Introducing Em – Taking RISC-V Software Over The Edge

Bob Frankel*

Co-Founder and CTO, Bespoke IoT

Abstract

*The **Em** programming language elevates embedded firmware development to a higher-level which has historically eluded C. Originally conceived in the 2009-2010 timeframe, **Em** has evolved over the last decade through a series of commercial deployments in low-power, low-cost wireless IoT applications. Thanks to novel optimization techniques employed by the underlying language translator, **Em** programs would invariably out-perform their hand-crafted C counterparts in terms of time and (especially) space.*

*Initial engagement with RISC-V began last year through **Em** support for two development boards used for edge-processing. With tenfold reductions in program footprint not uncommon, opportunities abound to target small RISC-V MCUs and SoCs with less than 32K of memory – pushing conventional edge-processing designs to the IoT fringe in terms of silicon size and power consumption.*

Beyond C

The **Em** programming language – expressly designed for 8/16/32-bit MCUs with very little memory – elevates embedded firmware development to a higher-level which has historically eluded C (or even C++). C continues to this day as the dominant programming language for deeply-embedded MCUs in general, as well as for memory-constrained RV32I-compliant RISC-V cores in particular – a testimony to the staying power of the C language over the past fifty years.

Originally conceived in the 2009-2010 timeframe at UC Santa Barbara [1], **Em** enabled undergraduate EE students (using this primer [2] for reference) to implement low-level MCU firmware using modern programming constructs such as *interface inheritance* and *component composition*. Remarkably, we found that higher-level programming did not necessarily lead to higher-levels of program overhead. Thanks to novel optimization techniques employed by the underlying language translator, **Em** programs would invariably outperform their hand-crafted C counterparts in terms of time and (especially) space.

From its baseline implementation within academia, the **Em** language and its growing library of runtime components (written in **Em**, of course!) has subsequently evolved and matured through a series of deployments in commercial IoT applications executing on low-cost, low-power hardware:

2011-2015 — a BLE stack running on 8-bit MCUs, licensed to early manufacturers of mobile-controlled "things";

2015-2019 — tracking individual point-of-purchase displays situated within stores operated by major retail chains; and

2019-2022 — collaboration with a large silicon vendor targeting cost-constrained designs for their wireless MCUs.

With application volumes projected as high as 100M units/year, the **Em** language played a critical role in keeping BOM costs in check as well as extending battery-life in these systems. Through its innate ability to reduce overall program size – sometimes tenfold – **Em** has allowed us to target lower-cost MCUs with less flash/SRAM. Smaller programs will also tend to perform the same functions in fewer instructions, enabling more time for deep-sleep and/or a slower processor clock – saving energy all around.

As of today, the **Em** language runtime has found its way onto more than twenty 8/16/32-bit MCUs from almost a dozen different silicon vendors. The **Em** language translator, which ultimately outputs ANSI C/C++ code for portability, has also targeted the most popular toolchains for embedded development (GCC, IAR, Keil, LLVM). Thanks to a recent rewrite of the translator into TypeScript, **Em** now enjoys robust language support within the VSCode IDE.

More important, perhaps, less than a handful of **Em** programmers have developed *thousands* of **Em** modules used (and often re-used!) across a broad range of IoT applications targeting resource-constrained MCUs. Due to the proprietary nature of these applications, however, the **Em** language and its runtime has remained closed – until now.

RISC-V Opportunities

Initial engagement with the RISC-V ecosystem began last year through **Em** support for two popular development boards used in edge-processing applications.

SiFive HiFive1 — As part of a RISC-V initiative at Rice University [3], several students became familiar with the **Em**

* Corresponding author : bob@biosbob.biz

programming language environment. These students also evaluated the **Em** runtime against the Freedom Metal library supplied by SiFive, finding that the latter's well-structured, object-oriented design in fact led to a tenfold increase in run-time footprint versus **Em**.

OpenISA RV32M1 (VEGAboard) — Designed and manufactured by NXP, **Em** support for the RV32M1's ZERO-RISCY and Cortex-M0+ CPUs has enabled meaningful side-by-side benchmarks (time, space, power) using a common set of on-chip memories and I/O devices – including a highly-configurable, multi-protocol 2.4 GHz radio used in other NXP parts. Using only this radio's low-level FSK PHY, the author demonstrated a minimal (yet compliant) **Em**-based BLE stack that executes in under 8K of SRAM; by contrast, NXP's own BLE stack consumes ~200K of RV32M1 memory and requires extensive link-layer support from the radio hardware.

While "resource-constrained" when compared with the cloud or even mobile devices, the MCUs used on these edge-processing boards have generous amounts of cached program flash as well as tightly-coupled SRAM data blocks. Non-trivial applications for these MCUs will usually have memory footprints measured in the hundreds of kilobytes.

By contrast, real-world IoT applications written in **Em** will typically execute in under 32K of memory – including a rudimentary task scheduler, drivers for all system peripherals, and a low-power wireless communication stack. This significant disparity in overall program size in turn leads us to frame a more fundamental question:

*If programming in **Em** can reduce software footprint by 10X, why not pursue similar economies in the silicon?*

Edge-processor roadmaps from leading chip manufacturers currently feature *more* (not less) hardware – larger memories, multiple CPUs, complex peripherals. Perhaps we can now leap "over-the-edge" and explore a new category of minimalist MCUs "on-the-fringe" of the IoT hierarchy.

The RISC-V community offers a wide-range of processor cores, including several entry-level offerings [4, 5] that benchmark favorably against ARM Cortex-M0. Pushing the envelope even further, minimalistic CPUs [6, 7] that today target small FPGAs could eventually supplant 8-bit MCUs currently entrenched at the IoT fringe. The "tiny-code" produced by programming in **Em** would further amplify the impact of these tiny RISC-V cores on overall system performance.

By virtue of their small silicon footprint, MCUs and SoCs built around these tiny RISC-V cores could potentially consume much less power than devices featuring (for example) a Cortex-M0 CPU. Results reported by Schiavone [8] encourage further exploration in this direction.

The "openness" of the RISC-V technology also encourages the design of tailor-made cores for specific application domains – such as implementing ultra-low power wireless DSP extensions for software-defined radio transceivers [9]. Orthogonal to these silicon improvements, writing digital baseband software in **Em** can only help the cause.

Next Steps

Logging more than a decade of real-world usage within resource-constrained embedded applications – plus some recent penetration into the RISC-V community – perhaps the time has finally come to *open-source* the **Em** programming language and its runtime. As **Em** enters its fifth-generation since inception, a provisionally named *Em-V* project would make the language broadly available, not only supporting RISC-V platforms but alternative 8/16/32-bit MCUs as well.

Easier said than done, however! Short of simply posting source code to a public GitHub repository, the author seeks guidance from the RISC-V community on *how* to best organize and operate the Em-V project to maximize its impact.

In the meanwhile, **Em** will continue to push the *tiny-code-for-tiny-chips* envelope – by working with active projects such as NEORV32[10] and X-HEEP [11], whose designs could potentially bring RISC-V closer to the fringe of low-power MCUs. With an ASIC forthcoming, X-HEEP could also provide an ideal platform to demonstrate the potential of **Em** – and to start moving deeply-embedded software beyond C.

References

[1] A. Amar. *Support for Resource Constrained Microcontroller Programming by a Broad Developer Community.* Doctoral thesis, UC Santa Barbara, 2010. tinyurl.com/yxd8s3tr.

[2] *Introducing Em* (UC Santa Barbara, 2010). tinyurl.com/bdftm8yu.`

[3] R. Simar. *RVR (Risc-V at Rice) Lab*. tinyurl.com/bp8dz6a7.

[4] OpenHW Group. *CORE-V CVE2 RISC-V IP.* tinyurl.com/33hycpfj.

[5] SiFive. *E2 Core IP Series*. tinyurl.com/b4tns3fz.

[6] O. Kindgren. *SERV – The SErial RISC-V CPU*. tinyurl.com/4pmt4v5d.

[7] B. Levy. *FEMTORV32 / FEMTOSOC: a minimalistic RISC-V CPU*. tinyurl.com/34zxyjmx.

[8] D. Schiavone. *Design of energy-efficient RISC-V based edge-computing devices*. Doctoral Thesis, ETH Zurich, 2020. doi.org/10.3929/ethz-b-000463184.

[9] H Amor. *A RISC-V ISA Extension for Ultra-Low Power IoT Wireless Signal Processing*. tinyurl.com/bdhty2f4.

[10] S. Nolting. *NEORV32*. https://tinyurl.com/57y2m82j

[11] ESL/EPFL. *X-HEEP (eXtendable Heterogeneous Energy-Efficient Platform)*. tinyurl.com/2mbjzxd5