RUNTIME SCRIPT INTERFACE

View online: https://www.construct.net/en/make-games/manuals/construct-3/scripting/scripting-reference/iruntime

The `IRuntime` script interface provides a way for JavaScript code in Construct to interact with the engine.

Accessing the runtime script interface

The runtime script interface is typically accessed with the name `runtime`. Note however this is not in a global variable: it is only passed in specific places.

All scripts in event sheets have the runtime interface passed to them as a parameter named `runtime`. For more information see Scripts in event sheets.

In script files, the runtime interface is only passed to the `runOnStartup()` method. Outside of that, it is your responsibility to pass it along wherever else it is needed. For more information see Script files.

Runtime events

The following events can be listened for using the `addEventListener` method.

Many input events are also fired on the runtime interface. This is so they can be used in worker mode, where the `window` and `document` objects are not available, and so input events cannot be listened for. The runtime passes copies of the event objects, since they may have had to be posted over a MessageChannel to the worker.

"resize"

Fires when the display size changes, such as upon window resizes. The event object has `cssWidth` and `cssHeight` properties with the size of the main canvas in CSS units, and `deviceWidth` and `deviceHeight` properties with the size of the main canvas in device units.

"window-maximized"

"window-minimized"

Fires when the main app window is maximized or minimized. These events are only supported in desktop exports (Windows, macOS and Linux).

"pretick"

"tick"

"tick2"

These events fire every tick, at different points during ticking the engine. Each tick, `"pretick"` fires first, then behaviors are ticked, then `"tick"` fires, then it runs event sheets, then `"tick2"` fires. Use the `runtime.dt` property to access delta-time and step the game's logic by that amount of time.

"beforeprojectstart"

"afterprojectstart"

Fired once only when the first layout in the project starts. `"beforeprojectstart"` fires before the ILayout event `"beforelayoutstart"` on the first layout, which in turn is before *On start of layout* triggers. `"afterprojectstart"` fires after the ILayout event `"afterlayoutstart"` on the first layout, which in turn is after *On start of layout* triggers. In both events, all instances on the first layout are created and available to modify.

> *These events can use async handler functions, and the runtime will wait for them to finish before continuing.*

"beforeanylayoutstart"

"afteranylayoutstart"

"beforeanylayoutend"

"afteranylayoutend"

Fired whenever any layout starts or ends. The *start* events fire before and after the *On start of layout* trigger, and the *end* events fire before and after the *On end of layout* trigger. The event object has the property `layout` which is the ILayout for the layout that is starting or ending.

> *These events can use async handler functions, and the runtime will wait for them to finish before continuing.*

"keydown"

"keyup"

Fired when keys are pressed and release on the keyboard. These pass copies of a KeyboardEvent.

"mousedown"

"mousemove"

"mouseup"

.

"dblclick"

Fired when mouse input is received. These pass copies of a MouseEvent.

> *You can use pointer events like `"pointermove"` instead of mouse events to cover both mouse and touch input.*

"wheel"

Fired when mouse wheel input is received. This passes a copy of a WheelEvent.

"pointerdown"

"pointermove"

"pointerup"

"pointercancel"

Fired when pointer input is received. This covers mouse, pen and touch input. These pass copies of a PointerEvent. Construct also adds a `lastButtons` property for `"mouse"` type pointers with the previous `buttons` state, which makes it easier to detect if different mouse buttons have been pressed or released in this event.

> *See also the Tracking pointers example for a demonstration of how multiple simultaneous pointers can be tracked with JavaScript code.*

"deviceorientation"

"devicemotion"

Fired when device orientation or motion sensor input is received. These pass copies of a DeviceOrientationEvent or DeviceMotionEvent respectively.

> *These events require permission to be granted before they will fire. See the `requestPermission()` method in the Touch script interface.*

"suspend"

"resume"

Triggered when the browser/app suspends and resumes execution. Normally when the app goes in to the background (e.g. minimized, or switched back to the home screen), execution of the app is suspended to conserve system resources and save battery power, firing the `"suspend"` event. When the app is reopened, the `"resume"` event is fired and execution resumes. The `isSuspended` property also reflects the current suspend state.

"save"

"load"

Fired when the savegame system saves or loads the state of the game. The `saveData` property of the event object is extra JSON data to save in the "save" event, and the corresponding last saved data in the "load" event. This allows custom data stored in scripts to be integrated with the savegame system. Note this is serialized to JSON so things like object references and complex types cannot be saved directly.

> *These events can use async handler functions, and the runtime will wait for them to finish before continuing.*

"instancecreate"

Fired whenever any new instance is created. The event object has an `instance` property referring to the IInstance (or derivative) that was created.

"hierarchyready"

Fired for the root instance in a hierarchy after all instances have finished creating - see the IWorldInstance event for more details. When fired on IRuntime, the event object has an `instance` property referring to the IWorldInstance (or derivative) that was created.

"instancedestroy"

Fired whenever any instance is destroyed. After this event, all references to the instance are now invalid, so any remaining references to the instance should be removed or cleared to `null` in this event. Accessing an instance after it is destroyed will throw exceptions or return invalid data. The event object has an `instance` property referring to the IInstance (or derivative) that was destroyed. It also has an `isEndingLayout` property to indicate if the object is being destroyed because it's the end of a layout, or destroyed for other reasons.

"loadingprogress"

Fired during a loader layout when the value of the `loadingProgress` property changes.

> *This will still fire even if the loader layout is completely empty, with a progress of 1 to indicate fully complete.*

Runtime APIs

addEventListener(eventName, callback)

removeEventListener(eventName, callback)

Add or remove a callback function for an event. See *Runtime events* above for the available events.

> *See the Handling multiple events example for a way to conveniently handle events.*

---

objects

An object with a property for each object class in the project. For example if the project has an object named *Sprite*, then `runtime.objects.Sprite` will refer to the IObjectClass interface for *Sprite*.

> *In some cases, objects may have names that aren't valid JavaScript identifiers. In this case you can use the string property syntax, e.g.* `runtime.objects["Sprite"]`.

---

getInstanceByUid(uid)

Get an instance (an IInstance or derivative) by its UID (Unique ID), a unique number assigned to each instance and accessible via its `uid` property.

---

globalVars

An object with a property for each global variable on an event sheet in the project. For example if the project has a global variable on an event sheet named *Score*, then `runtime.globalVars.Score` provides access to the global variable from script.

> *In some cases, event variables may have names that aren't valid JavaScript identifiers. In this case you can use the string property syntax, e.g.* `runtime.globalVars["Score"]`.

---

mouse

A shorthand reference to the Mouse script interface. This is only set if the Mouse plugin is added to the project.

---

keyboard

A shorthand reference to the Keyboard script interface. This is only set if the Keyboard plugin is added to the project.

---

touch

A shorthand reference to the Touch script interface. This is only set if the Touch plugin is added to the project.

timelineController

A shorthand reference to the Timeline Controller script interface. This is only set if the Timeline Controller plugin is added to the project.

platformInfo

A shorthand reference to the IPlatformInfo script interface. Note this always available regardless of whether the Platform Info plugin has been added to the project.

collisions

The ICollisionEngine interface providing access to collision APIs.

layout

An ILayout interface representing the current layout.

getLayout(layoutNameOrIndex)

Get an ILayout interface for a layout in the project, by a case-insensitive string of its name or its zero-based index in the project.

getAllLayouts()

Return an array of ILayout interfaces representing all the layouts in the project, in the sequence they appear in the Project Bar.

goToLayout(layoutNameOrIndex)

End the current layout and switch to a new layout given by a case-insensitive string of its name, or its zero-based index in the project (which is the order layouts appear in the Project Bar with all folders expanded). Note the layout switch does not take place until the end of the current tick.

renderer

The IRenderer script interface representing Construct's renderer, which is used for drawing content. Note that content can only be drawn in certain events, such as ILayer's `"beforedraw"` and `"afterdraw"` events. However you may use the renderer to load resources such as textures at any time, including on startup.

assets

An IAssetManager interface providing access to project assets like sound and music files or other project files, as well as audio decoding utilities.

storage

An IStorage interface providing access to storage from scripts. Storage is shared with the Local Storage plugin.

projectId

A string of the project ID, as specified in Project Properties.

projectName

A string of the project name, as specified in Project Properties.

projectUniqueId

A string of random characters that Construct automatically generated to uniquely identify this specific project.

projectVersion

A string of the project version, as specified in Project Properties.

viewportWidth

viewportHeight

getViewportSize()

Read-only numbers with the project viewport size, as specified in Project Properties. The method returns both values at the same time.

loadingProgress

Return the current load progress on a loader layout in the range 0 to 1. This is the same as the *LoadingProgress* system expression. For more information, see the tutorial how to make a custom loading screen.

imageLoadingProgress

Return the current loading progress of any memory management *Load* system actions that are currently busy, on a scale of 0-1. This is the same as the *ImageLoadingProgress* system expression.

sampling

A read-only string indicating the project's *Sampling* property with one of the strings `"nearest"`, `"bilinear"` or `"trilinear"`.

isPixelRoundingEnabled

A boolean representing the project's *Pixel rounding* setting.

gameTime

Return the in-game time in seconds, which is affected by the time scale. This is equivalent to the *time* system expression.

wallTime

Return the in-game time in seconds, which is not affected by the time scale.

Note this is not exactly equivalent to the wallclocktime system expression, as it only measures time elapsed in-game, rather than since the project started.

tickCount

A read-only number with the number of ticks that have run since the project started.

timeScale

Set or get a number that determines the rate at which time passes in the project, e.g. 1.0 for normal speed, 2.0 for twice as fast, etc. This is useful for slow-motion effects or pausing.

dt

Return the value of delta-time, i.e. the time since the last frame, in seconds.

dtRaw

Return the wall-clock time in seconds since the last frame. Unlike *dt*, the "raw" value is not affected by the game time scale or the min/max delta-time clamping.

minDt

maxDt

Set the limits on the delta-time (`dt`) measurement. If the real-world delta-time increases above the maximum delta-time, it stops increasing the measurement used by Construct, corresponding to a dropping framerate causing the project to run in slow-motion. Conversely if the real-world delta-time decreases below the minimum delta-time, it stops decreasing the measurement used by Construct, corresponding to an increasing framerate causing the project to run in fast-forward. The defaults are a minimum delta-time of 0 (meaning the project never goes in to fast-forward mode) and a maximum delta-time of 1 / 30 (corresponding to a framerate of 30 FPS), meaning the project begins to go in to slow-motion as the framerate drops below 30 FPS. Going in to fast-forward can be useful for a "catch-up time" mode, and going in to slow-motion with low framerates is useful to prevent objects stepping too far every frame which can result in skipped collisions and unstable gameplay.

> *Note the inverse relationship between the framerate and delta-time: an increasing framerate results in an decreasing delta-time, and a decreasing framerate results in increasing delta-time.*

---

framerateMode

Change the project *Framerate mode* property at runtime. This can be one of the following strings: `"vsync"`, `"unlimited-tick"` or `"unlimited-frame"`. For more details, see the corresponding project property.

---

framesPerSecond

A read-only number indicating how many frames per second (FPS) the project is rendering. The most common display refresh rate is 60 Hz, so typically an efficiently designed project will render at 60 FPS. Note however if nothing is changing on-screen, then nothing is rendered, and so the FPS measurement may fall to 0 or display a lower result; this does not indicate poor performance, only that fewer frames are necessary to render. The `ticksPerSecond` property indicates how frequently the engine is stepping, which may be different to the frames rendered per second.

---

ticksPerSecond

A read-only number indicating how many ticks per second (TPS) the project is running at. Each tick processes the logic of the game. Usually a new frame is also rendered every tick, but if nothing changes then rendering a frame is skipped; further, depending on the framerate mode, stepping the engine and drawing frames may happen at different rates. Therefore the ticks per second may produce a different measurement to the frames per second. Usually the project will continually tick even if nothing is visually changing, and only stop ticking if the project is suspended, such as by being minimized or going in to the background.

---

cpuUtilisation

A timer-based estimate of Construct's main thread CPU usage over the past second, as a percentage in the range 0-1. This can help with performance measurements. Note however this only measures time in function calls that Construct knows about, so it may not include time spent running custom JavaScript code. Timer-based measurements can also be unreliable as most modern CPUs deliberately slow down if not fully loaded in order to save power, so the reading can be misleadingly high unless the system is under full load.

---

gpuUtilisation

A timer-based estimate of the GPU usage over the past second, as a percentage in the range 0-1. Not all devices support this, in which case this returns `NaN`. Timer-based measurements can also be unreliable as most modern GPUs deliberately slow down if not fully loaded in order to save power, so the reading can be misleadingly high unless the system is under full load.

isSuspended

A read-only boolean indicating if the runtime is currently suspended. When suspended, the runtime stops ticking and drawing anything, and remains inactive. This is normally done to save resources while in the background, such as when a browser tab is in the background or the application window minimized. The `"suspend"` and `"resume"` events are fired when this property changes.

exportDate

A read-only Date object representing the time the project was exported from Construct. In preview mode, this is the time the preview was launched.

callFunction(name, ...params)

Call a function in an event sheet, by a case-insensitive string of its name. Each parameter added after the name is passed to the function. There must be at least as many parameters provided as the function uses, although any additional parameters will be ignored. If the function has a return value, it will be returned from this method, else it returns `null`.

setReturnValue(value)

This can only be called from a script in an event sheet function with a return type other than *None*. It is essentially equivalent to the *Set return value* action. However the fact this method can be called from script can make it easier to return a value from script from an event sheet function. For example an event sheet function could contain a single script action with the code `runtime.setReturnValue(getMyValue())`, which means anywhere the event sheet function is called it returns the value of calling `getMyValue()` in JavaScript.

signal(tag)

Triggers *On signal*, resumes any events waiting for a signal with the given tag, and resolves any promise returned by `waitForSignal().`

waitForSignal(tag)

Returns a Promise that resolves when the given tag is signalled. It may be signalled by either the script API or an event sheet.

random()

Return a random number in the range [0, 1). This is similar to `Math.random()`, but can produce a deterministic sequence of values if the Advanced Random object overrides the system random.

destroyMultiple(iterable)

Destroys all instances in the iterable of IInstance (such as an array or Set of instances). This is equivalent to a for-of loop calling `destroy()` on every instance; however it is significantly more efficient when destroying large numbers of instances, as the internal engine updates necessary after destroying an instance are only processed once rather than repeatedly.

sortZOrder(iterable, callback)

Sort the relative Z order of all the IWorldInstances given by *iterable*, using a custom sort function as the *callback* which receives two *IWorldInstance* to compare as arguments. An example using a *myZOrder* instance variable for sorting a Sprite object's instances is given below.

```
runtime.sortZOrder(runtime.objects.Sprite.instances(),
        (a, b) => a.instVars.myZOrder - b.instVars.myZOrder);
```

saveCanvasImage(format, quality, areaRect)

Take a screenshot of the current display canvas. All parameters are optional - if none are specified, the entire canvas is saved in PNG format. The `format` parameter is a string of the MIME type of the image format to use, e.g. "image/png". Where a lossy format is used like "image/jpeg", the `quality` parameter is a number from 0-1 for the compression quality. A subset of the canvas area can be saved (e.g. for saving a cropped image) by specifying a DOMRect for the `areaRect` parameter using units of device pixels. The method returns a Promise that resolves with a Blob of the resulting image.

invokeDownload(url, filename)

Invoke a download of the resource at the given *url*, downloading with the given *filename*. Locally-generated content can be downloaded with this method using either a data URI or blob URL for *url*.

isInWorker

A read-only boolean indicating if the runtime is currently running in the context of a Web Worker. This is controlled by the *Use worker* project property. In worker mode, a more limited set of browser APIs is available. See Functions and classes available to Web Workers.

getHTMLLayer(index)

Retrieve the HTML element used to contain all HTML content on a given HTML layer in the project. The index is the zero-based index of the HTML layer (not the Construct layer). This method can only be used in DOM mode - if the project property *Use worker* is *Yes* then calling this method will throw an exception. This method is useful to obtain a parent element in which to insert custom HTML content such that it appears layered according to one of Construct's HTML layers. For more information see HTML layers.

async alert(message)

Show an alert message prompt using the alert() method. This is provided as a runtime method since it forwards the call to the DOM in worker mode. Note that unlike the standard browser `alert()` method, this is an async method - in worker mode it returns a promise that is resolved when the alert is closed, and execution in the worker will continue while the alert is showing. In DOM mode, the alert is blocking and will stop all execution while the alert is showing (but it still returns a promise that resolves when the alert is closed).

> *This method is also made available as a global `alert()` function in worker mode. This is to help make sure simple test code works as expected, even if the code is unintentionally run in the context of a Web Worker, where the browser `alert()` method is not normally available.*

Runtime SDK events and APIs

These events and APIs are normally only needed for use with the Addon SDK and should not normally be used when using JavaScript coding in Construct. However as they are also part of the `IRuntime` interface they are also documented here.

SDK events

"afterload"

Fired after the `_loadFromJson()` call after the rest of the runtime has finished loading some saved state. This means all objects are available and can be looked up by their UID. For example to save a reference to an instance, save its UID to JSON, load its UID and store it in `_loadFromJson()`, and then look it up with `getInstanceByUid()` in the "afterload" event.

SDK APIs

sdk

Access the ISDKUtils interface which implements more APIs specific to the addon SDK.