timingsrc v3

Release 1.0

Jun 12, 2023

Main

Timingsrc is hosted at GitHub.

Timingsrc

A programming model for time sensitive Web applications, based on the Timing Object. Precise timing, synchronization and control enabled for single-device and multi-device Web applications.

CHAPTER 1

Introduction

The Web is arguably the most important platform for multi-media, with universal reach and a rich selection of powerful media frameworks. This includes built-in frameworks such as MediaElement, WebAudio, WebGL, and WebAnimation – and also a host of external frameworks, extensions, plugins, components or tools for rendering or visualizing all kinds of data and media types.

With so many powerful frameworks for rendering and visualization, the idea of combining them is both intuitive and highly attractive. After all, flexible composition is a defining characteristic of the Web. For example, live coverage of sport car racing might target co-presentation of a number of media types, including camera angles, audio commentary, sound effects, data-driven infographics, animated maps, social media and more.

However, co-presentation of timed media content requires fairly precise synchronization, and the Web has little support for this. The Web is primarily a platform for **embedding** independent media frameworks. It offers no particular mechanism for precise **coordination** across different media frameworks.

The consequences of this are quite visible. Media providers are eagerly extending their offerings with more data sources and streams, yet without the ability to time-align them correctly, user experiences may quickly become inconsistent, annoying, confusing, or simply broken. A well known example is soccer goal alerts going off 30 seconds before the goal *happens* in the live video stream.

In the media industry, low-latency streaming is sometimes suggested as a remedy for such issues. This though, assumes that media content is a single video stream, or that all media contents can be assembled into a single container ahead of distribution. Importantly, the promise of the IP/Web domain is quite the opposite, with media experiences assembled on consumer devices (i.e. late binding) leveraging a multitude of independent content sources, distribution mechanims and production chains, as well as exploiting a variety of data formats and interactive rendering technologies. In this world, different production chains may yield substantially different end-to-end delays, and low-latency streaming may even contribute to the variation.

So, the core issue is not with data distribution, but rather with the user experience. Going back to the fundamentals, media experiences have always been defined with a concept of presentation timeline at heart, as a basis for consistent presentation/playback of media content. The core problem right now is that each content source defines its own presentation timeline and simply expects the user to adopt it. Clearly, this approach does not scale beyond a single content source. Instead, what is needed is:

1) A user timeline. An independent presentation timeline for the user experience

2) The ability to align the presentation timelines of each content source / media framework to the user timeline

3) The ability to synchronize user timelines across connected devices, allowing consistent media experiences spanning multiple devices.

It appears that such a timeline concept will be central going forward, as linear media is gradually being re-invented for the IP/Web domain, Yet the Web platform does not have such a concept. As gaps go, this gap is pretty significant. And to be frank, rather embarrassing too. The Web is arguably the most important multi-media platform world wide. Yet, ironically, multi-media playback is largely delegated to embedded frameworks, and cross framework playback is simply not supported.

Timingsrc is a JavaScript framework adressing this gap. Timingsrc introduces a much needed programming model for timing, synchronization and control on the Web. The central concept is the *Timing Object*. It provides a generic timeline concept with time-controls and events. Media frameworks connected to a shared timing object may align their internal presentation timelines precisely to the timing object and implement swift reactions to shared media control. Furthermore, timing objects may be synchronized globally, if connected to *online timing providers* such as *Shared Motion Timing Provider* for global synchroninization. An introduction to the *Timing Object* programming model is published as *the motion model* in a book chapter titled Media Synchronization on the Web.

CHAPTER 2

Module

2.1 Source code

Timingsrc at GitHub.

2.2 Include as script

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript" src="https://webtiming.github.io/timingsrc/lib/
↪timingsrc-v3.js">
        </script>
        <script type="text/javascript">
            console.log(`hello world timingsrc version ${TIMINGSRC.version}!`);
        </script>
    </head>
    <body>
    </body>
</html>
```

Full source https://webtiming.github.io/timingsrc/lib/timingsrc-v3.js

Minified source https://webtiming.github.io/timingsrc/lib/timingsrc-min-v3.js

2.3 Include as ES6 module

```
<!DOCTYPE html>
<html>
```

(continues on next page)

(continued from previous page)

```
    <head>
        <script type="module">
            import * as TIMINGSRC from "https://webtiming.github.io/timingsrc/lib/
↪timingsrc-esm-v3.js";
            console.log(`hello world timingsrc version ${TIMINGSRC.version}!`);
        </script>
    </head>
    <body>
    </body>
</html>
```

Full source https://webtiming.github.io/timingsrc/lib/timingsrc-esm-v3.js

Minified source https://webtiming.github.io/timingsrc/lib/timingsrc-min-esm-v3.js

2.4 Namespace

```
// utils
export * as utils from './util/utils.js';
export * as motionutils from './util/motionutils.js';
export {default as BinarySearch} from './util/binarysearch.js';
export {default as endpoint} from './util/endpoint.js';
export {default as eventify} from './util/eventify.js';
export {default as Interval} from './util/interval.js';
export {default as CueCollection} from './dataset/cuecollection.js';
export {default as Timeout} from './util/timeout.js';

// timing object
export {default as TimingObject} from './timingobject/timingobject.js';
export {default as SkewConverter} from './timingobject/skewconverter.js';
export {default as DelayConverter} from './timingobject/delayconverter.js';
export {default as ScaleConverter} from './timingobject/scaleconverter.js';
export {default as LoopConverter} from './timingobject/loopconverter.js';
export {default as RangeConverter} from './timingobject/rangeconverter.js';
export {default as TimeshiftConverter} from './timingobject/timeshiftconverter.js';
export {default as TimingSampler} from './timingobject/timingsampler.js';
export {default as PositionCallback} from './timingobject/positioncallback.js';

// timed data
export {default as Dataset} from './dataset/dataset.js';
export {default as Subset} from './dataset/subset.js';
import {default as PointModeSequencer} from './sequencing/pointsequencer.js';
import {default as IntervalModeSequencer} from './sequencing/intervalsequencer.js';
export function Sequencer(axis, toA, toB) {
    if (toB === undefined) {
        return new PointModeSequencer(axis, toA);
    } else {
        return new IntervalModeSequencer(axis, toA, toB);
    }
};

// ui
export {default as DatasetViewer} from './ui/datasetviewer.js';
export {default as TimingProgress} from './ui/timingprogress.js';
```

(continues on next page)

(continued from previous page)

```

export const version = "v3.0";
```

CHAPTER 3

Quickstart

This quickstart tutorial demonstrates playback of any kind of timed data.

3.1 Step 1 : Create a Webpage

Setup a webpage and initialise key **timingsrc** concepts:

- *Timing Object* for playback control
- *Dataset* for cue management
- *Sequencer* for cue playback

```
<!DOCTYPE html>
<html>
    <head>
        <script type="module">
            import {TimingObject, Dataset, Sequencer, Interval} from "https://
↪webtiming.github.io/timingsrc/lib/timingsrc-module-v3.js";
            const to = new TimingObject();
            const ds = new Dataset();
            const activeCues = new Sequencer(ds, to);
            ...
        </script>
    </head>
    <body>
        <div id="cues"></div>
        ...
    </body>
</html>
```

Also, unless the timing object is to be remote controlled by an external timing object, the webpage needs to define some playback controls, for instance see: example-basic-controls.

3.2 Step 2 : Load cues

There are no restrictions regarding data format or data source. As long as data can be made available within the webpage, it can be used in timed presentation. In this example, we will simply use som mock data.

```
// mockup timed data
const data = [
    {id:"a", text: 'A', start: 0, end: 1 },
    {id:"b", text: 'B', start: 2, end: 3 },
    {id:"c", text: 'C', start: 4, end: 5 },
    {id:"d", text: 'D', start: 6, end: 7 },
    {id:"e", text: 'E', start: 8, end: 9 },
    ...
];

// make cues
const cues = data.map(item => {
    return {
        key: item.id,
        interval: new Interval(item.start, item.end),
        data: item
    };
});

// load into dataset
ds.update(cues);
```

3.3 Step 3 : Render dataset cues

In this example the dataset is assumed to be static. If the dataset is dynamic, use **change**, **remove** events to keep the visualization up to date.

```
// construct a list from dataset cues
document.getElementById("cues").innerHTML = [...ds.values()]
    .map(function(cue){
        let text = JSON.stringify(cue.data);
        if (activeCues.has(cue.key)) {
            return `<div id=${cue.key} class="active">${text}</div>`;
        } else {
            return `<div id=${cue.key}>${text}</div>`;
        }
    })
    .join("\n");
```

3.4 Step 4 : Render active cues

Finally, render active cues by specifying what happens when:

- an *inactive* cue becomes *active*
- an *active* cue becomes *inactive*

In this example, this is done simply by setting or removing the *css* classname *active* on cue list elements.

```
activeCues.on("change", (eArg, eInfo) => {
    let el = document.getElementById(eArg.key);
    if (el) {
        el.classList.add("active");
    }
});

activeCues.on("remove", (eArg, eInfo) => {
    let el = document.getElementById(eArg.key);
    if (el) {
        el.classList.remove("active");
    }
});
```

3.5 Ready

Ready to load the page and start controlling the timing object.

CHAPTER 4

Version

Timingsrc v3 is a major revision on timingsrc v2. The two versions are functionally equivalent. However, as version 3 makes adjustments to the API's, v3 is **not backwards compatible** with v2.

4.1 Major changes

- In v2 the *Sequencer* did cue management internally and the Sequencer provided access both to **cues** and **active cues**. In v3 cue management is made explicit by introducing the *Dataset* as an independent concept. In v3 the pattern is to create a dataset an then to create one or more sequencers connected to the dataset.
- V2 had issues with efficiency as update batch sizes grew beyond O(1K) cues. V3 is a reimplementation with efficiency in focus providing scalable performance measures for update batch sizes at least beyond O(100K), see *Dataset Performance*.
- V3 simplifies and aligns API's of dataset and sequencer. Both concepts are collections of cues with identical API for cue access. Datasets are used for cue management, whereas sequencers are used for playback. A sequencer provides a dynamic view into a the **active cues** of its dataset.
- In v3, dataset have become a valuable tool for management, lookup and visualization of timed data, useful also without sequencers.

4.2 Minor changes

- Unsubscribe from events `EventProviderInterface.off()` is changed so that it takes a subscription handle returned by `EventProviderInterface.on()`.
- Event type **events** in v2 is renamed to **batch**, see *Batch Event*.
- The sequencer constructor signature changed from *Sequencer(toA[, toB])* to *Sequencer(dataset, toA[, toB])*.
- Sequencers no longer support primitives for cue manipulation. This is now handled exclusively by the dataset, see *Dataset Update*.

- Sequencer events no longer contain detailed information about the cause of the event, such as movement direction and interval entry point.
- Sequencer no longer optimises precision of *setTimeout* as was the case in v2.
- V3 uses modern Javascripts features such as *class*, *arrow functions* and *module imports*.
- V3 also brings extensive code cleanup, refactoring, improved code design and more unittests for internal modules.

CHAPTER 5

Standardization

The W3C Multi-device Timing Community Group was created in 2015 to advocate standardization of the *Timing Object* as the core part of a much needed timing model for the Web. As part of this initiative, a the Timing Object Draft Specification was published and timingsrc was created as a reference implementaion for this proposal.

Since then, the *Multi-device Timing Community Group* has been included within the scope of the Media and Entertainment Interest Group, responsible for standardization of Web technologies related to media. *Multi-device Timing* is also included in the roadmap of the interest group. Beyond this, the *Media and Entertainment Interest Group* has not yet addressed the *gap* concerning time controls across media components and frameworks.

Current standardization activities (2020) are still predominantly **media centric** as they mostly address synchronization relative to HTML5 media playback. As a general approach though, this is both limiting and short sighted, making it an unfortunate choice of timing model for the Web (see Media Synchronization on the Web).

The *Timing Object* is the foundation for a **new timing model**, opening up for synchronization and consistency across media sources, media types, media components or media frameworks. Also, crucially, this timing model expands the scope of synchronization and consistency from local media experiences (i.e. within a Web page) to globally distributed media experiences.

Though no formal steps have been taken with respect to standardization of the *Timing Object*, the timingsrc JavaScript implementation is ready to use. It has has been maturing through steady use since 2015, and recently it is seeing increased usage from Web programmers around the world, not least after Corona. It seems the boost of online activity is making issues with synchronization and consistency even more evident.

Note: The Timing Object Draft Specification has not been updated since its original publications, so deviations made by the timingsrc implementation have yet not been included.

CHAPTER 6

Contributions

6.1 Authors

Ingar M. Arntzen

- mailto://inar@norceresearch.no
- mailto://ingar.arntzen@motioncorporation.com
- https://github.com/ingararntzen
- https://www.linkedin.com/in/ingararntzen/

Njål T. Borch

- mailto://njbo@norceresearch.no
- mailto://njaal.borch@motioncorporation.com
- https://github.com/snarkdoof
- https://www.linkedin.com/in/njaal-borch-5754a11/

6.2 Acknowledgements

Version 1 of timingsrc were developed at Norut Northern Research Institute, now part of Norwegian Research Centre (NORCE), and funded in part by MediaScape, an EU FP7 project. Version 2 of timingsrc was developed and maintained across several media projects at NORCE. Version 3 is a private contribution by Ingar Arntzen.

Ingar Arntzen is the main developer for *Timing Object*, *Timing Converter*, *Dataset* and *Sequencer*. Njål Borch is the main developer for *MediaSync*, the adapter for synchronization of HTML5 media elements.

CHAPTER 7

Demo TimingObject

This demonstrates control and rendering of the *Timing Object*.

- Control position, velocity or acceleration by clicking the buttons. **P+1** means to increment the position. **V=0** is to set the velocity to zero
- Position may also be controlled by clicking the progress timeline.

Demo

demofile

7.1 Code

```
<!DOCTYPE html>
<html>
    <head>

        <style type="text/css">
            .ctrl-label {
                display:inline-block;
                width:100px;
            }
            .ctrl-btn button {
                width:50px;
            }
            .progress {
                width: 100%; /* Full-width */
                appearance: none;
                border-radius: 5px;
                height: 5px;
                background: #d3d3d3; /* Grey background */
```

(continues on next page)

(continued from previous page)

```
                outline: none;
            }
        </style>

        <script type="module">

            import {
                TimingObject,
                TimingSampler,
                TimingProgress
            } from "https://webtiming.github.io/timingsrc/lib/timingsrc-esm-v3.js";

            /*
                Create TimingObject
            */
            const to = new TimingObject({range:[0,10]});

            /*
                Visualize Timing Object Position

                Either use timeupdate event for fixed frequency
                sampling, or create sampler at custom frequency.
            */
            const pos_elem = document.getElementById("position");
            const vel_elem = document.getElementById("velocity");
            const acc_elem = document.getElementById("acceleration");
            const rng_elem = document.getElementById("range");

            // refresh position every 100 ms
            const sampler = new TimingSampler(to, {period:100});
            sampler.on("change", function () {
                let vector = to.query();
                let rng = to.range;
                pos_elem.innerHTML = `${vector.position.toFixed(2)}`;
                vel_elem.innerHTML = `${vector.velocity.toFixed(2)}`;
                acc_elem.innerHTML = `${vector.acceleration.toFixed(2)}`;
                rng_elem.innerText = `[${rng}]`;
            });

            // progress
            const progress_elem = document.getElementById("progress");
            const progress = new TimingProgress(to,
                                               progress_elem,
↪{sampler:sampler});

            /*
                Connect buttons
            */
            document.getElementById("p-").onclick = function () {
                to.update({position:to.pos-1});
            };
            document.getElementById("p").onclick = function () {
                to.update({position:0});
            };
            document.getElementById("p+").onclick = function () {
                to.update({position:to.pos+1});
            };
```

(continues on next page)

(continued from previous page)

```
            document.getElementById("v-").onclick = function () {
                to.update({velocity:to.vel-1});
            };
            document.getElementById("v").onclick = function () {
                to.update({velocity:0});
            };
            document.getElementById("v+").onclick = function () {
                to.update({velocity:to.vel+1});
            };
            document.getElementById("a-").onclick = function () {
                to.update({acceleration:to.acc-1});
            };
            document.getElementById("a").onclick = function () {
                to.update({acceleration:0});
            };
            document.getElementById("a+").onclick = function () {
                to.update({acceleration:to.acc+1});
            };

        </script>
    </head>
    <body>
        <p>
            <div style="font-weight:bold;">State</div>
            <div>
                Position: <span id="position" style="color:red;"></span>
            </div>
            <div>
                Velocity: <span id="velocity"></span>
            </div>
            <div>
                Acceleration: <span id="acceleration"></span>
            </div>
            <div>
                Range: <span id="range"></span>
            </div>
        </p>
        <p>
            <input type="range" min="0" max="100" value="0" id="progress" class=
↪"progress">
        </p>
        <p >
            <div style="font-weight:bold;">Controls</div>
            <div class="ctrl-btn">
                <span class="ctrl-label">Position:</span>
                <button id="p-">P-1</button>
                <button id="p">P=0</button>
                <button id="p+">P+1</button>
            </div>
            <div class="ctrl-btn">
                <span class="ctrl-label">Velocity:</span>
                <button  id="v-">V-1</button>
                <button id="v">V=0</button>
                <button id="v+">V+1</button>
            </div>
            <div class="ctrl-btn">
```

(continues on next page)

(continued from previous page)

```
                <span class="ctrl-label">Acceleration:</span>
                <button id="a-">A-1</button>
                <button id="a">A=0</button>
                <button id="a+">A+1</button>
            </div>
        </p>
    </body>
</html>
```

CHAPTER 8

Demo TimingConverter

This demonstrates a *Timing Object* and a *Skew Converter*.

- Control either one by clicking the buttons or the progress.
- Adjust the skew by clicking one of the *set skew* buttons.

Demo

demofile

8.1 Code

```
<!DOCTYPE html>
<html>
    <head>

        <style type="text/css">
            .ctrl-label {
                display:inline-block;
                width:100px;
            }
            .ctrl-btn button {
                width:80px;
            }
            .progress {
                width: 100%; /* Full-width */
                appearance: none;
                border-radius: 5px;
                height: 5px;
                background: #d3d3d3; /* Grey background */
                outline: none;
```

(continues on next page)

(continued from previous page)

```
            }
        </style>

        <script type="module">

            import {
                TimingObject,
                TimingSampler,
                TimingProgress,
                SkewConverter
            } from "https://webtiming.github.io/timingsrc/lib/timingsrc-esm-v3.js";

            const progress_options = {range:[0,13]};

            /*
                Create TimingObject
            */
            const to = new TimingObject({range:[0,10]});
            const to_pos_elem = document.getElementById("to_pos");

            // progress
            const to_progress_elem = document.getElementById("to_progress");
            const to_progress = new TimingProgress(to,
                                                   to_progress_elem, progress_
↪options);

            /*
                Connect buttons
            */
            document.getElementById("to_reset").onclick = function () {
                to.update({position:0});
            };
            document.getElementById("to_pause").onclick = function () {
                to.update({velocity:0});
            };
            document.getElementById("to_play").onclick = function () {
                to.update({velocity:1});
            };
            document.getElementById("to_reverse").onclick = function () {
                to.update({velocity:-1});
            };


            /*
                Skew Converter
            */
            const c = new SkewConverter(to, 2);
            const c_pos_elem = document.getElementById("c_pos");
            const c_skew_elem = document.getElementById("c_skew");


            // progress
            const c_progress_elem = document.getElementById("c_progress");

            const c_progress = new TimingProgress(c,
                                                  c_progress_elem, progress_options);
```

(continues on next page)

(continued from previous page)

```
            /*
                Connect buttons
            */
            document.getElementById("c_reset").onclick = function () {
                c.update({position:0});
            };
            document.getElementById("c_pause").onclick = function () {
                c.update({velocity:0});
            };
            document.getElementById("c_play").onclick = function () {
                c.update({velocity:1});
            };
            document.getElementById("c_reverse").onclick = function () {
                c.update({velocity:-1});
            };

            // skew
            document.getElementById("skew_1").onclick = function () {
                c.skew = 1;
            };
            document.getElementById("skew_2").onclick = function () {
                c.skew = 2;
            };
            document.getElementById("skew_3").onclick = function () {
                c.skew = 3;
            };

            c.on("skewchange", () => {
                c_skew_elem.innerHTML = `${c.skew.toFixed(2)}`;
            });

            /*
                Sample positions of both timing object and converter
            */
            const sampler = new TimingSampler(c, {period:200});
            sampler.on("change", function () {
                let to_pos = to.pos;
                let c_pos = c.pos;
                to_pos_elem.innerHTML = `${to_pos.toFixed(2)}`;
                c_pos_elem.innerHTML = `${c_pos.toFixed(2)}`;
                to_progress.refresh(to_pos);
                c_progress.refresh(c_pos);
            });

        </script>
    </head>
    <body>
        <p>
            <div style="font-weight:bold;">Timing Object</div>
            <div>
                <span class="ctrl-label">Position:</span>
                <span id="to_pos" style="color:red;"></span>
            </div>
            <div class="ctrl-btn">
                <button id="to_reset">Reset</button>
                <button id="to_play">Play</button>
                <button id="to_pause">Pause</button>
```

(continues on next page)

(continued from previous page)

```
                <button id="to_reverse">Reverse</button>
            </div>
            <input type="range" min="0" max="100" value="0" id="to_progress" class=
↪"progress">
        </p>
        <p>
            <div style="font-weight:bold;">Skew Converter</div>
            <div>
                <span class="ctrl-label">Position:</span>
                <span id="c_pos" style="color:red;"></span>
            </div>
            <div>
                <span class="ctrl-label">Skew:</span>
                <span id="c_skew" style="color:red;"></span>
            </div>
            <div class="ctrl-btn">
                <span class="ctrl-label" > Set skew: </span>
                <button id="skew_1">1.0</button>
                <button id="skew_2">2.0</button>
                <button id="skew_3">3.0</button>
            </div>

        </p>
        <p>

            <div class="ctrl-btn">
                <button id="c_reset">Reset</button>
                <button id="c_play">Play</button>
                <button id="c_pause">Pause</button>
                <button id="c_reverse">Reverse</button>
            </div>
            <input type="range" min="0" max="100" value="0" id="c_progress" class=
↪"progress">

        </p>
    </body>
</html>
```

CHAPTER 9

Demo TimingProvider

This is a demo of online synchronization, based on the *Shared Motion Timing Provider*.

- Opening this page on multiple devices (or browser tabs) simultaneously to verify consistency.
- Reloade the demo on one device/tab while the demo is running on others.
- *Shared Motion Timing Provider* is hosted online, so others might be playing with the demo too.

Demo

demofile

9.1 Code

```
<!DOCTYPE html>
<html>
<head>
    <style type="text/css">
        .ctrl-btn button {
            width:80px;
        }
    </style>

    <script type="text/javascript" src="https://www.mcorp.no/lib/mcorp-2.0.js"></
↪script>
    <script type="module">

        import {TimingObject} from "https://webtiming.github.io/timingsrc/lib/
↪timingsrc-esm-v3.js";

        const to = new TimingObject();
```

(continues on next page)

(continued from previous page)

```
        // MCorp App
        const app = MCorp.app("8456579076771837888", {anon:true});
        app.ready.then(function() {
            to.timingsrc = app.motions["shared"];
        });

        // Hook up buttons UI
        document.getElementById("reset").onclick = function () {
            to.update({position:0});
        };
        document.getElementById("pause").onclick = function () {
            to.update({velocity:0});
        };
        document.getElementById("play").onclick = function () {
            to.update({velocity:1});
        };
        document.getElementById("reverse").onclick = function () {
            to.update({velocity:-1});
        };

        // Hook up text UI
        let pos_elem = document.getElementById('position');
        to.on("timeupdate", function () {
            pos_elem.innerHTML = `${to.pos.toFixed(2)}`;
        });

    </script>
</head>
<body>
    <p>
        <div>
            Position: <span id="position" style="color:red;"></span>
        </div>
        <div class="ctrl-btn">
            <button id="reset">Reset</button>
            <button id="play">Play</button>
            <button id="pause">Pause</button>
            <button id="reverse">Reverse</button>
        </div>
    </p>
</body>
</html>
```

CHAPTER 10

Demo MediaSync

Warning: The mediasync library currently has issues with Safari on iOS, presumably due to some subtle changes concerning the media support on this platform. Please try with another browser if you are having issues.

Demo

This is a demo of HTML5 video synchronization using the *Timing Object*.

- Skip to a different position by clicking the timeline progress.

demofile

10.1 Code

```
<!DOCTYPE html>
<html>
<head>
    <style type="text/css">
        .ctrl-btn button {
            width:80px;
        }
        .progress {
            width: 100%; /* Full-width */
            appearance: none;
            border-radius: 5px;
            height: 5px;
            background: #d3d3d3; /* Grey background */
            outline: none;
        }
```

(continues on next page)

(continued from previous page)

```
        #video {
            display:inline;
        }
    </style>
    <script type="text/javascript" src="https://www.mcorp.no/lib/mcorp-2.0.js"></
↪script>
    <script src="https://mcorp.no/lib/mediasync.js"></script>

    <script type="module">

        import {
                TimingObject,
                TimingSampler,
                TimingProgress
            } from "https://webtiming.github.io/timingsrc/lib/timingsrc-esm-v3.js";

        // timing object
        const to = new TimingObject({range:[0,100]});

        // MCorp App
        const app = MCorp.app("8456579076771837888", {anon:true});
        app.ready.then(function() {
            to.timingsrc = app.motions["shared"];
        });

        // Hook up buttons UI
        document.getElementById("reset").onclick = function () {
            to.update({position:0});
        };
        document.getElementById("pause").onclick = function () {
            to.update({velocity:0});
        };
        document.getElementById("play").onclick = function () {
            to.update({velocity:1});
        };
        document.getElementById("reverse").onclick = function () {
            to.update({velocity:-1});
        };


        // refresh position every 100 ms
        const sampler = new TimingSampler(to, {period:100});

        // position
        const pos_elem = document.getElementById("position");
        sampler.on("change", function () {
            pos_elem.innerHTML = `${to.pos.toFixed(2)}`;
        });

        // progress
        const progress_elem = document.getElementById("progress");
        const progress = new TimingProgress(to,
                                            progress_elem,
↪{sampler:sampler});

        // Set up video sync
        const sync1 = MCorp.mediaSync(document.getElementById('player1'), to);
```

(continues on next page)

(continued from previous page)

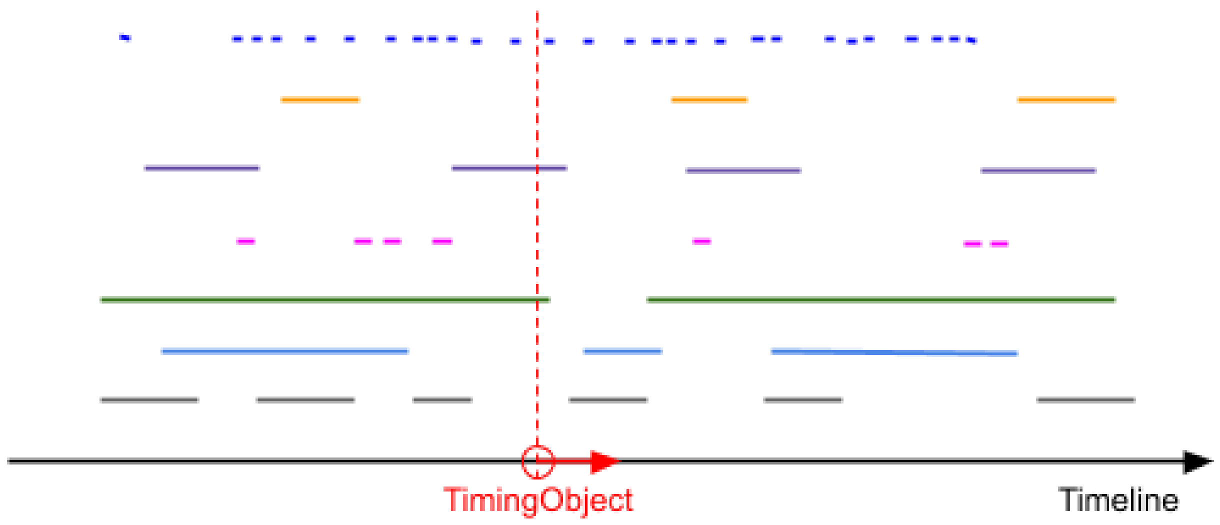
```
        // Set up video sync
        const sync2 = MCorp.mediaSync(document.getElementById('player2'), to);

    </script>
</head>
<body>
    <p>
        <div>
            Position: <span id="position" style="color:red;"></span>
        </div>
        <div class="ctrl-btn">
            <button id="reset">Reset</button>
            <button id="play">Play</button>
            <button id="pause">Pause</button>
            <button id="reverse">Reverse</button>
        </div>
    </p>
    <p>
        <input type="range" min="0" max="100" value="0" id="progress" class="progress
↪">
    </p>
    <p>
        <video id="player1" style="width:49%" autoplay>
            <source src="https://mcorp.no/res/bigbuckbunny.webm" type="video/webm" />
            <source src="https://mcorp.no/res/bigbuckbunny.m4v" type="video/mp4" />
        </video>
        <video id="player2" style="width:49%" autoplay>
            <source src="https://mcorp.no/res/bigbuckbunny.webm" type="video/webm" />
            <source src="https://mcorp.no/res/bigbuckbunny.m4v" type="video/mp4" />
        </video>
    </p>
</body>
</html>
```

CHAPTER 11

Demo Sequencer Point Mode

Sequencing timed data using a single *Timing Object* (see *Point Mode*).

- Data elements get activated (red) as the timingobject comes with their intervals *(start, end)*.
- The set of active data elements is visualized just below the position.
- Skip to a different position by clicking the timeline progress.
- Remove data elements at any time by clicking the appropriate X button.



Demo

demofile

11.1 Code

```
<!DOCTYPE html>
<html>
    <head>

        <style type="text/css">
            .ctrl-label {
                display:inline-block;
                width:100px;
            }
            .ctrl-btn button {
                width:80px;
            }
            .progress {
                width: 100%; /* Full-width */
                appearance: none;
                border-radius: 5px;
                height: 5px;
                background: #d3d3d3; /* Grey background */
                outline: none;
            }
            .active {color:red}
        </style>

        <script type="module">
            import {
                TimingObject, Dataset, Sequencer, Interval,
                TimingProgress, TimingSampler, DatasetViewer
            } from "https://webtiming.github.io/timingsrc/lib/timingsrc-esm-v3.js";

            /*
                Create TimingObject, Dataset and Sequencer
            */
            const to = new TimingObject({range:[0,30]});
            const ds = new Dataset();
            const activeCues = new Sequencer(ds, to);

            /*
                Visualize Timing Object Position
            */

            // refresh position every 100 ms
            const sampler = new TimingSampler(to, {period:100});

            // position
            const pos_elem = document.getElementById("position");
            sampler.on("change", function() {
                pos_elem.innerHTML = `${to.pos.toFixed(2)}`;
            });

            // progress
            const progress_elem = document.getElementById("progress");
            const progress = new TimingProgress(to,
                                                    progress_elem,
↪{sampler:sampler});
```

(continues on next page)

(continued from previous page)

```
            /*
                Connect buttons
            */
            document.getElementById("play").onclick = function () {
                to.update({velocity:1});
            };
            document.getElementById("pause").onclick = function () {
                to.update({velocity:0});
            };
            document.getElementById("reverse").onclick = function () {
                to.update({velocity:-1});
            };
            document.getElementById("reset").onclick = function () {
                to.update({position:0, velocity:0});
            };

            /*
                Mockup Timed Data
            */
            const data = [
                {id:"a", text: 'A', start: 0, end: 1 },
                {id:"b", text: 'B', start: 2, end: 3 },
                {id:"c", text: 'C', start: 4, end: 5 },
                {id:"d", text: 'D', start: 6, end: 7 },
                {id:"e", text: 'E', start: 8, end: 9 },
                {id:"f", text: 'F', start: 10, end: 11 },
                {id:"g", text: 'G', start: 12, end: 13 },
                {id:"h", text: 'H', start: 14, end: 15 },
                {id:"i", text: 'I', start: 16, end: 17 },
                {id:"j", text: 'J', start: 18, end: 19 },
                {id:"k", text: 'K', start: 20, end: 21 },
                {id:"l", text: 'L', start: 22, end: 23 },
                {id:"m", text: 'M', start: 24, end: 25 },
                {id:"n", text: 'N', start: 26, end: 27 },
                {id:"o", text: 'O', start: 28, end: 29 }
            ];

            /*
                Load timed cues into dataset
            */
            const cues = data.map(item => {
                return {
                    key: item.id,
                    interval: new Interval(item.start, item.end),
                    data: item
                };
            });
            ds.update(cues);

            /*
                Visualize cues in dataset
            */

            class CuesViewer extends DatasetViewer {

                constructor(ds, activeCues, elem) {
                    super(ds, elem);
```

(continues on next page)

(continued from previous page)

```
                    this._activeCues = activeCues;

                    // listen for click events on root element
                    elem.addEventListener("click", e => {
                        // find cue key from div wrapping button
                        let key = e.path[1].id;
                        e.stopPropagation();
                        ds.removeCue(key);
                    })
                }

                cue2string(cue) {
                    let key = cue.key;
                    let text = JSON.stringify(cue.data);
                    if (this._activeCues.has(cue.key)) {
                        return `
                            <div id=${key} class="active">
                                <button>X</button>
                                <span>${text}</span>
                            </div>`;
                    } else {
                        return `
                            <div id=${key}>
                                <button>X</button>
                                <span>${text}</span>
                            </div>`;
                    }
                }
            }
            let cues_elem = document.getElementById("cues");
            let cues_viewer = new CuesViewer(ds, activeCues, cues_elem);

            /*
                Visualize active cues
            */
            let active_elem = document.getElementById("active");
            activeCues.on("change", (eArg, eInfo) => {
                let el = document.getElementById(eArg.key);
                if (el) {
                    el.classList.add("active");
                }
                active_elem.innerText = `${eArg.new.data.text}`;
            });
            activeCues.on("remove", (eArg, eInfo) => {
                let el = document.getElementById(eArg.key);
                if (el) {
                    el.classList.remove("active");
                }
                active_elem.innerText = "";
            });

        </script>

    </head>
    <body>
        <p>
            <div>
```

(continues on next page)

(continued from previous page)

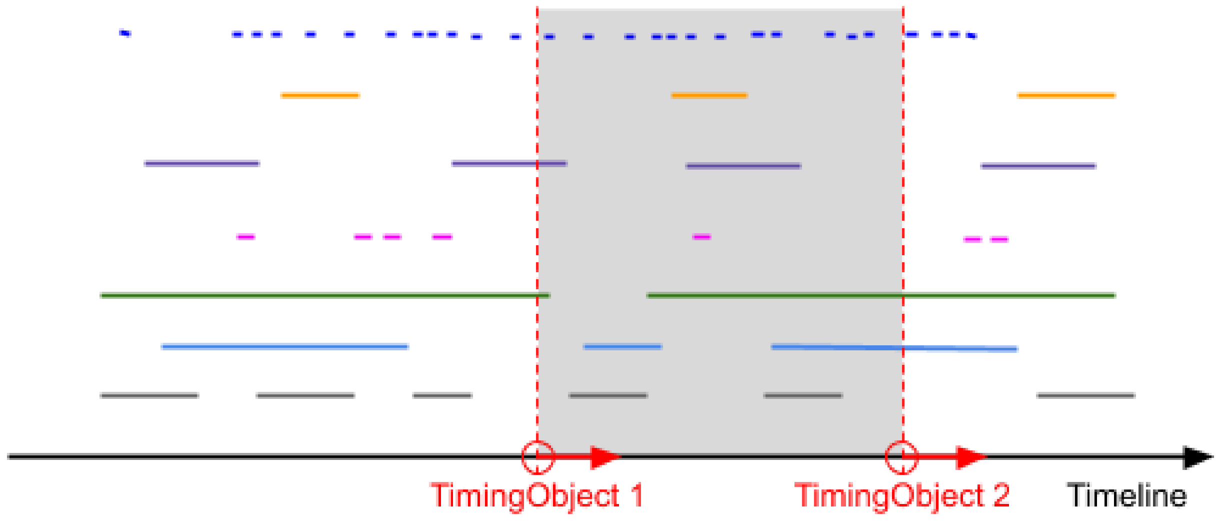
```
                <span class="ctrl-label">Position:</span>
                <span class="active" id="position"></span>
            </div>
            <div>
                <span class="ctrl-label">Active:</span>
                <span class="active" id="active"></span>
            </div>
        </p>
        <p>
            <div class="ctrl-btn">
                <button id="reset">Reset</button>
                <button id="play">Play</button>
                <button id="pause">Pause</button>
                <button id="reverse">Reverse</button>
            </div>
        </p>
        <p>
            <input type="range" min="0" max="100" value="0" id="progress" class=
↪"progress">
        </p>
        <p>
          <div id="cues"></div>
        </p>
    </body>
</html>
```

CHAPTER 12

Demo Sequencer Interval Mode

Sequencing timed data using two *Timing Object* (see *Interval Mode*).

- Data elements get activated (red) as their intervals *(start, end)* are overlapping with the interval between the two timing objects.
- The set of active data elements is visualized just below the position.
- Skip to a different position by clicking the timeline progress.
- Remove data elements at any time by clicking the appropriate X button.



Demo

demofile

12.1 Code

```
<!DOCTYPE html>
<html>
    <head>

        <style type="text/css">
            .ctrl-label {
                display:inline-block;
                width:100px;
            }
            .ctrl-btn button {
                width:80px;
            }
            .progress {
                width: 100%; /* Full-width */
                appearance: none;
                border-radius: 5px;
                height: 5px;
                background: #d3d3d3; /* Grey background */
                outline: none;
            }
            .active {color:red}
            .active-cues {
                color:red;
            }

            #activecues div {
                display: inline-block;
            }
        </style>

        <script type="module">
            import {
                TimingObject, Dataset, Sequencer, Interval,
                TimingProgress, TimingSampler, SkewConverter,
                DatasetViewer
            } from "https://webtiming.github.io/timingsrc/lib/timingsrc-esm-v3.js";

            /*
                Create TimingObject, Dataset and Sequencer
            */
            const to = new TimingObject({range:[0, 30]});
            const to2 = new SkewConverter(to, 4);
            const ds = new Dataset();
            const activeCues = new Sequencer(ds, to, to2);

            /*
                Visualize Timing Object Position
            */

            // refresh position every 100 ms
            const sampler = new TimingSampler(to, {period:100});

            // position
            const pos_elem = document.getElementById("position");
            const pos_elem2 = document.getElementById("position2");
```

(continues on next page)

(continued from previous page)

```
            sampler.on("change", function() {
                pos_elem.innerHTML = `${to.pos.toFixed(2)}`;
                pos_elem2.innerHTML = `${to2.pos.toFixed(2)}`;
            });

            // progress
            const progress_elem = document.getElementById("progress");
            const progress2_elem = document.getElementById("progress2");
            const progress_options = {sampler:sampler, range:[0,34]};
            const progress = new TimingProgress(to,
                                                progress_elem,
                                                progress_options);

            const progress2 = new TimingProgress(to2,
                                                 progress2_elem,
                                                 progress_options);


            /*
                Connect buttons
            */
            document.getElementById("play").onclick = function () {
                to.update({velocity:1});
            };
            document.getElementById("pause").onclick = function () {
                to.update({velocity:0});
            };
            document.getElementById("reverse").onclick = function () {
                to.update({velocity:-1});
            };
            document.getElementById("reset").onclick = function () {
                to.update({position:0, velocity:0});
            };

            /*
                Mockup Timed Data
            */
            const data = [
                {id:"a", text: 'A', start: 0, end: 1 },
                {id:"b", text: 'B', start: 2, end: 3 },
                {id:"c", text: 'C', start: 4, end: 5 },
                {id:"d", text: 'D', start: 6, end: 7 },
                {id:"e", text: 'E', start: 8, end: 9 },
                {id:"f", text: 'F', start: 10, end: 11 },
                {id:"g", text: 'G', start: 12, end: 13 },
                {id:"h", text: 'H', start: 14, end: 15 },
                {id:"i", text: 'I', start: 16, end: 17 },
                {id:"j", text: 'J', start: 18, end: 19 },
                {id:"k", text: 'K', start: 20, end: 21 },
                {id:"l", text: 'L', start: 22, end: 23 },
                {id:"m", text: 'M', start: 24, end: 25 },
                {id:"n", text: 'N', start: 26, end: 27 },
                {id:"o", text: 'O', start: 28, end: 29 }
            ];

            /*
                Load timed cues into dataset
```

(continues on next page)

(continued from previous page)

```
            */
            const cues = data.map(item => {
                return {
                    key: item.id,
                    interval: new Interval(item.start, item.end),
                    data: item
                };
            });
            ds.update(cues);

            /*
                Visualize cues in dataset
            */

            class CuesViewer extends DatasetViewer {

                constructor(ds, activeCues, elem) {
                    super(ds, elem);
                    this._activeCues = activeCues;

                    // listen for click events on root element
                    elem.addEventListener("click", e => {
                        // find cue key from div wrapping button
                        let key = e.path[1].id;
                        e.stopPropagation();
                        ds.removeCue(key);
                    })
                }

                cue2string(cue) {
                    let key = cue.key;
                    let text = JSON.stringify(cue.data);
                    if (this._activeCues.has(cue.key)) {
                        return `
                            <div id=${key} class="active">
                                <button>X</button>
                                <span>${text}</span>
                            </div>`;
                    } else {
                        return `
                            <div id=${key}>
                                <button>X</button>
                                <span>${text}</span>
                            </div>`;
                    }
                }
            }
            let cues_elem = document.getElementById("cues");
            let cues_viewer = new CuesViewer(ds, activeCues, cues_elem);


            /*
                Visualize active cues in dataset
            */
            activeCues.on("change", (eArg, eInfo) => {
                let el = document.getElementById(eArg.key);
                if (el) {
```

(continues on next page)

(continued from previous page)

```
                    el.classList.add("active");
                }
            });

            activeCues.on("remove", (eArg, eInfo) => {
                let el = document.getElementById(eArg.key);
                if (el) {
                    el.classList.remove("active");
                }
            });

            /*
                Visualize list of active cues
            */
            class ActiveCuesViewer extends DatasetViewer {
                cue2string(cue) {
                    return `${cue.data.text}`;
                }
            }
            let active_cues_elem = document.getElementById("activecues");
            let active_cues_viewer = new ActiveCuesViewer(activeCues, active_cues_
↪elem);

        </script>

    </head>
    <body>
        <p>
            <div>
                <span class="ctrl-label">Position 1:</span>
                <span class="active" id="position"></span>
            </div>
            <div>
                <span class="ctrl-label">Position 2:</span>
                <span class="active" id="position2"></span>
            </div>
            <div>
                <span class="ctrl-label">Active:</span>
                <span class="active-cues" id="activecues"></span>
            </div>
        </p>
        <p>
            <div class="ctrl-btn">
                <button id="reset">Reset</button>
                <button id="play">Play</button>
                <button id="pause">Pause</button>
                <button id="reverse">Reverse</button>
            </div>
        </p>
        <p>
            <input type="range" min="0" max="100" value="0" id="progress" class=
↪"progress">
            <input type="range" min="0" max="100" value="0" id="progress2" class=
↪"progress">
        </p>
        <p>
          <div id="cues"></div>
```

(continues on next page)

(continued from previous page)

```
        </p>
    </body>
</html>
```

CHAPTER 13

Timing Object

Contents

13.1 Introduction

```
<!DOCTYPE html>
<html>
    <head>
        <script type="module">
            import {
                TimingObject
            } from "https://webtiming.github.io/timingsrc/lib/timingsrc-module-v3.js";
            const to = new TimingObject({range:[0,10]});
        </script>
    </head>
    <body></body>
</html>
```

The timing object is a simple concept representing timeline state (e.g. media offset) and timeline controls (e.g. play/pause). Similar constructs can be found in most media frameworks, yet typically they are internal to each framework. The main purpose of the timing object is rather to provide a generic timeline construct to be used across media frameworks (see *Introduction*).

Demo

See *Demo TimingObject*

As illustrated by the demo, the timing object is similar to an advanced stop watch. If started with a velocity, its position changes predictably in time, until at some point later, it is paused, or perhaps the position is reset. It may be queried for its current position at any time. For example, it should take exactly 2.0 seconds for the position to advance from 3.0 to 5.0, if the velocity is 1.0. The timing object supports discrete jumps on the timeline, which may be useful for controlling slide shows or playlists. Velocity is useful for the control of any linear/timed media, including continuous media such as audio and video. Acceleration may not be commonly required, but it is there if you need it. Crucially, the timing object provides a *change* event, emmitted every time its behavior has been altered. This allows timing sensitive components to quickly detect changes and respond by correcting their behaviour accordingly.

A draft specification for the timing objects has been published with the W3C. The timing object concept was first published under the name Media State Vector.

13.2 Definition

Timing objects are logical clocks, defined by an internal clock and a vector.

internal clock The internal clock of a timing object always counts **seconds** since some shared time origin. In *timingsrc*, the internal clock is based on performance.now. The time origin of *performance.now* relates to the initialization of the Web page, so any timing object created within a single browsing context will all use the same internal clock. Note that this internal clock has no relation to any external clock. Note also that *performance.now* returns timestamps in **milliseconds**, so values are converted to **seconds** within the timing object implementation.

internal vector The internal vector describes the initial state of the *current movement* of the timing object; **(position, velocity, acceleration, timestamp)**. The vector timestamp is from the internal clock of the timing object. Future states of the timing object may be calculated precisely from the initial vector and elapsed time. Timing object behaviour may easily be modified by supplying a new initial vector.

```
let vector = {
    position: 12.0,             // position (units)
    velocity: 1.0,              // velocity (units/second)
    acceleration : 0.0,         // acceleration (units/second/second)
    timestamp : 7.234           // timestamp (seconds)
};
```

Timing objects may serve a variety of purposes within an application, so the **value** and **unit** of the timing object **position** is application specific. However, in the context of media applications **position** would typically be the duration since the beginning of some media session, in seconds.

query The query operation of the timing object is a cheap calculation useful for periodic sampling. It returns a fresh vector snapshot, calculated from the internal vector.

```
function query(internal_clock, internal_vector) {
    let pos = internal_vector.position;
    let vel = internal_vector.velocity;
    let acc = internal_vector.acceleration;
    let ts = internal_vector.timestamp;
    let now = internal_clock.now();
    let delta = now - ts;
    return {
```

(continues on next page)

(continued from previous page)

```
        position : pos + vel*delta + 0.5*acc*delta*delta,
        velocity : vel + acc*delta,
        acceleration : acc,
        timestamp : now
    };
}
```

update The update operation of the timing object accepts a vector specifying new values for position, velocity and acceleration, used to reset the internal vector of the timing object. If say **position** is omitted from the new vector, this means to preserve **position** as it was just before the update request was processed.

```
// play, resume
to.update({velocity:1.0});

// pause
to.update({velocity:0.0});

// jump to 10 and play from there
to.update({position:10.0, velocity:1.0})

// jump to 10, keep current velocity
to.update({position:10.0})
```

change event Whenever a timing object is updated, a **change** event is emitted from the timing object. The change event represents the start of a new movement. By subscribing to **change** events, media frameworks and components may monitor the timinig object and implement timely reactions to changes in timing object behavior.

```
// handle change event
to.on("change", () => {
    let v = to.vector;
    let moving = (v.velocity != 0.0 || v.acceleration != 0.0);
    if (moving) {
        console.log("moving!");
    } else {
        console.log("not moving!");
    }
});
```

timeupdate event For convencience, timing objects also provide an event for periodic sampling of the timing object. The **timeupdate** event is emitted at 5Hz (every 200 milliseconds) whenever the velocity (or acceleration) of the timing object is non-zero. So, if the timing object is paused, no events are emmitted util the timing object is unpaused.

```
// use timeupdate event to sample timing object position
to.on("timeupdate", function() {
    console.log(to.query().position);
});
```

Alternatively, if a different sampling frequency is required, a timing sampler may be used.

```
const sampler = new TimingSampler(to, {period:50});
sampler.on("change", function () {
    console.log(to.query().position);
});
```

rangechange event Event triggeres whenever the range is changed.

13.3 Programming with Timing Objects

Timing objects are resources used by a Web application, and the programmer may define as many as required. What purposes they serve in the application is up to the programmer. If the application needs a shared clock, simply starting a timing object (and never stopping it) might be sufficient. If the timing object position should be milliseconds, set the velocity to 1000 (advances the timing object position with 1000 milliseconds per second). If the timing object represents media offset, specify the playback position, the velocity, and perhaps a media duration (range). For videos where offset is measured in seconds or frames, set the velocity accordingly. Or, for certain musical applications it may be practical to let the timing object position represent beats, given a fixed BPM (beats per minute). Note also that the timing object may represent time-changes with any kind of floating-point variable. For instance, if data is organized according to height above sea level, it might be appropriate to animate how data changes during continuous vertical movement. In this case the timing object could represent meters or feet above sea level, and positive and negative velocities would allow you to move gradually both upwards and downwards.

CHAPTER 14

Timing Converter

Contents

14.1 Introduction

```
<!DOCTYPE html>
<html>
    <head>
        <script type="module">
            import {
                TimingObject,
                SkewConverter
            } from "https://webtiming.github.io/timingsrc/lib/timingsrc-module-v3.js";
            const to = new TimingObject({range:[0,10]});
            const c = new SkewConverter(to, 2);
```

(continues on next page)

(continued from previous page)

```
        </script>
    </head>
    <body></body>
</html>
```

Timing converters are useful when you need an alternative representation for a *Timing Object*. For example, co-presentation of different media sources might be a problem if said media sources refer to different timelines. Given that the relation between the timelines is known, this can be solved by either of the following approaches.

1) convert all timestamps of media sources to match the timeline of the timing object
2) convert a timing object to match the timeline of each media source

The second approach is often most attractive, as converting timestamps in media data may often be inconvenient, costrly or otherwise undesireable. Simply converting the media clock is typically a much easier solution.

Demo

See *Demo TimingConverter*

So, as the name suggests, timing converters **convert** timing objects, for instance by *skewing* or *scaling* the timeline of the original timing object. This may be useful in video playback, where the position of a timing object typically represents media offset in seconds. In this case, a timing converter could be used to create and alternative representation based on frame numbers, with playback velocity set to 24 or 25 frames per second (fps), depending on the media format. Or, for music it might be sensible to use beat number as position, and beats per second (bps) as velocity.

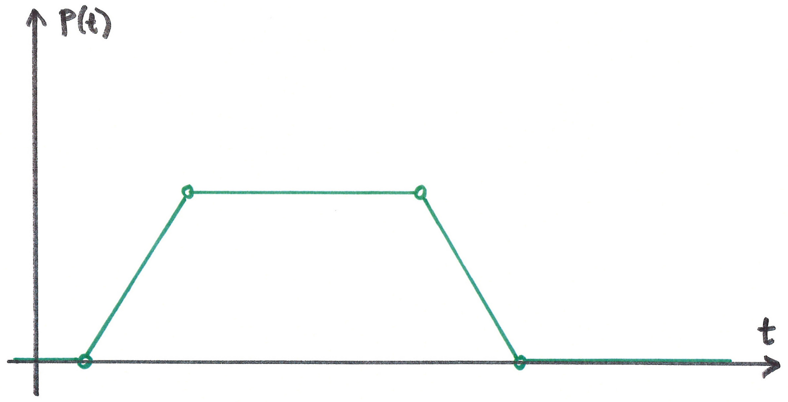
14.2 Definition

A *Timing Converter* provides an alternative representation for a *Timing Object*.

- a timing converter **is** a timing object, which also depends on a parent timing object.
- the *timingsrc* property of a Timing Converter identifies its parent timing object.
- a timing converter implements some modification relative to its *timingsrc*, but never modifies its *timingsrc* in any way.
- multiple timing converters may share *timingsrc*.
- a timing converter can itself be the *timingsrc* of another timing converter.
- a timing converter may forward update requests to the parent, converting the request into the parent timeline if necessary.

So, a chain or hierarchy of timing converters may be created, where all timing converters ultimately depend on a common timing object as root. Each timing converter typically provides a single modification. More complex modifications may for instance be created by combining multiple timing converters.

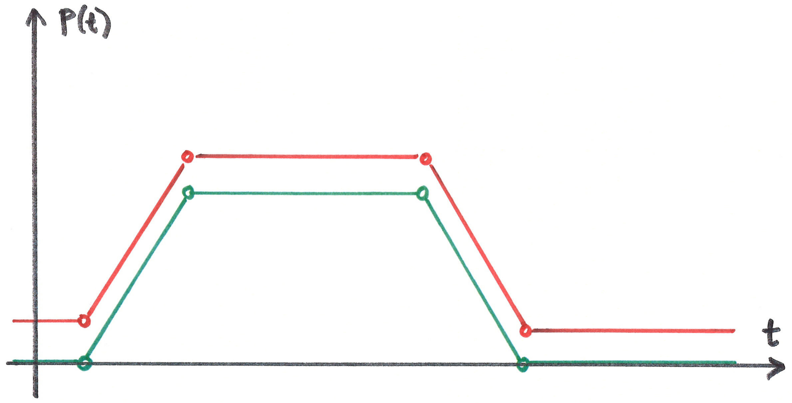
14.3 Position-time diagrams



Position-time diagrams are helpful for illustrating the behavior of timing objects and timing converters. In the above figure, the x-axis (horizontal) is time, and the y-axis (vertical) is the position of the timing object. The figure illustrates a sequence of 4 updates to the timing object, where each circle is denoted by a circle. Initially (time 0) the position of the timing object is 0.

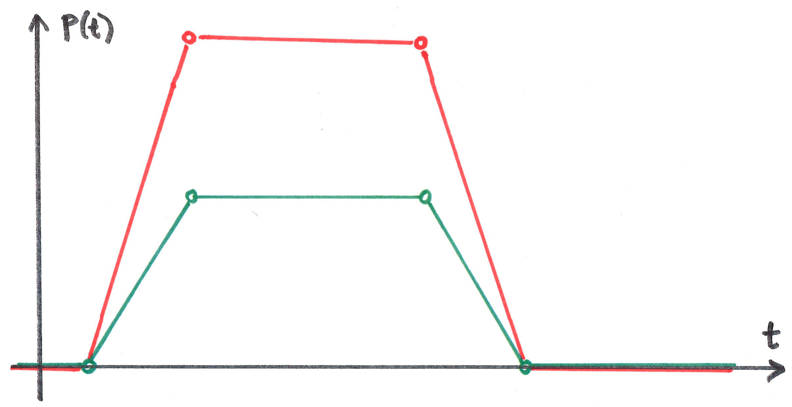
1) start the timing object (positive velocity). The position increases linearly.
2) pause the timing object. Position remains unchanged.
3) starts the timing object backwards (negative velocity). Position decreases linearly.
4) pause the timing object at the exact moment when position becomes 0. (This may for instance be enforced by the timing object itself, as a range restriction.)

14.4 Skew Converter



The effect of the skew converter is illustrated with red coloring. A positive skew is supplied, shifting all positions in the positive direction.

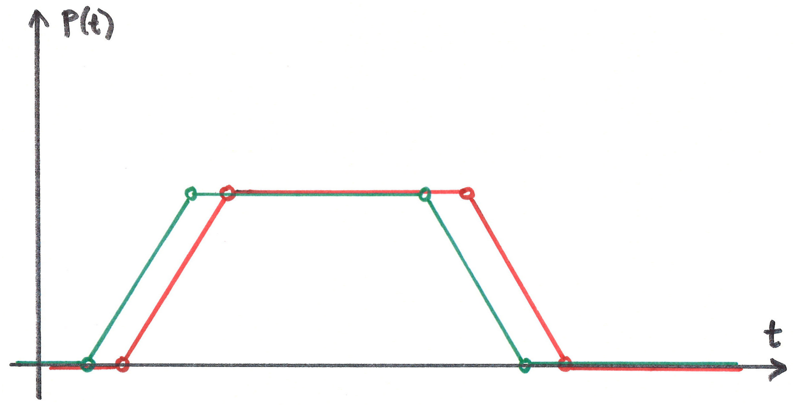
14.5 Scale Converter



Scaling the by a factor means that all values (position, velocity and acceleration) are multiplied by that factor.

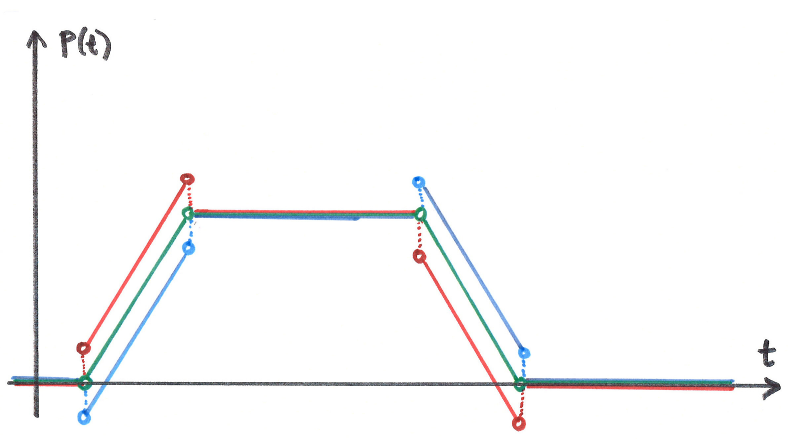
For example, a factor 1000 scales values in seconds to values in milliseconds. Velocity 1s/s becomes velocity 1000ms/s.

14.6 Delay Converter



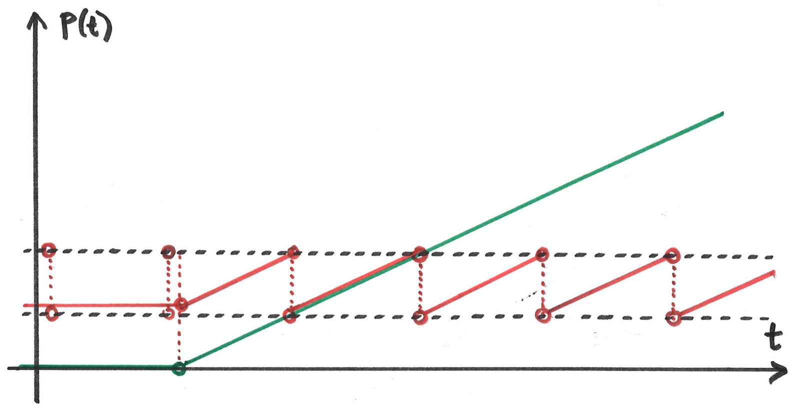
Delay converter re-plays the behaviour of the timing object, with a fixed delay. Update events are delayed too. Delay converters are read-only in the sence that they do not accept update requests.

14.7 Timeshift Converter



Timeshift converter timeshifts the behavior of the timing object. Red color is ahead in time (speculative). Blue color is after in time. When the position is static, time-shifting has no effect. The Timeshift converter does not timeshift update events.

14.8 Loop Converter



The two dotted black lines illustrate a range restriction for the loop converter. When the timing object is inside this range, the loop converter will be equal to the timing object. When the timing object is outside, its position is translated to a value within the range, i.e. modulo of range length.

CHAPTER 15

Timing Provider

Contents

15.1 Introduction

```
// assign timing provider to timingsrc property of timing object
to.timingsrc = timing_provider;
```

A *Timing Provider* is a proxy object for a **remote timing resource**. Remote timing resources exist *outside* the Web page (i.e. browsing context) of the timing object. The remote timing resource might live in another process on the same computer or on a remote server. By acting as proxy, timingprovider objects allow a *Timing Object* to be *connected* to a remote timing resource. If the remote timing resource is hosted online, this opens up for consistent media control in the global scope.

Demo

See *Demo TimingProvider*

15.2 External timing

The timingsrc programming model is based on the idea that consistency and shared media control is achieved by shared access to timing objects. This idea is the focus of the *Introduction*, where timing objects are proposed for consistency across media frameworks within a single Web page. Importantly, with online timing resources, this idea can be extended to globally shared media experiences, without imposing additional complexity on the developers, see Media Synchronization on the Web. As such, timing providers extend the scope of the **timingsrc** programming model from local to global.

15.3 Custom Timing Providers

It might be possible to derive a standardized protocol for communication with remote timing services. However, in the interest of future innovation this approach was not recommended by the Timing Object Draft Specification. Instead, the proposal defines an API for timing provider proxy objects, opening up for custom implementations of timing services and assiciated timing providers. This decoupling between specific timing services and application code maximizes flexibilty, with the *Timing Object* as an in-between mediator.

15.4 Timing Provider Functionality

A timing provider (TP) runs client-side and exchanges messages between a client-side timing object (TC) and a remote timing service (TS). In particular, TP will forward update request vectors from the timing object TS to the remote timing service TS, and receive asynchronous notifications of vector updates in the opposite direction.

The overall goal is that timing objects connected to the same remote timing resources are always equal. If queried at the same time, all timing objects should yield the same results in terms of position, velocity and acceleration.

This though is not straight forward, since the clocks used by the remote timing service and timing objects (see *Timing Object*) are generally not the same. To solve this, clock timestamps must be converted back and forth via a shared clock, typically the clock of the remote timing resource. To do this conversion, the **skew** between the clocks must be estimated:

Important: TS_CLOCK = TC_CLOCK + SKEW

The idea is that timing providers implement skew estimation, which can then be used by the timing object to do the necessary conversions. This way, implementing a timing provider proxy object is reasonably easy.

1) Provide an estimate for SKEW. P must have access to TC_CLOCK. An estimate for TS_CLOCK must be obtained by live messaging with TS. Likely, the SKEW estimate should be updated periodically to account for clock drift or adjustments made to system clocks (e.g. NTP clock synchronization).
2) Forward request update vector from TC to TS, unchanged.
3) Forward notification update vector from TS to TC, unchanged.

Note: Direct forwarding of update notification in 3) implies that there is no mechanism for ensuring that vector updates are applied at exactly the same time. Importantly though, updates will eventually have the same effect even if they are not applied at the same time. Inconsistencies are limited to the brief duration when one timing object has received an update while another has not. This is rarely noticed in practice.

15.5 Shared Motion Timing Provider

Shared Motion is provided by Motion Corporation through **InMotion**, a generic, online timing service for IP-connected clients and Web agents. *Shared Motion* by Motion Corporation can be used directly with the *Timing Object*. To test this please follow these simple steps:

15.5.1 1. Create MCorp App

- goto https://dev.mcorp.no
- create MCorp App
- **MOTION_NAME**: create a named motion inside your app
- **APPID**: copy the APPID from your MCorp App

15.5.2 2. Connect Timing Object to Shared Motion

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript" src="http://www.mcorp.no/lib/mcorp-2.0.js"></
↪script>
        <script type="module">
            import {
                TimingObject
            } from "https://webtiming.github.io/timingsrc/lib/timingsrc-module-v3.js";
            const to = new TimingObject();
            const app = MCorp.app("APPID", {anon:true});
            app.ready.then(function() {
                to.timingsrc = app.motions["MOTION_NAME"];
            });
        </script>
    </head>
    <body>
    </body>
</html>
```

Documentation for MCorp App initialization at https://dev.mcorp.no

CHAPTER 16

MediaSync

Contents

16.1 Introduction

```
<!DOCTYPE html>
<html>
    <head>
        <script src="https://mcorp.no/lib/mediasync.js"></script>
        <script type="module">
            import {
                TimingObject
            } from "https://webtiming.github.io/timingsrc/lib/timingsrc-esm-v3.js";
            const to = new TimingObject({range:[0,100]});
            const sync = MCorp.mediaSync(document.getElementById('player'), to);
        </script>
    </head>
    <body>
        <video id="player" autoplay></video>
    </body>
</html>
```

MediaSync is JavaScript wrapper for HTML5 media elements, allowing precisly timed playback and control for audio and video on the Web, using the *Timing Object*. This is achived by adjusting the offset of the media element so that it always matches the the timing object. The wrapper code periodically compares *currentTime* property of the media element with the *position* of the timing object. If the difference grows too large, larger adjustments are implemented by *seekTo* operations whereas more gradual corrections are achived by modifications to the playbackrate.

Demo

See *Demo MediaSync*

MediaSync is a common purpose library. It is not optimised for any particular combination of OS, media codecs or browser implementation. Despite this, and despite a number weaknesses in HTML5 media elements with respect to precisely timed playback, *MediaSync* demonstrates the feasibility of echoless synchronization across the Internet. See for instance this demonstration on YouTube. A technical report evaluating synchronization of HTML5 media elements is available here.

The MediaSync JavaScript library is maintained by Motion Corporation and may be downloaded from their site: https://mcorp.no/lib/mediasync.js.

16.2 Reservations

Support for precise synchronization of HTML5 media is **experimental** and subject to certain reservations.

1) Codecs and format issues are notorious for audio and video on the Web, and certain options/combinations may hurt the ability for precise synchronization.
2) Synchronization of live media streams is possible, but depends on the session timeline being correctly tied to the media content timeline. In particular, if the media player starts from *currentTime* 0 whenever the viewer session starts, session timeline and content timeline are *independent*. If so, synchronization is not possible, unless the relation between the two timelines may be derived by other means.
3) Repeated buffering due to limited data access is not a great starting point for precise synchronization.
4) Media capabilities vary accross platforms, and platform specific media issues may impact the support for precise synchronization. For instance, IOS devices have issues with the implementation of variable playbackrate.
5) Stricter autoplay policies in Web browsers may require user involvement becore any playback can start.
6) Capability for precisely time controlled playback is currently not required (or even recommended) by W3C standards. For this reason, synchronization performance is not evaluated and new browser versions may therefore include sudden changes that affect synchronization (for better or worse).
7) Timed media playback requires a short initializion phase before precise and stable playback can be achieved. This is expected to take 0-3 seconds, and allows the media element to load data and home in on the correct playback offset. If precicely timed playback is needed from the very start, one trick could be to pad the media content with a preamble, allowing synchronization to be started a little earlier. In this case it might be attractive to hide the media element until the original starting point is reached.

CHAPTER 17

Interval

Contents

17.1 Introduction

Interval is used by *Dataset* and *Sequencer* to define the validity of objects or values in relation to a timeline. Intervals describe either a *continuous line segment* or a *singular point*. In the context of media, intervals define the *temporal validity* of timed media content.

17.2 Definition

Following standard mathematical notation intervals are expressed by two endpoint values **low** and **high**, where **low <= high**. Interval endpoints are either **open** or **closed**, as indicated with brackets below:

> e.g.: **[a,b] [a,b) (a,b] (a,b)**

If **low == high** the interval is said to represent a *singular* point **[low, high]**, or simply **[low]** for short. Endpoints of singular point intervals are always closed.

Infinity values may be used to create un-bounded intervals. Endpoints with infinite values are always closed.

e.g.: **[a, Infinity] [-Infinity, a] [-Infinity, Infinity]**.

```
// [4.0] - singular point
itv = new Interval(4.0);

// [4.0] - singular point
itv = new Interval(4.0, 4.0);

// [4,6.1) - interval
itv = new Interval(4.0, 6.1, true, false);

// (4,6.1) - interval
itv = new Interval(4.0, 6.1, false, false);

// [4,6.1] - interval
itv = new Interval(4.0, 6.1, true, true);

// (4,6.1] - interval
itv = new Interval(4.0, 6.1, false, true);

// [4,6.1) - default endpoints
itv = new Interval(4.0, 6.1);

// [4,->] - un-bounded
itv = new Interval(4.0, Infinity);
```

Note: Knowing how to create intervals is likely sufficient for basic usage of *Dataset* and *Sequencer*. The rest of this section provides a reference for advanced usage and details concerning ordering and comparison of intervals on a timeline.

17.3 Endpoint Types

Intervals are defined as a pair of interval endpoints. The table below shows that there are four distinct types of endpoints, and that endpoints have three distinct properties

- **value**: numerical value
- **bracket-side**: true if high else low
- **bracket-type**: true if closed else open

symbol	name	value	bracket-side	bracket-type
[a	low-closed	a	false	true
(a	low-open	a	false	false
a]	high-closed	a	true	true
a)	high-open	a	true	false

Singular intervals have two endpoints **[a** and **a]**, even though they only have one value. In order to distinguish endpoints of a singular interval, boolean flag **singular** is added to the representation.

Endpoints are therefor represented by a four-tuple

[value, bracket-side, bracket-type, singular].

17.4 Endpoint Ordering

Correct ordering of points and endpoints is important for consistency of media state, media navigation and playback. Ordering is straight forward as long as endpoint values are different in value. For instance, *2.2]* is ordered before *(3.1* because *2.2 < 3.1*. However, in case of equality, sensitivity to properties **bracket-side**, **bracket-type** and **singular** is required to avoid ambiguities.

The internal ordering of point **p** and the four endpoint types with value **p** is, from left to right:

> **p), [p, p, p], (p**

Or, by name:

> *high-open, low-closed, value, high-closed, low-open*

Endpoints of singular intervals are orders as regular values.

Based on this ordering we may define the comparison operators **lt(e1, e2)** and **gt(e1, e2)**, where **e1** and **e2** are either endpoints or regular points values.

> **lt(e1, e2)** returns true if **e1** is before **e2**, and false if **e1** is equal to or after **e2**.
>
> **gt(e1, e2)** returns true if **e1** is after **e2**, and false if **e1** is equal to or before **e2**.

17.5 Interval Comparison

Intervals may overlap partly, fully, or not at all. More formally, we define interval comparison in terms of interval relations:

> The operator **cmp(a, b)** compares interval **a** to interval **b**. The comparison yields one of seven possible relasions: OUTSIDE_LEFT, OVERLAP_LEFT, COVERED, EQUAL, COVERS, OVERLAP_RIGHT, or OUTSIDE_RIGHT.

The **cmp(a,b)** operator is then defined in terms of simpler operators **lt**, **gt** and **inside**. The operator **inside(e, i)** evaluates to true if a point or an endpoint **e** is inside interval **i**. Interval **i** is in turn defined by its two endpoints **i.low** and **i.high**.

> **inside(e, i) = !lt(e, i.low) && !gt(e, i.high)**

Interval relations OUTSIDE_LEFT, OVERLAP_LEFT, COVERED, EQUAL, COVERS, OVERLAP_RIGHT and OUTSIDE_RIGHT are defined as follows:

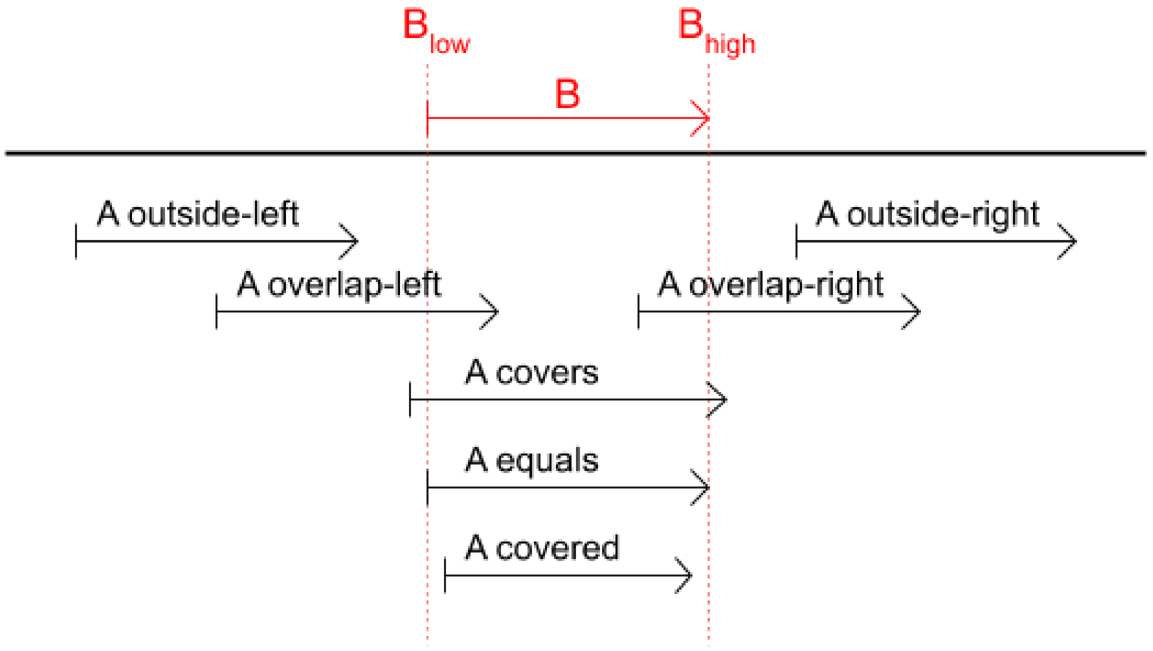


Fig. 1: This illustrates the different interval relations yielded by **cmp(a,b)** when seven diffent intervals A are compared to the same interval B.

cmp(a, b)	**description**	**definition**
OUTSIDE LEFT	a is outside b on the left	• a.high *lt* b.low
OVERLAP LEFT	a overlaps b from left	• a.high is *inside* b • a.low is *gt* b.low • a.high is *lt* b.high
COVERED	a is covered by b	• a.low *inside* b && a.high *inside* b • b.low *!inside* a || b.high *!inside* a
EQUAL	a is equal to a	• a.low *inside* b && a.high *inside* b • b.low *inside* a && b.high *inside* a
COVERS	a covers b	• a.low *!inside* b || a.high *!inside* b • b.low *inside* a && b.high *inside* a
OVERLAP RIGHT	a overlaps b from right	• a.low is *inside* b • a.low is *gt* b.low • a.high is *gt* b.high
OUTSIDE RIGHT	a is outside b on the right	• a.low *gt* b.high

Here are a few examples of comparisons between intervals a and b.

a	b	cmp(a, b)
[2,4>	[4]	OUTSIDE_LEFT: a is outside b on the left
[2,4>	<2,4]	OVERLAP_LEFT: a overlaps b from left
[2,4>	[2,4]	COVERED: a is covered by b
[2,4>	[2,4>	EQUAL: a is equal to b
[2,4>	<2,4>	COVERS: a covers b
[2,4>	<1,3>	OVERLAP_RIGHT: a overlaps b from right
[2,4>	<1,2>	OUTSIDE_RIGHT: a is outside b on the right

17.6 Interval Match

The operation **match(a, b, mask)** returns true if interval a *matches* interval b. **mask** defines what interval relations are accepted as a *match*. Each interval relation is associated with a mask value. Multiple relations may then be be aggregated (AND'ed) into the appropriate mask.

mask	int	relation
0b1000000	64	OUTSIDE_LEFT
0b0100000	32	OVERLAP_LEFT
0b0010000	16	COVERED
0b0001000	8	EQUALS
0b0000100	4	COVERS
0b0000010	2	OVERLAP_RIGHT
0b0000001	1	OUTSIDE_RIGHT

The *default* value of match **mask** is 62 (0b0111110), which implies that all relations except OUTSIDE_LEFT and OUTSIDE_RIGHT are counted as a match.

CHAPTER 18

Cue

A **Cue** is a triplet **(key, interval, data)** represented by a simple **Javascript** object.

```
let cue = {
    key: "mykey",
    interval: new Interval(2.2, 4.31),
    data: {...}
};
```

key Unique key. Any value or object that may be used as a key with `Map`. The purpose of cue *key* is to uniquely identify a cue object within a collection of cue objects.

interval Defines the validity of the cue in reference to a numerical dimension, typically a timeline. Intervals represent a contiguous segment on the timeline or singular points (see *Interval*).

data The **data** property is an externally defined value or object associated with the cue. Typically, cue properties **key** and **interval** are derived from values within the cue **data** object (see *Cue Creation*).

18.1 Cue Creation

Cues are typically created by wrapping **application-defined data objects**. These objects often include properties which define object uniqueness, within some application specific namespace. Property names such as **id**, **key** and **uuid** are often used for this purpose. If so, such object identifiers may be used as cue keys.

Additionally, application objects may define timestamps, durations or other numerical values indicating the validity of the object in reference to a timeline. Property names such as **ts**, **start**, **end** and **duration** are often used for this purpose. If so, cue interval objects may be created from these values.

```
// application object
let subtitle = {
    id: 1234,
    text: "This is some text",
    start: 24.3,
```

(continues on next page)

(continued from previous page)

```
    end: 28.7
};

// cue from application object
let cue = {
    key: subtitle.id,
    interval: new Interval(subtitle.start, subtitle.end);
    data: subtitle
};
```

CHAPTER 19

Cue Collection

Contents

19.1 Introduction

Cue Collection specifies an interface for a collection of (key, cue) pairs, extended with events.

Cue collection emulates the Javascript `Map` API by defining methods **has()**, **get()**, **keys()**, **values()** and **entries()**. Also, the number of (key,cue) pairs managed by the cue collection is exposed by the property **size**.

Cue collection also defines three events: **batch**, **change** and **remove**. Events allow *modifications* of the cue collection to be detected by subscribers. Events are implemented as defined in *Events*.

Note: The *Cue Collection* interface is only **read-only** and does *not* specify any mechanisms for modifying the collection of (key, value) pairs. Methods for modification are left for specific implementations of the interface, such as *Dataset* and *Sequencer*.

19.2 Events

19.2.1 Modification Types

Cue collection supports two types of modifications: *membership-modifications* and *cue-modifications*:

membership-modifications Cues *inserted* into or *deleted* from the cue collection.

cue-modifications *Modification* of cues in cue collection.

19.2.2 Change and Remove

Cue collection defines events **change** and **remove**. Each event report modifications concerning an single (key, cue) pair.

change

- *inserted* cues (*membership-modification*)
- *modified cues* (*cue-modification*)

remove

- *deleted cues* (*membership-modification*)

```
// cue collection
let cc;

cc.on("change", function(eArg) {
    console.log("change")
});

cc.on("remove", function(eArg) {
    console.log("remove")
});
```

Note: It would also have been possible to expose three events (**insert**, **modify**, **delete**) instead of two events (**change**, **remove**). However, the latter is often more convenient, as **insert** and **modify** events are frequently handled the same way. On the other hand, if the distincion matters the event argument of the *change* event may be used to tell them apart. See *Event Argument*.

19.2.3 Batch Event

Cue collection additionally defines a **batch** event which delivers multiple **change** and **remove** together. This is relevant for implementations supporting modification of multiple cues in one (atomic) operation. If so, the **batch** event makes it possible to process *concurrent* events in one operation, and making decisions based on the whole batch, as opposed to single events.

The event argument **eArg** of the **batch** event is simply a list of event arguments for individual **change** and **remove** events.

```
// cue collection
let cc;

```

(continues on next page)

(continued from previous page)

```
cc.on("update", function (eArgList) {
    eArgList.forEach(function(eArg) {
        if (eArg.new != undefined) {
            if (eArg.old != undefined) {
                console.log("modify");
            } else {
                console.log("insert");
            }
        } else {
            if (eArg.old != undefined) {
                console.log("delete");
            } else {
                console.log("noop");
            }
        }
    });
});
```

Note: Cue collection may emit a **batch** event including event arguments where both **eArg.new** and **eArg.old** are undefined, i.e. **noop** events.

19.2.4 Event Argument

Cue collection events provide an event argument **eArg** describing the modification of a single cue. The event argument is a simple object with properties **key**, **new** and **old**:

```
// Event Argument
let eArg = {key: ..., new: {...}, old: {...}}
```

key key (unique in cue collection)

old cue *before* modification, or undefined if cue was inserted.

new cue *after* modification, or undefined if cue was deleted.

This table show values **eArg.old** and **eArg.new** may assume for different events and modification types.

modification	event	eArg.old	eArg.new
insert	change	undefined	cue
modify	change	cue	cue
delete	remove	cue	undefined
noop		undefined	undefined

Distinguishing between modification types is easy:

```
// cue collection
let cc;

cc.on("change", function(eArg) {
    if (eArg.old == undefined) {
        console.log("insert");
    } else {
```

(continues on next page)

(continued from previous page)

```
        console.log("modify");
    }
});

cc.on("remove", function(eArg) {
    console.log("delete")
});
```

19.3 Cue Ordering

By default cue collections do not enforce a particular ordering for its (key, cue) pairs. If needed, order may be specified on the constructor. The *cues()* method will then returne an ordered list of cues. In addition, cue events will be delivered in the correct order. Ordering options may also be supplied directly to the `CueCollection.cues()` and will take precedence over constructor options. This applies for both *Dataset* and *Sequencer*, which are both cue collactions.

```
// order by keys
function cmp(cue_a, cue_b) {
    return cue_a.key < cue_b.key;
}

let cc = new CueCollection({order:cmp})

// unordered iterator of cues
let cues_iterator = cc.values()

// ordered list of cues
let cues_list = cc.cues();
```

CHAPTER 20

Dataset

Contents

20.1 Introduction

Dataset manages a collection of cues, implements the *Cue Collection* and adds support for flexible and efficient cue modification and lookup, even for large volumes of cues. *Cues* are simple Javascript objects:

```
let cue = {
    key: "mykey",
    interval: new Interval(2.2, 4.31),
    data: {...}
}
```

Dataset maps *keys* to *cues*, like a `Map`. In addition, *cues* are also indexed by their positioning on the timeline (see *Interval*), allowing efficient search along the timeline. For instance, the **lookup** method returns all cues within a given lookup interval.

Dataset is useful for management and visualization of large datasets with timed data, represented as *cues*. Typical examples of timed data include log data, user comments, sensor measurements, subtitles, images, playlists, transcripts, gps coordinates etc.

Furthermore, the dataset is carefully designed to support *precisely timed playback* of timed data. This is achieved by connecting one or more *Sequencers* to the *Dataset*.

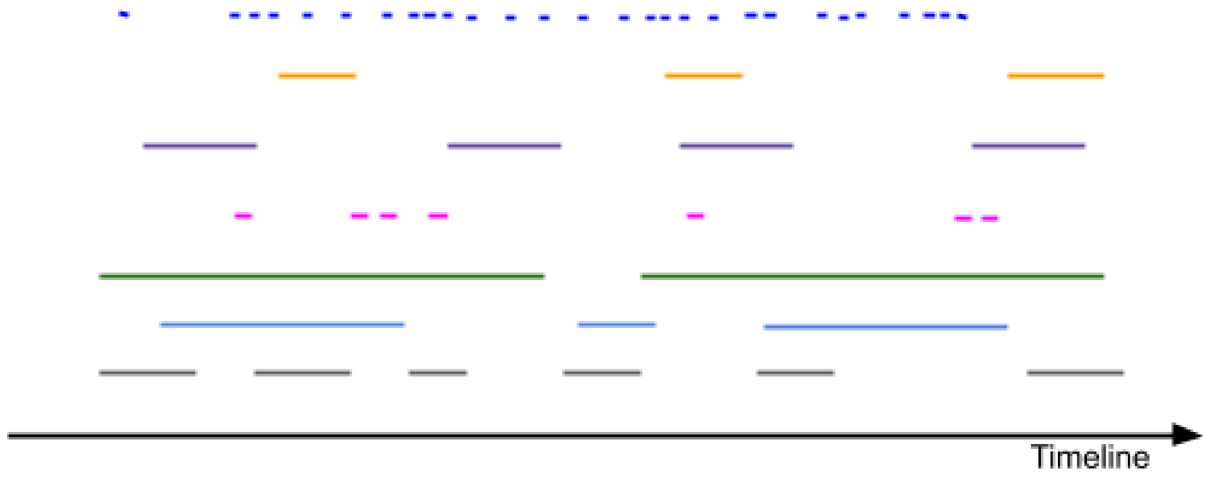


Fig. 1: This illustrates multiple tracks (different colors) of timed data. Each colored line segment is a cue, with horizontal placement and length indicating cue validity in reference to the timeline. Tracks may simply be different types of cues, e.g. comments, gps-coordinates, videos, images, audio snippets, etc. A single dataset may hold all kinds of cues, collectively defining the state of a media presentation, see *Linear Media State*.

20.2 Example

```
// create dataset
let ds = new Dataset();

// timed data
let subtitles = [
    {
        id: "1234",
        start: 123.70,
        end: 128.21,
        text: "This is a subtitle"
    },
    ...
];

// create cues from subtitles data
let cues = subtitles.map(function (sub) {
    let itv = new Interval(sub.start, sub.end);
    return {key: sub.id, interval: itv, data: sub};
});
```

(continues on next page)

(continued from previous page)

```
// insert cues
ds.update(cues);

// lookup cues
let result_cues = ds.lookup(new Interval(120, 130));

// delete cues
ds.update(cues.map(function(cue) {
    return {key: cue.key};
});
```

20.3 Update

Dataset provides a single operation **update(cues)** allowing cues to be **inserted**, **modified** and/or **deleted**. The argument **cues** defines a list of cue arguments (or a single cue argument) to be **inserted** into the dataset. If a cue with identical key already exists in the dataset, the *pre-existing* cue will be **modified** to match the provided cue argument. If a cue argument includes a key but no interval and no data, this means to **delete** the *pre-existing* cue.

```
let ds = new Dataset();

// insert
ds.update({
    key: "key1",
    interval: new Interval(2.2, 4.31),
    data: "foo"
});

// modify
ds.update({
    key: "key2",
    interval: new Interval(4.4, 6.9, false, false),
    data: "bar"
});

// delete
ds.update({key: "mykey"})
```

For convenience, intervals in cue arguments may also be specified as an array, leaving it to the dataset to create *Interval* objects for internal use. Also, **addCue** and **removeCue** methods provide shorthand access to **update**. For instance, the above code example may be rewritten as follows:

```
let ds = new Dataset();
ds.addCue("key1", [2.2, 4.31], "foo");
ds.addCue("key2", [4.4, 6.9, false, false], "bar");
```

See also *Update Convenience Methods* for more details.

20.3.1 Cue Management

When a cue is inserted into the dataset, it will be managed until it is deleted at some later point. All cue access operations (e.g. **cues**, **lookup**) provide direct access to managed cues.

Warning: Cues managed by dataset are considered **immutable** and must **never** be modified directly by application code. Always use the **update** operation to implement cue modification.

If managed cue objects are modified by external code, no guarantees can be given concerning functional correctness. By default, the dataset does not offer any protection against external cue modification. However, in safe mode *Object.freeze()* is applied to all cues, implying that attempted modification should throw an exception (strict mode). This is useful for evaluation but should likely be turned off in production, as use of *Object.freeze()* comes with a performance penalty.

```
let ds = new Dataset({safe:true})
```

Important:

- **always** create cue arguments as new objects with desired state
- **never** reuse managed cue objects as arguments to update

The dataset will throw an exception if a currently managed cue object is used as cue argument with the **update** operation.

Unwanted modifications of managed cues may also occur if the *cue.data* property is subject to external modification. *Object.freeze()* does not protect against this. For instance, the data object may a reference to an object which is managed by an application specific data model. If this is the case one approach would be to copy data objects as part of cue creation. Another approach is to add one level of indirection, adding only immutable object id's to the dataset. This though would imply that data changes can not be detected by the dataset.

20.3.2 Cue Arguments

Dataset also supports *partial* cue modification. *Partial* modification means to modify *only* the *cue interval property* or *only* the *cue data property*. For convenience, partial cue modification allows this to be done without restating the *unmodified* part of the cue. Partial cue modification is specified simply by omitting the property which is not to be replaced. The omitted property will then be preserved from the *pre-existing* cue. This yields four types of legal cue arguments for the **update** operation:

Type	Cue argument	Text
A	{key: "mykey"}	no interval, no data
B	{key: "mykey", interval: ... }	interval, no data
C	{key: "mykey", data: ... }	no interval, data
D	{key: "mykey", interval: ..., data: ... }	interval, data

Note: Note that `{key: "mykey"}` is *type A* whereas `{key: "mykey", data:undefined}` is type C. The type evaluation is based on `cue.hasOwnProperty("data")` rather than `cue.data === undefined`. This ensures that `undefined` may be used as a data value with cues.

Similarly, cue intervals may also take the value `undefined`. Without an interval cues become invisible to the **lookup** operation, yet still accessible through `Map` operations **has, get, keys, values, entries**. Otherwise, if cue interval is defined, it must be an instance of the `Interval` class.

Note: Cue intervals are often derived from timestamps which are also part of cue data. This implies that inconsistency may be introduced, if the interval is changed, without also changing the associated timestamps in the data property – or the other way around.

Though not criticial for the integrity of the dataset, such inconsistencies might be confusing for users. For instance if timeline playback does not match timestamps in cue data.

Rule of thumb:

- Avoid cue type B modification if timestamps are part of data.
- Similarly, avoid type C modification of timestamps in data, if cue intervals are derived from these timestamps.

In summary, the different types of cue arguments are interpreted according to the following table.

Type	Cue NOT pre-existing	Cue pre-existing
A	NOOP	DELETE cue
B	INSERT interval, data undefined	MODIFY interval, PRESERVE data
C	INSERT data, interval undefined	MODIFY data, PRESERVE interval
D	INSERT cue	MODIFY cue

20.3.3 Cue Equality

Cue modification has *no effect* if cue argument is equal to the *pre-existing* cue. The dataset will detect equality of cue intervals and avoid unneccesary reevaluation of internal indexes. However, the definition of *object equality* for cue data may be application dependent. For this reason the **update** operation allows a custom equality function to be specified using the optional parameter *equals*. Note that the equality function is evaluated with the cue data property as arguments, not the entire cue.

```
function equals(a, b) {
    ...
    return true;
}

ds.update(cues, {equals:equals});
```

The default equality function used by the dataset is the following:

```
function equals(a, b) {
    // Create arrays of property names
    let aProps = Object.getOwnPropertyNames(a);
    let bProps = Object.getOwnPropertyNames(b);
    let len = aProps.length;
    let propName;
    // If properties lenght is different => not equal
    if (aProps.length != bProps.length) {
        return false;
    }
    for (let i=0; i<len; i++) {
        propName = aProps[i];
        // If property values are not equal => not equal
        if (a[propName] !== b[propName]) {
            return false;
        }
    }
    // equal
```

(continues on next page)

(continued from previous page)

```
    return true;
}
```

Given that object equality is appropriately specified, **update** operations may safely be repeated, even if cue data have not changed. For instance, this might be the case when an online source of timed data is polled repeatedly for updates. Results from polling may then be forwarded directly to the **update** operation. The return value will indicate if any actual modifications occured.

20.3.4 Update Result

The **update** operation returns an array of items describing the effects for each cue argument. Result items are identical to event arguments **eArg** defined in *Event Argument*.

```
// update result item
let item = {key: ..., new: {...}, old: {...}}
```

key Unique cue key

old Cue *before* modification, or undefined if cue was inserted.

new Cue *after* modification, or undefined if cue was deleted.

It is possible with result items where both **item.new** and **item.old** are undefined. For instance, this will be the case if a cue is both inserted and deleted as part of a single update operation (see *Batch Operations*).

20.3.5 Batch Operations

The **update()** operation is *batch-oriented*, implying that multiple cue operations can be processed as one atomic operation. A single batch may include a mix of **insert**, **modify** and **delete** operations.

```
let ds = new Dataset();

let cues = [
    {
        key: "key_1",
        interval: new Interval(2.2, 4.31),
        data: "foo"
    },
    {
        key: "key_2",
        interval: new Interval(4.4, 6.9),
        data: "bar"
    }
];

ds.update(cues);
```

Batch oriented processing is crucial for the efficiency of the **update** operation. In particular, the overhead of reevaluating internal indexes may be paid once for the accumulated effects of the entire batch, as opposed to once per cue modification.

Warning: Repeated invocation of **update** within a single processing task is an **anti-pattern** with respect to performance! Cue operations should if possible be aggregated and applied together as a single batch.

```
// cues
let cues = [...];

// NO!
cues.forEach(function(cue)) {
    ds.update(cue);
}

// YES!
ds.update(cues);
```

20.3.6 Cue Chaining

It is possible to include several cue arguments concerning the same key in a single batch to **update**. This is called *chained* cue arguments. Chained cue arguments will be applied in the given order, and the net effect in terms of cue state will be equal to the effect of splitting the cue batch into individual invokations of **update**. Internally, chained cue arguments are collapsed into a single cue operation with the same net effect. For instance, if a cue is first inserted and then deleted within a single batch, the net effect is *no effect*.

Correct handling of chained cue arguments introduces an extra test within the **update** operation, possibly making it slightly slower for very large cues batches. If the cue batch is known to *not* include any chained cue arguents, this may be indicated by setting the option *chaining* to false. The default value for *chaining* is true.

```
ds.update(cues, {chaining:false});
```

Warning: If the *chaining* option is set to false, but the cue batch still contains chained cue arguments, this violation will not be detected. The consequences are not grave. The *old* value of result items and event arguments will be incorrect for chained cues.

20.4 Update Convenience Methods

The dataset defines a few convenience methods for updating the dataset implemented on top of the basic update primitive. Single cue operations **addCue** for inserting or modifying a cue and **removeCue** to delete a cue. These operations support *Batch Operations* through repeated invocation. Cue arguments will be buffered by an internal **builder** object and submitted as a **single** update operation on the dataset, just after the current JS task has completed. The result from the update operation is availble on a **updateDone** promise.

```
ds
    .addCue("key_1", new Interval(1,2), data)
    .removeCue("key_2")
    .addCue("key_1", new Interval(1,3), data);

ds.updateDone.then((result) => {console.log(result)});
```

Note: Once resolved, the **updateDone** promise is replaced by a new promise for the next update operation, but still available on the same **updateDone** property. So, for later update results just access the **updateDone** getter property again.

```
function show_result(update_result) {
    console.log("update result");
}

ds.updateDone.then(show_result);
ds.addCue("k", new Interval(612, 10000), "k")

setTimeout(() => {
    ds.addCue("l",  new Interval(614, 10000), "l");
    ds.updateDone.then(show_result);
}, 1000);
```

To specify **options** for *Batch Operations* use a custom **builder** object.

```
let options;
let builder = ds.makeBuilder(options);

builder.updateDone.then(()=>{console.log("result")});
builder
    .addCue("key_1", new Interval(1,2), data);
    .removeCue("key_2");
    .clear();
```

Tip: For interactive use **_addCue** and **_removeCue** avoid buffering cue arguments by using the update primitive directly.

```
let update_result = ds._addCue("key_1", new Interval(1,2), data);
```

20.5 Lookup

The operation **lookup(interval, mask)** identifies all cues *matching* a specific interval on the timeline. The parameter **interval** specifices the target interval and **mask** defines what interval relations count as a *match*, see *Interval Match*. Similarly, dataset provides an operation **lookup_delete(interval, mask)** which deletes all cues matching a given interval. This operation is more efficient than **lookup** followed by cue deletion using **update**.

20.5.1 Lookup endpoints

In addition to looking up cues, dataset also supports looking up *cue endpoints*. The operation **lookup_endpoints(interval)** identifies all cue endpoints **inside** the given interval, as defined in *Interval Comparison*. The operation returns a list of (endpoint, cue) pairs, where endpoint is the *low* or the *high* endpoint of the cue interval.

```
{
    endpoint: [value, high, closed, singular],
    cue: {
        key: "mykey",
        interval: new Interval(...),
        data: {...}
    }
}
```

The endpoint property is defined in *Endpoint Types*.

20.6 Events

Dataset supports three events **batch**, **change** and **remove**, as defined in *Cue Collection*.

20.7 Cue Ordering

See *Cue Ordering*.

20.8 Performance

The dataset implementation targets high performance with high volumes of cues. In particular, the efficiency of the **lookup** operation is important as it is used repeatedly during media playback. The implementation is therefor optimized with respect to fast **lookup**, with the implication that internal costs related to indexing are paid by the **update** operation.

The **lookup** operation depends on a sorted index of cue endpoints, and sorting is performed as part of the **update** operation. For this reason, **update** performance is ultimately limited by sorting performace, i.e. `Array.sort()`, which is O(NlogN) (see sorting complexity). Importantly, support for *batch operations* reduces the sorting overhead by ensuring that sorting is needed only once for a each batch operation, instead of repeatedly for every cue argument. The implementation of **lookup** uses binary search to identify the appropriate cues, yielding O(logN) performance. The crux of the lookup algorithm is to resolve the cues which *COVERS* (see :ref:'interval-comparison') the lookup interval in sub linear time.

To indicate the performance metrics of the dataset, some measurements have been collected for common usage patterns. For this particular test a standard laptop computer is used (Lenovo ThinkPad T450S, 4 cpu Intel Core i5-53000 CPU, Ubuntu 18.04). Tests are run with Chrome and Firefox, with similar results. Though results will vary between systems, these measurements should at least give a rough indication.

Update performance depends primarily the size of the cue batch, but also a few other factors. The update operation is more efficient if the dataset is empty ahead of the operation. Also, since the update operation depends on sorting internally, it matters if the cues are mostly sorted or random order.

Tests operate on cue batches of size 100.000 cues, which corresponds to 200.000 cue endpoints. Results are given in milliseconds.

INSERT	100.000 sorted cues into empty dataset	278
INSERT	100.000 random cues into empty dataset	524
INSERT	100.000 sorted cues into dataset with 100.000 cues	334
INSERT	100.000 random cues into dataset with 100.000 cues	580
INSERT	10 cues into dataset with 100.000 cues	2
LOOKUP	100.000 endpoints in interval from dataset of 100.000 cues	74
LOOKUP	20 endpoints from dataset with 100.000 cues	1
LOOKUP	50.000 cues in interval from dataset of 100.000 cues	80
LOOKUP	10 cues in interval from dataset of 100.000 cues	1
LOOKUP_DELETE	50.000 cues in interval from dataset with 100.000 cues	100
LOOKUP_DELETE	10 cues in interval from dataset with 100.000 cues	1
DELETE	50.000 random cues from dataset with 100.000 cues	280
DELETE	10 random cues from dataset with 100.000 cues	10
CLEAR	Clear dataset with 100.000 cues	29

The results show that the dataset implementation is highly efficient for **lookup** operations and **update** operations with modest cue batches, even if the dataset is preloaded with a large volume of cues (100.000). In addition, (not evident from this table) **update** behaviour is tested up to 1.000.000 cues and appears to scale well with sorting costs. However, batch sizes beyond 100.000 are not recommended, as this would likely hurt the responsiveness of the webpage too much. To maintain responsiveness it would make sense to divide the batch in smaller parts and spread them out in time. Use cases requiring loading of over 100.000 cues might also be rare in practice.

CHAPTER 21

Sequencer

Contents

21.1 Introduction

The *Sequencer* implements precisely timed *playback* of *timed data*. Playback is controlled using one or two *TimingObjects*. Timed data is represented as *cues* managed by a *Dataset*.

Demo

Demo Sequencer Point Mode sequencing timed data using a single timing object (see *Point Mode*).

Demo Sequencer Interval Mode sequencing timed data using two timing objects (see *Interval Mode*).

21.2 Linear Media State

Continuous media experiences require *media state* to be well defined along its timeline. For *discrete* media content, cues tied to points or intervals on the timeline is a simple and efficient mechanism for achieving this goal:

> At any given point **p** on the timeline, the **media state** at point **p** is given by the set of all cues with an interval covering point **p**.

For instance, by using cues with back-to-back intervals **... [a,b), [b,c), ...** one may ensure that the entire timeline is covered by media content. The use of open and closed brackets removes any ambiguity regarding the media state at interval endpoints.

Importantly, this definition is also a solid basis for implementing *navigation* and *playback* of the *media state*. For example, jumping from one point to another on the timeline requires a quick transition between two different media states, i.e. deactivation of some cues and activation of others. Furthermore, during continuous media playback, cues must be activated and deactivated at the correct time and in the correct order.

The sequencer encapsulates all of this, leaving the programmer to specify appropriate actions as cues become active and inactive, by implementing handlers for sequencer **change** and **remove** events.

21.3 Definition

- The sequencer implements *Cue Collection* and holds a **subset** of the cues managed by its source *Dataset*.
- At any time, the sequencer holds the particular subset of cues that are **active** cues.
- The sequencer emits **change**, **remove** and **batch** events (see: *Cue Collection*) as cues are **activated** or **deactivated** during playback.

Active cues Cues are **active** or **inactive** based on the playback position, and how it compares to the *cue interval*, which defines the **validity** of the cue on the timeline. The sequencer may well be an empty collection, if no cues are **active** at a particular time.

Precisely timed events As *playback position* gradually changes during timed playback, cues must be activated or deactivated at the correct time. The sequencer dynamically manipulates its own cue collection and precisely schedules **change** and **remove** events (see: *Cue Collection*) for activation and deactivation of cues.

Flexible timeline navigation and playback Sequencers have full support for all kinds of navigation and playback allowed by *Timing Object*. This includes jumping on the timeline, setting the playback velocity, backwards playback and even accelerated playback. For instance, jumping on the timeline might cause all active cues to be deactivated, and a new set of cues to be activated.

Dynamic dataset Sequencers support dynamic changes to its source *Dataset*, at any time, also during playback. Cues added to the dataset will be activated immediately if they should be active. Cues removed from the dataset will be deactivated, if they were active. Modified cues will stay active, stay inactive, be activated or be deactived, whichever is appropriate.

Sequence of timed events The **change** and **remove** events of the sequencer provide the full storyline (i.e. sequence of transitions) for the set of active cues. This also includes initialization, due to the *Initial Events* semantics of the **change** event. The **change** event will initially emit cues that are already active - immediately after the subscription is made. After that, **change** and **remove** events will communicate all subsequent changes, including changes to cue data.

21.4 Programming Model

From the perspective of the programmer, the sequencer is a **dynamic, read-only view** into a *Dataset* of cues. The view can *always* be trusted to represent the set of active cues correctly, and to communicate all future changes as **change** and **remove** events, at the correct time. This makes for an attractive programming model, where precisely timed playback-visualizations of timed data can be achieved simply by implementing handlers for sequencer events. In other words, the programmer only needs to specify what it means for a cue to become active or inactive.

As such, the sequencer encapsulates all the timing-related complexity, and transforms the challenge of *time-driven visualization* into a challenge of *data-driven visualization*. Reactive data visualization is already a rich domain with mature practices and a broad set of tools and frameworks to go with them. So, the sequencer essentially bridges the gap; allowing timed visualizations to reap the fruits of modern data visualation tools.

from data-driven to time-driven visualization

As a trivial example, this demonstrates playback of subtitles in a Web page (without the need for a video).

```javascript
/*
    Simplistic subtitle playback

    const subtitles = [{
        id: "1234",
        start: 123.70,
        end: 128.21,
        text: "This is a subtitle"
    }, ...]
*/

let ds = new Dataset();
let to = new TimingObject();
let activeCues = new Sequencer(ds, to);

// subtitle DOM element
let elem = document.getElementById("subtitle");

// create and load cues
let cues = subtitles.map(sub => {
    let itv = new Interval(sub.start, sub.end);
    return {key: sub.key, interval: itv, data: sub};
});
ds.update(cues);

activeCues.on("change", function (eArg) {
    // activated subtitle
    elem.innerHTML = eArg.new.data.text;
});

activeCues.on("remove", function (eArg) {
    // deactivate subtitle
    elem.innerHTML = "";
});

// start playback !
to.update({velocity:1});
```

Note: Note how the application-specific part of this example is only a few lines of code (highlighted lines) limited to making cues from specific data format (20-22) and rendering cues (17, 28, 33).

21.5 Sequencer Modes

The sequencer supports two distinct modes of operation, *point mode* and *interval mode*, with different definitions for **active** cues.

21.5.1 Point Mode

Point mode means that sequencing is based on a *moving sequencing point*.

The sequencer is controlled by a single timing object and uses the *position* of the timing object as *sequencing point*.

A cue is **active** whenever the *sequencing point* is **inside** the **cue interval**.

Point mode sequencing is the traditional approach for sequencing timed data based on a media clock.

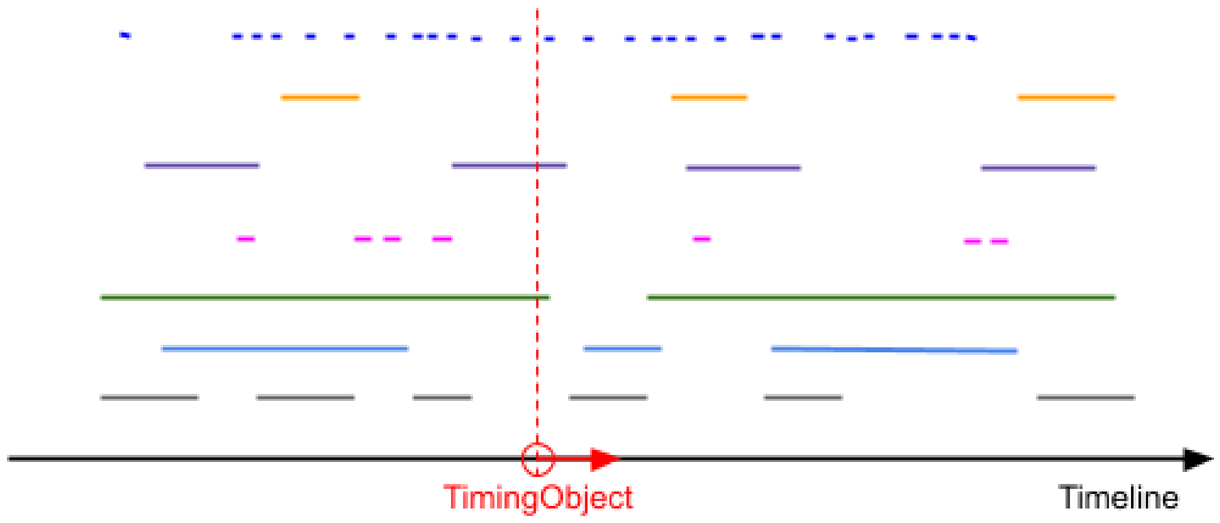


Fig. 1: The figure illustrates a set of cues and a timing object. The vertical dashed line shows the position of the timing object on the timeline. Cues that are intersected by this line, one green and one purple, are *active*. As the timing object moves to the right, a blue cue will soon be activated to, just before the green cue is deactivated.

Initialise a sequencer in point mode by supplying a single timing object.

```
// dataset
let ds;
// timing object
let to = new TimingObject();
// point mode sequencer
let activeCues = new Sequencer(ds, to);
```

Demo

Demo Sequencer Point Mode sequencing timed data using a single timing object (see *Point Mode*).

21.5.2 Interval Mode

Interval mode means that sequencing is based on a *moving sequencing interval*.

The sequencer is controlled by two timing objects, and the sequencer uses the *positions* of the two timing objects to form the *sequencing interval*.

A cue is **active** whenever at least one point **inside** the *sequencing interval* is also **inside** the **cue interval**.

Interval mode is useful for playback of sliding windows of timed data. For instance, interval mode sequencing can be used in conjuction with point mode sequencing, to prefetch timed data just-in-time for point mode sequenced rendering.

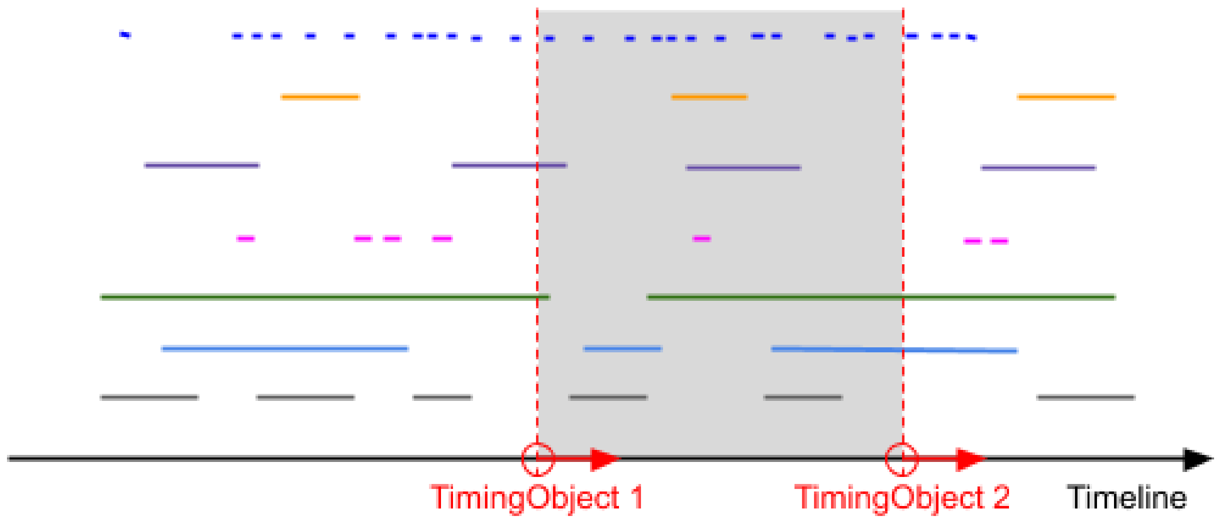


Fig. 2: The figure illustrates a set of cues and two timing objects. The vertical dashed lines shows the positions of the timing objects on the timeline. Cues that are visible between these two lines are *active*. In this case, the active cues include 2 gray, 2 light-blue, 2 green, 1 pink, 2 purple, 1 yellow and 12 blue cues. As both timing objects move to the right, the first event will be the activation of the blue cue to the right of the second timing object.

Initialise a sequencer in interval mode by supplying two timing objects.

```
// dataset
let ds;

// timing object
let to1 = new TimingObject();

/*
    skewconverter
    creaate timing object 10.0 ahead of to1
*/
let to2 = new SkewConverter(to1, 10.0);

// interval mode sequencer
let s2 = new Sequencer(ds, to1, to2);
```

Demo

Demo Sequencer Interval Mode sequencing timed data using two timing objects (see *Interval Mode*).

21.6 Cue ordering

During playback, if multiple cues share endpoint values, playback events will also be due at the same time. In this case, cue ordering is based on *Endpoint Ordering*. Endpoint ordering is used for forward movement, or no movement. For backward movement, endpoint ordering is reversed.

Changes in the dataset may also cause events to be emitted for multiple cues at the same time. For instance, if new cues are inserted into the dataset, some of them might immediately become active. In this case, cue ordering is still based on *Endpoint Ordering* and movement direction. For forward movement or no movement, cues are ordered by their *low* endpoints. For backward movement, cues are ordered by their *high* endpoints, and the ordering is reversed.

By default, accessors **keys()**, **values()** and **entries()** do not provide any guarantees with respect to cue ordering.

21.7 Events

Sequencer supports three events **batch**, **change** and **remove**, as defined in *Cue Collection*.

21.8 Cue Ordering

See *Cue Ordering*.

CHAPTER 22

Events

Contents

22.1 Introduction

All classes in timingsrc uses a custom framework for event notification which supports the *Initial Events* pattern.

22.2 Terminology

event provider

- defines one or more **named events**
- accepts subscriptions and un-subscriptions of **event callbacks** for **named events**.
- triggers **event notification** for **named events** by invoking subscribing **event callbacks**.

event consumer

- subscribes by associating **event callback** with **named event** of **event provider**
- receives **event notification** by **event callback** invocation

22.3 Subscription and unsubscription

Event consumers subscribe and un-subscribe to events using operations **.on()** and **.off()** of the event provider. For instance, this is how to subscribe to and un-subscribe from a *change* event.

```
// event provider
let ep;

// register handler with named event
let sub = ep.on("change", function (eArg, eInfo) {
    // handle change event
});

// unregister subscription
ep.off("change", sub);
```

It is safe to subscribe or unsubscribe from within an event callback. For instance, this can be used to implement **fire once** semantics.

```
// event provider
let ep;

// subscribe
let sub = ep.on("change", function() {
    ep.off("change", sub);
});
```

22.4 Event Callback

22.4.1 Execution

When an event is triggered, the execution of event callbacks is always decoupled using `Promise.then()`. This avoids nested invocation of event callbacks which may be confusing and hard to debug.

22.4.2 Same Callback

It is safe to use the same event callback with multiple subscriptions. For instance, in some cases it may be practical to handle different event types using only one callback. If needed, the *eInfo* parameter of *event_callback()* identifies the source of the event, i.e. the event provider and the event name.

22.4.3 This

It is also possible to control the value of the `this` object during event callback execution. This is useful when the callback handler is a class method, thus the callback handler must be invoked with `this` set to the class instance. There are at least three ways to achieve this.

One approach is to wrap the event handler in a function which explicitly invokes the event handler with the correct `this` object.

```
class EventConsumer {

    constructor(eventProvider) {
        this.ep = eventProvider;
        // subscribe to event from event provider
        let self = this;
        this.sub = this.ep.on("change", function(eArg, eInfo) {
            self.onevent(eArg, eInfo);
        });
    }

    // event handler as class method
    onevent(eArg, eInfo) {...}
}
```

Another approach is to use `.bind()`.

```
class EventConsumer {

    constructor(eventProvider) {
        this.ep = eventProvider;
        // subscribe to event from event provider
        this.sub = this.ep.on("change", this.onevent.bind(this));
    }

    // event handler as class method
    onevent(eArg, eInfo) {...}
}
```

Or, you can explicitly specify the `this` object as an option with *`EventProviderInterface.on()`*.

```
class EventConsumer {

    constructor(eventProvider) {
        this.ep = eventProvider;
        // subscribe to event from event provider
        this.sub = this.ep.on("change", this.onevent, {ctx:this});
    }

    // event handler as class method
    onevent(eArg, eInfo) {...}
}
```

22.5 Initial Events

The traditional semantic of events systems is that events convey **state changes**. So, when an event consumer subscribes to an event, there will be no event notification until the next state change occurs. This yields a common pattern when mirroring *stateful* event providers:

1. Request a snapshot of the currect state
2. Subscribe to future state changes. For each state change, update the snapshot accordingly.

In code, this might look something like this:

```
// event provider
let ep;

// refresh UI based on current state of event provider
function refresh (state) {...}

// request initial state
let state = ep.get_state();
refresh(state);

// subscribe to future state changes
ep.on("change", function(eArg) {
    /*
        update state somehow
        - apply diff from eArg
        - or, fetch the current state
    */
    state = ep.get_state();
    refresh(state);
});
```

The basic idea of **initial events** is to simplify so that we handle both initial state and subsequent state changes the same manner, with a single event callback.

```
// event provider
let ep;

// refresh UI based on current state
function refresh (state) {...}

// subscribe to future state changes
ep.on("change", function(eArg) {
    /*
        update state somehow
        - apply diff from eArg
        - or, fetch the current state
    */
    state = ep.get_state();
    refresh(state);
});
```

For this to be correct, the event provider must provide the initial state as event notifications, prior to delivering events as usual. The **initial events** semantic thus simplifies application code and shifts initialization complexity from the event consumer onto the event provider.

The initial events semantic only affects a few details in the `EventProviderInterface()`. Primarily, there is an extra event. The *eInfo.init* parameter of `event_callback()` is `true` for initial events. It is also possible to opt out of initial events semantic, by specifying `{init:false}` as option to `EventProviderInterface.on()`.

CHAPTER 23

Events API

Events API is common to all objects implementing the *Events*. This includes *Dataset* and *Sequencer*.

`event_callback`(*eArg*, *eInfo*)

Callback function for event notification, invoked by event provider.

Arguments

- **`eArg`** (*`object`*) – Event argument. Application specific object defined by event provider. May be `undefined`. Typically used to describe the state transition that caused the event to be triggered.
- **`eInfo`** (*`object`*) – Event information. Generic object defined by event provider.

  eInfo.src event provider object

  eInfo.name event name

  eInfo.sub subscription object

  eInfo.init true if event is **init event**

`class EventProviderInterface`()

Event provider interface

`EventProviderInterface.`**`on`**(*name*, *callback*[, *options*])

Register a callback for events with given name. Returns subscription handle.

Arguments

- **`name`** (*`string`*) – event name
- **`callback`** (*`function`*) – *`event_callback()`*
- **`options`** (*`object`*) – Callback options

  options.ctx Specify context for `this` object in event callback. If not specified, `this` is the event provider.

  options.init Boolean. If false, opt out of **init event semantics**.

Throws Error if event name is not defined.

Returns object subscription. Use subscription handle to cancel subscription with `off()`.

`EventProviderInterface.`**`off`**`(`*subscription*`)`

Un-register a callback for given subscription handle.

Arguments

- **`subscription`** (*`object`*) – subscription handle from `on()`

CHAPTER 24

Cue Collection API

The cue collection API is common to all objects implementing the *Cue Collection* interface. This includes *Dataset* and *Sequencer*.

class CueCollection(*options*)

Arguments

- **options.order** (*object*) – order, see *Cue Ordering*

Constructor abstrac base class for cue collection.

CueCollection.**size**
see JS Map Documentation

CueCollection.**has**(*key*)
see JS Map Documentation

CueCollection.**get**(*key*)
see JS Map Documentation

CueCollection.**keys**()
see JS Map Documentation

No particular ordering is guaranteed.

CueCollection.**values**()
see JS Map Documentation

No particular ordering is guaranteed.

CueCollection.**entries**()
see JS Map Documentation

No particular ordering is guaranteed.

CueCollection.**cues**(*options*)
The cues method is similar to `CueCollection.values()`, but conveniently adds support for sorting the resulting cues. See *Cue Ordering*. Order supplied in this function take precedent over order supplied in constructor.

Arguments

- **`options.order`** (*`object`*) – ordering of cues
  - string "low" : ascending order by lower cue interval endpoint
  - string "high": ascending order by higher cue interval endpoint
  - function cmp: custom order by supplying ordering function, similar to *Array.sort(cmp)*.

Returns Array array of cues

`CueCollection.`**`on`**`(`*name*, *callback*[, *options*]`)`
see *`EventProviderInterface.on()`*

`CueCollection.`**`off`**`(`*name*, *subscription*`)`
see *`EventProviderInterface.off()`*

CHAPTER 25

Timing Object API

class TimingObject(*options*)

Timing object constructor.

Arguments

- **options** (*object*) – options for timing object creation
- **options.range** (*Array*) – range of timing object timeline [low, high]
- **options.position** (*float*) – initial position
- **options.velocity** (*float*) – initial velocity
- **options.acceleration** (*float*) – initial acceleration

TimingObject.**vector**

Current state vector of timing object.

Returns object initial state vector.

TimingObject.**range**

Current range restrictions on timing object.

Param Array range new range : [low, high]

Returns Array range : [low, high]

TimingObject.**ready**

Promise resolved when timing object is ready

Returns Promise ready promise

TimingObject.**pos**

Convenience accessor for timing object position, based on query.

Returns float current position

TimingObject.**vel**

Convenience accessor for timing object velocity, based on query.

Returns float current velocity

`TimingObject.`**`acc`**

Convenience accessor for timing object acceleration, based on query.

Returns float current acceleration

`TimingObject.`**`timingsrc`**

Setter/getter property current parent of timing object.

timingsrc is `undefined` if timing object is local object (does not have a parent). Otherwise timingsrc may be *Timing Object* or *Timing Provider*

Param object timingsrc new timingsrc

Returns object timingsrc

`TimingObject.`**`isReady`**`()`

Timing object ready state (internal vector defined)

Returns boolean true if timing object is ready

`TimingObject.`**`query`**`()`

Query timing object.

see *TimingObject Query*

Returns vector current state vector

`TimingObject.`**`update`**`(`*vector*`)`

Update timing object.

see *TimingObject Update*

Returns Promise update promise

`TimingObject.`**`on`**`(`*name*, *callback*[, *options*]`)`

Register event handler

see `EventProviderInterface.on()`

see *Change Event*, *Timeupdate Event* and *Rangechange Event*

`TimingObject.`**`off`**`(`*name*, *subscription*`)`

Unregister event handler

see `EventProviderInterface.off()`

CHAPTER 26

Timing Converter API

Contents

26.1 Introduction

All timing converters implement the *Timing Object API*.

26.2 Skew Converter API

Skew Converter shifts all positions of its parent timingsrc by *skew*. Skew can be set at any time, and an *skewchange* event is emitted whenever the skew is changed.

`class SkewConverter`(*timingsrc*, *skew*)

Arguments

- **`timingsrc`** (`object`) – parent timingobject or timingprovider
- **`skew`** (`float`) – initial skew

Creates a skew converter tied to parent *timingsrc*. Skew converter defines **skewchange** event.

`SkewConverter.`**`skew`**

Param float skew new skew

Returns float skew current skew

26.3 Scale Converter API

Scale Converter multiplies the vector of its parent timingsrc with a factor. This factor can be set at any time, and an *scalechange* event is emitted whenever the scale is changed.

`class ScaleConverter`(*timingsrc*, *factor*)

Arguments

- **`timingsrc`** (*`object`*) – parent timingobject or timingprovider
- **`factor`** (*`float`*) – initial factor

Creates a scale converter tied to parent *timingsrc*. Scale converter defines **scalechange** event.

`ScaleConverter.`**`factor`**

Param float factor new factor

Returns float factor current factor

26.4 Loop Converter API

Loop Converter is essentially a modulo operation on its parent timingsrc, looping the position of the converter over values within its range.

`class LoopConverter`(*timingsrc*, *range*)

Arguments

- **`timingsrc`** (*`object`*) – parent timingobject or timingprovider
- **`range`** (*`Array`*) – initial range, e.g. [low,high]

Creates a loop tied to parent *timingsrc*.

26.5 Delay Converter API

Delay Converter mirrors the behaviour of its parent timingsrc, yet with a fixed delay.

`class DelayConverter`(*timingsrc*, *delay*)

Arguments

- **`timingsrc`** (*`object`*) – parent timingobject or timingprovider
- **`delay`** (*`float`*) – initial delay

Creates a delay converter tied to parent *timingsrc*. Delay converter defines **delaychange** event.

`DelayConverter.`**`delay`**

Param float delay new delay

Returns float delay current delay

26.6 Timeshift Converter API

Timeshift Converter projects the current behavior of the parent timingsrc into the future, or back in time. Positive offset is speculative, essentially predicting future states of the parent timingsrc.

`class TimeshiftConverter`(*timingsrc*, *offset*)

Arguments

- **`timingsrc`** (*`object`*) – parent timingobject or timingprovider
- **`offset`** (*`float`*) – initial time offset

Creates a timeshift converter tied to parent *timingsrc*. Timeshift converter defines **offsetchange** event.

`TimeshiftConverter.`**`offset`**

Param float offset new time offset

Returns float offset current time offset

CHAPTER 27

Timing Provider API

see *Timing Provider*

class TimingProvider()

Abstract constructor function for timing provider object

Returns object timingProvider

TimingProvider.**vector**

Get current state vector of timing provider

Returns object vector current state vector of timing provider.

TimingProvider.**skew**

Get current skew of timing provider clock - relative to local clock

Returns float skew current skew

TimingProvider.**range**

Get current range restrictions of timing provider

Returns Array range range of timing provider, [low, high]

TimingProvider.**update**(*vector*)

Request update to current state vector of timing provider

Arguments

- **vector** (*object*) – update vector

  Update vectors may be partially complete. For instance, to change the position, only the new position must be given.

TimingProvider.**on**(*type*, *callback*)

Register callback on timingprovider event. Support for *Initial Events* is *not* required. Supported event-types: **skewchange** and **vectorchange**

Arguments

- **type** (*string*) – event type

- **`callback`** (*`function`*) – event callback

CHAPTER 28

Interval API

class Interval(*low*[, *high*[, *lowInclude*[, *highInclude*]]])

Arguments

- **low** (*float*) – leftmost endpoint of interval
- **high** (*float*) – rightmost endpoint of interval
- **lowInclude** (*boolean*) –

  low endpoint value included in interval
  true means **left-closed**
  false means **left-open**
  true by default
- **highInclude** (*boolean*) –

  high endpoint value included in interval
  true means **right-closed**
  false means **right-open**
  false by default

If only **low** is given, or if **low == high**, the interval is singular. In this case **lowInclude** and **highInclude** are both true.

If **low** is *-Infinity*, **lowInclude** is always true If **high** is *Infinity*, **highInclude** is always true

Interval.**low**
float: left endpoint value

Interval.**high**
float: right endpoint value

Interval.**lowInclude**
boolean: true if interval is left-closed

Interval.**highInclude**
boolean: true if interval is right-closed

`Interval.`**`singular`**
boolean: true if interval is singular

`Interval.`**`finite`**
boolean: true if both **low** and **high** are finite values

`Interval.`**`length`**
float: interval length (**high-low**)

`Interval.`**`endpointLow`**
endpoint: low endpoint [value, false, lowInclude, singular]

`Interval.`**`endpointHigh`**
endpoint: low endpoint [value, true, highInclude, singular]

`Interval.`**`toString`**`()`

Returns string

Human readable string

`Interval.`**`covers_endpoint`**`(`*p*`)`

Arguments

- **`p`** (*`number`*) – point

Returns boolean True if point p is inside interval

Test if point p is inside interval.

See *Interval Comparison*.

```
let a = new Interval(4, 5)  // [4,5)
a.covers_endpoint(4.0)  // true
a.covers_endpoint(4.3)  // true
a.covers_endpoint(5.0)  // false
```

`Interval.`**`equals`**`(`*other*`)`

Arguments

- **`other`** (`Interval`) – interval to compare with

Returns boolean true if intervals are equal

See *Interval Comparison*.

`Interval.`**`compare`**`(`*other*`)`

Arguments

- **`other`** (`Interval`) – interval to compare with

Returns int comparison relation

Compares interval to another interval, i.e. **cmp(interval, other)**. See *Interval Comparison*.

```
let a = new Interval(4, 5)  // [4,5)
let b = new Interval(4, 5, true, true)  // [4,5]
a.compare(b) == Interval.Relation.COVERED  // true
b.compare(a) == Interval.Relation.COVERS   // true
```

`Interval.`**`match`**`(`*other*[, *mask=62*]`)`

Arguments

- **other** (Interval) – interval to compare with

Returns boolean true if intervals match

Matches two intervals. Mask defines what consitutes a match. See *Interval Match*.

```
let a = new Interval(4, 5)  // [4,5)
let b = new Interval(4, 5, true, true)  // [4,5]
a.match(b) // true
b.match(a) // true
```

`Interval.`**`Relation`**

```
{
    OUTSIDE_LEFT: 64,   // 0b1000000
    OVERLAP_LEFT: 32,   // 0b0100000
    COVERED: 16,        // 0b0010000
    EQUALS: 8,          // 0b0001000
    COVERS: 4,          // 0b0000100
    OVERLAP_RIGHT: 2,   // 0b0000010
    OUTSIDE_RIGHT: 1    // 0b0000001
}
```

`Interval.Interval.Relation.`**`OUTSIDE_LEFT`**

`Interval.Relation.`**`OVERLAP_LEFT`**

`Interval.Relation.`**`COVERED`**

`Interval.Relation.`**`EQUAL`**

`Interval.Relation.`**`COVERS`**

`Interval.Relation.`**`OVERLAP_RIGHT`**

`Interval.Relation.`**`OUTSIDE_RIGHT`**

`Interval.`**`cmpLow`**(*interval_a*, *interval_b*)

Arguments

- **interval_a** (Interval) – interval A
- **interval_b** (Interval) – interval B

Returns int

a < b : -1
a == b : 0
a > b : 1

Use with Array.sort() to sort Intervals by their low endpoint.

```
a = [
    new Interval(4,5),
    new Interval(2,3),
    new Interval(1,6)
];
a.sort(Interval.cmpLow);
// [1,6), [2,3), [4,5)
```

`Interval.`**`cmpHigh`**(*interval_a*, *interval_b*)

Arguments

- **interval_a** (Interval) – interval A
- **interval_b** (Interval) – interval B

Returns int

a < b : -1
a == b : 0
a > b : 1

Use with Array.sort() to sort Intervals by their high endpoint.

```
a = [
    new Interval(4,5),
    new Interval(2,3),
    new Interval(1,6)
];
a.sort(Interval.cmpHigh);
// [2,3), [4,5), [1,6)
```

CHAPTER 29

Dataset API

class Dataset(*options*)

Arguments

- **options.order** (*object*) – see *Cue Ordering*

Creates an empty dataset.

Dataset.**update**(*cues*[, *options*])

Arguments

- **cues** (*iterator*) – iterable of cues or single cue
- **options** (*object*) – options

Returns Array list of cue change items

Insert, replace and delete cues from the dataset. For details on how to construct cue parameters see *Update*. For details on return value see *Update Result*.

- options.equals: custom equality function for cue data.

  See *Cue Equality*.

- options.chaining: support chaining. True by default.

  See *Cue Chaining*.

- options.safe: safe mode. False by default.

  See *Cue Management*.

- options.debug: debug mode. False by default.

  Performs integrity testing of internal datastructures after each update operation, throwing exceptions if not passed.

Dataset.**addCue**(*key*, *interval*, *data*)

Add or replace a single cue. See *Update Convenience Methods*.

Arguments

- **`key`** (*`object`*) – cue key
- **`interval`** (`Interval`) – cue interval
- **`data`** (*`object`*) – cue data

Returns Dataset dataset dataset

`Dataset.`**`removeCue`**(*key*)

Remove a single cue. See *Update Convenience Methods*.

Arguments

- **`key`** (*`object`*) – cue key

Returns Dataset dataset dataset

`Dataset.`**`makeBuilder`**(*options*)

Make cue argument builder object with options. See *Update Convenience Methods*.

Params object options update options (see *`Dataset.update()`*)

Returns object builder cue argument update builder

- builder.addCue(key, interval, data)
- builder.removeCue(key)

`Dataset.`**`clear`**()

Returns Array list of change items: cue changes caused by the operation

Clears all cues of the dataset. Much more effective than iterating through cues and deleting them.

`Dataset.`**`lookup`**(*interval*[, *mask*])

Arguments

- **`interval`** (`Interval`) – lookup interval
- **`mask`** (*`int`*) – match mask

Returns Array list of cues

Returns all cues matching a given interval on dataset. Lookup mask specifies the exact meaning of *match*, see *Interval Match*.

Note also that the lookup operation may be used to lookup cues that match a single point on the timeline, simply by defining the lookup interval as a single point, see *Definition*.

`Dataset.`**`lookup_endpoints`**(*interval*)

Arguments

- **`interval`** (`Interval`) – lookup interval

Returns Array list of {endpoint: endpoint, cue:cue} objects

Lookup all cue endpoints on the dataset, within some interval see *Lookup endpoints*.

`Dataset.`**`lookup_delete`**(*interval*[, *mask*])

Arguments

- **`interval`** (`Interval`) – lookup interval
- **`mask`** (*`int`*) – match mask

Returns Array list of cue change items

Deletes all cues *matching* a given lookup interval. Similar to *lookup*, see *Lookup*.

`Dataset.`**`size`**
: see `ObservableMapInterface.size()`

`Dataset.`**`has`**`(`*key*`)`
: see `ObservableMapInterface.has()`

`Dataset.`**`get`**`(`*key*`)`
: see `ObservableMapInterface.get()`

`Dataset.`**`keys`**`()`
: see `ObservableMapInterface.keys()`

`Dataset.`**`values`**`()`
: see `ObservableMapInterface.values()`

`Dataset.`**`entries`**`()`
: see `ObservableMapInterface.entries()`

`Dataset.`**`cues`**`(`*options*`)`
: see *`CueCollection.cues()`*

`Dataset.`**`on`**`(`*name*, *callback*[, *options*]`)`
: see *`EventProviderInterface.on()`*

`Dataset.`**`off`**`(`*name*, *subscription*`)`
: see *`EventProviderInterface.off()`*

CHAPTER 30

Sequencer API

class Sequencer(*dataset*, *to_A*[, *to_B*], *options*)

Arguments

- **dataset** (`Dataset`) – source dataset of sequencer
- **to_A** (`TimingObject`) – first timing object
- **to_B** (`TimingObject`) – optional second timing object
- **options.order** (`object`) – see *Cue Ordering*

Creates a sequencer associated with a dataset.

`Sequencer.`**dataset**
Dataset used by sequencer.

`Sequencer.`**size**
see `CueCollection.size()`

`Sequencer.`**has**(*key*)
see `CueCollection.has()`

`Sequencer.`**get**(*key*)
see `CueCollection.get()`

`Sequencer.`**keys**()
see `CueCollection.keys()`

`Sequencer.`**values**()
see `CueCollection.values()`

`Sequencer.`**entries**()
see `CueCollection.entries()`

`Sequencer.`**cues**(*options*)
see `CueCollection.cues()`

`Sequencer.`**on**(*name*, *callback*[, *options*])
see `EventProviderInterface.on()`

`Sequencer.`**`off`**`(`*name*, *subscription*`)`
    see *`EventProviderInterface.off()`*

CHAPTER 31

MediaSync API

class MCorp.**mediaSync**(*elem*, *to*[, *options*])

Constructor function. Returns handle for controlling synchronization.

Arguments

- **elem** (*HTMLMediaElement*) – The HTMLMediaElement to synchronize
- **to** (TimingObject) – The timingobject to synchronize after
- **options** (*object*) – Synchronization options
- **options.skew** (*float*) – (default 0.0) Skew for timing object position, ehead of synchronization. Tip: calculate by start offset of content - start position of timing object.
- **options.automute** (*boolean*) – (default true) Mute the media element when playing too fast (or too slow)-
- **options.mode** (*string*) – (default "auto") - "skip": Force "skip" mode - i.e. don't try using playbackRate. - "vpbr": Force variable playback rate. Normally not a good idea - "auto" (default): try playbackRate. If it's not supported, it will struggle for a while before reverting. If 'options.remember' is not set to false, this will only happen once after each browser update.
- **options.debug** (*object*) – (default null) If debug is true, log to console, if a function, the function will be called with debug info
- **options.target** (*float*) – (default 0.025 - 25ms ~ lipsync) Target precision. Default is likely OK, if we can do better, we will. Target too narrow, makes for a more skippy experience. When using variable playback rates, this parameter is ignored (target is always 0)
- **options.remember** (*boolean*) – (default true) Remember the last experience on this device - stores support or lack of support for variable playback rate. Records in localStorage under key "mediascape_vpbr", clear it to re-learn.

Returns object mediaSync mediaSync object

`MCorp.mediaSync.`**`getSkew`**`()`
Get the current skew

Returns float skew current skew

`MCorp.mediaSync.`**`setSkew`**`(`*skew*`)`
Skew the timing object. The same effect can be achieved by using a *Skew Converter*.

Arguments

- **`skew`** (*`float`*) – new skew

`MCorp.mediaSync.`**`setOption`**`(`*key*, *value*`)`
Set or update options

Arguments

- **`key`** (*`string`*) – The option key to set
- **`value`** (*`object`*) – The option value to set

```
sync.setOption("debug", false); // Disable debugging
sync.setOption("target", 0.1); // Change to coarser target
```

`MCorp.mediaSync.`**`getMethod`**`()`
Get the current method for synchronization

Returns string method "skip" or "playbackrate"

`MCorp.mediaSync.`**`setMotion`**`(`*to*`)`
Set the timing object to synchronize after

Arguments

- **`to`** (`TimingObject`) – The timingobject to synchronize after

`MCorp.mediaSync.`**`stop`**`()`
Stop synchronization

CHAPTER 32

Welcome to timingsrc!

Timingsrc is a programming model for precisely timed Web applications. The model is based on the *Timing Object*, which allows precise synchronization and control across multiple media sources, media types, UI components and media frameworks.

For **online** timing support, connect an online *Timing Provider* to the *Timing Object*. The *Shared Motion Timing Provider* is hosted online and provides millisecond precise timing **globally** for Web clients and is open for non-commercial experimentation.

Need to synchronize HTML5 video? Check out *Demo MediaSync*

Need to synchronize timed data? Check out *Demo Sequencer Point Mode* or *Demo Sequencer Interval Mode*

Need to go online? Check out *Demo TimingProvider*

CHAPTER 33

Timing Object

```
let to = new TimingObject();
```

The *TimingObject* is the central concept of the timingsrc programming model. In essence, the timingobject is a timeline with an API for control. If you set velocity, the position on the timeline will increase in time according to that velocity. The timing object additionally supports behavior like time-shifting, different velocities (including backwards), and acceleration.

- *Timing Object*
- *Timing Object API*

CHAPTER 34

Timing Converter

```
let c = new SkewConverter(to, 4.0);
```

A *TimingConverter* is a special kind of timing objects that depends on a *parent* timing object. Timing converters are useful when you need an alternative representations for a timing object. For instance, timing converters may be used to skew or scale the timeline.

- *Timing Converter*
- *Timing Converter API*

CHAPTER 35

Timing Provider

```
let to = new TimingObject({provider: timing_provider});
```

Timing objects may be connected to remote timing resources, i.e. timing resources which live outside the browsing context, for instance hosted by an online timing service. This is done by initializing the timing object with a *Timing-Provider*. Timing providers are proxy objects to external timing resources, allowing timing objects to be used across different service implementations for timing resources.

- *Timing Provider*
- *Timing Provider API*

CHAPTER 36

Dataset and Sequencer

```
let ds = new Dataset();
let s = new Sequencer(ds, to);
```

Consistent playback of timed data is a key use case for the timing object. This is achieved using *Dataset* and *Sequencer*. Dataset allows any type of time data to be represented as cues. Sequencers dynamically provides the set of active cues, always consistent with the timing object. Both dataset and sequencer implement the :ref‘cuecollection‘ interface.

- *Cue Collection*
- *Cue Collection API*
- *Dataset*
- *Dataset API*
- *Sequencer*
- *Sequencer API*

CHAPTER 37

MediaSync

```
let ms = new MediaSync(to, video_element);
```

Another important use case is consistent playback of HTML5 audio and video. This is achieved by connecting the video element to the timing object, using the *MediaSync* wrapper.

- *MediaSync*
- *MediaSync API*

CHAPTER 38

Indices and tables

- genindex

Index