

RETRACTED




uJVM: Lightweight Java Virtual Machine for embedded systems

Oleksandr S. Moliavko¹, Taras A. Drozdovskiy¹, Vitalii M. Petrychenko¹, and Oleg E. Kopysov¹

¹ Samsung R&D Institute Ukraine ¹

DOI: [10.21105/joss.01338](https://doi.org/10.21105/joss.01338)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Submitted: 20 February 2019

Published: 19 April 2019

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC-BY](#)).

Background

Embedded computing systems have already become omnipresent in day-to-day life, and in the foreseeable future their applications and importance will increase even further. However, developing S/W for these systems is generally slow, complex and error-prone process, largely because this S/W must comply with limitations imposed by specific H/W platforms and use platform-specific libraries or assembly-language fragments to access H/W devices (storage, I/O ports, timers, etc.). These limitations make almost any application locked to specific platform, greatly increasing development costs and forcing the developers to start from scratch if a similar S/W is necessary for a different H/W platform. Also, writing a lot of low-level code makes applications difficult to debug and maintain. In an era where embedded applications often process confidential data, S/W errors become security vulnerabilities that can have severe consequences. However, manually written low-level code typical for C apps can be removed altogether by using Java for S/W development. Java apps are largely immune to C issues with memory leaks, undefined behavior and buffer overruns; besides, they're inherently cross-platform, so they can be written and debugged without even having access to actual H/W on which these apps are supposed to run. These advantages make Java, at least in theory, an attractive alternative to C for developing embedded system S/W; however, Java Virtual Machine implementations suitable for embedded systems are either proprietary and inaccessible to general community, or very minimalistic and lack functionality necessary for efficient applications. uJVM was developed as a part of research effort at Samsung into cross-platform S/W development for embedded systems. We expect it to be useful for both commercial applications and research; areas of research it will be used in include, among others, comparative C/Java performance evaluation, standard compliance, thread safety and memory management mechanisms. Memory management is of particular importance for embedded systems which usually have limited RAM capacities and are supposed to run continuously for extended time periods; faults in memory manager that can cause memory leaks or excessive fragmentation can be fatal in embedded systems, in which applications cannot rely on OS-provided mechanisms. Performance penalties of Java applications when compared to similar C counterparts are expected to be significant, however, many of modern embedded systems have computing power that far exceeds one actually necessary, so this issue is not expected to be as critical as efficient memory management. Standard compliance is important for cross-platform applications; since in Java the ability of H/W platform to run application is largely determined by JVM that runs on this H/W, much attention must be paid to uJVM standard compliance.

Implementation overview

uJVM was designed in accordance with overall architecture described in Java Virtual Machine Specification Java SE 7 Edition (Lindholm, Yellin, Bracha, & Buckley, 2013). So basically it can be divided into 3 parts:

- Class loader Class loader of uJVM can load contents of classes stored in .class or .jar files into memory, perform linking and initialization.
- Managed memory areas (“The java memory model,” 2009) uJVM supports thread specific register storage, memory allocation for heap and thread-specific stacks and garbage collection.
- Execution engine Due to severe memory footprint restrictions, uJVM has only byte-code interpreter as execution engine - implementing JIT compiler would greatly increase memory consumption. uJVM execution engine gives Java code access to H/W drivers (e.g., UART, GPIO and SysTick for most platforms) and JNI methods. Note, however, that as of January 2019 Java classes that directly contain native C code are not supported yet.

As of March 2019, uJVM is available for following H/W platforms:

- STM32F103-BluePill
- STM32F4Discovery
- STM32-E407
- STM32F429I-DISCO
- STM32F769I-DISCO
- Arduino MEGA 2560
- MSP-EXP432P401R
- EK-TM4C1294XL
- NuMaker-PFM-M2351
- SAML11 Xplained Pro

Real-time operating systems for which uJVM has working proof-of-concept applications are:

- ZephyrOS
- aFreeRTOS
- NuttX
- OP-TEE platform (as a trusted application implementation).

Note that the source code for these applications is not present in repository, but can be added easily if need arises.

uJVM was developed from the outset as an Open Source implementation of Java Virtual Machine for embedded systems and other computational environments with severe resource limitations. It has much lower memory footprint than other available JVM implementations (bare-metal variant including a simple Java application can run in as little as 6 kB of RAM and needs approximately 45 kB of Flash ROM for storage) and has working variants for a variety of architectures including x86_64, AVR and ARM (Cortex-M3 (“ARM cortex-m3 technical reference manual,” 2006), Cortex-M4 (“ARM cortex-m4 technical reference manual,” 2010) and Cortex-M23 (“ARM cortex-m23 processor technical reference manual,” 2010) families). Build system provided in uJVM project allows

the developer to configure uJVM, disabling unnecessary features to conserve resources or enabling specific H/W capabilities support (e.g., using FPU to accelerate floating-point calculations will provide an order of magnitude acceleration in calculation-intensive tasks).

Potential future development and applications

There are plans to obtain Oracle certification of uJVM as Java SE 7 compliant JVM implementation by fully passing TCK, to make uJVM suitable for applications that require certified JVM implementations. uJVM source code and documentation are available at <https://github.com/Samsung/uJVM>. Development plans include improving JNI support, execution speed and library support, including H/W peripheral control libraries.

Expected audience for uJVM project

uJVM project can benefit from contributions from several categories of specialists:

- Developers with experience in C language app development for resource-constrained environments - for improvement of Java bytecode interpreter and its subsystems
- Developers with H/W library development experience - for extending support of H/W devices in future Java applications
- Java developers - for providing standard library support by uJVM
- Application developers who will use uJVM as a platform for practical applications

Research applications

At first, creating a Java Virtual Machine capable of running useful applications on resource-constrained embedded systems was considered difficult, and uJVM was itself developed as a part of proof-of-concept research project. However, now that uJVM is a reality, there is an ongoing research on using it for implementing security-related S/W components of LF Edge ecosystem. Among others, a smart home appliance controllers based on Cortex-M3 chips are expected to use Java applications running on uJVM for processing user's biometric authentication data. As reaction speed and resource consumption of generic solutions already available on market were deemed unsatisfactory, uJVM running under Zephyr RTOS (<https://zephyrproject.org>, <https://github.com/zephyrproject-rtos/zephyr>) or as a bare-metal application was proposed as an alternative. While memory consumption is found to be roughly similar to C-language equivalent Zephyr apps, ease of debugging and verification seem to make uJVM an appealing alternative to C-based framework. Performance of said Java apps was not evaluated yet as of time of this writing, but preliminary proof-of-concepts seem to operate fast enough to provide user interface free of lags and freezes.

References

ARM cortex-m23 processor technical reference manual. (2010). Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0550c/cortex_m23_r1p0_technical_reference_manual_DDI0550C_en.pdf

ARM cortex-m3 technical reference manual. (2006). Retrieved from https://static.docs.arm.com/ddi0439/b/DDI0439B_cortex_m4_r0p0_trm.pdf

ARM cortex-m4 technical reference manual. (2010). Retrieved from https://static.docs.arm.com/ddi0439/b/DDI0439B_cortex_m4_r0p0_trm.pdf

Lindholm, T., Yellin, F., Bracha, G., & Buckley, A. (2013). The java® virtual-machine specificationjava se 7 edition. Retrieved from <https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf>

The java memory model. (2009). Retrieved from <http://www.cs.umd.edu/%7Epugh/java/memoryModel/>