Trace Event Format

Last updated: October 2016
nduca@, dsinclair@

Introduction

The Trace Event Format is the trace data representation that is processed by the Trace Viewer[1] application. Trace Viewer recognizes four input variations.

- Two are representations of the JSON format:
  a. A string dump of a JSON array.
  b. A JSON object.
- Two are representations of the Linux ftrace data:

[1] https://github.com/catapult-project/catapult/tree/master/tracing

a. Text dump starting with **# tracer:**.
b. An Android systrace HTML dump.

This document describes the JSON input format and provides links for more information on the Linux ftrace functionality.

JSON Format

The JSON events can be provided in two slightly different structures.

JSON Array Format

The simplest format accepted by the trace viewer is the JSON array format. Essentially, an array of event objects. The events do not have to be in timestamp-sorted order.

```
[ {"name": "Asub", "cat": "PERF", "ph": "B", "pid": 22630, "tid": 22630, "ts": 829},
  {"name": "Asub", "cat": "PERF", "ph": "E", "pid": 22630, "tid": 22630, "ts": 833}
]
```

JSON Array Format

When provided as a string to the importer the **]** at the end of the JSON Array Format is optional. The Trace Viewer importer will automatically add a **]** if needed to turn the string into valid JSON.. This is to support tracing systems that can not cleanly finish writing the trace. For example, when tracing the exit of a program.

JSON Object Format

The JSON Object Format allows for more flexibility in the data passed to Trace Viewer. There is one required property, ***traceEvents***, and some optional properties. The events do not have to be in timestamp-sorted order.

```
{
  "traceEvents": [
    {"name": "Asub", "cat": "PERF", "ph": "B", "pid": 22630, "tid": 22630, "ts":
829},
    {"name": "Asub", "cat": "PERF", "ph": "E", "pid": 22630, "tid": 22630, "ts":
833}
  ],
  "displayTimeUnit": "ns",
  "systemTraceEvents": "SystemTraceData",
  "otherData": {
    "version": "My Application v1.0"
  },
  "stackFrames": {...}
  "samples": [...],
}
```

JSON Object Format

If provided ***displayTimeUnit*** is a string that specifies in which unit timestamps should be displayed. This supports values of *“ms”* or *“ns”*. By default this is value is *“ms”*.

If provided ***systemTraceEvents*** is a string of Linux ftrace data or Windows ETW trace data. This data must start with **# tracer:** and adhere to the Linux ftrace format or adhere to Windows ETW format.

If provided, ***powerTraceAsString*** is a string of BattOr power data.

If provided, the **stackFrames** field is a dictionary of stack frames, their ids, and their parents that allows compact representation of stack traces throughout the rest of the trace file. It is optional but sometimes very useful in shrinking file sizes.

The **samples** array is used to store sampling profiler data from a OS level profiler. It stores samples that are different from trace event samples, and is meant to augment the traceEvent data with lower level information. It is OK to have a trace event file with just sample data, but in that case **traceEvents** must still be provided and set to []. For more information on sample data, refer to the global samples section.

If provided, **controllerTraceDataKey** is a string that specifies which trace data comes from tracing controller. Its value should be the key for that specific trace data. For example, {..., "controllerTraceDataKey": "traceEvents"} means the data for traceEvents comes from the tracing controller. This is mainly for the purpose of clock synchronization.

Any other properties seen in the object, in this case ***otherData*** are assumed to be metadata for the trace. They will be collected and stored in an array in the trace model. This metadata is accessible through the **Metadata** button in Trace Viewer.

Event Descriptions

There are a common set of fields for each of the events.

```
{
  "name": "myName",
  "cat": "category,list",
  "ph": "B",
  "ts": 12345,
  "pid": 123,
  "tid": 456,
  "args": {
    "someArg": 1,
    "anotherArg": {
      "value": "my value"
    }
  }
```

```
}
```

General Event Structure

- **name:** The name of the event, as displayed in Trace Viewer
- **cat**: The event categories. This is a comma separated list of categories for the event. The categories can be used to hide events in the Trace Viewer UI.
- **ph:** The event type. This is a single character which changes depending on the type of event being output. The valid values are listed in the table below. We will discuss each phase type below.
- **ts:** The tracing clock timestamp of the event. The timestamps are provided at microsecond granularity.
- **tts:** Optional. The thread clock timestamp of the event. The timestamps are provided at microsecond granularity.
- **pid**: The process ID for the process that output this event.
- **tid**: The thread ID for the thread that output this event.
- **args:** Any arguments provided for the event. Some of the event types have required argument fields, otherwise, you can put any information you wish in here. The arguments are displayed in Trace Viewer when you view an event in the analysis section.

Optional

- **cname:** A fixed color name to associate with the event. If provided, cname must be one of the names listed in trace-viewer's base color scheme's reserved color names list

Phases

The following table lists all event types and their associated phases:

Event type	Event phases
Duration Events	**B** (begin), **E** (end)
Complete Events	**X**

Instant Events	`i`
	
	Deprecated
	`I`

Counter Events	**C**
Async Events	**b** (nestable start), **n** (nestable instant), **e** (nestable end) *Deprecated* **S** (start), **T** (step into), **p** (step past), **F** (end)
Flow Events	**s** (start), **t** (step), **f** (end)
Sample Events	**P**

Object Events	**N** (created), **O** (snapshot), **D** (destroyed)
Metadata Events	**M**
Memory Dump Events	**V** (global), **v** (process)
Mark Events	**R**
Clock Sync Events	**c**
Context Events	**(**, **)**

Source: https://source.chromium.org/…/trace_event_common.h;l=1070-1093

Duration Events

Duration events provide a way to mark a duration of work on a given thread. The duration events are specified by the **B** and **E** phase types. The **B** event must come before the corresponding **E** event. You can nest the **B** and **E** events. This allows you to capture function calling behaviour on a thread. The timestamps for the duration events *must* be in increasing order for a given thread. Timestamps in different threads do not have to be in increasing order, just the timestamps within a given thread.

The only required fields for the **E** events are the **pid**, **tid**, **ph** and **ts** fields, all others are optional. If you provide **args** to both the **B** and **E** events then the arguments will be merged. If there is a duplicate argument value provided the **E** event argument will be taken and the **B** event argument will be discarded.

```
{"name": "myFunction", "cat": "foo", "ph": "B", "ts": 123, "pid": 2343, "tid": 2347,
 "args": {
   "first": 1
 }
},
{"ph": "E", "ts": 145, "pid": 2343, "tid": 2347,
 "args": {
   "first": 4,
   "second": 2
 }
}
```

Duration Event Example

In this example we will create a single slice in the Trace Viewer. The slice is named **myFunction** and has a single category **foo**. The slice starts at time **123**µs and will have a duration of **22**µs. The event will be displayed as part of process **2343** and thread **2347**. There will be two arguments attached to the slice, **first: 4** and **second: 2**.

```
{ ..., "ts": 1.0, "tid": 1, "ph": "B", "name": "A"},
{ ..., "ts": 1.1, "tid": 1, "ph": "B", "name": "Asub"},
{ ..., "ts": 3.9, "tid": 1, "ph": "E"},
{ ..., "ts": 4.0, "tid": 1, "ph": "E"}
```

Nested Duration Event Example

In the above example, there will be two slices created. One for function **A** and one for function **Asub**. When drawn in Trace Viewer the **Asub** slice will be nested under the **A** slice. The **A** function will be recorded as having a duration of **3**µs. **Asub** will have a recorded duration of **2.8**µs.

```
  { ..., "ts": 1.0, "tid": 1, "ph": "B", "name": "A"},
  { ..., "ts": 0.9, "tid": 2, "ph": "B", "name": "B"},
  { ..., "ts": 1.1, "tid": 1, "ph": "E"},
  { ..., "ts": 4.0, "tid": 2, "ph": "E"}
```

Thread Interleaving of Duration Events

In the above, even though the timestamps are out of order, everything will work correctly because within a given thread they are strictly increasing.

While duration events allow you to trace the function flow, they must be nested. It is not possible to have non-nested duration events for a given thread. If you need to have durations that do not nest property you should use Async events instead.

Stack Traces on Duration Events

Stack traces can be associated with duration events. There are two ways to provide a stack trace. First, you can provide an **sf** field that is an id for a stackFrame object in the "stackFrames" map:

```
{
  traceEvents: [
    { ..., "ph": "B", "name": "A", "sf": 7},
    { ..., "ph": "E", "name": "A", "sf": 9}
  ],
  stackFrames: {
    "5": { name: "main", category: "my app" },
    "7": { parent: "5", name: "SomeFunction", category: "my app" },
    "9": { parent: "5", name: "SomeFunction", category: "my app" }
  }
}
```

A duration event with a stack trace using the global stackFrame table

You can have different stack traces on begin and end events, or omit either one.

Stack traces can also be given as raw stacks. In this form, instead of **sf** you provide a **stack** array:

```
{
  traceEvents: [
    { ..., "ph": "B", "name": "A", "stack": ["0x1", "0x2"]}
  ]
```

```
}
```

A directly-specified stack

A stack is just an array of strings. The 0th item in the array is the rootmost part of the callstack, the last item is the leafmost entry in the stack, e.g. the closest to what was running when the event was issued. You can put anything you want in each trace, but strings in hex form ("0x1234") are treated as program counter addresses and are eligible for symbolization.

You cannot specify both a sf and a stack field at once.

Complete Events

Each complete event logically combines a pair of duration (B and E) events. The complete events are designated by the **X** phase type. In a trace that most of the events are duration events, using complete events to replace the duration events can reduce the size of the trace to about half.

There is an extra parameter **dur** to specify the tracing clock duration of complete events in microseconds. All other parameters are the same as in duration events. The **ts** parameter indicate the time of the start of the complete event. Unlike duration events, the timestamps of complete events can be in any order.

An optional parameter **tdur specifies the thread clock duration of complete events in microseconds.**

```
{"name": "myFunction", "cat": "foo", "ph": "X", "ts": 123, "dur": 234, "pid": 2343,
"tid": 2347,
 "args": {
   "first": 1
 }
```

Complete Event Example

Complete events support stack traces as well. The **sf** and **stack** fields, if given, specify the stack trace for the start of the event. The **esf** and **estack** fields, if given, specify the end stack trace of the event. The **esf** and **estack** fields have the same exact contents and rules as with duration event stack traces.

Instant Events

The instant events correspond to something that happens but has no duration associated with it. For example, vblank events are considered instant events. The instant events are designated by the **i** phase type.

There is an extra parameter provided to instant events, **s**. The **s** property specifies the scope of the event. There are four scopes available global (**g**), process (**p**) and thread (**t**). If no scope is provided we default to thread scoped events. The scope of the event designates how tall to draw the instant event in Trace Viewer. A thread scoped event will draw the height of a single thread. A process scoped event will draw through all threads of a given process. A global scoped event will draw a time from the top to the bottom of the timeline.

```
{"name": "OutOfMemory", "ph": "i", "ts": 1234523.3, "pid": 2343, "tid": 2347, "s":
"g"}
```

Instant Event Example

Thread-scoped events can have stack traces associated with them in the same style as duration events, by putting either **sf** or **stack** records on the event. Process-scoped and global-scoped events do not support stack traces at this time.

Counter Events

The counter events can track a value or multiple values as they change over time. Counter events are specified with the **C** phase type. Each counter can be provided with multiple series of data to display. When multiple series are provided they will be displayed as a stacked area chart in Trace Viewer. When an **id** field exists, the combination of the event **name** and **id** is used as the counter name. Please note that counters are process-local.

```
{..., "name": "ctr", "ph": "C", "ts":  0, "args": {"cats":  0}},
{..., "name": "ctr", "ph": "C", "ts": 10, "args": {"cats": 10}},
{..., "name": "ctr", "ph": "C", "ts": 20, "args": {"cats":  0}}
```

Counter Event Example

In the above example the counter tracks a single series named **cats**. The cats series has a value that goes from 0 to 10 and back to 0 over a 20µs period.

```
{..., "name": "ctr", "ph": "C", "ts":  0, "args": {"cats":  0, "dogs": 7}},
{..., "name": "ctr", "ph": "C", "ts": 10, "args": {"cats": 10, "dogs": 4}},
{..., "name": "ctr", "ph": "C", "ts": 20, "args": {"cats":  0, "dogs": 1}}
```

Multi Series Counter Example

In this example we have a single counter named **ctr**. The counter has two series of data, **cats** and **dogs**. When drawn, the counter will display in a single track with the data shown as a stacked graph.

Async Events

Async events are used to specify asynchronous operations. e.g. frames in a game, or network I/O. Async events are specified with the **b**, **n** and **e** event types. These three types specify the *start*, *instant* and *end* events respectively. You can emit async events from different processes and different threads. Each async event has an additional required parameter **id**. We consider the events with the same **category** and **id** as events from the same event tree. An optional **scope** string can be specified to avoid **id** conflicts, in which case we consider events with the same **category**, **scope**, and **id** as events from the same event tree. For instance, if we have an event, A, and its child event, B. Trace viewer will infer the parent-child relationship of A and B from the fact that event A and B have the same **category** and **id**, and the fact that A's start and end time pair encompasses that of B. When displayed, an entire async event

chain will be drawn such that the parent will be the top slice, and its children are in rows beneath it. The root of the async event tree will be drawn as the top-most slice with a dark top border.

```
{"cat": "foo", "name": "async_read", "id": 0x100, "ph": "b", "args": {"name" :
"~/.bashrc"}},
{"cat": "foo", "name": "async_read", "id": 0x100, "ph": "e"}
```

Async Event Example

Async events can have nested async events in them. For example, you might begin to load from a server, receive the headers, then later receive the body. A nested async event should have the same **category** and **id** as its parent.

```
{"cat": "foo", "name": "url_request", "ph": "b", "ts": "0", "id": 0x100},
{"cat": "foo", "name": "url_headers", "ph": "b", "ts": "1", "id": 0x100},
{"cat": "foo", "name": "http_cache", "ph": "n", "ts": "3", "id": 0x100},
{"cat": "foo", "name": "url_headers", "ph": "e", "ts": "2", "id": 0x100,
 "args": {
   "step": "headers_complete"
   "response_code": 200,
 }
},
{"cat": "foo", "name": "url_request", "ph": "e", "ts": "4", "id": 0x100}
```

Nestable Async Event Example

In the UI, this would show up as a large multi-colored **url_request** bar that begins at the **url_request** “b” time and end at the **url_request** “e” time. A second colored bar named **url_headers** that starts at the **url_headers** “b” time and ends at the **url_headers** “e” time, will be drawn in the row underneath the **url_request** bar. A third thin bar named **http_cache** that starts and ends at “n” time, will be displayed as nested underneath the **url_headers** bar. Trace-viewer reconstructs the event tree according to the timestamp, **ts**, of events with the same **category** and **id**. If Trace-viewer does not find a matching begin or end event, an warning will be displayed in the analyze view when user clicks on the slice that is not probably formed.

Flow Events

The flow events are very similar in concept to the Async events, but allow duration to be associated with each other across threads/processes. Visually, think of a flow event as an arrow between two duration events. With flow events, each event will be drawn in the thread it is emitted from. The events will be linked together in Trace Viewer using lines and arrows.

The flow events are designated by the **s, t,** and **f** phase types. These have the same meanings as the corresponding Async event. But, flow events must **bind** to specific slices in order to exist. There are two types of binding that can happen:

- enclosing slice binding, meaning the flow event is bound to its enclosing slice

- next slice binding, meaning the flow event is bound to the next slice that happens in time

Each flow event phase has a specific type of binding point:

- Flow start (ph='s'): binding point is always "enclosing slice"
- Flow step (ph='t'): binding point is "enclosing slice"
- Flow end (ph='f'): binding point is "next slice". If the event contains bp="e" then the binding point is "enclosing slice."

A bit of detail on "next slice" binding rules: Next slice is defined as the next slice to begin >= the timestamp of the flow. In the case of multiple slices having the same timestamp as the flow event, the earliest event in the trace buffer is chosen.

If there is no slice that encloses or follows a flow event [based on its phase type], then the flow event is considered invalid.

If the time-distance between the **s** and **f** phase events is very short then the flow lines will be nearly vertical, and will be invisible in the Trace Viewer until zoomed in. If the **s** event is lost (due to the limited size of the tracing buffers) then the flow lines will not be displayed at all. However the queue_duration (the time the event spent in the queue, which is the time between the **s** and **f** events) may still be available since it is stored in the **f** event.

Sample Events (deprecated)

Sample events provide a way of adding sampling-profiler based results in a trace. They get shown as a separate track in the specified thread. Sample events are designated by the **P** phase type.

```
{"name": "sample", "cat": "foo", "ph": "P", "ts": 123, "pid": 234, "tid": 645 }
```

Sampling Event Example

Samples are drawn similar to instant events in that they have 0 duration so they are, effectively, a line in the Trace Viewer. They are useful if you have a very hot function and you do not want to emit thousands of events but want to get a general understanding of how much a function is executed.

Sample events can have stack traces associated with them in the same style as duration events, by putting either **sf** or **stack** records on the event.

Object Events

Objects are a building block to track complex data structures in traces. Since traces go for long periods of time, the formative concept for this is a way to identify objects in a time varying way. To identify an object, you need to give it an **id**, usually its pointer, e.g. id: "0x1000". But, in a long running trace, or just a trace with a clever allocator, a raw pointer 0x1000 might refer to two different objects at two different points in time. Object events do not have an **args** property associated with them.

In the trace event format, we think in terms of object instances. e.g. in C++, given class `Foo` , `new Foo()` at `0x1000` is an instance of `Foo`. There are three trace phase types associated with objects they

are: **N**, **O**, and **D**, object created, object snapshot and object destroyed respectively.

```
{"name": "MyObject", "ph": "N", "id": "0x1000", "ts": 0, "pid": 1, "tid": 1},
{"name": "MyObject", "ph": "D", "id": "0x1000", "ts": 20, "pid": 1, "tid": 1}
```

Object Events Example

This sequence of events shows **MyObject**, identified by `0x1000` that is alive from 0µs to 20µs.

Objects, like slices, obey the convention of the start time being inclusive and the deletion time being exclusive.

```
{"name": "MyOtherObject", "ph": "N", "id": "0x1000", "ts": 20, "pid": 1, "tid": 1},
{"name": "MyOtherObject", "ph": "D", "id": "0x1000", "ts": 25, "pid": 1, "tid": 1}
```

Reusing Object IDs

So, if we also had this set of events in addition to the **MyObject** events, then, a **MyOtherObject** reference at 20µs points at a **MyOtherObject** instance, not a **MyObject** instance.

Sometimes, we may want to use an **id** that is not a memory address. To avoid conflicting with another object **id**, we can optionally specify a **scope**, similar to other events that have an **id**, like async events.

```
{"name": "MyObject", "ph": "N", "id": "0x1000", "scope": "some_scope", "ts": 0,
"pid": 1, "tid": 1}
```

Declaration of an Object with scoped ID

Snapshots

As mentioned above, object instance commands do not have an **args** property. Meaning, you can not associate data with object instances. This may seem silly, until you think about how traces are time varying. In software, very few things stay constant for their lifetime – rather they vary over time. Object snaphots allow you to output object state at a given instant in time. Object snapshots are designated with the **O** phase type.

```
{"name": "MyObject", "id": "0x1000", "ph": "O", "ts": 10, "pid": 1, "tid": 1,
 "args": {
  "snapshot": {...}
 }
}
```

Object Snapshot Example

The ellipsis inside the snapshot field can be anything you want, as much as you want. You can, for instance, put a complete dump of your rendering system into the snapshot.

Inside Trace Viewer we have the facility to use custom displays for different named objects. This allows,

for example, the Frame Viewer and the TCMalloc heap dump output to have vastly different displays.

Snapshot category override

By default, an object snapshot inherits the category of its containing trace event. For instance the MyObject snapshot created by this has the "cat" category:

```
{"name": "MyObject", "id": "0x1000", "ph": "O", "ts": 10, "pid": 1, "tid": 1,
 "cat": "cat",
 "args": {
  "snapshot": {...}
 }
}
```

Normal category handling example

However, sometimes the object being snapshotted needs its own category. This happens because the place that creates an object snapshot's values is often separate form where the objects' constructor and destructor is called. Categories for the object creation and deletion commands must match the snapshot commands. Thus, the category of any object snapshot may be provided **with** the snapshot itself:

```
{"name": "MyObject", "id": "0x1000", "ph": "O", "ts": 10, "pid": 1, "tid": 1,
 "cat": "cat",
 "args": {
  "snapshot": {
    "cat": "real_cat"
    ...
  }
 }
}
```

Category override example: this produces a MyObject snapshot with category=real_cat

Snapshot base_type override

By default, object snapshots and new/delete commands must use the same name throughout:

```
{"name": "MyObject", "id": "0x1000", "ph": "N", ...}
{"name": "MyObject", "id": "0x1000", "ph": "O", ...}
{"name": "MyObject", "id": "0x1000", "ph": "D", ...}
```

Normal naming

However, with polymorphic classes, it is often desired that the snapshots have the subtype's name in order to associate custom viewers with that type in the UI. This often conflicts with the desire to issue the N and D phase commands in the base type constructor: the base constructor and destructor cannot know the subtype, which makes it impossible to simply replace the "name"="MyObject" with

"name"="MySubObject" throughout. The solution is this:

```
{"name": "MyObject", "id": "0x1000", "ph": "N", ...}
{"name": "MyObjectSubType", "id": "0x1000", "ph": "O", "args": {
  "snapshot": {
    "base_type": "MyObject",
    ... regular args fields as usual…
    }
  }
}
{"name": "MyObject", "id": "0x1000", "ph": "D", ...}
```

base_type-based naming

References

In any complex system you'll have objects that reference each other. This ability is available in the trace format. For any event which allows **args**, e.g. duration events, async events, snapshot events, the **args** will be scanned after the trace file is loaded for properties of the form: `{"field": {"id_ref": "0x1000"}}`. The value for the **id_ref** must match the **id** provided by an object event. When replaced, the entire value of the `field` property will be substituted with the object snapshot. To reference a snapshot of an object with scoped **id**, we use the following form: `{"field": {"id_ref": "0x1000", "scope": "id_scope"}}`.

```
{"name": "SliceA", "ph": "B", "ts": 10, "pid": 1, "tid": 1,
 "args": {
  "obj": {
   "id_ref": "0x1000"
  }
 }
}
```

Object Reference Example

Given the above, the importing system will look for an **id** that is alive at 10µs, then look for a snapshot that was taken at 10µs, and replace the `obj` value with the object snapshot. If an object is created at 0µs, and has snapshots at 10µs and 20µs, and is deleted at 30µs, references bind in the following way:

Reference	Snapshot
5µs	10µs
15µs	10µs
20µs	20µs

25µs	20µs

Metadata Events

Metadata events are used to associate extra information with the events in the trace file. This information can be things like process names, or thread names. Metadata events are denoted by the **M** phase type. The argument list may be empty.

There are currently 5 possible metadata items that can be provided:

- **process_name**: Sets the display name for the provided pid. The name is provided in a **name** argument.
- **process_labels**: Sets the extra process labels for the provided pid. The label is provided in a **labels** argument.
- **process_sort_index**: Sets the process sort order position. The sort index is provided in a **sort_index** argument.
- **thread_name**: Sets the name for the given tid. The name is provided in a **name** argument.
- **thread_sort_index**: Sets the thread sort order position. The sort index is provided in a **sort_index** argument.

For the two sort_index items the **sort_index** argument value specifies the relative position you'd like the item to be displayed. Lower numbers are displayed higher in Trace Viewer. If multiple items all have the same sort index then they are displayed sorted by name and, given duplicate names, by id.

```
{"name": "thread_name", "ph": "M", "pid": 2343, "tid": 2347,
 "args": {
  "name" : "RendererThread"
 }
}
```

Metadata Event Example

In this example we are setting the **thread_name** for tid **2347** to be **RenderThread**.

Memory Dump Events

Memory dump events correspond to memory dumps of (groups of) processes. There are two types of memory dump events:

- **Global** memory dump events, which contain system memory information such as the size of RAM, are denoted by the **V** phase type and
- **Process** memory dump events, which contain information about a single process's memory usage (e.g. total allocated memory), are denoted by the **v** phase type.

All global (zero or one) and process (zero or more) memory dump events associated with a particular simultaneous memory dump have the same dump ID.

```
[
```

```
  {'id': 'dump_id', 'ts': 10, 'ph': 'V', 'args': {... global_args ...}},
  {'id': 'dump_id', 'ts': 11, 'ph': 'v', 'pid': 42, 'args': {... process_args ...}},
]
```

Global and process memory dump events example.

The precise format of the global and process arguments has not been determined yet (see the Memory Dumping design document for more details).

Mark Events

Mark events are created whenever a corresponding navigation timing API mark is created. Currently, this can happen in two ways:

1. Automatically at key times in a web page's lifecycle, like `navigationStart`, `fetchStart`, and `domComplete`.
2. Programmatically by the user via Javascript. This allows the user to annotate key domain-specific times (e.g. `searchComplete`).

```
[
  {'name': 'firstLayout', 'ts': 10, 'ph': 'R', 'cat': 'blink.user_timing',  'pid': 42, 'tid': 983},
]
```

Mark event example.

Clock Sync Event

Trace Viewer can handle multiple trace logs produced by different tracing agents and synchronize their clock domains. Clock sync events are used for clock synchronization. Clock sync events are specified by the **clock_sync** name and the **c** phase type.

Clock sync event can be recorded by an issuer or a receiver. The issuer asks the receiver to record the clock sync event. The issuer needs to record the sync ID and how long it takes for the receiver to do the recording, which is mainly for better precision. The receiver will record the sync ID for clock sync. Please see the tracing clock sync architecture design document for detailed design.

There are 2 args: **sync_id** and **issue_ts**.

- **sync_id**: The ID for the clock sync marker (string).
- **issue_ts**: The timestamp when the clock sync marker is issued in issuer's clock domain.

When the tracing agent is a receiver, only **sync_id** is needed and **ts** is the timestamp when the agent receives the marker.

```
[
  { "name": "clock_sync", "ph": "c", "ts": 123,
```

```
    "args": {
      "sync_id": "guid1"
    }
  }
]
```

Clock Sync Event Example - receiver

When the tracing agent is the issuer, both **sync_id** and **issue_ts** are needed. **ts** is the timestamp after the clock sync marker is recorded by the receiver in issuer's clock domain. The period between **issue_ts** and **ts** is how long the receiver takes to do the recording.

```
[
  { "name": "clock_sync", "ph": "c", "ts": 6790,
    "args": {
      "sync_id": "guid1",
      "issue_ts": 6789
    }
  }
]
```

Clock Sync Event Example - issuer

Context Events

Context events are used to mark sequences of trace events as belonging to a particular context (or a tree of contexts). There are two defined context events: **enter** and **leave**.

The enter event with phase **"("** adds a context to all following trace events on the same thread until a corresponding leave event with phase **")"** exits that context.

Contexts ids refer to context object snapshots. Context objects can also form a tree – see FrameBlamer for details.

```
[
  { "name": "SomeContextType", "ph": "(", "ts": 123, "id": "0x1234"},
  … Events belonging to context 0x1234 …
  { "name": "SomeContextType", "ph": ")", "ts": 123, "id": "0x1234"}
]
```

Context Event Example - entering and leaving

Linking IDs

Events with phase **"="** can be used to specify that two ids are identical. The first **id** is given as the **id** of the event, and the second **id** is given in the **linked_id** field of **args**.

```
[
  { "cat": "foo", "name": "async_read", "ph": "b", "id": "0x1000"},
  { "cat": "foo", "name": "link", "ph": "=", "id": "0x1000",
    "args": {"linked_id": "0x2000"}},
  { "cat": "foo", "name": "async_read", "ph": "e", "id": "0x2000"}
]
```

Linking IDs Example - beginning and ending an event using different IDs

More about IDs

By default, async event ids are considered global among processes. So events in different processes with the same **category** and **id** are grouped in the same tree. On the other hand, by default, object event ids are considered process local. So, it is possible to create two different objects with the same **id** in different processes. These default behaviors can be too limiting in some cases. Therefore, we introduced a new **id** field, called **id2**, that can be used instead of the default **id** field and explicitly specify if it is process-local or global.

```
[
  {"cat": "foo", "name": "async_read", "ph": "b", "id2": {"local":
"0x1000"}},
  {"cat": "foo", "name": "my_object", "ph": "N", "id2": {"global":
"0x2000"}},
  …
]
```

Explicitly specifying a process-local ID and a global ID

StackFrames Dictionary

To support efficient storage of stack traces, stack traces on events can be compactly represented with the help of a second array on the toplevel trace file. The stack frames dictionary is a mapping of stack frame IDs to a stack frame object. IDs can be any string or integer. Both must be provided.

The stack frames dictionary is of the form:

```
{
  frame_id: { ... stack frame… },
  ... more frames
}
```

Stack frames dictionary

A frame itself is of the form:

```
{
  'category': 'libchrome.so',
  'name': 'CrRendererMain',
  'parent': 1
}
```

Stack frame for "a" from "libchrome.so", pointing at stack frame 1 as its parent

If the stack frame has no parent, then the field should be omitted:

```
{
  'category': 'libc.so',
  'name': '_crtmain',
}
```

Root stack frame

Global Samples

A global sample is used to store OS level sampling data in the trace file, e.g. the results of hardware-assisted or timer-driven counter sampling.

```
{
  'cpu': 0, 'tid': 1, 'ts': 1000.0,
  'name': 'cycles:HG', 'sf': 3, 'weight': 1
}
```

Sample from cpu 0, thread 1, at 1000 microseconds, for the cycles:HG counter, with stack trace starting at stack frame 3, and weight of 1.

CPU may be left undefined. All other fields are mandatory. The 'value' field is the weight of the sample, to be used in summing samples together to determine their relative impact.

Linux Debug Format

The Linux debug format accepts the output of **function** tracing in **ftrace**. For more information see: http://lwn.net/Articles/365835/. The Trace Viewer importer looks for the **# tracer:** identifier to decide if it can import the data. It expects this entry to be on the first line if the provided file is a text file. If an Android systrace HTML file is provided the importer will scan the file for the **# tracer:** entry and start importing from that point.

Appendix

Last updated Sept 2020

The above document hasn't been updated for a few years. May be inaccuracies, as well, so please get in contact with @paul_irish to help fix them.

Thread Instruction Counts

Covered in PerfPlanet and implemented here, thread instruction counts can be added to a trace. It adds these fields to some events:

- **tidelta –** number of instructions of this event
- **ticount –** number of instructions at the start of this event

Additional Resources

A few tools parse & analyze trace events as used in Chrome:

- Catapult/traceviewer
- Lighthouse/tracehouse
- Speedscope
- Tracerbench