Dynamic Audio Library

This document gives a brief overview of the “Dynamic Audio Library”, an update to the Teensyduino Audio library code which aims to enable construction, use and destruction of AudioStream and AudioConnection objects at runtime, rather than constraining them to be defined solely at compile time.

Audio library concepts – original

Update cycles

The audio library runs an update cycle at regular intervals, under interrupt, causing all active AudioStream objects to process as necessary to keep a flow of audio data to the outputs. With 128-word audio blocks and an audio rate of 44.1kHz, a cycle needs to be run every 2.9ms.

The update cycle relies on a linked list of AudioStream objects, which are processed in linkage order until all active linked objects have had their `update()` function called.

Foreground processing

Between update cycles the application is free to run foreground processes as needed. These may consist of receiving MIDI data, sensing button presses, potentiometer or encoder changes, and modifying AudioStream objects’ behaviour via their published APIs. The library provides `AudioNoInterrupt()` and `AudioInterrupt()` functions to ensure that any such changes can be made so as to leave the system in a consistent state.

Audio block

An audio block (`audio_block_t`) is a structure used to pass a fragment of audio data between two AudioStream objects. Typically this will be 128 signed 16-bit integers (plus some overhead), though is it possible to re-define this. A fixed-size pool of such blocks is defined at compile time, and AudioStream objects may allocate, transmit, receive and release them as needed.

In a well-ordered system few will be needed, as any block transmitted to one or more recipients can be released as soon as all recipients have processed it. However, some AudioStream objects need to retain a number of blocks as buffer memory, and a system which is imperfectly-ordered or contains loops can also result in some blocks persisting in use between update cycles.

Audio blocks have no record of which AudioStream objects refer to them, but do have a reference count which, if it drops to zero, allows the library to release the block back into the pool of available blocks. AudioStream objects, on the other hand, must retain a record of which blocks they (co-) own, in order to make use of them and release them when finished with.

Audio connection

An AudioConnection object (`AudioConnection`) provides a connection between two AudioStream objects, a source and a destination. A source may "feed" multiple destinations, but a destination must only have a single source. Every AudioStream object has a `destination_list` member which points to a linked list of AudioConnection objects, linking it to its destinations. Transmitting an audio block effectively notifies each destination that new audio data is available.

AudioStream object

The AudioStream object (`AudioStream`) forms the base class for the ecosystem of derived classes which produce, process and emit audio data. As noted above, each active object's `update()` function is called during an update cycle, and (typically) the resulting audio is eventually transmitted via a hardware interface (I2S, USB etc.) so it may be heard.

Compile-time definition

With the existing Audio library, essentially all of the foregoing topology is defined at compile time. In particular, AudioStream and AudioConnection objects are statically allocated, and cannot be changed. The order of the update cycle's linked list is dictated by the order in which AudioStream objects were defined.

Dynamic audio library – additional concepts

The aim of the dynamic audio library is to retain as much as possible of the existing ecosystem, while allowing more flexibility in "re-wiring" a system at runtime to suit changing needs. To that end, it must be possible to create and destroy AudioStream and AudioConnection objects, and change the connections defined by the AudioConnection objects.

Update cycles

These remain as before, but as ever it is preferable to attempt to keep a logical ordering of the update list. This can be achieved initially by ensuring that when an AudioConnection is made, the source AudioStream is updated before the destination, i.e. the source is linked into the update list first. The definition of "first" may be unclear if a connection loop is created:

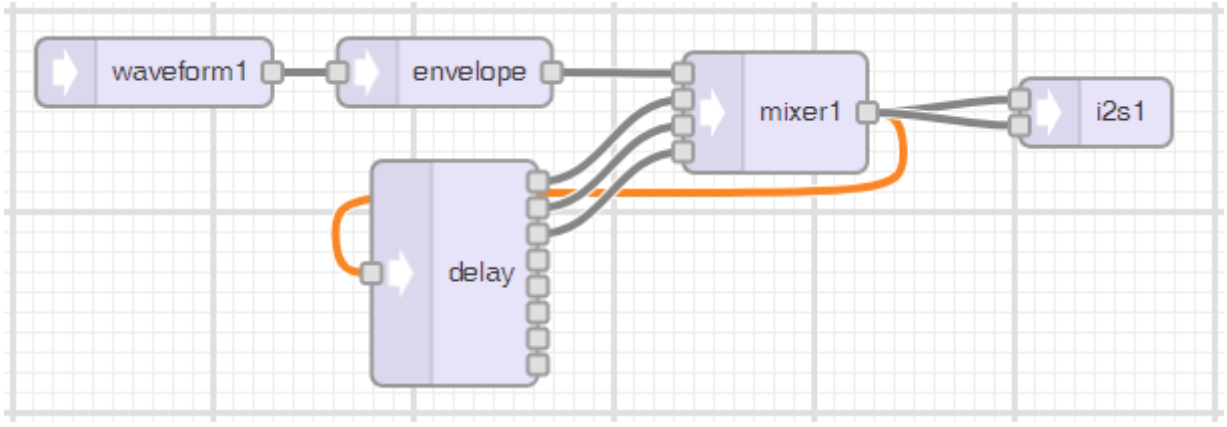

Figure 1: Example of a looped connection

Foreground processing

No changes are required here, apart from to note that objects created but then going out of scope may lead to unexpected results, but must not result in a crash.

It is the responsibility of the foreground processing code to keep track of all dynamic AudioStream and AudioConnection objects extant in the system at any given moment. Should any become “lost” this would constitute a memory leak; if they are destroyed but a reference is retained then there may be an attempt to use them: either of these is likely to result eventually in system instability.

Audio block

Again, no changes are envisaged to these at the moment, though an ability to add to the available pool would potentially be helpful.

Because an AudioStream object may have a number of blocks allocated to it at the time it is destroyed, measures must be taken to ensure blocks do not become “lost” to the pool. This is dealt with in the AudioStream section.

Audio connection

In order to change a system’s topology, it is necessary to be able to construct, connect, disconnect, reconnect and destroy AudioConnection objects freely.

- AudioConnection objects may be created with `new`, and destroyed with `delete`
- In addition, they may be defined as automatic variables in a block, and hence destroyed when the block ends
- API functions provided are:
    - `connect(src,dst)`: connects src AudioStream output 0 to dst AudioStream input 0
    - `connect(src,n, dst,m)`: as previous, but output n to input m
    - `connect()`: reconnects a connection that was disconnected, provided the AudioStream objects concerned are still in existence, and the relevant input “pin” is not in use
    - `disconnect()`: disconnects an existing connection: does *not* destroy the object

Forming a connection defines the update order of the AudioStream objects involved, *if* it has not already been defined by a previous connection. If neither AudioStream object is connected to the active update list, the objects may remain inactive and consume no processor time during an update cycle (see Additional AudioStream API functions). A set of inactive AudioStream objects with interconnections will always become active and linked in to the update list as soon as the first connection is formed between any of its members and an active AudioStream object.

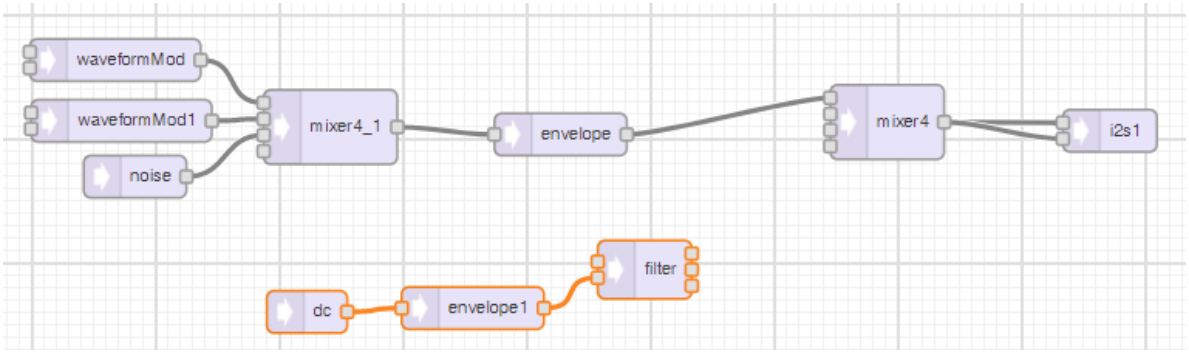

Figure 2: Orange highlighted objects are active by default, but may be made inactive

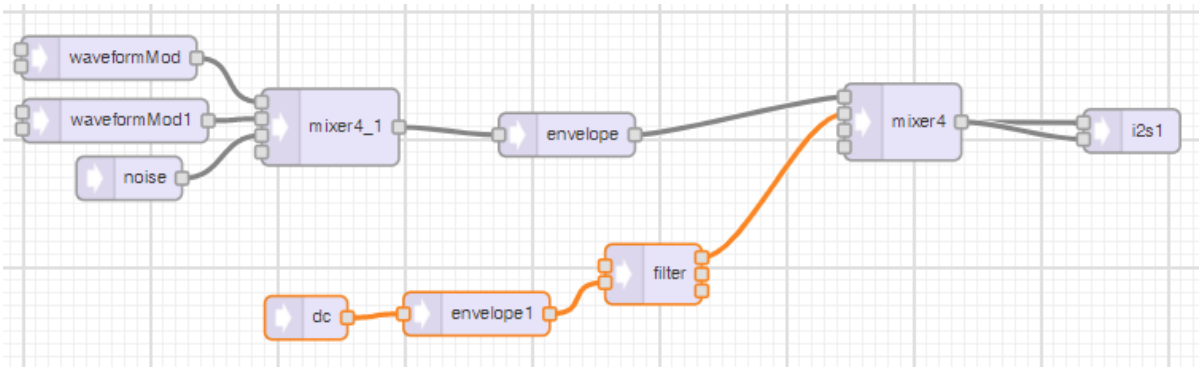

Figure 3: Objects are now active, will execute after envelope *and before* mixer4

[At the time of writing, AudioStream objects only become *unlinked* from the active update list when *all* their connections have been disconnected]

AudioStream objects

In order to change a system's topology, it is necessary to be able to construct, connect, disconnect, reconnect and destroy AudioStream objects freely.

- AudioStream objects may be created with `new`, and destroyed with `delete`
- In addition, they may be defined as automatic variables in a block, and hence destroyed when the block ends
- No additional API functions are required for the AudioStream base class
- Every AudioStream derived class *must* now provide a destructor to ensure it is destroyed cleanly, and in particular that **any audio blocks it has allocated to it are released cleanly**
    - to aid in this, AudioStream.h now provides macros which can assist in releasing a single or multiple audio blocks in a safe manner
    - Any memory associated with (but not part of) the object must be dealt with explicitly by the foreground code after the AudioStream object has been destroyed. An example of this is the AudioEffectChorus object, which is supplied with a pointer to its working memory when `AudioEffectChorus::begin()` is called: as the library has no way of knowing how this memory was acquired, it is unable to free it up.
- The AudioStream base class destructor ensures all associated AudioConnections are disconnected and the AudioStream object is unlinked from active or inactive update lists as required.

Block release functions and macros

AudioStream.cpp has an additional variant of the `release()` function to simplify releasing an array of audio blocks, and both variants have an optional boolean parameter to determine whether interrupts are re-enabled after the release has been done:

```
static void release(audio_block_t * block, bool enableIRQ = true);

static void release(audio_block_t** blocks, int numBlocks, bool enableIRQ = true);
```

AudioStream.h provides two new macros which are intended to ensure safe release of any implementation-specific audio blocks associated with an AudioStream object that is being destroyed. Note that it is NOT required to release the blocks pointed to by the `inputQueue` array, as these are part of the base class and its destructor takes care of those.

Both macros disable interrupts and set the object to be inactive prior to executing any other code: it is advisable, therefore, to make a call to one of these prior to any other code in the object's destructor. DO NOT re-enable interrupts in your derived class's destructor: this will be done at the end of the base class's destructor.

`SAFE_RELEASE(…)` calls `release(__VA_ARGS__)`, typically to release a single audio block:

```
class AudioInputAnalog : public AudioStream
{
public:
     ~AudioInputAnalog() {SAFE_RELEASE(block_left);}
private:
     static audio_block_t *block_left;
};
```

It can also be used to release an array of blocks:

```
class AudioAnalyzeFFT1024 : public AudioStream
{
public:
     ~AudioAnalyzeFFT1024() {SAFE_RELEASE(blocklist,8);}
private:
     audio_block_t *blocklist[8];
};
```

Releasing multiple blocks with disparate names requires the use of `SAFE_RELEASE_MANY(n, ...)`:

```
class AudioInputI2SQuad : public AudioStream
{
public:
     ~AudioInputI2SQuad()
     {
       SAFE_RELEASE_MANY(4,block_ch1,block_ch2,
                         block_ch3,block_ch4);
     }
private:
     static audio_block_t *block_ch1;
     static audio_block_t *block_ch2;
     static audio_block_t *block_ch3;
     static audio_block_t *block_ch4;
};
```

[At the time of writing `AudioEffectDelayExternal` needs a mechanism to release the external memory allocated to it]

Additional AudioStream API functions

A number of functions are provided to enable or disable AudioStream objects which do not form part of the normal update list (see Figure 2). To preserve consistency of operation, from version 0.2-alpha of the library onwards these objects *will* by default execute and consume CPU time. They may be disabled individually or globally. A set of connected objects is termed a “clan”.

- `setClanActive(bool)`: calling this function on any object will enable or disable execution of it and all other objects in its clan
- `bool isClanActive()`: returns `true` if this object is a member of an active clan
- `static setAllClansActive(bool)`: calling this function globally or on any object will enable or disable execution of all objects not in the normal update list
- `static bool areAllClansActive()`: returns `true` if execution of all objects is enabled

Separate flags are used for enabling individual and global execution, so executing `object.setClanActive(true)` followed by `AudioStream::setAllClansActive(false)` will leave `object` (and all other members of its clan) executing.