Dynamic Tempo

Sequence Creation

```
inline Sequence::Sequence (std::vector<TempoChange> tempos, std::vector<TimeSigChange> timeSigs, std::vector<KeyChange> keys,
                           LengthOfOneBeat lengthOfOneBeat)
{
    if (keys.empty())
        keys.push_back ({});

    assert (tempos.size() > 0 && timeSigs.size() > 0);
    assert (tempos[0].startBeat == BeatPosition());
    assert (timeSigs[0].startBeat == BeatPosition());
    assert (keys[0].startBeat == BeatPosition());

    // Find the beats with changes
    std::vector<BeatPosition> beatsWithChanges;
    beatsWithChanges.reserve (tempos.size() + timeSigs.size() + keys.size());

    for (const auto& tempo : tempos)
    {
        beatsWithChanges.push_back (tempo.startBeat);

        hash_combine (hashCode, tempo.startBeat.inBeats());
        hash_combine (hashCode, tempo.bpm);
        hash_combine (hashCode, tempo.curve);
    }

    for (const auto& timeSig : timeSigs)
    {
        beatsWithChanges.push_back (timeSig.startBeat);

        hash_combine (hashCode, timeSig.startBeat.inBeats());
        hash_combine (hashCode, timeSig.numerator);
        hash_combine (hashCode, timeSig.denominator);
        hash_combine (hashCode, timeSig.triplets);
    }

    for (const auto& keyChange : keys)
    {
        beatsWithChanges.push_back (keyChange.startBeat);

        hash_combine (hashCode, keyChange.startBeat.inBeats());
        hash_combine (hashCode, keyChange.key.pitch);
        hash_combine (hashCode, keyChange.key.scale);
    }

    std::sort (beatsWithChanges.begin(), beatsWithChanges.end());
    beatsWithChanges.erase (std::unique (beatsWithChanges.begin(), beatsWithChanges.end()),
                            beatsWithChanges.end());

    // Build the sections
    TimePosition time;
    BeatPosition beatNum;
    double ppq          = 0.0;
    size_t timeSigIdx   = 0;
    size_t tempoIdx     = 0;
    size_t keyIdx       = 0;

    auto currTempo   = tempos[tempoIdx++];
    auto currTimeSig = timeSigs[timeSigIdx++];
    auto currKey     = keys[keyIdx++];

    const bool useDenominator = lengthOfOneBeat == LengthOfOneBeat::dependsOnTimeSignature;

    for (size_t i = 0; i < beatsWithChanges.size(); ++i)
    {
        const auto currentBeat = beatsWithChanges[i];
        assert (std::abs ((currentBeat - beatNum).inBeats()) < 0.001);

        while (tempoIdx < tempos.size() && tempos[tempoIdx].startBeat == currentBeat)
            currTempo = tempos[tempoIdx++];

        if (timeSigIdx < timeSigs.size() && timeSigs[timeSigIdx].startBeat == currentBeat)
            currTimeSig = timeSigs[timeSigIdx++];

        if (keyIdx < keys.size() && keys[keyIdx].startBeat == currentBeat)
            currKey = keys[keyIdx++];

        const bool nextTempoValid = tempoIdx < tempos.size();
        double bpm = nextTempoValid ? calcCurveBpm (currTempo.startBeat.inBeats(), currTempo, tempos[tempoIdx])
                                    : currTempo.bpm;

        int numSubdivisions = 1;

        if (nextTempoValid && (currTempo.curve != -1.0f && currTempo.curve != 1.0f))
            numSubdivisions = static_cast<int> (std::clamp (4.0 * (tempos[tempoIdx].startBeat - currentBeat).inBeats(), 1.0, 100.0));

        const auto numBeats = BeatDuration::fromBeats ((i < beatsWithChanges.size() - 1)
                                                         ? ((beatsWithChanges[i + 1] - currentBeat).inBeats() / (double) numSubdivisions)
                                                         : 1.0e6);

        for (int k = 0; k < numSubdivisions; ++k)
        {
            Section it;

            it.bpm              = bpm;
            it.numerator        = currTimeSig.numerator;
            it.prevNumerator    = it.numerator;
            it.denominator      = currTimeSig.denominator;
            it.triplets         = currTimeSig.triplets;
            it.startTime        = time;
            it.startBeat        = beatNum;

            it.secondsPerBeat   = SecondsPerBeat { useDenominator ? (240.0 / (bpm * it.denominator))
                                                                  : (60.0 / bpm) };
            it.beatsPerSecond   = 1.0 / it.secondsPerBeat;

            it.ppqAtStart = ppq;
            ppq += 4 * numBeats.inBeats() / it.denominator;

            it.key              = currKey.key;

            if (sections.empty())
            {
                it.barNumberOfFirstBar = 0;
                it.beatsUntilFirstBar = {};
                it.timeOfFirstBar = {};
            }
            else
            {
                const auto& prevSection = sections[sections.size() - 1];

                const auto beatsSincePreviousBarUntilStart = (time - prevSection.timeOfFirstBar) * prevSection.beatsPerSecond;
                const auto barsSincePrevBar = (int) std::ceil (beatsSincePreviousBarUntilStart.inBeats() / prevSection.numerator - 1.0e-5);

                it.barNumberOfFirstBar = prevSection.barNumberOfFirstBar + barsSincePrevBar;

                const auto beatNumInEditOfNextBar = BeatPosition::fromBeats ((int) std::lround ((prevSection.startBeat + prevSection.beatsUntilFirstBar).inBeats())
                                                                             + (barsSincePrevBar * prevSection.numerator));

                it.beatsUntilFirstBar = beatNumInEditOfNextBar - it.startBeat;
                it.timeOfFirstBar = time + it.beatsUntilFirstBar * it.secondsPerBeat;

                for (int j = (int) sections.size(); --j >= 0;)
                {
                    auto& tempo = sections[(size_t) j];

                    if (tempo.barNumberOfFirstBar < it.barNumberOfFirstBar)
                    {
                        it.prevNumerator = tempo.numerator;
                        break;
                    }
                }
            }

            sections.push_back (it);

            time = time + numBeats * it.secondsPerBeat;
            beatNum = beatNum + numBeats;

            bpm = nextTempoValid ? calcCurveBpm (beatNum.inBeats(), currTempo, tempos[tempoIdx])
                                 : currTempo.bpm;
        }
    }
}
```

Curve Calculations

```
inline double Sequence::calcCurveBpm (double beat, const TempoChange t1, const TempoChange t2)
{
    const auto b1 = t1.startBeat.inBeats();
    const auto b2 = t2.startBeat.inBeats();
    const auto bpm1 = t1.bpm;
    const auto bpm2 = t2.bpm;
    const auto c = t1.curve;

    const auto [x, y] = getBezierPoint (b1, bpm1, b2, bpm2, c);

    if (c >= -0.5 && c <= 0.5)
        return getBezierYFromX (beat,
                                b1, bpm1, x, y, b2, bpm2);

    double x1end = 0;
    double x2end = 0;
    double y1end = 0;
    double y2end = 0;

    getBezierEnds (b1, bpm1, b2, bpm2, c,
                   x1end, y1end, x2end, y2end);

    if (beat >= b1 && beat <= x1end)
        return y1end;

    if (beat >= x2end && beat <= b2)
        return y2end;

    return getBezierYFromX (beat, x1end, y1end, x, y, x2end, y2end);
}

inline std::pair<double /*x*/, double /*y*/> getBezierPoint (double x1, double y1, double x2, double y2,
                                                             double c) noexcept
{
    if (y2 > y1)
    {
        auto run  = x2 - x1;
        auto rise = y2 - y1;

        auto xc = x1 + run / 2;
        auto yc = y1 + rise / 2;

        auto x = xc - run  / 2 * -c;
        auto y = yc + rise / 2 * -c;

        return { x, y };
    }

    auto run  = x2 - x1;
    auto rise = y1 - y2;

    auto xc = x1 + run / 2;
    auto yc = y2 + rise / 2;

    auto x = xc - run  / 2 * -c;
    auto y = yc - rise / 2 * -c;

    return { x, y };
}

inline void getBezierEnds (const double x1, const double y1, const double x2, const double y2, const double c,
                           double& x1out, double& y1out, double& x2out, double& y2out) noexcept
{
    auto minic = (std::abs (c) - 0.5f) * 2.0f;
    auto run   = minic * (x2 - x1);
    auto rise  = minic * ((y2 > y1) ? (y2 - y1) : (y1 - y2));

    if (c > 0)
    {
        x1out = x1 + run;
        y1out = (float) y1;

        x2out = x2;
        y2out = (float) (y1 < y2 ? (y2 - rise) : (y2 + rise));
    }
    else
    {
        x1out = x1;
        y1out = (float) (y1 < y2 ? (y1 + rise) : (y1 - rise));

        x2out = x2 - run;
        y2out = (float) y2;
    }
}

inline double getBezierYFromX (double x, double x1, double y1, double xb, double yb, double x2, double y2) noexcept
{
    // test for straight lines and bail out
    if (x1 == x2 || y1 == y2)
        return y1;

    // test for endpoints
    if (x <= x1)  return y1;
    if (x >= x2)  return y2;

    // ok, we have a bezier curve with one control point,
    // we know x, we need to find y

    // flip the bezier equation around so its an quadratic equation
    auto a = x1 - 2 * xb + x2;
    auto b = -2 * x1 + 2 * xb;
    auto c = x1 - x;

    // solve for t, [0..1]
    double t;

    if (a == 0)
    {
        t = -c / b;
    }
    else
    {
        t = (-b + std::sqrt (b * b - 4 * a * c)) / (2 * a);

        if (t < 0.0f || t > 1.0f)
            t = (-b - std::sqrt (b * b - 4 * a * c)) / (2 * a);
    }

    jassert (t >= 0.0f && t <= 1.0f);

    // find y using the t we just found
    auto y = (std::pow (1 - t, 2) * y1) + 2 * t * (1 - t) * yb + std::pow (t, 2) * y2;
    return y;
}
```

Beats <-> Time <-> Bars

```
inline BeatPosition toBeats (const std::vector<Sequence::Section>& sections, TimePosition time)
{
    for (int i = (int) sections.size(); --i > 0;)
    {
        auto& it = sections[(size_t) i];

        if (it.startTime <= time)
            return it.startBeat + (time - it.startTime) * it.beatsPerSecond;
    }

    auto& it = sections[0];
    return it.startBeat + ((time - it.startTime) * it.beatsPerSecond);
}

inline TimePosition toTime (const std::vector<Sequence::Section>& sections, BeatPosition beats)
{
    for (int i = (int) sections.size(); --i >= 0;)
    {
        auto& it = sections[(size_t) i];

        if (toPosition (beats - it.startBeat) >= BeatPosition())
            return it.startTime + it.secondsPerBeat * (beats - it.startBeat);
    }

    auto& it = sections[0];
    return it.startTime + it.secondsPerBeat * (beats - it.startBeat);
}

inline TimePosition toTime (const std::vector<Sequence::Section>& sections, BarsAndBeats barsBeats)
{
    for (int i = (int) sections.size(); --i >= 0;)
    {
        const auto& it = sections[(size_t) i];

        if (it.barNumberOfFirstBar == barsBeats.bars + 1
              && barsBeats.beats.inBeats() >= it.prevNumerator - it.beatsUntilFirstBar.inBeats())
            return it.timeOfFirstBar - it.secondsPerBeat * (BeatDuration::fromBeats (it.prevNumerator) - barsBeats.beats);

        if (it.barNumberOfFirstBar <= barsBeats.bars || i == 0)
            return it.timeOfFirstBar + it.secondsPerBeat * (BeatDuration::fromBeats (((barsBeats.bars - it.barNumberOfFirstBar) * it.numerator)) + barsBeats.beats);
    }

    return {};
}

inline BarsAndBeats toBarsAndBeats (const std::vector<Sequence::Section>& sections, TimePosition time)
{
    for (int i = (int) sections.size(); --i >= 0;)
    {
        auto& it = sections[(size_t) i];

        if (it.startTime <= time || i == 0)
        {
            const auto beatsSinceFirstBar = ((time - it.timeOfFirstBar) * it.beatsPerSecond).inBeats();

            if (beatsSinceFirstBar < 0)
                return { it.barNumberOfFirstBar + (int) std::floor (beatsSinceFirstBar / it.numerator),
                         BeatDuration::fromBeats (std::fmod (std::fmod (beatsSinceFirstBar, it.numerator) + it.numerator, it.numerator)),
                         it.numerator };

            return { it.barNumberOfFirstBar + (int) std::floor (beatsSinceFirstBar / it.numerator),
                     BeatDuration::fromBeats (std::fmod (beatsSinceFirstBar, it.numerator)),
                     it.numerator };
        }
    }

    return { 0, {} };
}
```

D: 16 £5k

Dynamic Tempo

D: 16 £5k

Sequence Creation

```
inline Sequence::Sequence (std::vector<TempoChange> tempos, std::vector<TimeSigChange> timeSigs, std::vector<KeyChange> keys,
                           LengthOfOneBeat lengthOfOneBeat)
{
    if (keys.empty())
        keys.push_back ({});

    assert (tempos.size() > 0 && timeSigs.size() > 0);
    assert (tempos[0].startBeat == BeatPosition());
    assert (timeSigs[0].startBeat == BeatPosition());
    assert (keys[0].startBeat == BeatPosition());

    // Find the beats with changes
    std::vector<BeatPosition> beatsWithChanges;
    beatsWithChanges.reserve (tempos.size() + timeSigs.size() + keys.size());

    for (const auto& tempo : tempos)
    {
        beatsWithChanges.push_back (tempo.startBeat);

        hash_combine (hashCode, tempo.startBeat.inBeats());
        hash_combine (hashCode, tempo.bpm);
        hash_combine (hashCode, tempo.curve);
    }

    for (const auto& timeSig : timeSigs)
    {
        beatsWithChanges.push_back (timeSig.startBeat);

        hash_combine (hashCode, timeSig.startBeat.inBeats());
        hash_combine (hashCode, timeSig.numerator);
        hash_combine (hashCode, timeSig.denominator);
        hash_combine (hashCode, timeSig.triplets);
    }

    for (const auto& keyChange : keys)
    {
        beatsWithChanges.push_back (keyChange.startBeat);

        hash_combine (hashCode, keyChange.startBeat.inBeats());
        hash_combine (hashCode, keyChange.key.pitch);
        hash_combine (hashCode, keyChange.key.scale);
    }

    std::sort (beatsWithChanges.begin(), beatsWithChanges.end());
    beatsWithChanges.erase (std::unique (beatsWithChanges.begin(), beatsWithChanges.end()),
                            beatsWithChanges.end());

    // Build the sections
    TimePosition time;
    BeatPosition beatNum;
    double ppq              = 0.0;
    size_t timeSigIdx   = 0;
    size_t tempoIdx     = 0;
    size_t keyIdx       = 0;

    auto currTempo    = tempos[tempoIdx++];
    auto currTimeSig  = timeSigs[timeSigIdx++];
    auto currKey      = keys[keyIdx++];

    const bool useDenominator = lengthOfOneBeat == LengthOfOneBeat::dependsOnTimeSignature;

    for (size_t i = 0; i < beatsWithChanges.size(); ++i)
    {
        const auto currentBeat = beatsWithChanges[i];
        assert (std::abs ((currentBeat - beatNum).inBeats()) < 0.001);

        while (tempoIdx < tempos.size() && tempos[tempoIdx].startBeat == currentBeat)
            currTempo = tempos[tempoIdx++];

        if (timeSigIdx < timeSigs.size() && timeSigs[timeSigIdx].startBeat == currentBeat)
            currTimeSig = timeSigs[timeSigIdx++];

        if (keyIdx < keys.size() && keys[keyIdx].startBeat == currentBeat)
            currKey = keys[keyIdx++];

        const bool nextTempoValid = tempoIdx < tempos.size();
        double bpm = nextTempoValid ? calcCurveBpm (currTempo.startBeat.inBeats(), currTempo, tempos[tempoIdx])
                                    : currTempo.bpm;

        int numSubdivisions = 1;

        if (nextTempoValid && (currTempo.curve != -1.0f && currTempo.curve != 1.0f))
            numSubdivisions = static_cast<int> (std::clamp (4.0 * (tempos[tempoIdx].startBeat - currentBeat).inBeats(), 1.0, 100.0));

        const auto numBeats = BeatDuration::fromBeats ((i < beatsWithChanges.size() - 1)
                                                         ? ((beatsWithChanges[i + 1] - currentBeat).inBeats() / (double) numSubdivisions)
                                                         : 1.0e6);

        for (int k = 0; k < numSubdivisions; ++k)
        {
            Section it;

            it.bpm              = bpm;
            it.numerator        = currTimeSig.numerator;
            it.prevNumerator    = it.numerator;
            it.denominator      = currTimeSig.denominator;
            it.triplets         = currTimeSig.triplets;
            it.startTime        = time;
            it.startBeat        = beatNum;

            it.secondsPerBeat   = SecondsPerBeat { useDenominator ? (240.0 / (bpm * it.denominator))
                                                                  : (60.0 / bpm) };

            it.beatsPerSecond   = 1.0 / it.secondsPerBeat;

            it.ppqAtStart = ppq;
            ppq += 4 * numBeats.inBeats() / it.denominator;

            it.key              = currKey.key;

            if (sections.empty())
            {
                it.barNumberOfFirstBar = 0;
                it.beatsUntilFirstBar = {};
                it.timeOfFirstBar = {};
            }
            else
            {
                const auto& prevSection = sections[sections.size() - 1];

                const auto beatsSincePreviousBarUntilStart = (time - prevSection.timeOfFirstBar) * prevSection.beatsPerSecond;
                const auto barsSincePrevBar = (int) std::ceil (beatsSincePreviousBarUntilStart.inBeats() / prevSection.numerator - 1.0e-5);

                it.barNumberOfFirstBar = prevSection.barNumberOfFirstBar + barsSincePrevBar;

                const auto beatNumInEditOfNextBar = BeatPosition::fromBeats ((int) std::lround ((prevSection.startBeat + prevSection.beatsUntilFirstBar).inBeats())
                                                                             + (barsSincePrevBar * prevSection.numerator));

                it.beatsUntilFirstBar = beatNumInEditOfNextBar - it.startBeat;
                it.timeOfFirstBar = time + it.beatsUntilFirstBar * it.secondsPerBeat;

                for (int j = (int) sections.size(); --j >= 0;)
                {
                    auto& tempo = sections[(size_t) j];

                    if (tempo.barNumberOfFirstBar < it.barNumberOfFirstBar)
                    {
                        it.prevNumerator = tempo.numerator;
                        break;
                    }
                }
            }

            sections.push_back (it);

            time = time + numBeats * it.secondsPerBeat;
            beatNum = beatNum + numBeats;

            bpm = nextTempoValid ? calcCurveBpm (beatNum.inBeats(), currTempo, tempos[tempoIdx])
                                 : currTempo.bpm;
        }
    }
}
```

Curve Calculations

```
inline double Sequence::calcCurveBpm (double beat, const TempoChange t1, const TempoChange t2)
{
    const auto b1 = t1.startBeat.inBeats();
    const auto b2 = t2.startBeat.inBeats();
    const auto bpm1 = t1.bpm;
    const auto bpm2 = t2.bpm;
    const auto c = t1.curve;

    const auto [x, y] = getBezierPoint (b1, bpm1, b2, bpm2, c);

    if (c >= -0.5 && c <= 0.5)
        return getBezierYFromX (beat,
                                b1, bpm1, x, y, b2, bpm2);

    double x1end = 0;
    double x2end = 0;
    double y1end = 0;
    double y2end = 0;

    getBezierEnds (b1, bpm1, b2, bpm2, c,
                   x1end, y1end, x2end, y2end);

    if (beat >= b1 && beat <= x1end)
        return y1end;

    if (beat >= x2end && beat <= b2)
        return y2end;

    return getBezierYFromX (beat, x1end, y1end, x, y, x2end, y2end);
}

inline std::pair<double /*x*/, double /*y*/> getBezierPoint (double x1, double y1, double x2, double y2,
                                                             double c) noexcept
{
    if (y2 > y1)
    {
        auto run  = x2 - x1;
        auto rise = y2 - y1;

        auto xc = x1 + run / 2;
        auto yc = y1 + rise / 2;

        auto x = xc - run  / 2 * -c;
        auto y = yc + rise / 2 * -c;

        return { x, y };
    }

    auto run  = x2 - x1;
    auto rise = y1 - y2;

    auto xc = x1 + run / 2;
    auto yc = y2 + rise / 2;

    auto x = xc - run  / 2 * -c;
    auto y = yc - rise / 2 * -c;

    return { x, y };
}

inline void getBezierEnds (const double x1, const double y1, const double x2, const double y2, const double c,
                           double& x1out, double& y1out, double& x2out, double& y2out) noexcept
{
    auto minic = (std::abs (c) - 0.5f) * 2.0f;
    auto run   = minic * (x2 - x1);
    auto rise  = minic * ((y2 > y1) ? (y2 - y1) : (y1 - y2));

    if (c > 0)
    {
        x1out = x1 + run;
        y1out = (float) y1;

        x2out = x2;
        y2out = (float) (y1 < y2 ? (y2 - rise) : (y2 + rise));
    }
    else
    {
        x1out = x1;
        y1out = (float) (y1 < y2 ? (y1 + rise) : (y1 - rise));

        x2out = x2 - run;
        y2out = (float) y2;
    }
}

inline double getBezierYFromX (double x, double x1, double y1, double xb, double yb, double x2, double y2) noexcept
{
    // test for straight lines and bail out
    if (x1 == x2 || y1 == y2)
        return y1;

    // test for endpoints
    if (x <= x1)  return y1;
    if (x >= x2)  return y2;

    // ok, we have a bezier curve with one control point,
    // we know x, we need to find y

    // flip the bezier equation around so its an quadratic equation
    auto a = x1 - 2 * xb + x2;
    auto b = -2 * x1 + 2 * xb;
    auto c = x1 - x;

    // solve for t, [0..1]
    double t;

    if (a == 0)
    {
        t = -c / b;
    }
    else
    {
        t = (-b + std::sqrt (b * b - 4 * a * c)) / (2 * a);

        if (t < 0.0f || t > 1.0f)
            t = (-b - std::sqrt (b * b - 4 * a * c)) / (2 * a);
    }

    jassert (t >= 0.0f && t <= 1.0f);

    // find y using the t we just found
    auto y = (std::pow (1 - t, 2) * y1) + 2 * t * (1 - t) * yb + std::pow (t, 2) * y2;
    return y;
}
```

Beats <-> Time <-> Bars

```
inline BeatPosition toBeats (const std::vector<Sequence::Section>& sections, TimePosition time)
{
    for (int i = (int) sections.size(); --i > 0;)
    {
        auto& it = sections[(size_t) i];

        if (it.startTime <= time)
            return it.startBeat + (time - it.startTime) * it.beatsPerSecond;
    }

    auto& it = sections[0];
    return it.startBeat + ((time - it.startTime) * it.beatsPerSecond);
}

inline TimePosition toTime (const std::vector<Sequence::Section>& sections, BeatPosition beats)
{
    for (int i = (int) sections.size(); --i >= 0;)
    {
        auto& it = sections[(size_t) i];

        if (toPosition (beats - it.startBeat) >= BeatPosition())
            return it.startTime + it.secondsPerBeat * (beats - it.startBeat);
    }

    auto& it = sections[0];
    return it.startTime + it.secondsPerBeat * (beats - it.startBeat);
}

inline TimePosition toTime (const std::vector<Sequence::Section>& sections, BarsAndBeats barsBeats)
{
    for (int i = (int) sections.size(); --i >= 0;)
    {
        const auto& it = sections[(size_t) i];

        if (it.barNumberOfFirstBar == barsBeats.bars + 1
             && barsBeats.beats.inBeats() >= it.prevNumerator - it.beatsUntilFirstBar.inBeats())
            return it.timeOfFirstBar - it.secondsPerBeat * (BeatDuration::fromBeats (it.prevNumerator) - barsBeats.beats);

        if (it.barNumberOfFirstBar <= barsBeats.bars || i == 0)
            return it.timeOfFirstBar + it.secondsPerBeat * (BeatDuration::fromBeats (((barsBeats.bars - it.barNumberOfFirstBar) * it.numerator)) + barsBeats.beats);
    }

    return {};
}

inline BarsAndBeats toBarsAndBeats (const std::vector<Sequence::Section>& sections, TimePosition time)
{
    for (int i = (int) sections.size(); --i >= 0;)
    {
        auto& it = sections[(size_t) i];

        if (it.startTime <= time || i == 0)
        {
            const auto beatsSinceFirstBar = ((time - it.timeOfFirstBar) * it.beatsPerSecond).inBeats();

            if (beatsSinceFirstBar < 0)
                return { it.barNumberOfFirstBar + (int) std::floor (beatsSinceFirstBar / it.numerator),
                         BeatDuration::fromBeats (std::fmod (std::fmod (beatsSinceFirstBar, it.numerator) + it.numerator, it.numerator)),
                         it.numerator };

            return { it.barNumberOfFirstBar + (int) std::floor (beatsSinceFirstBar / it.numerator),
                     BeatDuration::fromBeats (std::fmod (beatsSinceFirstBar, it.numerator)),
                     it.numerator };
        }
    }

    return { 0, {} };
}
```

Audio File Reading