Simultaneous Recording Program for the DT7816
Progress Report

From Aidan Johnson
To Wu-Jung Lee
Date 19 September 2018

Introduction

The purpose of this progress report is to explain the current state of software development for the simultaneous recording program for the DT7816 henceforth referred to as `recorder`. Besides the project progress, this report will discuss the current challenges and issues faced, and their proposed solutions. There are two versions of the program; one is the pre-alpha version (located in the `master`[1] branch of the repository) and the other is the alpha ('current') version (located in the `double-buffer-dev`[2] branch). The alpha version has two variants: the 'original' ping-pong buffer version and the 'current' asynchronous buffered version. After many man-hours, development on the 'current' version has been halted by an unsolved obstacle: drops of recorded samples. The 'proposed' version seeks to address that.

Background

The recording dropouts are present in both versions of the software. The pre-alpha suffered from dropouts because of the synchronous, rather than asynchronous, producer-consumer circular queue, where the producer is a write pointer and the consumer is a read pointer. This version relied on the external `RingBuf`[3] library for the circular queue or buffer as explained in Figure 1. The early findings demonstrated that the pre-alpha:

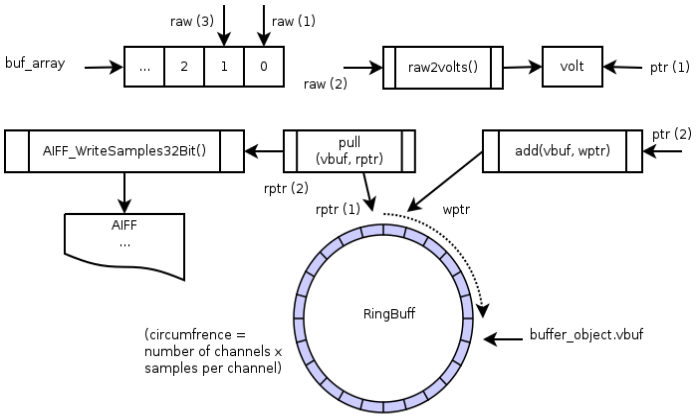


Figure 1: Circular queue from the pre-alpha version workflow diagram.

1. Writes files limited to a sample-length determined by the maximum number of samples in the asynchronous input buffers, resulting in files of miniscule and impractical time-length.

---

[1] https://github.com/aidanjohnson/dt7816/tree/master
[2] https://github.com/aidanjohnson/dt7816/tree/double-buffer-dev
[3] https://github.com/wizard97/ArduinoRingBuffer

2. Writes AIFF recordings using an unreliable, erred, and obscure library called `LibAiff`[4].
3. Ignores the asynchronous input/output (AIO) API provided by the manufacturer, Measurement Computing Corporation (MCC) née Data Translation (DT).

Buffering Control Regime

Motivating this report, it was decided that instead of using a circular buffer approach, a double buffer for this producer-consumer type problem should be used, thus prompting the creation of the alpha version. Also known as a ping-pong buffer, this scheme allows input samples to be stored in one buffer while the other is written to file, alternating between the 'ping' buffer and 'pong' buffer. Moreover, when implemented asynchronously[5], the read-write process can occur simultaneously. The first finding above was addressed by implementing a feature in the 'original' version for the user to set the file time-length in seconds. Expanding on the existing recording regime, the option to write the recording to a CSV file was added to the 'current' version because MCC's example code prefers to write data this way. This option serves as a performance and quality benchmark for writing to AIFF files. The second finding fortunately was not devastating. `LibAiff` was already too cumbersome of an API and had insufficient documentation anyway. As a replacement, the program now uses `libsndfile`[6], which is mature in documentation, widely used, and most importantly, records (in theory) about as well as the CSV file stream.

Nevertheless, the 'original' regime has one devastating flaw; during the transition from the write-to-ping whilst write-from-pong, and vice-versa, (that is, a requeue) samples are dropped. A gap equal to about 60 samples occurs, which equals 300 μs, if sampling at 200 kHz (period 5 μs). One explanation for this was that the regime does not properly use the API in order take full advantage of the DT7816 board's AIO capabilities. Once interpreted, the vaguely and sparsely documented API was discovered to readily implement the desired AIO, addressing the third issue in the 'current' version. But this did not resolve the dropout in general. Instead the dropout occurs each buffer transition with a protracted dropout every requeue.

As will been demonstrated in the next two sections, the 'original' and 'current' alpha version are unsatisfactory and must be improved upon for field deployment. Lastly, one final note: refer to the header file `recorder_helpers.h` and its corresponding '.c' `recorder_helpers.c` file for the custom wrapper function names used below in the diagrams of Figures 2, 3, and 4.

Explicit Double (Ping-Pong) Buffer

The 'original' explicit implementation of a double buffer, or ping-pong buffer, is described graphically below in Figure 2. The topmost section of the flow chart in Figure 2, up to where an AIFF file is created, only relates to the setup and configuration of the AIO stream, channels, triggers, and buffers (the setup). It also concerns the feature for cycling on and off according to

---

[4] http://aifftools.sourceforge.net/libaiff/
[5] On AIO: https://docs.microsoft.com/en-gb/windows/desktop/FileIO/synchronous-and-asynchronous-i-o
[6] https://en.wikipedia.org/wiki/Libsndfile

sunset and sunrise times, and desired duration of recording. As such, these features will not be discussed in detail as they are not relevant to addressing the issues listed earlier.

The remainder of the flow chart is the ping-pong buffer mechanism. Once the recording cycle is enabled, the program follows the algorithm outlined below. To aid in mapping them to Figure 2 the specific function is included in brackets. The steps are:

1. Create a file descriptor for the AIFF to be written to (`createAIFF()`, which calls `libsndfile`'s public function).

1.1. If just entering the recording while loop, that is, it just became 'night', then:

1.1.1. Remember it is night (`night = 1`).

1.1.2. Arm and start the input stream (`armStartStream()`).

1.1.3. Remember it is the first buffer to be written to file.

1.2. Otherwise, go to **1.1.3** and continue to **2**.

2. Wait indefinitely (`aio_wait(-1)`) until one of the ping buffer has been completely filled by the input stream.

2.1. If the buffer is not complete (`aio_wait() < 1`), go back to **2**.

2.2. Otherwise, proceed to **3**.

3. Increment the number of file buffers completed.

3.1. If the number of buffers completed is equal to the number of buffers to be written per file, then go to **4**.

3.2. Otherwise,

3.2.1. If the number of buffers is even, then write the pong buffer (`write(PONG)`) whilst indefinitely waiting for the ping buffer to fill completely (`aio_wait(-1)`). When full, return to **2**.

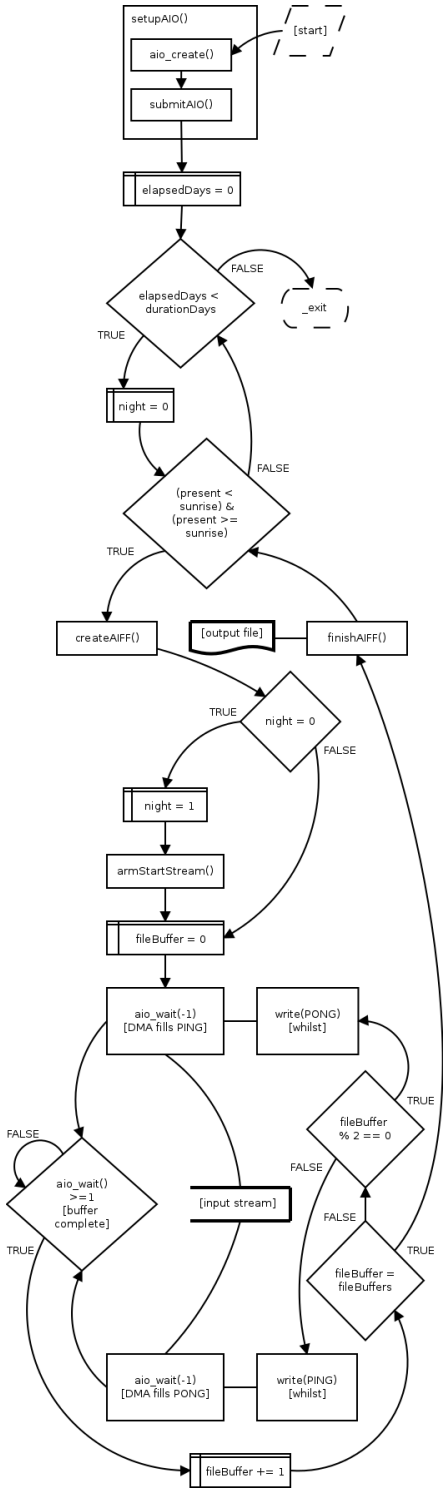


Figure 2: Process flow for the ping-pong buffered implementation. Rhombi correspond to conditional if-else logic and rectangles correspond to helper and API function calls. Text in brackets are non-code annotations included for readability. An arrow indicates direction of process flow and a line indicates a joint process that occurs at the same time.

3.2.2. Otherwise (the number of buffers is odd), write the ping buffer (`write(PING)`) whilst continuing and returning to **2**.

4. Clean up after writing to file by closing the file descriptor (`finishAIFF()`), and then return to **1** if still night. Otherwise wait until night and proceed with **1**.

The control (ping and pong) buffers of the AIO are requeued for filling after being read and written from. The stream is set to requeue buffers (`aio_create(inStream, 0, isInAIODone, isInStreamEmpty)`, where the call back `isInAIODone`, during creation. This function is called when a buffer is complete, and always returns `1` after writing to file, which enables requeuing for input stream `inStream`). This step is essential for the ping-pong buffering to work as designed in both this explicit and the implicit version.

Implicit Asynchronous Double Buffer

Before launching into describing this improved process flow, first a brief comment. This implicit implementation has the added flexibility of increasing the number of 'queued' buffers to quantity larger than 2 (at most 128). It is advisable to increase the number of queued AIO buffers (at least two) as you use higher sampling frequencies. With this in mind, the program automatically sizes the buffers as to not exceed the 16-bit sampling limit (65536 samples), which was discovered earlier in the development process.

The 'current' implicit implementation of a multi-buffer regime is described graphically below in Figure 3 and 4. Like in the previous implementation, the leftmost third of the diagram predominantly sets up and configures the input stream. The algorithmic steps (baring the setup and time dependent conditionals) are as follows:

1. If just entering night (`night = 0`) proceed to **2**, else, skip to **3**.

2. Arm and start the input stream (`armStartStream()`). Also remember it is night now (`night = 1`).

3. Asynchronously sample input stream, blocking for 0 milliseconds (`aio_wait(0)`) and return `1`, whilst simultaneously the control buffers submitted (`submitAIO()`) allocated in setup (`setupAIO()`) are filled. That is:

3.1. The first queued buffer is filled with input stream data.

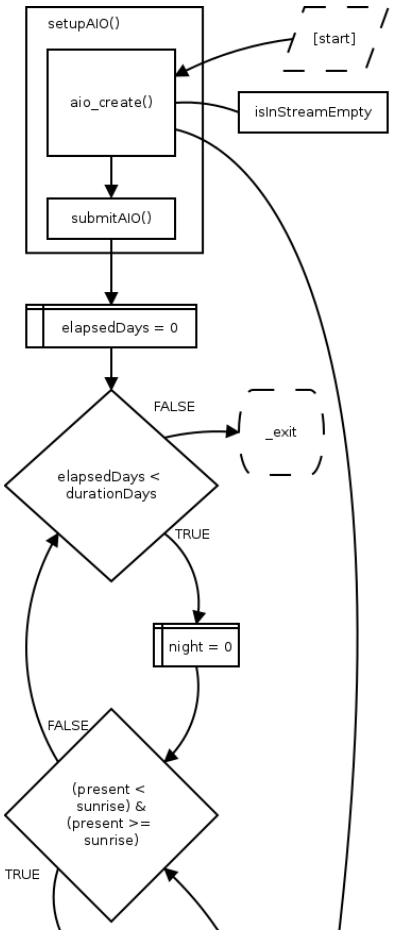


Figure 3: Process flow for the AIO multi-buffered implementation. The encompassing rectangles indicate a broader in scope helper function that calls other functions within its scope. All other shapes have the same key

3.2. Once done, or entirely full, queue the next buffer to be completed, returning to **3.1** and proceeding, and at the same time continuing to the next step.

3.3. As you continue to write the samples in the previous buffer to file, with this new, current buffer go to **1**.

4. If this is the first buffer to be written to file, then:

4.1 Create an AIFF file (`createAIFF()`) and continue to the next step.

4.2 If otherwise, skip to next step.

5. With the data in this buffer, write it to the AIFF file created in **4.1**.

6. Increment by one and remember the number of buffers written for this file.

6.1. If the number of buffers written is equal to the number to be written, then skip step **7** and proceed.

6.2. Clean up after the AIFF file by closing its file descriptor (`finishAIFF()`).

6.3. Set and remember the number of buffers written to the now closed file as zero.

7. Requeue the now completed buffer so it can be used again to store input samples.

8. Repeat by going to **3** and continue yet again.

This process allows data to be buffered and written simultaneously as data is written to file *ad infinitum*, or until night ends, etc. The key difference between this implementation with the 'original' one is that the API that interfaces with the input control buffers is used to not only requeue the buffers but also write to file using the `isAIODone()` function, which is called back at control buffer completions. Refer to moderately helpful comments in `dt78xx.h` and `dt78xx.c` for explanation on the AIO API and `example-applications/dt78xx-examples/aio-out` for an example program.

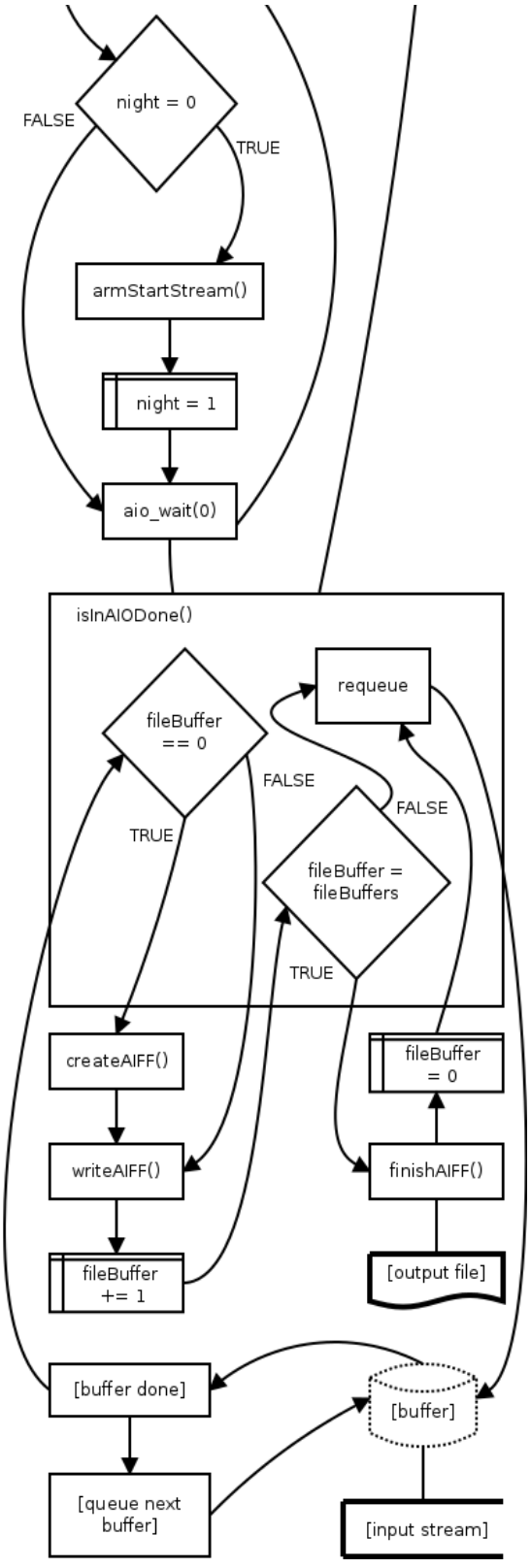


Figure 4: Figure 3 continued from its bottom.

As stated for the 'original' version, when creating the input stream in the `recorder_helpers.c`, a pointer to the callback function `isInAIODone()` is passed. This callback, which writes the completed buffer to file once complete and while the analog input fills the next buffer, is called by the API during the input stream recording (see `dt78xx_aio.h`). The manufacturer in this header file gives a vague visual that corroborates—to my understanding—this process flow. It will aid the reader to study the `aio-out` example program, albeit for analog output, for a similar implementation. Notice that `aio_wait()` is called continuously and `aio_create()` was passed the callback function. The difficulty in understanding the API stems from poor manufacturer documentation, which makes little to no mention of the asynchronous process. It is not just me; the ocean engineer in Rhode Island I've been in contact with has the same complaint. Not to mention he has found tech support unhelpful and vague at best.

Results

For both CSV and AIFF file recording demonstrations of *only* the 'current' implicit buffer implementation, plots of the recorded data in the temporal and spectral domain are provided. Each recording is 60 seconds long. All of the recordings were made using the implicit double buffer AIO. Under either file formats are the sampling frequency and two demonstrative analog signals. Each signal, which is defined by its frequency, amplitude, and waveform, has a file corresponding to its recording is provided in addition to the time- and frequency-domain plots. (The best way for quick visual and auditory inspection is done by opening AIFF files Audacity.)

Before showing the plots, a few concerning observations. Both the CSV and AIFF recordings suffer from classic beat waveform interference[7] patterns when sampling at close to the Nyquist frequency. These patterns also occur under the same conditions using the `aio-in` example program. However, the frequency content of the beat waveform is a pure sinusoid with the correct frequency so I am unsure what is the going on. Interestingly, if the sine waveform frequency is made a quarter that of the sampling frequency instead of less than but about half, the beat pattern disappears and the recording is a pure sign wave as should be expected.

Finally, `libsndfile`'s AIFF writer results in recordings that are similar. Although these are inferior to the CSV written ones, due to the considerable slower write speed that introduces dropouts-offs into the recording after moving to the next queued buffer. (These dropouts are best seen in Audacity rather than the plots. To see the whole data file in real-time run plot.py for the desired data set. This program works for both AIFF and CSV, and conveniently can be scrolled in samples. You can change the range of samples for the following plots in the Jupyter notebook `plot_csv_aiff.ipynb` in the `.zip` folder. The plots below do not show all or near all of the samples as there are just too many samples in a 1-minute data set to show on one plot.) It is disappointing that the AIFF recording results poor with their periodic dropouts, which are absent in the CSV recordings.

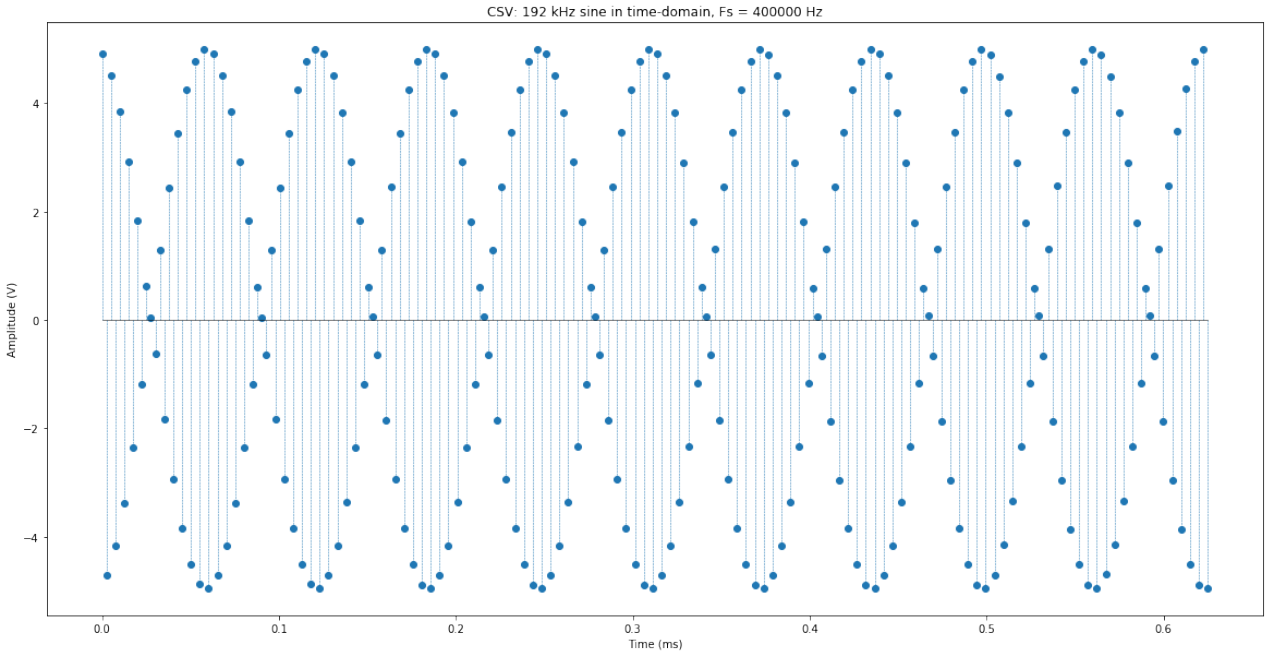
[7] http://hyperphysics.phy-astr.gsu.edu/hbase/Sound/beat.html

CSV

Signal: 192 kHz 5 V sine sampled at 400 kHz

- Recording file: `csv/400/sine/192kHz400_20131219T021907434105Z.csv`

(a)



(b)

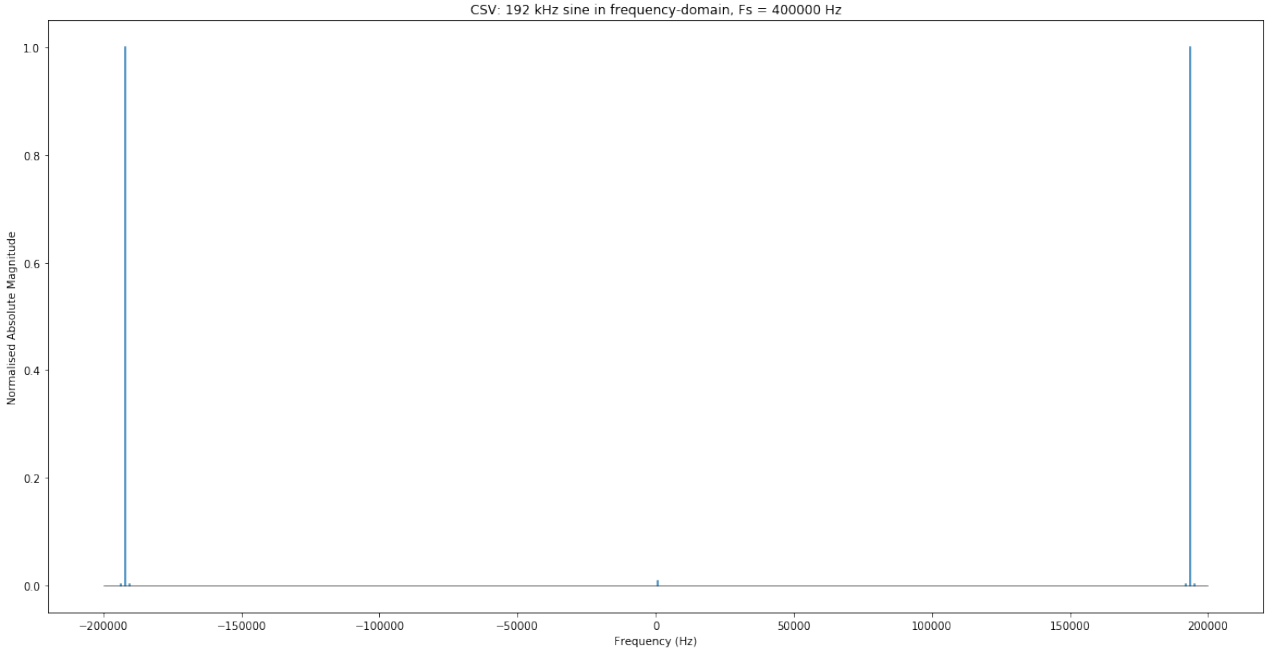
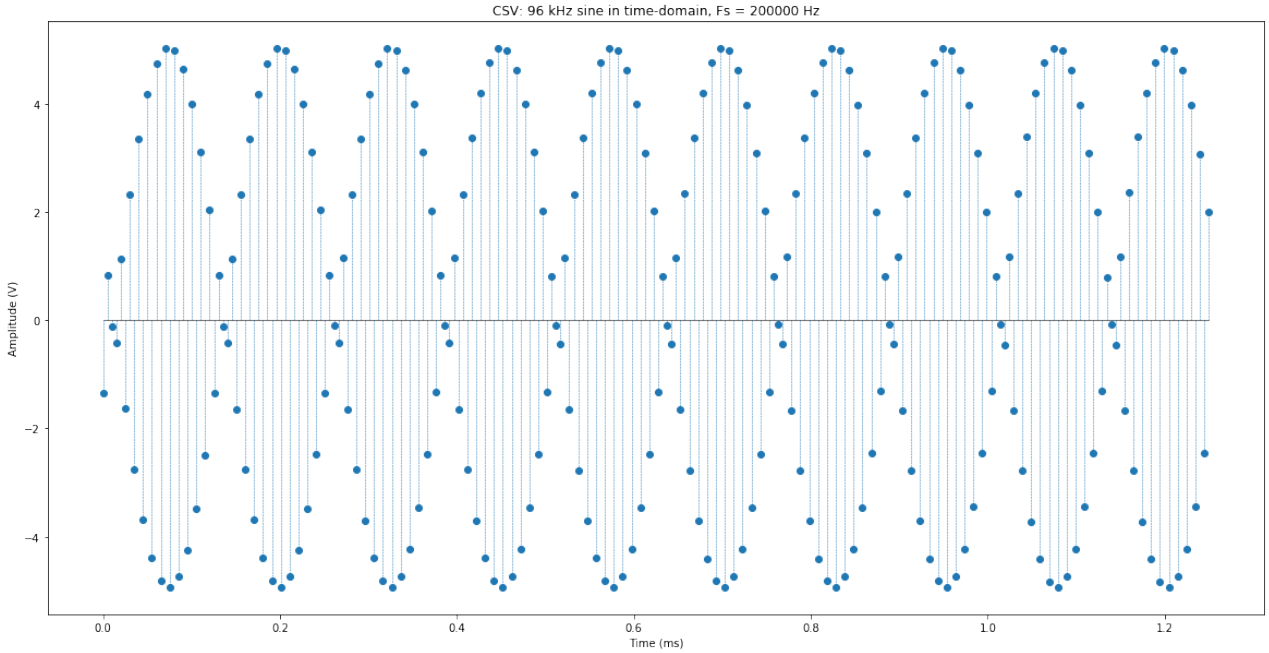


Figure 5: (a) A portion of temporal domain data of 1 min of sampling. (b) Spectral domain for the signal in (a).

Signal: 96 kHz 5 V sine sampled at 200 kHz

- Recording file: `csv/200/sine/96kHz200_20180909T033007480050Z.csv`

(a)



(b)

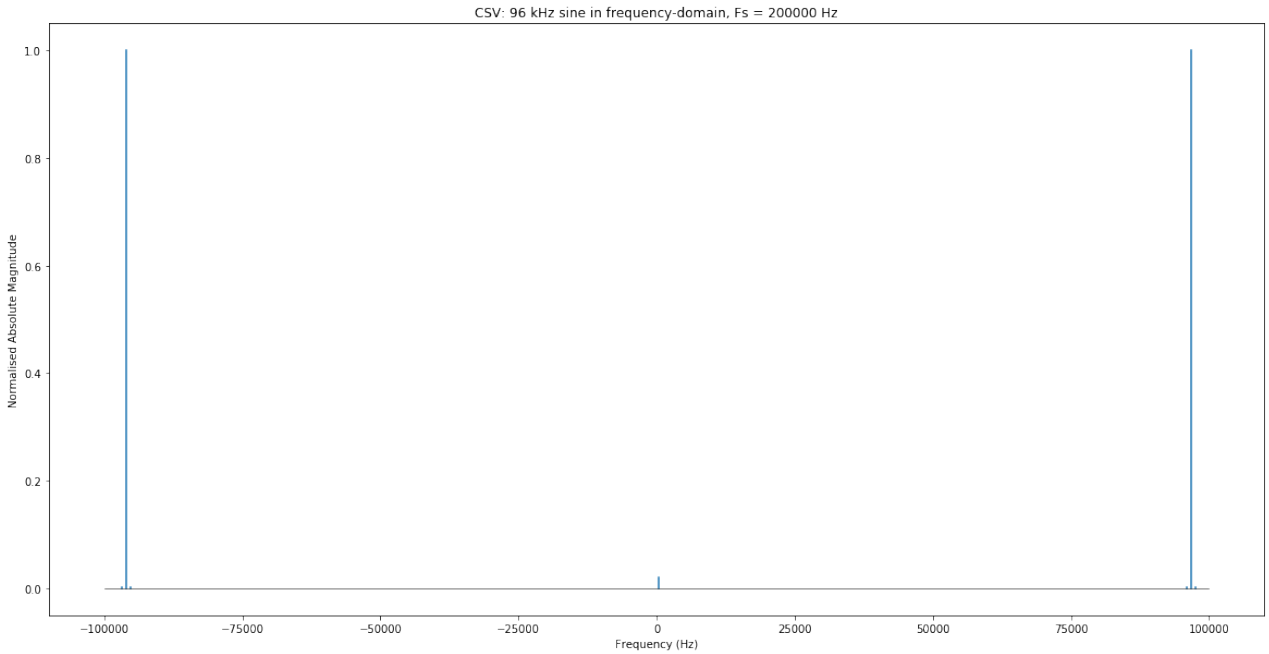
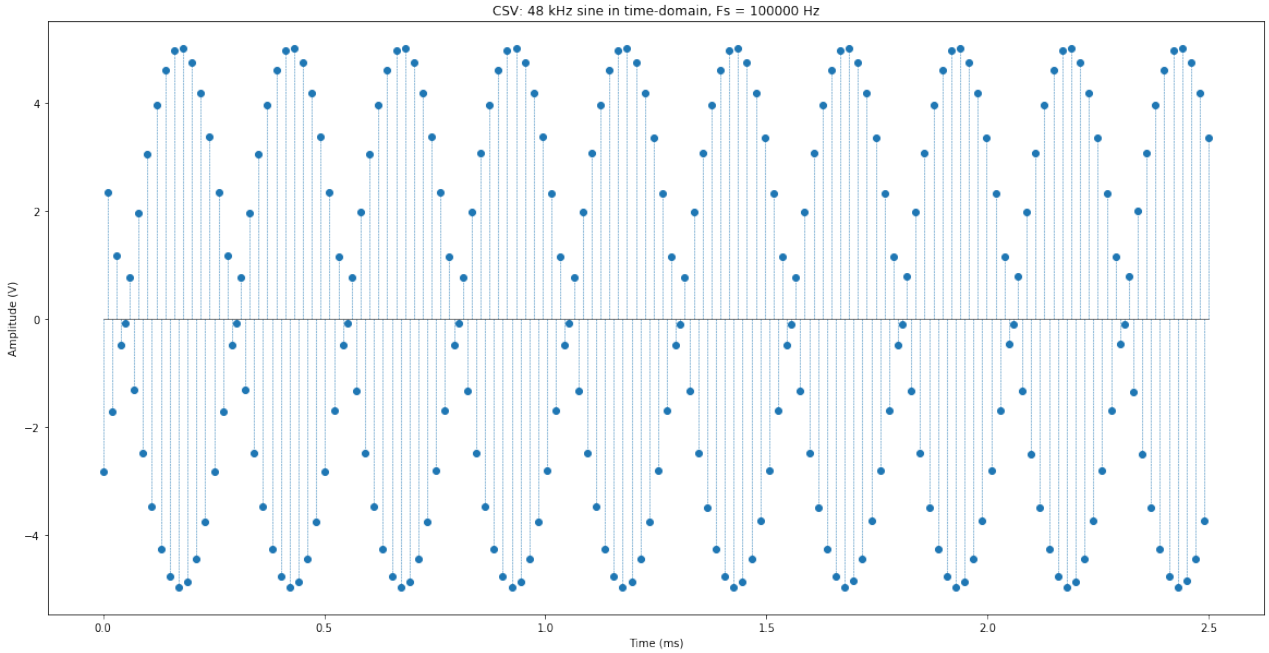


Figure 6: (a) A portion of temporal domain data of 1 min of sampling. (b) Spectral domain for the signal in (a).

Signal: 48 kHz 5 V sine sampled at 100 kHz

- Recording file: `csv/100/sine/48kHz100_20180909T024527672838Z.csv`

(a)



(b)

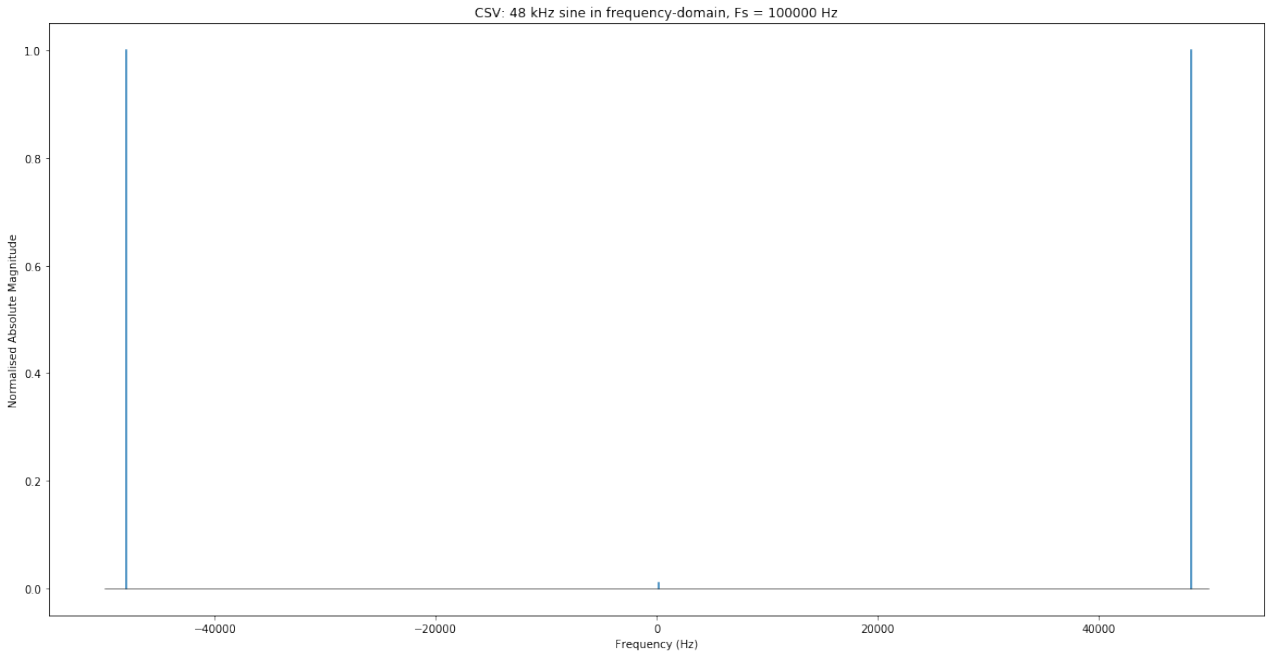


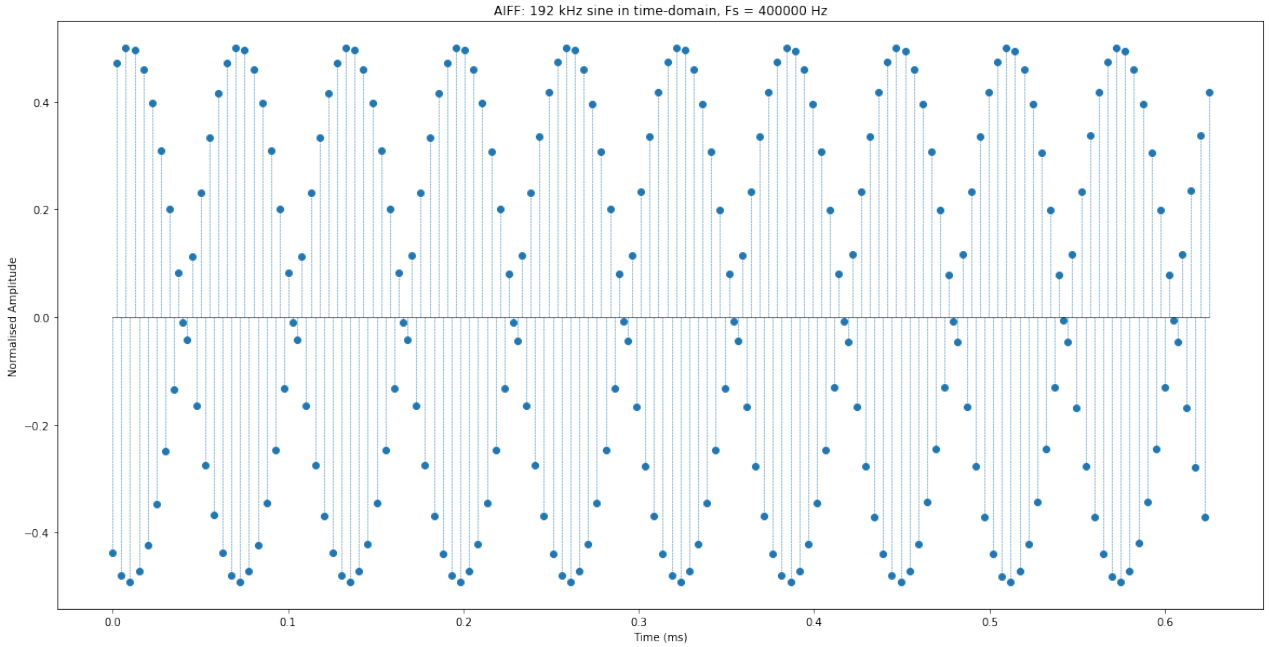
Figure 7: (a) A portion of temporal domain data of 1 min of sampling. (b) Spectral domain for the signal in (a).

AIFF

Signal: 192 kHz 5 V sine sampled at 400 kHz

- Recording file: `aiff/400/sine/192kHz400_20180911T063429741633Z.aiff`

(a)



(b)

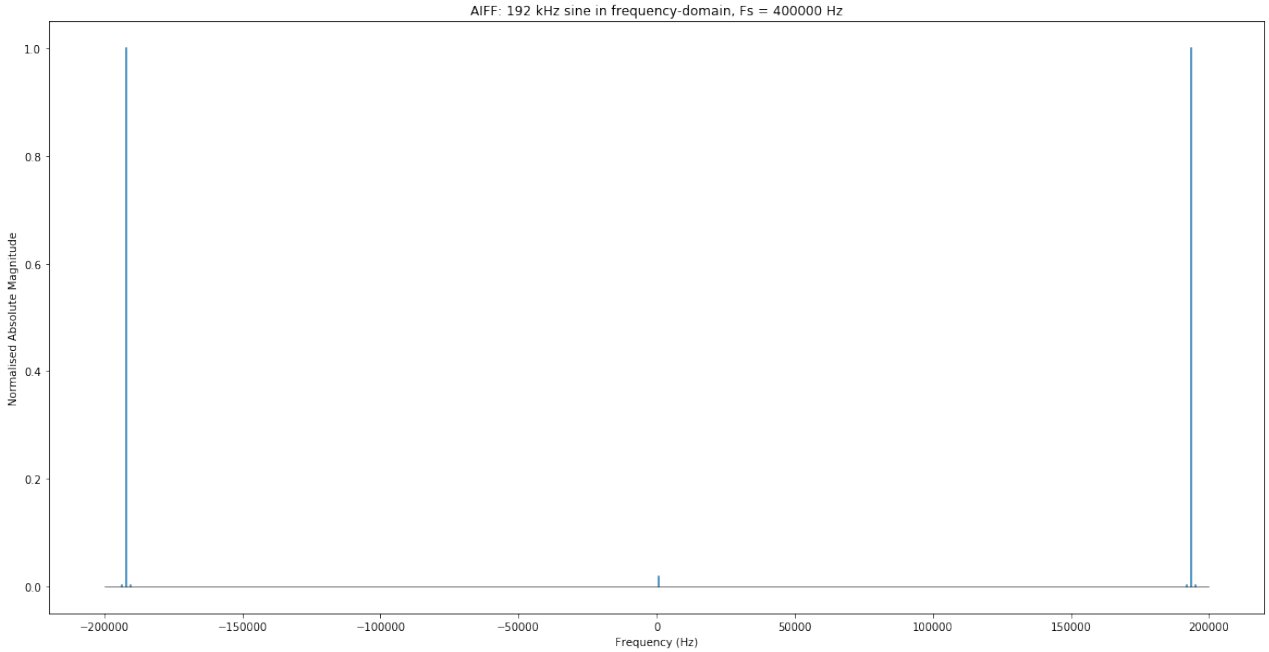
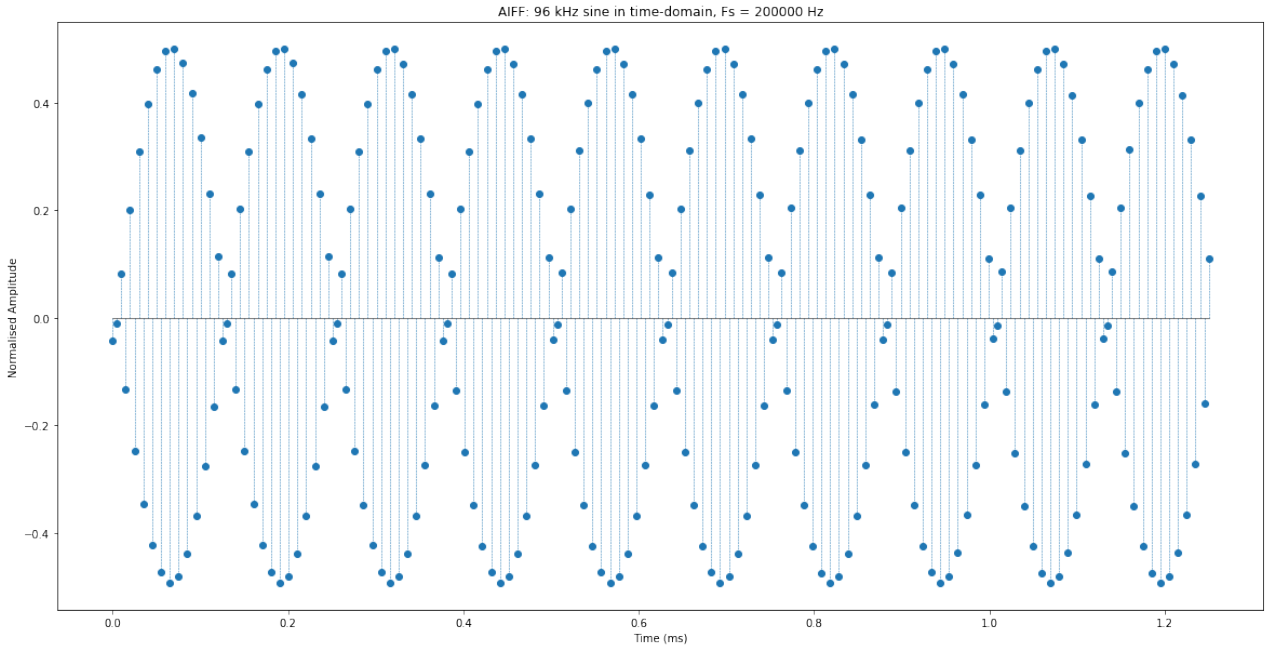


Figure 8: (a) A portion of temporal domain data of 1 min of sampling. (b) Spectral domain for the signal in (a).

Signal: 96 kHz 5 V sine sampled at 200 kHz

- Recording file: `aiff/200/sine/96kHz200_20180911T062911695433Z.aiff`

(a)



(b)

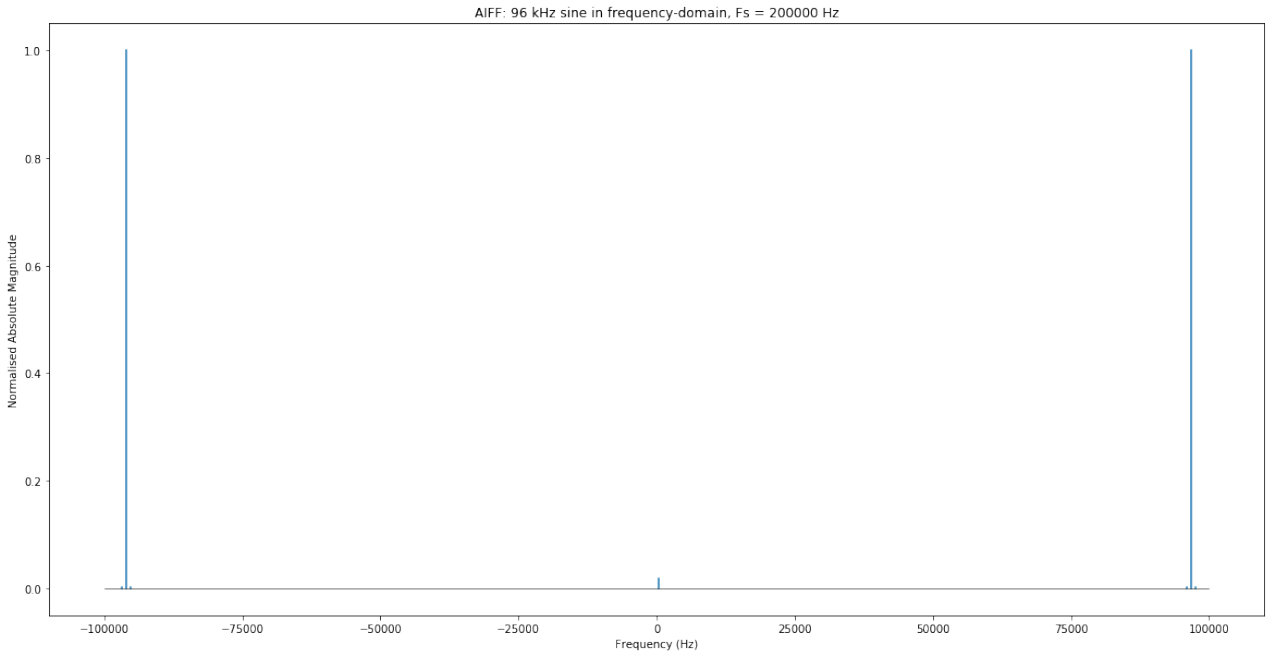
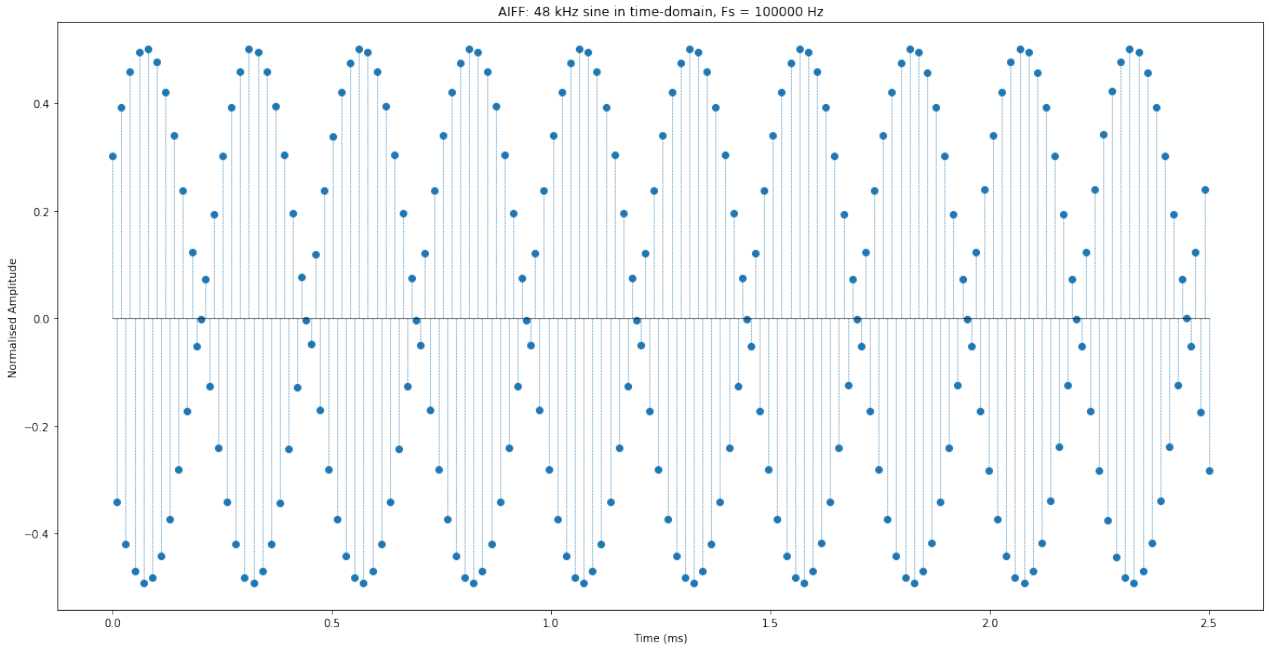


Figure 9: (a) A portion of temporal domain data of 1 min of sampling. (b) Spectral domain for the signal in (a).

Signal: 48 kHz 5 V sine sampled at 100 kHz

- Recording file: `aiff/100/sine/48kHz100_20180911T062226700725Z.aiff`

(a)



(b)

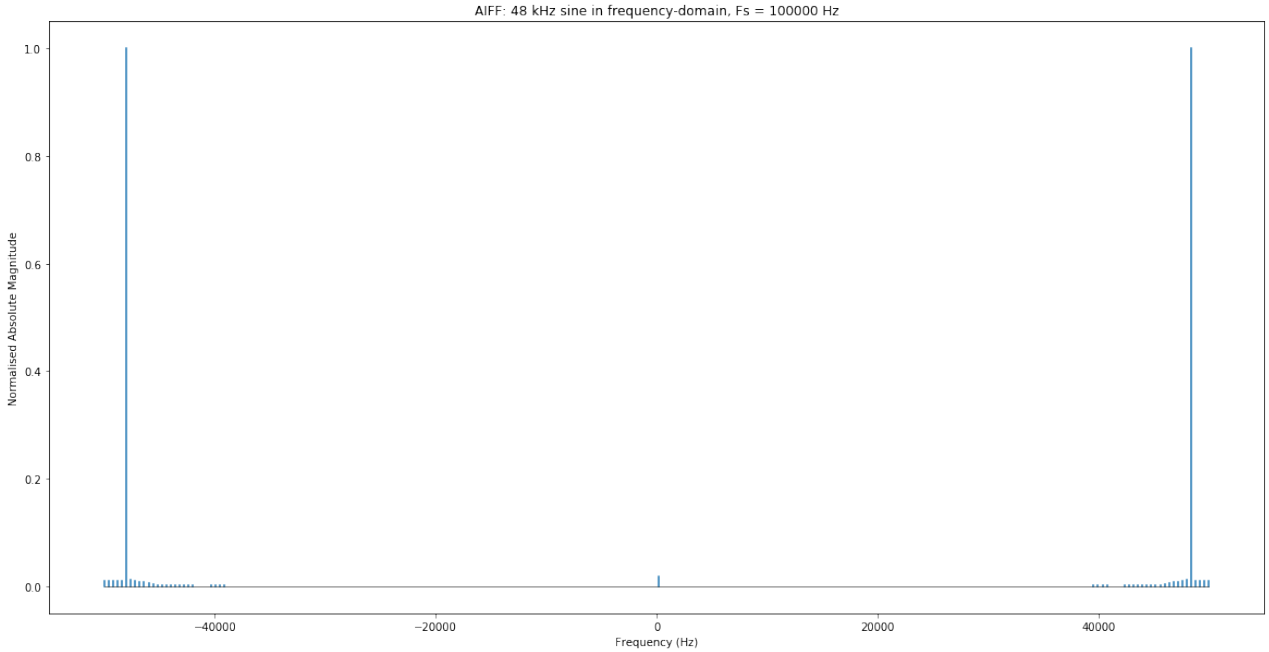
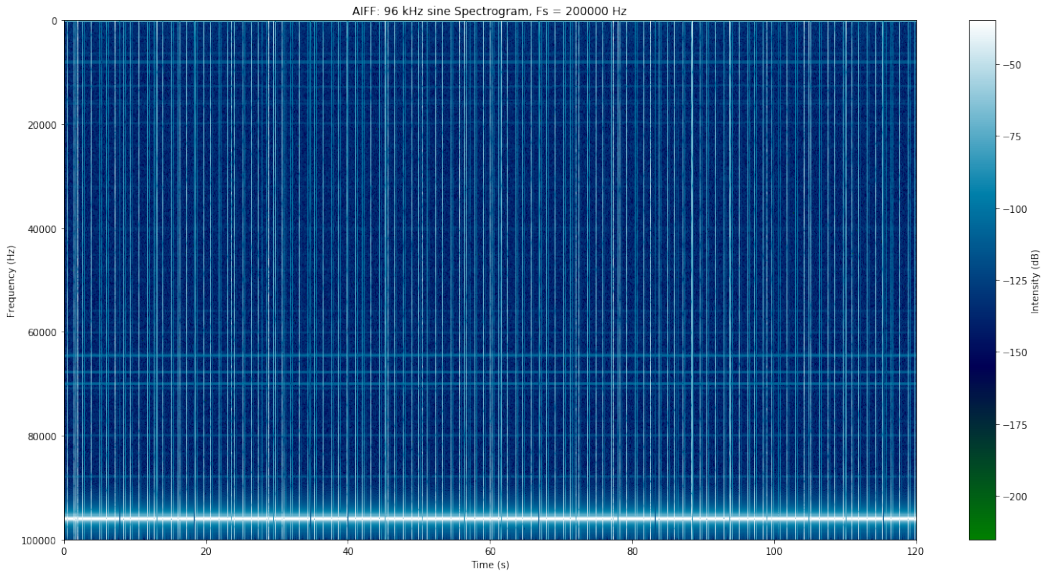


Figure 10: (a) A portion of temporal domain data of 1 min of sampling. (b) Spectral domain for the signal in (a).

Spectrograms

The best visual quality comparison metric is the spectrogram of the 1-minute files. Based on the number of dropouts, which show up as vertical all-frequency bands in the spectrogram, the CSV writer performs better than the AIFF writer in the ‘current’ version. See Figures 11 and 12 below.

(a)



(b)

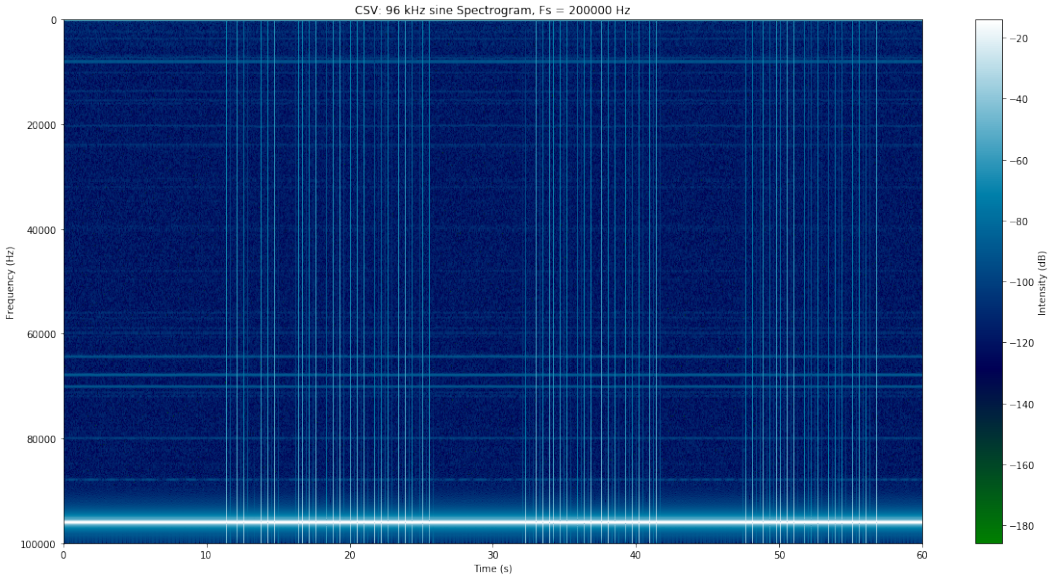
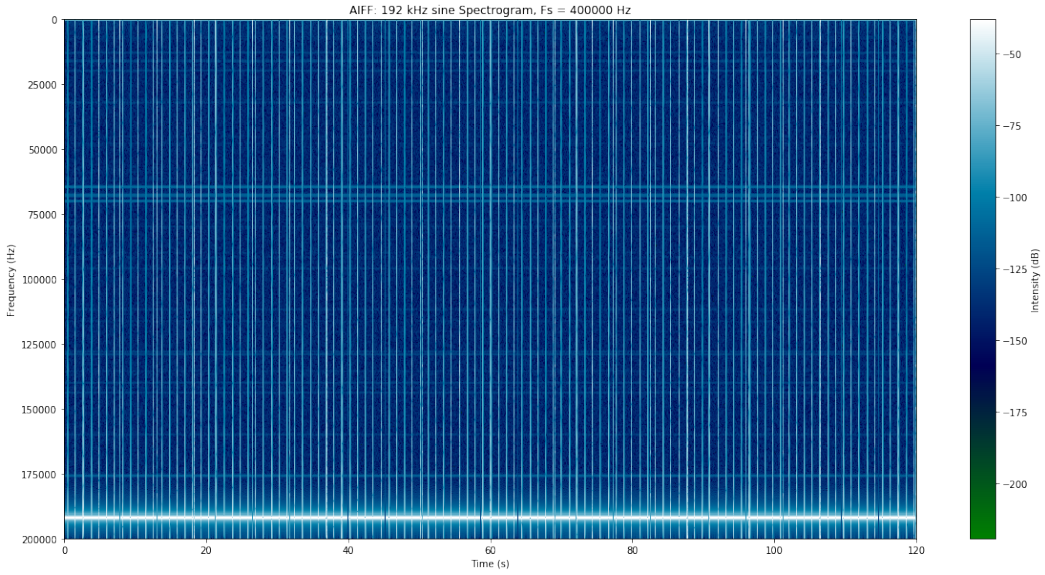


Figure 11: Spectrogram for (a) the AIFF writer and (b) the CSV writer.

(a)



(b)

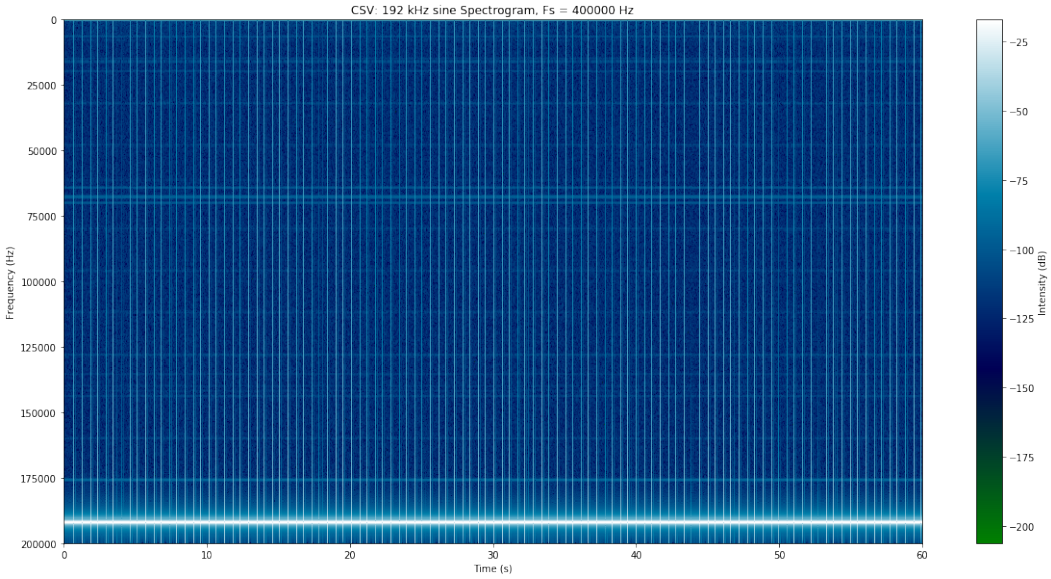


Figure 12: Spectrogram for (a) the AIFF writer and (b) the CSV writer.

Next Steps

It can be definitively said the attempted solution—the ‘current’ version—to the issues of the pre-alpha version and the alpha’s ‘original’ version is unacceptable. The question remains of how to best implement the buffered input-to-file regime. (My contact at the University of Rhode Island has been stumped by this same problem, and again paltry help from MCC.) Fortunately, after the draft of this report, an answer seems to have been found.

Proposed Buffering Regime

The apparent solution is a blend of both the ‘original’ and ‘current’ versions. The ‘original’ version rightly:

- Requeues all buffers once they had been filled and then written to file.
- Calls `aio_wait()`, blocking computation for an indefinite period after a filled buffer was written to file, and then only proceeding when that buffer was complete.

Similarly, the ‘current’ version rightly has flexibility in the number of buffers queued instead of only the two required by ping-pong buffering. Instead of specifically accessing one of the two buffers in the ‘original’, each of the at least two and at most 128 buffers are accessed, read, and written to file. In sum, this regime comprises the ‘proposed’ version.

With these successes, the buffering can be implemented as follows and visually described in Figure 13 (right). As before, the portion of the flowchart above the `night = 0` can be ignored. This portion of the ‘proposed’ version is the same as the setup preliminaries in the ‘current’ version except that `isAIODone()` always returns `1` and does not write files.

1. If just entering night (`night = 0`) proceed to **2**, else, skip to **3**.

2. Arm and start the input stream (`armStartStream()`). Also remember it is night now (`night = 1`).

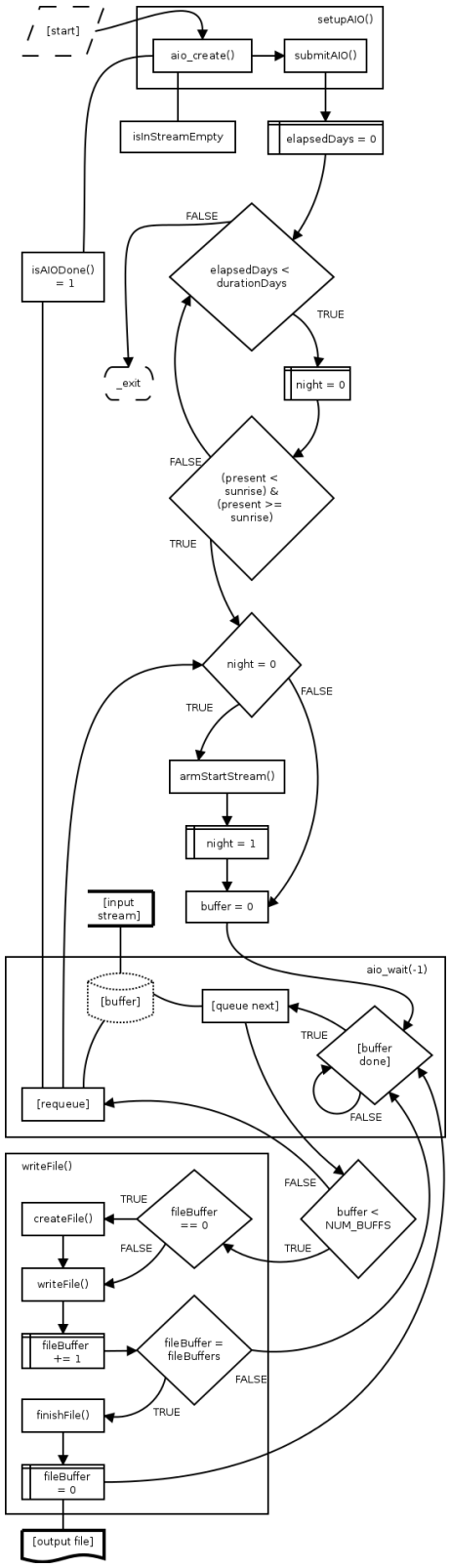


Figure 13: Same key as before (Figure 3).

3. Asynchronously wait as long as is required to fill at least one of the `NUM_BUFFS` buffers.

3.1. If the buffer is done proceed to next step.

3.2. Otherwise wait (indefinitely) until the buffer completely full.

4. Queue the next buffer for the input stream.

5. If the number of buffers filled (`buffer`) is less than the total number (`NUM_BUFFS`):

5.1. Then proceed to the next step.

5.2. Otherwise requeue all the buffers as according to `isAIODone()` and return to step **1**.

6. If the number of buffers written to file (`fileBuffer`) is zero, then:

6.1. Create a new AIFF file (`createFile()`).

6.2. Otherwise, proceed to step **7**.

7. Write the completed buffer to file (`writeFile()`), and then increment the number of buffers written to file by one.

8. If the number of buffers written to file is equal to the total number of buffers calculated for the set time-duration of the file (`fileBuffers`), then:

8.1. Finish writing the AIFF file by cleaning up and closing the file (`finishFile()`). The recording file has been created. Proceed to the next step.

8.2. Otherwise, go back to step **3**.

9. Reset the number of buffers written to file to zero. Then go back to step **3**.

Regime Performance

Full disclosure, the proposed regime has yet to be rigorously evaluated for performance. Nonetheless, when sampling a 5 V 50 kHz sine wave at 200 kHz, the recorded AIFF waveform lacks any dropouts. Since the entire recording cannot be displayed one plot, open the file:

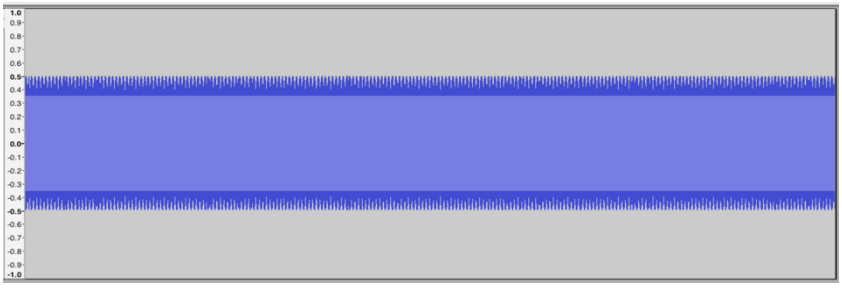


Figure 14: Screenshot of `tests/50kHz200_20180919T08163934047Z.aiff` in Audacity.

```
tests/50kHz200_20180919T08163934047Z.aiff
```

in Audacity for visual and auditory inspection. A screenshot of the entire recording is Audacity is included below in Figure 14. Note how it does not have dropouts unlike the previous two alpha versions. Dropouts would have appeared as chunks of data at zero and periodic spikes above

the 0.5 level (that is, 5 V). (Open one of the data set files from either the ‘original’ or ‘current’ version to confirm this for yourself.) Lastly, for added detail, Figure 15 (below) provides the spectrogram of this same file.

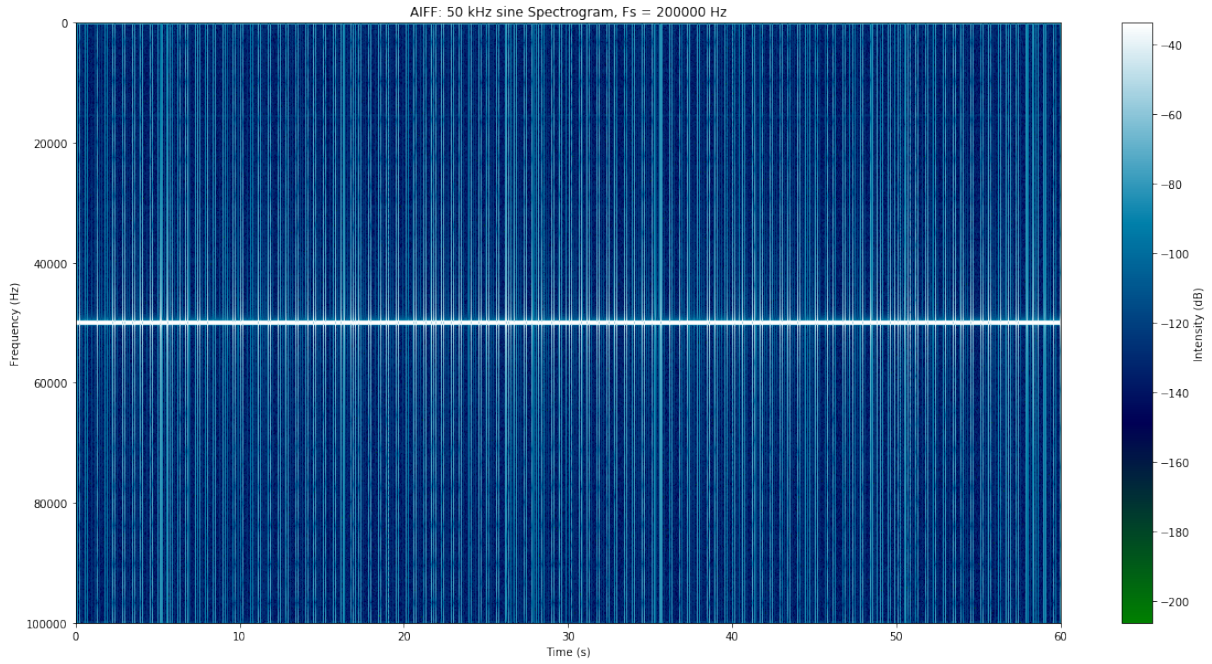


Figure 15: Spectrogram of the file plotted in Figure 12.

Conclusion

As articulated since its introduction and via GitHub repository messages, this report discusses the current state of development of the autonomous and continuous buffered audio recording program. The current technical issues have been explained and a solution for the recording regime has been proposed. The progress has transitioned from a pre-alpha that recorded input signals after saving the data to a buffer, interrupting continuous recording, to an alpha version. This version has undergone three attempts at different recording regimes. Efforts have culminated with the third ‘proposed’ version that produces not sample dropouts, thereby achieving the necessary satisfactory performance. Further testing remains to be performed on this ‘proposed’ version to support developer confidence in the program. At that point, the ‘proposed’ version can be elevated as a beta version.