# Nap virtual machine description and behavior

The nap virtual machine is an application virtual machine designed and implemented to run the compiled bytecode of the nap scripting language. It is a virtual machine based on the specifications of a hypothetical computer: the nap machine. From a technical point of view it is a process running on a (real) host operating system, interpreting the bytecode commands of the virtual machine and performing specific actions.

The virtual machine is created when the process starts, and it is destroyed when the application it runs ends, or there is an unrecoverable error.

## Components

The Nap virtual machine has the following components:

* registers – to move data around
* flags – to handle the result of Boolean operations
* stack – for tracking the variables
* built in functions – for easy approach of complex notions
* opcodes – to be executed
* return value – to be populated at the return from a function

## Registers

There are 256 registers in the virtual machine for the basic types supported by the VM

* int: reg integer(0) ... reg integer (255)
* byte: reg byte(0) ... reg byte (255)
* real: reg real(0) ... reg real(255)
* string: reg string(0) ... reg string(255)

There are also another 255 registers for generic types: reg gen(0) .. reg gen(255). These are used for moving object references.

## Underlying architecture

The int registers are represented as int64\_t C types. Generally, every number (ie: variable) you use in *nap* maps to int64\_t.

The real registers and values are represented as double.

The bool values at the internal implementation map to int.

The byte values map to an uint8\_t value. They can be specified in the script in either as a normal number or as a character in single quotes (‘A’ -> 65). The compiler automatically transofrms the character into its numeric value.

The string registers on the internal implementation map to a Unicode string (encoding is UTF-32 Big Endian) however when used with the *nap* runtime API you will get strings as per your default system encoding and character representation (ie: null terminated char\*).

On the implementation side the string registers always hold a copy of the actual string that was passed into them. Firstly we de-allocate the memory of a string register (free) and then allocate the required size (as per UTF-32 BE encoding) and copy the required data in the register itself. This is valid for all initializations.

## Index registers

Index registers: there are 256 available index registers (thus allowing arrays with 256 dimensions to be used). They are populated by the interpreter and they are NOT automatically cleaned by a @ccidx call, but the user must manually insert a clidx instruction to clear the index registers.

Registers are owned by the threads operating them, so there is no such thing as context swap on thread swapping.

Each bytecode file contains the layout of the classes defined in them, this way the executor knows how to handle the class methods, specifically the attributes.

## Binary codes for type specifiers

Table 1: bytecode types

|  |  |
| --- | --- |
| int | 0x01 |
| real | 0x02 |
| string | 0x03 |
| char | 0x04 |
| byte | 0x08 |
| immediate value | 0x11 |
| ref | 0xEF |
| variable | 0xFE |

Immediate value size specifier:

|  |  |  |  |
| --- | --- | --- | --- |
| byte | 0x8 | A byte type, 8 bit | uint8\_t |
| short | 0x16 | A short type, 16 bit | uint16\_t |
| long | 0x32 | A long type 32 bits | uint32\_t |
| huge | 0x64 | A 64 bit value | uint64\_t |

Immediate values are always written as unsigned values to the bytecode stream, and they are preceded by a size specifier. If the first bit of the size specifier is set, the value is to be interpreted as a signed value.

## Stack

The Nap stack is local to each active thread in the system. It contains:

* the variables that the current operations can access
* call context delimiters (named markers) - on each call-context opening a new opening delimiter is pushed on the stack, and on each call context closing operation the variables are purged from the stack. For objects the destructor if any is called.
* The stack has an internal stack pointer, which always starts from zero and grows. This stack pointer is not supposed to be accessed by either the application programming interface or any assembly level commands.

## Strings to number conversion

When converting a string to a number the following conventions are used:

* String starts with 0, it is considered octal
* String starts with 0x it is considered hexadecimal
* String starts with 0b it is considered to be binary
* Anything else is considered decimal

## Flags

The results of logical operations is stored in a boolean flag, called the LBF (Last Boolean Flag) and it is modified automatically by the operations that compare things (eq, lt, gt, ...) and the first jlbf (Jump if Last Boolean Flag is set) or clbf (clear Boolean flag) clears this flag, so that it can be used safely by the next operation.

### lbf

The last boolean result flag. This flag is set by the first comparison operator (eq, neq, gt, lt, ...) and cleared by the first access via jlbf or by clbf command.

This flag has three states:

* true
* false
* undecided

The default state of this flag is undecided. The jlbf or by clbf commands set the flag state to undecided.

# Return values

The special registers rv <type> denote a return value. They are automatically populated by the return opcode and their value must be fetched from the rv registerrs in order for it to be usable by the code.

# Built in functions

The following built in functions are interpreted by the virtual machine

### @#ccidx

Returns the value of the multi dimensioned variable var\_name at the position [reg\_idx(0), reg\_idx(1), ... , reg\_idx(number\_of\_indexes - 1)]

In case one of the indexes is out of range for the variable it throws an exception and halts the executin of the application.

##### Syntax

**@#ccidx**(<var\_name>, <number\_of\_indexes>)

##### Parameters

var\_name - the name of the variable. In the nap source file it was declared as <type> var\_name[x,y,..,z] where x, y, …, z are the dimensions of the indexes.

number\_of\_indexes - the number of indexes that were used in the call. For a two dimensional matrix this is two.

In case you use this function on a dynamic dimension variable and the index is greater than the current size the following will happen:

### @#grow

Adds a new dimension to the variable and updates the internal data structures of the bytecode interpreter application. Does not guarantee to preserve the existing data.

In the output bytecode the @#grow always must be preceded with a call mnemonic, but the compiler will replace the call opcode with an internal opcode. Do NOT use the internal opcode (if you have reverse engineered it from the bytecode file). It’s internal and undocumented.

##### Syntax

**call @#grow**(<var\_name>, <dimensions>)

##### Parameters

var\_name - the name of the variable

dimension - the new dimension. In case the dimension is -1 the variable will be treated as dynamic dimension variable and in case of access appropriate operations will be taken (the variable will be resized to fit the largest index that was requested. If this happens on a read the returned value will be zero, on a write it will put the required value at the (newly) allocated space). Multi dimension variables may not have dynamic dimensions, if they have the application might behave

### @#crea

Creates an instance of the given class into the given variable name. It will NOT call the default constructor, all the properties of the class will be initialized to a default value of 0 for numbers and "" (empty) for strings. If the class contains other class type variables, they will be initialized with another @#crea call.

The layout of the class is stored in the **.class** section of the bytecode file.

##### Syntax

**@#crea**(<class\_name>, <var\_name>)

##### Parameters

class\_name – the name of the class which is supposed to be created

var\_name – the target variable.

### @#len

Returns the length of the parameter which can only be a string variable

##### Syntax

**@#len**(<string\_variable>

##### Parameters

string\_variable – the variable whose length is supposed to be calculated

# Op-codes

The nap-vm is an opcode based virtual machine similar to real life hardware. The following groups of opcodes are in the emulated hardware:

* Mathematical operations
* Bitwise operations
* Data manipulation operations
* Comparison operations
* Jumping operations
* Stack handling operations

## Mathematical operations

The following mathematical operations do three things:

1. do an operation between the target and source
2. Store the result in target

### add

add target, source

Adds the source to the target, stores the result in target.

target - register/variable/indexed

source - register/variable/indexed/number

#### Binary opcode

|  |  |
| --- | --- |
| add | 0xAD |

### sub

sub target, source

Subtracts the source from the target, stores the result in target.

target - register/variable/indexed

source - register/variable/indexed/number

#### Binary opcode

|  |  |
| --- | --- |
| sub | 0x8D |

### mul

mul target, source

Multiplies the source with the target, stores the result in target.

target - register/variable/indexed

source - register/variable/indexed/number

#### Binary opcode

|  |  |
| --- | --- |
| mul | 0xA8 |

### div

div target, source

Divides the source with the target, stores the result in target. In case source is zero, an Exception is thrown

target - register/variable/indexed

source - register/variable/indexed/number

#### Binary opcode

|  |  |
| --- | --- |
| div | 0xD8 |

### mod

mod target, source

Performs the modulus operation between the source and the target, stores the result in target.

target - register/variable/indexed

source - register/variable/indexed/number

### inc

inc target

The target is incremented and the result is stored in the same location.

target - register/variable/indexed

#### Binary opcode

|  |  |
| --- | --- |
| inc | 0x1C |

### dec

dec target

The target is decremented and the result is stored in the same location.

target - register/variable/indexed

### neg

neg target

Negates the sign of the target and stores the result in the same location.

target - register/variable/indexed

### pos

pos target

Makes the sign of the target to be positive, and stores the result in the same location.

target - register/variable/indexed

## Bitwise operations

The following operations do three things:

1. do the operation between the target and source

2. Store the result in target

3. Store the result of the operation in the LBF

### and

and target, source

Performs the "and" operation between the source and the target, stores the result in target.

target - register/variable/indexed

source - register/variable/indexed/number

### or

or target, source

Performs the "or" operation between the source and the target, stores the result in target.

target - register/variable/indexed

source - register/variable/indexed/number

### xor

xor target, source

Performs the "xor" operation between the source and the target, stores the result in target.

target - register/variable/indexed

source - register/variable/indexed/number

### shl

shl target, count

Shifts left the target count times, stores the result in target.

target - register/variable/indexed

count - register/variable/indexed/number

### shr

shr target, count

Shifts right the target count times, stores the result in target.

target - register/variable/indexed

count - register/variable/indexed/number

### bcom

bcom target, source

Performs the bitwise complementary operation between the source and the target, stores the result in target.

target - register/variable/indexed

source - register/variable/indexed/number

### not

not target

The "not" operation is applied to the target and the result is stored in the same location.

target - register/variable/indexed

## Data opcodes

### mov

mov target, source

Moves the source into the target

target - can be any register/variable/indexed

source - can be register/variable/indexed/number

One of the operands always needs to be a register, moving variable into another variable is not allowed.

#### Binary opcode

|  |  |
| --- | --- |
| mov | 0xB8 |

Format of the bytecode stream.

0xB8 <TARGET> <SOURCE>

where TARGET is:

EE <REGISTER\_SPECIFIERS>

FE <VARIABLE\_SPECIFIERS>

Where

REGISTER\_SPECIFIER is:

<01|02|03><8BIT\_INDEX> depending on the type (int, real, string)

VARIABLE\_SPECIFIER is:

<32BIT\_INDEX>

and SOURCE is:

11 <IMMEDIATE\_DESCRIPTOR>

FE <VARIABLE\_SPECIFIER>

where IMMEDIATE\_DESCRIPTOR is:

<SIZE\_SPECIFIER><VALUE>

where SIZE\_SPECIFIER is either 0x08, 0x16, 0x32, 0x64 and VALUE is a value 8, 16, 32, 64 bits long

### clidx

clidx

Clears the index registers to default 0

## Comparison opcodes

The following operations do things:

1. do the comparison operation
2. If the lbf is either true or false they perform an “and” operation between the result obtained at 1 else Store the result of the comparison in the LBF

This logic allows code like: 1 < a < 5 to be interpreted correctly. The nap-script compiler generates code for “full” boolean operations (such as 1 < a and a < 5) which does not use this shortcut, but this quick comparison will result in code using this feature.

### gt

gt target, source

Performs the greater than comparison operation between the source and the target, stores the result in target in lbf.

target - register

source - register/variable/indexed/number

### gte

gte target, source

Performs the greater than or equal comparison operation between the source and the target, stores the result in target in lbf.

target - register

source - register/variable/indexed/number

### lt

lt target, source

Performs the less than comparison operation between the source and the target, stores the result in target in lbf.

target - register

source - register/variable/indexed/number

### lte

lte target, source

Performs the less than or equal comparison operation between the source and the target, stores the result in target in lbf.

target - register

source - register/variable/indexed/number

### eq

eq target, source

Checks if the source and the target are equal or not, stores the result in target in lbf.

target - register

source - register/variable/indexed/number

#### Binary opcode

|  |  |
| --- | --- |
| eq | 0xE0 |

### neq

neq target, source

Checks if the source and the target are equal or not, stores the result in target in lbf.

target - register

source - register/variable/indexed/number

## Jump opcodes

### jlbf

jlbf label

Performs a jump to the given label if the last boolean flag is set to true, and sets the value of the last boolean flag to undecided.

label - is the label where the jump will go.

### jmp

jmp label

Performs an unconditional jump. label - is the label where the jump will go.

label - is the label where the jump will go.

### call

Calls the procedure specified in the first parameter. The procedure can be either internal (i.e.: defined in this source file), or external (i.e.: defined in an imported file) or can be a bound procedure to the host application (defined via the nap API).

In case of the bytecode generated by the standard nap compiler upon a call the code will also contain a pushall and popall command in order to ensure the correct state of the registers.

#### Syntax

For calling a simple procedure or a function:

call @procedure

For calling members methods of objects:

call $object@Class.method

This will initialize the “this” object of the executor thread to the value after the $ sign

#### Binary opcode

|  |  |
| --- | --- |
| call | 0xCA |

#### Bytecode representation

The call opcode in the bytecode file is followed by a 32 bit index value representing the location in the bytecode stream where the called method is to be found.

### leave

Leaves the current procedure and returns to the caller.

## Stack handling commands

The following commands are responsible for handling the stack:

### marksn

Puts a named marker on the stack for the virtual machine bytecode interpreter to mark the location of a section where new variables are to be declared. This marker is put there automatically on every block start by the compiled code which is generated by the compiler but you can also insert markers manually. In this case you are responsible for manually managing these markers. The virtual machine will know that on a clrsn operation it needs to delete the content of the stack till this point.

#### Opcode

|  |  |
| --- | --- |
| marksn | 0xB7 |

#### Syntax

marksn <marker\_name>

Places a marker called marker\_name on the stack

#### Example

marksn test\_marker

### clrsn

Instructs the virtual machine to erase all the objects on the stack till it finds a specific marker (set by marksn). In case there is no matching marker found on the stack the VM will exit with a stack underflow error message. Also will remove the marker object itself.

#### Opcode

|  |  |
| --- | --- |
| clrsn | 0xB6 |

#### Syntax

clrsn <marker\_name>

Removes the objects from the stack till the marker called marker\_name.

#### Example

clrsn test\_marker

### push

Pushes is a multi-purpose command, and it’s used to push a variable, a register or an immediate value onto the stack (For variables also has a mode of declaring the variable name for further access). A variable can be pushed more than once onto the stack, but only the specific declaration push will declare it.

#### Opcode

|  |  |
| --- | --- |
| push | 0xBC |

#### Syntax

The push mnemonic has a non-uniform parameterization depending on what is being achieved. It can be used to declare a variable, or simply push a value onto the stack.

##### Declaring a variable using push

After the push instruction comes the type specifier for the value being initialized (int, float, string). After the type specifier follows the fully qualified name of the variable.

###### Example

push int global.z

Will declare a variable z in the global namespace

##### Pushing a register on the stack

push reg int (0)

##### Pushing a variable on the stack

push global.z

##### Pushing an immediate value on the stack

push 23

##### Bytecode stream

1. Pushing a variable: 0xBC <uint8\_t: var\_type> 0xFE <uint32\_t: index>

where

* 1. 0xBC – the bytecode for the push
  2. <var\_type> is from Table 1
  3. 0xFE – indicates that we are pushing a variable
  4. <index> is the index of the variable from the meta section

### peek

Peeks a variable with a specified type from the stack from a given position (also it declares it for further access). Peek does not remove the variable from the stack. The typical usage of peek is at a function call, where it declares the parameters of the function as local variables without removing them

#### Opcode

|  |  |
| --- | --- |
| peek | 0xCD |

#### Syntax

peek <type> <immediate> <variable>

The peek command is followed by a type identifier (int, real, string) and an immediate value (a number, representing the zero based index of the variable from the top of the stack), then the name of the variable we want to put the value in.

The index is like: 0 – for the last element pushed on the stack, 1 for the previous, etc…

### retv

Instructs the virtual machine to populate the return value of the current function with the parameter of the command.

|  |  |
| --- | --- |
| retv | 0xFD |

### leave

Instructs the virtual machine to leave the current function and jump to the last entry in the call frames vector, decreasing the call frame counter.

### pop

Instructs the virtual machine to fetch the last element from the stack and place it in the target of the operation. Also removes the element from the stack.

#### Opcode

|  |  |
| --- | --- |
| pop | 0xCB |

#### Syntax

After the pop mnemonic comes the type indicator.:

1. Popping a register – the keyword reg followed by the type of it (int, real, string) and the index, which to improve readability can be enclosed in parentheses
2. Popping a variable – the mnemonic is followed by the variable name with full context

##### Example – popping a register

**pop** reg int (0)

##### Example – popping a variable

**pop** global.y

#### Example

int y = 1;

asm

{

mov reg int (0), global.y

**add** reg int (0), 32

**push** reg int (0)

pop global.y

}

After running the example above the value of the y variable should be 33.

### pushref

Pushes the reference of an object onto the stack

### pushall

Pushes all the registers in the internal stack

### popall

Pops the registers from the internal stack

## Miscellaneous

### clbf

Clears the Last Boolean Flag, resetting its state to unknown.

## Construction of objects

The constructor of a class always has to push a “this” onto the stack in case it was invoked with an assignment. If it was not invoked with an assignment, but with a @crea function call, it will not invoke a push to the stack of the this.

## Format of the bytecode file

|  |  |  |  |
| --- | --- | --- | --- |
| Start | Field | Size | Explanation |
| 0x00 | file type | 8 bits | it is either 0x32 or 0x64 depending on the size of the addresses |
| 0x01 | meta address  MA | 32bits | The address of the meta section of the file. |
| 0x05 | stringtable address  SA | 32bits | The address of the stringtable of the file. All the strings that were extracted in the application will go to this specific location. |
| 0x09 | jumptable address  JA | 32bits | The address of the jumptable |
| 0x0d | Function table address  FA | 32bits | The address of the function table |
| 0x11 | Register count | 8 bits | How many registers are needed when running this application |
| 0x12 | Flags | 8 bits | A reserved set of flags that will be used for various purposes, such as to denote if there is debug information in the file, etc… |
| 0x13 | Bytecode of the application | ... | This is the bytecode of the application. |
| $MA | The meta section | ?? | The meta section contains the variables of the application in a full call context qualified name. |
| $SA | The stringtable of the application | ?? | Contains the strings of the application. All string entries are in UTF-32BE format, and upon load they will be converted to the system locale using the iconv library. |
| $JA | The jumptable of the application | ?? | Contains the jump locations of the application |

### The Meta section

The meta section is formatted like the text “.meta” followed by a list of entries.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Start | Field | Size | Explanation |
|  | $MA | .meta | 5 bytes | The .meta string indicating the start of the meta section |
|  | $MA + 0x05 bytes | count | 32 bits | The number of variable in the file |
| LIST | $MA + 0x05 bytes + 32 bits | var\_index | 32/64 bits | the index of the variable on the stack |
|  | $MA + 0x05 bytes + 32/64bits + 32 bits | var\_type | 8 bits | The type of this variable: 0 this is a variable from this VM, 1 this is a variable from a VM up in the VM chain |
|  | $MA + 0x05 bytes + 32/64bits + 32 bits + 8bits | var\_name\_len | 16 bits | The length of the name of the variable, only if this is a global variable. If it’s not global, it will be zero. |
|  | $MA + 0x05 bytes + 32/64bits + 16 bits + 32 bits + 8bits | var\_name | $var\_name\_len bytes | The name of the variable, but only f it is a global variable ie: var\_name\_leng != 0 |

### The string table

The string table is similar to the meta section.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Start | Field | Size | Explanation |
|  | $SA | .str | 4 bytes | The .str string indicating the start of the string section |
|  | $SA + 0x04 bytes | count | 32 bits | The number of stringtable entries |
| LIST | $SA + 0x04 bytes + 32 bits | string\_index | 32/64 bits | the index of the string as referred to in the bytecode |
|  | $SA + 0x04 bytes + 32/64 bits + 32 bits | string\_length | 32 bits | the length of this specific string |
|  | $SA + 0x04 bytes + 32/64 bits + 32 bits + 32 bits | the\_string | $string\_length bytes | The string itself |

### The jumptable

the jumptable looks like

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | Start | Field | Size | Explanation |
|  | | $JA | .jmp | 4 bytes | The .jmp string indicating the start of the jumptable section |
|  | | $JA + 0x04 bytes | count | 32 bits | the number of elements in the jumptable |
| LIST | | $JA + 0x04 bytes + 32 bits | bytecode\_location | 32/64 bits | the location in the bytecode |
|  | | $JA + 0x04 bytes + 32 bits + 32 bits | label\_type | 8 bits | 0 if this is a simple jump label, 1 if this is a function, 2 if this is the method of a class |
|  | type  1  ||  2 | $JA + 0x04 bytes + 32 bits + 32 bits + 8 | label\_length | 16 bits | The length of the name of the label. Used only if this label represents a method, function call |
|  | type  1  ||  2 | $JA + 0x04 bytes + 32 bits + 32 bits + 16 bits + 8bits | label\_name | <label\_length> bytes | The fully qualified label name, used only if the label represents a method/function call |

### The function table

The function table looks like:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Start | Field | Size | Explanation |
|  | $FA | .fun | 4 bytes | The .fun string indicating the start of the function table section |
|  | $FA + 4 bytes | count | 32 bits | the number of elements in the fun table |
| LIST < count> elements | $FA + 4 bytes + 32 bits | Jumptable index | 32 bits | The index of the function in the jumptable |
|  | $FA + 4 bytes + 32 bits + 32 bits | Fun name length | 16 bits | The length of the function name |
|  | $FA + 4 bytes + 32 bits + 32 bits | Function name | <Fun name length> bytes | The function name. ASCII only |
|  | $FA + 4 bytes + 32 bits + 32 bits + 16bits + <Fun name length> bytes | Return type | 8 bits | The return type |
|  | $FA + 4 bytes + 32 bits + 32 bits + 16bits + 8bits <Fun name length> bytes | Number of parameters | 8 bits | The number of parameters |
| LIST1 < Number of parameters> elements | $FA + 4 bytes + 32 bits + 32 bits + 8bits + 8 bits + 16 bits <Fun name length> bytes + <number of parameters before this> bytes | Parameter type | 8 bits | The type of a parameter:  Same as from the opcodes list |

### Variable handling in the bytecode

Every time a variable is being referred to in the bytecode during the code generation the application searches for the variable and if it’s the first time it’s used it creates a new entry in the meta table. Into the bytecode stream what goes out is the meta\_location of the variable (which is effectively the index of the variable in the meta vector, not the location of the variable).

Same is valid for jumptable and string table objects: index is used in bytecode instead of real value.

### FUTURE STUFF: The binary file

Each method has its own stack. Written to the binary file. The application on quit might save the stacks. Next time will continue from where it left last time.

Each variable is identified by a unique number generated at compile time, this is written to the file. The variables are referred with this number later during compile

They are being pushed to the proper stack obtained at compile time. There is a map of variable id's and stacks.

# The nap runtime

The nap runtime is a library binding the compiler and the virtual machine in order to allow the clients of the library to execute nap script in an embedded environment.

# Interrupts

The interrupt system is responsible for handling out of VM requests, such as mounting a file in the virtual file system or printing output to the screen. Extensions to the interrupt system are stored either in external DLLs and loaded as necessary upon user invocation, or represented as a series of assembler commands.

### Interrupt 4

Executes an external function. Load

# regs 0 -> teh signature, first character is the return type

# regs 1 -> the name of the function

# regs 2 -> the name of the library (- if none)

# regi 0 -> the index in the pregenerated table

# Run Time compilation

This is invoked by the “execute” nap script command. This command takes in a string or a number, indicating the nap commands or the nap command block.

This command always returns a number, indicating the bytecode block id where the compiled code resides.

The VM has two dedicated interrupts (2 and 3) which points back to the compiler. Interrupt 2 takes in the regs(0) and produces a full nap bytecode file, with the following exception:

1. It takes a reference to the already running VM stack (not the stack of the VM, but the list of VMs that are running), and when searching for a function or variable it extends its search towards this stack.
2. Methods declared in the “sub” will be visible from the OTHER “subs” but not from the main, since at the time of compilation those methods do not exist, but at the next level of compilation they do.
   1. Unless in the “main” these methods were declared with “forward”.
      1. If during the execution of a “main” VM a “forward” method was executed, but it had no definition in the “sub”-compiled VMs the execution will either stop or do nothing …

Interrupt 2 returns a value in regi(0) indicating the bytecode block id.

Interrupt 3 takes in an integer in regi(0) and executes the bytecode found at the given location.

In the nap compiler generated code these two are outputted as the result of compiling nap\_execute() immediately one after the other.

## How the nap VM runs an application

An application is represented by a single bytecode file.

// TODO

## Debug information in the file

The nap compiler can compile in debug info in the file if requested.

In this case the names of the “inner” variables will be put in the meta table.

The following is the debug section of the file:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Start | Field | Size | Explanation |
|  | $DBG | .dbg | 4 bytes | The .dbg string indicating the start of the debug section |
|  | $DBG + 4 bytes | dbg\_count | 32bits | The number of debug info objects in the file |
|  | $DBG + 4 +32 bits | dbg\_file\_count | 16bits | The number of participating files in this bytecode file (ie: how many files were imported while compiling this file) |
| LiST1  DBG\_FILES | $DBG + 4 bytes + 32 bits + 16 bits | file\_index | 16 bits | The index of the participating file |
|  | $DBG + 4 bytes + 32 bits + 16 bits + 16 bits | file\_name\_length | 16 bits | The length of the file name |
|  | $DBG + 4 bytes + 32 bits + 16 bits + 16 bits + 16 bits | file\_name | file\_name\_length \* 8bits | The name of the file |
| LIST2 | $DBG + 4 bytes + 32 bits + 16 bits + sizeof(LIST1) | debug\_info | dbg\_count \* sizeof(debug\_info) | A list of debug\_info structures |

The Debug section comes after the jumptable.

### The Debug Info structure

The following is the debug info structure:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Field Name | Field type | Size | Explanation |
|  | bytecode\_start | uint32\_t | 32 bits | The location in the bytecode stream of where the bytecode for this debug\_info starts. |
|  | bytecode\_end | uint32\_t | 32 bits | The location in the bytecode stream of where the bytecode for this debug\_info ends.  Between bytecode\_start and bytecode\_end there can be more assembly commands each participating in the current operation |
|  | file\_index | int16\_t | 16 bits | The index of the file in which the command was found. Reference to DBG\_FILES section. |
|  | start\_line | int16\_t | 16 bits | The start line in the file where the command is |
|  | end\_line | int16\_t | 16 bits | The end line in the file where the command is |
|  | command\_length | int32\_t | 32 bits | The length of the command |
|  | command | string | <command\_length> bytes | The actual command for the debugging without source. These last two fields are enabled with a special compiler switch (enable sourceless debug or similar) |
|  | patch\_command\_length | int32\_t | 32 bits | The length of the patch command |
|  | Patch command | string | <patch:command\_length> bytes | The actual patch command for the original command. See below. |

## Patching the debug file

When the sourceless debug feature is enabled while debugging it is possible to modify the commands that are stored in the file. This is the workflow

1. User debugs
2. User finds bug in a line of code
3. User chooses to patch a line
   1. Writes the commands in the debugger shell
   2. The debugger calls the compiler to compile the modified commands (in the same manner as the “nap\_execute” is being compiled).
   3. The debugger fetches out from the newly compiled the exact bytecodes for the new command.
   4. The debugger fetches the bytecodes for the “old” command and compares the length with the length of the bytecodes of the new command.
      1. If they are equal, simply replaces them in the memory and in the file
      2. If they differ (new is longer(shorter):
         1. Creates a new mem buffer with correct size
         2. Copies the old content till the start of the command
         3. Copies the new commands’ bytecodes
         4. Copies the old content after the old command bytecodes
         5. Patches the jumptable to have the correct values (ie: adds (substracts) the difference between new size / old size to the values)
         6. Patches the debug\_info structures to have the correct value
   5. The debugger writes the memory back to the disk
4. The debugger can restart

## How to save floats in the binary file

When the application reads a float value from the source file it splits it in two parts: first is the number without the decimal points (ie: multiplied with 10 on the Nth) and the second is the N:

1.2345 = 12345 \* 10^(-4)

What goes to the file:

12345

4

## External functions

1. in the .out file write the definiton of the external methods (a new section in the file is requried for this)

2. when calling an external function instead:

    a. move in regs[0] the name of the function

    b. push the variables normally on the stack

    c. issue an intr\_4 call

    d. intr\_4 does the following:

          i. from regs[0] takes the name of the function.

          ii. from the bytecode file loads the external functions definition and according to the parameter list will pop from the stack the values in order (either peeks an int or a string or a float)

          iii. with the peeked values it will create a nap\_ext\_par\_desc structure

          iv. from the return type and the parameter list (or already defined by the compiler) it will identify the signature of the function (such as: viiis)

          v. from the signature it will identify the correct nap\_ext\_call\_v\_\_i function and call it.

          vi. if the return type is not void it will handle correctly the return value by populating the proper rv register