DIGILENT®
www.digilentinc.com

1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Introduction

This document describes the Digilent® chipKIT™ compatible Internet library. The focus for the development of this library was to provide a comprehensive Internet feature set for use in an embedded system that does not support a real time OS. In particular, this library was designed for the Digilent® chipKIT™ microcontroller product line. This includes the Digilent® Arduino hardware/software compatible products.

The model for this library is tailored to the Initialize/Loop type program structure where system tasks are typically executed once per each pass through the loop. As a result, all DNETcK methods are designed to return control to the loop quickly so other, potentially more critical, system tasks can be serviced. Also the methods are designed not to allocate any memory from the heap. This will eliminate any chance that the DNETcK library will consume, leak, or fragment the heap to the point of failure. In the few cases where additional memory is needed, the programmer is required to supply the buffer space to the library.

The DNETcK library is broken up into 3 functional units consisting of 5 classes. The 3 units are 1. Hardware initialization, ARP, DNS, DHCP, Time, and Stack Maintenance; 2. TCP; and 3. UDP; both TCP and UDP compose of two classes each, Client and Server. The Hardware initialization, ARP, DNS, DHCP, Time and Stack Maintenance is implemented in the static DNETcK class that should not be instantiated. Make direct references to the static methods off of the class name, for example DNETck::periodicTasks(). The remaining 4 classes TcpClient, TcpServer, UdpClient, and UdpServer must all be instantiated before use.

The TCP and UDP classes mirror each other with a remote/local endpoint pairing socket model for the client; and a listen/accept model for the server. For TCP, the client must make a connection to a remote endpoint. isConnected() is called to determine if the client is connected. For UDP, the client only needs to resolve a remote IP/Port endpoint and associate it with a local IP/Port endpoint for communication between the Remote/Local endpoints. isEndPointResolved() is called to determine if the endpoints have been setup for communications.

For the purposes of this document I will define an endpoint-pair as the association of a Remote IP/Port with a local IP/Port on a socket. For TCP, this is typically what makes up a connection, but for UDP this is just a conversation between endpoints with no explicit connection.

For both TCP and UDP the server classes are very similar. startListening() should be called to start listening on a server port and avaliableClients() should be called to see if any clients are waiting to be accepted. acceptClient() will initialize a TcpClient/UdpClient instance that has a socket set up for the Remote/Local endpoints. Once a client is accepted from the server, the server completely releases all management of that client to you. Only if you call close() on the client instance, releasing the underlying socket, could the remote client contact the server again to set up a new client instance. Closing the server will close all un-accepted clients still being managed by the server and stops the listening process; but it will not close any clients already accepted by you.

A major difference between TCP and UDP is that TCP creates a connection and establishes a reliable protocol in that data is not lost, the order of data transmission is retained, and that the connection looks like a continuous stream of data. Whereas UDP has no explicit connection, is an unreliable

protocol where data can be lost, the order of datagrams is not guaranteed, and data is sent in packets or datagrams rather than a stream of data. However, UDP does insure the integrity of each individual datagram, that is, the datagram is all or nothing; there is no concept of receiving a partial datagram. Because UDP is unreliable, users are required to implement their own reliability, which means the ability to re-request a lost packet. As a result, many stack implementations of UDP are very aggressive at throwing out datagrams if a new one comes in. Both the Microchip® MAL and the WIZnet W5100 are very aggressive at tossing the last received datagram when a new datagram comes in; even though there is plenty of memory to hold multiple datagrams in there socket buffer. This discarding of datagrams can be very annoying to the programmer as many datagrams may be tossed just because the programmer did not capture the datagram in time; even though there was plenty of memory to hold multiple datagrams. The DNETcK implementation will capture and cache multiple datagrams; however DNETcK needs a buffer to cache these datagrams. Both the UdpClient and UdpServer class constructors require the programmer to supply a buffer for this cache. Clearly, the bigger the buffer, the more datagrams can be cached. DNETcK puts a restriction that the buffer must be at least 32 bytes long. The UDP protocol restricts datagrams to be smaller than 64K. The Microchip® MAL can only handle datagrams up to 1536 bytes (by default). The WIZnet W5100 has an 8K receive buffer memory that must be shared across all sockets so there may be some flexibility on the maximum datagram size. However, Ethernet defines the maximum packet payload size of 1500 bytes, and may be the practical limit for many implementations. As a practical matter, it is less likely to see datagrams larger than 1500 bytes and this can help in determining the amount of memory to assign to the DNETcK UDP datagram caches. In determining how much space should be allocated for the UDP datagram cache, you should consider the maximum datagram size that you expect to receive, how many datagrams you expect to come in before you can process them, and how much RAM you have to allocate towards the cache. If the application is totally unknown, then a cache size of a few thousand bytes will usually work.

As mentioned earlier, the DNETcK Internet classes are designed to work well in a looping state machine type programming environment; with short and repetitive passes through the main loop. The DNETcK Internet implementation is considered to be just one of potentially many services that must be maintained in each pass of the main loop. For this reason, the DNETcK methods all can be specified to return quickly. But this also means that the Internet task at hand may not have completed within the call. DNETcK provides “is” functions that can be call repeatedly to determine if a longer task spanning several main loop interactions is complete or not. For example, let say you want to create a TCP connection to a remote host www.digilentinc.com. You would construct a TcpClient, call the connect() method to specify the remote hostname (www.digilentinc.com) and port and then call isConnected() each time through your main loop until the connection is successfully established. The connection may take a relative long time as a DNS lookup of the hostname must be done, as well as an ARP; and then you must wait for the remote client to accept and set up your connection.

The “is” methods are designed to be called repeatedly and will return false until their task is complete. Therefore a false does not mean that the method has failed, it usually just means that the method is not done completing its task. A true means the task has successfully completed and you can move on to the next step. However, if the “is” method does have a hard failure, a false will be returned. To determine if a false is a hard failure or not, call the “is” method specifying the optional status parameter and then pass that status to DNETcK:: isStatusAnError(). If DNETcK:: isStatusAnError() returns true, then the status is a hard error and the task has failed. If DNETcK:: isStatusAnError() returns false, then the status is just an indication of where it is in the task and no action is needed except to wait for the process to complete; that is, continue to call the “is” method until the task completes.

While the DNETcK “is” methods are internally implemented to return as quickly as possible; an optional msBlockMax parameter is provided to specify the maximum timeout period allowing the task time to complete before returning. This timeout can be anywhere from an immediate return (DNETcK:: msImmediate) to an infinite wait (DNETcK:: msInfinite). If no timeout period is specified, a default value of 15 seconds is used, that is, immediate is not the default setting. For ease of use reasons it was deemed more functional to use a default timeout period that would typically allow the task to complete rather than an immediate return. In general if the default timeout is used, a false return would most likely indicate that a hard error had occurred because sufficient time was given for most tasks to complete; however the status should be checked to be sure. A word of caution though, the only thing that is run during the block time is the Internet stack, if you have other tasks that need to be serviced, the 15 second default may (most likely) be excessive. Fortunately, the default timeout period can be changed with a call to DNETcK:: setDefaultBlockTime(); so if you wish the default time to be immediate, call DNETcK:: setDefaultBlockTime(DNETcK:: msImmediate) in the setup() call in your sketch; but remember to explicitly call the “is” method each time through the loop until the task is complete. The “is” methods can be called at any time and as often as you like.

Only “is” methods take timeout values except for one notable exception. TcpClient.writeStream() takes a timeout value as the write buffer may be larger than the underlying socket buffer. TcpClient.writeStream() will loop writing as many bytes to the socket as it can, flushing the socket, and then writing again until the buffer is completely transmitted. This can take time, maybe more time than you want and is why this method provides a timeout value. If TcpClient.writeStream() can’t write the whole buffer within the timeout period, it will return the number of bytes actually written and you can then call TcpClient.writeStream() at a later time to finish the write.

DNETcK Static Class

DNETcK is a static class that initializes the hardware, calls DHCP (if requested) to get IP information, and starts the IP stack. Once the stack is running, DNETck provides access to DNS, ARP, and Time services, along with some other miscellaneous functions. The IP stack can be completely stopped with the end() method. Once stopped, the stack can be reinitialize with a different configuration, and restarted again with the begin() method. Since DNETcK is a static class, it cannot be instantiated and its methods should be called directly off of the class name, e.g. DNETck:: begin().

Constants

unsigned long DNETcK::msImmediate

Has a value of 0 and is used to specify that a method should do as little as possible and to return to the caller as quickly as possible without waiting for the task to complete.

unsigned long DNETcK::msInfinite

Has a value of 0xFFFFFFFF and is used to specify that a method should block indefinitely and only return to the caller if the task has been completed, or until a hard error occurs.

MAC DNETcK::zMAC

Represents a zero filled MAC, this can be used as an initializer to a MAC variable.

IPv4 DNETcK::zIPv4;

Represents a zero filled IPv4, this can be used as an initializer to a IPv4 variable.

DNETcK::iReservedPort

Has a value of 0 and is an invalid port value and is used to specify to the IP Stack that an ephemeral should be automatically selected when setting up a local port.

DNETcK::iWellKnownPorts

Has a value of 1 and is the start of well-known port numbers. Well-known ports have a range from 1 – 1023; for example the http port 80 falls in the well-known port range.

DNETcK::iRebootPort

Has a value of 69 and is the Microchip reboot port number. A UDP datagram sent to this port will reboot the PIC32 processor. This conflicts with the typical usage for port 69 which is the UDP TFTP (Trivial File Transfer Protocol) port.

DNETcK::iRegisteredPorts = 1024;

Has a value of 1024 and is the start of the IANA registered ports. The IANA registered port range is from 1024 – 49151. Typically if you want to assign a port for your personal use you can usually pick

an un-assigned port in this range as few applications will randomly pick the same port; although possible. In theory, if you need a protected port number you would have the IANA register it for you. Many ports in this range are unofficial in that they have not been officially assigned by the IANA but are widely known and observed by convention.

DNETcK::iPersonalPorts35 = 35000
DNETcK::iPersonalPorts38 = 38000
DNETcK::iPersonalPorts39 = 39000
DNETcK::iPersonalPorts44 = 44000
DNETcK::iPersonalPorts45 = 45000
DNETcK::iPersonalPorts46 = 46000

All of these ports are within the IANA registered range but at the time of the writing of this document were not assigned and each starts a range of 1000 of unassigned ports. These are probably safe to use for personal use, but should be checked for uniqueness or registered with the IANA if critical.

DNETcK::iEphemeralPorts

Has a value of 49152 and is the start of ephemeral port dynamically assigned by the IP Stack to be used as temporary local port numbers. Unfortunately the Microchip MAL does not comply with this rule and assigns ports in the IANA range. The Microchip MAL assigns TCP ephemeral ports in the range of 1024 – 5000; and UDP ephemeral ports in the range of 4096 – 8192.

DNETcK::iMaxPort

Has a value of 0xFFFF and is a valid port number. It just marks the highest port number allowed.

Enums

DNETcK::STATUS

This is enum that represents the status of a task. Most status codes are self-explanatory and their values can be found in DNETcK.h. Not all status codes are errors. To determine if a particular status is a hard error call DNETck::isStatusAnError().

Structures

```
typedef union
{
   byte        rgbIP[4];
   uint32_t    u32IP;
} IPv4;
```

The IPv4 data type represents an IPv4 Internet Protocol 4 byte address. IPv6 addressing is not supported by DNETcK.

```
typedef struct
{
   byte rgbMAC[6];
} MAC;
```

The MAC data type represents a physical Media Access Control address.

```
typedef struct
{
   IPv4              ip;
   unsigned short    port;
} IPEndPoint;
```

The IPEndPoint data type represents an IPv4 address / port pair. This can represent either a remote or local endpoint.

Methods

```
bool DNETcK::begin(void)
bool DNETcK::begin(const IPv4& ip)
bool DNETcK::begin(const IPv4& ip, const IPv4& ipGateway)
bool DNETcK::begin(const IPv4& ip, const IPv4& ipGateway, const IPv4& subnetMask)
bool DNETcK::begin(const IPv4& ip, const IPv4& ipGateway, const IPv4& subnetMask,
                   const IPv4& ipDns1)
bool DNETcK::begin(const IPv4& ip, const IPv4& ipGateway, const IPv4& subnetMask,
                   const IPv4& ipDns1, const IPv4& ipDns2)
bool DNETcK::begin(const MAC& mac)
bool DNETcK::begin(const MAC& mac, const IPv4& ip)
bool DNETcK::begin(const MAC& mac, const IPv4& ip, const IPv4& ipGateway)
bool DNETcK::begin(const MAC& mac, const IPv4& ip, const IPv4& ipGateway,
                   const IPv4& subnetMask)
bool DNETcK::begin(const MAC& mac, const IPv4& ip, const IPv4& ipGateway,
                   const IPv4& subnetMask, const IPv4& ipDns1)
bool DNETcK::begin(const MAC& mac, const IPv4& ip, const IPv4& ipGateway,
                   const IPv4& subnetMask, const IPv4& ipDns1, const IPv4& ipDns2)
```

Parameters:

mac	The physical MAC address to use. If omitted the chipKIT assigned hardware MAC is used.
ip	The static IP address to use, if omitted, DHCP is used to get your IP address and other network settings.
ipGateway	The ip address of the gateway to use. If omitted ip[0]:ip[1]:0.1 is used. This is ignored if DHCP is used.
subnetMask	The subnet mask of the local network, if omitted 255.255.255.0 is used. This is ignored if DHCP is used.

ipDns1	1 of 2 DNS servers to use, if omitted ipGateway is used as the primary DNS server as some gateways will act as a DNS forwarder. This is ignored if DHCP is used.
ipDns2	The 2nd of 2 DNS servers to use, if omitted only the 1st DNS sever is used. This is ignored if DHCP is used

Return:

True if the Internet stack was initialized, false if the stack has already been initialized. To reinitialize call end() and then call begin() again with new parameters.

begin() initializes the hardware, sets up the stack addresses, and starts the IP stack. If DHCP is indicated by not specifying an IP address, DHCP is called to set the network addresses. It may take some time for DHCP to complete and on return from begin() the stack will probably not be completely initialized. Call DNETcK::isInitialzied() to determine if the stack is completely initialized.

void DNETcK::end(void)

Parameters:

None

Return:

None

end() completely terminates all stack operations. The stack is halted, and the hardware is de-initialized. Once the stack has ended, begin() can be called again with new addresses to restart the stack. All instances of TcpClient, TcpServer, UdpClient, UdpServer remain, but their operation becomes indeterminate. It is best to close() all instances before calling end().

bool DNETcK::isInitialzied(void)
bool DNETcK::isInitialzied(DNETcK::DNetStatus * pStatus)
bool DNETcK::isInitialzied(unsigned long msBlockMax)
bool DNETcK::isInitialzied(unsigned long msBlockMax, DNETcK::DNetStatus * pStatus)

Parameters:

pStatus	An optional parameter to receive the current state of initialization.
msBlockMax	The maximum number of milliseconds to wait for the task to complete before returning. If omitted the default block time as specified by setDefaultBlockTime() is used.

Return:

True if the stack is initialized and ready for full operation. False if the stack is not completely initialized. Typically a false does not indicate an error, only that initialization is not complete. Call DNETck::isStatusAnError(*pStatus) to determine if a hard error occurred.

isInitialzied() may be called repeatedly to determine if the stack has fully initialized. If a msBlockMax time is specified, isInitialzied() enters a tight loop executing the stack until the stack is initialized or until the block timeout time has elapsed. If the timeout elapses, isInitialzied() will return false and the status will be the current state of initialization. Unless the status is a hard error, a false does not terminate the initialization process and isInitialzied() may be called again to determine if the stack has been initialized.

With normal usage, it is not typically necessary to call DNETcK::isInitialzied() before calling other DNETcK methods as any dependent methods will call DNETcK::isInitialzied() internally. However, if DNETcK::msImmediate is specified on these dependent calls, they may fail because the stack was not completely initialized. To avoid such errors use a timeout value on the method that will allow the stack to initialize or call DNETcK::isInitialzied() repeatedly until it succeeds.

The most time consuming part for stack initialization is waiting for DHCP to complete. If DHCP is not used and you specify your entire network addresses explicitly; stack initialization will be nearly immediate.

void DNETcK::periodicTasks(void)

Parameters:

None

Return:

None

periodicTasks() executes the stack tasks and when called on a regular basis keeps the stack alive. This method is called internally by most of the DNETcK methods and as long as you are making DNETcK calls the stack will stay alive. However, it is good practice to call periodicTasks() once through your main loop. This will ensure the stack stays alive even when you are not making stack calls. The stack is not interrupt driven and if the stack is neglected for a long enough time packets will be missed and your application will not respond to ARPs, Pings, SNTP updates, or connection requests.

unsigned long DNETcK::setDefaultBlockTime(unsigned long msDefaultBlockTime)

Parameters:

msDefaultBlockTime This sets the default number of milliseconds a blocking method will wait before returning without completing. By default this time is set to 15000ms, or 15 seconds.

Return:

Returns the previously set value.

For most applications the default value will work just fine and this method does not need to be called. However, if you have a time critical application you may wish to change this to meet your timing needs. Probably the most likely change is to make the default DNETcK::msImmediate so all stack methods return immediately. Calling DNETcK::setDefaultBlockTime(DNETcK::msImmediate) in your setup() function will cause all methods taking a msBlockMax parameter ("is" methods and writeStream) to default to msImmediate (or whatever you set). As a result of setting the default timeout to DNETcK::msImmediate is that you will need to call the "is" methods and wait for them to succeed before proceeding. Applications with time critical components other than the IP Stack will probably want to set the default to DNETcK::msImmediate or some other appropriate so the stack methods do not introduce excessive delays.

bool DNETcK::isStatusAnError(DNETcK::DNetStatus status)

Parameters:

status	The status code to check.

Return:

True if the status code is a hard/fatal error. False if the status code is just an intermediate state and the method is proceeding normally.

Many status codes are just progression states and not errors. isStatusAnError() is called to determine if a status code is a hard error. A status code is considered a hard error if normal stack operations won't eventually move the stack to the next state. For example, in resolving a UDP hostname to an IP and MAC address, DNS and ARP must be called to resolve the hostname. This process can take some time to complete and will transverse many states. In this example the progression would look like DNSResolving, ARPResolving, AcquiringSocket, Finalizing, and then completing at EndPointResolved. None of these states are hard errors. However, if at some time during the process something failed and the state was set to say ARPResolutionFailed, this would be a hard error.

bool DNETcK::getMyMac(MAC *pMAC);
bool DNETcK::getMyIP(IPv4 *pIP);
bool DNETcK::getGateway(IPv4 *pIPGateway);
bool DNETcK::getSubnetMask(IPv4 *pSubnetMask);
bool DNETcK::getDns1(IPv4 *pIPDns1);
bool DNETcK::getDns2(IPv4 *pIPDns2);

Parameters:

pMAC	A pointer to a MAC to receive the MAC address.
pIP	A pointer to an IPv4 to receive the local IPv4 address.
pIPGateway	A pointer to an IPv4 to receive the local network's gateway address.

pSubnetMask A pointer to an IPv4 to receive the local network's subnet mask.

pIPDns1 A pointer to an IPv4 to receive the primary DNS server address.

pIPDns2 A pointer to an IPv4 to receive the secondary DNS server address.

Return:

True if the required address was obtained without error, false if the information is not available yet. If false is returned the returned data will be garbage. The most likely reason for a false is that the stack has not been initialized yet and you should try again after DNETcK::isInitialzied() succeeds.

These are helper functions to retrieve what DHCP has assigned as your network addresses. These values will be identical to those specified on the begin() method if DHCP is not used.

unsigned long DNETcK::secondsSinceEpoch(void)
unsigned long DNETcK::secondsSinceEpoch(DNetStatus * pStatus)

Parameters:

pStatus An optional parameter to receive whether the time is since Epoch (DNETcK::TimeSinceEpoch) or since system powerup (DNETcK:: TimeSincePowerUp).

Return:

The number of seconds since Epoch (1/1/1970), or since system powerup.

On system powerup the internal clock starts ticking and secondsSinceEpoch() will return how many seconds has elapsed since powerup. If a DNS server is available, secondsSinceEpoch() will attempt to contact one of a list of known SNTP Time Servers to get the current time referenced to Epoch (Jan 1, 1970). Once Epoch time is obtained, secondsSinceEpoch() will start reporting Epoch time instead of time since powerup. It can take several seconds after powerup for secondsSinceEpoch() to start reporting Epoch time, so it is suggested that the status code be checked before relying on the time. The SNTP servers will be contacted at 10 minute intervals updating the current time. The internal system clock will be used to maintain the current time in-between updates from the time servers. SNTP servers are contacted on a rotating basis as to not overload any one server. It is assumed that the SNTP Time Servers are accurate, but there is no guarantee of that and if one time server is off, time can wander; even going backwards of a server is substantially off. secondsSinceEpoch() is designed to be used for time stamping and is not recommended for program timers/delays; a system timer such as millis() should be used for that.

bool DNETcK::isDNSResolved(const char * szHostName, IPv4 * pIP)
bool DNETcK::isDNSResolved(const char * szHostName, IPv4 * pIP, unsigned long msBlockMax)
bool DNETcK::isDNSResolved(const char * szHostName, IPv4 * pIP, DNetStatus * pStatus)
bool DNETcK::isDNSResolved(const char * szHostName, IPv4 * pIP, unsigned long msBlockMax, DNetStatus * pStatus)

Parameters:

szHostName	A pointer to a zero terminates string representing a hostname to resolve. The string needs to remain valid until isDNSResolved() succeeds, or fails with a hard error.
pIP	A pointer to an IPv4 to receive the IP address of the hostname
pStatus	An optional parameter to receive the current state of resolution.
msBlockMax	The maximum number of milliseconds to wait for the task to complete before returning. If omitted the default block time as specified by setDefaultBlockTime() is used.

Return:

True if the name has been resolved to an IP address; false if the name has not been resolved yet or a hard error occurred.

Resolution can take a long time, so a false response does not necessary mean the hostname could not be resolved, it probably just means it is still working. Call isStatusAnError() to see if a hard error has occurred. isDNSResolved() is dependent on DNETcK::isInitialzied() and if the IP stack is not initialized, msBlockMax will be applied to DNETcK::isInitialzied() and then again to isDNSResolved(). So until DNETcK::isInitialzied() succeeds, it is possible for isDNSResolved() to block twice the block time. Only one DNS resolution may take place at a time, and the underlying stack may be resolving a name (such as a SNTP time server). Under this condition a status of DNSBusy will be returned. This is not a hard error as once the DNS engine becomes available isDNSResolved() will proceed as normal. isDNSResolved() should be called repeatedly with the same parameters until it succeeds, or until a hard error occurs. In particular, the hostname string must remain valid until the name is resolved.

void DNETcK::terminateDNS(void)

Parameters:

None

Return:

None

terminateDNS() forces the DNS engine to release itself. It is a last resort method and should never need to be called. It should only be called if DNSBusy is returned from isDNSResolved() over a very long period of time and clearly the DNS engine is stuck. Forcefully freeing the DNS engine can have unpredictable results and usually indicates a programming error.

void DNETcK::requestARPIpMacResolution(const IPv4& ip)

Parameters:

ip	An IPv4 ip address of the machine to request the MAC address from.

Return:

None

This broadcasts an ARP request on the local area network requesting the MAC address for the machine with the specified IP address. This is a broadcast and may fail to be received by the intended receiver. If isARPIpMacResolved() fails over an extended period of time, say a second or so, this method should be called again to broadcast another request.

The broadcast is limited to the local network and will only return MAC addresses for machines on the local network. If the IP address refers to a machine outside of the local network expect the router/gateway MAC to be returned as the router/gateway is the machine that will handle/forward IP addresses outside of the local network.

bool DNETcK::isARPIpMacResolved(const IPv4& ip, MAC * pMAC);
bool DNETcK::isARPIpMacResolved(const IPv4& ip, MAC * pMAC,
unsigned long msBlockMax);

Parameters:

ip	An IPv4 ip address of the machine to request the MAC address from, this must be the same as that specified on requestARPIpMacResolution().
pMAC	A pointer to a MAC to receive the requested MAC address.
msBlockMax	The maximum number of milliseconds to wait for the task to complete before returning. If omitted the default block time as specified by setDefaultBlockTime() is used.

Return:

True if the MAC address has been received/resolved; false if still waiting for a response.

isARPIpMacResolved() waits for the MAC response. It is possible that the broadcast request attempted by requestARPIpMacResolution() was lost, and if after a period of time isARPIpMacResolved() does not succeed requestARPIpMacResolution() should to be called again to broadcast another request. If after several attempts isARPIpMacResolved() still does not succeed, then the IP address probably cannot be resolved. isARPIpMacResolved() is dependent on DNETcK::isInitialzied() and if the IP stack is not initialized, msBlockMax will be applied to DNETcK::isInitialzied() and then again to isARPIpMacResolved(). So until DNETcK::isInitialzied() succeeds, it is possible for isARPIpMacResolved() to block twice the block time.

TcpClient

The TcpClient instance represents a connection between a remote IP/Port and the local machine's IP/Port (endpoint-pair). A unique socket is assigned to the instance to manage the connection and data stream. The usage is to connect() to a remote endpoint (remote IP/Port) and then call isConnected() to see if you are connected. It may take a while for the connection to be made as it takes time to resolve the hostname and then establish a connection with the remote host. isConnected() should be called repeatedly until it returns true or until a hard error occurs. If msBlockMax is not specified on the isConnected() call, the default Block time (by default is 15 seconds) is used. Experience has shown that many connections will not be made within 10 seconds, but most are made in 15 seconds. This was the driving reason for making the default block time 15 seconds. But clearly this is only an estimate and may still need adjusting. Just be aware that this is a long time to block the main loop in your application. If you have other time critical tasks, you should adjust the block time to a more suitable value and call isConnected() repeatedly until it succeeds. isConnected() can be called at any time and as often as you want; it is the primary method to determine the health of the connection. Requesting the status from this method will return the current status of the connection.

available() calls isConnected() internally so you may go directly from the connect() call to the available() call without calling isConnected(). available() calls isConnected() specifying msImmediate and will immediately return 0 bytes available if there is no connection. If you are confident that the connection will eventually succeed, or your sketch makes no sense if the connection is never made, you can just call connect() and then loop on available() waiting for bytes to be available; just be aware that if the connection is never made you will loop forever on available().

available() may return a count of unread bytes even if the connection is lost, that is, isConnected() returns false. This will only happen if the connection was once made, data was received, and then the connection was dropped. avaliable() will return the unread bytes and you can readStream() them even after the connection is lost. To determine if the TcpClient instance can be closed, it might be best to use a compound condition such as "if(available() ==0 && !isConnected())".

If your connection is lost and you want to reestablish the connection you must close() the TcpClient instance and then call connect() again specifying the connection parameters. Closing an instance of TcpClient will return the instance to its "just constructed" state; you may also close() the instance and then connect() to a different remote endpoint.

Whenever writeStream() or writeByte() is called the bytes are immediately flushed to the wire. Clearly for writeByte() this can be inefficient and therefore you should do your own buffering. Because there is no way to know if you wanted to write one byte and have it transmitted right now, or cache a bunch of writeByte() writes before flushing, it was deemed easier to just document the flushing behavior and have you deal with it. Providing a flush method would probably be the most overlook method and would create obscure bugs as bytes sit un-transmitted in the socket buffer.

On a writeStream() call you may write more data than will fit in the underlying socket buffer. writeStream() will loop writing and flushing data until all of the data has been written to the wire. This could potentially take longer than you wish if you have other time critical tasks in your main loop. For this reason an optional msBlockMax can be specified on the writeStream(). The writeStream() will stop writing data if this time is exceeded and will return the number of bytes actually written. If not all bytes have been written you will need to make another call to writeStream() writing all of the remaining unwritten bytes. If msBlockMax is not specified the default block time is used. The default

block time is 15 seconds unless reassigned by DNETcK::setDefaultBlockTime(). If the default block time was set to DNETcK::msImmediate (by calling DNETcK::setDefaultBlockTime(DNETcK::msImmediate), or setting msBlockMax = DNETcK::msImmediate), then writeStream will write as much as will fit in the underlying socket and flush it, but will not make repeated writes in an attempt to write all bytes. As a practical matter the msBlockMax parameter can be ignored, except in extreme conditions of transmitting very-very large blocks of data.

Unlike the Arduino Ethernet implementation, there can only exist one TcpClient instance per endpoint-pair (socket). To prevent copies, both the TcpClient assignment operator and copy constructor have been defined as private class methods and will cause a compiler error if you attempt to copy the instance. The reason for the rather aggressive copy prevention is because unobvious behavior occurs if the socket is closed on one copy, but then the other copy is continued to be used; or if writes and reads occur out of order on competing copies. Use a pointer to the original instance where you might be tempted to use a copy; a compare of the pointers will clearly indicated that they are the same instance and therefore the same socket and endpoint-pair.

Inherits from the Print Class

All of the virtual Print Class functions are implemented in the following way.

```
void TcpClient::write(uint8_t bData)
{
	writeByte(bData);
}

void TcpClient::write(const uint8_t *buffer, size_t size)
{
	writeStream(buffer, size);
}

void TcpClient::write(const char *str)
{
	if(str != NULL)
	{
		writeStream((const byte *) str, strlen(str));
	}
}
```

The write methods were declared private and non-virtual so they will be hidden from TcpClient as that would be confusing/redundant with writeStream. Print() and println() are both exposed off of TcpClient. However, if you pass TcpClient to a method “as-a” Print class, the write methods will be available off of the Print class.

Copy Prevention

Both the copy constructor and assignment operator have been defined as private methods to produce a compiler error if an instance of TcpClient is attempted to be copied. As noted earlier, unpredictable behavior will occur if competing readStream()s, writeStream()s or close()s occurs on copies of an instance. Use a pointer to the instance in places where tempted to use a copy.

```
private:
   TcpClient&  operator=(TcpClient& tcpClient);
   TcpClient(TcpClient& tcpClient);
```

Constructors

TcpClient()

Constructs an uninitialized instance of the TcpClient class. connect() must be called to set the remote endpoint. If close() is called on the instance, it returns it to this newly constructed state. After close() is called, connect() can be called on the instance specifying a different remote endpoint, or the same one to attempt a reconnect.

Methods

bool connect(const char *szRemoteHostName, unsigned short remotePort)
bool connect(const IPEndPoint& remoteEP)
bool connect(const IPv4& remoteIP, unsigned short remotePort)
bool connect(const char *szRemoteHostName, unsigned short remotePort,
DNETcK::DNetStatus * pStatus)
bool connect(const IPEndPoint& remoteEP, DNETcK::DNetStatus * pStatus)
bool connect(const IPv4& remoteIP, unsigned short remotePort,
DNETcK::DNetStatus * pStatus)

Parameters:

szRemoteHostName A pointer to a zero terminates string representing a remote hostname to connect to. This string must remain valid until the connection is successfully made. Use isConnected() to determine if the connection has been made.

remoteEP An IPEndPoint structure containing the remote IPv4 address and remote port to connect too.

remoteIP An IPv4 of the remote IP to connect to.

remotePort The remote port to connect to.

pStatus An optional parameter to receive the current state of the connection process.

Return:

True if a socket was successfully assigned; false if the socket was not assigned. A false is a hard error and the connection process will not proceed.

connect() sets the remote endpoint, assigns a socket and gets the connection process started. The only successful status for this method is DNETcK::WaitingConnect. isConnected() should be called to determine when the connection is completed.

bool isConnected(void)
bool isConnected(DNETcK::DNetStatus * pStatus)
bool isConnected(unsigned long msBlockMax)
bool isConnected(unsigned long msBlockMax, DNETcK::DNetStatus * pStatus)

Parameters:

pStatus	An optional parameter to receive the current state of connection.
msBlockMax	The maximum number of milliseconds to wait for the task to complete before returning. If omitted the default block time as specified by setDefaultBlockTime() is used.

Return:

True if the connection is made; false if still waiting for the connection to be made, the connection was dropped, or if a hard error occurred. Call isStatusAnError() to see if a hard error has occurred.

isConnected() can be called at any time to determine if the connection is active. available(), peekStream() and readStream() will continue to work even if the connection is dropped as long as there are unread bytes in the socket. writeStream() will fail if the connection is dropped. In the main loop a compound condition such as “if(available() ==0 && !isConnected())” might be best to determine if the TcpClient instance should be closed. It should be noted that isConnected() will not block if a connection is lost, it will return immediately with DNETcK::LostConnect. Blocking will only occur during the initial connection process and from then on a call to isConnected() will return immediately.

size_t available(void)

Parameters:

None

Return:

The number of available bytes ready to be read from the socket.

available() will return the number of bytes available to read even if the connection was dropped. If the connection has not be set up yet, available() will return 0. available() will run the stack prior to checking the unread bytes as to return the most accurate count possible. available() is a non-blocking call and will return as quickly as possible.

void discardReadBuffer(void)

Parameters:

None

Return:

None

This discards any unread bytes in the socket. After discardReadBuffer() is called available() will return 0 until new bytes are received.

size_t readStream(byte *rgbRead, size_t cbReadMax)

Parameters:

rgbRead	A pointer to a buffer to receive bytes from the socket.
cbReadMax	The maximum size of the supplied read buffer.

Return:

The number of bytes actually read. If the socket has no bytes to read, 0 is returned.

readStream() is a non-blocking call and will return as soon as it has read as many bytes as allowed/possible from the socket. readStream() may be called to read unread bytes even if the connection has been lost.

size_t peekStream(byte *rgbPeek, size_t cbPeekMax)
size_t peekStream(byte *rgbPeek, size_t cbPeekMax, size_t index)

Parameters:

rgbPeek	A pointer to a buffer to receive bytes from the socket.
cbPeekMax	The maximum size of the supplied peek buffer.
index	A zero based index into the socket buffer of where to start peeking for bytes. If omitted 0 is used.

Return:

The number of bytes actually peeked. If the socket has no bytes to peek, or the index was out of bounds, 0 is returned.

peekStream() is just like readStream() except it does not remove the bytes from the socket and the bytes can be peeked or read again. peekStream() is a non-blocking call and will return as soon as it has read as many bytes as allowed/possible from the socket. peekStream() may be called to read unread bytes even if the connection has been lost.

int readByte(void)

Parameters:

None

Return:

The next byte to read casted to an int. If the socket has no bytes to read, -1 is returned.

readByte() is a non-blocking call and will return as soon as the byte is read. readByte() may be called to read unread bytes even if the connection has been lost.

int peekByte(void)
int peekByte(size_t index)

Parameters:

index	A zero based index into the socket buffer to peek for the byte.

Return:

The peeked byte casted to an int. If the socket has no bytes to peek or the index is out of bounds, -1 is returned.

peekByte() is just like readByte() except it does not remove the byte from the socket and the byte can be peeked or read again. peekByte() is a non-blocking call and will return as soon as the byte is read. peekByte() may be called to read unread bytes even if the connection has been lost.

size_t writeStream(const byte *rgbWrite, size_t cbWrite)
size_t writeStream(const byte *rgbWrite, size_t cbWrite, DNETcK::DNetStatus * pStatus)
size_t writeStream(const byte *rgbWrite, size_t cbWrite, unsigned long msBlockMax)
size_t writeStream(const byte *rgbWrite, size_t cbWrite, unsigned long msBlockMax, DNETcK::DNetStatus * pStatus);

Parameters:

rgbWrite	A pointer to a buffer of bytes to be written out to the connection
cbWrite	The number of bytes to write
msBlockMax	The maximum number of milliseconds to wait for the operation to complete before returning. If omitted the default block time as specified by setDefaultBlockTime() is used.
pStatus	An optional parameter to receive the status of the write

Return:

The number of bytes actually written, this number may be less than cbWrite if msBlockMax time was exceeded or if the connection was lost.

writeStream() immediately flushes the data out to the connection. You may write more data than will fit in the underlying socket buffer. writeStream() will loop writing and flushing data until all of the data is written out. This can potentially take longer than you wish if you have other time critical applications in your main loop. For this reason an optional block time can be specified on the writeStream() to limit the amount of time writeStream() will take to before returning. writeStream() will stop writing data if this time is exceeded and will return the number of bytes actually written. If not all bytes have been written you will need to make another call to writeStream() to finish writing out the unwritten bytes. If DNETcK::msImmediate is specified as the block time, only as much as will fit in the underlying socket will be written and writeStream will not loop attempting to write more data.

int writeByte(byte bData)
int writeByte(byte bData, DNETcK::DNetStatus * pStatus)

Parameters:

bData	The byte to write out to the connection
pStatus	An optional parameter to receive the status of the write

Return:

1 if the byte was written, 0 if it was not. Check the status or call isConnected() for status if the byte does not write out.

writeByte() immediately flushes the byte to the connection and is inefficient if called repeatedly. If you have many bytes to write, cache your bytes into a buffer first and then call writeStream().

void close(void)

Parameters:

None

Return:

None

This discards any unread bytes in the socket, releases the socket and returns the TcpClient instance back to its “just constructed” state. Once closed the TcpClient instance can be used to make a new connection to a remote host by calling connect() with new remote host parameters.

bool getRemoteEndPoint(IPEndPoint *pRemoteEP);

Parameters:

pRemoteEP A pointer to an IPEndPoint structure to receive the remote IP and Port.

Return:

True if the remote endpoint was successfully filled in; false if the connection was never made and the remote endpoint is not available. If false is returned the remote endpoint data will be garbage.

getRemoteEndPoint() can be called if the connection is lost as long as it is called before the TcpClient instance is closed. If a reconnect to a server is needed, you can get the remote server endpoint by calling getRemoteEndPoint(), then close() the TcpClient instance to put it back to its “just constructed” state, and then call connect(*pRemoteEP) to attempt to reconnect to the server. Since the endpoint data contains the IP address instead of the hostname, the reconnect will be faster as DNS will not be called to resolve the hostname.

bool getLocalEndPoint(IPEndPoint *pLocalEP);

Parameters:

pLocalEP A pointer to an IPEndPoint structure to receive the local endpoint IP and Port.

Return:

True if the local endpoint was successfully filled in; false if the connection was never made and the local endpoint is not available. If false is returned the local endpoint data will be garbage.

getLocalEndPoint() can be called if the connection is lost as long as it is called before the TcpClient instance is closed. The IP address will be the same as that returned by DNETcK::getMyIP() as the local endpoint is the local machine. However the port will be the local TCP assigned ephemeral port for the connection and will only be available once the socket is assigned and the connection is made.

bool getRemoteMAC(MAC *pRemoteMAC);

Parameters:

pRemoteMAC A pointer MAC to receive the MAC address of the remote machine.

Return:

True if the remote MAC was successfully filled in; false if the connection was never made and the remote MAC is not available. If false is returned the MAC data will be garbage.

getRemoteMAC() can be called if the connection is lost as long as it is called before the TcpClient instance is closed. The MAC will be that as returned by the ARP request. This means the MAC will be the MAC address of a machine on your local network. If the remote IP is not on your local network the MAC will typically be that of your router/gateway as your router/gateway is the bridge between your local network and the internet.

TcpServer

TcpServer uses the Listen/Accept model where you start listening on a local port and when a connection is made TcpServer will allow you to accept a TcpClient to communicate to that connection. Once you accept a client, TcpServer releases all management for that connection to the TcpClient. If you want to close the connection, you must do so on the TcpClient. Unlike the TcpServer implementation in Arduino, TcpServer will never return another TcpClient to an actively open and accepted TcpClient. The only way that TcpServer might get control of that connection again is for you to close the TcpClient and then have the remote endpoint attempts to reestablish a new connection to the listening port.

If TcpServer is closed all resources held by TcpServer are released and the instance is returned to its "just constructed" state. This includes closing all unaccepted TcpClients and releasing their sockets. Closing the server will not affect any already accepted TcpClients. Once TcpServer is closed, you can reuse the TcpServer instance to start listening on another (or same) local port by calling startListening().

When constructing a TcpServer instance you may specify the maximum number of sockets (cMaxPendingClients) that TcpServer will use for unaccepted TcpClients. This allows you to limit how many sockets TcpServer will take from the system. If you do not specify the maximum number of sockets, by default TcpServer will use 3. The sockets are not taken until needed, and if there are a lot of other sockets being used by the system, it is possible that TcpServer will not be able to acquire a socket to continue listening. As long as the number of unaccepted clients is less than the maximum sockets allowed, TcpServer will keep trying to get a socket until a socket becomes available and then TcpServer will automatically start listening again. If however, the unaccepted clients reach the maximum allowed sockets, TcpServer will automatically stop listening. Since the maximum socket count only applies to the number of sockets being managed by TcpServer, accepting a client will reduce it's in use count as TcpServer gives up all management of that client; and listening will automatically start again. If you want to know if the server is actively listening right now, call isListening(), you can request a status to see why TcpServer might not be listening. TcpServer checks and attempts to listen on each call to availableClients(). Since the maximum number of sockets is specified on the constructor, the "just constructed state" retains this number. If you close() the TcpServer returning it to its "just constructed" state and startListening() again, the same maximum number of sockets (pending clients) is used. If you want to change the maximum number of sockets allowed, you must reconstruct a new instance of TcpServer specifying a new value.

If availableClients() returns more then 1, you may wish to prioritize who you accept based on its remote endpoint. You can get the remote endpoint for an unaccepted client by calling getAvailableClientsRemoteEndPoint() specifying the zero based index of the AvailableClients. Typically a simple loop walking the total number of availableClients() checking the remote endpoint is done to see who is waiting to be accepted. Then by specifying the desired index on acceptClient(), you can accept the client you want in the order you want. If you see a remote endpoint you don't want to accept; then acceptClient() it and immediately close() the TcpClient instance without any further processing. Once you have closed a TcpClient, you may reuse that instance of TcpClient to accept another client from acceptClient().

TcpServer attempts to retain a first connected first accepted order in indexing. However, accepting a client out of order, or if a connection is dropped before the client is accepted can cause the index order to change. It should be noted that if a remote endpoint connects with the server and then drops the connection, the server may automatically purge that client from its accept list if and only if no data

was ever sent from the remote client. If data was sent, the server keeps the client so that you can accept it and read the data that was sent; even though the connection was dropped. For these reasons, after looping through the availableClients() your indexes are only valid until the next call to availableClients() or acceptClient() as these methods potentially can change the number and order of availableClients(), thus messing up your indexing.

Copy Prevention

Both the copy constructor and assignment operator have been defined as private methods to produce a compiler error if an instance of TcpServer is attempted to be copied. Unpredictable behavior will occur if competing copies call acceptClient() or close(). Use a pointer to the instance in places where tempted to use a copy.

```
private:
   TcpServer&  operator=(TcpServer & tcpServer);
   TcpServer (TcpServer& tcpServer);
```

Constructors

TcpServer()
TcpServer(int cMaxPendingClients)

Constructs an instance of the TcpServer class specifying the maximum number of unaccepted clients it will manage before it stops listening. In effect, this limits the maximum number of sockets that it will consume from the system and allows you control over socket usage.

Parameters:

cMaxPendingClients The Maximum number of unaccepted clients, or sockets, before TcpServer will stop listening. If omitted, 3 is used.

Methods

bool startListening(unsigned short localPort)
bool startListening(unsigned short localPort, DNETcK::DNetStatus * pStatus)

Parameters:

localPort The port to start listening on.

pStatus An optional parameter to receive the status, this will typically fail only if a socket could not be obtained or an invalid/in use port number was specified.

Return:

True if a socket was acquired and listening on the port started. False if it could not start to listen and startListening() should be called again once the problem is corrected.

bool isListening(void)
bool isListening(DNETcK::DNetStatus * pStatus)

Parameters:

pStatus An optional parameter to receive the listening status.

Return:

True if the TcpServer instance is currently listening, False if it is not listening.

isListening() returns the current actively listening state. It is possible for the server not to be listening because there are no more sockets to listen on, or because the number of unaccepted clients has reached its maximum allowed as specified on the constructor, or because stopListening() was called. Listening will automatically resume when these conditions correct themselves.

void stopListening(void)

Parameters:

None

Return:

None

Forces the TcpServer instance to suspend listening; resume listening by calling resumeListening(). When stopped, all other server methods such as acceptClient() and availableClients() will work as expected. When stopped, the listening socket is returned to the system.

void resumeListening(void)

Parameters:

None

Return:

None

Returns the TcpServer instance to its listening state; this is usually called to resume listening after a stopListening() was called. Just because resumeListening() is called does not mean that listening has started. If a socket is not available, or if the maximum allowed sockets (pending clients) are in use, listening will be suspended until these conditions are corrected; however, listening will automatically start once conditions allow. Call isListening() to see if the server is actually listening.

void close(void)

Parameters:

None

Return:

None

This releases all resources held by the TcpServer instance, and closes all unaccepted clients. The instance is returned to its “just constructed” state and startListening() may be called on the instance with a new port number. Any already accepted clients are not closed.

int availableClients(void)

Parameters:

None

Return:

The number of pending clients waiting to be accepted; zero if none are waiting.

This returns the number of pending clients waiting to be accepted via acceptClient(). This method is also a workhorse method that checks to see if the listening socket has a connection on it and moves that connection to a pending client state, it will also checks to make sure that there is a listening socket waiting for new connections. All processing is done first so that the number of availableClients() returned is as up-to-date as possible. Be aware that availableClients() will invalidate any indexing obtained from a prior availableClients() call. While the TcpServer attempts to preserver a first connect first accept order, it is possible for old connections to be dropped; and that can change the number and order of unaccepted clients.

bool acceptClient(TcpClient * pTcpClient)
bool acceptClient(TcpClient * pTcpClient, int index)
bool acceptClient(TcpClient * pTcpClient, DNETcK::DNetStatus * pStatus)
bool acceptClient(TcpClient * pTcpClient, int index, DNETcK::DNetStatus * pStatus)

Parameters:

pTcpClient	A pointer to a “just constructed” TcpClient instance.
index	A zero based index into the availableClients() to accept. If omitted 0 is used.
pStatus	An optional parameter to receive the status, typically this is just parameter checking such as index out of bounds, or TcpClient is already in use.

Return:

True if the pending client was accepted, the TcpClient instance was initialized to the connection, and the TcpServer instance releases all management of the client. False if there was some reason the TcpClient could not be made.

acceptClient() releases a pending client to you. Once released, the server will no long manage the connection in anyway. You must provide a pointer to a “just constructed” TcpClient instance to accept the client. A TcpClient instance that was closed may also be used as the accepting TcpClient instance. The index must be less than that returned by availableClients(). An index of 0 will return the first client to connect to the server as the server retains a first connects first accept order. acceptClient() may fail if no clients are available, the TcpClient instance passed is already in use, if you passed in a NULL for the TcpClient instance, or if the index is out of bounds.

bool getAvailableClientsRemoteEndPoint(IPEndPoint *pRemoteEP)
bool getAvailableClientsRemoteEndPoint(IPEndPoint *pRemoteEP, int index);
bool getAvailableClientsRemoteEndPoint(IPEndPoint *pRemoteEP, MAC * pRemoteMAC, int index);

Parameters:

pRemoteEP A pointer to an IPEndPoint to receive the remote IP an Port.

pRemoteMAC A point to a MAC to receive the remote MAC address.

index A zero based index into the availableClients() to accept. If omitted 0 is used.

Return:

True if the remote endpoint information was obtained, false if the index was out of bounds.

getAvailableClientsRemoteEndPoint() allows you to peek at the endpoints of waiting clients. Call availableClients() to see how many clients are waiting and then loop through indexing each available client for its endpoint. Calling getAvailableClientsRemoteEndPoint() is safe and will not change the indexing order as this call does not “run” the server. However, if you call acceptClient() with the index you want, acceptClient() will change the indexing order as the accepted client is removed from the list and the server is “run”. After an acceptClient() call you will need to call availableClients() again and walk the new list from scratch. If you request a remote MAC, understand that this is the MAC as returned by ARP and will be a MAC of a machine on your local network. If the remote IP is not on your local network, the MAC will most likely be that of your gateway/router. Typically remote MACs are uninteresting and is why this is an optional parameter.

bool getListeningEndPoint(IPEndPoint *pListeningEP)

Parameters:

pListeningEP A pointer to an IPEndPoint to receive the listening IP an Port.

Return:

True if the listening endpoint information was obtained, false if an error occurred.

This is a somewhat redundant method but is provided for completeness and ease of use. The information returned will be your local IP address, which you can get from DNETcK: getMyIP(), and the port number as specified on startListening(). getListeningEndPoint() may be useful if you are listening on several different ports and have several TcpServer instances and you wish to see who is who by retrieving their listening port number.

UdpClient

UdpClient is a connectionless pairing of a Remote IP/Port with a local IP/Port on a socket which will be referred to as an endpoint-pair. Loosely this can be thought of as the UDP parallel to the TCP connection; which by the way also has an endpoint-pair on its socket. UDP never has an established connection and therefore can't drop a connection it doesn't have. So once an endpoint-pair is established, UDP just sends and receives datagrams in the blind. Looking at the TcpClient methods we have connect() and isConnected(), the parallels in UdpClient are setEndPoint() and isEndPointResolved(). setEndPoint() sets the remote endpoint we want to communicate to, and isEndPointResolved() lets us know when we can send/receive datagrams to/from that remote endpoint. In theory, isEndPointResolved() wouldn't really be needed if we knew the remote IP/Port/MAC upfront and then setEndPoint() could completely set up the socket; but usually we either start with a hostname or an IP, and rarely do we have the MAC. isEndPointResolved() will walk through DNS to resolve a hostname to an IP, and then ARP is called to resolve the IP to a MAC.

Unlike TCP, where the connection protocol is defined, UDP has no connection protocol and DNETcK must implement each step of resolving the hostname and IP. With each step there is a possibility of errors. So keep calling isEndPointResolved() until it succeeds or fails with a hard error. You can request the status from isEndPointResolved() and then check the status by calling DETcK::isStatusAnError() to determine if a hard error occurred.

If you specify a hostname to setEndPoint(), DNS will be called to resolve the hostname to an IP, and then ARP is called to resolve the IP to a MAC address. If you specify an IP, only ARP is called to resolve the MAC. If you specify the remote MAC, the socket is setup directly and datagrams are ready to be transmitted/received. Be careful when specify the MAC as if you get this wrong, your datagrams will be lost into the abyss with no error as the datagram will go on the wire just fine, but nothing will be there to pick it up.

While DNS is a defined protocol, ARP is UDP broadcast/response mechanism. Just because an ARP request was put on the wire does not mean anyone will respond. Sometimes we get no response because the request was missed, sometimes because there is no-one with that IP address and thus there will be no response. The requester has to just try a few times and if nothing comes back assume it failed. ARPs are local network requests, so they tend to be fast as they are local with no routing. If the IP address is outside of the subnet, the router/gateway will respond with its MAC address. The constructor for UdpClient allows you to request how many times you wish to broadcast an ARP request, and how long to wait before re-broadcasting the ARP request before failing. By default this is set to 3 rebroadcasts waiting 1 second between retries for a total of 3 seconds. If the ARP request is not answered after that, it is assumed to fail. isEndPointResolved() does not block while waiting for the ARP response, it just checks on each call to see if the wait time has elapsed and if it has it rebroadcasts the request until the retry count has been met. In general, the default is probably good enough and you can omit these parameters on the constructor. However, if you know that your ARP requests should be succeeding and they are not, try increasing these values.

Once you have the endpoint resolved to a remote IP/port/MAC you are ready to send/receive. UDP is a datagram protocol, not a stream protocol like TCP. UDP only insures the integrity of a datagram as a whole. That is, if you receive the datagram, you know you got the whole datagram. There is no guarantee that you will receive the datagram, or that you will receive the datagrams in the order that they were sent. Since there are no connections, each datagram stands on-its-own, and each datagram can be routed independently through different network paths, datagram by datagram; one could be routed through China, the next through India, and you will probably lose the ones that take

the gratuitous routes. UdpClient reflects the datagram concept in its methods. For example, when writing, it is assumed you have built the complete datagram in your write buffer as you expect it to go out on the wire. This is why there is only a writeDatagram() method and no WriteByte method; in general a 1 byte datagram is probably not common. Of course, you could create a datagram buffer of 1 byte and write that.

Likewise, reading is by the datagram as well. Because there is no guarantee that a datagram will reach its intended recipient; an annoying yet all too common implementation is to throw out any unread datagrams when a new datagram comes in. This was deemed excessively annoying and DNETcK caches unread datagrams in a datagram cache so you don't lose them. available() will return the number of bytes in the next datagram in the cache. There may be more datagrams in the datagram cache, but available() will only let you know about the next datagram. Normal usage would be to supply a read buffer large enough to read the whole datagram on your next call to readDatagram(). However, because datagrams can be up to 64K, DNETcK allows you to read partial datagrams. If you read a partial datagram, the next call to available() will return the number of unready bytes in the datagram. But you will know that you are reading a partial datagram because you supplied a read buffer smaller than that reported by your first call to available(). available() will not report the size of the next datagram in the datagram cache until the current datagram is completely read.

As datagrams are received they are put in the datagram cache in FIFO order. However, should the cache become full the oldest datagrams in the cache are discarded to make room for the new datagrams, so the cache always contains the most recent datagrams that will fit. When a datagram is discarded, the whole datagram is discarded as the UDP specification requires that the datagram integrity be preserved and attempting to keep a partial datagram would violate that. If a datagram comes in that is physically bigger than the whole cache, the datagram is discarded without disturbing the current contents of the cache. If a call to readDatagram() has only read a partial datagram, the datagram cache is frozen to ensure that the remaining partial datagram is not discarded as new datagrams come in. As long as there is a partial unread datagram in the cache, only new incoming datagrams that will fit is the unused portion of the cache will be saved. If the cache becomes full, new datagrams are discarded until the partial datagram is completely read. Once the partial datagram is completely read, the cache is unfrozen and the oldest datagrams are discarded to make room for new ones. It is fairly important that the datagram cache be sized so you can read them faster than the cache fills. As long as there is room in the cache, any datagram that is received on the socket will be available for you to read.

The datagram cache is specified on the constructor to UdpClient and is a non-optional parameter to the constructor. You should specify this cache size to be large enough to hold the largest datagram you expect to receive multiplied by the number of datagrams that might come in before you can read them. There is no default datagram cache as that would require DNETcK to either reserve static memory or acquire it dynamically from a heap. Both can take unnecessary resources from the already restricted environment, and heaps can fragment over time and fail. For reliability and resource usage reasons, it was considered best to just have the cache supplied on the constructor. The only requirement on the cache size is that it must be at least 32 bytes long. As a guideline only, a cache of 1-2K is probably a good starting point if you have no idea what your usage is.

Closing an UdpClient will return the UdpClient instance to its "just constructed" state. Since the cache is specified on the constructor, the cache is still in use by the closed instance. Also the ARP retries and wait times will remain the same. You may call setEndPoint() on the closed instance to reestablish a new endpoint-pair on the socket, and use the instance for a new communication.

Copy Prevention

Both the copy constructor and assignment operator have been defined as private methods to produce a compiler error if an instance of UdpClient is attempted to be copied. Unpredictable behavior will occur if competing copies of an UdpClient attempts to read datagrams out of the same datagram cache or is closed. Use a pointer to the instance in places where tempted to use a copy.

```
private:
   UdpClient&  operator=( UdpClient & udpClient);
   UdpClient (UdpClient& udpClient);
```

Constructors

UdpClient(byte * rgbCache, size_t cbCache)
UdpClient(byte * rgbCache, size_t cbCache, unsigned long msARPWait, unsigned long cARPRetries)

Parameters:

rgbCache	A pointer to a buffer to be used as the datagram cache. This buffer must remain valid for the life of the UdpClient instance. Even if the instance is closed, this cache must remain valid. Only after the instance has been destructed can you release the cache buffer.
cbCache	The size in bytes of the cache buffer. This must be at least 32 bytes. This should be sized to be a few times larger than the largest datagram you expect to receive.
msARPWait	The number of milliseconds to wait before retransmitting an ARP broadcast request. If omitted it defaults to 1000ms or 1 second.
cARPRetries	The number of ARP rebroadcast attempts before abandoning an ARP IP to MAC resolution. If omitted this defaults to 3.

This constructs an UdpClient instance that can be used to setup a endpoint-pair to start sending/receiving datagrams to/from a remote endpoint. When constructed a buffer must be supplied to be used as the datagram cache. This cache should be sized to be larger than the largest expected datagram multiplied by how many datagrams you expect to be cached before your application can read/remove (readDatagram()) the datagrams from the cache. The ARP retry count (cARPRetries) is a maximum number of ARP broadcast requests that will be broadcasted on the local network before the IP to MAC resolution is abandoned. The ARP wait time (msARPWait) is the number of milliseconds to wait for a response from an ARP broadcast before resending another ARP broadcast request. The wait time is non-blocking, that is, it is wall time and is checked on each call to isEndPointResolved(). Typically the APR defaults are good for most applications.

Methods

bool setEndPoint(const char *szRemoteHostName, unsigned short remotePort)
bool setEndPoint(const IPv4& remoteIP, unsigned short remotePort)
bool setEndPoint(const IPEndPoint& remoteEP)
bool setEndPoint(const char *szRemoteHostName, unsigned short remotePort, unsigned short localPort)
bool setEndPoint(const IPv4& remoteIP, unsigned short remotePort, unsigned short localPort)
bool setEndPoint(const IPEndPoint& remoteEP, unsigned short localPort)
bool setEndPoint(const IPEndPoint& remoteEP, MAC& remoteMAC)
bool setEndPoint(const IPv4& remoteIP, unsigned short remotePort, MAC& remoteMAC)
bool setEndPoint(const IPEndPoint& remoteEP, MAC& remoteMAC, unsigned short localPort)
bool setEndPoint(const IPv4& remoteIP, unsigned short remotePort, MAC& remoteMAC, unsigned short localPort)

Parameters:

szRemoteHostName A pointer to a zero terminated string of the remote hostname to communicate with. DNS will be called to resolve the hostname to an IPv4 address. If no DNS servers have been specified, isEndPointResolved() will fail with a DNS error. The string must remain valid until isEndPointResolved() succeeds.

remotePort The remote endpoint port number to communicate to.

remoteIP An IPv4 of the remote endpoint to communicate to.

remoteEP An IPEndPoint containg the remote IPv4 and port number of the remote endpoint to communicate to.

remoteMAC The MAC address of the machine on the local network to send UDP packets to. It is generally better to not specify this and allow an ARP request to resolve this for you as if you get this wrong, your UDP datagrams will be lost.

localPort In the endpoint-pair, this is the local port number to use. In general this parameter can be omitted and the underlying socket engine will pick an appropriate and available ephemeral port.

Return:

True if the endpoint resolution process started successfully; false if an error occurred.

A true response does not mean the endpoint was successfully resolved, only that the process started without error; isEndPointResolved() should be called to determine if the endpoint was resolved. Specifying a remote MAC will significantly improve the speed of the endpoint resolution process. However, if you specify an inaccurate remote MAC, the endpoint data in the socket will be set up with the incorrect MAC. Data will be transmitted on the wire just fine, however with the wrong MAC address no-one will pick up the datagram and it will be lost. Only specify the MAC if you are sure you have the correct one; otherwise don't specify the remote MAC and allow ARP to automatically resolve the MAC for you. If setEndPoint(UdpClient::broadcastIP, port) is called with the special predefined

remote UdpClient::broadcastIP IP address, the datagram will be broadcasted to all machines on your local network. Any machines listening on the port should pick it up.

bool isEndPointResolved(void);
bool isEndPointResolved(DNETcK::DNetStatus * pStatus)
bool isEndPointResolved(unsigned long msBlockMax)
bool isEndPointResolved(unsigned long msBlockMax, DNETcK::DNetStatus * pStatus)

Parameters:

pStatus	An optional parameter to receive the current state of the endpoint resolution process.
msBlockMax	The maximum number of milliseconds to wait for the task to complete before returning. If omitted the default block time as specified by setDefaultBlockTime() is used.

Return:

True if the endpoint was resolved to a remote IP/Port/MAC; false if still waiting for resolution or if a hard error occurred. Call isStatusAnError() to see if a hard error has occurred.

isEndPointResolved() can be called at any time to determine if the endpoint has been completely resolved to the remote IP/Port/MAC. Once the endpoint is successfully resolved it is not necessary to call isEndPointResolved() again; but calling it again will always return a true result. Once isEndPointResolved() succeeds, you may start sending/receiving datagrams to/from the remote endpoint via writeDatagram() / readDatagram(). However, just because an endpoint is resolved it does not mean that the endpoint is listening or will send/receive datagrams. Remember, UDP is a connectionless unreliable protocol; it can fail for many reasons including that the endpoint does not exist. Possible hard failures for isEndPointResolved() are DNS or ARP failures, and will prevent the endpoint from being completely resolved.

void close(void)

Parameters:

None

Return:

None

This releases all resources except the datagram cache held by the UdpClient instance and returns the instance to its “just constructed” state. Since both the datagram cache and the ARP parameters are specified on the constructor, these values remain active even after the close(). Once closed the instance may call setEndPoint() to set up a new endpoint-pair, or the instance can be provided to acceptClient() to accept a UdpClient from a UdpServer.

size_t available(void)

Parameters:

None

Return:

The number of available bytes in the next datagram ready to be read from the datagram cache.

available() will return the number of bytes in the next datagram available in the datagram cache. If a readDatagram() was called with a buffer size smaller than the datagram, a partial datagram will be read and available() will return the number of remaining unread bytes in the partial datagram.

void discardDatagram(void)

Parameters:

None

Return:

None

This discards/removes the next datagram as reported by available() from the datagram cache. The following datagram in the cache will become visible to available() and readDatagram().

size_t readDatagram(byte *rgbRead, size_t cbReadMax)

Parameters:

rgbRead A pointer to a buffer to receive the datagram

cbReadMax The maximum size of the supplied read buffer.

Return:

The number of bytes actually read. If the datagram cache is empty 0 is returned.

readDatagram() will only return bytes from the next datagram in the datagram cache. If cbReadMax is less than the number of bytes in the next datagram, a partial datagram will be read and available() will report the remaining unread bytes in the datagram. If a partial datagram is read, the datagram cache will freeze in as much as the remaining partially read datagram will not be discarded should the cache overfill. Call readDatagram() repeatedly until the datagram is completely read. Once completely read,

the datagram cache will unfreeze and new incoming datagrams will overwrite the oldest datagrams in the cache in the event that the cache overfills.

size_t peekDatagram(byte *rgbPeek, size_t cbPeekMax)
size_t peekDatagram(byte *rgbPeek, size_t cbPeekMax, size_t index)

Parameters:

rgbPeek	A pointer to a buffer to receive bytes from the datagram.
cbPeekMax	The maximum size of the supplied peek buffer.
index	A zero based index into the datagram of where to start peeking for bytes. If omitted the bytes are peeked from the start of the datagram.

Return:

The number of bytes actually read. If the datagram has no bytes to read, or the index was out of bounds, 0 is returned.

peekDatagram() is just like readDatagram() except it does not remove the datagram from the datagram cache. Be careful not to call available() between peekDatagram() and readDatagram() as available() will run the cache and the datagram could potentially be discarded if a new datagram comes in and overfills the cache, thus discarding the datagram just peeked. peekDatagram() can be called as many times as you like safely as peekDatagram() does not run the datagram cache.

int peekByte(void)
int peekByte(size_t index)

Parameters:

index	A zero based index into the datagram of where to peek the byte. If omitted the first byte in the datagram is peeked.

Return:

The peeked byte casted to an int. If the datagram has no bytes to read or the index is out of bounds, -1 is returned.

peekByte() is just like peekDatagram() with a peek count of 1 byte and that the byte is returned as an int.

long int writeDatagram(const byte *rgbWrite, size_t cbWrite)

Parameters:

rgbWrite A pointer to a buffer containing the datagram. The buffer will be transmitted directly as a complete datagram on the wire.

cbWrite The number of bytes in the datagram.

Return:

The number of bytes actually written if successful. Otherwise the negative (-) size of the maximum datagram supported by the underlying socket. The datagram is not sent on failure.

The UDP specification allows for a maximum datagram size of 64K, but since a datagram is transmitted as a whole, the socket's transmit buffer size limits the largest datagram that can be sent. The transmit buffer is about 1500 bytes, less headers. Datagrams approaching 1500 bytes will fail to transmit.

bool getRemoteEndPoint(IPEndPoint *pRemoteEP);

Parameters:

pRemoteEP A pointer to an IPEndPoint structure to receive the remote IP and Port.

Return:

True if the remote endpoint was successfully filled in; false if the endpoint-pair was never set up and is not available. If false is returned the remote endpoint data will be garbage.

getRemoteEndPoint() can be called any time after isEndPointResolved() succeeds, otherwise the endpoint returned will be garbage.

bool getLocalEndPoint(IPEndPoint *pLocalEP);

Parameters:

pLocalEP A pointer to an IPEndPoint structure to receive the local IP and Port.

Return:

True if the local endpoint was successfully filled in; false if the endpoint-pair was never set up and is not available. If false is returned the local endpoint data will be garbage.

getLocalEndPoint() can be called any time after isEndPointResolved() succeeds, otherwise the endpoint returned will be garbage.

bool getRemoteMAC(MAC *pRemoteMAC);

Parameters:

pRemoteMAC A pointer MAC to receive the MAC address of the remote machine.

Return:

True if the remote MAC was successfully filled in; false if the endpoint-pair was never set up and is not available. If false is returned the MAC data will be garbage.

getRemoteMAC() can be called any time after isEndPointResolved() succeeds, otherwise the MAC returned will be garbage. The MAC will be that as resolved by the ARP request; or as set by setEndPoint(). This means the MAC will be the MAC address of a machine on your local network. If the remote IP is not on your local network the MAC will typically be that of your router/gateway.

UdpServer

UdpServer uses the Listen/Accept model very similar to that of TcpServer. In fact the methods of UdpServer and their conceptual usage are exactly the same. The main differences come in that UDP is a connectionless protocol and that we must supply buffers for the UDP datagram cache.

When considering the connectionless nature of UDP, we need to invent a metric that is somehow the corollary to a TCP connection. If we think of UDP as a conversation between a local and remote endpoint, we can talk about the detection of an endpoint-pair. This pair is how the UdpServer determines that a new socket needs to be allocated and a new UdpClient should be created. The UdpServer listens on a local port, and when UdpServer sees a datagram sent to its listening port from a remote IP/Port that has not already been assigned to an active socket, a new socket and UdpClient is created; and that establishes our UDP conversation. Once the socket is setup, any datagrams coming from that remote IP/Port to our local IP/Port will be reported to the already setup socket and UdpClient. Recognizing that a UDP conversation is not a connection, we understand that the UdpClient will exist indefinitely, or until we explicitly close() the UdpClient. Closing the client is the only way to release the socket back to the system.

As with TcpServer, UdpServer is told the maximum number of sockets, or pending clients, it can allocate. Once that limit has been hit UdpServer will not allocate another socket and thus must stop listening until a client is accepted thus reducing the number of unaccepted clients. Unlike TcpServer, with each socket we must assign a UDP datagram cache or we will lose an annoying number of datagrams before we have a chance to accept the client and read the datagrams. On the constructor of UdpServer we will need to specify the maximum number of sockets as well as a buffer to use for the datagram caches. The buffer will be split evenly across the maximum number of sockets; the code looks like “cbSocketCache = cbCache / cMaxPendingClients”. So if we want each of our sockets to have a datagram cache of 1k each, and we are going to allow 3 sockets (pending clients), then we need to construct UdpServer with a cache buffer of 3K (3 * 1K = 3K).

When an UdpClient is accepted from the UdpServer, the datagram cache is copied from the server’s cache to the client’s cache. Since it is possible that the client’s cache is smaller than the one used by the server, some datagrams may be discarded when copied. During the copy, only whole datagrams are copied, and as many datagrams as possible will be copied from the server’s cache to the client’s cache. For this reason, it is probably best to define the client’s cache to be equal to or greater than the server’s individual socket cache; this will ensure no datagrams will be lost in the copy. In our UdpServer example with a 3K datagram cache and maximum pending clients of 3, we would construct an UdpClient instances with at least a 1K datagram cache.

UdpServer will listening on a local port and when an endpoint-pair is detected the socket will report this as an availableClients(). UdpServer allocates sockets as needed and if there are a lot of other sockets being used by the system, it is possible that UdpServer will not be able to acquire a socket to continue listening and will then stop listening. However, as long as the number of unaccepted clients is less than the maximum sockets allowed, UdpServer will keep trying to get a socket until a socket becomes available and then UdpServer will automatically start listening again. If however, the unaccepted clients reaches the maximum allowed sockets (pending clients), UdpServer will automatically stop listening. Since the maximum socket count only applies to the number of sockets being managed by UdpServer, accepting a client will reduce it’s in use count as UdpServer gives up all management of that client; and listening will automatically start again. If you want to know if the server is actively listening right now call isListening(), you can request a status to see why UdpServer

might not be listening. UdpServer does this checking and attempting to listen on each call to availableClients().

If availableClients() returns more then 1, you may wish to prioritize who you process based on its remote endpoint. You can get the remote endpoint for an unaccepted client by calling getAvailableClientsRemoteEndPoint() specifying the zero based index of the availableClients() you want. Typically a simple loop walking the total number of availableClients() checking the remote endpoint is done to see who is waiting to be accepted. Then by specifying the desired index on acceptClient() you can accept the client you want in the order you want. If you see a remote endpoint you don't want to accept; then acceptClient() it and immediately close() the UdpClient instance without any further processing. Once you have closed a UdpClient, you may reuse that instance of UdpClient to accept another client from acceptClient().

UdpServer attempts to retain a first detect first accepted order in indexing. However, accepting a client out of order will cause the indexing to change. For this reason, after looping through the availableClients() your indexing is only valid until the next call to acceptClient() as this method will change the number and order of availableClients(), thus messing up your indexing.

When UdpServer is closed all resources except the UDP datagram cache are released and the instance is returned to its "just constructed" state. Beware, the UDP datagram cache is still in use and cMaxPendingClients still applies as these were specified on the constructor. Closing the server will not affect any already accepted UdpClients; but all unaccepted UdpClients will be Closed(). Once the UdpServer is closed, you can reuse the UdpServer instance to start listening on another (or same) local port by calling startListening().

Copy Prevention

Both the copy constructor and assignment operator have been defined as private methods to produce a compiler error if an instance of UdpServer is attempted to be copied. Unpredictable behavior will occur if competing instances attempt to do an acceptClient() or close(). Use a pointer to the instance in places where tempted to use a copy.

```
private:
   UdpServer&  operator=(UdpServer & udpServer);
   UdpServer (UdpServer & udpServer);
```

Constructors

UdpServer(byte * rgbCache, size_t cbCache, int cMaxPendingClients)

Parameters:

rgbCache	A pointer to a buffer to be split evenly across cMaxPendingClients sockets to be used as the sockets datagram cache.
cbCache	The total number of bytes in the cache, each socket cache will be "cbCache / cMaxPendingClients" bytes.

cMaxPendingClients The Maximum number of unaccepted clients, or sockets, before UdpServer will stop listening. Unlike TcpServer this is a required parameter as it is critical in the calculation of the sockets UDP datagram cache.

Constructs an instance of the UdpServer class specifying the maximum number of unaccepted clients it will manage before it stops listening. In effect, this limits the maximum number of sockets consumed and gives you control over socket usage. A datagram cache is also required and the datagram cache will be split evenly across cMaxPendingClients individual socket datagram caches. The Cache must remain valid until UdpServer is destructed; just closing the instance does not release the cache.

Methods

bool startListening(unsigned short localPort)
bool startListening(unsigned short localPort, DNETcK::DNetStatus * pStatus)

Parameters:

localPort The port to start listening on.

pStatus An optional parameter to receive the status, this will typically fail only if a socket could not be obtained or an invalid / in use port number was specified.

Return:

True if a socket was acquired and listening on the port started. False if it could not start to listen and startListening() should be called again once the problem is corrected.

This will cause the UdpServer instance to start listening on the specified port and create UdpClients, and sockets as endpoint-pairs are detected.

bool isListening(void)
bool isListening(DNETcK::DNetStatus * pStatus)

Parameters:

pStatus An optional parameter to receive the listening status.

Return:

True if the TcpServer instance is currently actively listening, false if it is not listening.

isListening() returns the current actively listening state. It is possible for the server not to be listening because there are no more sockets to listen on, or because the maximum number of unaccepted clients has been reached, or because stopListening() was called. Listening will automatically resume when these conditions are corrected.

void stopListening(void)

Parameters:

None

Return:

None

Forces the UdpServer instance to suspend listening; call resumeListening() to resume. When stopped, all other server methods such as acceptClient() and availableClients() will work as expected. When stopped, the listening socket is returned to the system.

void resumeListening(void)

Parameters:

None

Return:

None

Resumes the UdpServer instance to its listening state; this is usually called to resume listening after a stopListening() was called. Just because resumeListening() is called does not mean that listening actually started. If a socket is not available, or if the cMaxPendingClients limit was hit, listening will not start. However, once these conditions have been corrected, listening will start automatically. Call isListening() to see if the server is actually listening.

void close(void)

Parameters:
None

Return:

None

This releases all resources except the datagram cache, and closes all unaccepted clients. The instance is returned to its “just constructed” state and startListening() may be called on the instance with a new port number. This will not close already accepted clients.

int availableClients(void)

Parameters:

None

Return:

The number of pending clients waiting to be accepted; zero if none are waiting.

This returns the number of pending clients waiting to be accepted via acceptClient(). This method is a workhorse method that checks to see if the listening socket has an endpoint-pair datagram on it and moves it to a pending client state, it will also check to make sure that there is a listening socket waiting for new connects, or attempts to acquire one if not. All processing is done first so that the number of availableClients() returned is as up-to-date as possible.

bool acceptClient(UdpClient * pUdpClient)
bool acceptClient(UdpClient * pUdpClient, int index)
bool acceptClient(UdpClient * pUdpClient, DNETcK::DNetStatus * pStatus)
bool acceptClient(UdpClient * pUdpClient, int index, DNETcK::DNetStatus * pStatus)

Parameters:

pUdpClient	A pointer to a “just constructed” UdpClient instance.
index	A zero based index into the availableClients() to accept. If omitted 0 is used.
pStatus	An optional parameter to receive the status, typically this returns just parameter checking status such as index out of bounds, or UdpClient is already in use.

Return:

True if the pending client was accepted, the UdpClient instance is initialized to the endpoint-pair, and UdpServer releases all management of the client. False if there was some reason the UdpClient could not be made.

acceptClient() releases a pending client to you. Once released, the server will not long manage the client in anyway. You must provide a pointer to a “just constructed” UdpClient instance. A UdpClient instance that was closed may also be used. When accepted, the server’s datagram cache is copied to the UdpClient’s datagram cache. If the client’s cache is smaller than the server’s cache some datagrams may be discarded. It is best to construct the UdpClient instance with a datagram cache equal to or larger than the UdpServers socket datagram cache to prevent datagrams being discarded. The index must be less than that returned by availableClients(). An index of 0 will return the first client detected as the server retains a first detect first accept order. acceptClient() may fail if no clients are available, the UdpClient instance passed is already in use (on another socket), if you passed in a NULL for the UdpClient instance, or if the index is out of bounds.

bool getAvailableClientsRemoteEndPoint(IPEndPoint *pRemoteEP)
bool getAvailableClientsRemoteEndPoint(IPEndPoint *pRemoteEP, int index);
bool getAvailableClientsRemoteEndPoint(IPEndPoint *pRemoteEP, MAC * pRemoteMAC, int index);

Parameters:

pRemoteEP A pointer to an IPEndPoint to receive the remote IP an Port.

pRemoteMAC A pointer to a MAC to receive the remote MAC address.

index A zero based index into the availableClients() to accept. If omitted 0 is used.

Return:

True if the remote endpoint information was obtained, false if the index was out of bounds.

getAvailableClientsRemoteEndPoint() allows you to peek at the endpoints of waiting clients. Call availableClients() to see how many clients are waiting and then loop through indexing each available client for its endpoint. Calling getAvailableClientsRemoteEndPoint() is safe and will not change the indexing order as this call does not "run" the server. However, if you call acceptClient() with the index you want, acceptClient() will change the indexing order as the accepted client is removed from the list and the server is "run". After an acceptClient() you will need to call availableClients() again and walk the new list from scratch. If you request a remote MAC, understand that this is the MAC as returned by ARP and will be a MAC of a machine on your local network. If the remote IP is not on your local network, the MAC will most likely be that of your gateway/router. Typically remote MACs are uninteresting and is why this is an optional parameter.

bool getListeningEndPoint(IPEndPoint *pListeningEP)

Parameters:

pListeningEP A pointer to an IPEndPoint to receive the listening IP an Port.

Return:

True if the listening endpoint information was obtained, false if an error occurred.

This is a somewhat redundant method but is provided for completeness and ease of use. The information returned will be your local IP address, which you can get from DNETcK: getMyIP() and the port number as specified on startListening(). getListeningEndPoint() may be useful if you are listening on several different ports and have several UdpServer instances and you wish to see who is who by retrieving their listening port number.