Introduction
The wearable category of digital devices has moved from future concept to market reality. The category is still immature, with product creators and consumers still actively exploring uses of these devices. One common indicator of a category which has not yet matured is the lack of standards, whether those are de facto standards like Windows on personal computers or true industry standards like HTTP and HTML on the web. There are many areas where standardization may apply to wearables including the digital hardware, energy, radio, and communication protocols.

This paper focuses on software standardization with the goal of enabling wearable products with built-in software that is reliable, cost-effective, secure, and built with modern technologies. Achieving this allows manufacturers to deliver wearable products to consumers with greater ease and predictability. The second goal is to allow users of wearables the freedom to install software from independent developers, just as they do on computers and mobile devices. Achieving this allows the wearables category to grow more rapidly by allowing third parties to add interoperability with their hardware, software, and cloud services, thereby giving consumers the choice of adapting their wearable hardware to their individual needs.

We propose the standardization of software for wearables by applying the JavaScript programming language. The success of JavaScript on the web, backend servers, and mobile is unquestioned. The language continues to evolve, carrying forward with it an ever growing ecosystem of skilled developers. This paper presents an argument for why standardizing at the language level is the right choice, technical data showing that JavaScript is a realistic choice, and a high-level outline of one way to apply JavaScript to wearable devices.
Re-introducing JavaScript

Because JavaScript is the foundation of this proposal, this section contains a brief re-introduction to JavaScript. Everyone is familiar with JavaScript because of its use on the web. The language has changed so much over its history – in capabilities, interoperability, performance, scope, and popularity – that a brief summary is helpful.

For twenty years, JavaScript has been the programming language of the web years. Designed and implemented in a mere six weeks, early versions of the language had some rough edges. Still, thanks to a solid foundation in established programming language practice dating back to LISP and continuous evolution of the JavaScript language, JavaScript has become the most popular programming language in use today. This popularity extends well beyond the original use of JavaScript on web pages. Today JavaScript is used as the language behind many web servers running the node.js framework. Many mobile applications, both for iOS and Android, are driven by JavaScript paired with an application framework. JavaScript is even finding its way into a handful of Internet-of-Things (e.g. embedded) devices. The reach of JavaScript continues to grow, as determined developers apply it to new computing domains.

The pervasiveness of JavaScript on the web has unquestionably contributed to its success. The productivity of developers working in JavaScript is also a strong contributing factor. JavaScript is a scripting language, a class of languages that provides higher level operations than C and C++. These operations generally allow programmers to achieve equivalent functionality in scripts using fewer lines of code than in C. Scripting languages also eliminate entire classes of mistakes that cause C programs to crash. Experienced programmers get more done with JavaScript. Less experienced programmers achieve results and stability far beyond what they could do in a language like C++.

The popularity of JavaScript has led to an ecosystem of developers that help one another solve problems. The answer to nearly any JavaScript language question is a web search away. There is an abundance of sample code, tutorials, and training available. All of this contributes to the continued adoption of JavaScript.

In its first decade, JavaScript was often criticized for being slow. The introduction of the V8 engine by Google changed that. By applying a wide range of sophisticated optimization techniques, V8 improved JavaScript performance by more than an order of magnitude. Ongoing optimization work by all the engines built for web browsers continues to improve performance. This performance leap allows JavaScript powered web pages to achieve previously unimagined levels of complexity and power, rivaling native applications. Further, it set the stage for the emergence on servers of Node.js, which is built on V8.

The success of JavaScript has led to the co-existence of several major JavaScript engine implementations. The most widely used are V8 from Google, Chakra from Microsoft, Nitro from Apple, and SpiderMonkey from Mozilla. Given the large size of the language specification (1500 pages) and the complexity of the performance optimizations it is reasonable to expect significant variation in the language itself, certainly in the details. Yet, there is almost none. Why?

Because JavaScript evolved on the web, there was no period of time when JavaScript programmers were writing code for a single JavaScript engine. JavaScript programmers expected a consistent language to be provided by every engine. This was not always achieved, but the need for interoperability led to JavaScript becoming a true international standard under the auspices of Ecma International. This effort is an on-going one, with Ecma's TC-39
JavaScript language committee meeting six times a year to work on improvements to the language and its specification. This standardization effort involves all the major JavaScript engine providers, developers of many different application frameworks, and companies whose business depends on JavaScript. This on-going effort has ensured JavaScript evolves to meet the shifting needs of the wide ranging domains making use of the language.

A clear, consistent, and complete language specification is necessary to achieve interoperability across engine implementations. However, it is not sufficient. Developers of engines need independent validation of their conformance to the standard. Therefore, the standardization work of Ecma has expanded to include a validation project known as test262. This test suite contains over 60,000 individual tests which measure an engine’s conformance with the intricate details of the language standard. Like the language specification, test262 is an on-going effort with new tests being developed to address newly identified issues and to cover edits and additions to the language specifications. The success of test262 in validating the entire language specification has delivered interoperability on the web, making the peaceful co-existence of multiple engines a reality. Further, it makes the launch of new engines far more feasible, as it provides a common way to validate them. Microsoft benefited from this when launching the Chakra JavaScript engine to replace the aging JavaScript engine used in Internet Explorer.

**Hardware diversity**

While wearables category is still taking shape, it is clear that the hardware used will remain diverse for the foreseeable future. There is no dominant manufacturer or product category on the horizon. This broad, and to some degree unpredictable, diversity of hardware capabilities presents a challenge for any standardization effort which, by necessity, must take a long term view.

Rather than focus on the differences, it is helpful to consider the similarities shared by the majority of wearable devices.

- Small physical size. The size tends to force reduced hardware capabilities in comparison to mobile devices. One impact of this is that user interactions change due to a smaller display size, or absence of a display.
- Battery powered. Wearables always operate on battery power, often a small battery due to the small physical size. This requires software to look beyond energy efficiency to becoming energy aware.
- Connectivity. Nearly all wearable devices have a wireless connection used to share sensor readings and receive notifications. There are many different wireless technologies with BLE, Wi-Fi, and LTE being the common.
- Resource constraints. The small physical size and low energy requirements tend to limit the available computing hardware including CPU power and speed, RAM, code space, and storage space.
- Responsive and reliable. Users expect the devices they wear to respond immediately and to have an even higher degree of reliability than their phones.
- Many sensor inputs.
- Cost remains important.

**Architectural layers**

The topic of software standardization is broad. It would be naive to suggest the full software stack can or should be standardized. Let’s consider each of the major software layers.
Operating system layer
At the bottom of software architecture stack is the operating system. Today wearable devices are built using a wide range of operating systems. This is unlikely to change, as the choice of operating system is based on a complex set of factors which include the underlying hardware, corporate preference, and operating system capabilities. Some wearable devices use large operating systems, typically a Unix or Linux variant such as Android Wear and watchOS. Devices built on these technologies tend to be more expensive and physically bigger, in part to accommodate larger batteries. Other systems use an RTOS, which is much lighter and energy efficient but also provides many fewer built-in capabilities, putting more of the software burden on the device manufacturer.

App layer
Apps are at the top of the software architecture stack. One intended outcome of software standardization is the potential for apps to be written in such a way that they can be used across wearable hardware from different vendors. Some of these apps will be very different from their app cousins on computers and mobile device due to the physical differences in the devices. The goal of standardization is to provide developers a consistent experience when building apps using tools that are common across wearable devices created by a variety of manufacturers. Doing this will allow app developers to re-use their knowledge and code on many different devices as they do today on computers, mobile, and servers. This in turn should lead to the development of a greater number of independent apps.

Application framework layer
Often apps are not built directly on the APIs provided by the operating system. Instead, app developers choose to build on an application framework. In doing so, an app developer is delegating responsibility for interactions with the operating system to the framework. App developers select a framework for many reasons, including a preference for the style of development, the quality of results, available features, or app portability. Because there are so many reasons for selecting a framework, it is no surprise that there a large number of app frameworks available for computer, web, and mobile development.
It is reasonable to expect that wearables, in time, will also have its own collection of application frameworks for developers to choose among. Consequently, there is no need to standardize the application framework layer. To do so would limit developer choice, and may be wasted effort as application frameworks come in and out of style quickly. The expectation that application frameworks will appear helps to focus the design of the lower software layers of the stack on being efficient hosts for app frameworks.

Runtime layer

The definition of the runtime layer used in this document is: the APIs available to an application running on the device. This includes portions of the operating system as well as libraries built on the operating system. The functionality available in a runtime varies widely. For example, a runtime to support a touch screen watch requires graphics and touch input APIs whereas a runtime to support a step counter requires neither graphics nor touch, but instead an accelerometer API. Portions of the runtime are often provided by the host operating system, such as memory management, whereas others are separate libraries built on top of the operating system (JSON parsing). The computer programming language or languages used for development on the device have their own distinct runtime requirements, the contents of which vary greatly. For C this is the “C standard library,” for Java it is the “Java Runtime Environment,” and for JavaScript it is “the host.”

The JavaScript language implies a set of requirements of the runtime. This in turn implies certain aspects of a runtime for wearables needs to be standardized as part of this proposal.

The minimum runtime required to run JavaScript code is designed to be independent of device and product category. The minimum runtime is insufficient for any particular product category. For wearable apps the runtime is incomplete as it does not include support for key capabilities of wearables including sensors, displays, radio communication, and energy management. Therefore this proposal goes beyond the minimum runtime for JavaScript to propose standardization of common runtime capabilities specific to wearable devices. These are discussed in the “Wearable runtime” section.

Language layer

It may seem unusual to consider the programming language among the layers in a device’s software stack. A common notion is that more-or-less any programming language can be used on any device, that the selection of language is a primarily a matter of developer preference. This is largely true on more capable devices, devices with effectively unlimited resources. The wearables category is now, and for the foreseeable future, defined by resource constraints. As a result, pragmatism suggests selecting a single programming language for apps to support well on wearables, rather than offering a variety of languages with uneven implementations.

The choice to focus on a single programming language may seem unusual. However, a cursory review of popular devices with flourishing independent app developer ecosystems shows that they tend to have a single preferred language. These include C on Unix/Linux, Pascal on the original Macintosh, C++ on Windows originally, C# on Windows today, JavaScript on the web, Java on Android, Objective C on IOS originally, Swift on iOS today, etc. Perhaps a single preferred language emerges because it allows the platform developer to focus their support and independent developers to most easily share knowledge and code.
If wearables is to standardize on a single programming language, the natural question arises as to which one. Programming languages may be divided into two groups. The first, native programming languages, generate output which is run directly (natively) by the microprocessor. These languages have the ability to run the fastest but are more difficult to learn and to use well. The second, scripting languages, generate output which is run by an interpreter (a kind of virtual microprocessor implemented in software). These languages are easier to learn and use safely, but are less efficient. Additionally, by separating the language from the underlying microprocessor through an interpreter, the code written is inherently more portable, allowing it to work on a wider range of devices. For the goal of reaching a large number of independent software developers, a scripting language is preferred. While there is some overhead, environments like the web show a scripting language engages the greatest number of developers.

Next the discussion turns to the selection of a scripting language. Let’s immediately set aside the idea of a new scripting language (WearableScript) as unnecessary due to the many existing scripting languages with active developer communities. The most widely used scripting language today is JavaScript, which is used for the web, backend servers, and mobile applications. The popularity of JavaScript brings with it an unmatched developer community. Further, over its twenty years, JavaScript has evolved into a thoroughly modern language with many language features to ease development.

JavaScript is designed for computers and mobile devices due to its start on the web. Running on smaller devices like wearables has not been a goal. By contrast the Lua scripting language is intended for embedding, with resource constraints considered from the start. If efficiency is the key concern, Lua has a clear advantage. However, the developer community for Lua is a small fraction of JavaScript’s, perhaps because the resource constraints have limited language features that ease development or perhaps because of JavaScript’s advantage as the scripting language of the web.

The work of Moddable Tech on the XS JavaScript engine demonstrates the feasibility of JavaScript on the hardware used for wearable devices. While perhaps not achieving quite the same resource use of Lua, nonetheless it is sufficiently efficient to make it the best choice for wearables when the benefits of JavaScript’s developer community size, programmer productivity, and proven standardization are also considered. The following section provides details on the XS engine as it relates to wearable devices.

**JavaScript on wearable devices**

The idea of JavaScript on wearable devices holds immediate appeal to many. But a fundamental question quickly arises: is JavaScript able to run well on wearable devices? This section addresses this question by reviewing technical details of the XS JavaScript virtual machine by Moddable Tech. It may well be that other JavaScript engines are also suitable for use on wearables. The goal of this section is to show that at least one implementation is feasible for wearables to validate the proposal to use JavaScript as the foundation for standardization in wearables. The XS engine is selected for this purpose because of the author’s experience.

XS is an independent JavaScript engine, written from scratch over a dozen years. Because XS has never been intended for use in web browsers, it differs from other JavaScript engines in many ways. The design point for XS is devices with significant constraints in performance, memory, and storage. XS development began during a collaboration with Sony, powering the Sony eReader product line. It has since been part of many consumer products from major consumer electronics companies including Palm, docomo, SoftBank, HP, VIZIO, and Whirlpool.
Because XS is designed for overall efficiency on constrained devices rather than high performance web browsing, the priorities of the implementation are distinct. Minimization of RAM use and ROM storage are at the forefront. This entails keeping the code and data size small, for both the engine itself and the scripts it runs. The XS engine uses two orders of magnitude less RAM and flash storage than JavaScript engines built for today’s web. While performance is not the first priority, it is always a consideration, and in some tasks XS exceeds the performance of web browsers, most notably in script start-up time.

First public demonstration

The most recent incarnation of the XS engine is the first to be truly practical on modern embedded microcontrollers. It was first publicly demonstrated at Ecma International’s TC-39, the JavaScript language standardization working group, meeting in May 2017 in New York.

The demonstrations primary run on an ESP8266 microcontroller based on an Xtensa CPU at 80 MHz, has 45 KB of available RAM (beyond what is used by the RTOS), 1 MB of available flash storage for code (including the RTOS), and Wi-Fi. The ESP8266 was selected as it is readily available and inexpensive (about $2 in volume). There is nothing particularly unique about the ESP8266 microcontroller, and the XS engine runs with equal efficiency on many other microcontrollers.

One additional demonstration at the meeting uses a Thunderboard Sense from Silicon Labs powered by a Gecko microcontroller that runs an ARM Cortex M4 CPU at 40 MHz, with 32 KB of RAM (total), and 256 KB of flash storage.

HTTP and TLS

The first demonstration, running on the ESP8266, shows communication with thePubNub realtime publish and subscribe API, combining JSON messaging over HTTPS using TLS 3.1 over Wi-Fi. The application is written in JavaScript as is the HTTP client and the bulk of the TLS implementation. Additionally, the demonstration runs attached to xsbug, the XS source level debugger, over USB allowing operation of the scripts to be paused, variables to be inspected, and code to be single stepped. With the full device firmware, XS engine, HTTP stack, TLS stack, and debugging support the device still has over 128 KB of free ROM.
The PubNub service is used for this demonstration because it is a popular publicly available commercial service intended for use by IoT products. Its use here shows that low cost microcontroller like the ESP8266 is able to communicate using modern, standard, secure network protocols.

Sensor hub

The second demonstration, this one on the Thunderboard Sense, is an interactive sensor hub connected to a low power display operating on a coin cell battery. The six sensors are sound level, accelerometer, light, humidity, temperature, and barometric pressure. Each sensor is displayed on a separate screen that shows both the sensor reading and an animated graphical representation of the value. The user cycles through sensor screens using physical push buttons. With only 256 KB to store the device firmware, XS virtual machine, scripts, and graphical assets just a few kilobytes of flash storage remains free.

Web Workers

The third demonstration, again running on the ESP8266, shows a subset of the HTML5 Web Workers API implemented on the XS engine. The main machine spawns four simultaneous workers, sending each several messages, at which point the worker exists, and a new one is spawned. The 45 KB of available RAM on the device is able to hold five simultaneous JavaScript virtual machines (main machine and four worker machines), each using about 8 KB. The demonstration spawns 38 virtual machines per second, including messages exchange with the main virtual machine.

The purpose of this demonstration is to show that a single JavaScript virtual machine can be quite small, and that creating a new virtual machine is fast enough, just a few milliseconds, that it can be performed on-demand. The ability to have several virtual machines or short-lived virtual machines has advantages for security and stability by isolating the access and lifetime of scripts.
Conformance with the standard

JavaScript is not only a programming language, it is an international standard that forms a significant part of the foundation of the web. The standard is implemented by engines from Apple, Google, Microsoft, and Mozilla to support their web browsers. The existence of several major, largely independent, implementations has created a need for strong conformance tools to ensure developers a single, consistent language for their work. The conformance test suite for JavaScript is test262, which contains approximately 60,000 tests as of this writing and continues to grow. Engines generally do not pass all tests, for a variety of reasons,

The XS virtual machine is regularly run against the 43,496 applicable test cases in the test262 conformance test suite. XS passes 99.5% of the applicable language tests and 99.1% of the tests for built-in objects. A summary of the tests262 results at the time of this writing are provided in Appendix A.

The results show the ability of XS to support the standard JavaScript language. The test results require some explanation, as the unique execution environment of micro-controllers imposes some limitations:

- Memory constraints place a practical limit on the number and size of objects that can be instantiated
- Script parsing is performed on a computer at build time, not on the embedded device while running. As a consequence, eval is unavailable and Function.prototype.toString returns an empty string. Both of these behaviors are permitted by the ECMAScript 2017 standard.
- Unicode is fully supported but accessing code points is not, which impacts a few string operations

When XS is used on a device with less constrained resources, it is able to support many of these features, including script parsing.

A developer deploying XS engine selects the JavaScript features to enable and disable to meet their ROM, and to a lesser degree RAM, budget. This is important for smaller devices. For example, the second demonstration which runs on the Gecko device with only 256 KB of flash storage includes only the built-in objects and methods the scripts use. The scripts do not notice (cannot observe) the absent objects and methods because they do not use them. For larger devices, certainly looking a few years forward, there may be no need to constrain the JavaScript language deployment.

As part of this proposal, we believe it will be helpful for the JavaScript language standard to consider small changes to benefit low resource target devices. Some proposals are already under consideration which could help including frozen realms (which focuses on security but has memory benefits as well) and weak references (which allow the script additional control over how memory is managed).

Wearable runtime

The JavaScript language, like most programming languages, defines important parts of the runtime, but not all the capabilities required by most applications. A set of runtime capabilities focused on the needs of developers creating applications for wearables is needed. These will augment the built-in objects of the JavaScript language.
Built on modules

The JavaScript 2015 Standard adds support for modules, a language level tool for packaging code into libraries for re-use and modularity. We propose to use modules as the mechanism to build the wearables runtime. Because modules are independent, they can be added or removed as appropriate for a particular wearable device. To expose their unique device capabilities manufacturers may extend the classes provided by standard modules or add new modules.

We recommend documenting the standard wearable modules separately from the language specification. By definition these modules address an audience of application developers focused on wearable devices, and may not have a broader use.

A model for this kind of domain specific JavaScript standard is ECMA-402, the ECMAScript Internationalization API Specification. Internationalization is an essential feature of web development, however with a breadth and depth that exceeds what makes sense for the core JavaScript language specification (ECMA-262). The Ecma TC-39 committee therefore choose to put the Internationalization work into a separate standard, one that builds on the base language standard. There are numerous other examples of standards on the web that build on the JavaScript language, many developed within the HTML5 WHATWG process.

The wearable standardization contemplated by this proposal will benefit from a close connection with ongoing language development. That will help ensure that the language continues to be feasible on wearable devices, and perhaps even adds features to benefit such devices. As a consequence, it is preferable to do this work within Ecma International.

**Note:** JavaScript modules always execute in “strict mode” a subset of the language which eliminates certain older features of the language that have proven troublesome for developers and engines. We recommend all JavaScript code on wearables be contained in a module, ensuring new devices need not support deprecated language features.

Runtime defined by JavaScript

Many capabilities commonly considered part of a runtime are provided by the JavaScript language and its built-in objects.

- date
- time
- Unicode character encoding
- string operations, including regular expressions
- binary data operations
- math intrinsics
- arrays
- maps and sets
- memory manager (garbage collecting)
- deferred execution and async operations
- shared memory buffers with atomic operations
Modules for wearables

The question next arises as to the modules that should be defined for wearables in a standard. We recommend that the effort initially focus less on being a rich suite of modules akin to an operating system and instead focus on a simple set of essential low level modules. Here “essential” is taken to mean capabilities that could not easily be implemented as a JavaScript module for a wearable device. In other words, they are capabilities which are tightly bound to the underlying hardware or operating system for functional or performance reasons.

This proposal does not attempt to define a minimum suite of modules for wearables. That is future work for a collaborative effort. However, examples of modules that are, and are not, contemplated as necessary for standardization by this proposal are useful for understanding the intended scope of this proposal.

- file system
- energy management
- radio connections (Wi-Fi, BLE)
- network socket (TCP, UDP)
- cryptographic primitives
- audio playback and capture

This proposal recommends minimal standardization in certain areas often considered part of a device’s runtime.

Network protocol APIs — The web and particularly Node.js runtimes have shown that there are many unique useful APIs to work with the same network protocol. The API style is often linked to priorities in the protocol implementation, whether that is performance, simplicity, scalability, or adherence to a particular programming methodology. Any standardization will necessarily favor some of those over others. Instead, network protocols can be built efficiently as scripts on a standard network socket objects allowing the wearable developer community to create focused solutions for specific needs.

Sensors — Clearly some sensors will need to be standardized, for example accelerometer, ambient light, and GPS. It is tempting to standardize many more. How most sensors will be used by applications is not always clear, which makes design of an appropriate API challenging. We recommend the initial work focus on those sensors which are well understood from their use in other environments, such as mobile. Device manufacturers will create APIs for these sensors, and in time these will develop into APIs that are sufficient mature for standardization.

Graphics — The visualization capabilities of wearables ranges from a single LED to small displays capable of rendering mobile-style user interfaces. A single graphics API is unlikely to meet the needs of the many kinds of mobile devices. Each category of wearable with display capabilities will require a rendering API. This should be done in a minimal way, however. There are many examples of extensive graphics APIs of which developers use only a tiny subset. In the constrained environment of wearables, the added weight of unused or seldom used capabilities is unsustainable. As a counter example, the team behind Moddable Tech has found a simple set of three primitives — fill rectangle, draw bitmap, and draw text — is sufficient to build surprisingly wide range of user interfaces.
Security and privacy

The topics of security and privacy are important, all the more so with personal devices like wearables. This proposal has no intention to standardize security or privacy policies, in part because the topic is so broad that achieving consensus on the goals alone would be challenging.

That said, any effort at standardization today must be aware of the security implications of its work. This standardization work should not enlarge the attack surface of the device, and ideally will help shrink it. The recommendation to use the JavaScript language arguably does shrink the attack surface. Because of its use on the web, the security of JavaScript has been carefully studied. Newer language capabilities have been designed with security in mind, and features have been added to the language to address security concerns. Certainly shifting the balance of code for a wearable device from C or C++ to JavaScript lowers the chance of mistakes that create security holes.

A further reduction in the attack surface is achieved by executing JavaScript code in an isolated sandbox, which does not have full access to the capabilities of the underlying hardware. These sandboxes can be short lived, preventing the accumulation of state which may unintentionally leak information or security credentials.

There is a need for modules that facilitate implementation of good security and privacy models, such as cryptographic primitives and the TLS protocol. Consistent with the overall spirit of this proposal, those modules should be general purpose tools for building secure communication, secure storage, secure execution, ensure user privacy rather than specific policy requirements for device manufacturers conforming to the standard.

Tools

The experience developing software to run on wearable devices using JavaScript will differ from building JavaScript software today for most developers creating applications for browsers, servers, and mobile. In all of these domains, the majority of work takes place on the developer’s computer which is able to run a web browser, web server, and mobile app simulator. A simulator for wearables will be realistic for some scenarios, but when working directly with the unique features of the wearable device, it is necessary to run and debug software directly on the wearable device.

The software for the wearable device will be built on the computer and transferred to the wearable device. This requires a tool chain to compile the JavaScript source code, link the code into an application, deploy to the device (over USB, Bluetooth, Wi-Fi), and debug the JavaScript running on the device. In addition, because wearables will be resource constrained in various ways, tools to measure resource use and performance will be essential.

This development process is certainly more complicated than the minimal web browser workflow of editing a text file and reloading a page in the browser. It is of a similar degree of complexity as mobile application development in JavaScript. This workflow does come with some benefits as it allows other tools to be run on the JavaScript code as part of the deployment process. These include tools to optimize the code in some way, linters (which detect problematic programming constructs), and transpilers, which many developers use to convert languages like TypeScript and CoffeeScript to JavaScript. All of these tools could be integrated into the wearable development workflow, giving developers more options for their product development workflow.
Appendix A – test262 results summary

These results are for the ECMAScript 2017 standard, completed in June 2017. The XS implementation used is from October 2017.

Language: 99% 19176/19269 (4072)
100% 231/231 arguments-object
100% 202/202 asi
100% 192/192 (15) block-scope
100% 38/38 comments
100% 90/90 computed-property-names
100% 30/30 destructuring
100% 54/54 directive-prologue
97% 250/256 eval-code
100% 3/3 export
99% 8503/8545 (1766) expressions
99% 281/283 function-code
100% 85/85 future-reserved-words
76% 50/65 global-code
100% 19/19 identifier-resolution
100% 267/267 identifiers
100% 4/4 import
100% 50/50 keywords
100% 118/118 line-terminators
97% 464/476 (6) literals
100% 269/269 module-code
100% 22/22 punctuators
100% 53/53 reserved-words
100% 22/22 rest-parameters
99% 7576/7588 (2285) statements
100% 94/94 white-space

Built-ins: 99% 25210/25416 (497) built-ins
99% 5106/5118 Array
100% 160/160 ArrayBuffer
100% 46/46 ArrayIteratorPrototype
100% 30/30 AsyncFunction
100% 220/220 Atomics
0% 0/0 (31) BigInt
100% 96/96 Boolean
100% 790/790 DataView
100% 1376/1376 Date
100% 106/106 decodeURI
100% 106/106 decodeURIComponent
96% 56/58 encodeURI
96% 56/58 encodeURIComponent
100% 74/74 Error
100% 16/16 eval
89% 755/845 (8) Function
100% 40/40 GeneratorFunction
100% 114/114 GeneratorPrototype
100% 58/58 global
100% 12/12 Infinity
100% 32/32 isFinite
100% 32/32 isNaN
100% 8/8 IteratorPrototype
  90% 218/242 (4) JSON
100% 283/283 Map
100% 22/22 MapIteratorPrototype
100% 542/542 Math
100% 12/12 NaN
100% 146/146 NativeErrors
100% 498/498 (1) Number
100% 6128/6128 (2) Object
100% 84/84 parseFloat
100% 118/118 parseInt
100% 622/622 (16) Promise
100% 467/467 Proxy
100% 274/274 Reflect
  97% 1544/1586 (433) RegExp
100% 374/374 Set
100% 22/22 SetIteratorPrototype
100% 112/112.SharedArrayBuffer
  98% 1965/1993 (2) String
  85% 12/14 StringIteratorPrototype
  97% 136/140 Symbol
100% 26/26 TypeError
100% 1250/1250 TypedArray
100% 728/728 TypedArrays
100% 12/12 undefined
100% 176/176 WeakMap
100% 150/150 WeakSet
Embedded solutions lean towards old school technologies… which is a terrible way to attract developers.

Overall problem…. absolutely nothing is standard today in wearables making it impossible to achieve critical mass. Android Wear is big. watchOS is closed. There are no other well established platforms a manufacturer can adopt to tap into third party developers (first to help build their products and to provide an ecosystem of “apps”).

**Software challenge**
Constraints tend to favor highly customized design and software as there is no overhead (physical or resource) for much generality.

Software is expensive to design and develop. That favors re-use which favors standards.

The product manufacturer cannot build all the application software consumers expect. Integration with different physical devices, mobile applications, cloud services, etc needs to happen organically by those with a vested interest.

**Language**
So…. what language? ECMA… so ECMAScript… which seemed insane until recently. But it is reasonable. Give some XS numbers on RAM and ROM and conformance and performance (60 FPS animations… TLS…)... boot time… Gecko work


Tap into knowledge of web and node developers, rather than building a new platform from zero.

ECMAScript is built with security in mind. Web browsers obsess about this and the language is part of that. Recent improvements make security stronger (modules, frozen objects) with more coming (private field syntax in classes).

Independent of OS and CPU. Which allows it to work across generations of devices, device categories, and silicon families, and manufacturers.

**Runtime**
The language gives a basic runtime. Time. But not timers. Async operations. But not network. Etc. “Standard C library” isn’t there. Stuff beyond that (e.g. screen, audio, sensors, networking above a socket, resource management) really isn’t there.

APIs are needed. Fortunately, ECMAScript modules are a powerful, modern way to package suites of functionality. See ECMA-402 for internationalization. Define new runtime modules that are needed for wearables (accelerometer, etc). Borrow from the web where possible, but with care as the runtime was design for different purposes. Minimum standard is small. Subclass to
add features so devices don’t carry excess baggage. Allows nonstandard enhancements in an opt-in manner.

Language is happy with low level capabilities now…. shared memory and atomics (threading) and data view and typed array (binary data) and asm.js (for crazy speed, still portable).

Localization already taken care of.

Pebble… which invented the smart watch used a simpler JavaScript to power their watch faces…. developer plugin…

May not include parser on device… but could include debugger.

**Extensions**

Modules are the extensibility mechanism. Apps can be modules. Drivers. Plug-ins. Etc.