ZeroJVM – An Extremely Lightweight Java Virtual Machine for Flipper Zero devices

Liu Kurafeeva Roman Beltiukov

1 Problem Introduction

Java is a high-level, object-oriented programming language that was designed to be easy to use and flexible. It is often used for building complex, large-scale applications, such as enterprise software, web applications, and mobile apps [6].

One of the key features of Java is that it is a "write once, run anywhere" language. This means that code written in Java can be run on any platform that supports the JVM without the need for any changes or modifications. This makes Java a great choice for developing cross-platform applications.

In addition to being used on traditional computing platforms like desktop and server systems, Java is also commonly used on embedded devices. Embedded systems are small, specialized computers that are integrated into other devices, such as phones, TVs, cars, and appliances. Because Java is platform-independent and has a wealth of libraries and tools for building robust applications, it is a popular choice for developing software for embedded devices.

However, implementing a JVM for an embedded device can be challenging. Different embedded devices have different hardware and software requirements, and the JVM must be tailored to the specific needs of each device. In this work, the authors are implementing a JVM for a specific embedded device – Flipper Zero, allowing them to use Java on that device and take advantage of its features and capabilities.

1.1 Flipper Zero

Flipper Zero [1] is an STM32-based multi-tool device capable of interaction with radio protocols, access control systems, hardware, and more. It's fully open-source and customizable, and there exist several different firmware images (including unofficial ones).

Flipper Zero hosts a FreeRTOS-based customized operating system, that allows running various external applications written in C, C++, or Rust. Flipper developers provide a specific build system to properly design and implement external applications considering API versions, device capabilities, and OS exported and available kernel functions.

As Flipper Zero is designed as an Everyday Carry (EDC) device, we decided to expand its capabilities by implementing a lightweight Java Virtual Machine runtime to execute compiled Java code. This allows Flipper Zero owners to port and execute existing Java programs on the device, making it even more versatile and useful for a variety of tasks.

1.2 Limitations of existing solutions

Several limitations prevent the usage of existing JVM implementations on the Flipper Zero device:

1. **Memory constraints:** Flipper Zero has around 130KB of free RAM available for the application. Given that due to OS configuration, an application should be loaded to the memory during execution, this means that both runtime, Java class code, and Java heap size should be less than 130KB together. This heavily limits the number of JVM implementations that could be used on the device and requires deploying an extremely lightweight solution to provide more memory for the actual code and execution.
2. **Compatibility:** Flipper Zero RTOS brings multiple compatibility issues to the existing implementations of the JVM, such as the lack of many default kernel functions (`calloc`, `fopen`, `fseek`, etc.), a specific way of building and deploying applications, and others.
3. **Licensing:** Multiple existing JVM implementations capable of running on STM32 devices are proprietary [5, 2] and require licensing for usage. Additionally, using proprietary JVMs may restrict the ability to modify or customize the JVM to fit the specific needs of the Flipper Zero device and its constraints.

2 Proposed Solution

Considering the limitations, we made several design choices for our implementation.

Goal. We need to design a Java Virtual Machine for Java 8, that would be extremely lightweight and provide basic capabilities for executing the compiled java code. The code would be compiled using a standard javac compiler and manually deployed to the device. The code should be successfully parsed and executed, and the runtime should finalize the execution and return control to the operating system afterward.

Capabilities. The JVM should be able to:

1. Load a single Java class named `Entrypoint.class`, located in a specific location on the filesystem, successfully parse it, and create an internal representation of this Java class in runtime memory.
2. Correctly create and finalize frames for functions execution, given information about stack and locals from javac.
3. Correctly represent and work with static fields, static methods, instances of the class `Entrypoint`, and also instance fields and instance methods.
4. Implement basic math operations for byte, short, int, and float variables.
5. Properly allocate memory for instances of the class `Entrypoint`.
6. Provide an `stdout` channel connected to the device display to allow execution results usage during everyday activities.

Code Interpreter. To reduce memory consumption and improve performance, our implementation of the JVM will use an interpretation mechanism for code execution. While ahead-of-time (AOT) compilation techniques can be used to optimize the execution of Java code, we will instead focus on creating a runtime that can execute arbitrary Java code, including code that has been ported from other platforms and is not specifically prepared for the Flipper Zero device. This will

make it easier for developers to use existing Java solutions on the Flipper Zero device without the need for additional preparation or modification.

Memory Management. To support class instance creation, the runtime should provide memory allocation using the free available memory of the device. In order to reduce memory and execution overhead, we will not implement Garbage Collector for object deallocation, instead allowing the operating system to reclaim all used memory after the program finishes its execution.

3 Implementation

In this section, the implementation of the ZeroJVM is described. The source code is published under an open-source license on GitHub [7].

3.1 Project structure

The ZeroJVM is implemented as a multi-target C project that could be compiled using provided Makefile for ‘x86‘ and ‘flipper‘ targets. The project uses the same source code for both targets, excluding specific entrypoint information and target-exclusive functions.

The implementation consists of the following:

- a program entrypoint that defines compiled Java class file location and initializes target-specific settings.
- `utils.h/.c` files that provide project-level structures and target-specific implementations of custom functions that differ for ‘flipper‘ and ‘x86‘, e.g. `calloc`, `fread`, etc.
- `loader.h/.c` files that provide Java class loading and parsing capabilities and initialize correct runtime structures for `Entrypoint` class implementation.
- `frame.h/.c` files that implement frame structure and needed functions for frame initialization and execution.

3.2 Class loading and parsing

When a Java application is started, the JVM begins by loading and parsing the class file named `Entrypoint.class` according to the specification of class file [3]. The JVM reads the bytecode of the class file, allocates Constant Pool, reads methods and fields of the class, assembles a class instance template that would later be used for creating new class instances, and then creates a corresponding map and representation of the class.

We intentionally do not verify the output of a `javac` compiler, including the type correctness during runtime.

At the end of class initialization, JVM starts a new frame for execution of `<clinit>` function of the class. The `<clinit>` method initializes static fields in a class, assigns initial values, and may also involve performing other tasks such as creating static objects or initializing static data structures.

After `Entrypoint.class` is loaded and initialized, ZeroJVM looks for a `public static void main(String argv[])` method and executes it.

3.3 Frame management and execution

Each function execution in JVM requires a separate frame with local variables and access to a stack. To support this, we developed the needed structures to be used by a Java bytecode interpreter during code execution.

To initialize a frame, ZeroJVM uses `maxlocals` and `maxstack` information to provide local variables array and separate stack for the frame. In addition to this, the frame also holds a reference to the current Java class, method bytecode length, and a reference to the bytecode itself. Before the initialization of each frame, a calling code should also provide an array of local parameters, parsed from the function descriptor and collected from the stack.

After the frame is initialized with all values described above, the calling code calls the `execute_frame` method and provides a reference to the frame as a parameter. The frame is executed until a proper instruction to return control to the previous code is read and executed.

The frame execution is implemented according to the Java Virtual Machine specification of instruction set [4]. During the frame execution, runtime sequentially reads java bytecode and executes a corresponding switch-case block, operating local and stack arrays. This can include invoking other methods and creating new frames for code execution. After the opcode corresponding to any of `return` instructions is executed, the frame allocates a new container for resulting output, finalizes the frame (by freeing the stack and the locals), checks that the current instruction pointer corresponds to a total bytecode length for this method, and returns a pointer to result to the previous frame.

In ZeroJVM, we implemented 90 instructions for Java opcodes, supporting goals established in section 2. We implemented and tested all program flow instructions (supporting loops and comparisons) and all instructions for byte/short/int/float math. We have not implemented any double, long arithmetic, and reference type instructions, which could be a target for future work.

3.4 Memory management

To support the instantiation of new exemplars of `Entrypoint` class, ZeroJVM implements a simple memory management mechanism referred to as Garbage (Un)Collector. During the initialization of new instances of the `Entrypoint` class, ZeroJVM allocates needed memory for the object from the free heap memory of an underlying operating system. All the allocated memory lives until the end of program execution, therefore implementing a no-op garbage collection mechanism similar to Epsilon GC. All allocated memory would be reclaimed by the system after the execution of ZeroJVM.

This approach prohibits ZeroJVM from successfully executing programs operating with multiple objects with a non-intersecting lifetime which combined heap usage is bigger than the actual heap size. While this is a flaw of the ZeroJVM, the memory consumption of any garbage collector would further reduce the available heap size of the program, though could be considered for future implementations.

3.5 HAL interaction

To provide basic user interaction and make it useful in everyday life, we decided to implement two different interaction scenarios with OS's Hardware Abstraction Layer: text-on-screen output and sound output.

Text-on-screen output. To provide basic output, we implemented a `print` function that takes formatted C-string and displays it on the Flipper Zero screen. To avoid initializing of `System`

class for `System.out.println` implementation, we reserved a signature `public static void println(parameter)` in the `Entrypoint` class for parameters `String`, `int`, and `float`. If, during program execution, there is a call to one of the methods with this signature, the runtime intercepts it and executes display initialization, and outputs the provided parameter on the screen. The text is displayed until the next display command or until the end of the application.

Sound output. We also implemented control of the internal Flipper Zero speaker by reserving a signature `public static void beep(float frequency, int duration_ms)` of the class `Entrypoint`. If, during program execution, one of the instructions calls this method, the runtime will intercept the call and use provided parameters to initialize the speaker and produce a sound of a given frequency and duration.

4 Conclusion

In conclusion, the implemented ZeroJVM runtime enables the execution of Java code on Flipper Zero devices. The runtime has a binary file size of 16 KB, leaving approximately 84 KB of memory available for Java code and heap. ZeroJVM implementation supports java class loading and parsing, basic math, basic OOP features, memory management, and hardware-level abstraction interaction. Some potential areas for future work on this JVM implementation include:

1. Expansion of opcode implementations to fully support all operations of reference JVM implementation
2. Partial implementation of core Java library classes to provide further portability and execution support
3. Implementation of dynamic class reading and parsing during runtime execution to provide full OOP capabilities
4. Development of a simplified garbage collector specifically designed for the constraints of the device

By addressing these issues, the ZeroJVM runtime could provide a full experience of programming using managed runtime language on Flipper Zero devices and facilitate the reusage of already existing programs on this platform.

References

[1] Flipper zero - portable multi-tool device for geeks. `https://flipperzero.one/`.

[2] Jamaicavm. `https://www.aicas.com/wp/products-services/jamaicavm/`.

[3] Chapter 4: the class file format. `https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html`.

[4] Chapter 6: the java virtual machine instruction set. `https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html`.

[5] Microej. `https://www.microej.com/`.

[6] Tiobe index. `https://www.tiobe.com/tiobe-index/`.

[7] Zero jvm. `https://github.com/nectostr/ZeroJVM`.