. For PureUdpChannel class it is necessary to define IP address of data receiver, UDP port number for data reception (data reception port is always specified) and UDP port number for data sending. Initialization string is formed based on these parameters. Communication is performed in local network ("127.0.0.1").

```
uint16_t inputUdpPort = 50000;
uint16_t outputUdpPort = 50001;
std::string initStringUdp = "127.0.0.1;" + std::to_string(inputUdpPort) + ";" +
std::to_string(outputUdpPort);
```

After that communication parameters are initialized for PureTcpChannel class. For PureTcpChannel class you must define IP address of data sender and TCP port. The sender in this case is a server (can also be a client). Initialization string is formed based on these parameters. Communication is performed in local network ("127.0.0.1").

```
uint16_t tcpPort = 50000;
std::string initStringTcp = "127.0.0.1;" + std::to_string(tcpPort) + ";0";
```

For PureSerialChannel class, you need to define the name of serial port in the system (name format differs in Windows and Linux operating systems), as well as exchange rate. Based on these parameters, an initialization string for PureSerialChannel class is formed (serial port number 1 is used for sending data).

```
uint32_t baudrate = 115200;
std::string serialPortName = "";
#if defined(linux) || defined(__linux) || defined(__linux__)|| defined(__FreeBSD__)
serialPortName = "/dev/ttyS1";
#elif defined(_WIN32) || defined(__WIN32__) || defined(WIN32)
serialPortName = "\\\\.\\COM1";
#endif
std::string initStringSerial = serialPortName + ";" + std::to_string(baudrate);
```

Once the initialisation strings for the different exchange classes (PureUdpChannel, PureTcpChannel and PureSerialChannel) have been formed, the application prompts the user to select the exchange channel type.

```
std::cout << "Enter type of channel: 1 - PureUdpChannel, 2 - PureTcpChannel, 3 -
PureSerialChannel : ";
int typeOfChannel = 0;
std::cin >> typeOfChannel;
```

After that, initialize the pointer to the Channel interface class with the desired object (according to the user's choice).

```
Channel* channel = nullptr;
if (typeOfChannel == 1)
{
    channel = new PureUdpChannel();
    if (!channel->init(initStringUdp))
    {
        std::cout << "ERROR: Pure UDP channel not init. Exit..." << std::endl;
        return -1;
    }
}
else if (typeOfChannel == 2)
{
    channel = new PureTcpChannel();
    if (!channel->init(initStringTcp))
    {
        std::cout << "ERROR: Pure TCP channel not init. Exit..." << std::endl;
        return -1;
    }
}
else
{
    channel = new PureSerialChannel();
    if (!channel->init(initStringSerial))
    {
        std::cout << "ERROR: Pure serial channel not init. Exit..." << std::endl;
        return -1;
    }
}
```

Once the channel is initialized, the variables required for communication (data buffers and variables) are initialized.

```
uint32_t dataBufferSize = 8192; // Any big value.
uint8_t* dataBuffer = new uint8_t[dataBufferSize];
uint32_t inputDataSize = 0;
uint32_t waitDataTimeoutMs = 1000; // 1000 ms to wait data.
```

After initialization of all required variables, the application waits in a loop for data to arrive (UDP packets, data via TCP connection or data from the serial port). The waiting time is set in the parameters.

```
while (true)
{
    // Wait 1000 ms for new data.
    int32_t conn_id = 0;
    if (channel->getData(dataBuffer, dataBufferSize, inputDataSize, conn_id,
waitDataTimeoutMs))
    {
        std::cout << "Get " << inputDataSize << " bytes" << std::endl;
    }
    else
    {
        std::cout << "No data" << std::endl;
    }
}
```

EXAMPLE OF WORKING WITH UDPCHANNEL CLASS

Example of the send data application

Below is a code of a simple send data application which uses UdpChannel class. Communication parameters (port numbers and IP address) are initialized in the source code. The application sends data at the time interval specified in the source code.

```
#include <iostream>
#include <cstring>
#include "UdpChannel.h"

using namespace cr::clib;

int main(void)
{
    // Prepare init string for UdpChannel.
    uint16_t inputUdpPort = 50000;
    uint16_t outputUdpPort = 50001;
    std::string destinationIp = "127.0.0.1";
    std::string initString = destinationIp + ";" + std::to_string(inputUdpPort) + ";" +
std::to_string(outputUdpPort);

    // Init UdpChannel.
    Channel* channel = new UdpChannel();
    if (!channel->init(initString))
    {
        std::cout << "ERROR: UDP channel not init" << std::endl;
        return -1;
```

```
    }

    // Init variables.
    uint32_t outputDataSize = 8192; // Any value.
    uint8_t* outputData = new uint8_t[outputDataSize];
    uint32_t cycleTimeMs = 500; // Eack 500 ms we send data.
    uint32_t waitConfirmTimeoutMs = 50; // 1000 ms to wait data confirmation.
    int32_t logicPort = 0;
    float channelBandwidthMbps = 10.0f;

    // Main loop.
    while (true)
    {
        // Prepare data to send.
        for (uint32_t i = 0; i < outputDataSize; ++i)
            outputData[i] = (uint8_t)(rand() % 255);

        // Send data.
        if (channel->sendData(outputData, outputDataSize, logicPort, 1,
channelBandwidthMbps, waitConfirmTimeoutMs))
            std::cout << "Send " << outputDataSize << " bytes and confirmed" << std::endl;
        else
            std::cout << "Data not confirmed" << std::endl;

        std::this_thread::sleep_for(std::chrono::milliseconds(cycleTimeMs));
    }

    return 1;
}
```

Explanation of the send data application

First the communication parameters are initialized. For UdpChannel class it is necessary to define IP address of data receiver, number of UDP port for data reception (data reception port is always specified and is necessary to implement Network Transport Protocol version 5.0 operation logic) and number of UDP port for data sending. Based on these parameters the initialization string for the class is generated.

```
uint16_t inputUdpPort = 50000;
uint16_t outputUdpPort = 50001;
std::string destinationIp = "127.0.0.1";
std::string initString = destinationIp + ";" + std::to_string(inputUdpPort) + ";" +
std::to_string(outputUdpPort);
```

After that the object of UdpChannel class is initialized.

```
Channel* channel = new UdpChannel();
if (!channel->init(initString))
{
    std::cout << "ERROR: UDP channel not init" << std::endl;
    return -1;
}
```

After the channel is initialized, the variables required for communication (data buffers and variables) are initialized. The channel bandwidth of the data exchange and the acknowledgement wait time are set. If the specified acknowledgement wait time **≤0**, the sender will not wait for an acknowledgement from the receiver.

```
uint32_t outputDataSize = 8192; // Any value.
uint8_t* outputData = new uint8_t[outputDataSize];
uint32_t cycleTimeMs = 500; // Eack 500 ms we send data.
uint32_t waitConfirmTimeoutMs = 50; // 1000 ms to wait data confirmation.
int32_t logicPort = 0;
float channelBandwidthMbps = 10.0f;
```

Once all the required variables have been initialized, the application sends the data in a loop. The data is sent according to the specified data interval.

```
while (true)
{
    // Prepare data to send.
    for (uint32_t i = 0; i < outputDataSize; ++i)
        outputData[i] = (uint8_t)(rand() % 255);

    // Send data.
    if (channel->sendData(outputData, outputDataSize, logicPort, 1, channelBandwidthMbps,
waitConfirmTimeoutMs))
        std::cout << "Send " << outputDataSize << " bytes and confirmed" << std::endl;
    else
        std::cout << "Data not confirmed" << std::endl;

    std::this_thread::sleep_for(std::chrono::milliseconds(cycleTimeMs));
}
```

Example of the data receiving application

Below is a code of a simple data receiving application which uses UdpChannel class. Communication parameters (port number and IP address) are initialized in the source code. The application waits for the data to arrive for the time specified in the source code.

```
#include <iostream>
#include <cstring>
#include "UdpChannel.h"

using namespace cr::clib;

int main(void)
{
    // Prepare init string for UdpChannel.
    uint16_t inputUdpPort = 50001;
    uint16_t outputUdpPort = 50000;
    std::string destinationIp = "127.0.0.1";
    std::string initString = destinationIp + ";" + std::to_string(inputUdpPort) + ";" +
std::to_string(outputUdpPort);

    // Init UdpChannel.
    Channel* channel = new UdpChannel();
    if (!channel->init(initString))
    {
        std::cout << "ERROR: UDP channel not init" << std::endl;
        return -1;
    }

    // Init variables.
    uint32_t dataBufferSize = 8192; // Any big value.
    uint8_t* dataBuffer = new uint8_t[dataBufferSize];
```

```
    uint32_t inputDataSize = 0;
    uint32_t waitDataTimeoutMs = 1000; // 1000 ms to wait data.
    int32_t logicPort = 0;

    // Main loop.
    while (true)
    {
        // Wait 1000 ms for new data.
        if (channel->getData(dataBuffer, dataBufferSize, inputDataSize, logicPort,
waitDataTimeoutMs))
        {
            std::cout << "Get " << inputDataSize << " bytes" << std::endl;
        }
        else
        {
            std::cout << "No data" << std::endl;
        }
    }

    return 1;
}
```

Explanation of the data receiving application

First the communication parameters are initialized. For UdpChannel class you should define IP address of data sender, UDP port for receiving data and UDP port for sending data (data sending port is always specified and is necessary to implement Network Transport Protocol version 5.0 operation logic). Data send and receive ports must be "mirrored" with the data send application. The initialization string for UdpChannel class is generated based on these parameters.

```
uint16_t inputUdpPort = 50001;
uint16_t outputUdpPort = 50000;
std::string destinationIp = "127.0.0.1";
std::string initString = destinationIp + ";" + std::to_string(inputUdpPort) + ";" +
std::to_string(outputUdpPort);
```

After that the object of UdpChannel class is initialized.

```
Channel* channel = new UdpChannel();
if (!channel->init(initString))
{
    std::cout << "ERROR: UDP channel not init" << std::endl;
    return -1;
}
```

Once the channel is initialized, the variables required for exchange (data buffers and variables) are initialized.

```
uint32_t dataBufferSize = 8192; // Any big value.
uint8_t* dataBuffer = new uint8_t[dataBufferSize];
uint32_t inputDataSize = 0;
uint32_t waitDataTimeoutMs = 1000; // 1000 ms to wait data.
int32_t logicPort = 0;
```

Once all the required variables have been initialized, the application waits in a loop for new data to arrive. The waiting time is set in the parameters.

```
while (true)
{
    // Wait 1000 ms for new data.
    if (channel->getData(dataBuffer, dataBufferSize, inputDataSize, logicPort,
waitDataTimeoutMs))
    {
        std::cout << "Get " << inputDataSize << " bytes" << std::endl;
    }
    else
    {
        std::cout << "No data" << std::endl;
    }
}
```

EXAMPLE OF WORKING WITH SERIALCHANNEL CLASS

Example of the send data application

Below is a code of a simple send data application which uses SerialChannel class. Communication parameters (serial port name and exchange rate) are initialized in the source code. The application sends data at the time interval specified in the source code.

```
#include <iostream>
#include <cstring>
#include "SerialChannel.h"

using namespace cr::clib;

int main(void)
{
    // Prepare init string for SerialChannel.
    uint32_t baudrate = 115200;
    std::string serialPortName = "";
#if defined(linux) || defined(__linux) || defined(__linux__)|| defined(__FreeBSD__)
    serialPortName = "/dev/ttyS2";
#elif defined(_WIN32) || defined(__WIN32__) || defined(WIN32)
    serialPortName = "\\\\.\\COM2";
#endif
    std::string initString = serialPortName + ";" + std::to_string(baudrate);

    // Init SerialChannel.
    Channel* channel = new SerialChannel();
    if (!channel->init(initString))
    {
        std::cout << "ERROR: Serial channel not init" << std::endl;
        return -1;
    }

    // Init variables.
    uint32_t outputDataSize = 512; // Any value.
    uint8_t* outputData = new uint8_t[outputDataSize];
    uint32_t cycleTimeMs = 500; // Each 500 ms we send data.
    int32_t logicPort = 0;
```

```
    // Main loop.
    while (true)
    {
        // Prepare data to send.
        for (uint32_t i = 0; i < outputDataSize; ++i)
            outputData[i] = (uint8_t)(rand() % 255);

        // Send data.
        if (channel->sendData(outputData, outputDataSize, logicPort, 1))
            std::cout << "Send " << outputDataSize << " bytes" << std::endl;
        else
            std::cout << "Data not confirmed" << std::endl;

        std::this_thread::sleep_for(std::chrono::milliseconds(cycleTimeMs));
    }

    return 1;
}
```

Explanation of the send data application

First the communication parameters are initialized. For SerialChannel class you need to define serial port name in the system and communication speed. The initialization string for SerialChannel class is formed based on these parameters.

```
uint32_t baudrate = 115200;
std::string serialPortName = "";
#if defined(linux) || defined(__linux) || defined(__linux__)|| defined(__FreeBSD__)
serialPortName = "/dev/ttyS2";
#elif defined(_WIN32) || defined(__WIN32__) || defined(WIN32)
serialPortName = "\\\\.\\COM2";
#endif
std::string initString = serialPortName + ";" + std::to_string(baudrate);
```

After that, the object of SerialChannel class is initialized.

```
Channel* channel = new SerialChannel();
if (!channel->init(initString))
{
    std::cout << "ERROR: Serial channel not init" << std::endl;
    return -1;
}
```

Once the channel is initialized, the variables required for communication (data buffers and variables) are initialized.

```
uint32_t outputDataSize = 512; // Any value.
uint8_t* outputData = new uint8_t[outputDataSize];
uint32_t cycleTimeMs = 500; // Each 500 ms we send data.
int32_t logicPort = 0;
```

Once all necessary variables have been initialized, the application executes data sending in a loop. The data is sent according to the specified data interval.

```
while (true)
{
    // Prepare data to send.
    for (uint32_t i = 0; i < outputDataSize; ++i)
        outputData[i] = (uint8_t)(rand() % 255);

    // Send data.
    if (channel->sendData(outputData, outputDataSize, logicPort, 1))
        std::cout << "Send " << outputDataSize << " bytes" << std::endl;
    else
        std::cout << "Data not confirmed" << std::endl;

    std::this_thread::sleep_for(std::chrono::milliseconds(cycleTimeMs));
}
```

Example of the data receiving application

The following is a code of a simple data receiving application which uses SerialChannel class. The communication parameters (serial port name and exchange rate) are initialized in the source code. The application waits new data for the time specified in the source code.

```
#include <iostream>
#include <cstring>
#include "SerialChannel.h"

using namespace cr::clib;

int main(void)
{
    // Prepare init string for SerialChannel.
    uint32_t baudrate = 115200;
    std::string serialPortName = "";
#if defined(linux) || defined(__linux) || defined(__linux__)|| defined(__FreeBSD__)
    serialPortName = "/dev/ttyS1";
#elif defined(_WIN32) || defined(__WIN32__) || defined(WIN32)
    serialPortName = "\\\\.\\COM1";
#endif
    std::string initString = serialPortName + ";" + std::to_string(baudrate);

    // Init SerialChannel.
    Channel* channel = new SerialChannel();
    if (!channel->init(initString))
    {
        std::cout << "ERROR: Serial channel not init" << std::endl;
        return -1;
    }

    // Init variables.
    uint32_t dataBufferSize = 8192; // Any big value.
    uint8_t* dataBuffer = new uint8_t[dataBufferSize];
    uint32_t inputDataSize = 0;
    uint32_t waitDataTimeoutMs = 1000; // 1000 ms to wait data.
    int32_t logicPort = 0;

    // Main loop.
    while (true)
    {
        // Wait 1000 ms for new data.
        if (channel->getData(dataBuffer, dataBufferSize, inputDataSize, logicPort,
waitDataTimeoutMs))
```

```
        {
            std::cout << "Get " << inputDataSize << " bytes" << std::endl;
        }
        else
        {
            std::cout << "No data" << std::endl;
        }
    }

    return 1;
}
```

Explanation of the data receiving application

First the communication parameters are initialized. For SerialChannel class you need to define serial port name in the system and data exchange speed. The initialization string for SerialChannel class is formed based on these parameters.

```
uint32_t baudrate = 115200;
std::string serialPortName = "";
#if defined(linux) || defined(__linux) || defined(__linux__)|| defined(__FreeBSD__)
serialPortName = "/dev/ttyS1";
#elif defined(_WIN32) || defined(__WIN32__) || defined(WIN32)
serialPortName = "\\\\.\\COM1";
#endif
std::string initString = serialPortName + ";" + std::to_string(baudrate);
```

After that, the object of SerialChannel class is initialized.

```
Channel* channel = new SerialChannel();
if (!channel->init(initString))
{
    std::cout << "ERROR: Serial channel not init" << std::endl;
    return -1;
}
```

After the channel is initialized, the variables required for exchange (data buffers and variables) are initialized. The waiting time for data arrival is specified.

```
uint32_t dataBufferSize = 8192; // Any big value.
uint8_t* dataBuffer = new uint8_t[dataBufferSize];
uint32_t inputDataSize = 0;
uint32_t waitDataTimeoutMs = 1000; // 1000 ms to wait data.
int32_t logicPort = 0;
```

After initializing all the required variables, the application waits in a loop for the arrival of the data. Waiting time to be set in the parameters.

```
while (true)
{
    // Wait 1000 ms for new data.
    if (channel->getData(dataBuffer, dataBufferSize, inputDataSize, logicPort,
waitDataTimeoutMs))
    {
```

```
        std::cout << "Get " << inputDataSize << " bytes" << std::endl;
    }
    else
    {
        std::cout << "No data" << std::endl;
    }
}
```