RUNTIME SCRIPTS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/runtime-scripts

Plugin and behavior addons have separate scripts that run in the context of the runtime (the Construct game engine) rather than the editor.

> *Effects don't use runtime scripts. They only provide shader code.*

Runtime documentation

In Construct's addon SDK, runtime scripts are based on the same APIs as used by Construct's scripting feature. The APIs specific to the Addon SDK can be found in the Addon SDK interfaces section of the scripting reference in the Construct manual. Further, all the APIs in the rest of the scripting reference section of Construct's manual are also accessible to addons.

Using modules

A major feature of the Addon SDK is first-class support for modules. These allow you to use `import` and `export` in your addon's runtime scripts.

Configuring use of modules

By default the Addon SDK samples are all already configured to use modules. However for clarity, to use modules an addon must be configured as follows:

1. The editor plugin must call `this._info.SetRuntimeModuleMainScript("c3runtime/main.js")` to set the runtime script file *main.js* as the main script. Then this is the only script that Construct loads.
2. All other runtime scripts must then be imported in main.js, e.g. `import "./instance.js";`
3. Add main.js to the `file-list` in addon.json so it works in developer mode

If an addon does not call `SetRuntimeModuleMainScript()`, then Construct automatically generates a main script that imports every runtime file. However this is not the right way to use modules if you want to do something like `import` a module in one of your existing runtime files.

Adding a module script

Assuming the addon is already configured to use modules - which all the Addon SDK samples are - then if you want to import a new module script named *mymodule.js*, these are the steps to follow.

1. Create the file *c3runtime/mymodule.js* and write an `export` in it.

2. In the editor plugin/behavior script, call `this._info.AddC3RuntimeScript("c3runtime/mymodule.js");` to add it as a runtime script, so the editor knows the file exists.
3. In addon.json add *c3runtime/mymodule.js* to `"file-list"` so the file is available in developer mode.
4. Import the new module in an existing runtime script. For example at the top of instance.js you could write: `import * as MyModule from "./mymodule.js";`

In short, when an addon is configured to use modules, all you need to do to add a new module script is to add it as a runtime file and then `import` it somewhere.

DOM calls in the C3 runtime

A major architectural feature of the runtime is the ability to host the runtime in a dedicated worker, off the main thread. In this mode it renders using OffscreenCanvas. With the modern web platform, many functions and classes are available in dedicated workers, and more are being added over time. Refer to this MDN guide on available APIs in workers.

Providing your addon's runtime calls only use APIs available in a worker, such as `fetch()` or IndexedDB, then it will not need any changes to support a worker. However if it does use APIs not normally available in a worker, then it will need some changes.

The principle behind making calls to the main thread in the C3 runtime is to split the runtime scripts in to two halves: the runtime side (that runs in the worker), and the DOM side (that runs on the main thread where the document is). The DOM side has full access to all browser APIs. The runtime side can issue DOM calls using a specially-designed messaging API built in to the runtime. Essentially instead of making a call, your addon can post a message with parameters for the call to the script on the DOM side, where the API call is really made. The DOM side can then send a message back with a result, or send messages to the runtime on its own, such as in response to events. The messaging APIs make this relatively straightforward. However one consequence to note is that a synchronous API call will become asynchronous, since the process of messaging to or from a worker is asynchronous.

Once this approach is used, there is no need to change anything to support the normal (non-Worker) mode. In this case both scripts will run in the same context and the messaging API will just forward messages within the same context too. Therefore this one approach covers both cases, and ensures code works identically regardless of whether the runtime is hosted in the main thread or a worker.

Using a DOM script

By default Construct 3 assumes no DOM scripts are used. If you want to use one, use the following call on IPluginInfo to enable one:

```
this._info.SetDOMSideScripts(["c3runtime/domSide.js"]);
```

Since an array of script paths is used, if you have a lot of DOM code, you can split it across different files. Don't forget to add these files to the file list in `addon.json`.

For documentation on the DOM messaging APIs, refer to DOMElementHandler (used in domSide.js), ISDKDOMPluginBase (used in plugin.js), and ISDKDOMInstanceBase (used in instance.js).

For an example demonstrating how to get started, see the **domElementPlugin** template in the C3 plugin SDK download. This demonstrates using the above APIs to create a simple `<button>` element in the DOM with a custom button text, and firing an *On clicked* trigger, with support for running in a Web Worker.

Supporting the debugger

Plugins and behaviors can display custom properties in the debugger by overriding the `_getDebuggerProperties()` method of the instance class. It should return an array of property sections of the form `{ title, properties }`, where `title` is a string of the section title and `properties` is an array of property objects. Each property object is of the form `{ name, value }` with an optional `onedit` callback. The name must be a string, and the value can be a string, number or boolean.

Editing properties

If an `onedit` callback is omitted, the debugger displays the property as read-only. If it is provided, the debugger allows the property to be edited. If it is changed, the callback is run with the new value as a parameter.

In many cases, editing a property does the equivalent of an action. To conveniently manage your code, you can implement actions as methods on your instance class, and call the same method from both the action and the debugger edit handler. As a public method is also accessible from Construct's scripting feature, it also makes the feature accessible from JavaScript code in projects.

Translation

By default, property section titles and property names are interpreted as language string keys. This allows them to be translated by looking them up in your addon's language file. Note property values do not have any special treatment. You can bypass the language file lookup by prefixing the string with a dollar character `$`, e.g. the property name `"plugins.sprite.debugger.foo"` will look up a string in the language file, but `"$foo"` will simply display the string `foo`.

The debugger runs in a separate context to the editor, and as such not all language strings are available. The language keys available in the debugger are:

- The addon name
- All property names

- All combo property items
- Everything under the "debugger" key

In general, if you need a language string for the debugger, simply place it under the "debugger" key, e.g. at "plugins.sprite.debugger.foo".

Sample code

The following code is used by the Sprite plugin to display its animation-related debugger properties. Notice how it uses language keys and calls actions to update properties.

```
_getDebuggerProperties()
{
	const prefix = "plugins.sample-plugin.debugger";
	return [{
		title: prefix + ".title",
		properties: [
			{name: prefix + ".speed",	value: this.speed,	onedit: v => this
			{name: prefix + ".angle",	value: this.angle,	onedit: v => this
		]
	}];
}
```