Quick intro - Why scripting languages in IoT (why not?)

- Usual claim: Scripting languages are slow
- Counter-claim: Well, scripting languages are FAST
- Not talking about run time now (you can blink an LED in a scripting language, no worries)

**Development Time!**
Quick intro - Why scripting languages in IoT (why not?)

Example:

- Team A uses a scripting language to prototype a product. They discard prototype 1, then discard prototype 2, then prototype 3 shows viability and they decide to ship it to customers to get money flow, while working on optimizing the product to get that 10-years battery life (hard even in C).

- Team B uses ol’ good C. All this time they keep working on prototype 1, which, as we know, yet to be discarded, twice.
Quick intro - Philosophy of “IoT” vs “Embedded”

- “Embedded” or “WSN” times: Industry to Customer: We have a technology. You need it.
- “IoT” times: Customer to Industry: We have a technology, we need it to run with that thing (pointing at a small chip).

Large share of Internet now runs applications developed in scripting languages (Python, Ruby, JavaScript, etc.) So, it’s a natural direction of interest to apply same workflow to IoT, down to “deeply embedded” devices.
Benefits of scripting languages

Initial acquaintance/learnability

- In general:
  - Scripting languages are (usually) easy to learn
  - A lot of people already know scripting languages
  - Scripting languages oftentimes share similar paradigm, so if one knows one language, one can easily learn another
  - Scripting languages are often interactive and interpreted == “plug and play”

- For embedded hardware:
  - There’s saturated market and competition. Many customers evaluate several products as a base for own designs. Having an option for customer to start and evaluate product easily is a good bonus.
Benefits of scripting languages

Rapid prototyping

● With a scripting language, it’s easy to make a quick prototype of/for some application for internal feasibility study, customer presentation, etc.

● This applies to both product-level development (e.g., make a prototype of smart lamp) and hardware evaluation (e.g. quickly play with various accelerometer types).
Benefits of scripting languages

Time to market

● If a prototype in a scripting language shows viability, it even can be developed further and shipped to customers
● “Higher-hanging fruit” than acquaintance/rapid prototyping usage
● Requires tooling around scripting language: unit/integration testing support, almost certainly byte compiler and perhaps IP protection support, robustness framework and crash reporting, OTA updates, etc.
● “Big scripting languages” have many of that features, so they “just” need to be adapted for IoT devices. Other features need to be developed.
● Scripting languages for IoT are at the beginning of their march, so support for above may be sparse or non-existent. As (if) they show viability, tooling may appear and “time of market” perspective become realistic.
Benefits of scripting languages

Easy extensibility by a user

- Smart devices are all about customizability by a user.
- You can ship a toolchain, SDK, and your app as object libraries, and let user develop “plugins” in C.
- Or, can let a user to upload/paste a script to send a tweet every time sun rises.
- What’s easier to implement and support for a vendor, and user for a user?
Benefits of scripting languages

Security of extensibility by a user

- Extending previous point, scripting environment provides a natural sandbox for extension code, to maintain product integrity and protection against attack vectors.
Benefits of scripting languages

Educational/Maker markets

- Success of Arduino and it becoming a de-facto standard in some areas, e.g. form-factor/connector layout, and IDE/unobtrusive C API show that “maker” market can’t be ignored.
- Projects like Raspberry Pi or BBC micro:bit show viability of “educational” markets too.
- Scripting languages are big benefit for both markets (e.g. Python API is an official hardware API for Raspberry Pi).
- Generally, scripting languages expansion to embedded devices started in maker community. It may become, or already becoming the next big thing after Arduino.
Drawbacks of scripting languages

- Require more resources (but minimum bar for a scripting language is just an average MCU now - 128K ROM / 16K RAM)
- Slower (but for many usages, fast enough)
- From the above, not as energy-efficient (but less important for always-powered or 99.9% in-sleep devices)
- There’re different scripting languages ;-) . Some may be more, or less familiar or liked by someone. Some may be more, or less suitable for hardware control and constrained resources.
- Look, many slides for benefits, and only one for drawback. Scripting languages must be good!
- Actually, drawbacks are well understood and obvious. Scripting languages have a good niche in rapid prototyping, user extensibility, and amateur markets, other usages are emerging.
History - Nothing is new

- Interpreted execution has a long history in deeply embedded usage, or in usage which would be considered such by current standard.
- P-Code and stuff.
- Forth bytecode has a long history of usage in very constrained devices.
- 8-bit console games oftentimes used bytecode produced from a DSL.
- Driving force is usually or primarily saving code space, not making it easier to program: Forth can be classified as a machine-independent stack assembler, game DSL are constrained, not general-purpose languages. The idea is that bytecode can be made more compact than native machine code, and have higher-level instructions saving even more.
History - Research projects around Scheme

● Marc Feeley, Danny Dube, University of Montreal, University of Laval

● 1996: BIT: “Scheme for microcontrollers that includes a real-time garbage collector” “We demonstrate that with this system it is clearly possible to run realistic Scheme programs with as little as 3 to 4 KB of RAM. Programs that access the whole Scheme library require only 13 KB of ROM.”

● 2003: PICBIT: 256+ bytes RAM, 22KB ROM, non-realtime GC

● 2003: PICOBIT: 100+bytes RAM, 4KB ROM, non-realtime GC
Modern Times - Python

- Python-on-a-Chip aka PyMite is a pioneer of open-source scripting languages for MCUs, development started 2002, first release 2003-03-18, SCM history dates back to 2006
- tinypy - first commit 2008-04-04
- MicroPython - first commit 2013-10-04
Modern Times - Lua

- eLua, first commit 2008-07-29
- http://www.eluaproject.net/doc/master/en_arch_ltr.html: “... relatively high Lua memory consumption at startup. It's about 17k for regular Lua 5.1.4, and more than 25k for some of eLua's platforms.” [With LTR patch: 5.42KB]
- Vanity: 429 stars on github, 1385 commits.
- Notable users: NodeMCU project for ESP8266
Modern Times - JavaScript

- Ducktape, first commit 2013-01-22, intended for embedding into (desktop) apps
- Espruino, first commit 2013-09-26
- V7, 2013-12-13, GPLv2
- JerryScript, 2014-07-01
- KinomaJS (xs6), 2015-03-01, (Linux-level devices in open-source release?)
Examples of commercial “off shelf” systems

- Synapse Wireless SNAPpy - Python, since ~2008
- Electric Imp - Squirrel lang, since ~2011
- Kinoma series (Marvell), since 2014 (Create, Element, HD) (Dates back to Kinoma, Inc. from 2002)
Categorization of programming languages

- Static vs dynamic typing (of variables)
- Strict vs weak typing (of values)
- Automatic vs manual memory management
- Support for higher-level container types (beyond arrays)
- Metaprogramming support (macros and similar)
- Scoping rules (lexical, dynamic, or mix-and-match)
- Programming paradigms (supported, “default”) - imperative, functional, declarative, object-oriented, prototype-based, asynchronous, callback-based, etc.

Almost all scripting languages are dynamically types (static typing extensions appear), offer automatic memory management, and provide some type of higher-level container (usually at least mapping type).
Categorization - JavaScript vs Python

- Both are dynamic languages. There's separate language to add static typing to JavaScript - TypeScript. Type annotations are part of standard Python syntax (but semantics isn't, left to external tools for now - type checkers, AOT and JIT compilers, etc.)
- Python is a vivid example of statically typed scripting language. Following are errors:
  - Array[1.0] - (array indices are by definition integers)
  - "10" + 1 (no automatic conversion of "10" to int (hello PHP), no automatic conversion of 1 to str (hello JavaScript), tell which you want)
- JavaScript is much more relaxed in typing language, up to:
  - foo.i_misspelled_this // accessing non-existent object property - no error
Categorization - JavaScript vs Python (cont.)

Available value types

- Python explicitly has integer and floating-point types. JavaScript numeric type is by spec floating-point.

- JavaScript has “object” type. It’s also oftentimes used as mapping type, but that use requires special care and extra legwork (hasOwnProperty()). By the spec, arrays in JavaScript are “objects” (i.e. mappings) with indices converted to string keys.

- Python has explicit and “clean” mapping and array (list) types. Lists use integer indexing and guarantee O(1) access. Object types are separate (use mapping type for underlying field storage).

- Python has plethora of types for various usages (mutable vs immutable lists, mapping, sets, queues, numeric arrays)
Categorization - JavaScript vs Python (cont.)

Execution model

- JavaScript came from web browser environment, so while not inherent from spec (ECMAScript), the de-facto environment is callback-based event loop.
- Python is a standard imperative language with functional and OO sub-paradigms.
- Callback-based event loop paradigm was explored in Python community with Twisted project since 2002 (long before somebody started to write “real” applications in JavaScript (NodeJS: 2009)).
- Got prominence, but never became leading paradigm due to obvious drawbacks (“callback mess”).
- What’s just one paradigm with Python, is the only choice (“take or leave” with JavaScript). No talking about multithreading with JS.
Categorization - JavaScript vs Python (conclusion)

- With Python, C/C++ will feel largely at home. Strict-typed nature, selection of clearly-scoped types are all supportive of this. Even classic string formatting is reminiscent of C:
  - `print("Hello, \%s\" \% "world")`

- With JavaScript, web programmers will feel at home.

Mottos:
Python: “Explicit is better than implicit”, “Batteries included”
JavaScript: “There’re good parts” (Douglas Crockford)
Challenges of scripting languages on embedded

- **Constrained resources**
  - On deeply embedded, RAM is counted by kilobytes, and usually there’re much more ROM.
  - It can be opposite on “midlevel” devices, e.g. low-cost Linux router may have 32MB of RAM and 4MB of ROM.

- **Constrained UI - screen, keyboard are exceptions**
  - Interaction happens over UART or networking (including wireless)
Challenges of scripting languages on embedded

RAM-saving measures:

● Efficient layout of object structures.
● Minimal size of object structures (but see fragmentation issues next slide)
● Use of tagged pointers (almost every implementation does that)
● ROMmable data structures (one of the highest benefits, MicroPython does that)
● Compressed pointers. If you have just 64K of heap, you need only 16 bits to address it. Actually, no point to address every bytes. With 8 bytes mimalloc block, can address 512K. That’s why JerryScript had heap limited to that size (patch submitted to lift it). This optimization “conflicts” with ROMmable structures, so yet to be explored in MicroPython.
● Most RAM optimizations above are expectedly in conflict with performance.
Challenges of scripting languages on embedded

- Memory **FRAGMENTATION** - the biggest challenge
- Not a problem of scripting languages or automatic memory management at all. Happens with any dynamic memory allocation. Not commonly seen on desktops due to “infinite” memory size of nowadays (billions of bytes, including virtual memory). But server applications server thousands and millions of users fail, fail, fail.
- One of the rules of thumb of (deeply) embedded development - no dynamic allocation.
- There’s theory behind it. Summarizing, the higher ratio of max allocation size to min allocation ratio, the higher fragmentation score and chance that fragmentation becomes fatal.
Challenges of scripting languages on embedded

Memory fragmentation (cont)

- If you always allocate same size, you have no fragmentation - well-known memory pools.
- If you still need to allocate different sizes, can have pools of different sizes (Linux SLAB). But what if some pool is full, and another empty? Maybe not a problem if you have few MBs, but if your RAM is 16K?
- Need to cap the allocation ratio, but capping max/mix alloc sizes.
- It usually doesn’t make sense to allocate too small blocks (e.g. 1 byte), nothing to store there and booking overhead too high. Plus, natural platform alignment and tagged pointers. E.g. for 32-bit platform, minimum sensible allocation is 4 bytes. Capping ratio, tagged pointers, bookkeeping overheads usually call to go few notches higher. (JrS: 8 bytes, MicroPython: 16 bytes, configurable)
Challenges of scripting languages on embedded

Memory fragmentation (cont)

- Capping maximum alloc size is much harder. One of the advanced technique is “chunking” - storing large objects (e.g. strings, array) in a linked list of separately allocated small, fixed-size chunks.

- Used to be implemented in JerryScript (string stored as 8-byte chunks), but was removed. Why? I don’t know, but know why it wasn’t yet implemented in MicroPython: it’s optimization, requiring a lot of modifications to code, which will interfere with maintenance and refactoring of code. MicroPython is not yet up to stage to implement it.
Challenges of scripting languages on embedded

Memory fragmentation (cont), other ways to fight fragmentation

- Maximize use of stack-like structures, where allocation/deallocation doesn’t lead to fragmentation. As anything else, this is partial measure - it works well while you have space in a structure, when it needs more space, it needs to be reallocated from main heap, increasing fragmentation.

- The obvious stack structure is C stack, and most implementations indeed use it heavily. It can overflow too with fatal consequences, so need checking.

- Use of statically allocated scratch buffers (limited help, no multithreading)

- Follow embedded golden rule - avoid (or minimize) allocation. E.g., by using in-place operations. In-place operations are natively supported by Python. E.g., you can allocate buffer once (“statically”) and then copy an arbitrary-sized file (transfer it over network, SPI, etc.) without further allocations.
Challenges of scripting languages on embedded

Automatic memory management and garbage collection

- AMM can help with fragmentation! “Just” use compacting garbage collector!
- Remember Marc Feeley, Danny Dube, 1996: BIT: “Scheme for microcontrollers that includes a real-time garbage collector”? Real-time and compacting, in 1996. Let’s look at source code:
  - README: “- I do not pretend this implementation is complete and can be used as is (even on the 68HC11).”
  - “- The implementation is not well commented and when there are comments, they are in french.”
- 30K C file. To remind, it implements Scheme with its trivial CONS-based structures (the only container structure is O(n) list).
Challenges of scripting languages on embedded

Automatic memory management and garbage collection

- (cont.)

- In the source, there’s no assembly or even call to setjmp. It’s known that garbage collection requires careful following all of root pointers. With optimizing compiler, especially on a RISC platform, some root pointers may be in registers. Not following them will lead to deallocation of used data and hard to diagnose data corruption or crashes afterwards. Maybe BIT written in a way to preclude pointers in registers? There’s no single “volatile” in the source either.

- So long way from a “toy” research project to applying those ideas to real-world scripting language, that among the embedded-targeted scripting language mentioned in this presentation, none uses compacting garbage collection. No talking about real-time GC.
Challenges of scripting languages on embedded

Automatic memory management and garbage collection

- There’s another reason why compacting GC isn’t favoured in current generation of embedded VHLL:
- One way to implement movable (compactable) objects is by extra indirection using handles. These handles need extra storage, hard to spend scarce 16KB of heap on that.
- Alternatively, there can be precise GC, with each object pointer clearly identifiable as such (to differentiate from literal data). Again, that requires extra storage.
Challenges of scripting languages on embedded

Automatic memory management and garbage collection

- Most of the current implementations uses conservative stop-the-world mark-and sweep GC.
- Reference counting isn’t favored, apparently because of the same reason as above - counts needs to be stored somewhere and “partial” benefit it gives (counts need to be updated on each operation, long delays due to cascading deletion are possible).
- However, JerryScript does augment GC with RefCount (such hybrid approach trait of more complex implementations, e.g. CPython uses it.)
- There must be a good reason for that, but in the meantime, following code crashes JerryScript:

```
    a = Array(); for (var i = 0; i < 10000; i++) { a.push("foo"); }
```

(string reference count is limited to 8K)
Challenges of scripting languages on embedded

ROM-saving measures:

- Avoid duplicating functionality. "There's more than one way to do it". No Perl on embedded.
- Compile with -Os except for the most critical code (VM interpreter loop).
- Efficient ROMmable structure layout. (E.g. Python types support a lot of operations, so a particular structure is usually sparse. To be explored in MicroPython (code and maintenance complexity, performance hit)).
- Concise (error) messages. Ultimately, error numbers to be looked up in a manual (trick from 1980ies)?
Challenges of scripting languages on embedded

Other challenges - API design

● Beyond technical challenges, there’re also design challenges, e.g. API design for an embedded language.

● The natural motion is to “inherit” API from “big” version of a language, but use subset of it. E.g. Python has vast and full-featured standard library, ECMAScript spec defines pretty barebones API, but Node.JS API became a de-facto extended API.

● Defining subset to support in embedded language is still a challenge, oftentimes, multiple “API layers” are supported for devices of different caps.

● One of the dichotomy is how much functionality to implement in C, and how much in a target language itself.
Challenges of scripting languages on embedded

Other challenges - API for hardware access

- Of a particular challenge is API design to access hardware features. Unlike other APIs (containers, serialization, networking, etc.), “big” language usually can’t help here, as desktop/server languages simply don’t have such APIs.
- An expected outcome is that everyone does it in their own special way. Oftentimes, design is driven by a particular hardware, and thus not portable to other hardware. If some design overcame that pitfall, it still may be not flexible enough for some (or various) usages.
- One particular dichotomy is functional vs object-oriented API:
  - `gpio_read(14); gpio_write(15, 1)` vs `pin1 = Pin(14); pin1.read()`
  - OO approach may seem more complex if not “bloated”, but it doesn’t have to be, but may offer e.g. better/more natural extensibility.
Case study - JerryScript

- [https://github.com/Samsung/jerryscript](https://github.com/Samsung/jerryscript)
- A Samsung project, a lot of development happens in partnership program with University of Szeged
- Apache 2.0 license, first commit 2014-07-01, 2175 commits, 1726 stars, 40 contributors on Github. Written in ANSI C.
- Implements just core ECMAScript 5.1 spec, there’s a sister project IoT.js ([https://github.com/Samsung/iotjs](https://github.com/Samsung/iotjs)) which implements subset of Node.JS API on top of JerryScript.
- JerryScript includes a port for “Unix” OSes (Linux, MacOSX, etc.) and number of RTOS ports (mbed, NuttX, RIOT, Zephyr).
- Includes extensive testsuite, but it’s intended to be run on Unix port, currently there’s no support to run it against an embedded target. There doesn’t seem to be test coverage measurement support either.
Case study - JerryScript

- Uses hand-written recursive descent parser, compiling directly to bytecode without AST - rather small memory usage to evaluate short statements, but limited opportunity for pre-processing for optimizations.
Case study - JerryScript

Heap memory/performance chart,
http://jerryscript.net/benchmark/benchmark.html
Case study - MicroPython

- [https://github.com/micropython/micropython](https://github.com/micropython/micropython)
- Development funded via series of Kickstarter campaigns, professional services contracts and otherwise, follows open-source development model.
- MIT licensed, first commit 2013-10-04, 6476 commits, 3700 stars, 126 contributors on Github. Written in ANSI C99.
- Implements Python3 language, with almost all features of Python3.4 and selected features of following version on language level. Implements some most important standard library modules and accompanied by sister project [https://github.com/micropython/micropython-lib](https://github.com/micropython/micropython-lib) to implement/port larger subset of standard library in Python.
- Mainline includes Unix, Windows, DJGPP (DOS) ports and STM32, CC3200, QEMU Cortex M, etc. ports.
Case study - MicroPython

- Includes extensive testsuite with support for running on embedded targets either over UART or embedded into application image. Codebase test coverage is 90+% (recently dropped due to inflow of extension modules, core coverage is ~95%). Testsuite is intended to be runnable with 16KB of heap.
- Includes developed interactive prompt with auto-completion, auto-indenting, paste mode, etc.
- Simple mark and sweep GC in a dedicated heap area with bookkeeping bitmap.
- Uses AST building parser: higher memory usage, but allows for more advanced processing, e.g. constant folding compile-time.
Case study - MicroPython

Code sizes/performance statistics chart

http://micropython.org/resources/code-dashboard/
JerryScript vs MicroPython Zephyr ports head to head

- **JerryScript, qemu_cortex_m3 target**
  - Binary size: 123680 (120.79KB), size(1) output:
    ```
    text    data    bss
    119360  4320     29868
    ```

- **MicroPython, qemu_cortex_m3 target**
  - Binary size: 119808 (117.00KB), size(1) output:
    ```
    text    data    bss
    116656  3149     29864
    ```

Both languages are compiled with 16KB heap (goes into BSS). It’s insightful that they have almost the same BSS size, meaning that the rest is Zephyr’s overhead. MicroPython is ~25% smaller with initialized data size (dwarfed by heap requirement though).
JerryScript vs MicroPython Zephyr ports head to head

- For code size, JerryScript port was taken as a reference, and MicroPython was configured to achieve about the same code size. As can be seen, JrS/Z port neatly leaves few KBs of code space for some simple driver (e.g. GPIO) for 128K FlashROM MCU, but more advanced usage (e.g. networking) would require 256K MCU.

- So, the idea is to compare what both implementations offer in the same size.
- JrS offers floating-point type, uPy separate 64-bit integer and float types.
- JrS Z port is built with “minimal” profile, and turned out that it doesn’t enable any string methods. MicroPython has 22 (find, replace, split, strip, tests, etc.)
- uPy has real array (list) type, with worst- and best-case access of C (constant). JrS arrays are implemented on top of hashmaps. No array methods in minimal profile, too (uPy has them).
Conclusions and future work

- Scripting languages for IoT is exciting area. Not entirely new, IoT movement may breathe new life into them and make standard, not niche, feature.
- Scripting languages already offer benefits and competitive advantage (getting customer on-board with product and rapid prototyping), and offer promise for more.
- One solution or one size-fits-all unlikely to work, just the same as for desktops/servers.
- There’re already established projects in the area, though they still need to go a long way towards more advanced features. (Gathering momentum would help.)
- JerryScript and MicroPython Zephyr ports are at their infancy, offering little more than UART interactive prompt. Bindings for networking API is work in progress, and other APIs are in queue. (Zephyr itself is a moving target so far.)
Thank You

#LAS16
For further information: www.linaro.org
LAS16 keynotes and videos on: connect.linaro.org