The Smartrek Virtual Machine

User Manual

Compiler Usage, Language Reference and VM Specification

Revision History

Revision	Date	Author(s)	Description
3.0.0	2022/12/05	GS	Version 3.0 changes
2.4.0	2022/11/14	GS	Version 2.4 changes
1.7.0	2022/05/27	GS	Version 1.7 changes
1.6.0	2022/04/11	GS	Version 1.6 changelog
1.5.3	2022/03/16	GS	Complete documentation for version 5
1.5.2	2022/03/07	GS	Interrupts
1.5.1	2022/03/05	GS	Add coroutines
1.5.0	2022/03/02	GS	Updates for version 5
1.4.1	2022/02/18	GS	Improvements and modifications regarding rc3
1.4.0	2022/01/22	GS	Updates for the version 4 of the compiler
1.3.1	2022/01/22	GS	Complete ASM and Opcode sections
1.3	2022/01/20	GS	Document created for smartrekvmc1.3

Contents

List of Figures

List of Tables

1 `smartrekvmc` changelog

Version 3.0

- **IMPROVEMENT** Proper string escape sequences
- **IMPROVEMENT** Various improvements to the type system
- **IMPROVEMENT** Various changes to the cast system
- **IMPROVEMENT** New standard library functions
- **IMPROVEMENT** LSP server improvements
- **FEATURE** Pointers
- **FEATURE** Lambdas and explicit closures.
- **FEATURE** Support for the STVM MMU
- **FEATURE** Experimental STVM JIT Support
- **BUGFIX** Errors when casting from/to Object
- **BUGFIX** Better Indexing Lexing
- **BUGFIX** The LSP Server stopped scanning for errors prematurely in some cases

Version 2.4

- **FEATURE:** Reflection
- **FEATURE:** Declare native function handles in source files

Version 2.3

- **IMPROVEMENT**: Faster SMK900 Sercom Reads
- **IMPROVEMENT**: Faster dynamic method call
- **IMPROVEMENT**: Optimize some STDLIB functions
- **FEATURE:** Nwk SMK900 interface
- **FEATURE:** Boost clock function
- **FEATURE:** JIT API
- **BUGFIX**: SPI reads are sometimes dropped
- **BUGFIX**: Error in `Short.reverseBytes`

Version 2.1

- **IMPROVEMENT**: Do not include unused static variables
- **IMPROVEMENT**: Increase maxiumum number of methods
- **IMPROVEMENT**: Included default STDLIB
- **FEATURE:** Inline CL statements
- **FEATURE:** String functions
- **FEATURE:** LSP Server Support
- **BUGFIX**: SMK900 lib SPI read

Version 1.7

- **IMPROVEMENT**: Bytecode optimization of arithmetic
- **IMPROVEMENT**: AST optimization improvement
- **FEATURE:** Add support for new GC

Version 1.6

- **IMPROVEMENT**: Improve ELF output
- **IMPROVEMENT**: Typechecking improvements in return types.
- **IMPROVEMENT**: Remove unused progmem arrays

- **IMPROVEMENT**: Various bytecode optimisations
- **FEATURE**: STDLIB Math functions
- **FEATURE**: Added `break` and `continue` in loops
- **BUGFIX**: Inline functions

Version 1.5

- **IMPROVEMENT:** Better string interpolation (e.g. $\backslash n$ to represent a newline)
- **IMPROVEMENT:** More accurate typechecking of function arguments
- **IMPROVEMENT:** Better detection of duplicate declarations
- **IMPROVEMENT:** Better inference of numeric types
- **FEATURE:** Boxed types STDLIB implementation
- **FEATURE:** Per method Inline pragma
- **FEATURE:** Interrupts
- **FEATURE:** Coroutines
- **FEATURE:** Eval compiler option
- **FEATURE:** Casts will perform conversion between types by default.
- **BUGFIX:** Patch operations on float type
- **BUGFIX:** Parser bugfix edgecases
- **BUGFIX:** Multiple typechecking bugfix
- Bugfix and improvements

Version 1.4

- **IMPROVEMENT:** Ast optimization
- **IMPROVEMENT:** Constant folding
- **IMPROVEMENT:** Dead branch removal
- **IMPROVEMENT:** Better typechecking in multiple places
- **FEATURE**: Annotations
- **BUGFIX:** Float stack corruption (require firmware update)
- **BUGFIX:** Lexer edge cases
- **BUGFIX:** Patch autocompile script for Windows
- **BUGFIX:** Type casting bug in some local variable declaration
- **BUGFIX:** Array indexing error when using integer types other than `int`
- **BUGFIX:** Multiple bugs regarding inline functions
- Bugfix and improvements

Version 1.3

- Automated tests with full simulator stack starting with this version
- ELF and CLF output formats
- **IMPROVEMENT:** Better error messages
- **IMPROVEMENT:** Better energy management when using the stdlib `EPD` class.
- **FEATURE:** RAM function support
- **FEATURE:** Operator overloading (U8, U32 and U64)
- **FEATURE:** Compiler options and tuning. (e.g. `-O2` vs `-Os`)
- **STDLIB:** `Emulator` class
- **BUGFIX:** Wrong fixity when mixing unary and binary operators
- Bugfix and improvements

Version 1.2

- **IMPROVEMENT:** Optimization for the `EPD` class

- **IMPROVEMENT:** The compiler will immediately abort when trying to compile non-existing files
- **IMPROVEMENT:** Function inlining
- **IMPROVEMENT:** Loop unrolling
- **IMPROVEMENT:** Dead code removal in static blocks
- **IMPROVEMENT:** Bytecode level optimizations
- **FEATURE:** SMK900 EIC (Interrupt Controller) Support
- **FEATURE:** Hardware SPI
- **FEATURE:** `--nodebug` flag to disable debugger trap
- **FEATURE:** `Terminal` standard library support to print characters through EPD
- **FEATURE:** Bytecode Assembly Syntax support
- **FEATURE:** `CircBuf` class for data acquisition.
- **BUGFIX:** Circular buffer will throw error on read when empty
- Bugfix and improvements

Version 1.1

- **BUGFIX:** Multiple bugfix regarding bytecode generation

Version 1.0

- **IMPROVEMENT:** Dead code removal
- **IMPROVEMENT:** Better error logging
- **FEATURE:** `static final` variable support
- **FEATURE:** Array initializer support
- **FEATURE:** Windows x86_64 support
- **BUGFIX:** Local variable will generate garbage code in some cases
- **BUGFIX:** Multiple bugfix regarding bytecode generation
- **BUGFIX:** Multiple bugfix regarding implicit `this`

Version 0.1

- Initial release

2 Getting Started

2.1 Obtain the compiler

The latest version of the compiler can be obtained from our website https://smartrek.io

2.2 Use the Compiler

To compile your source file(s), use the following command

```
smartrekvmc --stdlib /path/to/smk900/ -o <output>.vmf <input1> <input2>
```

This will compile the `<input1>` and `<input2>` files, linking with the standard library located at `/path/to/smk900/` into the binary file `<output>.vmf`

Please note that the `vmf` format is a binary format and is *not* compatible directly with the USB bootloader of the modules. A converter is also provided to convert from `vmf` to `uf2`, the latter being the format recognized by the USB bootloader. See the documentation for the conversion tool for more details.

2.3 Compiler options

2.3.1 `-h, --help`

Print the help text of the compiler.

2.3.2 `--stdlib ARG`

Sets the location of the standard library. This should *always* point to the version of the smk900/ folder matching the version of the compiler. Any other configuration is untested and prone to breaking.

Note the trailing slash at the end of the path. It is mandatory.

By default, this argument does not need to be specified and a version of the standard library is included with the compiler and is used by default

2.3.3 `--nostdlib`

Do not use any standard library, including the one included with the compiler.

2.3.4 `-o, --output ARG`

Sets the output path of the compiler. Depending on the file extension specified, the compiler will either output a `vmf` or `elf` file.

The default for this value is empty, that is the compiler will compile all files and perform linking, but will not output the resulting binary to disk.

2.3.5 `-c, --dump-clf ARG`

Output the `CLF` (internal intermediate representation) to the given file. When using the compiler as a library, this representation can be read back as a `clf` object and used as a starting point for further compilation.

2.3.6 `-O [0, 2, s]`

Sets the optimization level, this will tune the compiler to produce code optimized for speed (with `-O 2`) or size (`-O s`)

2.3.7 --eval STRING

Evaluate the given Common lisp `STRING` before starting compilation. This can be used to load user compiler patches and features at runtime.

2.3.8 --keep-dead-code

This option will cause the compiler to skip the dead code removal step, note that this will cause the resulting file to be too large for the Portia target when using the standard library.

2.3.9 --inline-complexity ARG

Specifies the eagerness of the compiler to inline functions. A larger number means that the compiler will perform inlining in more cases.

This value is an integer above `0`. A sane default value would be around `10`.

2.3.10 -d, --debug

By default, most errors such as parse errors, typecast errors, syntax errors, etc. are trapped and logged to stdout in a standard format. With this flag, no errors are trapped, and the debugger will always activate when an error is triggered.

Note here that the debugger mentioned is the debugger for the `smartrekvmc` program, not a debugger for the VM bytecode.

2.3.11 -n, --nodebug

Disable the debugger completely, when an error is triggered. The compiler will simply exit on error. Useful in scripts

2.3.12 --optimize-ast, --no-optimize-ast

Enable/Disable the AST optimization pass.

2.3.13 --reflection

Include reflection data in the VMF. This allows the VM and the VM Host to access variables and methods by name, but takes a lost of additionnal space in the VMF.

2.3.14 --types-are-a-suggestion

Disable type checking in most places. The compiler will not report unsound cast (e.g. `(int) string`), nor casts that have potential non critical issues (e.g. `(unsigned) int`). The compiler will manage types similarly to version `1.3` with this option, the types only suggesting certain optimization to the compiler. This option is provided primarly to enable legacy code to be used with the new version of the compiler. We recommend fixing the

typechecking issues by adding casts or additional checks instead of using this option.

3 The STVM language

This section is a reference for the STVM language, the input language of the Smartrek Virtual Machine Compiler. This language is statically typed and object-oriented. It is most similar to Java but has a few differences.

3.1 Types

3.1.1 Void

The `void` type is the unit type. It is not stored in memory since it has only one inhabitant. Values of type `void` cannot be created directly from code.

Values of type `void` can be created by coercing any other value to void. Note that this will leak stack, since the void value will never be popped from the stack because the compiler will assume it is not being stored.

```
static void foo () {
    @Coerce(void) 4;
}
```

3.1.2 Boolean

`boolean` is a true/false type. Its literals are `true` and `false`. Internally, any non-zero value is considered as `true`, but truth values created by the compiler are always represented internally as `0` and `1`.

3.1.3 Numeric types

Table 2 describes the different numeric types supported by the compiler. Some of them are native types and some of them are emulated by the compiler or the standard library. Non-native numeric types can't be used as arrays directly.

Table 2: Numeric types supported by the STVM language.

VM Name	Description	Size (Bits)	Native
`byte`	signed	8	Yes
`u8`	unsigned	8	Compiler
`short`	signed	16	Yes
`int`	signed	32	Yes
`unsigned`	unsigned	32	Compiler
`U64`	unsigned	64	Stdlib
`float`	IEEE-754	32	Yes

Number literals are always of type `int` and can then be casted to the desired type. The exception to that are literals containing a decimal point, which are of type `float`. Number literals can be specified in any of the following formats:

```
0x1fCAb87;
0b1001010;
217
-10
813.72
```

In some cases, the compiler can automatically lower the types of the literals. For instance, the following is valid:

```
byte b = 3;
```

But the following is a warning:

```
byte b = 2000;
```

All numbers regardless of their types are stored internally as 32-bit integers (except stdlib based types, which are actually objects). This has some unexpected side effects as shown in the stack section.

Another syntax for `int` literals is to enclose a char between single quotes. The character ASCII value is used as the value of the literal.

```
int i = 'a';
```

3.1.4 Objects

Objects are created from a class with the new syntax.

```
Object object = new Object();
```

All classes are all subclasses of the `Object` class. Field Variables for an instance are accessed using the dot syntax.

```
object.field
```

By convention, all native types start with a lowercase letter and all object types start with an uppercase letter (e.g `Object`, `String`).

3.1.5 Arrays

Arrays are a homogenous collection of known length. Not all types can be used as arrays. Allowed types are:

- Any object type

- byte
- short
- int
- float

Array types are indicated by appending `[]` to the type. For instance an `int` array is of type `int []` and a `byte` array is type `byte []`.

Arrays, contrary to the stack representation, store its content packed. That is a `byte []` uses 1 byte per element, a `short []` uses 2 bytes per element, and everything else use 4 bytes per element.

To create array instances, two types of syntax are provided. The first, simply allocates an array of the given type and length:

```
int array_length = 4;
new byte[10];
new int[array_length];
```

In this case the created array's content is set to 0 before the expression returns the array pointer. Array literals can also be used to predefine the array's content:

```
new byte[] { (byte) 0x00, (byte) 0xFF, (byte) 0x45 };
new int[] { 100, 3000, -19};
```

Array indexing is done with the `[]` syntax:

```
int [] arr = new int [] { 0, 1, 2 };
int element = arr[2];
arr[3] = 4;
```

The length of an array can be obtained by using the following syntax:

```
int len = arr.length;
```

3.1.6 PROGMEM

Progmem byte and int arrays act as read only arrays. Their content is stored directly in the VMF output and is never copied to RAM. They are completely implemented on the compiler side and every lookup is converted to the appropriate FLASH memory lookup native call.

Progmem arrays must be static variables defined with an array literal. The content of those variables will be compiled as the bytecode for a non-callable function. This will allow code to index their content directly from flash. Here is an example of creation for a progmem byte array:

```
static progmem byte[] varName = new byte[] {
    // Include content (comma separated bytes)
    0x00, 0x01, 0x05, 0xFF, 0x82, ...
    };
```

Pointers to `progmem byte` arrays can be passed around to functions using the following syntax:

```
void useProgmemArray(progmem byte arr, int length) {
    for(int i = 0; i < length; i++) {
        dosomething(arr[i]);
    }
}
```

Note that the length is passed as an argument too, since it is not possible to get the length for a `progmem array`. The above example also show that the read syntax for progmem arrays is the same as for traditional arrays.

```
useProgmemArray(varName, 0x100);
```

3.1.7 String

Strings in the STVM language are objects of the native class type `String` but have custom syntax allowing them to be manipulated more easily. Strings literals are enclosed in double quotes:

```
String s = "This is a string";
```

`Strings` are immutable by default, but may be transformed into a character or byte array by using the `String` functions `getBytes` or `asCharArray`. In most cases the compiler is able to optimize these calls to a no op and thus require no copy.

`Strings` can be appended to numbers and other `Strings` to create a new `String` :

```
String sb = "";
String s = sb.append(42.627);
System.out.println(s);
```

The compiler will automatically use the `append` functions when combining `Strings` with the + operator. The following is equivalent to the last example:

```
System.out.println("" + 42.627);
```

The same native functions will be called in both cases.

Strings included in source code can contain escape sequences to include special characters in the string.

```
"Newline:\n,Tab:\t,Backslash:\\,Hex code:\x55, etc"
```

3.1.8 Pointers

Pointer types allow user code to pass stack locations around to be used as a output parameter. Note that while in C arrays are represented by contiguous regions of memory, it is not the case for STVM arrays. This means that it is *not* possible to obtain a pointer to the location `&array[3]` for instance.

Pointers are used automatically when closure capture arguments by reference:

```
int acc = 0;
Method m = [&acc]() { *acc = *acc + 1; };
```

But they may also be created and passed manually to functions in order to provide output parameters:

```
int out2;
int * ptr = &out2;
int out1 = someFunctionWithTwoOutputs(ptr);
// or directly: someFunctionWithTwoOutputs(&out2)
```

Pointers are created using the & unary operator. The compiler will try to find the location of the given element and will return the pointer if the location is on stack. Elements typically stored on the stack by the STVM are static variables and local variables.

Most objects are implicitly stored and passed by pointers in the STVM. The pointer construct defines explicit pointers. For instance, although a `String` object is stored using a pointer on the stack, it is not a pointer object and cannot be dereferenced. The reference to this pointer on stack can be obtained with the & operator, resulting in a `String*`.

In particular, the new operator creates an object of an *implicit* pointer type. This is why you have

```
Object o = new Object();
```

and not

```
Object * o = new Object();
```

3.1.9 Method

Functions and methods in the STVM language can be manipulated and stored in variables. A variable of type `Method` holds a pointer to a STVM function.

In order to create a `Method` value, the pragma `@Method` can be used. The syntax is `@Method CLASS.NAME` where `CLASS` is the name of the class the method is contained in and `NAME` is the name of the method. In case the method is overloaded, it is possible that the name is not sufficient to uniquely identify a method. In that case, if we have the class

```
class Cls {
    static void F(int i) {...}
    static void F(float i) {...}
}
```

We can distinguish the two `F` method with:

```
@Method{ (I) } Cls.F
@Method{ (F) } Cls.F
```

The standard library specifies methods that can be used on `Method` objects, allowing them to be called dynamically for instance.

Other ways to create method objects is by reflection on a method name by using

```
Method.fromName("ClassName.MethodName.")
```

or by loading an array of bytecode and defining its calling convention using

```
Method.fromBytecode(numberOfArgs, numberOfLocals, arrayOfBytecode)
```

Finally, explicit closures also result in a Method object.

3.2 Operators

The available operators are described in this section. The described specifications for the operators match the action on native types. Compiler and Stdlib based types overload these operators to change their meaning in the context of these types. For instance, addition on two `u8` numbers `a` and `b` is compiled as `0xFF & (a + b)`.

Furthermore, if any arithmetic operator is used on a `float` and an `int`, the `int` will be converted to `float` before the operation takes place with two `floats`.

Table 3 show the precedence of the operators. An operator with a higher precedence (higher in the table) binds more strongly than an operator with a lower precedence.

Table 3: Operator Precedence Table

Operators	Description
++ –	Postfix Incrementation
! - ~	Unary operators
* / %	Multiplicative operators
+ -	Additive operators
<< >> >>>	Shift operators
< > <= >=	Relational comparator
== !=	Relational equality
&	Bitwise and
^	Bitwise xor
|	Bitwise or
&&	Logical and
||	Logical or
=	Assignment (and variations)

3.2.1 Ternary operator

```
condition ? ifTrue : ifFalse;
```

Depending on the value of `condition`, evaluating as a `boolean`, the expression returns `ifTrue` or `ifFalse`. This operator is short-circuiting, meaning that only the selected branch is actually evaluated.

3.2.2 Assignment

Assignment to a location is done through the = operator, or one of the following variations:

```
+= -= *= /= %=
<<= >>= >>>=
&= |= ^=
&&= ||=
```

The LHS must evaluate to an lvalue, which can contain array indexing and field access. The final targets are any non-final variable type or array location. Here are some examples:

```
foo = 12;
arr[0] = false;
this.obj[2].field = 0.5;
```

3.2.3 Postfix Incrementation

The postfix incrementation operators `++` and `--` are placed just after an lvalue. The value returned by the operator is the current value of the lvalue. After the value is returned by the operator, the value of the lvalue is incremented by `1`. Here is an example:

```
i++
```

3.2.4 !

Performed on a boolean, returns the opposite value. Internally, it is implemented as `!x` equal to `1 - x`. Here is an example

```
if (!condition) { .... }
```

3.2.5 Unary -

Takes the negative value of its argument. Internally, it is implemented as `- x` equal to `0 - x`. Here is an example:

```
- 200;
- (arr[0])
```

3.2.6 ~

Binary not. Invert each bit in the 32 bit internal representation of the number. Here is an example:

```
~ 0xFFFFFFFF
```

3.2.7 +

Perform addition on the given 32 bit internal representation of the arguments.

If the first operand is a `String`, the `String` is instead appended with the given `String`, `int` or `float`.

```
// simple addition
5 + 7;
// 7 will be converted to 7.0 before the addition
5.2 + 7;
// Simple addition
5.2 + 7.8;
// Result in "foobar", this is a new object, "foo" is not modified
"foo" + "bar";
// Result in "foo7", this is a new object, "foo" is not modified
"foo" + 7;
```

3.2.8 Binary -

Perform subtraction on the given 32 bit internal representation of the arguments.

3.2.9 Unary *

Get the value of a pointer to a stack location. Can be used as a Lvalue of assignment.

```
int foo = 0;
int * ptr = &foo;
*ptr = *ptr + 7; // Increment the value of foo
System.out.print(*ptr); // Print the value of foo (7)
```

3.2.10 *

Perform signed multiplication on the given 32 bit internal representation of the arguments.

3.2.11 /

Perform signed division on the given 32 bit internal representation of the arguments. If the right argument is zero, a `DIVBY0` error is thrown by the VM engine.

3.2.12 %

Perform the remainder operation on the two arguments. It matches the % operation on two `int32_t` for the C compiler used to compile the VM engine. The operation is done on the internal 32bit representation of the arguments.

3.2.13 <<

Perform a left shift. The left argument is shifted by the amount of bits corresponding to the value of the right argument. The operation is done on the internal 32bit representation of the arguments

3.2.14 >>

Perform an arithmetic right shift. That is a right shift with sign-extension. The left argument is shifted by the amount of bits corresponding to the value of the right argument. The operation is done on the internal 32bit representation of the arguments.

3.2.15 >>>

Perform an unsigned right shift. That is a right shift with zero-extension. The left argument is shifted by the amount of bits corresponding to the value of the right argument. The operation is done on the internal 32bit representation of the arguments.

3.2.16 Unary &

Syntax to get the reference of a stack location. Results in a pointer

```
class Eg {
    static int loc1;
    static void f() {
        int * loc1_ptr = &loc1; // Pointer to loc1
        int ** loc1_ptr = &loc1_ptr; // Pointer to the local variable
↪   loc1_ptr
        int v = 7;
        int * var_ptr = &v; // Pointer to the local variable v
    }
}
```

3.2.17 &

Performs the binary and operation on each of the bits in the 32bit internal representation of the arguments.

3.2.18 &&

Takes two booleans and perform the logical operation and on them. Contrary to other languages, this operator is not short-circuiting, both sides of the operator will always be

executed.

3.2.19 |

Performs the binary or operation on each of the bits in the 32bit internal representation of the arguments.

3.2.20 ||

Takes two booleans and perform the logical operation or on them. Contrary to other languages, this operator is not short-circuiting, both sides of the operator will always be executed.

3.2.21 ^

Performs the binary xor operation on each of the bits in the 32bit internal representation of the arguments.

3.2.22 <

Returns a boolean indicating whether the LHS is strictly smaller than the RHS

3.2.23 >

Returns a boolean indicating whether the LHS is strictly larger than the RHS

3.2.24 <=

Returns a boolean indicating whether the LHS is smaller or equal to the RHS

3.2.25 >=

Returns a boolean indicating whether the LHS is larger or equal to the RHS

3.2.26 ==

Returns a boolean indicating whether the LHS is equal to the RHS

3.2.27 !=

Returns a boolean indicating whether the LHS is unequal to the RHS

3.3 If

The `if (condition) { ... } else { ... }` statement can be used to write conditional code.

Just the `if` part:

```
if (condition) {
    // statements
}
```

Both `if` and `else`:

```
if (condition) {
    // statements
} else {
    // statements
}
```

In both cases, `condition` has to evaluate to a `boolean`.

3.4 Loops

Inside of a loop, the `break` and `continue` statements can be used to exist the loop, or to skip to the next iteration, respectively.

3.4.1 While

The `while` loop will continuously evaluate as long as the condition is true.

```
while (condition) {
    // statements
}
```

The `condition` has to evaluate to a `boolean`

3.4.2 For

The `for` loop provide an initializer, where a local variable can be declared, a stop condition set, and an increment statement.

An example of usage:

```
for (int i = 0; i < length; i++) {
    // statement
}
```

3.4.3 Dowhile

The do while loop acts the same as the while loop, but the condition is only checked at the end of the loop's body.

```
do {
    // statements
} while (condition);
```

Here, `condition` must evaluate to a `boolean`.

3.5 Explicit closures

STVM supports explicit closures, also called lambda functions or just lambda in STVM, are anonymous functions where the captured values are explicitly enumerated. Values can be captured by value or by pointers.

The basic syntax for a Lambda function is:

```
[capture1, capture2, ...] ( type1 arg1, type2 arg2, ... ) { code }
```

Capture can be either a local variable name, or a reference to a local variable name. If other type of data need to be captured, they can be moved to a local variable beforehands.

As with other Methods, the `Method.call` family of functions are used to call a method.

```
class ClosureEg {
    public static void foreach(int[] arr, Method m) {
        for(int i = 0; i < arr.length; i++) {
            m.vcall(arr[i]);
        }
    }
    static void foo() {
        // Simple lambda without a capture
        foreach(new int[]{1,2,3,5,8,13},
                [] (int arg) {
                    System.out.println(arg + 7);
                });
        // Close over local i.
        int i = 42;
        foreach(new int[]{1,2,3,5,8,13},
                [i] (int arg) {System.out.println(arg + i);} );
        // Close over reference to accumulator
        int accumulator = 0;
        foreach(new int[] {1,2,3,5,8,13},
                [&accumulator] (int arg) { *accumulator = *accumulator + arg;
↪   });
    }
}
```

Note that everytime a function containing a lambda is called, a heap allocation is required for the frame containing the captures.

3.6 Variables

3.6.1 Local Variables

Local variables are temporary variables created on the stack. These variables are defined in the content of a method or static block.

```
void methodName (int arg1) {
    int local2 = 10;
    Object local3;
    String local4;
    // implicit this is local variable 0

}
```

Note that method arguments (including the implicit `this`) are stored in the same way as local variables, but their values is automatically set when the method is called. These variables are scoped to the nearest block, this means that accessing local variables outside its scope will be prevented by the compiler.

```
int method () {
    if (condition) {
        int foo = 7;
    }

    return foo;
}
```

This scoping allow the compiler to reuse local variables in different branches, reducing the amount of stack required to call a function:

```
void method () {
    if (condition) {
        int foo = 7;
    } else {
        float bar = 0.82;
    }
}
```

It this example the compiler will choose to use the local variable slot `1` for both `foo` and `bar` (the slot `0` being taken by the implicit `this`).

3.6.2 Field Variables

Field variables are the member data of a class. They are defined (with or without a default value) directly in the body of the class.

```
class ObjName {
    int field0 = 0;
    float field1;

    ...
}
```

Fields are accessed as in the following example.

```
ObjName obj = new ObjName();
int fieldValue = obj.field0;
```

Inside a non static method, the `this` argument can be used to access the current instance's fields.

```
float method () {
    this.field0 += 100;
    return this.field0 + this.field1;
}
```

In many cases the `this` can be elided and the field variable can be used as if it were a local variable:

```
float method2 () {
    field0 += 100;
    return field0 + field1;
}
```

3.6.3 Static Variables

Static variables are global variables that are only attached to a class lexically. They are created inside the body of the class they are attached to. If a default value is specified, the expression and variable assignment will be evaluated at the initialization of the VM. Static variables without a default value are set to `0` instead.

```
class ObjName {
    static int static0 = 100;
    static float static1;

    ...
}
```

Static variables are accessed by prepending the name of the class before them:

```
float method () {
    ObjName.static0 += 100;
    return ObjName.static0 + ObjName.static1;
}
```

Inside the class, in static methods, static variables can be accessed directly as if they were local variables:

```
static float method2 () {
    static0 += 100;
    return static0 + static1;
}
```

3.6.4 Static Final Variables

`static final` or `final` variables are class-level entities that are not actually stored in memory but are replaced directly at their usage site. This makes the following

```
static final int foo = 4;
```

Equivalent to the usage of a C define:

```
#define foo (4)
```

The expression on the RHS of the final definition is simply inlined into each usage site. For example, the following two example will produce the exact same bytecode:

```
// Example 1, using final variables
class eg1 {
    // Or equivalently: static final int foo = 4;
    final int foo = 4;

    static {
        perform_action(foo);
        something_else(foo);
    }
}

// Example 2, inlining the value 4 directly.
class eg2 {
    static {
        perform_action(4);
        something_else(4);
    }
}
```

Since this variable type is simply an inline expression, enclosed function calls are repeated each time the variable is used:

```
class eg3 {
    static int counter = 0;
    static int next() {
        return counter++;
    }
    final int foo = eg3.next();

    void main (String args[]) {
        System.out.print("Counter: " + foo); // Counter: 0
        System.out.print("Counter: " + foo); // Counter: 1
    }
}
```

As expected during normal usage, final variables are not writable, since (assuming `static final foo = 4;`) `foo = 10;` would be compiled into `4 = 10;` which is a syntax error. But, if the expression given to the final variable is an lvalue, the expansion result would look like the following:

```
class abusing_final_variables {
    static byte[] tx_buffer = new byte[5];
    final byte rssi = tx_buffer[0];
    final byte voltage = tx_buffer[1];
    final int sensor_value = ((int[]) tx_buffer)[1];

    static void main(String[] args) {
      ...
      rssi = VM.GetRSSI();
      voltage = SleepCtrl.ReadVoltage();
      sensor_value =
      VM.Send(tx_buffer)
      ...
    }
}
```

But expansion of finals in an lvalue location is explicitly forbidden by the compiler.

3.7 Cast

A cast from one type to another can be performed. A cast will, if required, perform the conversion between the source and target types. The syntax of a cast is as follows:

```
(<type>) <exp>
```

Where `<type>` is the type to which the expression `<exp>` is to be cast to.

For instance, the following cast:

```
int v = ...;
(float) v
```

Will convert the integer stored in variable `v` to a float.

This:

```
int v = -2000;
byte b = (byte)v;
```

Will convert the integer to a byte. The byte value will be truncated to 8 bits and the value will now be: `0xFFFFFFD0`, or -208. Note that all stack values are sign-extended to 32bits, so the number is actually `0xFFFFFFD0` and not `0xD0`.

If the target type can't be reasonably obtained from the type of the expression without conversion, the compiler will throw an error. To force the compiler to accept unsafe conversions, use the coerce annotation. If the `@Coerce` annotation is used, the cast will *not* alter the data or perform conversion, it only changes the type marker in the code.

3.8 Classes

In the STVM language, everything has to be contained in a class. Typically, user code will be composed of a single file containing a single abstract class, but it does not have to be

the case. Exactly one class need to have a static method called `main` that takes a single argument, a `String` array. Currently, the radio firmware always pass a `null` pointer as the argument to the function.

```
class ClassName {
    ...
}
```

Classes can contain different type of field and variable definitions, such as field variables, static variables and progmem arrays. Classes also contain functions. The different allowed type of functions are described in the following versions.

Since the number of classes that can be instantiated is limited to 14, it is desirable to mark classes that are not to be instantiated as `abstract`, so they don't use a spot that could be used by other classes. Note that since inheritance is not supported, methods can't be marked as abstract, contrary to other languages such as Java.

3.8.1 Static Blocks

Static blocks contain code to be evaluated at the initialization of the VM.

```
class ClassName {
    static {
        ...
    }
    ...
}
```

In most cases this code will want to check for the BOOTUP trigger, to evaluate code only at the radio module bootup sequence, and not after every error recovery. A more complete example would look like:

```
class ClassName {
    static {
        ...
        int execType = VM.GetExecType();
        if(execType == EVM.EXECTYPE_BOOTUP){
            ....
        }
    }
    ...
}
```

3.8.2 Class Methods

Class methods can be defined to extend the possible operation on an instance of a class:

```
class ClassName {
    int method (int arg) {
        return arg + 1;
    }
```

```
    int method2 () {
        return method (10);
    }
    ...
}
```

Class methods are accessible through their instances. When inside another class method of the same class, the implicit object `this` can be omitted when calling other class methods or accessing field variables.

```
ClassName obj = new ClassName();
int ret = obj.method(10);
int ret2 = obj.method2();
```

3.8.3 Native methods

In order to expose native host methods to STVM, they can be bound to a path in a class. The following syntax is used:

```
abstract class Classname {
    static native ReturnType functionName(Arg0Type arg0, Arg1Type arg1, ...)
↪   = 0x1734;
}
```

This relults in the method at `Classname.functionName` to call the native function with id `0x1734`, which must be defined by the host.

3.8.4 Static Methods

Static methods are methods attached to a class but that do not take an instance of the class as a first implicit argument. Static methods are marked by prepending the keyword `static` to the method declaration.

```
class ClassName {
    static int method (int arg) {
        return arg + 1
    }
    static int method2 () {
        return method (10);
    }
}
```

Static methods are accessible by prepending the class name to the method call. When inside another static method of the same class, the class name can be omitted, as well as for static variables.

```
int ret = ClassName.method(10);
int ret2 = ClassName.method2();
```

3.8.5 Constructors

Constructors are methods that are called automatically at the creation of an object.

The object creation process is as follows

1. A heap object of the right size is allocated
2. Default values for field variables are computed
3. The selected constructor is called

The syntax for a constructor is as follows, note that the name of the class and the name of the constructor must match. Constructors can't be used inside `abstract` classes.

```
class ClassName {
    ClassName (int a, int b) {
        ...
    }
    ClassName () {
        ...
    }
    ...
}
```

The above example define two constructors. The constructor that will be call depend on the types and number of the arguments inside the new expression:

```
ClassName constructor1 = new ClassName(1, 2);
ClassName constructor2 = new ClassName();
```

3.9 Annotations

Annotations can be added at various places in the code to direct the compiler in a more favorable direction.

3.9.1 `@Inline`

ALWAYS inline the current function call. This annotation should be placed just before a function call.

```
public class A {
  public static void main(String args []) {
    @Inline lowCostFunction(1, otherFunction(2));
  }

  static int otherFunction(int a) {return a + 1;}
  static int lowCostFunction(int a, int b) {return a + b;}
}
```

Here in any case, lowCostFunction will be inlined. otherFunction might or might not be inlined, depending on compiler options

3.9.2 @Coerce

Make any cast succeed without any second thought. This can be used for pointer casting or other shenanigans in versions above 1.3 where typechecking might throw errors on some cast that were previously (wrongly) accepted.

```
// Ok, but will leak stack space
(void) 12;
// Ok, it is a downcast
(short) 12;

// Ok, it is an upcast (done automatically in most cases)
byte b;
(int) b;

// Errors, types not compatible
(Object[]) true;

// Compiles, but you will probably never want that to happen
@Coerce (Object[]) true;

// In this case, coerce won't have any effect since the cast is already safe
@Coerce (int) 100;;

To lower the severity of unsafe casts to a warning locally, you can specify
↪  the error type to throw with the annotation:
// Will throw a warning but still compiles
@Coerce{:warn} (Object[]) true;
// Default behavior
@Coerce{:error} (Object[]) true;
```

3.9.3 @on-parse

Evals the argument of the annotation at parsing time. The annotation needs to be placed in a valid position. Can be used to set compiler options locally or to perform fine-tuning of the parsing process.

```
@on-parse{(format t "I will be printed when the expression is parsed")}
someCode();
```

3.9.4 Hints for Methods

Both class and static methods can contain annotations that will hint some possible optimizations for the compiler.

1. Inlining
   With `@Inline` or `@NoInline` it is possible to force the compiler to inline the function at every call site or to force the compiler to never inline the function.
   The default is to let the compiler decide when to inline method calls.

   ```
   public class InlineHints {

       static void @Inline func() {
           // This function will be inlined everytime
       }
   ```

```
    int @NoInline otherFunc() {
        // This function will never be inlined
    }
}
```

2. Keep
   With the `@Keep` annotation, the method will not be removed from the compilation result, even if the method is not directly used elsewhere.

```
public class Keep {

    static void dontKeep() {}

    static void @Keep keep1() {}

    // Multiple annotations can be used if desired
    static void @NoInline @Keep keep2() {}

    static void main(String[] args) {
        // No use of dontKeep, keep1 and keep2
        // Only dontKeep will be deleted
    }
}
```

3.10 Namespacing

The STVM language does not support namespacing, packages or encapsulation.

3.11 Reflection

When the compiler is invoked with the `--reflection` flag, reflection data is included in the `.VMF` file.

This is used to provide named functions and variables as might be required by the host. For instance, a host might decide to call the method `System.onTick` every time a specific event happen, or read the value of a static variable named `Class.Name` to determine some configuration value.

Reflection data is also available from the STVM itself by using the native functions:

```
static native String JVM.refract(int ref, int type);
static native int JVM.reflect(String ref);
static native String JVM.prism(String ref);
static native boolean JVM.reflectionAvailable();
```

With these functions, it is possible to get the name of a static variable, field, class or method. The opposite is also possible: obtain a method from a name (facilitated by the method `Method.fromName`).

3.12 Intermediate Assembly Language

Internally the compiler translates the STVM language code to an intermediate representation before generating the opcodes. This layer of abstraction is useful because it makes linking the different method together much easier. Additionally, the user can leverage this internal representation through a special syntax to directly emit intermediate opcodes.

An example for the Intermetiate Assembly Language (*IAL*) is shown below. Note that since the compiler requires every stack operations be balanced (in terms of the amount of `push` and `pop`) at compile time, the stack impact of the IAL snippet should be specified. A IAL snippet is parsed as an expression and can be used wherever an expression is expected. The content of the IAL is not checked and can thus cause compiler errors as well as hard to debug runtime errors. Thus, this syntax should be used sparingly. A IAL snippet is assumed to be of type `void` by the compiler, if that is not the case, the

```
// Example of the usage of the emulation instruction.
// A stack impact of -1 is specified since the OP_EMULATION opcode that is
// generated by the IAL opcode EMULATION pops one value from the stack.
@-1{EMULATION};
```

A good example of usage of this syntax is in the standard library implementation of the Emulator, a class that can run code from RAM. This function is included with the relevant parts commented.

```
public void run() {
    ...

    // Reserve stack space for user level code.
    // The instructions by the user are not known
    // ahead of time, so in some cases, the EMULATION
    // opcode will trigger a stack push or pop.
    // This will make sure extra stack is reserved
    // for such cases.
    @32{};

    while(pc < length) {
        tmp = 0xFF & buffer[pc];

        // Goto modifies PC
        if (tmp == Emulator.OP_GOTO) {
            ...
        } else if (tmp >= Emulator.OP_IFEQ && tmp <= Emulator.OP_IF_ICMPLE){
            // By coercing to void the compiler is forced not to pop the
            // value from the stack at the end of the statement. As such,
            // the value computed here is used by the next IAL snippet
            @Coerce (void) (tmp | (3 << 16));
            // The pop of the above value is performed here (-1 stack
            // impact). This will balance the stack push and pops for
            // the compiler.
            @-1{EMULATION};
            // This is a IAL level goto. Labels are resolved internally
            // by the linker. Labels must be unique. Note that no number
            // is specified before the IAL code. This is because if the
            // stack impact is 0, it can be omitted.
            @{ (goto "Emulator/run/end") };
```

```
            pc += ((0xFF & buffer[pc+1]) << 8) + (0xFF & buffer[pc+2]);
        } else {
            // Like above, this builds the instruction for the
            // IAL snippet below.
            @Coerce (void) (tmp + ((buffer[pc+1] & 0xFF) << 24)
                    + ((buffer[pc+2] & 0xFF) << 16));
            // Run the instruction, as above
            @-1{EMULATION};
            // This IAL snippet defines a label at the current position.
            @{ (l "Emulator/run/end") };
            pc += pc_inc(tmp);
        }
    }

    // Release the stack space reserved above
    @-32{};
}
```

The IAL syntax is loosely related to the supported opcodes. The table 4 shown the mapping between IAL syntax and the generated `OPCODES`.

Table 4: Intermediate Assembly Level Syntax

IAL	Opcodes
	Generic operations
`NOP`	`OP_NOP`
`POP`	`OP_POP`
`POP2`	`OP_POP2`
`DUP`	`OP_DUP`
`DUP2`	`OP_DUP2`
`SWAP`	`OP_SWAP`
`EMULATION`	`OP_EMULATION OP_NOP OP_NOP`
	Pushing constants
`(LIT b)`	Smallest possible immediate for the integer b
`(BIPUSH b)`	`OP_BIPUSH b`
`(SIPUSH s)`	`OP_SIPUSH s:msb s:lsb`
`(CONST -1)`	`OP_ICONST_M1`
`(CONST 0)`	`OP_ICONST_0`
`(CONST 1)`	`OP_ICONST_1`
`(CONST 2)`	`OP_ICONST_2`
`(CONST 3)`	`OP_ICONST_3`
`(CONST 4)`	`OP_ICONST_4`
`(CONST 5)`	`OP_ICONST_5`
`(CONST 0.0)`	`OP_FCONST_0`
`(CONST 1.0)`	`OP_FCONST_1`
`(CONST 2.0)`	`OP_FCONST_2`
`(LDC arg)`	`OP_LDC arg`
	Variables
`(LOAD ix)`	Smallest instruction to load local `ix`
`(STORE ix)`	Smallest instruction to store local `ix`
`(IINC ix bl)`	`OP_IINC ix bl`

Continued on next page

Continued from previous page

IAL	Opcodes
(GETSTATIC static)	OP_GETSTATIC static:msb static:lsb
(PUTSTATIC static)	OP_PUTSTATIC static:msb static:lsb
(GETFIELD field)	OP_GETFIELD field:msb field:lsb
(PUTFIELD field)	OP_PUTFIELD field:msb field:lsb
	Arithmetic
IADD	OP_IADD
+	OP_IADD
FADD	OP_FADD
-	OP_ISUB
FSUB	OP_FSUB
*	OP_IMUL
FMUL	OP_FMUL
/	OP_IDIV
FDIV	OP_FDIV
%	OP_IREM
INEG	OP_INEG
FNEG	OP_FNEG
<<	OP_ISHL
>>	OP_ISHR
>>>	OP_IUSHR
and	OP_IAND
or	OP_IOR
xor	OP_IXOR
I2F	OP_I2F
F2I	OP_F2I
	Conditionals
FCMP	OP_FCMPL
(IF EQ label 0)	OP_IFEQ plus the appropriate label position
(IF NE label 0)	OP_IFNE plus the appropriate label position
(IF LT label 0)	OP_IFLT plus the appropriate label position
(IF GT label 0)	OP_IFGT plus the appropriate label position
(IF GE label 0)	OP_IFGE plus the appropriate label position
(IF LE label 0)	OP_IFLE plus the appropriate label position
(IF EQ label)	OP_IF_ICMPEQ plus the appropriate label position
(IF NE label)	OP_IF_ICMPNE plus the appropriate label position
(IF LT label)	OP_IF_ICMPLT plus the appropriate label position
(IF GT label)	OP_IF_ICMPGT plus the appropriate label position
(IF GE label)	OP_IF_ICMPGE plus the appropriate label position
(IF LE label)	OP_IF_ICMPLE plus the appropriate label position
(GOTO label)	Generate OP_GOTO inst. plus the appropriate label position
	Calling
(RETURN T)	OP_IRETURN
RETURN	OP_RETURN
(CALL loc)	Generate OP_INVOKESTATIC plus the id for the method loc
(NEW class)	OP_NEW static:msb static:lsb

Continued on next page

Continued from previous page

IAL	Opcodes

`CALLCONT`
`YIELD`
`CONTRET`
	`OP_CALLCONT`
	`OP_YIELD`
	`OP_CONTRET` (Return from interrupt)

	Arrays
`(ALOAD type)`	Generate the correct array load inst. for `type`
`(ASTORE type)`	Generate the correct array store inst. for `type`
`(NEWARRAY type)`	Generate the array creation inst. matching `type`
`ARRAYLENGTH`	`OP_ARRAYLENGTH`
	Linker control
`(L label)`	Generate the given label at the current position
`(LABEL label)`	Generate the given label at the current position
`(PUSHCALL loc)`	Push the method number for `loc` to the stack.
`(PUSHW label)`	Push the label offset to the stack

4 The VM Execution Environment

This section describes the internals of the VM engine and its interaction with the firmware of the radio module. The environment you use might not be a SpiderMesh radio module and although this section contains useful information, not everything will be applicable. The Resources section describes the memory resources of the virtual machine and how the VM uses them to store the objects. The VM State and Entrypoints section describes the conditions under which the VM is started and interrupted.

4.1 Resources

The flash and RAM sizes that are reserved for user VM code varies between radio firmware revisions. Typical values are about 8 KiB of space for flash and 8 KiB of space for RAM. To get the actual RAM amount, the native function `JVM.freeHeapAmount()` will return the size of the free heap block in bytes.

The ram space is used by both the stack and the heap. Most directly accessible values are stored on the stack while indirectly accessed object are stored in the heap.

4.1.1 Stack

All stack objects are 32 bits. Local variables, static variables and intermediary computation values are located on the stack. The unobvious side effect are for native integer types smaller than 32 bits. Both `byte` and `short` are stored **sign extended** to 32 bits. Non-native software unsigned types can be used if unsigned types are really needed.

Let's say we have the following array: `byte arr[] = new byte[] {0x01, 0xF0};` which contain the 16 bit L-E number `0x01F0`. An apparent way to get that value could be:

```
short leNumber = arr[1] + (arr[0] << 8);
```

But let's see what happens in the stack:

```
arr[1]              //  pushes 0xF0 to the stack.
                    //But remember it is sign extended,
                    //  we get { 0xFFFFFFF0 } in the stack
arr[0]              //  pushes 0x01 to the stack.
                    //  So we get { 0x00000001 } in the stack
8                   //  push the constant 8 to the stack
                    //  { 0x00000008, 0x00000001, 0xFFFFFFF0 }
<<                  //  Performs the shift
                    //  { 0x00000100, 0xFFFFFFF0 }
+                   //  Perform the addition
                    //  { 0x000000F0 }
```

In this case, we get the wrong result.

The solution is to use U8 numbers, or manually clipping the sign extension with a bitwise and operator.

```
int leNumber = U8.convert(arr[1]) + (U8.convert(arr[0]) << 8);
```

The stack is normally growing from the bottom of the heap region except inside of a coroutine, where the stack grows inside of the heap memory region inside the continuation. When a continuation is allocated, it must contain enough stack space for its execution. If this is not the case, heap corruption will occur and crash the VM engine. No checks are performed to ensure the stack is large enough.

4.1.2 Heap

Pointer type content is allocated in the heap. The pointer type is an integer id that represents a location in the heap. This pointer can be stored in the stack to indicate the location of an object or of an array. Heap locations use a simple header that contain the id, the length of the memory block and whether the block is an object or an array.

Arrays have a 1 byte type id before their content. The memory footprint of an array is thus:

```
sizeof(VM_heapHeader_t) + 1 + length * sizeof(arraytype)
```

So for instance, a `byte[10]` has a memory footprint of 15 bytes. But, a pointer to this array is probably somewhere in stack, so an additional 4 bytes are used there.

Objects use a 4 byte class id. Then, the class fields are stored (4 bytes each). The memory footprint of an object is thus:

```
sizeof(VM_heapHeader_t) + 4 * ( 1 + number_of_class_fields )
```

For instance, the following class

```
class Eg {
  int a;
  short arr[22];
  byte b;

  static int hello = 7;
}
```

Has three class fields (static variables don't count and are stored separately on the stack) The memory allocation is thus 20 bytes. But, the short array must also be allocated on the heap, Its heap entry will use 49 bytes. Also, the class instance pointer is probably stored somewhere in stack, so an additional 4 bytes are used there too.

1. Heapsearch
   A *heapsearch* happens when a 32bit VM pointer is converted to a RAM pointer. This is necessary to access or edit the content of any heap object. Both arrays and objects are stored on the VM heap. This means that indexing can be quite time-consuming for nested types. For instance, accessing the inner integer in a type `class {int field[];}[]` (written as `objarray[ix].field[ix2]`) does 3 heap searches.
   (a) Locate `objarray`'s content.
   (b) Locate the content of the object at position `ix` in the array.
   (c) Locate the `field`'s content.
   Since the heap is implemented similarly to a linked list, the time required to lookup an object is dependent on the number of objects in the heap and on the *freshness* of the object. (Older objects are further on the list)

4.1.3 MMU

The MMU, or Memory Management Unit of the VM is a virtual memory mapping between VM numbers and real hardware pointers, providing sandboxing for VM programs.

The heap is mapped at `0x80000000-0x80007FFF` with each value in this range representing the

Other MMU mappings can be created by either the host or from the VM. If no methods are provided in a library using the MMU, or if such a library is being developed, the way to work with other pages of the MMU is to coerce the coresponding pointer to the right object type.

For instance, if a memory region in mapped to the MMU page `0x20000000-0x3FFFFFFF`, it can be used:

```
byte[] rawArray = @Coerce(byte[])0x20000000;
rawArray[0] = 5;
System.out.println(rawArray[10000]);
```

Note that heap specific features will *result in an error*. For instance, the length of an array is computed from its heap block size. Since no such block exist for memory regions outside of the heap, calling `rawArray.length` from the sample above would result in an error.

Objects can be created outside of the heap on MMU pages using this syntax:

```
class Eg { int a; Eg(int i) { a = i; } }
Eg o = (@Coerce(Eg)0x20000100) new Eg(144);
System.out.println(rawArray[256]); // -> 144
System.out.println(o.a);           // -> 144
```

Note how the array created in the previous example overlaps with the object. This is due to the fact that no memory management is done for object outside of the heap, the user is resposible to manage memory locations themselves.

Finally, it is possible to create MMU mappings directly from STVM code. This can be used to abstract over some piece of hardware, or to provide a way to easily access memory in a generic way.

Here is an example, mapping a SPI peripheral's registers to a contiguous accessible memory region:

```
class SPIMMU {
    static Spi spi;

    static unsigned readFunction(int ptr, int ix, int size) {
        ptr += ix; // Consider offset, ptr+ix is the real location requested
↪   by the user.
        unsigned ret;
        spi.start();
        // Perform SPI Calls to read SIZE bytes from the spi port.
        ...
        ret = ...;
        spi.stop();
        return ret;
    }

    static unsigned writeFunction(int ptr, int ix, int size, int value) {
        ptr += ix; // Consider offset, ptr+ix is the real location requested
↪   by the user.
        spi.start();
        // Perform SPI Calls to write SIZE bytes to the spi port.
        ...
        spi.stop();
        // Always return 0 from the writeFunction
        return 0;
    }

    static void init(Spi connection) {
        spi = connection;
        JVM.MMU_attach(1, // Page number, page 4 is reserved for the Heap
                       @Method SPIMMU.readFunction,
                       @Method SPIMMU.writeFunction);
    }
}
```

Once the page is attached, pointers inside of this page can be safely used and access will cause a call to the `SPIMMU.readFunction` and `SPIMMU.writeFunction`.

4.2 Configuration

For the VM to be used as the execution environment, multiple configuration actions must be performed.

1. Set the `VM Engine Selection` register to `JVM`. `EVM`, the default, is the legacy VM engine.
2. Enable the desired VM triggers registers (register 15)
3. Upload the VM code to the radio module.

In order to set the registers, the SpiderMesh IDE software can be used. This software is available from our website https://smartrek.io. The figure 1 shows the configuration

of these registers. See the SpiderMesh IDE documentation for more details on using the software to set register values.

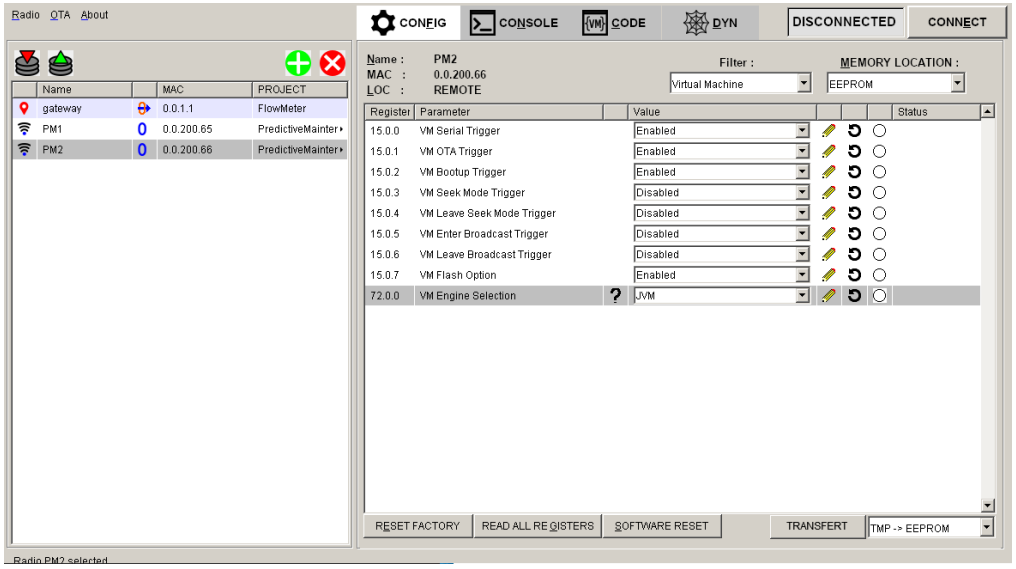

Figure 1: Configuration of the Virtual Machine Parameters from the SpiderMesh IDE Software

After these three conditions are met, the VM will execute starting from the next reset.

4.3 VM State and Entrypoints

There are two types of VM **events**, where the radio firmware will call user code under certain circumstances:

1. Bootup
2. Virtual Machine Triggers (such as `ENTER_SEEKMODE` or `AIRCOMMAND`)

There are three types of VM **entrypoints**, code constructs that determine where in the VM code the execution will start.

1. Static initializers (`static{}` blocks and static variable initialization)
2. Main function call (Call from the start of the `static void main (String args [])` function)
3. Main function resume (The execution will continue at the last instruction executed at the last event)

Table 5 present the properties of the different entrypoints regarding their execution and the action when the engine is killed or interrupted. In this context, an interruption is when the VM return control to the main firmware in a non-normal way. A normal way would be for instance to return from the main function. A non-normal way would be if the engine is stopped when its time limit is reached, or when it is interrupted from software with the native function `JVM.softwait()`

After power-on, the radio module will trigger a **bootup** VM event. This event will call the **static initializer** entry point to init the virtual machine. Then the VM is placed in stopped mode.

Table 5: Interruption effect on the different entrypoints

Entrypoint	Timeout will interrupt	Action when interrupted
Static initializers	No	Error
Main function call	Yes	Pause
Main function resume	Yes	Pause
When in a critical region	Yes	Error

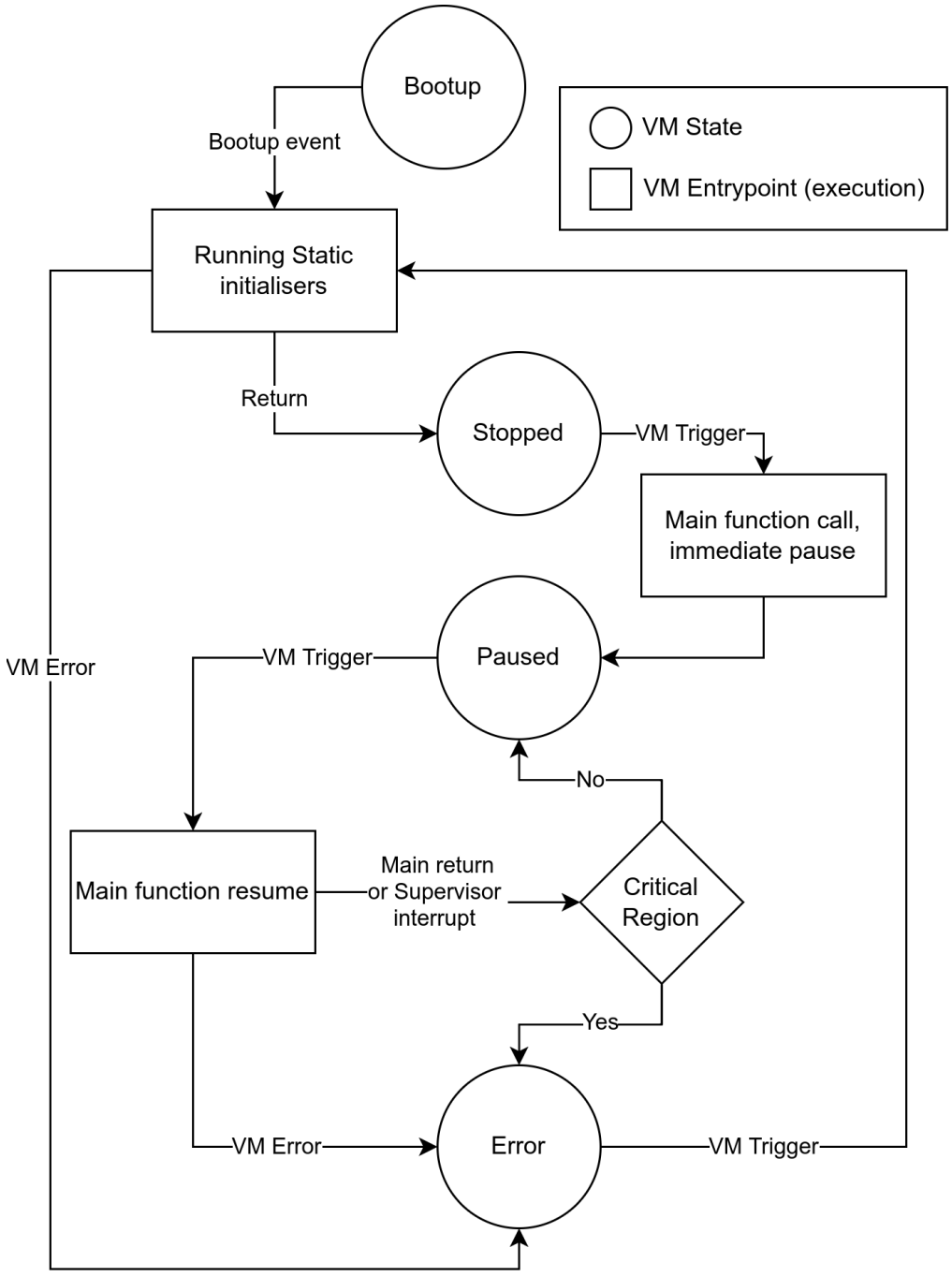

Figure 2: State Diagram of the Virtual Machine during its Execution Cycle

When a **virtual machine trigger** happen afterwards, the entrypoint selected to react to the event will depend on the *current VM state*. The VM can be in any of the following states when a VM trigger occurs

- Stopped
- Paused
- Error

If the VM is in *error* mode, the VM has to be reinited. This may fail, or cause a desync, as there might not be enough time to proceed to the full initialization. To fix this, keep static initializer code as small as possible when the VM trigger is not `EXECTYPE_BOOTUP`. After the initialization is complete, the VM is placed in stopped mode and the VM trigger check is performed again.

If the VM is in stopped mode, the main function is called. The VM is immediately paused at the first instruction of the main function. Then the VM trigger check is performed again.

Finally, if the VM is in paused mode, the execution is resumed at the pause point.

The figure 2 show the different entrypoint and event interaction.

4.3.1 VM Supervisor

The supervisor automatically places the VM in pause mode when there is no more time for execution. This can happen if the radio module has to take control to perform an action such as a SpiderMesh broadcast cycle. The supervisor is also needed for multiple native functions to function correctly such as `VM.Delay`, `JVM.softwait`, or any function related to timing.

A call to a supervisor enabled function while the supervisor is not running will stop JVM execution and crash the module which will cause a WDT Reset.

The WDT is always enabled during the VM execution and can reset the radio module in some cases, even if the VM does not run for more time than allowed, such as bootup. Bootup code may take a long time, or can even never end if desired. In this case, manual calls to `VM.ResetWathdog()` are required to prevent a hardware reset at least every 5 seconds. Table 6 shows the state of the supervisor during the different possible entrypoints. When the supervisor is allowed to pause the VM, it will also automatically reset the WDT.

Table 6: Supervisor properties for the different entrypoints

Entrypoint	Supervisor running	Supervisor can pause VM
Static init (bootup)	Yes	No
Static init (after error)	No	No
VM triggered `main()` execution	Yes	Yes

4.3.2 Critical Region

In some cases, code needs to finish execution before the end of trigger, such as when the `ENTER_SEEKMODE` trigger is used to reset control signals when communication is lost. In this case, the critical region can be used by the programmer to force an error whenever the VM is stopped by the supervisor.

The function `JVM.EnterCriticalRegion()` and `JVM.LeaveCriticalRegion()` can be used to control the critical region state.

4.4 The Garbage Collector

The garbage collector frees unused heap blocs. A bloc is unused when the RAM does not contain an active pointer to it.

The GC is triggered whenever an allocation operation fails because of a lack of memory. The allocation is tried again after the GC completes. The GC can also be triggered manually with the native function `JVM.gc()`

The GC is relatively slow, but shouldn't leak memory. In order to reduce the load on the GC, check the subsection Preventing allocation.

As of version 1.7, the GC more optimally collects small amount of objects. By manually keeping the generated garbage low (by calling `JVM.gc()` frequently for instance), the engine can execute code involving heap objects much quicker.

The GC can't run inside of a continuation.

4.4.1 Preventing allocation

Both allocation and garbage collection is a very slow process. A common pattern when developing VM code for a sensor is allocating an array, filling it, and then sending it using the standard library function `VM.Send(byte[])`.

```
class Sensor {
    static void main(String [] args){
        byte[] tx_buffer = new byte[5];
        tx_buffer [0] = ...;
        ...
        VM.Send(tx_buffer);
    }
}
```

This will allocate a new array on every main call, eventually filling the memory and requiring a GC pass every few executions. This is not a problem, unless your `main` function is time sensitive, but the allocation is easy to prevent by reusing the same array every time:

```
class Sensor {
    static byte[] tx_buffer = new byte[5];
    static void main(String [] args) {
        ...
        tx_buffer[0] = ...;
        ...
        VM.Send(tx_buffer);
        ...
    }
}
```

Another example of allocation is when using the serial port to help debug VM code:

```
System.out.println("foo: " + foo);
```

This statement will create multiple copies and allocations, as explained in the String and StringBuilder section. It is possible to split the call to println in such a way that no allocation or copy is performed.

```
System.out.print("foo: ");
System.out.println(foo);
```

This can reduce the strain on the GC when the print operation is done often.

4.5 VM Errors

The virtual machine will abort on many types of errors. When that happens, the VM will restart as described in SECTION. The error value is stored internally and can be accessed by the user through the `int JVM.GetError()` function. For debugging, a print statement could be used at JVM initialization that outputs the error code to the serial port.

```
System.out.println("[SMK900] VM Init :" + JVM.GetError());
```

The table 7 show the error types that can be triggered by the VM and why.

4.6 Opcodes

The table 8 list the opcodes supported by the VM environment

Every instruction of the STVM language is encoded on 8 bits. Furthermore, up to two bytes can follow the instruction that are used as arguments for the instruction. In the table below, shortcuts are used to refer to these bytes

- `bh`: The upper byte of the argument, or `pc+1`
- `bl`: The lower byte of the argument, or `pc+2`
- `w`: The 16 bit representation of `bh, bl`, `bl` being the LSB and `bh` being the MSB

Most instructions affect the stack directly. In the table below, shortcuts are used to refer to different stack elements:

- `TOS` the most recent entry to the stack
- `NOS` the second most recent entry to the stack
- `stack[-i]` The ith most recent entry to the stack
- `locals[i]` The ith local variable (`locals` being though of here as a pointer to the first local in the stack)
- `static[i]` The ith static variable (`static` being though of here as a pointer to the first static in the stack)

If `NOS` and `TOS` are used, they are popped after the instruction if appropriate.

Some pseudo functions are used in the pseudocode description of the opcodes:

- `push(x)`: Push `x` to the stack. `x` becomes the `TOS` and the old `TOS` becomes the `NOS`

Table 7: Virtual Machine Error Codes

ID	Name	Description
0	NO_ERROR	
1	HEAP_CORRUPTED	The heap format is invalid
2	HEAP_OOM	No space to alloc requested memory
3	HEAP_OOM_ID	All possible memory ids are used
4	HEAP_OOR	Could not find heap pointer in RAM
5	STACK_OOM	Can't reserve stack memory
6	STACK_UNDERRUN	Stack has a negative length
7	STACK_CORRUPTED	Return from method PC is invalid
8	ARRAY_TYPE	Trying to create an array of an invalid type
9	UNKNOWN_METHOD	Trying to call an invalid method
10	UNKNOWN_CLASS	Class or method not found
11	UNCALLABLE	Calling a non callable Object (not a closure)
12	FILE_FORMAT	Code too large
13	SEGFAULT	Used when a memory copy uses an invalid segment
14	OPCODE	Invalid opcode
15	DIVBY0	Division by 0
16	USER1	User reserved error code
17	USER2	User reserved error code
18		Reserved
19		Reserved
20	STACK_OVERRUN	Internal stack management error
21	JIT_COMPILE	JIT Compilation failed
22	JIT_LINK	JIT Linking Step failed
23	JIT_OTHER	Other JIT related errors

- `pop(x)`: Returns the `TOS` and pops it from the stack
- `jump(x)`: Sets `pc` to `x`
- `call(x)`: Sets `pc` to the start of the given method.
- `return`: Return from a `void` method
- `return(x)`: Return the value `x` from a non `void` method

Note that opcodes starting with `OP_I` acts on `int` and opcodes starting with `OP_F` acts on `float`.

Table 8: Opcodes Supported in the VM Execution Environment

Mnemonic	Opcode	#	Args	Description
Generic operations				
OP_NOP	0x00	0		No operation
OP_POP	0x57	0		`pop()`
OP_POP2	0x58	0		`pop() pop()`
OP_DUP	0x59	0		`push(TOS)` (TOS not popped)
OP_DUP2	0x5c	0		`push(NOS) push(TOS)` (TOS/NOS not popped)
OP_SWAP	0x5f	0		`push(TOS) push(NOS)`
OP_EMULATION	0xcb	0		See section OP_EMULATION
Pushing constants				
OP_ICONST_M1	0x02	0		`push(-1)`
OP_ICONST_0	0x03	0		`push(0)`
OP_ICONST_1	0x04	0		`push(1)`
OP_ICONST_2	0x05	0		`push(2)`
OP_ICONST_3	0x06	0		`push(3)`
OP_ICONST_4	0x07	0		`push(4)`
OP_ICONST_5	0x08	0		`push(5)`
OP_FCONST_0	0x0b	0		`push(0.0)`
OP_FCONST_1	0x0c	0		`push(1.0)`
OP_FCONST_2	0x0d	0		`push(2.0)`
OP_BIPUSH	0x10	1	bh	`push(bh)`
OP_SIPUSH	0x11	2	w	`push(w)`
OP_LDC	0x12	1	bh	See section OP_LDC
Variables				
OP_ILOAD	0x15	1	bh	`push(locals[bh])`
OP_FLOAD	0x17	1	bh	`push(locals[bh])`
OP_ILOAD_0	0x1a	0		`push(locals[0])`
OP_ILOAD_1	0x1b	0		`push(locals[1])`
OP_ILOAD_2	0x1c	0		`push(locals[2])`
OP_ILOAD_3	0x1d	0		`push(locals[3])`
OP_FLOAD_0	0x22	0		`push(locals[0])`
OP_FLOAD_1	0x23	0		`push(locals[1])`
OP_FLOAD_2	0x24	0		`push(locals[2])`
OP_FLOAD_3	0x25	0		`push(locals[3])`
OP_ISTORE	0x36	1	bh	`locals[bh] = pop()`
OP_FSTORE	0x38	1	bh	`locals[bh] = pop()`
OP_ISTORE_0	0x3b	0		`locals[0] = pop()`
OP_ISTORE_1	0x3c	0		`locals[1] = pop()`

Continued on next page

Continued from previous page

Mnemonic	Opcode	#	Args	Description
OP_ISTORE_2	0x3d	0		`locals[2] = pop()`
OP_FSTORE_0	0x44	0		`locals[0] = pop()`
OP_FSTORE_1	0x44	0		`locals[1] = pop()`
OP_FSTORE_2	0x45	0		`locals[2] = pop()`
OP_IINC	0x84	2	bh bl	`locals[bh] = locals[bh] + bl`
OP_GETSTATIC	0xb2	2	w	`push(static[w]))`
OP_PUTSTATIC	0xb3	2	w	`static[w] = pop()`
OP_GETFIELD	0xb4	2	w	Push the field w of the object at TOS
OP_PUTFIELD	0xb5	2	w	Sets the field w of object NOS to TOS
OP_DEREFGET	0x18	0		Dereferentiate the pointer at the given location
OP_DEREFSET	0x19	0		Sets the value in the pointer at the given location
OP_LOCALREF	0x1e	2	w	Get the offset of the given local variable
Arithmetic				
OP_IADD	0x60	0		`push(NOS + TOS)`
OP_FADD	0x62	0		`push(NOS + TOS)`
OP_ISUB	0x64	0		`push(NOS - TOS)`
OP_FSUB	0x66	0		`push(NOS - TOS)`
OP_IMUL	0x68	0		`push(NOS * TOS)`
OP_FMUL	0x6a	0		`push(NOS * TOS)`
OP_IDIV	0x6c	0		`push(NOS / TOS)`
OP_FDIV	0x6e	0		`push(NOS / TOS)`
OP_IREM	0x70	0		`push(NOS % TOS)`
OP_INEG	0x74	0		`push(-TOS)`
OP_FNEG	0x76	0		`push(-TOS)`
OP_ISHL	0x78	0		`push(NOS <<TOS)`
OP_ISHR	0x7a	0		`push(NOS >>TOS)`
OP_IUSHR	0x7c	0		`push(NOS >>>TOS)`
OP_IAND	0x7e	0		`push(NOS & TOS)`
OP_IOR	0x80	0		`push(NOS | TOS)`
OP_IXOR	0x82	0		`push(NOS ^ TOS)`
OP_I2F	0x86	0		Convert TOS from `int` to `float`
OP_F2I	0x8b	0		Truncate TOS from `float` to `int`
Conditionals				
OP_FCMPL	0x95	0		`NOS < TOS ? -1 : NOS > TOS ? 1 : 0`
OP_FCMPG	0x96			`NOS < TOS ? -1 : NOS > TOS ? 1 : 0`
OP_IFEQ	0x99	2	w	`if(pop() eq 0): jump(w)`
OP_IFNE	0x9a	2	w	`if(pop() neq 0): jump(w)`
OP_IFLT	0x9b	2	w	`if(pop() < 0): jump(w)`
OP_IFGE	0x9c	2	w	`if(pop() ge 0): jump(w)`
OP_IFGT	0x9d	2	w	`if(pop() > 0): jump(w)`
OP_IFLE	0x9e	2	w	`if(pop() le 0 ): jump(w)`
OP_IF_ICMPEQ	0x9f	2	w	`if(NOS eq TOS): jump(w)`
OP_IF_ICMPNE	0xa0	2	w	`if(NOS neq TOS): jump(w)`
OP_IF_ICMPLT	0xa1	2	w	`if(NOS < TOS): jump(w)`
OP_IF_ICMPGE	0xa2	2	w	`if(NOS ge TOS): jump(w)`

Continued on next page

Continued from previous page

Mnemonic	Opcode	#	Args	Description
OP_IF_ICMPGT	0xa3	2	w	`if(NOS > TOS): jump(w)`
OP_IF_ICMPLE	0xa4	2	w	`if(NOS le TOS ): jump(w)`
OP_GOTO	0xa7	2	w	`jump(w)`
Calling				
OP_IRETURN	0xac	0		`return(pop())`
OP_FRETURN	0xae	0		`return(pop())`
OP_RETURN	0xb1	0		`return`
OP_INVOKESPECIAL	0xb7	0		See the section on the Invoke opcode
OP_INVOKESTATIC	0xb8	2	bh bl	See the section on the Invoke opcode
OP_NEW	0xbb	2	w	Alloc a pointer for an object of class w
Arrays				
OP_IALOAD	0x2e	0		`push(NOS[TOS])`
OP_FALOAD	0x30	0		`push(NOS[TOS])`
OP_AALOAD	0x32	0		`push(NOS[TOS])`
OP_BALOAD	0x33	0		`push(NOS[TOS])`
OP_SALOAD	0x35	0		`push(NOS[TOS])`
OP_IASTORE	0x4f	0		`stack[-2][NOS] = TOS`
OP_FASTORE	0x51	0		`stack[-2][NOS] = TOS`
OP_AASTORE	0x53	0		`stack[-2][NOS] = TOS`
OP_BASTORE	0x54	0		`stack[-2][NOS] = TOS`
OP_SASTORE	0x56	0		`stack[-2][NOS] = TOS`
OP_NEWARRAY	0xbc	1	bh	Alloc a pointer for an array of type bh (length TOS)
OP_ANEWARRAY	0xbd	0		Alloc a pointer for an object array of length TOS
OP_ARRAYLENGTH	0xbe			Push the length of the array pointed at by TOS

4.6.1 OP_LDC

Since instructions are limited to a 16 bit argument, the largest immediate that can be loaded from code is a 16 bit value, as the instruction OP_SIPUSH is doing. When a larger constant has to be loaded, the *constant page* is used. A flash section of the VMF is reserved for 32bit constant values. These values are stored in a continuous block of memory. The OP_LDC instruction will index this memory and push the 32bit value located at the given position. The position is given as a 1 bit argument bh. See the following example that show a possible opcode compilation for two example constants:

```
class ClassName {
    static void main (String [] args) {
        int x = 0x66554433;
          // OP_LDC 0x00 ; Load the first constant in memory, 0x66554433
          // OP_ISTORE_1 ; Write the constant to the local variable
        System.out.print ("This is a string");
          // OP_LDC 0x01 ; Load the pointer to the string,
          //    a 32bit constant
          // OP_INVOKESTATIC <System.out.print(LString;)>
          //    ; Call the function
    }
}
```

4.6.2 OP_INVOKEXXX

The invoke family of opcodes are used to call native functions, static methods and class methods. The two opcodes in this family `OP_INVOKESPECIAL` and `OP_INVOKESTATIC` work in the same way and differ only on where the method to call is located. `OP_INVOKESTATIC` takes a `16` bit argument representing the id of the method or function to call. This id is predefined for native functions and determined at linking time by the compiler for user defined methods. `OP_INVOKESPECIAL` pops an argument from the stack and uses it as the method id. This opcode is mainly used in the call methods for the `Method` object.

A native call will simply hook into C level firmware code and continue to the next instruction.

A method call will push to the stack information required to return from the function and also will reserve the right amount of stack for the called method. This amount of memory is calculated ahead of time by the compiler.

A jit method will use the target specific calling convention for jit methods.

The arguments to the method, including the leading `this` object for class methods are to be pushed to the stack prior to the invoke instruction. The last argument is the `TOS` (after the method id is popped by `OP_INVOKESPECIAL` if applicable) and older arguments are to be pushed first.

4.6.3 OP_EMULATION

The emulation instruction pop a 32 bit value from the stack. The value is interpreted as in table 9

Table 9: OP_EMULATION Opcode Value Interpretation Format

Bits	Description
24-31	bh (MSB of w)
16-23	bl (LSB of w)
8-15	Unused
0-7	Opcode

Note that if the emulated instruction is a call, the return instruction normally return to `pc+3`, since in a typical invoke instruction, there are two bytes that are arguments and not instructions after the invoke opcode byte. Note that other instructions that expect arguments will also advance `pc` by their expected number of arguments. In any case, it is always preferred to append two `OP_NOP` opcodes after any `OP_EMULATION` instruction to make sure the VM interpreter stays aligned to an instruction and doesn't start to interpret argument data as instructions.

4.7 Intricacies

This section describes corner cases in the VM execution. Some examples of optimizations here may already be done by the compiler.

4.7.1 Getting bytes of an integer

The representation of a int array is

```
Byte:   0   1  2  3  4    5  6  7  8
Value:  10  <  arr[0]  >  <  arr[1] >
```

The representation of a byte array is:

```
Byte:   0      1      2      3      4      5      6      7      8
Value:  8  arr[0] arr[1] arr[2] arr[3] arr[4] arr[5] arr[6] arr[7]
```

By coercing from `int[]` to `byte[]` and back, it is easy and space efficient to access and modify the bytes of an `int` or `int` array. By using such a cast:

```
int[] iarr = new int[1];
iarr[0] = 0x11223344;
byte[] barr = iarr;      // (GET__ iarr)
                         // (STORE barr)
barr[2] = 0x55;          // (LOAD barr)
                         // I_CONST_2
                         // (BIPUSH 0x55)
                         // BASTORE
System.out.println(iarr[0]); // 0x11553344
```

Without including the local variable, we see that the write only require a single instruction BASTORE

The traditional way would produce:

```
int[] iarr = new int[1];
iarr[0] = 0x11223344;
iarr[0] = (iarr[0]           // (LOAD iarr)
                             // I_CONST_0
                             // (LOAD iarr)
                             // I_CONST_0
                             // IALOAD
           & 0xFF00FFFF)     // (LDC 0xFF00FFFF)
                             // IAND
           | (0x55 << 16);   // (BIPUSH 0x55)
                             // (BIPUSH 16)
                             // ISHL
                             // IOR
                             // IASTORE
System.out.println(iarr[0]); // 0x11553344
```

And require the allocation of constants for the mask and for simplifying `0x55 « 16` into `0x00550000` when possible.

4.7.2 Static Init Evaluation Order

Static variables initialization values (e.g. `static int foo = 3;`) and static blocks (e.g. `static { foo(); }`) are evaluated whenever the VM gets initialized. The compiler

guarantees that all static variables will be defined before the static blocks are executed. The order of evaluation of static variables initialization values and static block are *undefined* and is subject to change between versions.

The current version evaluates both the static variables and static blocks in reverse order. For instance if we compile the two following files:

```
// file1.java
class file1 {
    static int A = C;
    static int B = A;
}

// file2.java
class file2 {
    static int C = 8;
    static {
        // Block D
    }
    static int E = 1;
}
```

with the following command line:

```
smartrekvmc file1.java file2.java
```

The evaluation order is as follows

- Static variables initialization
  - `file2.java`, as the evaluation is reverse
    - E gets assigned to `1`
    - C gets assigned to `8`
  - `file1.java`
    - B gets assigned to the value of `A`, most likely `0` at this time of the execution, but the actual value is undefined
    - A gets assigned to the value of `C`, which is `8`
- Static blocks are executed
  - The block `D` executes

To prevent the error for `B`, it is possible to move the variable declaration, but the recommended approach is to use static final variables for constants, which are replaced at compile time

Note that the standard library files are added at the end of the file list when added to the compilation is started. This means that every standard library static initializer is executed before any user code.

This is an example for a recent version of the compiler, but execution order *should not* be relied upon.

5 Standard library

The standard library is documented using JavaDoc like comments on the methods and classes of the library. Please refer to your implementation of the stdlib for details. Most native functions are also documented there.