CMakeLists.txt

set project

```
cmake_minimum_required (VERSION 3.0)
project (cross)
```

set standard

```
set (CMAKE_CXX_STANDARD 17)
```

ask for system

```
if(CMAKE_SYSTEM_NAME STREQUAL "Linux")
        add_definitions(-DIS_LINUX)
elseif(CMAKE_SYSTEM_NAME STREQUAL "Windows")
        add_definitions(-DIS_WINDOWS)
else()
        add_definitions(-DIS_MAC)
endif()
```

add subdirectories

```
add_subdirectory(source)
```

add library

```
add_library(
        Communication
        SocketConnection.cpp
        serial.cpp
)
```

include directories

```
target_include_directories (Communication PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

add executable

```
add_executable(demo demo.cpp)
```

build type

```
if(${BUILD_TYPE} STREQUAL "Debug")
```

link libraries

```
target_link_libraries (
        demo
        LINK_PUBLIC
        Source
        Factory
        CustomLib
        Communication
        Collector
        wsock32
        ws2_32
        "${PROJECT_SOURCE_DIR}/winLibs/debug/ftdi1.lib"
        "${PROJECT_SOURCE_DIR}/winLibs/debug/libiconvStaticD.lib"
        "${PROJECT_SOURCE_DIR}/winLibs/debug/libusb-1.0.lib"
        "${PROJECT_SOURCE_DIR}/winLibs/debug/libxml2-static.lib"
)
```

add cpack

```
SET(CPACK_GENERATOR "ZIP")
SET(CPACK_DEBIAN_PACKAGE_MAINTAINER "david hasselhoff")
SET(CPACK_PACKAGE_NAME "demopack")
SET(CPACK_INCLUDE_TOPLEVEL_DIRECTORY "False")
INSTALL(TARGETS demo)
INCLUDE(CPACK)
```

add tests

```
enable_testing()
add_test(
        TEST1 catch_test
        COMMAND $<TARGET_FILE:testing> --success
)
```

running cmake

setup

```
> git clone https://github.com/duchonic/CleanMake.git
> mkdir build
> cd build
```

run

linux

```
> cmake -G "Eclipse CDT4 - Unix Makefiles"
        -DCMAKE_TOOLCHAIN_FILE=/usr/share/buildroot/toolchainfile.cmake -DBUILD_TYPE=Release
        -DCMAKE_ECLIPSE_VERSION=4.9 ~/CleanMake/
```

windows

```
> cmake -G "Visual Studio 16 2019" -A Win32 -DCMAKE_BUILD_TYPE=Debug ../CleanMake
```

mac

```
> cmake ../CleanMake
```

create a package

```
> cpack
```

pthread vs thread

includes

```
#ifdef IS_LINUX
        #include <pthread.h>
#elif IS_WINDOWS
        #include <thread>
#endif
```

create

posix

- **thread** An opaque, unique identifier for the new thread returned by the subroutine.
- **attr** An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
- **start_routine** The C++ routine that the thread will execute once it is created.
- **arg** A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

c++11

- **f** Callable object to execute in the new thread
- **args...*** arguments to pass to the new function

```
bool AppDriver::measure(){
        ...
                g_closeThread = false;

#ifdef IS_LINUX
                // start thread that check for LastTrigger
                pthread_t thread_id;
                pthread_attr_t attr;
                pthread_attr_init(&attr);
                pthread_create(&thread_id, &attr, &checkForAck, &isLastTrigger);
#elif IS_WINDOWS
                // start thread that check for LastTrigger
                std::thread thread_id(&checkForAck, &isLastTrigger);
#endif
```

thread

```
static void* checkForAck(void *hasAck);
static void* checkForAck(void* hasAck)
{
        while (!(*((bool*)(hasAck)))) {
                *((bool*)(hasAck)) = SyncModuleAppDriver::getInstance()->isLastTrigger();
                if (g_closeThread)
                        break;
#ifdef IS_LINUX
                SLEEP_ms(1);
#elif IS_WINDOWS
                std::this_thread::sleep_for(std::chrono::milliseconds(1));
#endif
        }
        return hasAck;
}
```

join

posix

The pthread_join() subroutine blocks the calling thread until the specified 'threadid' thread terminates.
When a thread is created, one of its attributes defines whether it is joinable or detached.
Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

c++11

Blocks the current thread until the thread identified by *this finishes its execution.
The completion of the thread identified by *this synchronizes with the corresponding successful return from join().
No synchronization is performed on *this itself.
Concurrently calling join() on the same std::thread object from multiple threads constitutes a data race that results in undefined behavior.

```
        g_closeThread = true;
#ifdef IS_LINUX
        pthread_join(thread_id, 0);
#elif IS_WINDOWS
        thread_id.join();
#endif
```

serial

includes

```
#ifdef IS_LINUX
        #include <termios.h> /* POSIX terminal control definitions */
#endif

#ifdef IS_WINDOWS
        #include <ftd2xx/ftd2xx.h>
#endif
```

init

posix

```
        std::ostringstream ss;
        //if(_comPort == COM_3){
        //      ss << "/dev/ttyS0";
        //}else{
                ss << DEV_FILE << _comPort - 1;
        //}

                _fd = open(ss.str().c_str(), O_RDWR | O_NOCTTY | O_SYNC);
        if (_fd == -1){
                /*
                * Could not open the port.
                */
                throw std::runtime_error("open_port: Unable to open " + ss.str());
        }else{
                //unsigned long opt = 1;
                //ioctl(fd, FIONBIO, opt);  // dont block on read
                //fcntl(fd, F_SETFL, 0);
        }
```

ftd2xx

```
FT_HANDLE ftHandle;
ftStatus = FT_ListDevices(&numDevs, NULL, FT_LIST_NUMBER_ONLY);
for (int id = 0; id < numDevs; id++) {
        ...
    ftStatus = FT_Open(id, &ftHandle);
}
```

write

posix

```
int SerialConnection::writeCom(const char* pBuffer, const int iSize)
{
        int n = write(_fd, pBuffer, iSize);
        return n;
}
```

ftd2xx

```
int SerialConnection::writeCom(const char* pBuffer, const int iSize)
{
        DWORD  bytesWritten = 0;
        DWORD txBytes = 0;
        DWORD rxBytes = 0;
        DWORD eventStatus = 0;

        FT_GetStatus(ftHandle, &rxBytes, &txBytes, &eventStatus);
        assert(ftStatus == FT_OK);
        // always call FT_GetStatus before FT_Read or else FT_Read will not work properly

        ftStatus = FT_Write(ftHandle, (char *)pBuffer, iSize, &bytesWritten);
        assert(ftStatus == FT_OK);
        return bytesWritten;
}
```

read

posix

```
int SerialConnection::readCom(char* pBuffer, const int iSize)
{
        int n = read(_fd, pBuffer, iSize);
        if(n == -1 && errno == EAGAIN)
                n = 0;
        return n;
}
```

ftd2xx

```
int SerialConnection::readCom(char* pBuffer, const int iSize)
{
        DWORD readBytes = 0;
        DWORD txBytes = 0;
        DWORD rxBytes = 0;
        DWORD eventStatus = 0;

        FT_GetStatus(ftHandle, &rxBytes, &txBytes, &eventStatus);
        assert(ftStatus == FT_OK);
        // always call FT_GetStatus before FT_Read or else FT_Read will not work properly

        ftStatus = FT_Read(ftHandle, pBuffer, iSize, &readBytes);
        assert(ftStatus == FT_OK);

        return readBytes;
}
```

control

posix

```
int SerialConnection::nrOfBytesAvailable()
{
        int bytesAvailable;
        ioctl(_fd, FIONREAD, &bytesAvailable);
        return bytesAvailable;
}

void SerialConnection::setNumberOfDataBits(char cNr)
{
        _options.c_cflag &= ~CSIZE;
        switch(cNr){
                case 5: _options.c_cflag |= CS5;
                                break;
                case 6: _options.c_cflag |= CS6;
                                break;
                case 7: _options.c_cflag |= CS7;
                                break;
                case 8: _options.c_cflag |= CS8;
                                break;
                default:
                        _options.c_cflag |= CS8;
        }
}
```

ftd2xx

```
int SerialConnection::nrOfBytesAvailable()
{
        DWORD rxBytes;
        DWORD txBytes;
        DWORD eventDWord;
        ftStatus = FT_GetStatus(ftHandle, &rxBytes, &txBytes, &eventDWord);
        assert(ftStatus == FT_OK);
        return rxBytes;
}

void SerialConnection::setNumberOfDataBits(char cNr)
{
        switch (cNr) {
                case 5: {
                        WordLength = 5;
                        break;
                }
                case 6: {
                        WordLength = 6;
                        break;
                }
                case 7: {
                        WordLength = FT_BITS_7;
                        break;
                }
                case 8: {
                        WordLength = FT_BITS_8;
                        break;
                }
                default: {
                        WordLength = FT_BITS_8;
                }
        }
        ftStatus = FT_SetDataCharacteristics(ftHandle, WordLength, StopBits, Parity);
        assert(ftStatus == FT_OK);
}
```

socket

includes

linux

sys/socket.h

```
int socket(int domain, int type, int protocol);
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
int listen(int socket, int backlog);
int accept(int socket, struct sockaddr *address, socklen_t *address_len);
ssize_t recv(int socket, void *buffer, size_t length, int flags); // flags : MSG_PEEK, MSG_OOB, MSG_WAITALL
//ssize_t send(int socket, const void *message, size_t length, int flags);
```

unistd.h

```
ssize_t write(int fd, const void *buf, size_t count);
```

windows

winsock2.h

```
SOCKET WSAAPI socket(int af,int type,int protocol);
int WSAAPI bind(SOCKET s,const sockaddr *name, int namelen);
int WSAAPI listen(SOCKET s, int backlog);
SOCKET WSAAPI accept(SOCKET s, sockaddr *addr, int *addrlen);
int WSAAPI recv(SOCKET s, char *buf, int len, int flags); // flags: MSG_PEEK, MSG_OOB, MSG_WAITALL
int WSAAPI send(SOCKET s, const char *buf, int len, int flags);
```

```
#ifdef IS_LINUX
        #include <sys/socket.h>
        #include <sys/ioctl.h>
        #include <arpa/inet.h>
        #include <net/if.h>
        #include <netinet/tcp.h>
        #include <netinet/in.h>
        #include <unistd.h>
#endif

#ifdef IS_WINDOWS
        #define WIN32_LEAN_AND_MEAN
        #include <windows.h>
        #include <winsock2.h>
        #include <ws2tcpip.h>
        #define DEFAULT_PORT "27015"
#endif
```

initialize

```
#ifdef IS_WINDOWS
        struct addrinfo* result = NULL;
        struct addrinfo hints;
        // Declare and initialize variables
        WSADATA wsaData;
        // Initialize Winsock
        WSAStartup(MAKEWORD(2, 2), &wsaData);
        ZeroMemory(&hints, sizeof(hints));
        hints.ai_family = AF_INET;
        hints.ai_socktype = SOCK_STREAM;
        hints.ai_protocol = IPPROTO_TCP;
        hints.ai_flags = AI_PASSIVE;

        // Resolve the local address and port to be used by the server
        getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
#endif

    _socket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

#ifdef IS_WINDOWS
        assert(_socket != INVALID_SOCKET);
#else
        assert(_socket != -1);
#endif

        memset(&_pRemoteAddress, 0, sizeof(_pRemoteAddress));   /* Clear struct */
        _pRemoteAddress.sin_family = AF_INET;                                   /* Internet/IP */
        _pRemoteAddress.sin_addr.s_addr = htonl(INADDR_ANY);    /* Incoming addr */
        _pRemoteAddress.sin_port = htons(PORT_NBR);                             /* server port */
        bind(_socket, (struct sockaddr *) &_pRemoteAddress, sizeof(_pRemoteAddress));
```

connect

```
bool SocketConnection::connect()
{
        if (listen(_socket, MAXPENDING) < 0) {
                DEBUG_LOG("socket error: " << listen(_socket, MAXPENDING));
#ifdef IS_WINDOWS
                DEBUG_LOG("errno: " << WSAGetLastError());
#endif
                return false;
        }

        #ifdef IS_WINDOWS
                int clientlen = sizeof(_pClientAddress);
        #else
                unsigned int clientlen = sizeof(_pClientAddress);
        #endif

        DEBUG_LOG("Waiting for connection...");
        if ((_iSocketStream = accept(_socket, (struct sockaddr *) &_pClientAddress, &clientlen)) < 0) {
                return false;
        }

#ifdef IS_WINDOWS
        //------------------------
        // Set the socket I/O mode: In this case FIONBIO
        // enables or disables the blocking mode for the
        // socket based on the numerical value of iMode.
        // If iMode = 0, blocking is enabled;
        // If iMode != 0, non-blocking mode is enabled.
        u_long iMode = 1;
        ioctlsocket(_iSocketStream, FIONBIO, &iMode);
#endif

        INFO_LOG("Connected to at " << inet_ntoa(_pClientAddress.sin_addr));
        return true;
}
```

disconnect

```
void SocketConnection::disconnect()
{
#ifndef IS_WINDOWS
        close(_iSocketStream);
        close(_socket);
#else
        closesocket(_iSocketStream);
        closesocket(_socket);
#endif
        INFO_LOG( "socket disconnected" );
}
```

send

```
bool SocketConnection::sendData(const std::string& strData)
{
#ifndef IS_WINDOWS
        ssize_t rc = write(_iSocketStream, strData.c_str(), strData.size() + 1);
#else
        int rc = send(_iSocketStream, strData.c_str(), strData.size() + 1, 0);
#endif
        return rc >= 0;
}
```

canReceive

```
bool SocketConnection::canReceiveData()
{
        char buffer;

#ifdef IS_WINDOWS
        int size = recv(_iSocketStream, &buffer, 1, MSG_PEEK);
        if(size < 0 && errno != EAGAIN){
                //ERROR_LOG("Lost connection to SYS size:" << size  << " errono: " << errno);
                //raise(SIGTERM); // raise SIGTERM to make a proper shutdown
                //return false;
        }

#elif IS_LINUX
        ssize_t size = recv(_iSocketStream, &buffer, 1, MSG_DONTWAIT | MSG_PEEK);
        if (size < 0 && errno != EAGAIN) {
                ERROR_LOG("Lost connection to SYS");
                raise(SIGUSR1); // raise SIGUSR1 to make a proper shutdown
                return false;
        }
#endif

        if ( (_msgEnd >= 0) || (size > 0)) {
                return true;
        }
        return false;
}
```

receive

```
bool SocketConnection::receiveData(std::string& strData)
{
        char* endTag = 0;
        strData.clear();

        while(!endTag){

#ifdef IS_WINDOWS
                int size = recv(_iSocketStream, &(_buffer[_msgEnd + 1]),
                                BUFF_SIZE - (_msgEnd + 1), 0);
                if (size < 0 && errno != EAGAIN && errno != EWOULDBLOCK && errno != ENOENT) {
                        //ERROR_LOG("Lost connection to SYS size:" << size << " errono: " << errno);
                        //raise(SIGTERM); // raise SIGTERM to make a proper shutdown
                        //return false;
                }

#elif IS_LINUX
                ssize_t size = recv(_iSocketStream, &(_buffer[_msgEnd + 1]),
                                BUFF_SIZE - (_msgEnd + 1), MSG_DONTWAIT);
                if (size < 0 && errno != EAGAIN && errno != EWOULDBLOCK && errno != ENOENT) {
                        ERROR_LOG("Lost connection to SYS");
                        raise(SIGUSR1); // raise SIGUSR1 to make a proper shutdown
                        return false;
                }
#endif

                if(size < 0)
                        size = 0;
                _buffer[_msgEnd + size + 1] = '\0';
                endTag = strstr(_buffer, MESSAGE_END);
                if(endTag){
                        // whole message received
                        strData.append(_buffer, endTag + strlen(MESSAGE_END));

                        // if we read too much copy it to the front of the buffer
                        int i = 0;
                        char* p = endTag + strlen(MESSAGE_END);
                        while(p != &(_buffer[_msgEnd + 1]) + size){
                                _buffer[i] = *p;
                                ++p;
                                ++i;
                        }
                        _msgEnd = i - 1;
                }else{
                        _msgEnd += size;
                }

        }
        return true;
}
```