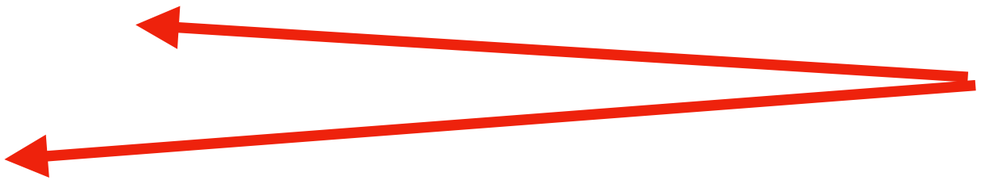
Double Buffering

Let non-realtime thread know that new data is available

Only swap indices if new data is available

Atomically clear new data bit and increment index



Add a new bit "BIT_NEWDATA" to the index variable


However, introduced race because we now unatomically load store the idx variable

```cpp
using FrequencySpectrum = std::array<float, 512>;

enum { BIT_IDX = (1 << 0), BIT_NEWDATA = (1 << 1)};

std::array<FrequencySpectrum,2> mostRecentSpectrum;
std::atomic<int> idx = {0};

void processAudio (const float* buffer, size_t n)
{
    auto freqSpec = calculateSpectrum (buffer, n);

    auto i = idx.load() & BIT_IDX;
    mostRecentSpectrum[i] = freqSpec;
    idx.store ((i & BIT_IDX) | BIT_NEWDATA);
}

void updateSpectrumUIButtonClicked()
{
    auto current = idx.load();

    if ((current & BIT_NEWDATA) != 0)
    {
        current = (current & BIT_IDX) ^ 1;
        idx.store (current);
    }

    displaySpectrum (mostRecentSpectrum[(current & BIT_IDX) ^ 1]);
}
```

78

Double Buffering

```
using FrequencySpectrum = std::array<float, 512>;

enum { BIT_IDX = (1 << 0), BIT_NEWDATA = (1 << 1)};

std::array<FrequencySpectrum,2> mostRecentSpectrum;
std::atomic<int> idx = {0};

void processAudio (const float* buffer, size_t n)
{
    auto freqSpec = calculateSpectrum (buffer, n);

    auto i = idx.load() & BIT_IDX;
    mostRecentSpectrum[i] = freqSpec;
    idx.store ((i & BIT_IDX) | BIT_NEWDATA);
}

void updateSpectrumUIButtonClicked()
{
    auto current = idx.load();

    if ((current & BIT_NEWDATA) != 0)
    {
        current = (current & BIT_IDX) ^ 1;
        idx.store (current);
    }

    displaySpectrum (mostRecentSpectrum[(current & BIT_IDX) ^ 1]);
}
```

Add a new bit "BIT_NEWDATA" to the index variable

However, introduced race because we now unatomically load store the idx variable

Let non-realtime thread know that new data is available

Only swap indices if new data is available

Atomically clear new data bit and increment index

Double Buffering

```
using FrequencySpectrum = std::array<float, 512>;

enum { BIT_IDX = (1 << 0), BIT_NEWDATA = (1 << 1), BIT_BUSY = (1 << 2)};

std::array<FrequencySpectrum,2> mostRecentSpectrum;
std::atomic<int> idx = {0};

void processAudio (const float* buffer, size_t n) {
    auto freqSpec = calculateSpectrum (buffer, n);

    auto i = idx.fetch_or(BIT_BUSY) & BIT_IDX;
    mostRecentSpectrum[i] = freqSpec;
    idx.store ((i & BIT_IDX) | BIT_NEWDATA);
}

void updateSpectrumUIButtonClicked() {
    auto current = idx.load();

    if ((current & BIT_NEWDATA) != 0) {
        int newValue;
        do {
            current &= ~BIT_BUSY;
            newValue = (current ^ BIT_IDX) & BIT_IDX;
        } while (! idx.compare_exchange_weak (current, newValue));

        current = newValue;
    }

    displaySpectrum(mostRecentSpectrum[(current & BIT_IDX) ^ 1]);
}
```