FullmoCommLib

Version 0.4.1
1/30/2013 10:27:00 PM

Table of Contents

FullmoCommLib

```
$Id: fullmocommlib.h 1448 2013-01-30 20:00:46Z heggelor $
```

FullmoCommLib - Simple Library for ASCII protocol applications via network

The **FCommChannel** C++ class provides easy and straightforward access to TCP/UDP sockets, along with basic text based telegram processing.

FCommChannel requires parts of the Qt v4.8 library under the LGPL license (www.qt-project.org). It uses Qt network socket classes to provide networking capabilities. It is intended for easy integration of network ASCII application protocols into existing software applications that do not provide straightforward network socket features.

Usage:

- Include the fullmocommlib.h header file in your source code
- add FullmoCommLib.lib as a linker input dependency
- run/distribute the following files along with your application:
    - FullmoCommLib.dll
    - QtCore4.dll
    - QtNetwork4.dll
    - the /Microsoft.VC90.CRT directory
-

Both 32-bit (i386) and 64-bit (amd64) versions of the DLLs can be compiled. See the corresponding subdirectories.

For non-C++ applications (C, Visual Basic, ..) a C-style wrapper for the FullmoCommLib object is available. Please use the fullmocommlibc.h instead of fullmocommlib.h.

Example Code C++

A simple Win32 Console application (Visual Studio 2008) could look like this:

```
#include <iostream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	cout << "FullmoCommLib C++ Style Demo" << endl;
	cout << "Enter remote ip adress and TCP port in format <ipadress>:<port>, e.g. 
192.168.2.100:10001" << endl;
	cout << "<ipadress>:<port> = ";
	char channelName[80];
	cin.getline(channelName, 80, '\n');
	if (strlen(channelName) == 0) {
		return 1;
	}

	FCommChannel fComm;

	cout << "Opening " << channelName << " ... " << endl;
	if (!fComm.open(channelName)) {
		cout << "Failed to connect" << endl;
	}
	else {
		cout << "Connected" << endl;

		cout << "Sending: Hello World!" << endl;

		int retVal = fComm.sendMsg("Hello World!\r", "ok\r", "error\r");
		if (retVal == 0) {
			printf("send ok, but no answer\n");
		}
		else if (retVal == 1) {
			printf("'ok' received\n");
		}
		else if (retVal == 2) {
			printf("'error' received\n");
		}
		else {
			printf("send failed\n");
		}

		// read first line of RX data
		char answerBuf[256];
		const int bufSize = 256;
		int answerSize = fComm.readMsg(answerBuf, bufSize, "\r");
		cout << "first RX line: " << answerBuf << endl;

		// done
		fComm.close();
		cout << "Socket closed" << endl;
	}
	cout << "Press [ENTER]..." << endl;
	cin.get();
	return 0;
}
```

Example Code C

Using the c-style API, the example would look like as follows. The c-style header file and function definitions can be easily re-used for other programming environments (Visual Basic, Labview, ...)

```
#include "fullmocommlibc.h"
#include "stdafx.h"
#include <string.h>

int _tmain(int argc, _TCHAR* argv[])
{
        printf("FullmoCommLib C-Style Demo\n");
        printf("Enter remote ip adress and TCP port in format <ipadress>:<port>, e.g. 
192.168.2.100:10001\n");
        printf("<ipadress>:<port> = ");
        char channelName[80];
        scanf_s("%s", channelName, 80);
        fgetc(stdin); // Discard ENTER
        if (strlen(channelName) == 0) {
                return 1;
        }

        printf("Opening %s ...\n", channelName);
        if (!fCommOpen(channelName)) {
                printf("Failed to connect\n");
                // never forget to clean up in C!
                fCommClose();
        }
        else {
                printf("Connected\n");
                printf("Sending: Hello World!\n");

                // clear all previous communication before sending the new command
                fCommClearMsgBuffer();

                int retVal = fCommSendMsg("Hello World!\r", "ok\r", "error\r");
                if (retVal == 0) {
                        printf("send ok, but no answer\n");
                }
                else if (retVal == 1) {
                        printf("'ok' received\n");
                }
                else if (retVal == 2) {
                        printf("'error' received\n");
                }
                else {
                        printf("send failed\n");
                }

                // read first line of RX data
                char answerBuf[256];
                const int bufSize = 256;
                int answerSize = fCommReadMsg(answerBuf, bufSize, "\r");
                printf("first RX line: %s\n", answerBuf);

                // done
                fCommClose();
                printf("Socket closed\n");
        }

        printf("Library cleanup...\n");
        fCommLibCleanup();
        printf("Press [ENTER]...\n");
        getchar();
        return 0;
}
```

C-style functions

FullmoCommLib C functions

```
    @verbatim
```

$Id: fullmocommlibc.h 770 2012-02-28 10:41:42Z heggelor $

For non-C++ applications, a traditional C-style API is provided to access the **FCommChannel** functions. The use is almost identical to the C++ class, except that you can provide an arbitrary channelId identifier to refer to a specific communication object. If left out, 0 (zero) is used and you can only manage one connection at a time.

fCommOpen, fCommClose, fCommIsOpen, ... are C wrapper functions around the **FCommChannel** object.

fCommOpen is used at the start of the communication and additionally creates a new **FCommChannel** object, if required.

fCommClose closes the channel and additionally deletes the **FCommChannel** object.

Please note: You must take care yourself that communication resources are properly cleaned up after use. Even if a fCommOpen() failed, you have to make a call to fCommClose() to destroy the underlying communication object. It is recommended to use fCommCleanup() to destroy all communication objects and resources in your shutdown/exit code.

See Also:

FCommChannel

API functions overview

```
void fCommLibCleanup();
int fCommOpen(const char *channelSpec, const unsigned int timeout = 30000, const int channelId =
0);
int fCommClose(const int channelId = 0);
int fCommIsOpen(const int channelId = 0);
int fCommSendMsg(const char *msg, const char *answer1 = 0, const char *answer2 = 0, const
unsigned int timeout = 10000, const int channelId = 0);
int fCommWaitForAnswer(const char *answer1 = 0, const char *answer2 = 0, const unsigned int
timeout = 10000, const int channelId = 0);
int fCommReadMsg(char *msgBuf, const unsigned int msgBufSize, const char *endOfMsg = 0, const int
channelId = 0);
int fCommClearMsgBuffer(const int channelId = 0);
int fCommParseMsg(const char *endOfMsg, const char *partSeparator = 0, const char *keyword = 0,
const unsigned int timeout = 10000);
int fCommGetMsgPart(char *msgPartBuf, const unsigned int msgPartBufSize, const unsigned int
partIndex = 0);
int fCommGetLastError(char *errMsgBuf, const unsigned int errMsgBufSize, const int channelId =
0);
int fCommHasError(const int channelId = 0);
```

Class Index

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Class Documentation

FCommChannel Class Reference

FCommChannel communication class for simple ASCII protocols.

`#include <fullmocommlib.h>`

Public Types

- enum { **parseRawDataBufSize** = 4096 }

Public Member Functions

- **FCommChannel** ()
- bool **open** (const char *channelSpec, const unsigned int timeout=30000)
  open a new communication channel
- bool **close** ()
  close the communication channel
- bool **isOpen** ()
  channel status: open or closed?
- int **sendMsg** (const char *msg, const char *answer1=0, const char *answer2=0, const unsigned int timeout=10000)
  send new ASCII command
- int **waitForAnswer** (const char *answer1=0, const char *answer2=0, const unsigned int timeout=10000)
  check incoming data and wait for answer
- int **parseMsg** (const char *endOfMsg, const char *partSeparator=0, const char *keyword=0, const unsigned int timeout=10000)
  Parse incoming answers.
- int **getMsgPart** (char *msgPartBuf, const unsigned int msgPartBufSize, const unsigned int partIndex=0)
  get the data parts of a parsed message
- int **readMsg** (char *msgBuf, const unsigned int msgBufSize, const char *endOfMsg=0)
  read receive data
- int **clearMsgBuffer** ()
  clear all previously accumulated messages
- bool **getLastError** (char *errMsgBuf, const unsigned int errMsgBufSize)
  (### error messages not yet supported)
- bool **hasError** ()
  (### error messages not yet supported)

Detailed Description

FCommChannel communication class for simple ASCII protocols.

FCommChannel encapsulates the network socket / device i.o. and provides a simplistic interface that can be used for reading and writing messages. **FCommChannel** is not event based currently. Incoming data is simply accumulated until processed by **FCommChannel::readMsg()**. See **Example Code C++** for a small example application.

Constructor & Destructor Documentation

FCommChannel::FCommChannel ()

```
$Id: fullmocommlibc.cpp 770 2012-02-28 10:41:42Z heggelor $
```

Member Function Documentation

int FCommChannel::clearMsgBuffer ()

clear all previously accumulated messages

This is useful in combination with **sendMsg()**/readMsg() or **sendMsg()**/parseMsg(), where you want to discard any messages in the input buffer first before calling **sendMsg()**.

Returns:

always 0.

bool FCommChannel::close ()

close the communication channel

Returns:

True, if an open connection has been closed. False if it was closed already.

int FCommChannel::getMsgPart (char * *msgPartBuf*, const unsigned int *msgPartBufSize*, const unsigned int *partIndex* = `0`)

get the data parts of a parsed message

this function is used instead of after a **parseMsg()** call to read out the interesting parts from the full answer.

Parameters:

msgPartBuf	character array where **parseMsg()** can store the parsed data as a zero-terminated ASCII string.
msgPartBufSize	the size of msgPartBuf
partIndex	from 0 to (partSize - 1). With "partSize" the number of message parts as returned by **parseMsg()**. If partIndex exceeds the amount of sections/parts available, and empty string is returned.

Returns:

the size of the message returned in msgBuf.

See Also:

parseMsg()

bool FCommChannel::isOpen ()

channel status: open or closed?

Returns:

True if the connection has been opened successfully and data can be sent or read.

bool FCommChannel::open (const char * *channelSpec*, const unsigned int *timeout* = `30000`)

open a new communication channel

(re)opens a TCP client connection.

Parameters:

channelSpec	for a TCP client connection use syntax (ip address):(tcp port number), e.g. "192.168.2.100:10001".
timeout	Timeout for the TCP connection request.

Returns:

True, if the connection could be openend and the connection has been established. False, if an error occurred opening the connection or the connection could not be established within 'timeout'.

int FCommChannel::parseMsg (const char * *endOfMsg*, const char * *partSeparator* = `0`, const char * *keyword* = `0`, const unsigned int *timeout* = `10000`)

Parse incoming answers.

This is used instead of **readMsg()** in applications where the answers always have a certain standard format and you want to

- split the answer data into individual messages,
- then split each message into separate message parts (e.g. several return values).

Parameters:

endOfMsg	string that marks the end of a message (could be a CR or a LF character
partSeparator	string that separates different parts within a message
keyword	optional string to consider only messages that contain 'keyword'. All other messages are ignored.
timeout	maximum time to wait for the message string

Returns:

0: no answers were detected within the specified timeout. 1...n: detected a message and it consists of (n) parts separated by the character(s) defined in the "separator" argument. -1: error while reading data

After **parseMsg()** returns with a value > 0, you can use **getMsgPart()** to read out the individual parts of the message (e.g. a returned measurement value, a CANopen object value).

See Also:

getMsgPart()

int FCommChannel::readMsg (char * *msgBuf*, const unsigned int *msgBufSize*, const char * *endOfMsg* = `0`)

read receive data

returns incoming data and removes it from the receive buffer.

Parameters:

msgBuf	character array where **readMsg()** can store the message data as a zero-terminated ASCII string.
msgBufSize	the size of msgBuf.
endOfMsg	a end-of-message token that can be used to split the received data into individual messages. If specified, **readMsg()** only takes the first message including the end-of-message string from the input buffer. If no end-of-message is found in the input buffer, readMsg returns zero and no message data. Using **readMsg()** without the endOfMsg argument would return everything that is left in the input buffer.

Returns:

the size of the message returned in msgBuf. 0: no more data is available (or there is data, but no separator/end-of-message character yet). -1: error while reading

int FCommChannel::sendMsg (const char * *msg*, const char * *answer1* = `0`, const char * *answer2* = `0`, const unsigned int *timeout* = `10000`)

send new ASCII command

This is the most basic command for sending out data, and - optionally - wait for an OK answer or an error message to come back.

Parameters:

msg	a null terminated 8-bit ASCII string with the message to send
answer1	same as in **waitForAnswer()**
answer2	same as in **waitForAnswer()**
timeout	same as in **waitForAnswer()**

Returns:

0: msg successfully sent but no answers were detected (or specified). 1: msg sent and answer1 came back 2: msg sent and answer2 came back -1: error while sending or reading data.

See Also:

waitForAnswer()

int FCommChannel::waitForAnswer (const char * *answer1* = `0`, const char * *answer2* = `0`, const unsigned int *timeout* = `10000`)

check incoming data and wait for answer

Parameters:

answer1	a null terminated 8-bit ASCII string which specifies a telegram or a token to wait for, e.g. an ACK / OK message.
answer2	alternative telegram/token to wait for, e.g. a NACK / ERROR message.
timeout	maximum time to wait for the message string

Returns:

0: no answers were detected within the specified timeout. 1: answer1 detected 2: answer2 detected -1: error while reading data.

waitForMsg() does not return any message data from the message buffer. Use **readMsg()** or **parseMsg()** for this purpose.

See Also:

readMsg()

Index