XS@TC53

Revised: December 10, 2019

Hello, my name is Patrick Soquet. At Moddable, I am in charge of XS, the JavaScript engine at the core of the Moddable SDK for embedded systems.

By the way, if you wonder, the name of the JavaScript engine, XS, refers to what is on the slide.

I often describe my work as two conflicting tasks. My first task is to implement JavaScript as defined by the ECMAScript Language Specification. My second task is to make JavaScript viable on embedded systems.

The tasks conflict because the specification is huge and because embedded systems are typically tiny, both in terms of memory and performance. But, for almost two decades now, I am thrilled by the challenge and I believe the solutions can help you build interesting products.

Today I will introduce you to XS by presenting the two parts of my work.

History

Before that, let me mention a few important events in the life of XS...

In 2006, Sony released the PRS-500, their first e-book reader, which used XS to run JavaScript for its user interface. XS has been used in other consumer devices but I cannot talk about most of them.

In 2014 and 2015, Marvell Semiconductor released Kinoma Create and Kinoma Element, two JavaScript powered construction kits for makers, both based on XS. Kinoma Element was our first micro-controller based device.

Also in 2015, XS was eventually open-source. For you it is a guarantee that you will always be able to get XS source code. For me, that was significant: I learned a lot from the source code of other JavaScript engines and I was happy for XS to become eventually available to everybody.

In 2017, Moddable was invited by Brendan Eich, the creator of JavaScript, to present XS to TC39, the ECMA technical committee that maintains and evolves the definition of JavaScript. Peter Hoddie demonstrated XS running JavaScript on micro-controllers. I told the story of XS and tried to explain how it works. As a result of our presentation, Moddable joined TC-39. Our role there is to advocate the ability to run JavaScript well on resource constrained hardware.

ECMAScript Language Specification

Let us begin with the first part of my work: to implement JavaScript...

You can browse the latest edition of the ECMAScript Language Specification at tc39.es/ecma262. The ECMAScript Language Specification is a dense, extensive and precise document. On this slide you can see the evolution of the document, from the initial 110 pages to the current 764 pages.

XS implements the whole specification. That is important! So I repeat: XS implements the whole specification. You do not want to build a product on a "micro", "nano", "pico" version of JavaScript because developers will be confused. XS allows you to remove parts of JavaScript, but that is your decision, based on the requirements of your product.

After the 6th edition, TC39 adopted a process based on proposals. Each proposal has a maturity stage. At stage 4, proposals are finished and will be published in the following edition of the specification.

Between editions, XS tries to implement some proposals at stage 3 and 4. For instance proposals like `dynamic import`, `BigInt`, `private fields and methods`, `top level await` and `WeakRef` are already implemented.

The precision of the specification really helped me a lot. On this slide you can see for instance an excerpt which defines the algorithm of the `RegExp` constructor. Of course you do not learn how to use JavaScript from the specification. But I learn how to implement JavaScript from the specification.

Test262

While I am working, I need to continuouly check if the implementation conforms to the specification... For that I rely on Test262.

Test262 is the official conformance test suite of the ECMAScript Language Specification. The test suite is divided into four parts:

- Language Syntax: 39,639 tests
- Standard Built-in Objects: 30,210 tests
- Internationalization API: 1684 tests
- Additional Features for Web Browsers: 1337 tests

The coverage is quite comprehensive. Every time there are new proposals, related tests are added to the suite. The results for various engines are available on test262.report

XS implements neither the Internationalization API (which is a separate specification) nor the Additional Features for Web Browsers (which the Moddable SDK is not). However I am pleased that XS was the most conformant engine at 91% on November 20. That can change every day of course!

I reorganize the chart to help you understand the results. If you are unfamiliar with JavaScript engines names, ChakraCore is Microsoft's, JavaScriptCore is Apple's, SpiderMonkey is Mozilla's, v8 is Google's. XS is in good company indeed!

Embedded Systems

Let us continue with the second part of my work: to make JavaScript viable on embedded systems.

The challenges of embedded systems are obvious: limited memory and limited performance. Compared to hardware that usually runs JavaScript on the client or the server sides, the differences are measured in orders of magnitude, not percentages: kilobytes instead of gigabytes, megahertz instead of gigahertz, single core instead of multiple cores...

On the slide there are data for three micro-controller based systems, my phone and my PC. I recommend that you look at the units!

The usual runtime model of JavaScript requires JavaScript engines to parse and run scripts and modules from their source code. That is what happens when you browse the web on your phone or on your PC. Browsers exploit all available resources to achieve their impressive speed, for instance by caching the shapes of objects or by transforming source code into machine code.

When resources are constrained, you need a different strategy...

Compile

XS always compiles modules or scripts source code into byte code. The XS compiler runs on a PC and only the resulting byte code is flashed into ROM where it is executed.

The part of XS that transforms source code into byte code and the part of XS that executes byte code are completely separate. In fact, most embedded devices cannot afford to transform source code into byte code themselves, so that part of XS is often absent, which saves some ROM at the cost of `eval`, `new Function` and `new Generator`.

The primary objective of byte code is of course to encode modules and scripts into something that is fast enough to decode. Most byte codes do not look at all like assembly instructions. Most byte codes directly refer to JavaScript expressions and statements.

The secondary objective of byte code is compression. A lot of byte codes have no values and take just 1 byte. Most byte codes with values have variations depending on the size of their values. A few examples:

- the integer byte code has variations for 1, 2 and 4 bytes values,
- the branches byte codes have also variations for 1, 2 and 4 bytes values, depending on how far the runtime has to jump.
- since JavaScript functions with more than 255 arguments and variables are rare, related byte codes have variations too, so most accesses and assignments take only two bytes.

Such variations may seem like a detail but every byte matters!

Link

While RAM is often extremely limited, embedded devices also have ROM which can be flashed from a PC. Not a lot of ROM, maybe a megabyte or two! The main strategy XS uses to run JavaScript on embedded devices is to run JavaScript as much as possible from ROM instead of RAM. Then XS uses several techniques to further reduce the amount of ROM it requires.

Having the byte code in ROM is not enough. There are the ECMAScript built-ins. There are the modules that applications need to do something useful, like a user interface framework, a secure network framework, etc. That means a lot of memory to define objects.

In the usual runtime model, when the application starts, built-ins are constructed then modules are loaded. The resulting objects are created dynamically in RAM. Even with the byte code in ROM, that requires too much RAM on embedded devices.

So XS allows developers to prepare the environment to run their application. On a PC, the XS linker constructs built-ins and load modules as usual, then save the resulting objects as C constants. Together with the C code of XS itself, of built-ins and frameworks, such C constants are built into the ROM image.

Run

When the application starts, everything is ready so the embedded device boots instantaneously. Nothing is ever copied from ROM to RAM so the application runs in a few kilobytes.

On the slide is a movie of the "balls" app running on a Moddable Zero. That is an ESP8266 at 80 MHz with 80 KB of RAM and 4 MB of Flash.

What about performance? I expect you will get enough demonstrations today to undestand it is no problem with XS. In fact, since JavaScript makes it so easy to experiment, applications can be tuned again and again to become faster and faster.

Notice also that XS does not force you to use only JavaScript. XS has a complete C programming interface. Critical section of your application can be implemented in C. To help you build your applications, the Modddable SDK proposes optimized modules, implemented in both JavaScript and C: I/O and network, graphics and user interface, etc.

Debug

Once you run your application, you want to debug it. To have a source level debugger is essential for any JavaScript engine. XS provides xsbug, a stand-alone debugger, on macOS, Linux and Windows.

xsbug is a TCP/IP server, waiting for the connection of XS engines. xsbug can debug several of them in parallel, each one being a tab in its window. XS engines can be TCP/IP clients directly, or indirectly thru a serial connection to the PC and a serial to TCP/IP bridge.

On the slide, you see xsbug paused at a breakpoint in the "balls" app. You can inspect local and global properties, step in and out functions, etc. Remember that while xsbug is on macOS, the XS engine is actually running on the micro-controller. Both communicate thru the serial connection.

Instrument

Besides debugging, XS also allows you to instrument your application. When resources are constrained, it is really useful to follow how your application is using them.

Firstly the XS engine instruments itself. For instance the XS engine reports the size of its heap and its stack, the number of garbage collections, etc. Secondly the instrumentation has a programming interface, so modules can also instrument themselves. For instance modules of the Moddable SDK report the number of sockets, the number of timers, the size of graphics buffers, etc.

Instrument samples are sent to xsbug periodically when the engine is running, or at every step when you are stepping in the debugger. xsbug displays the evolution of each instrument, as you can see on the slide.

Thanks

Of course nothing I talked about would have been possible without the help, inspiration and patience of my team. So I thank them and I thank you for your attention.