

## NanoVM on the ZNEO

4/22/2011  
Nathan Scott

### ***Project Abstract***

The NanoVM is a small subset of the Java Virtual Machine, built to run on the AVR ATmega. It allows the programmer to have a faster turn-around time in terms of development by providing very basic libraries to complete basic tasks. For example, input and output is simplified by streams, as well as LED libraries. It is not very good for full-blown projects, although solid enough for fast prototyping of ideas, or simple programs.

On the ATmega, it does this by directly overriding the flash memory and the loader itself, replacing it with NanoVM. Programs are then loaded on through a serial cable.

The project aimed to port NanoVM over to the ZNEO. Both are microcontrollers that operate in a 16-bit address space and have similar functionality. The ZNEO is more robust in terms of sensors and peripherals, but the base system is essentially the same.

### ***Status***

The ported version of the program, unfortunately, would not flash to a ZNEO without a warning from the compiler, stating that you will override part of the actual flash bits (hence not being able to ever flash the board again). While this would be fine for the ATmega, this is certainly not the case with the ZNEO, as the ZNEO has far more memory. Memory space was a defining characteristic for the other board.

The class loader works as expected though, and it can push programs with the NanoVM tool. The NanoVM tool is essentially an add-on for the Java compiler that alters the class files a little bit and bundles them into an Intel .hex file to be loaded onto the board. This allows multiple classes to be loaded into one file and transferred without problems.

There are no compilation errors or warnings. Just make sure to use the large memory model, as the errors reside in one space and the actual programs in another. Also “disable optimizations for easier debugging”.

### ***Specification***

I used the most recent version of NanoVM as of this writing, version 1.6. The homepage was located at <http://www.harbaum.org/till/nanovm/index.shtml>. You need the Java RXTX libraries as well, which you can find online. There are included installation scripts in /doc.

There is a wiki, although it's some what outdated and in German. You can view the wiki through a translation service of your choice at <http://www.mikrocontroller.net/articles/NanoVM> (it's pretty obscure on the website).

You will also need Sun's (now Oracle's) Java compiler; just Google for it. That's all you, aside from the ZNEO and a serial cable.

## Implementation & Construction

I took all the files and loaded them into the ZNEO IDE. Switch the project settings to the large memory model and disable compiler optimizations. In `error.h`, change the `ERROR_HEAP_BASE` to `0x800000`, and in `heap.c` change the initial value of `heap_base` to `0xFFB000`. This is where the stack and heap will start. Make sure that `config.h` is the file from the ZNEO directory. Make sure `#undef DEBUG` is included, as ALL of the built-in debug code for the AVR breaks everything. If you do not change the heap and error heap base, you will get a memory overlap warning and they will most likely corrupt each other.

`native_zneo.h` is where you can define the C functions that will be mapped to the Java functions, and naturally `native_zneo.c` is where you fill in the function body. You also need to put these define the classes and methods in `native.h`. Just follow the pattern – classes are offset by a certain integer, and methods are enumerated starting from 1 in every class. NanoVM is interesting because it can statically calculate the amount of memory any class will potentially take, therefore when it allocates new memory for an object it can mix the stack and heap together. This also helps with garbage collection.

Now, there was plenty of other code changes but mostly they had to do with data types and working around the pre-existing code. If you want to build it from the originally NanoVM source, expect most time to be taken in fixing data types and figuring out the parallels between the ZNEO and the ATmega. Understanding how to properly configure the board is half the battle, then getting the ZNEO libraries imported at the proper points, as well as getting UART serial working. If you do this from within the IDE, it isn't too difficult, but it will take time.

The next step is actually building files. The VM tool takes advantage of the Java Native Interface (JNI). Under `/java/nanovm/[platform]` (in this case, `zneo`) you can find functions that will map to native C functions. All you need to do is specify the method signature like “`public static native void method();`”. Compile that file with the Java compiler.

Under `/tool`, you can find a small batch script that will automatically call the VM tool and upload a compiled class file. The full command is

```
“java -jar NanoVMTool.jar .\config\Zneo.config ..\java LED”
```

The idea is to call the jar (`java -jar NanoVMTool.jar`), specify what platform you're uploading for (so it can properly detect how to upload), and the actual class file(s) you want to upload. In `/tool/config/` you need to set the properties for `Zneo.config` to match your machine (e.g. switch the COM ports and native classes).

There is another way to get a program on the board. The NVM tool will create, if you specify the UNIX output, a `.nvm` file in the directory where you called it. Inside will be a bunch of gibberish, but with `simple_converter`, it takes the gibberish and gives out a hex version, one that can be sent along with the source code onto the board from the IDE. This is a fall back measure if the `loader.c` file fails, or if the tool fails to upload to the board through serial. Pipe the output the `.nvm` file into `simple_converter` and it will spit out code that is properly formatted to standard out. Take that output and place it in `nvmdefault.h`, overriding the array already there. Re-compile in the IDE and upload to the board. The code *should* work.

It's a fairly complicated dance to add native methods, but once you do it's really just a matter of using the tool to get the programs onto the board.

## Retrospective

C is a beautiful thing, but for a long time I wondered why Java was so rarely supported on embedded systems. There *are* some major advantages to Java at times, namely garbage collection

(or the lack of needing to specifically handle pointers). At this point., I understand completely. The overhead required to have a functioning Java Virtual Machine on an embedded platform does not make a lot sense – at least not in this way. Maybe if there was a small interpreter written that would simply convert the Java byte code to embedded opcodes, that would be efficient and useful. As it is though, running an entire JVM seems a little silly.

I mapped out the file system hierarchy at the very beginning and started figuring out which files went where and how they interacted – that was crucial. From there, things seemed clearer and it was a matter of squashing errors from the IDE and making things compatible.

If I knew what I know now, I would have simply picked a different project. Too much software. NanoVM was developed over the course of over a nine months, exclusively devoted to the AVR ATmega. I know it was just a port, and I wanted something difficult, but it felt pretty brutal at the end.