# Teensy Audio Library EtherNet objects

V1.0 December 2019

The Teensy audio ethernet library provides several basic services.

* Ethernet interface control using the common Wiznet modules and Paul Stoffregen’s Ethernet library to drive them.
* The ability to operate on a 10/100 Mbit Class C network.
* Ethernet audio streaming between hosts – delivered via an output object on one host, and an input object on another
* The ability to broadcast a stream to all hosts on the network.
* The ability to subscribe an input object to one of the available incoming streams.
* A messaging service between hosts – accessible at the user program level.

### Ethernet audio design approach

Implementation assumes a standard 10/100 compatible ethernet network. Class C networking is assumed. A single network adapter is assumed, running a single UDP processor, listening on a single Port.

Audio and control packets are processed and queued (using its own queuing capability) by the ethernet\_control object, and are then processed into Teensy audio streams by separate input and output objects. This is because multiple input and output objects can be supported by a single ethernet control. Where audio blocks are being queued by the ethernet control, audioStream allocate() and release() are used for the audio buffers attached to each queue item.

### V1.0.0 release

Sufficient functionality is available to handle multiple two-channel audio transmissions from a master Teensy to any number of clients, with a set of basic control and status functions. Other configurations may work, but are largely untested in the initial release.

## Streams & Packets

As ethernet speed is limited, the overhead in each UDP audio packet is kept as small as possible. Other than audio data, only a packet type, netAudioStream ID and a packet sequence number (4 bytes total) are included in each transmitted audio packet.

Each audio stream is described by a netAudioStream structure, maintained by the transmitting host and shared with all clients via status messages.

Control packets are generally shorter than audio packets, so all meta-information is transmitted with each packet.

## Technology

### Ethernet

Any ethernet module supported by the ethernet library may be used. However, it is strongly recommended that Wiz5500 or later modules supported by the ethernet library are used. Wiz5100 modules are too slow for more than the most basic audio traffic (see Paul’s benchmarks at <https://www.pjrc.com/arduino-ethernet-library-2-0-0/>), but are adequate for most other traffic types, e.g. control traffic and MIDI.

Hosts are interconnected by standard 10/100 ethernet switches. For best performance, a stand-alone network should be used, as QoS is difficult to guarantee with other network traffic.

Given current ethernet library v2.0 / Wiz5500 (1.1MB/sec) performance, each interface could theoretically support 8 channels, however only 2 are supported at this stage. 4 could be supported without resorting to Jumbo packets which UDP may not correctly re-assemble.

Only UDP packets are supported. One UDP port (8888) and socket is used for all audio and control packets.

For best operation, several options need to be enabled in the Ethernet library.

Ethernet.h (line 39) should be modified to enable bigger RX/TX buffers (the library is normally in your Arduino installation’s hardware/teensy/avr/libraries/ethernet/src/ folder).

#define ETHERNET\_LARGE\_BUFFERS // Line 48

#define MAX\_SOCK\_NUM 2

W5100.h (lines 21 & 28) should be modified to enable fast SPI (in utilities subfolder of above library).

#define SPI\_ETHERNET\_SETTINGS SPISettings(30000000, MSBFIRST, SPI\_MODE0) //

Which will increase the input and output buffers from 2kB to 8kB. The library will work without making these changes, however processing will be slower (SPI speed) and may result in higher packet loss (buffer size).

### SPI

The SPI interface is quite busy – as chip control traffic, as well as audio packets need to pass over SPI. It is not recommended to share other SPI traffic with Ethernet, if audio traffic is to be transmitted.

Also, performance is substantially improved if the SPI speed is increased to 30Mbs (see above).

#### Addressing and protocols

UDP is supported over IP as the most straightforward means of asynchronously passing individual packets on information between hosts.

Control\_ethernet supports one IP address, and one port –for audio and control packets.

IP addressing is Class C, with 192.168.1.X as default. The Class C subnet can be changed in code ( setMyNet() ) to anything that the Ethernet library supports.

The hostID ( setMyID() ) should be set after the network address is changed, as it also sets the interface IP address.

The library uses broadcast addresses for some control functions and for “universal” audio streams.

#### MAC addresses

MAC addresses are set dynamically to “DE AD BE EF FE xx” where xx is the hostID.

#### Ports

One ports is currently used (8888) for audio streams and control/status messages. This simplifies socket handling at the Ethernet interface.

The library supports 255 packet types on each port, providing flexibility for audio, control and other traffic. Some packet types are reserved (see below).

### Maximum audio channels

The practical limitations on audio channels are twofold.

* The Wiznet devices cannot handle jumbo packets, imposing a limit of 1500 bytes including Ethernet and Datagram headers, thus 4 channels x 128 samples could be carried in a single datagram (see below).
* SPI appears to be the limiting factor in transmission speed – see <https://www.pjrc.com/arduino-ethernet-library-2-0-0/>  
  Teensy 3’s and 4’s can handle ~1 MB (10 Mb) /sec throughput, with SPI set to 30Mb/sec.  
  This equates to 2900 bytes/update() cycle – or ~12 channels total.

To avoid the SPI bottleneck, avoid broadcasting audio streams (i.e. transmit to specific hosts) wherever possible.

### Teensy

Teensy 3.5, 3.6 or 4.0 are recommended, both for processing power and SPI speed.

To avoid differences in audio clock speeds, which result in packet loss, it is recommended that all Teensys are of the same (or compatible types). To test this, turn on CE\_SYNC\_DEBUG in ethernet\_control.cpp this will generate a square wave on Pin 3 (even though the #define is for pin 2) at half the update() frequency.

## Audio Objects

There are currently three Ethernet audio objects.

* **Control\_ethernet** – which handles the ethernet interface and all raw datagram traffic. It currently uses UDP/IP for all transport.
* **Input\_net** – which converts Ethernet datagrams into audio streams.
* **Output\_net** – which converts audio streams into Ethernet datagrams.

### Control\_Ethernet

One control\_ethernet object manages all communications for a physical ethernet interface (e.g. a Wiznet module).

It handles low level communications management and interface setup. It also handles all Datagram traffic, as multiple input and output audio objects may subscribe to its services.

Audio processing and logical audio functions are in input\_net.xxx or output\_net.xxx. They are friend classes with this class.

Relatively little error checking is in place – other than malformed packet rejection by the Wiznet chip. Audio packets are tagged with an 8 bit sequence number, providing for a 256 packet window to detect dropped packets. While dropped packets are detected, they are not handled in any way by the library. If no packet is available for an update cycle, the input object transmits a block of zero samples.

Unlike TCP, UDP does not automatically resend broken packets. This could be implemented in code, but has not, to avoid packet storms and delays to (synchronous) packet transmission.

### Input\_Net

Convert Ethernet datagrams into audio streams.

Input\_net currently implements 2 channels (one stereo audio stream).

Several input or output objects may be defined on a single host, if more channels are needed to a single target, or multiple independent streams are needed.

Multiple input objects can subscribe to the same stream.

### Output\_Net

Convert audio streams into Ethernet datagrams.

Two inputs are currently implemented.

Several objects may be defined on a single host, e.g. to share audio to different hosts, or more channels to a single host (or broadcast).

## Net audio streams

control\_ethernet maintains a streamsIn[ ] list and a streamsOut[ ] list.

A stream is fully identified by its streamID and originating hostID

Each sending host regularly transmits a status packet for each stream it generates – to the same target address(es) the stream is sent to (i.e. individual hosts or broadcast). This occurs approximately every 4 seconds (UPD\_MSGS\_SECS – by each output object’s update() code).

To receive audio data, user code needs to subscribe to an incoming stream ( subscribeStream(id) ), after checking available streams by parsing the output of getStream(id) for each id in   
[0 .. getActiveStreams()]. Input objects will deliver silence when unsubscribed.

It should be noted that all stream data may not be immediately available on start up, and streams may become temporarily or permanently unavailable due to hosts being unavailable (e.g. unplugged or powered off). While the transmitting host and its streamID may be the same each time, the id value passed to subscribeStream(id) may differ, as it is determined by which status packets arrive first at the receiving host. ALWAYS identify the correct stream to subscribe, by comparing hostID and the host streamID.

If a stream becomes unavailable (getStream(0 returns this information), user code can choose to release the stream ( releaseStream(ID) ) or wait for resumption. All-zero content (silence) blocks will be provided while the stream remains subscribed if there is no reception of that stream.

## Class structures and Datagram types

Each datagram has a byte-length type identifier, included as the first character of the payload. This determines where the datagram is handled, and how it is routed.

|  |  |  |
| --- | --- | --- |
| **pktType** | **Use for** | **Conditions** |
| 0 | Reserved |  |
| 1-15 | Audio data packets | Content[] payload must be a multiple of 128 samples (256 bytes).  1500 byte UDP/IP total packet including Ethernet and Datagram headers.  Maximum transmission rate for the microcontroller must not be exceeded. |
| 16-31 | MIDI or other Audio library “content” packets (TBD) |
| 32-63 | library ethernet audio object control/status  32 - stream info broadcast | 256 byte payload limit (audio buffers are used for processing), headers are stored separately. |
| 64-95 | MIDI or other Audio library control/status packets |
| 96-127 | Ethernet library control and status messages.  96 – Ethernet / UDP    98 – Other audio object control and status  110 – MIDI control/status  General control and status messages ??? |
| 128-255 | Available for user-defined packet types |

As well as the packet type, further differentiation may be included in the payload structure, e.g. the following may be the payload of a user control packet, say pktType == 99:

typedef struct myMessage

{

byte mySubType;

short data1;

float data2;

} myMsg;

### audioDatagrams

All audio packets are transmitted in an audioDatagram wrapper. The datagram has the packet type (byte), packet *sequence number* (byte), and the netStreamID (byte), followed by the audio samples. The packet type defines the size of the payload.

The packet *sequence number* is designed to detect dropped packets, however little processing for this has been implemented. No input packet being available on update simply results in audio silence.

Audio sample size (16 bits), rate (44.1 KHz) and samples/block (128) are set by core audio library.

Currently the two channel samples are sequential (Type 2 packet = default. See audio2Datagram).

Audio datagrams should not exceed the MTU limits of the network hardware (generally 1500 bytes) when wrapped in IP & UDP (28 bytes overhead). The audio packet overhead is 4 bytes.

More channels can be supported by duplicating input and output objects.

|  |  |  |
| --- | --- | --- |
| Channels | Sample bytes | Total Packet |
| 2 | 512 | 538 |
| 4 | 1025 | 1050 |

Packet types 1 to 15 are reserved for audio datagrams

***< stream struct here>***

### controlDatagrams

All other communications are delivered in control datagrams

* Command messages from one host to another
* Status messages (such as Stream info, channel names, etc)
* Other payloads (e.g. MIDI messages)

### audioControlDatagrams

**netAudioStream** (see above) is transmitted regularly to all target hosts.

### Queue and buffer management

256 byte buffers from the audio pool [ allocate() and release() ] are used for datagrams stored in the input and output queues (see above). AudioMemory() needs to be set high enough to accommodate queued packets. A starting value of 40 is suggested, monitored and adjusted as needed.

Received datagrams are aged each update() cycle ( PKT\_DECAY\_A and PKT\_DECAY\_C ) and are discarded if not released by client objects, to reduce memory loading. Generally, 2 cycles should be sufficient for double-buffered content to be processed.

### General Messaging Protocol

To enhance multi-host audio set-ups, a general messaging protocol has been provided for inter-host communication. pktTypes 128-255 are available for this use.

User-level control packets, while aged and deleted by the queue manager, should be regularly processed by the user program. This is especially important if multiple control messages are sent each update() cycle (e.g. MIDI notes)as the standard control queue is quite short (MAX\_CONTROL\_QUEUE 32). At minimum, the following code – which discards any incoming user control messages should be run at least once each update cycle, to avoid running out of control queue slots. The overhead is low, so it can be run each loop() cycle.

controlQueue cQbuf;

loop()

{

while (EtherNet1.getQueuedUserControlMsg(&cQbuf))

if (cQbuf.pktType == 196) // my defined packet type.

; // do something with this packet

else

; // discard all other control messages

}

## To Do

### PoE

It is possible, at a later date that POE may be supported allowing Teensy modules to be line-powered. 12-20V DC is proposed allowing (cheap MP2307-based) buck converter modules to be used for local supply at whatever voltage is required, to reduce line losses.

The only modules currently offering PoE are the Funduino 5100 modules, which use a non-standard (pin 7 = neg & 8 = pos) PoE connection.



The standard 48V PoE arrangement uses pins 4&5 = pos, 7&8 = neg. On most Wiznet modules pins 4&5, 7&8 are terminated 75 ohms to ground, making PoE difficult to implement with standard PoE ethernet switches or injectors. A standalone PoE injector board could be created to fulfil this purpose. The switch would need to be protected from the non-standard voltages.

### Packet validation

Check packet length from UDP handler against calculated stream packet size, or headers + payload calculation.

It does not appear that UDP Checksums are validated in the Wiznet or UDP libraries.

## Bugs

1. Differing Audio clocks (T3.x – T4.0) causes dropped packets every 2,500 packets – 0.05% loss rate.