Comments on Audio Devices

Document No P1913r0
Date October 6, 2019
Project The C++ Programming Language
Audience SG 13 – I/O
Authors Frank Birbacher (Bloomberg LP), frank.birbacher@gmail.com

In regards to paper P1386, revision 2, we discuss use cases that we don't see supported. We want a portable sample type, an abstract concept for devices to allow third party additions, a unique identifier for devices to find them on the next run, and a null device to always be available. We want to let the OS suspend the processing of audio and an easy way to shutdown gracefully.

Contents

1 Revisions

Initial revision.

2 Introduction

In this paper we'll give feedback on the P1386, revision 2. We'll list a number of use cases which we'd like to see supported by the C++ audio facilities and our proposed changes to P1386. Each use case has its own section which opens with the use case as a statement. It follows a discussion on why we think this use case is not met with the current proposal and then a subsection on what we think needs to be done. Open questions that we haven't fully explored yet are given last, if any.

Listing 1: Force support of one sample type

```
static_assert(audio_device::supports_sample_type<short>());
```

3 Use Case: Portable Code

> As a C++ user I want to write a piece of Standard-only code that outputs or records audio and can run on any conforming C++ implementation even if it isn't the most efficient way of achieving that effect.

Standard conforming C++ code should allow to write portable software including all the facilities the Standard offers. One can use all C++ facilities and they have defined behavior even on systems where that facility isn't implemented in the most efficient way. By using implementation defined knowledge one can write more performant programs while tying it to the respective implementation.

With sample types in the audio proposal there is no single good value which forces users to code for an unknown set of types. Compile time queries can possibly check for certain integer and floating point types, but not for implementation defined types. If there was a typedef for a sample type then it's properties would still need to be discovered through type traits: is it integer or float, is it signed, how many bits does it have? This work should be done by the implementation.

3.1 What We Propose

- Support the sample type `short` everywhere.

The implementation must apply required conversions on the audio buffers to match the sample type in use by the hardware. Choose an integral type to avoid issues on hardware without floating point support. Choose a 16 bit type that can easily be converted to float and smaller int types for whatever the actual hardware supports, yet has reasonable resolution to be usable. An integer type with at least 16 bit is supported on all implementations and `short` is the smallest realization of such. Using `short` over `int16_t` or similar typedefs allows to safely overload functions and specialize templates. Choose a signed type with an unambiguous zero-is-silence value.

Listing 2: How to provide third party devices

```
namespace someVendor {
    class XAudioDevice { ... };
        // Models the std AudioDevice concept.

    class XAudioList { ... };
        // Models the std AudioDeviceList concept.

    XAudioList audioInputList();
    XAudioList audioOutputList();
} // close namespace
```

4 Use Case: Third Party Library

> As an audio hardware vendor I want to provide a C++ library to my clients that can easily fit into existing code which uses C++ audio functionality.

The proposed design for audio devices in P1386 defines a class `audio_device` whose objects can be obtained through a few library functions. There is a fixed set of functions to obtain devices that don't allow extension. The lack of constructors in `audio_device` prevents the user of the standard library to construct other instances. This also applies to third party vendors who would like to make their proprietary devices available as an audio device in the same way the Standard does.

4.1 What We Propose

Consult the example in listing 2.

- Introduce a concept of an audio device and describe the concrete `audio_device` in terms of it.
- Introduce a concept of a range of audio devices and describe `audio_device_list` in terms of it.

If in addition to the concrete `audio_device` the standard contained an abstract concept for audio devices, of which the concrete is an example, third party vendors could supply classes of their own that model the same concept. In terms of wording this there would be little change to the text. Instead of specifying a class `audio_device` with its member functions we only need to say "a class A is an audio device, if it has the following members with the following semantics."

Listing 3: Device descriptor with identifier

```
class audio_device_descriptor {
    // Value type to identify an audio device.
public:
    string_view vendor_id;
    string device_name;
    string device_id;

    // Add ordering and hashing support.
};

optional<audio_device> get_audio_device_by_id(string_view id);
```

5 Use Case: Discover Same Device on Next Run

> As an application developer I want to allow my users to select an audio device from a list of all devices provided by native C++ and third party vendors, as far as supported by my application, where the list has human readable names or descriptions such that the choice can be persisted in a file on disk and programmatically selected again on the next run of my application.

The proposed design for audio devices states that a device has a human readable name as one of its attributes, but doesn't need to be unique. The proposed device id is unique, but also implementation defined. There is no portable way to write it to a file. This poses the question of how to select a particular device again given on a subsequent run of a program.

Also neither the name nor the id needs to be stable in any way because there is no intent stated. It's not stated whether in case a device is removed and later added again its id will remain the same. Additional discussion on this:

- Require the name to be non-empty. This allows to differentiate from a default constructed string when processing a list of devices and is for convenience. It also prevents a UI from showing an empty entry without any visible characters.
- The string content should have some specific encoding. By saying nothing we would assume the execution character set. An alternative would be to specify UTF-8 to allow rich names.

5.1 What We Propose

Device Identifier

- Rename the existing `device_id` to `native_handle`.

- Introduce a new method `device_id` to return a unique string.
- This identifier shall be unique among the devices from the same vendor.

The identifier cannot be empty and must contain only `isgraph` characters from the execution character set. It is unique among the devices discovered through the union of `get_audio_input_device_list` and `get_audio_output_device_list`. Make a note that the identifier is intended to be linked to the serial number of the audio hardware that's being used. This means that plugging two physical devices of the same model of sound card into the same computer they show up as different identifiers, but might show the same name. Swapping one for the other on the same USB port, for example, would also change the identifier.

Value Type for Descriptor with Vendor Identifier

- Introduce a notion of a vendor identifier as a string which every third party vendor can choose.
- Introduce a value type to hold a tuple of the vendor id, device name and device id.
- Make a function to get a device given its id.

The vendor string should be a global static string from the execution character set. Choose an identifying string for the standard library. Define equality, ordering, and hashing on the descriptor type.

5.2 Open Questions

- Should the device lists give devices or descriptors?

Listing 4: Null device always present

```
audio_device get_null_audio_device();
    // Return the device that discards output and produces
    // zero bytes as input.

// or
audio_device make_null_audio_device(string_view name);
    // Make a null device with the given name.
```

6 Use Case: Null Device Always Present

> As an application developer I want to avoid the hassle around checking for "no devices" in a number of places: when instantiating my audio related classes, when persisting a choice to disk, when loading everything from disk, and possibly when running tests.

The null device was the proposed fall back for C++ implementations that don't support any audio hardware. But it could also be supported by implementations that actually have real devices. It could be that all audio devices have been unplugged or are in use by other applications. In this case they could still supply a null device. So why not have it be always available?

6.1 What We Propose

Compare listing 4.

- Introduce an instance of `audio_device` that can act as an output and input device, but discards all output and provides zero bytes as input.

Make that instance available via a new library call `get_null_audio_device`. Give it a fixed identifier "NullDevice" that is the same across all implementations.

6.2 Open Questions

- Should this device be included in the devices discovered by the list functions `get_audio_input_device_list` and `get_audio_output_device_list`?
- Should this device be supplied through a factory function that allows multiple instances to be made?

7 Use Case: Allow the OS to Pause Audio Processing

The low-level abstraction over hardware is great when the hardware can be used exclusively. Other areas call for more cooperation as the following use case shows.

> As an operating system provider I want to pause and resume audio input and output of an application in order to moderate the use of real hardware between multiple applications.

7.1 Discussion

The concept of an audio device shall not specify that input and output will continuously happen, but only that on the same device they happen together at the same rate. This should allow the operating system to suspend calls for buffer updates. The amount or rate by which audio processing actually happens is detached from any clock, so it can vary independently. If audio processing may become arbitrarily slow that's the same as pausing it.

That model could work for something that only uses audio and in that case we have two issues. First, a pure audio player may have controls that should possibly reflect the state of playback. Just stopping to call the I/O callback doesn't provide interaction with the user interface. Second, if the audio is tied to some other playback, for example it is tied to video playback, the video playback wouldn't be paused at the same time. Or the video could continue mute, but when the audio resumes its timestamps have gone out of sync related to the video.

7.2 Open Questions

- What are the use cases we want to support?
- What is the model around OS interaction that fits with these use cases?
- How can the timestamps make sense in relation to other events proceeding?

Listing 5: Example for UB that's hard to fix

```
void runAudio(audio_device dev) {
    MyData data;

    dev.connect([&data] (auto&, audio_device_io<float>& io) {
        // fill 'io' from 'data'
    });

    dev.start();
    doSomethingThatCanThrow();  // PROBLEM
    dev.stop();  // ALSO A PROBLEM
}
```

8 Use Case: Graceful Shutdown

> As a user of C++ I want to have an easy way to ensure my callbacks are not invoking undefined behavior during shutdown of my program.

The code in listing 5 is a straightforward use of the proposed library, but also contains some issues around object lifetime. Library design in C++ is based around RAII, but the proposed audio library doesn't provide any means in this direction to govern the start/stop calls on a device. In the example the call to `stop` can be skipped due to an exception escaping the call to `doSomethingThatCanThrow`. But even if `stop` is called it does not guarantee that the configured callback isn't run anymore. This easily causes undefined behavior due to accessing `data` after its destruction.

8.1 What We Propose

- Add a function to wait for the completion of audio processing.
- Add an RAII guard that ensures a device is stopped fully.
- Remove the return value from `stop`.

There should be a function like `join` to wait for the audio thread to complete processing. That function could then be used in the destructor of an RAII guard to wait for shutdown in cases of a regular return as well as in case of an exception. The guard would call `stop` followed by `join`. This however means that the call to `stop` should better not fail. The return value from `stop` is not actionable and should be removed. Furthermore the joining of the processing should also ensure that the stored callback objects are destroyed in order to release their bound resources.

9 Summary

The sections above show use cases that we don't see supported by P1386 currently. We state where we see the issues, citing from the audio proposal, and give suggestions how we would like the proposal to change. Certain open questions are listed last in every section.

We'd like the use cases to be discussed and the proposed changes to be considered. In general we like moving the audio proposal forward for inclusion into C++.