Cameron's networking library(TCP)

Note: This library uses the library zlib, and I am not, nor do I claim to be the writer of it.

Index:

Classes

Network:

```
CAMSNETLIB int InitializeNetworking();
```

Initializes networking.

Remarks:
- Must be called before you can do anything(lol)

Return value:
- Zero:
  - When function succeeds.
- Non Zero:
  - See WSAStartup.

```
CAMSNETLIB int CleanupNetworking();
```

Cleans up networking.

Return value:
- Zero:
  - When function succeeds.
- Non Zero:
  - See WSACleanup.

Remarks:
- Must be called for every InitializeNetworking call.

Client:

```
CAMSNETLIB TCPClientInterface* CreateClient(cfunc msgHandler, dcfunc
disconFunc, UINT maxSendOps = 5, const BufferOptions& buffOpts =
BufferOptions(), const SocketOptions& sockOpts = SocketOptions(), UINT
olCount = 10, UINT sendBuffCount = 8, UINT sendCompBuffCount = 2, UINT
sendMsgBuffCount = 2, float keepAliveInterval = 30.0f, void* obj = nullptr);
```

Parameters:
- msgHandler – Pointer to a function with the following signature:
  - `void MsgHandler(TCPClientInterface& client, MsgStreamReader streamReader)`
  - This is where all packets are received/posted.

- disconFunc – Pointer to a function with the following signature:
  - void function(TCPClientInterface& client, bool unexpected)
  - Function called when you get disconnected from server.
- [optional] maxSendOps
  - Maximum number of concurrent send operations.
- [optional] buffOpts – See BufferOptions
- [optional] sockOpts – See SocketOptions
- [optional] olCount - Initial overlapped count.
- [optional] sendBuffCount - Initial amount of default size buffers dedicated to a standard send. Must be >= 1.
- [optional] sendCompBuffCount - Initial amount of buffers dedicated to sending compressed data. Must be >= 1.
- [optional] sendMsgBuffCount - Initial amount of buffers dedicated to sendmsg. Must be >= 1.
- [optional] keepAliveInterval – Interval at which client sends keep alive packets to server, used to detect half dropped connections.
  - A value of 0.0f is treated as no keepalive packets sent.
- [optional] Obj - Pointer to a class object, that is passed to the msghandler function; it is mostly used in oop.

Creates, and initializes a client object.

Remarks:
- Must be called before you can do anything.
- A call to DestroyClient is required.

Return value:
- TCPClientInterface*

```
CAMSNETLIB void DestroyClient(TCPClientInterface*& client);
```

Destroys the specified client object.

```
virtual bool Connect(const LIB_TCHAR* dest, const LIB_TCHAR* port, bool ipv6
= false, float timeOut = 5.0f) = 0;
```

Parameters:
- dest - IP address or hostname of server.
- port – Port server is listening on.
- ipv6 – Specifies if address pointed to by dest is an ipv6 address.
- timeOut – period of time in which function will timeout if a successful connection has not been made.

Attempts to connect to the destination ip address.
To connect to a hostname use Socket::HostNameToIP.

It waits/blocks until either:
1) A successful connection has been established.
2) Timeout period has expired.

Return value:
- True:
  - When function succeeds.
- False:
  - If client is already connected, or function fails.

```
virtual void Shutdown() = 0;
```

Immediately shuts down connection to server, and performs cleanup.
It waits/blocks until function returns.

Remarks:
- May not be called in Disconnect handler or MsgHandler.

```
virtual void Disconnect() = 0;
```

Disconnects client from server.
It does not wait/block.

```
virtual bool RecvServData(DWORD nThreads = 2, DWORD nConcThreads = 1) = 0;
```

Initializes socket, and receiving thread and starts receiving data from server.

Parameters:
- nThreads – Number of threads that wait on IOCP packets to dequeue.
  - This is not the same as nConcThreads, as if the thread sleeps or waits and another thread will be awoken.
  - Should be <= number of processors on your machine.
  - Must be >= nConcThreads.
- nConcThreads – Maximum number of concurrent threads.

Remarks:
- Must be called before you can send any type of data to server.

Return value:
- True:
  - When function succeeds.
- False:
  - If client is not connected.
  - Receive thread fails to create.

```
virtual bool SendServData(const BuffSendInfo& buffSendInfo, DWORD nBytes) =
0;
```

Parameters:

- buffSendInfo – buffer info
  - Must be value returned by GetSendBuffer.
- nBytes – number of bytes to send.

Sends data to connected server.

```
virtual bool SendServData(const MsgStreamWriter& streamWriter) = 0;
```

Parameters:

- streamWriter – must be returned by CreateOutStream.

```
virtual void SendMsg(char type, char message) = 0;
virtual void SendMsg(const std::tstring& user, char type, char message) = 0;
```

Sends a message to connected server in the format of TYPE, MESSAGE.
These functions are wrappers to SendServData.
Data is not compressed when calling these functions.

```
virtual void SetFunction(cfunc function) = 0;
```

Sets the clients function/message handler, it is called whenever a message is received.

```
virtual bool IsConnected() const = 0;
```

Return value:

- True:
  - If connected.
- False:
  - If not connected.

```
virtual UINT GetOpCount() const = 0;
```

Returns current overlapped operation count as an atomic operation.

```
virtual UINT GetMaxSendOps() const = 0;
```
Returns max number of send operations + 1, you passed to constructor.

```
virtual bool ShuttingDown() const = 0;
```

Returns true if shutting down, false otherwise as an atomic operation.

```
virtual const SocketOptions GetSockOpts() const = 0;
```

Returns SocketOptions structure you passed to constructor.

```
virtual const Socket GetHost() = 0;
```

Returns connected socket.

```
virtual void* GetObj() const = 0;
```

Returns a pointer to the object you specified in constructor.

Return value:

- void*

Server:

```
CAMSNETLIB TCPServInterface* CreateServer(sfunc msgHandler, ConFunc conFunc, DisconFunc disFunc, UINT maxPCSendOps = 5, const BufferOptions& buffOpts = BufferOptions(), const SocketOptions& sockOpts = SocketOptions(), UINT singleOlPCCount = 10, UINT allOlCount = 30, UINT sendBuffCount = 35, UINT sendCompBuffCount = 15, UINT sendMsgBuffCount = 10, UINT maxCon = 20, float keepAliveInterval = 30.0f, void* obj = nullptr);
```

Parameters:

- msgHandler - Pointer to a function with the following signature:
  - void MsgHandler(TCPServInterface& serv, ClientData* const client, MsgStreamReader streamReader)
  - This is where all packets are received.
- conFunc - Pointer to a function with the following signature:
  - void function(TCPServInterface& serv, ClientData* data)
  - Called after client is added to server.
- disFunc - Pointer to a function with the following signature:
  - void function(TCPServInterface& serv, ClientData* data, bool unexpected)
  - Called after client is removed from server.
  - If !data->pc.IsConnected() then either server is shutting down, or client had read errors, which is a good sign of an unauthorized client.
- [optional] maxPCSendOps.
  - Maximum number of concurrent single send operations per client(this option only applies for sending to a single client).
- [optional] buffOpts – See BufferOptions
- [optional] sockOpts – See SocketOptions

- [optional] singleOlPCCount - Initial overlapped count per client. Must be >= 1.
- [optional] allOlCount – Initial overlapped count for sending to multiple clients. Must be >= 1.
- [optional] sendBuffCount - Initial amount of default size buffers dedicated to a standard send. Must be >= 1.
- [optional] sendCompBuffCount - Initial amount of buffers dedicated to sending compressed data. Must be >= 1.
- [optional] sendMsgBuffCount - Initial amount of buffers dedicated to sendmsg. Must be >= 1.
- [optional] maxCon – The maximum amount of clients the server can support before it automatically starts rejecting clients.
- [optional] keepAliveInterval – The frequency the server sends keep alive messages to connected clients, to keep them from timing out.
    - A value of 0.0f is treated as no keepalive packets sent.
- [optional] Obj - Pointer to a class object, that is passed to the msghandler function; it is mostly used in oop.

Creates, and initializes a server object.

Remarks:

- Must be called before you can do anything(lol)
- A call to DestroyServer is required.

Return value:

- TCPServInterface*

```
CAMSNETLIB void DestroyServer(TCPServInterface*& server);
```

Destroys the specified server object.

```
virtual IPv AllowConnections(const LIB_TCHAR* port, ConCondition
connectionCondition, DWORD nThreads = 8, DWORD nConcThreads = 4, IPv ipv =
ipv4, UINT nConcAccepts = 4) = 0;
```

Parameters:

- port – Port to listen on.
- connectionCondition – function with the following signature:
    - bool function(TCPServInterface& serv, const Socket& socket);
- nThreads – Number of threads that wait on IOCP packets to dequeue.
    - This is not the same as nConcThreads, as if the thread sleeps or waits and another thread will be awoken.
    - Should be <= number of processors on your machine.
    - Must be >= nConcThreads.
- nConcThreads – Maximum number of concurrent threads.

- ipv – Ipv version of your address to listen on.
  - If you only have an ipv4 address pass ipv4.
- nConcAccepts – number of concurrent accept operations per IPv.
  - The higher this value is the faster new connections can be made under high connection/disconnection load.

Allows connections on the specified port.

Remarks:
- Must be called before you can send any type of data to server.

Return value IPv:
Returns any combination of any of the following:
- Ipv4:
  - If ipv4 was successfully binded.
- Ipv6:
  - If ipv6 was successfully binded.
- Ipnone:
  - If function has already been called.
  - If nothing was binded successfully.
  - Function fails.

```
virtual bool SendClientData(const BuffSendInfo& buffSendInfo, DWORD nBytes,
ClientData* exClient, bool single) = 0;
virtual bool SendClientData(const BuffSendInfo& buffSendInfo, DWORD nBytes,
ClientData** clients, UINT nClients) = 0;
virtual bool SendClientData(const BuffSendInfo& buffSendInfo, DWORD nBytes,
std::vector<ClientData*>& pcs) = 0;
```

Parameters:
- buffSendInfo – buffer info
  - Must be value returned by GetSendBuffer.
- nBytes – number of bytes to send.

```
virtual bool SendClientData(const MsgStreamWriter& streamWriter, ClientData*
exClient, bool single) = 0;
virtual bool SendClientData(const MsgStreamWriter& streamWriter, ClientData**
clients, UINT nClients) = 0;
virtual bool SendClientData(const MsgStreamWriter& streamWriter,
std::vector<ClientData*>& pcs) = 0;
```

Parameters:
- streamWriter – must be returned by CreateOutStream.

Sends data to specified clients.

Remarks:

- Handle must be closed after this is called either with WaitAndCloseHandle, or CloseHandle.
- First overload, the value of single determines what the function does
  - If single is true it sends only to address specified.
  - If single is false, and addr is not connected, it sends to all clients currently connected to the server.
  - If single is false, and addr is connected, it sends to all clients, excluding the addr specified.

Return value:
- bool – true if success, false otherwise.

```
virtual void SendMsg(ClientData* exClient, bool single, short type, short
message) = 0;
virtual void SendMsg(ClientData** clients, UINT nClients, short type, short
message) = 0;
virtual void SendMsg(std::vector<ClientData*>& pcs, short type, short
message) = 0;
virtual void SendMsg(const std::tstring& user, short type, short message) =
0;
```

Sends a message to specified clients in the format of TYPE, MESSAGE.
These functions are wrappers to SendClientData.
Data is not compressed when calling these functions.

```
virtual ClientData* FindClient(const std::tstring& user) const = 0;
```

Return value:
- A pointer to the ClientData, specified by user.

```
virtual void DisconnectClient(ClientData* client) = 0;
```

Disconnects connected client on the server.

```
virtual void Shutdown() = 0;
```

Immediately shuts down all connections to server, and performs cleanup.
It waits/blocks until function returns.

Remarks:
- May not be called in Disconnect handler, Connect handler, or MsgHandler.

```
virtual ClientAccess GetClients() const = 0;
```

Return value:

- ClientAccess object that can be used to access the array of server clients.

```
virtual USHORT ClientCount() const = 0;
```

Return value:

- Returns the number of connected clients.

```
virtual UINT MaxClientCount() const = 0;
```

Return value:

- Returns maxCon you passed to constructor.

```
virtual bool MaxClients() const = 0;
```

Return value:

- True:
  - If number of connected clients is at the maximum number of clients.
- False:
  - If number of connected clients is less than maximum clients.

```
virtual bool IsConnected() const = 0;
```

Return value:

- True:
  - If listening socket has been binded.
- False:
  - If listening socket has not been binded.

```
virtual bool ShuttingDown() const = 0;
```

Return value:

- Returns if server is shutting down as an atomic operation.

```
virtual const Socket GetHostIPv4() const = 0;
```

Return value:

- Returns ipv4 host socket, if was binded in AllowConnections.

```
virtual const Socket GetHostIPv6() const = 0;
```

Return value:

- Returns ipv6 host socket, if was binded in AllowConnections.

```
virtual UINT SingleOlPCCount() const = 0;
```

Return value:
- Returns initial overlapped count per client.

```
virtual UINT GetMaxPcSendOps() const = 0;
```
Return value:
- Max number of per client send operations + 1, you passed to constructor.

```
virtual UINT GetOpCount() const = 0;
```

Return value:
- Gets the current overlapped operation count as an atomic operation.

```
virtual const SocketOptions GetSockOpts() const = 0;
```

Returns SocketOptions structure you passed to constructor.

```
virtual void* GetObj() const = 0;
```

Returns a pointer to the object you specified in constructor.

Return value:
- Void*

StreamAllocInterface

```
virtual BuffSendInfo GetSendBuffer(DWORD hiByteEstimate, CompressionType
compType = BESTFIT) = 0;
```

Creates a send buffer for use for sending data.

Parameters:
- hiByteEstimate – High byte estimate for sending including MSG_OFFSET, your data size may not be greater than this when sending or you will get an access violation.
- compType – Desired compression type.

Remarks:
- When you call this function you must pass the value returned to Send…Data or you will potentially create a memory leak.
- This function is highly optimized to reduce contention improve alignment and optimize allocation time, and is highly recommended you use default server allocators.

```
virtual BuffSendInfo GetSendBuffer(BuffAllocator* alloc, DWORD nBytes,
CompressionType compType = BESTFIT) = 0;
```

Creates a send buffer for use given the allocator specified for sending data.

Parameters:

- alloc – user defined allocator.
- nBytes – High byte estimate for sending including MSG_OFFSET, your data size may not be greater than this when sending or you will get an access violation.
- compType – Desired compression type.

Remarks:

- When you call this function you must pass the value returned to Send…Data or you will potentially create a memory leak.

```
virtual MsgStreamWriter CreateOutStream(DWORD hiByteEstimate, short type,
short msg, CompressionType compType = BESTFIT) = 0;
```

Wraps GetSendBuffer(DWORD, CompressionType) to create a StreamWriter.

```
virtual MsgStreamWriter CreateOutStream(BuffAllocator* alloc, DWORD nBytes,
short type, short msg, CompressionType compType = BESTFIT) = 0;
```

Wraps GetSendBuffer(BuffAllocator* , DWORD, CompressionType) to create a StreamWriter.

```
virtual const BufferOptions GetBufferOptions() const = 0;
```

Return Value:

- Returns the buffer options object stored in the class.

Socket

```
Socket(const LIB_TCHAR* port, bool ipv6 = false);
```

Binds socket with the port specified.

Parameters:

- Port – port on which socket listens for incoming connections.
- Ipv6 – specifies if host is to listen using IPv6 address.

```
Socket(SOCKET pc);
```

Creates a socket with value from pc.

Parameters:

- pc – SOCKET in which to copy to.

```
void SetSocket(SOCKET pc);
```

Sets sockets value to pc.

```
SOCKET GetSocket() const;
```

Returns SOCKET associated with object.

```
bool Bind(const LIB_TCHAR* port, bool ipv6 = false);
```

Binds socket with the port specified.

Parameters:

- Port – port on which socket listens for incoming connections.
- Ipv6 – specifies if host is to listen using IPv6 address.

```
Socket AcceptConnection();
```

Waits for incoming connection.

Return Value:

- Valid Socket:
    - If successful connection is established.
- Socket()
    - Internal error has occurred.

```
bool Connect(const LIB_TCHAR* dest, const LIB_TCHAR* port, bool ipv6, float
timeout);
```

Parameters:

- dest – IP address or hostname to connect to.
- port – port on which to establish connection
- ipv6 – specifies if dest points to a ipv6 address.
- timeout – timeout before function returns.

Return Value:

- True:
  - A successful connection has been established.
- False
  - Failure to establish connection.
  - Timeout period has occurred.
  - Internal error has occurred.

```
void Disconnect();
```

Forces destruction of all instances of the socket.

```
long ReadData(void* dest, DWORD nBytes);
```

Reads nBytes data from socket. This function waits until nBytes have been received.

Parameters:

- dest – pointer to buffer to receive data.
- nBytes – number of bytes to read.

Return Value:

- > 0:
  - Number of bytes function received, it is always equal to nBytes.
- = 0:
  - Connection has gracefully closed.
- = SOCKET_ERROR(-1)
  - Connection has ungracefully closed.

```
long SendData(const void* data, DWORD nBytes);
```

Sends nBytes data to socket. This function waits until nBytes have been sent.

Parameters:
- dest – pointer to data to send.
- nBytes – number of bytes to send.

Return Value:
- > 0:
  - Number of bytes function sent, it is always equal to nBytes.
- = SOCKET_ERROR(-1)
  - Connection has ungracefully closed.

```
bool AcceptOl(SOCKET acceptSocket, void* infoBuffer, DWORD localAddrLen,
DWORD remoteAddrLen, OVERLAPPED* ol);
```

Accepts a connection as a overlapped operation. See AcceptEx for more info.

```
long RecvDataOl(WSABUF* buffer, OVERLAPPED* ol, DWORD flags = 0, UINT
bufferCount = 1, LPWSAOVERLAPPED_COMPLETION_ROUTINE cr = NULL);
```

Recvs data as an overlapped operation. See WSARecv for more info.

```
long SendDataOl(WSABUF* buffer, OVERLAPPED* ol, DWORD flags = 0, UINT
bufferCount = 1, LPWSAOVERLAPPED_COMPLETION_ROUTINE cr = NULL);
```

Sends data as an overlapped operation. See WSASend for more info.

```
bool IsConnected() const;
```

Return Value:
- True:
  - Socket is not set to null or INVALID_SOCKET.
- False:
  - Socket is set to null or INVALID_SOCKET.

```
bool SetTCPRecvStack(int size = 0);
```

Sets the TCP recv size, which may or may not be equivalent to the to the TCP window size.

Return value:
- Returns true if success, false if fail.

```
bool SetTCPSendStack(int size = 0);
```

Sets the TCP send size, which may or may not be equivalent to the to the TCP window size.

Return value:
- Returns true if success, false if fail.

```
bool SetNoDelay(bool b = true);
```

Enables or disables the nagle algorithm.

Return value:
- Returns true if success, false if fail.

```
static bool SetAcceptExContext(SOCKET accept, SOCKET listen);
```

Sets accept socket's context to listens context.

Parameters:
- accept – accept socket passed to AcceptEx.
- listen – listen socket used for AcceptEx.

```
bool GetTCPRecvStack(int& size);
```

Retrieves the TCP recv size, which may or may not be equivalent to the to the TCP window size.

Return value:
- Returns true if success, false if fail.

```
bool GetTCPSendStack(int& size);
```

Retrieves the TCP send size, which may or may not be equivalent to the to the TCP window size.

Return value:
- Returns true if success, false if fail.

```
bool GetTCPNoDelay(bool& b);
```

Outputs if nagle algorithm is enabled in b.

Return value:
- Returns true if success, false if fail.

```
bool SetBlocking();
```

Changes socket to blocking mode.

```
bool SetNonBlocking();
```

Changes socket to non-blocking mode.

```
const SocketInfo& GetInfo();
```

Returns const reference to SocketInfo.

```
static int GetHostName(LIB_TCHAR* dest, DWORD nameLen);
```

Wraps around winsock func gethostname();

```
static char* Inet_ntot(in_addr inaddr, LIB_TCHAR* dest);
```

Wraps around winsock func inet_ntoa;

```
static std::tstring GetLocalIP(bool ipv6 = false);
```

Returns your local IP address.

Parameters:

- Ipv6 – specifies if function should return an ipv6 address.

Return Value:

- std::tstring()
  - Function fails.
- Valid string:
  - Function succeeds.

```
static std::tstring HostNameToIP(const LIB_TCHAR* host, bool ipv6 = false);
```

Converts host name into an IP address and returns it.

Parameters:

- host – pointer to hostname.
- ipv6 - specifies if buffer should receive an ipv6 address.

Return Value:

- std::tstring()
  - Function fails.
- Valid string:
  - Function succeeds.

```
static void GetAcceptExAddrs(void* buffer, DWORD localAddrLen, DWORD
remoteAddrLen, sockaddr** local, int* localLen, sockaddr** remote, int*
remoteLen);
```

Parses the remote and local addresses retrieved from AcceptEx into a usable form. See GetAcceptExSockaddrs for more info.

SocketInfo

```
void Cleanup();
```

Cleans up data allocated by class.

```
void SetAddr(sockaddr* addr, bool cleanup);
```

Parameters:

- addr – pointer to addr to copy from.
- cleanup – pass true if it was allocated with alloc and false if you are calling this with AcceptEx.

```
std::tstring GetPortStr() const;
```

Returns socket's port as a string.

```
USHORT GetPortInt() const;
```

Returns socket's port.

```
static std::tstring FormatIP(SockaddrU addr);
```

Formats IP address from SockaddrU union and returns it as a string.

```
std::tstring GetIp() const;
```

Returns socket's IP address.

```
bool CompareIp(const SocketInfo& rhs) const;
```

Compares current IP address with rhs's IP address. Note that this should be used when you do not need the ip address formatted as a string.

Return Value:

- True:
  - IP addresses are identical.
- False:
  - IP addresses are not identical.

```
bool IsIpv4() const;
```

Checks if address is an ipv4 address.

```
bool IsIpv6() const;
```

Checks if address is an ipv6 address.

```
SockaddrU GetSockAddr() const;
```

Returns SockaddrU containing pointer to sockaddr associated with socket, which can be accessed by other members if need casted

SocketOptions

```
SocketOptions(int tcpSendSize = 0, int tcpRecvSize = 65536, bool noDelay = true);
```

Parameters:

- tcpSendSize – Correlates to TCP send window size, but may not be equivalent. Also maximum size you can send in one call.
  - If value is zero, application should send using your buffers directly, instead of the TCP window.
    - Setting this value to 0 should result in an increase in performance if sending medium to large amounts of data.
    - Setting this value to 0 requires the application to lock the buffers you pass, accessing them after they are posted to be sent will result in an access violation or potentially a crash.
- tcpRecvSize – Correlates to TCP receive window size, but may not be equivalent.
  - If value is zero, If application should receive using underlying application buffers directly instead of the TCP window.
    - Setting this value to 0 requires the application to lock the buffers you pass, accessing them after they are posted to be sent will result in an access violation or potentially a crash.

        - Note setting this value to 0 could have much more serious performance problems than setting tcpSendSize to 0.
        - If the client is busy processing data, it will be unable to receive more data until it is finished.
    - Setting this value to 0 will decrease the throughput of each individual client. While increasing the total amount of clients that can be handled.
- noDelay – If application should override the nagle algorithm. Removes the delay of 200ms when sending TCP packets.
    - Note this option is overridden if tcpSendSize option is set to 0.
    - This option will generally improve performance if you send a lot of really small packets periodically.
        - If you need data to be sent out faster than that set this value to false.
            - A good reason to do this is for a server side game.
    - This means that the TCP window pools data sent to it and sends it out periodically 5 times per second.

```
int TCPSendSize() const;
```

Return value:

- Returns tcpSendSize value passed to constructor.

```
int TCPRecvSize() const;
```

Return value:

- Returns tcpRecvSize value passed to constructor.

```
bool NoDelay() const;
```

Return value:

- Returns noDelay value passed to constructor.

BufferOptions

```
BufferOptions(UINT maxDataBuffSize = 4096, UINT maxReadSize = 100000, int
compression = 9, int compressionCO = 512);
```

Parameters:

- maxDataBuffSize
    - Maximum data buffer size for a application buffer(pre-allocated optimized and aligned), also maximum size that can be received in one call.
        - This value should be an increment of page boundaries(4KB).
    - Note this value is not directly equal to buffer size.

    - Due to internals of the library it is (maxDataBuffSize – sizeof(size_t)) rounded up to the next page boundary.
- maxReadSize
    - Maximum amount of data that can be received in a single buffer before it is treated as a security threat.
- Compression
    - Compression value of 1-9 used to compress data if nBytes >= compressionCO
- compressionCO
    - Used to decided if to compress a send buffer.

```
UINT GetMaxDataBuffSize() const;
```

Return value:

- Returns actual maxDataBuffSize.

```
UINT GetMaxDataCompSize() const;
```

Return value:

- Returns calculated maximum compressed buffer size.

```
UINT GetMaxDataSize() const;
```

Return value:

- Returns actual maxDataSize.

Remarks:

In order to use an application optimized send buffer, your data size may not be greater than this.

```
UINT GetMaxReadSize() const;
```

Return value:

- Returns maxReadSize value passed to the constructor – sizeof(DataHeader).

```
int GetCompression() const;
```

Return value:

- Returns compression value you passed to the constructor.

```
int GetCompressionCO() const;
```

Return value:

- Returns compressionCO value you passed to the constructor.

```
int GetPageSize() const;
```

Return value:
- Returns retrieved page of the system.

BuffAllocator

In order to allocate data using your own allocation methods you must write a class and inherit from BuffAllocator and override alloc and dealloc.

MsgStream

```
MsgStream(char* data, UINT capacity)
```

Parameters:
- data – pointer to allocated data.
- Capacity – capacity of allocated data.

```
char GetType() const
```

Return value:
- Type

```
char GetMsg() const
```

Return value:
- Msg

```
UINT GetSize() const
```

Return value:
- Capacity of stream.

```
UINT GetDataSize() const
```

Return value:
- Capacity of stream excluding MSG_OFFSET.

```
template<typename T> std::enable_if_t<std::is_arithmetic<T>::value, T>
operator[](UINT index) const
```

Return value:

- Returns data at position index, of type of T.

```
bool End() const
```

Return value:

- True:
  - If you have reached end of stream.
- False:
  - If you have not reached end of stream.

MsgStreamWriter

```
MsgStreamWriter(const BuffSendInfo& buffSendInfo, UINT capacity, short type, short msg)
```

Parameters:

- buffSendInfo – buffer info, must be returned by GetSendBuffer.
- capacity – capacity of allocated data.
- type – type of msg.
- msg – msg.

```
template<typename T> void Write(const T& t)
template<typename T> MsgStreamWriter& operator<<(const T& t)
```

Writes data referenced by t to stream.

```
template<typename T>
std::enable_if_t<std::is_arithmetic<T>::value> Write(T* t, UINT count)
```

Writes count data of type T of data pointed to by t.

```
template<typename T>
std::enable_if_t<std::is_arithmetic<T>::value> WriteEnd(T* t)
```

Writes data pointed to by t, until end of stream is reached.

```
template<typename... T>
static UINT SizeType()
```

Returns the sum of the size of the types passed(arithmetic types only).

```
template<typename... T>
static UINT SizeType(const T&... t)
```

Returns the sum of the size of the objects passed.

MsgStreamReader

```
MsgStreamReader(char* data, UINT capacity)
```

Parameters:

- data – pointer to data, obtained from msgHandler
- capacity – capacity of allocated data, obtained from msgHandler(note this is excluding nBytes, so you must pass nBytes – MSG_OFFSET).

```
template<typename T> T Read()
```

Return value:

- Value of type T read in from stream.

```
template<typename T> void Read(T& t)

template<typename T> MsgStreamReader& operator>>(T& dest)
```

Reads data in from stream.

```
template<typename T>
std::enable_if_t<std::is_arithmetic<T>::value, T>* Read(UINT count)
```

Reads count T from stream and returns a pointer to this data.

```
template<typename T>
std::enable_if_t<std::is_arithmetic<T>::value, T>* ReadEnd()
```

Reads T until end of stream and returns a pointer to this data.

Defining own MsgStream specializations

The message stream classes cannot function correctly for any object/class that contains a pointer, or reference. This is due to the fact that pointers and references are relative to that instance of the program, and cannot be transferred. For this reason, Read, Write, WriteEnd, ReadEnd, operator[], and SizeType<typename…>() are restricted to arithmetic types.

However despite this limitation, you are able to freely specialize these functions to suit your needs through the use of class specialization. The system is setup as follows:

Let X represent your typename. Note that X may be a templated type, and can substitute with template parameters.

You define the following class and the following functions:

```
class StreamWriter::Helper<X> : public HelpBase<X>
{
public:
	Helper(StreamWriter& stream) : HelpBase(stream){}
	void Write(const X& t);
	static UINT SizeType(const X& t);
}
class StreamReader::Helper<X> : public HelpBase<X>
{
public:
	Helper(StreamReader& stream) : HelpBase(stream){}
	X Read();
};
```

The rest of the functions are simply wrappers around these functions such as operator<< or operator>>.

Within these functions you may use stream to access member functions of StreamReader and StreamWriter. This includes other class specializations. Note: you can also call functions from Helper<T>, though it is recommended you go through stream.

There are plenty of examples of MsgStream specialization in the streamext.h.

Other:

```
CAMSNETLIB void WaitAndCloseHandle(HANDLE& hnd);
```

Waits for the specified handle to be triggered, then closes the specified handle.

Server and Client auto-handle KeepAlive messages, and may be optionally handled accordingly in msgHandler.

Notice:

THIS IS A UNICODE BUILD ATTEMPTS TO USE MULTIBYTE/ASCII WILL RESULT IN A CRASH.

Examples

Note the following is pseudo code, and should only be used to understand the framework.

Client:

-Creating a basic client:

```
void MsgHandler(TCPClientInterface& clint, MsgStreamReader streamReader)
{
	char* dat = streamReader.GetData();
	const short type = streamReader.GetType(), msg = streamReader.GetMsg();
	switch (type)
	{
		case TYPE_CHANGE:
		{
			switch(msg)
			{
			case MSG_CHANGE_DISCONNECT:
			{
				//Notify user server is a client has disconnected
				break;
			}
			}
		}
		break;

		// Handle other cases

	}

}
```

```
void DisconnectHandler(TCPClientInterface& client, bool unexpected) // for disconnection
{
	if(unexpected)
	{
		// Notify user they have been disconnected
	}

	// Most likely do nothing because you caused the disconnection
}

InitializeNetworking();

TCPClientInterface* client = CreateClient(&MsgHandler, &DisconnectHandler);
bool res = client->Connect(L"ip", L"port number");
if (res)
{
	res = client->RecvServData();
	if (res)
	{
		//Ready to send packets
	}
}

CleanupNetworking();
```

-Sending packets from client to server:

```
//Sends the number 5 to the server, excluding msg_type, and msg
BuffSendInfo sendInfo = client->GetSendBuffer(nBytes + MSG_OFFSET);
*((short*) sendInfo.buffer) = TYPE_;
*((short*) sendInfo.buffer + 1) = MSG_;
client->SendServData(sendInfo, nBytes + MSG_OFFSET);

or

//Sends the number 5 to the server, including msg_type, and msg
auto streamWriter = clint.CreateOutStream(nBytes, TYPE_, MSG_)
streamWriter.Write(…);
client->SendServData(streamWriter);
```

Server:

-Creating a basic server:

```
bool CanConnect(TCPServInterface& serv, const Socket& socket)
{
	return true; //Possibly ip ban list?
}

void DisconnectHandler(TCPServInterface& serv, ClientData* data, bool unexpected)
```

```
{
	//Do whatever possibly log disconnections?
}

void ConnectHandler(TCPServInterface& serv, ClientData* data)
{
	//Do whatever possibly log connections?
}



//Handles all incoming packets
void MsgHandler(TCPServInterface& serv, ClientData* const clint, MsgStreamReader
streamReader)
{
	auto clients = serv.GetClients();
	const USHORT nClients = serv.ClientCount();

	char* dat = streamReader.GetData();
	const short type = streamReader.GetType(), msg = streamReader.GetMsg();
	//Switch type and msg for all your packets

}

InitializeNetworking();

//Optional port map on router
MapPort(port, L"TCP", L"Server");
TCPServInterface* serv = CreateServer(&MsgHandler, &ConnectHandler, &DisconnectHandler);
bool res = serv->AllowConnections(L”port”, CanConnect);
if (res)
{
	//Ready to send packets
}

CleanupNetworking();
```

-Sending packets from server to client:

```
BuffSendInfo bi = serv.GetSendBuffer(nBy + MSG_OFFSET);
((short*)bi.buffer)[0] = type;
((short*) bi.buffer)[1] = message;
serv.SendClientData(msgInfo, nBy + MSG_OFFSET, client, true);

Or

//Sends a clients’ username to another client
auto streamWriter = serv.CreateOutStream(StreamWriter::SizeType(clients->user), TYPE_,
MSG_);

streamWriter.Write(clients[i]->user);
serv.SendClientData(streamWriter, clint, true);
```