Scripting Languages in IoT: Challenges and Approaches

Paul Sokolovsky, Linaro

Benefits of Scripting Languages

Perl, PHP, Ruby, Python, JavaScript - Very High-Level languages

- Easy to learn (*many* people already did that)
- Powerful and concise (increases productivity)
- Vast, easy to use 3rd-party libraries (even more productivity)

For some segments, like Web Development, “LHLLs” like C/C++ aren’t practical and VHLLs largely displacing “MHLL” like Java/C#.

Scripting Languages in IoT

- Try to capitalize on the above benefits while developing for IoT?
- Talking not just about “server” and “gateway” sides, but also “device” side (aka deeply embedded).
- But do scripting languages allow fine and detailed control of scarce resources on such devices?
- Can we run a scripting language in a dozen of KBs of RAM and few hundreds of code (ROM) at all?
- Turns out we can, and there’s even a choice.

Meet the Contenders

- MicroPython, Python3 subset implementation
- JerryScript + Zephyr.js, JavaScript5 + Node.js subset implementation

Trivia/Vanity

https://github.com/micropython/micropython	https://github.com/jerryscript-project/jerryscript https://github.com/01org/zephyr.js
Github stars: 4310	Github stars: 2140 + 43
Github forks: 986	Github forks: 236 + 25
Commits: 7312	Commits: 2329 + 713
Github contributors: 150	Github contributors: 48 + 16
Github issues/pullreqs: 1407 / 1430	Github issues/pullreqs: 464 + 210 / 1097 + 464

JavaScript - The Golden Hammer

Appeared 1995 as a browser/web pages scripting language, remained dormant for awhile and then infamous for small API lacking many features, and cross-browser compatibility issues, finally blossomed with Web 2.0.

If it’s good for web pages, it must be good for everything, hence Node.js (“server-side”, largely extended API), and it’s only natural to want to see it in embedded. Except…

"I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."

- Abraham Maslow (of Maslow's hierarchy of needs fame)

Python - Serial #2

Appeared in 1991 and relatively quickly became “multiparadigm”, used in various areas of CS/IT, with “strongholds” in scientific computing, system administration, web development, and education. Probably never was #1 on any language popularity list, but usually in top 5 (sometimes #2).

Many languages implementations (PyMite is one of the first scripting languages implementations targetting microcontrollers).

Problems: Python2/Python3 split.

Myth - I’ll take that few-MB tarball and put it on a chip

Both JavaScript and Python developed vast module libraries for desktop/server usage. But that’s it - they are written for desktop and servers. There is a huge difference in resource scale between those and a deeply embedded devices. Laptop this is presented from has 16GB of RAM. 16KB is still middle-line for current MCUs.

16GB / 16KB = 1,000,000 = 10^6 = million times difference

The most useful software for such devices would need to be written from scratch, not ported.

But, the smaller the language, the easier ...

However, the smaller a language, the smaller its implementation is (VM size, standard types/library size), which is quite beneficial for resource-constrained devices, and JavaScript has an edge here. Indeed, JerryScript is a full implementation of ECMAScript5 standard (which is 2 generations behind the current standard).

Python’s motto is “batteries included”, hinting at “unalienable” and wide-coverage standard library. The language itself is feature-packed too. So, implementing “of all Python” for deeply embedded devices doesn’t make sense. MicroPython implements subset (say, 80%) of Python3.5 language, and even smaller subset of the standard library.

… also, trickier it gets

But being small is a drawback too, let’s see this by an extreme case of a B*F* language, which effectively implements an abstract Turing Machine. VM for it is very small, but writing application in it is very challenging! So, this problem does affect smaller-scoped languages: smaller core means less features, more to implement yourself, then many people reinvent the wheel, then it gets harder to select what *you* should use with “there’s more than one way to do it”, etc., etc.

… and even trickier it gets

23 Mar 2016

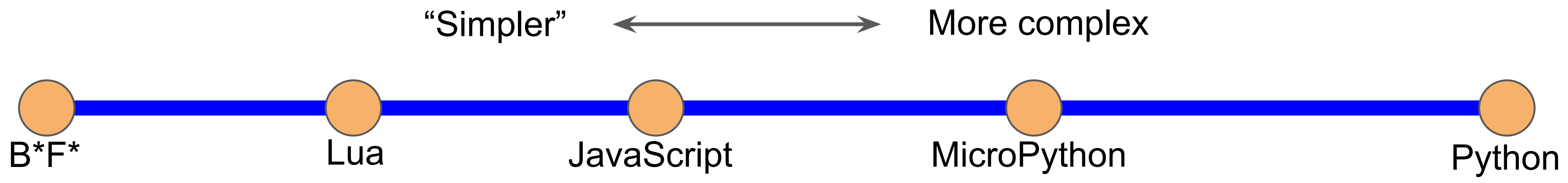
“Disgruntled developer breaks thousands of JavaScript, Node.js apps”

http://www.zdnet.com/article/disgruntled-developer-breaks-thousands-of-javascript-node-js-apps/

“Thousands of Node.js programs rely on the 17-line 'left-pad' npm package to function.”

Finding a sweet spot

Being small for a language is not just benefit, it’s a risk and liability: of lacking functionality, of need to (re)implement it, of many people doing that leading to a mess, then someone wanting to clean it up and breaking it all. Yet being big is ruled out. Finding a sweet spot would be good.



(Not up to scale!)

With standard library, it becomes all the same

While JerryScript implements all of ECMAScript5, ECMAScript5 itself is pretty bare development environment. Ahem, a Turing Machine, not even input/output capability. Projects like Zephyr.js take it from there, and of course, they try to implement further APIs, like Node.js and various WHATWG standards. So, let’s look at Console.log(). https://console.spec.whatwg.org/#formatting-specifiers

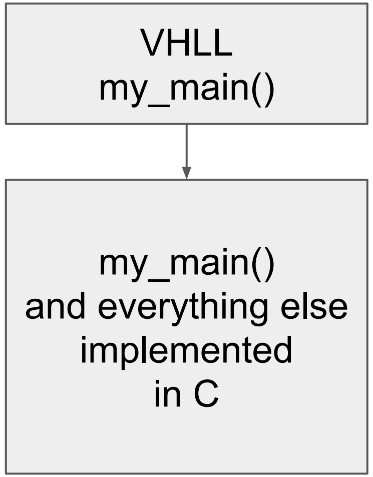
```
console.log("%d", 1)
```

Zephyr.js doesn’t support that (maybe yet).

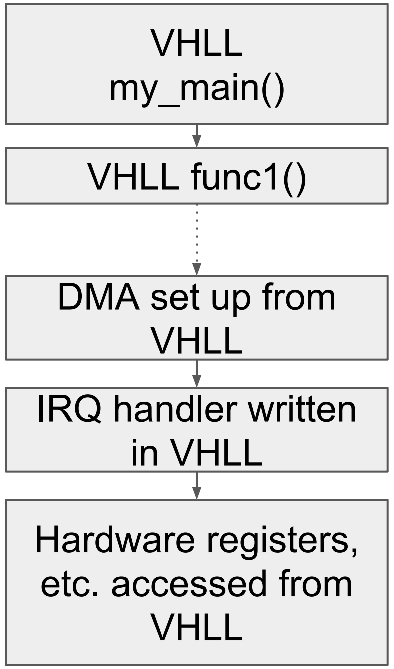
Bottom line: Any embedded lingo taking full-fledged desktop/server language as a base would hit “a subset issue”.

What do develop a language for? (1/3)

Two extremes:



(Degenerate case)



(MicroPython way)

What do we develop language for? (2/3)

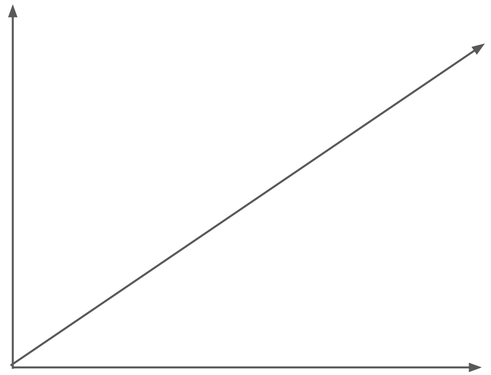
MicroPython: Be general-purpose, “Turing complete” language. Start from the simplest things, and develop bottom up. We want anything to be possible to develop in MicroPython, and eat our dogfood on that way but trying to develop as many libraries as possible in Python, and only later optimize to C what makes sense.

Zephyr.js: “Start from the middle” approach, then grow (both ways hopefully). Example: OCF connectivity module was added before generic socket module.

What do we develop language for? (3/3)

The aims which turn out to be pretty orthogonal

Develop really great support for a particular hardware, allow to develop apps taking the most out of that hardware (aka specific-product development). Downside: hardware gets old, project becomes useless.



A real viable project.

Develop common paradigm and API, covering baseline and the most important features across various hardware, aka framework and ecosystem development. Downside: can't get all the goodies of a particular hardware.

Targets support

MicroPython supports many hardware targets in-tree and even more out of tree. But high attention is paid to the core and consistency between different ports. When Zephyr RTOS port was started, it from beginning was considered generic, hardware-neutral (that’s what an OS for, even if it’s RTOS, right?)

Zephyr.js so far officially supports just 2 boards: Arduino 101 and FRDM-K64F. Arduino 101 is the primary target with really great support (BLE, various hardware interfaces, etc.), FRDM-K64F lags behind. Initially there were various hardcoded assumptions precluding to use other Zephyr boards, but a patch to enable at least generic Zephyr GPIO was contributed.

Linux port

Both MicroPython and Zephyr.js have Linux (POSIX) port.

POSIX port is of utter importance for MicroPython, because that’s what runs the regression testsuite by default. It’s also full-fledged, well supported port on its own - MicroPython targets not just MCU systems, but also small Linux systems, like OpenWRT, etc. (Also it targets desktop, cloud, mobile, etc. - we’re just short-handed somewhat.)

Zephyr.js’ Linux port is somewhat underloved - doesn’t even quit on app termination, has only basic functionality. (Mind that the project is pretty young!) Hopefully it will be developed further, as it’s really useful for testing (that’s on my TODO too).

Testing

All of JerryScript, Zephyr.js and MicroPython have CI integration using Travis CI. MicroPython also can run its 97% coverage testsuite conveniently on a host (using POSIX port) or on an embedded device (via serial and other connection methods).

JerryScript also has a comprehensive testsuite running on a host, though coverage isn’t known, not there’s on-device running support.

Zephyr.js testsuite is in the formation stage, with a couple of dozens of tests, some of which require manual interaction with a device, some on Linux port, some not, and they aren’t yet categorized per which are which or allow convenient local running during development.

Default development environment

MicroPython has a default interactive prompt with autoindenting and autocompletion.

Zephyr.js default modus operandi is to produce firmware image with a bundled application which can be deployed and application run.

There’s a separate interactive mode for Z.js called “ashell”, but so far it seems to mostly support Arduino 101.

MicroPython can easily bundle a user application with firmware which will run on boot too. Both support both textual source and pre-compiled bytecode for applications.

On to specific language details, including technicals

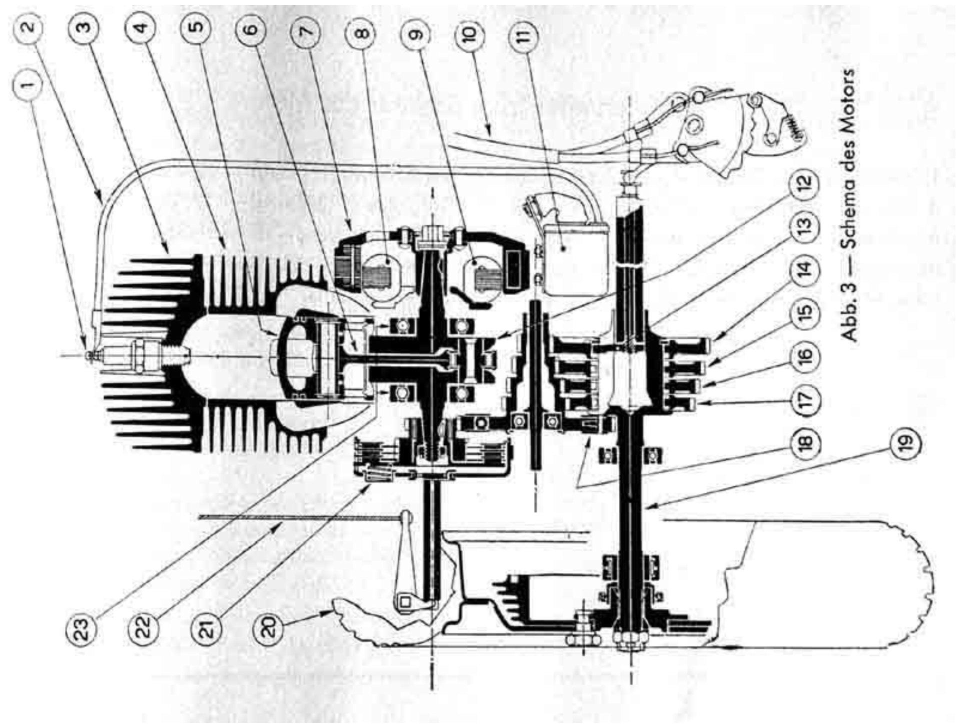


Abb. 3 — Schema des Motors

Strict vs weak typing

All (most) scripting languages are dynamically typed, but Python is strictly typed in addition to that.

~~Array[1.0]~~ - Error; ~~“10” + 1~~ - Error (Result in JS: “101”, in PHP: 11)

Python has explicit integer vs floating-point, so you’re always at control what your app uses (which is important for low-resource embedded systems).

JavaScript has only one numeric type, which is floating-point, and to support 32-bit integer precision, have to be double. Indeed, JrS *public* API:

```
double jerry_get_number_value (const jerry_value_t value);
```

(MicroPython can be built without floating-point support.)

Hierarchy of “variable strictness”

Lua: print(my_mispelled_var) - no error, valid (though special) value (nil)

JS: Lua’s case error, but: obj.mispelled_prop - no error, valid (though special) value (undefined)

Python: Both Lua’s and JS’ cases are errors, but: my_mispelled_var = obj.mispelled_prop = 1 (can assign to “wrong” name, new var/prop created)

Java/C/C++: All of the above are errors. Object types need to be “declared”, objects and variables - “defined” to be accessed/assigned. Price: “COBOL fingers”.

Containers

JS native container is “object”, which is used to host objects with prototype inheritance. With some boilerplate code can be used as a dictionary/mapping type. There’s an Array type, but per ES5 spec, it’s implemented in terms of object, with numeric indexes converted to string keys. JrS is faithful of that implementation. Recently, ES6 feature, TypedArrays were contributed to JrS, which represent true arrays, but only for numeric values.

Python’s strictly typed nature calls to inventory of well-defined container types without fuzziness: its dictionary is just a dictionary, its list guarantee O(1) access, its numeric array were part of the core language for a long time, and its objects are classical objects, literally (but used dictionary as attribute store).

Memory Management

MicroPython is solely garbage collected language, because that’s the most memory efficient way. JrS uses combined GC + reference counting (that’s scheme used by “big” Python for example). Some objects are reference counted, some not. To save bits, less than 32 bits are used for counters, and if you try hard, they can overflow. JrS pre-1.0 used 16-bit compressed pointers, which is great memory-saving measure, but limited heap size to 512K. v1.0 added configuration for full 32-bit pointers. There seems to have been another advanced optimization, called chunking, bit it was dropped. In MicroPython, we’d like to try compressed pointers and chunking some day, but don’t haste at all, working on more practical features and stabilizing the core, because maintenance cost for these optimizations are quite high.

Memory Management - Challenges

Scripting languages are inherently RAM-bound and produce high memory traffic. If simple heap allocation techniques are used, this can lead to severe fragmentation and application faults. And indeed, both MicroPython and JerryScript use simplistic memory management schemes due to overall resource constraints. In MicroPython, we would like to develop compacting garbage collector and perhaps even real-time compacting GC, but those are resource-intensive tasks (at least a man-month for 1st and up to a man-year for 2nd), so we're looking for stakeholders and sponsors. In the meantime, we're trying to make "peephole" style optimizations to avoid unneeded (re)allocations and provide allocation-free APIs and operations (Python support those natively!). This work might become largely superseded by compacting GC, but that yet need to surface and will be disruptive change, while no-alloc optimization are local and incremental and can still provide benefits given uPy's "everything can be done in Python" approach, which includes interrupt handlers written in Python, no-jitter realtime operations, etc.

Hardware APIs

Not inherited from upstream languages - need to define themselves, and that's ha-a-a-rd. Like, hard to define general, functional, extensible, concise, beautiful API. Defining "something" is oh so easy, indeed, every embedded lingo impl does that (reinventing the wheel).

Common in uPy and Z.js: object-oriented approach to GPIO objects (vs pure functional in many other implementation). But: I heard that's changing in Z.js.

```
pin = Pin(port, pin_no, Pin.OUT); pin.value(1)
```

vs

```
val = gpio_get(pin_no); gpio_set(pin_no, 1)
```

Conclusions (on approaches)

- JrS implements whole EcmaScript5 (that’s not much!), while uPy seeks to find a sweet spot of support subset of Python3 (mostly there).
- Both languages are and will be subsets of “desktop/server” functionality (can reuse knowledge, but also need to learn new tricks).
- uPy is strictly typed and have good inventory of strict types, JrS is weakly typed with JS object being primary fuzzy type, arrays implemented on top of it, ES6 typed arrays contributed recently.
- uPy favors easy to use development environment, but deployment is right next in row. Z.js puts “production first”, but development comfort - well, hopefully it’s coming.
- Memory management remains next big challenge (but that’s true for any system with dynamic memory allocation, even written in C/C++).

Conclusions (general)

- JavaScript is NOT automagically the only and right choice for deeply embedded IoT, both the language and its current implementationS have their share of issues.
- Fortunately, there’s an alternative - MicroPython ;-).
- Don’t trust my word, give a try to both yourself!
- Both need contributions, which goes beyond working on implementations themselves, but e.g. trying them for your applications and giving a feedback.
- If you’re still into C, well, bear your cross.
- If you like MicroPython, tell my boss, so I can work more on it ;-).