chipKIT™ Ethernet TCPIP Classes

Introduction

The chipKIT™ Ethernet library implements the standard Ardunio Ethernet library for use with the chipKIT™ MAX32 and chipKIT™ Network Shield. While the chipKIT™ Ethernet library mimics the Arduino Ethernet library class for class, method for method; the chipKIT™ classes implement a few additional features to support DHCP and DNS.

The Arduino Ethernet TCPIP library consists of 3 classes, the base Ethernet class that sets up your MAC and IP address, the Client class that initiates connections to servers, and the Server class that listens and connects with clients. The Arduino Ethernet class only supports static IP addresses and you must supply your MAC address. The chipKIT™ Ethernet class will use DHCP if no IP address is specified and will use the Microchip assigned MAX32 hardware MAC if no MAC address is specified. However, if you specify an IP address, the chipKIT™ Ethernet class will default back to using that address as a static IP address. In addition, the chipKIT™ Ethernet classes support DNS as well. If DHCP is used, the DHCP server will automatically populate your IP, subnet, gateway and DNS server addresses. If DHCP is not specified, you must supply a static IP and you are giving the option to supply up to 2 DNS servers. If no DNS server is specified, then the gateway address is automatically used as a default as many gateways act as a DNS forwarder. In addition, the Client constructor is overloaded to take a URL hostname, string IP, or IP array as a connection identity.

As with the existing Arduino Ethernet classes, the base Ethernet class is pre-instantiated in the library and has the instance name of “Ethernet”; there should only be one instance of the Ethernet class, do not instantiate another one in your sketch. However, neither the Client nor Server classes are pre-instantiated so you need to instantiate them as needed. You may have a Server class instantiation for each port you want to listen on, and you may have as many Client class instantiations as you like. An oddity of the Arduino Client class usage is that the Server class will create Client class instances each time Server.availiable() is called. This can create a proliferation of Client instances, many of which may point to the same socket; and is considered the same client. In addition, when the Client instance is destructed, say by going out of scope, the underlying socket is not closed. If you want to close the socket you must explicitly call the Client.stop() method. To help with knowing if 2 Client instances are the same, the chipKIT™ implementation of the Client class will allow you to compare the instances to each other (clientA == clientB). If they compare as true, they both are the same, that is, they both are connected to the same socket.

One issue to be aware of is that the chipKIT™ Ethernet library is implemented mostly in software whereas the Arduino implementation is mostly in hardware. As a result, the chipKIT™ implementation requires that the Ethernet stack must be called on a regular basis to keep the stack alive. While most Arduino sketches will compile and run just fine without modification (other than to specify the chipKIT™ Ethernet library instead of the Arduino Ethernet library), the chipKIT™ Ethernet stack will run smoother if each time through the loop() function at least one chipKIT™ Ethernet library method is called. This can be any of the Server or Client class

methods such as available(); or if no Ethernet methods are needed, the newly added method Ethernet.PeriodicTasks() should be called. This will ensure that the Ethernet stack stays alive and can service background tasks such as reading, writing or responding to pings. Specifically, when designing a sketch, replace any delay() functions with a timed while loop that continuously calls Ethernet.PeriodicTasks(), or if a forever/do-nothing loop is used to terminate a sketch's activity, than Ethernet.PeriodicTasks() should be called within the forever/do-nothing loop. Examples of the chipKIT™ versions of the TelnetClient, WebClient, and WebServer have been provided so you can see and compare with the original Arduino sketches. While the original Arduino sketches will compile and run as originally written with the chipKIT™ Ethernet library, they work better with the few minor changes as shown in the chipKIT™ examples; that is, to keep the Ethernet stack alive.

Ethernet Class

This class is pre-instantiated by the library and should not be instantiated by the sketch. The instance name is "*Ethernet"* and has the following methods.

Ethernet Class Instance Methods

void Ethernet.begin();
void Ethernet.begin(uint8_t * mac);
void Ethernet.begin(uint8_t * mac, uint8_t * ip);
void Ethernet.begin(uint8_t * mac, uint8_t * ip, uint8_t * gateway);
void Ethernet.begin(uint8_t * mac, uint8_t * ip, uint8_t * gateway, uint8_t * subnet);
void Ethernet.begin(uint8_t * mac, uint8_t * ip, uint8_t * gateway, uint8_t * subnet, uint8_t * dns1);
void Ethernet.begin(uint8_t * mac, uint8_t * ip, uint8_t * gateway, uint8_t * subnet, uint8_t * dns1, uint8_t * dns2);

Parameters:

mac: The MAC (Media Access Control) address for the device (array of 6 bytes). This is the Ethernet hardware address of your shield. On chipKIT™ boards this parameter is optional and defaults to the board's MAC address as assigned by Microchip and will have the form of 00:04:A3:xx:xx:xx. If you wish to use the default MAC address, yet still assign a static IP address, then pass an array of 6 zero's for the MAC address and the actual hardware MAC address will be used.

ip: The IP address of the device (array of 4 bytes). On the chipKIT™ boards this parameter is optional and defaults to using DHCP to obtain an IP address. If DHCP is indicated, the gateway, subnets and DNS servers will have no meaning as they will be overwritten by those values specified by DHCP.

gateway: The IP address of the network gateway (array of 4 bytes). This parameter is optional and defaults to the device IP address with the last octet set to 1 or is set by DHCP.

subnet: The subnet mask of the network (array of 4 bytes). This parameter is optional and defaults to 255.255.255.0 or is set by DHCP.

dns1: The primary DNS server to use (array of 4 bytes). This parameter is optional and defaults to the gateway IP as often gateways will act as a DNS forwarder. If you wish to override this and not specify any primary DNS server, stet this to 0.0.0.0.

dns2: The secondary DNS server to use (array of 4 bytes). This parameter is optional and defaults to 0.0.0.0

Returns:

None

This method initializes the Ethernet hardware, network settings and starts the Ethernet stack. For more information see the Arduino documentation at http://arduino.cc/en/Reference/EthernetBegin.

void Ethernet.PeriodicTasks(void);

Parameter:

None

Returns:

None

This method will execute the periodic stack tasks needed to keep the chipKIT™ Ethernet stack alive. This method is provided so that the stack can be maintained if no other Ethernet methods are going to be called for a long time. Either this method or any other Ethernet Server or Client method should be called at least once through the loop() function.

unsigned int SecondsSinceEpoch(void);

Parameter:

None

Returns:

The number of seconds since Jan 1, 1970.

This function obtains the current time as reported by the underlying MAL SNTP module. Use this value for absolute time stamping. The value returned is the number of seconds since 01-Jan-1970 00:00:00.

The underlying MAL SNTP module implements the Simple Network Time Protocol. The module updates its internal time every 10 minutes using a pool of public global time servers. It then calculates reference times on any call to SecondsSinceEpoch() using the internal Tick timer module.

The SNTP module is good for providing absolute time stamps. However, it should not be relied upon for measuring time differences (especially small differences). The pool of public time servers is implemented using round-robin DNS, so each update will come from a different server. Differing network delays and the fact that these servers are not verified implies that this time could be non-linear. While it is deemed reliable, it is not guaranteed to be accurate. In order for this method to work properly, a valid DNS server must be specified or DHCP must be used to obtain a valid DNS server so that the time server names may be resolved.

The millis() or micros() functions provide much better accuracy (since it is driven by a hardware clock) and resolution, and should be used for measuring timeouts and other internal requirements.

When calculating the current time remember that we use the Gregorian calendar where leap year occurs once every 4 years except on the century change where leap year is skipped. However, every 4 centuries the century leap year is not skipped. Year 2000 was one of those "don't skip" century leap years. So from Jan 1, 1970 to Mar 1, 2000 there was 30 years and 8 leap days ('72, '76, '80, '84, '88, '92, '96, 2000) and 59 days for a normal Jan, and Feb, for a total of 11017 days; or 951,868,800 seconds. But this is until Mar 1, 2000; after Feb 29th, 2000.

Client Class

An instance of the Client class represents a connection to a socket. The chipKIT™ Ethernet implementation allows for a total of 10 TCPIP sockets shared between all server and client connections. It is possible for many Client instances to represent the same socket; such "equal" Client instances only consume 1 socket.

Client Constructors

*Client(uint8_t *ip, uint16_t port);*
*Client(const char *szURL, uint16_t port);*

Parameter:

ip: A 4 byte array representing the IP remote address to connect to.

szURL: A zero terminated string representing the remote URL hostname to connect to. The URL will be resolved to an IP address through DNS. If no DNS servers are specified, the IP address may fail to resolve.

port: The port number to connect to on the remote machine.

This constructs a Client instance and specifies the remote machines address and port to connect to. The connection is actually made when the connect method is called. For more information see the Arduino documentation at http://arduino.cc/en/Reference/ClientConstructor.

Client Operators

operator bool();

Is true (if(clientA) {}) if the Client has a valid socket (it is a valid Client instance), false if the socket is invalid. This is different than a connected socket. Use the connected method to see if the socket is currently connected to the remote machine. This is an existing yet undocumented operator in the Arduino Client class. Documentation is provided here for completeness.

uint8_t operator==(const Client& otherClient);
uint8_t operator!=(const Client& otherClient);

The == and != operators allow for testing of 2 client instances to determine if they represent the same socket (if(clientA == client){}), that is they are the same client. If true they represent the same socket, false if they are different sockets. All invalid sockets will compare as equal to each other. This is a new operator provided only in the chipKIT™ Ethernet implementation.

chipKIT™ Client Methods

void SetSecTimeout(unsigned int cSecTimeout);

Parameters:

cSecTimeout: It is possible for it to take a long time to connect to a remote server or for large writes to complete. It is also possible for the connection to be dropped during these calls. cSecTimeout specifies the longest time in seconds that these operations should take before abandoning the attempt. By default this time is set to 30 seconds.

Returns:

None

Arduino Compatible Client Methods

These chipKIT™ Client class methods are functionally identical to the Arduino Client class methods.

uint8_t connect(void);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ClientConnect. This method will fail and abort its attempt to connect should the timeout period as specified by SetSecTimeout() is exceeded.

uint8_t connected(void);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ClientConnected.

int available(void);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ClientAvailable.

void write(uint8_t b);
void write(const char *str);
void write(const uint8_t *buf, size_t size);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ClientWrite. These methods will break the buffer up into multiple writes if the buffer is larger than the TCPIP internal write buffer. As a result these methods could take a long time to complete or the connection might be dropped during the attempt. These methods will abort if any individual write attempt exceeds the timeout period as specified by SetSecTimeout().

void print(…);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ClientPrint.

void println(…);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ClientPrintln.

int read(void);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ClientRead.

*int read(uint8_t *buf, size_t size);*

Parameters:

buf: This is a byte pointer to a buffer to receive the read data.

Size: This is the size of the buffer in bytes, and thus the maximum number of bytes that can be read.

Returns:

The actual number of bytes read, 0 if no bytes are available.

This is an implemented yet undocumented Arduino Ethernet method; it is documented here for completeness.

int peek(void);

Parameters:

None

Returns:

The next byte to be read; however this does not remove the byte from the input buffer, a subsequence call to read() will return this byte. If the input buffer is empty, -1 is returned.

This is an implemented yet undocumented Arduino Ethernet method; it is documented here for completeness.

void flush(void);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ClientFlush. A note of caution, this flushes the input buffer! Do not get this confused with the Microchip's Ethernet flush API which flushes the output buffer.

void stop(void);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ClientStop.

Server Class

An instance of the server class will listen on a port as specified in the Server class constructor.

Server Constructor

Server(uint16_t port);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ServerConstructor.

ChipKIT™ Server Methods

void SetSecTimeout(unsigned int cSecTimeout);

Parameters:

cSecTimeout: It is possible for it to take a long time for large writes to complete. It is also possible for the connection to be dropped during these calls. cSecTimeout specifies the longest time in seconds that write operations should take before abandoning the attempt. By default this time is set to 30 seconds.

Returns:

None

Arduino Compatible Server Methods

void begin(void);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ServerBegin.

Client available(void);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ServerAvailable. If there are no bytes to read on any connected Client, then an invalid Client instance is returned.

By checking the Boolean value of the instance (if(clientA){}) will indicate if the Client instance is valid or not. A false indicates that the client instance is not valid and no bytes may be read, a true indicates that the Client instance is valid and bytes maybe available for reading. Client.available() should be called to determine the number of available bytes to read. See the Boolean Client class operator for more information.

virtual void write(uint8_t b);
virtual void write(const char *str);
virtual void write(const uint8_t *buf, size_t size);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ServerWrite. These methods will break the buffer up into multiple writes if the buffer is larger than the TCPIP internal write buffer. As a result these methods could take a long time to complete or the connection might be dropped during the attempt. These methods will abort if any individual write attempt exceeds the timeout period as specified by SetSecTimeout().

void print(…);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ServerPrint.

void println(…);

See the Arduino Ethernet documentation at http://arduino.cc/en/Reference/ServerPrintln.

UDP Class

The UDP class is an undocumented class implemented by the Arduino Ethernet implementation. It should be understood that the implementation is not as complete as the TCPIP implementation and has some idiosyncrasies. Since the Arduino implementation is not documented, please be tolerant as to what is said here about the Arduino implementation as it is my best understanding of it, and I may very well be in error.

UDP is a datagram unreliable protocol, that is, there is no official connection made between client and server and each packet stands on its own and there is no guarantee that the packet will make it to its intended destination. As a result, writes are typically made in the blind, just blasting the packet out on the wire with the hope that the IP and MAC address is valid and that the internet will resolve the path to its destination. Be aware, if the IP or MAC is not valid or there is a network failure, the packet will be lost into the abyss with no feedback that it did not succeed. Typically, in order to obtain a MAC address, an ARP request is done on the IP to resolve the MAC address; this is at least some indication that the IP is valid. However, there is no need to make the ARP request if the MAC is already known; so it is possible to send a UDP datagram without an ARP. The chipKIT™ implementation will automatically do ARPs for you, and will cache the last MAC address resolved on each socket (UDP class instance).

In the Arduino implementation you can send (sendPacket()) to any IP/port address on any UDP class instance (socket), and you may receive from any IP on any socket. While the chipKIT™ implementation attempts to mimic this, it has a limitation that it will only receive on the same IP/port that was specified on the last sendPacket(); although each sendPacket() may switch to any IP/port on any call. Receiving on the same IP/port as what was sent on is usually not a

problem as what you usually want is to send/receive to/from the same IP/port on any given instance of the UDP class (socket). If you wish to listen for any IP on an open port, then create a new instance of the UDP class and specify the port on the UDP begin() method. But once data is received, the IP/port is set by the sender and is stuck on that IP/port until closed or until a sendPacket() is used to change it.[1]

An idiosyncrasies of the available() method is that it will return the number of data bytes available on the socket including the 8 byte UDP header. However, when reading the bytes (readPacket()) the header is stripped from the returned data and will be 8 bytes less than that reported by available(). The chipKIT™ implementation mimics the available() behavior by always reporting the number of data bytes (non-header) available, plus 8.[2]

The Arduino implementation seems to be a true datagram implementation, that is, what comes in on the wire, packet for packet, is what you get when you do a readPacket(). If 2 packets come in, the second packet will overwrite the first packet, even if there was room in the buffers for both.[3] The chipKIT™ implementation is based on a socket implementation and the data is buffered. As a result, available() will return the number of total data bytes received (+8), which potentially will be the concatenation of several UDP datagrams. The IP/port address reported by readPacket() will be the last IP/port sent to the socket and the buffer may potentially contain data from different remote IP/port senders; although this could only occur if a sendPacket() was used to change the IP/port before all of the data was read out of the socket.[4] The chipKIT™ implementation currently uses a buffer size of 1536 (0x600) bytes, and if more data comes in before the socket is read, the new data will overwrite the oldest data in the buffer.

chipKIT™ UDP Methods

void SetSecTimeout(unsigned int cSecTimeout);

Parameters:

cSecTimeout: When sending/writing URL hostnames may need to be resolved via DNS to an IP address; and a MAC addresses must be resolved from the IP address via an ARP call, as well as larger buffers may take a long time to transmit. cSecTimeout specifies the longest time in seconds that these operations can take before abandoning the attempt. By default this time is set to 30 seconds.

Returns:

None

---

[1] I can probably change this to mimic Arduino better by clearing the remote IP/port on the socket, but this would come at the expense of not caching the last ARP MAC resolution. Thus each and every sendPacket() would do an ARP request to resolve the MAC address.

[2] In my opinion, available() should not report the 8 byte header count.

[3] I have not specifically verified that datagrams will overwrite each other, but am making this assumption by code inspection. If knowing this for fact is important to you, please verify this for yourself.

[4] It is possible for the chipKIT™ implementation to retain only the last datagram, but was considered more useful to buffer as much data as possible as typically reading and writing, once started, is on the same IP/port and keeping as much data as possible is desirable.

Arduino Compatible UDP Methods

uint8_t begin(uint16_t port)

Parameters:

port: This is the port to listen on.

Returns:

None

int available()

Parameters:

None.

Returns:

The number of bytes available for reading plus 8. The 8 represents the unreadable 8 byte UDP header. The chipKIT™ implementation just adds 8 to the number of available bytes in an attempt to be compatibility with the Arduino implementation.

*uint16_t sendPacket(uint8_t * rgbBuff, uint16_t cbBuff, uint8_t * rgbIP, uint16_t port)*
*uint16_t sendPacket(uint8_t * rgbBuff, uint16_t cbBuff, const char szURL[], uint16_t port)*
*uint16_t sendPacket(const char sz[], uint8_t * rgbIP, uint16_t port)*
uint16_t sendPacket(const char sz[], const char szURL[], uint16_t port)

Parameters:

rgbBuff: A pointer to an array of bytes to send. The array must be a complete datagram; that is, the packet will be immediately transmitted on the wire.

cbBuff: The number of bytes to send.

sz: A zero terminated string to transmit.

rgbIP: A 4 byte array specifying the remote IP address to send the datagram to.

szURL: A zero terminated string representing the hostname to send the datagram to. DNS will be used to resolve the hostname to an IP address.

port: The remote port to send the datagram to.

Returns:

The number of bytes actually transmitted.

This may be cut short if the timeout value aborted the transfer, or zero if the DNS/ARP failed to resolve the IP/MAC address. If a broadcast is desired, set the IP address to 255.255.255.255 and the port address to 0xFFFF.

*int readPacket(uint8_t * rgbBuff, uint16_t cbBuff)*
*int readPacket(uint8_t * rgbBuff, uint16_t cbBuff, uint8_t * rgbIP, uint16_t * pwPort)*
*int readPacket(char * sz, uint16_t cbsz, uint8_t * rgbIP, uint16_t &wPort)*

Parameters:

rgbBuff: A pointer to a byte array to receive the read data.

cbBuff: The maximum size of rgbBuff in bytes.

rgbIP: The remote IP address who sent the data.

pwPort: A pointer to a WORD to receive the remote port of who sent the data.

wPort: A “by Reference” WORD to receive the remote port of who sent the data.[5]

sz: A pointer to a character array to receive a zero terminated string.

cbsz: The maximum number characters (bytes) in sz including the zero terminator.

Returns:

The number of bytes read.

The will not return the 8 byte UDP header.

void stop()

Parameters:

None.

Returns:

None.

This closes the UDP socket and releases all resources back to the system. Destructing the class does not close the socket, it must be explicitly stopped.

[5] Not sure why &wPort is by reference rather than a pointer. My only thought is that this might have made the method signature significantly different as to not conflict with the uint8_t*/char* of the other method. However, the Microchip compiler distinguishes between uint8_t* and char* just fine and &wPort could have been a WORD* as in the other method. However, for compatibility with the existing implementation, this method signature was not changed.