

# The nano VM Manual

Version 1.3.11

for nano vm 2.0.0

Copyright © 2012 Stefan Pietzonke

[jay-t@gmx.net](mailto:jay-t@gmx.net)

21<sup>th</sup> January 2012



# Contents

1. Intro	5
2. The Basics	6
2.1 The Registers	6
2.2 Move constants into registers	6
2.3 Move registers into variables	6
2.4 Move registers into register	7
2.5 Variable declaration	7
2.6 First program	7
2.7 Opcodes	8
3. Console i/o	9
3.1 Console output	9
3.2 Console input	10
3.3 Opcodes	11
4. Preprocessor	12
4.1 Register names	12
4.2 Includes	12
4.3 Line parser	13
4.4 Opcodes	14
5. Program flow	15
5.1 Jumps	15
5.2 Subroutine calls	15
5.3 The Stack	16
5.4 Opcodes	19
6. Math operators	20
7. Variables	22
7.1 Arrays	22
7.1.1 Opcodes	25
7.2 Strings	25
7.2.1 Opcodes	26
8. Type conversion	27
9. Time functions	27
10. Internal variables	28
11. Files	29
11.1 File handling	29
11.2 Opcodes	32
11.3 Error codes	33
12. Memory	34
12.1 Organization	34
12.2 Virtual memory	34
12.3 Error codes	35
13. TCP/IP Sockets	36
13.1 Introduction	36
13.2 Client/Server	36
13.3 Opcodes	42
13.4 Error codes	43
14. Processes	43
14.1 Introduction	43
14.2 Opcodes	43
14.3 Error codes	43
15. Shell arguments	44
15.1 Introduction	44
15.2 Opcodes	44
16. Usage	45
16.1 The assembler nanao	45
16.2 The VM nano	45
16.3 Installation	45
16.4 Compiling	47
17. Pointers	48
18. Hash function	49
19. Threads	49
20. Appendix	52
21. Internal Design	53



# 1. Intro

The nano VM is a register based virtual machine. The VM uses assembler as the programming language. It's possible to write complex programs. I wrote a small webserver with it.

The main features are:

- data types: byte, short int, long int, double and string.
- arrays
- ANSI output functions: text styles, text locating, cursor moving...
- file i/o
- TCP/IP sockets
- virtual memory support for machines without MMU
- portable (100% C)
- licensed under the GPL
- new (1.1.9) pthreads support!

This manual starts with the simple things and includes a lot examples.

## History

I started this project somewhere in 2002. I wanted to find out, if I can write my own VM. If someone asks me why? Then my answer is: "why not?". I did it just for fun.

Also I got new skills while developing it. I learned how to do the TCP/IP stuff and much more. I won't miss this experience.

## Version 2.0.0 README!

I made some big changes in the 2.0.0 release:

First of all: strings are now operated in string registers. You have to use **push\_s** and **pull\_s**! And some other new stuff needed for a threading safe environment.

This breaks compatibility with the 1.x versions, and I'm rewriting the examples now. A few of them are already converted and are available via subversion with the source:

```
svn co https://nano-vm.svn.sourceforge.net/svnroot/nano-vm nano-vm
```

The latest stuff will allways be there.

And the flow graphics server is hosted on sourceforge too:

```
svn checkout svn://svn.code.sf.net/p/flow-server/code-0/trunk flow-server-code-0
```

## 2.0 The Basics

### 2.1 The Registers

The nano virtual machine is register based. There are 32 registers for long integer- and double numbers. To do something with numbers, they must be moved into registers first. We can move a constant or a variable into a register.

### 2.2 Move constants into registers

Lets say we want to move "10" into register "0". We want to move an integer number, so the register will be "L0". The "L" before the register number stands for long integer. The opcode to do this is "push". This looks like this:

```
push_i      10, L0;
```

The "push\_i" opcodes means push a short integer number into an integer register. The semicolon ";" marks the end of the opcode. The assembler ignores all lines without a semicolon. An exception are preprocessor commands. They are explained later in this manual.

To move 40000 into register "1", we would do this:

```
push_l      40000L, L1;
```

The "L" behind the number marks it as long int. The ranges for the integer numbers are:

```
byte          : 0 to 255  
int (short int): -32768 to 32767  
lint (long int): -2147483648 to 2147483649
```

For double numbers (floating point), we have to use "push\_d":

```
push_d      123.456, D0;
```

The range of double numbers is big:

```
double: -1.7*10^-308 to 1.7*10^308
```

To move the int variable "x" into register "1":

```
push_i      x, L1;
```

### 2.3 Move registers into variables

To move registers into variables we use the "pull" opcode:

```
pull_i      L0, x;
```

Moves register "L0" into variable "x". The variable must be of int type.  
An example for a double register:

```
pull_d      D0, z;
```

## 2.4 Move registers into registers

We can move the contents from one register to another:

```
move_l    L1, L2;
```

This moves register “L1” to “L2”.  
And the double example:

```
move_d    D1, D2;
```

## 2.5 Variable declaration

We have five types of variables: **byte**, **int**, **lint**, **double** and **string**.

To declare a byte variable, which can hold a value from 0 to 255, we do this:

```
byte b;
```

For all other types it's the same thing. We have to declare the type and the variable name.

To **declare a string** we have to create an **array**:

```
string s[13];
```

This string “s” can store **12 chars**. You have to set the array size 1 bigger than the string length.

And this is a **double array** example:

```
double d[100];
```

The double array “d” can store **100 double** numbers.

## 2.6 First program

Now we can write a simple program. We declare an int variable and store a value in it:

```
int x;

push_i    10, L0;
pull_i    L0, x;      x = 10

push_i    0, L1;      set return value "0"
exit      L1;         exit program
```

The “exit” opcode ends the program and sets a return value for the shell environment.  
We set “0”, so it means no error.

**Note:** you have to take care that the program flow reaches the “exit” opcode!

If you forget to set “exit”, the assembler stops and prints a warning message. Every program must have an exit point.

The return value is important if you start your program from a shell script. In your script you can check the return value. And for example break the script, if your program failed to do something.

## 2.7 Opcodes

L = long register, D = double register, S = string register  
BV = byte variable, IV = int variable, LV = long int variable  
DV = double variable, SV = string variable  
V = variable, N = integer variable

{ } = optional (arrays)

### Variable declaration

```
byte      BV{[NV]};  
int       IV{[NV]};  
lint     LV{[NV]};  
double   DV{[NV]};  
string   SV[NV];
```

### Variable, constant to register

```
push_b    BV, L;  
push_i    IV, L;  
push_l    LV, L;  
push_d    DV, D;  
push_s    SV, S;
```

### Register to variable

```
pull_b    L, BV;  
pull_i    L, IV;  
pull_l    L, LV;  
pull_d    D, DV;  
pull_s    S, SV;
```

### Register to register

```
move_l    L1, L2;          L1 to L2  
move_d    D1, D2;          D1 to D2  
move_s    S1, S2;          S1 to S2
```

## 3. Console i/o

### 3.1 Console output

To print something in the console, we use the “print” opcode. Here is a “hello world” example:

```
// hello world

push_i      0, L0;
push_i      1, L1;

push_s      "Hello world!", S0;
print_s     S0;
print_n     L1;

exit        L0;
```

The two slashes “//” at the beginning mark the whole line as a comment. The assembler ignores this line.

The “print\_s” opcode prints a string. The “print\_n” opcode prints the number of new lines as set in the register. In this case it is one new line.

Now open a shell and change to the “nano/prog” directory. Start the assembler by typing:

```
nanoa hello
```

In the console you will see something like this:

```
stefan@tux:~/nano/prog$ nanoa hello
nano assembler 0.99.2 (c) 2006 by jay-t@gmx.net
== free software: GPL ==
compiled by gcc version: 3.4.6 on Apr 23 2006
loading program:
hello.na
ok
saving program:
hello.no
ok
```

Now we start the program:

```
nano hello
```

And this is the output:

```
stefan@tux:~/nano/prog$ nano hello
nano vm 0.99.2 (c) 2006 by jay-t@gmx.net
== free software: GPL ==
compiled by gcc version: 3.4.6 on Apr 23 2006
loading program:
hello.no
ok
Hello world!
```

Now type this:

```
nano hello -q
```

And you will see the following:

```
stefan@tux:~/nano/prog$ nano hello -q
Hello world!
```

The “-q” option stands for “quiet”. No start messages are printed. But you will still get error messages if something went wrong.

## 3.2 Console input

To read data from the console we use the “input” opcode. The following example reads two numbers and multiplies them:

```
1 | // calc_4.na
2 |
3 | push_s          "first number: ", S0;
4 | print_s        S0;
5 |
6 | input_s        S0;
7 | val_l          S0, L0;
8 |
9 | push_s          "second number: ", S0;
10| print_s        S0;
11|
12| input_s        S0;
13| val_l          S0, L1;
14|
15| mul_l          L0, L1, L2;
16|
17| print_l        L2;
18| push_i         1, L3;
19| print_n        L3;
20|
21| push_i         0, L4;
22| exit          L4;
```

**Note:** The line numbers are for reference only. They are not a part of the program. The new opcodes are “input\_s”, “val\_l” and “mul\_l”.

With “input\_s” we read a string from the console and store it in the string variable “n”. The “val\_l” opcode converts the string into a number and stores it in the register. The program **reads** two **strings** in line **6** and **12**. And **converts** them **to numbers** in line **7** and **13**.

In line **15** we multiply the registers “L0” and “L1” and store the result in register “L2”. Or in plain math:  $L2 = L0 * L1$ . Line **17** prints the result.

Do you remember how to assemble a program? Now assemble “calc\_4.na” and check if it's really working:

```
stefan@tux:~/nano/prog$ nano calc_4 -q
first number: 200
second number: 10
2000
```

So everything is fine!

### 3.3 Opcodes

L = long register, D = double register  
V = variable, S = string register

#### Console output

print_l	L;	
print_d	D;	
print_s	S;	
print_n	L;	newlines
print_sp	L;	spaces
print_c;		clear console
print_a	L;	prints the char from the ASCII-code of "L" example: "65" -> "A"
print_v	V;	variable name

#### Textstyles:

print_b;		bold
print_i;		italic
print_u;		underline
print_r;		reset to normal style

#### Cursor:

locate	Ly, Lx;	locates cursor at line "Ly" and row "Lx"
curson;		cursor on
cursoff;		cursor off
cursleft	L;	cursor steps to left
cursright	L;	
cursup	L;	
cursdown	L;	

#### Console input

input_l	L;	saves a number in "L"
input_d	D;	
input_s	S;	saves a string in "S"
inputch_l	L;	reads one char and converts to the register type
inputch_d	D;	
inputch_s	S;	

## 4. Preprocessor

### 4.1 Register names

In all previous examples we used the normal register names like “L0” or “L1”. For simple programs this may be OK. But if programs get larger, it will be hard to remember what register “L5” was for. For this cases you can use the “#setreg” opcodes.

**Note:** All preprocessor functions start with a double cross “#”.

To assign name “null” to register “L0” we use “#setreg\_l”:

```
#setreg_l L0, null;
```

And for double registers “#setreg\_d”. Here is a new version of “calc\_4.na” with register names:

```
// calc_5.na

#setreg_l L0, null;
#setreg_l L1, one;
#setreg_l L2, num1;
#setreg_l L3, num2;
#setreg_l L4, mul;

#setreg_s S0, firstnum;
#setreg_s S1, secondnum;

#setreg_s S2, input;

push_i 0, null;
push_i 1, one;

push_s "first number: ", firstnum;
push_s "second number: ", secondnum;

print_s firstnum;

input_s input;
val_l input, num1;

print_s secondnum;

input_s input;
val_l input, num2;

mul_l num1, num2, mul;

print_l mul;
print_n one;

exit null;
```

### 4.2 Includes

With the include function “#include” we can load a nano assembler file into our program. If you know C, then you should be familiar with this.

To include file “foobar.nah” we do this:

```
#include <foobar.nah>
```

Lets look at this in an example. The following program calculates the circumference of a given diameter.

The formula is:

```
circumference =  $\pi$  * diameter
```

```
// circle_1.na
```

```
#include <math.nah>

#setreg_l      L0, null;
#setreg_l      L1, one;
#setreg_l      L2, two;

#setreg_d      D0, pi;
#setreg_d      D1, inp_d;
#setreg_d      D2, circumf;

#setreg_s      S0, inp;
#setreg_s      S1, diameter;
#setreg_s      S2, circumference;

push_i         0, null;
push_i         1, one;
push_i         2, two;

push_s         "diameter:      ", diameter;
push_s         "circumference: ", circumference;

push_d         m_pi, pi;

print_s        diameter;
input_s        inp;

val_d          inp, inp_d;
mul_d          pi, inp_d, circumf;

print_s        circumference;
print_d        circumf;
print_n        two;

exit          null;
```

Line **3** **includes** the “math.nah” file, which defines math constants. In line **19** the “M\_PI” variable is stored in the **pi register**.

### 4.3 Line parser

The line parser functions set the chars for a quote, comma or semicolon. If you want to print this string:

```
“foobar”
```

Then this would not work:

```
print_s    ""foobar"";
```

It would confuse the parser. You have to set a new char for a string begin and end:

```
#setquote  ' ;  
print_s    "'foobar" ';
```

## 4.4 Opcodes

L = long register, D = double register, S = string register  
St = string

### Register names

```
#setreg_l  L, name;  
#setreg_d  D, name;  
#setreg_s  S, name;  
  
#unsetreg_all_l;          unset all L register names  
#unsetreg_all_d;          unset all D register names  
#unsetreg_all_s;          unset all S register names
```

### Includes

```
#include <filename>      includes the file to the program
```

### Line parser

```
#setquote      St;          change quote to string  
#setsepar      St;          change separator to string  
#setsemicolon  St;          change semicolon to string
```

The default parser settings are: a string is surrounded by a double quote: "  
Opcode arguments are separated by a comma: ,

example:

```
#setquote      ' ;  
#setsepar      | ;  
  
print_s        '"foo", "bar" ' ;  
print_n        L0 ;  
  
mul_l          L1 | L2 | L3 ;
```

## 5. Program flow

### 5.1 Jumps

Jumps control the flow through a program. All previous examples ran from top to bottom. The following example prints the numbers from “1” to “10”:

```
1 | // loop_1.na
2 |
3 | #setreg_l L0, null;
4 | #setreg_l L1, one;
5 | #setreg_l L2, loop;
6 | #setreg_l L3, maxloop;
7 |
8 | push_i    0, null;
9 | push_i    1, one;
10 | push_i    1, loop;
11 | push_i    10, maxloop;
12 |
13 | lab printloop;
14 |   print_l  loop;
15 |   print_n  one;
16 |
17 |   inc_l    loop;
18 |   lseq_jump_l loop, maxloop, printloop;
19 |
20 |   exit     null;
```

Line **10** sets the **loop counter** “loop” to “1”. Line **11** sets the “maxloop” register to “10”. Line **13** **declares** the **label** “printloop”. The “inc\_l” opcode in line **17** **increases** the **loop counter** by one.

Line **18**: The “lseq\_jump\_l” opcode **checks if** “loop” is **less or equal** “maxloop”. If this is true, then the program **jumps to** the **label** “printloop”.

### 5.2 Subroutine calls

```
1 | // loop_2.na
2 |
3 | #setreg_l L0, null;
4 | #setreg_l L1, one;
5 | #setreg_l L2, loop;
6 | #setreg_l L3, maxloop;
7 |
8 | push_i    0, null;
9 | push_i    1, one;
10 | push_i    1, loop;
11 | push_i    10, maxloop;
12 |
13 | lab printloop;
14 |   jsr      printnum;
15 |
16 |   inc_l    loop;
17 |   lseq_jump_l loop, maxloop, printloop;
18 |
19 |   exit     null;
20 |
21 | lab printnum;
22 |   print_l  loop;
```

```

23|   print_n     one;
24|   rts;

```

This program does the same thing as the first loop example “loop\_1.na”. Line **21** to **24** are the **subroutine**, which prints the number. In line **14** the “jsr” opcode **calls** the **subroutine** “printnum”. In Line **24** the “rts” opcode **jumps back to** the line **16**.

It's possible to call a subroutine from within a subroutine.

### 5.3 The Stack

On the stack we can store registers and strings. This is useful to give arguments to a subroutine. This is a simple example:

```

        ston;                activate stack

[... ]

        stpush_l   x;        push x to stack
        stpush_l   y;        push y to stack

        jsr        multiply; call subroutine

        stpull_l   num;      get result from stack

[... ]

lab multiply;
        stpull_l   L1;        get second argument from stack (y)
        stpull_l   L0;        get first argument from stack (x)

        mul_l      L0, L1, L2;
        stpush_l   L2;        push result to stack
        rts;

```

The stack fills from bottom to top. If we take something from the stack, we get the object on the top. The object is removed from the stack and saved in a register or string.

The “stpush\_l” opcode pushes a long register to the stack. The “stpull\_l” saves the top of the stack into a long register. The other “stpush” and “stpull” opcodes work the same way.

The “multiply” subroutine is independent from the rest of the program. It can use all 32 registers.

**Note:** normally you have to save the registers before you call the subroutine. And restore the registers after the subroutine ends. I will explain this in the next example.

To save all registers we use “stpush\_all” opcode. They can be restored with “stpull\_all”.

The next example prints a multiplication table:

```

stefan@tux:~/nano/prog$ nano multable_2 -q
 1  2  3  4  5  6  7  8  9 10
 2  4  6  8 10 12 14 16 18 20
 3  6  9 12 15 18 21 24 27 30
 4  8 12 16 20 24 28 32 36 40
 5 10 15 20 25 30 35 40 45 50
 6 12 18 24 30 36 42 48 54 60
 7 14 21 28 35 42 49 56 63 70

```

8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

```

1 | // multable_2.na
2 |
3 | #setreg_l      L0, null;
4 | #setreg_l      L1, one;
5 | #setreg_l      L2, two;
6 | #setreg_l      L3, x;
7 | #setreg_l      L4, y;
8 | #setreg_l      L5, xmax;
9 | #setreg_l      L6, ymax;
10| #setreg_l      L7, numlen;
11| #setreg_l      L8, maxspace;
12| #setreg_l      L9, spaces;
13| #setreg_l      L10, num;
14|
15| #setreg_s      S0, numstr;
16|
17| lint numv;
18|
19| ston;
20|
21| push_i         0, null;
22| push_i         1, one;
23| push_i         2, two;
24| push_i         4, maxspace;
25|
26| push_i         1, y;
27| push_i         10, xmax;
28| push_i         10, ymax;
29|
30| lab yloop;
31|   push_i       1, x;
32|
33| lab xloop;
34|   stpush_all_l;
35|   stpush_l     x;
36|   stpush_l     y;
37|
38|   jsr          mul;
39|
40|   stpull_l     num;
41|   pull_l       num, numv;
42|
43|   stpull_all_l;
44|   push_l       numv, num;
45|
46|   str_l        num, numstr;
47|   strlen       numstr, numlen;
48|
49|   sub_l        maxspace, numlen, spaces;
50|
51|   print_sp     spaces;
52|   print_l      num;
53|
54|   inc_l        x;
55|   lseq_jump_l  x, xmax, xloop;

```

```

56 |     print_n         one;
57 |
58 |     inc_l           y;
59 |     lseq_jump_l    y, ymax, yloop;
60 |
61 |     exit            null;
62 |
63 |
64 | lab mul;
65 |     stpull_l       L0;
66 |     stpull_l       L1;
67 |
68 |     mul_l           L0, L1, L2;
69 |
70 |     stpush_l       L2;
71 |     rts;

```

The line **19 activates** the **stack** with “ston”. The default stack size is 4096 bytes. If you need more space, then you can increase the stack with setting the register “stsize”:

```

#include <vm.nah>

push_l     100000L, stsize;

stack_set;
ston;

```

Line **34 saves all long registers**. Line **35** and **36 push** the arguments “x” and “y” to the **stack**. Line **38 calls** the **subroutine** “mul”.

The **subroutine takes** the **arguments from the stack** (line **65** and **66**). After the multiplication the **result is pushed to the stack**.

The following is a bit tricky. We have to get the result and restore the registers of the main program. First we **store** the **result** from the stack **to variable** “numv” (line **40** and **41**). Then we **take** the **saved registers** from the stack (line **43**). Finally the “num” **register gets** the **return value** from the “numv” variable (line **44**).

## 5.4 Opcodes

L = long register, D = double register  
S = string register

### Jumps

jmp	label;	jumps to label
jsr	label;	jumps to subroutine label
rts;		return from subroutine
jmp_l	L, label;	jumps to the label, if "L" is true (1)
jsr_l	L, label;	jumps to subroutine, if "L" is true (1)
eq_jmp_l	L1, L2, label;	jumps to the label, if "L1" is equal "L2"
neq_jmp_l	L1, L2, label;	not equal
gr_jmp_l	L1, L2, label;	greater
ls_jmp_l	L1, L2, label;	less
greq_jmp_l	L1, L2, label;	greater or equal
lseq_jmp_l	L1, L2, label;	less or equal
eq_jsr_l	L1, L2, label;	jumps to subroutine, if "L1" is equal "L2"
neq_jsr_l	L1, L2, label;	see above!
gr_jsr_l	L1, L2, label;	
ls_jsr_l	L1, L2, label;	
greq_jsr_l	L1, L2, label;	
lseq_jsr_l	L1, L2, label;	

### Stack

ston;		activate stack
stpush_l	L;	store register "L" on stack
stpush_d	D;	
stpush_s	S;	
stpush_all_l;		store all long registers on stack
stpush_all_d;		
stpush_all_s;		
stpull_l	L;	get register "L" from stack
stpull_d	D;	
stpull_s	S;	
stpull_all_l;		get all long registers from stack
stpull_all_d;		
stpull_all_s;		
show_stack;		prints the stack
stelements	L;	returns the number of elements
stgettype	L;	returns the object, see types.nah

## 6. Math operators

Opcodes with more than one argument, return the result in the last register.

L = long register, D = double register  
S = string register

inc_l	L;	increase by one
inc_d	D;	
dec_l	L;	decrease by one
dec_d	D;	
add_l	L, L, L;	
add_d	D, D, D;	
sub_l	L, L, L;	
sub_d	D, D, D;	
mul_l	L, L, L;	
mul_d	D, D, D;	
div_l	L, L, L;	
div_d	D, D, D;	
smul_l	L, L, L;	shift multiply
sdiv_l	L, L, L;	shift division
and_l	L, L, L;	logical and
or_l	L, L, L;	logical or
band_l	L, L, L;	bitwise operators
bor_l	L, L, L;	
bxor_l	L, L, L;	
mod_l	L, L, L;	modulo
eq_l	L, L, L;	equal
eq_d	D, D, L;	
eq_s	S, S, L;	
neq_l	L, L, L;	not equal
neq_d	D, D, L;	
neq_s	S, S, L;	
gr_l	L, L, L;	greater
gr_d	D, D, L;	
ls_l	L, L, L;	less
ls_d	D, D, L;	
greq_l	L, L, L;	greater or equal
greq_d	D, D, L;	
lseq_l	L, L, L;	less or equal
lseq_d	D, D, L;	
srnd_l	L;	initalize random number generator
rand_l	L;	get random number
rand_d	D;	get random number (0 - 1)

abs_l	L, L;	absolute value
abs_d	D, D;	
ceil_d	D, D;	round to upper value
floor_d	D, D;	round to lower value
exp_d	D, D;	
log_d	D, D;	
log10_d	D, D;	
pow_d	D1, D2, D3;	rise D1 by the power of D2
sqrt_d	D, D;	square root
cos_d	D, D;	
sin_d	D, D;	
tan_d	D, D;	
acos_d	D, D;	arcus cosin
asin_d	D, D;	
atan_d	D, D;	
cosh_d	D, D;	hyperbol cosin
sinh_d	D, D;	
tanh_d	D, D;	
not_l	L, L,	logical not

# 7. Variables

## 7.1 Arrays

This creates an array for **10 int** numbers:

```
int array[10];
```

In this case the array size is declared by a constant. You can use a variable too:

```
int array[max];
```

This creates an array with a size of the value of "max".

The following example stores 10 random numbers in an array. And prints them:

```
// array_2.na

    #setreg_l  L0, null;
    #setreg_l  L1, one;
    #setreg_l  L2, i;
    #setreg_l  L3, max;
    #setreg_l  L4, rand;
    #setreg_l  L5, randstart;

    push_i    0, null;
    push_i    1, one;
    push_i    9, max;
    push_i    2006, randstart;

// declare array with space for 10 numbers
    int randa[10];

// initialize random number generator
    srand_l   randstart;

    move_l    null, i;

lab init_randa;
    rand_l    rand;

// store rand in array randa
    move_i_a  rand, randa, i;

    inc_l     i;
    lseq_jump_l i, max, init_randa;

    move_l    null, i;

lab print_randa;

// get random number from array randa
    move_a_i  randa, i, rand;

    print_l   rand;
    print_n   one;

    inc_l     i;
    lseq_jump_l i, max, print_randa;

    exit     null;
```

The “move\_i\_a” opcode **stores** the register “rand” in the array “randa”. The “i” register is the **array index** and sets the position in the array. In the loop the “i” register counts from **0** to **9**.

The “move\_a\_i” opcode **reads** from the array.

## Multi dimensional arrays

Arrays can have more than one dimension. Lets say we have an array with two dimensions:

```
int a[5][5];
```

The array “a” can store **25** numbers. It has **5 rows** and **5 columns**. Now we want to store **4** at row **3** and column **2**. We use “y” for the row and “x” for the column:

```
    0 1 2 3 4 (x)
0 |
1 |
2 |  4
3 |
4 |
```

(y)

So **y = 2** and **x = 1**. But we can only use one index with the array opcodes. We have to calculate the index first:

```
index = y * xsize + x
```

The xsize for this array is **5**:

```
index = 2 * 5 + 1
index = 11
```

The array index is 11. Now here is the example:

```
// array_3.na
// multi dimensional array

#setreg_l    L0, null;
#setreg_l    L1, one;
#setreg_l    L2, index;
#setreg_l    L3, xsize;
#setreg_l    L4, ysize;
#setreg_l    L5, x;
#setreg_l    L6, y;
#setreg_l    L7, num;

#setreg_s    S0, s;

push_i      0, null;
push_i      1, one;
push_i      5, xsize;
push_i      5, ysize;

int xsizev;
int ysizev;

pull_i      xsize, xsizev;
pull_i      ysize, ysizev;
```

```

int a[ysizev][xsizev];

push_i      4, num;
push_i      2, y;
push_i      1, x;

// calculate array index
mul_l       y, xsize, index;
add_l       index, x, index;

move_i_a    num, a, index;

push_s      "stored ", s;
print_s     s;
print_l     num;
push_s      " in index: ", s;
print_s     s;
print_l     index;
print_n     one;

exit        null;

```

## Free arrays

To free the allocated memory of an array, you can use “dealloc”. If a program tries to read or write to a freed array, you get a “overflow” error message.

```
dealloc     a;      free array "a"
```

Nano deallocates all arrays on program end. But you can use this to resize an array.

## Resize arrays

To resize an array we have to deallocate it first. Then we declare it with a new size:

```

int a[10];

[...]

dealloc a;

int a[20];

```

### 7.1.1 Opcodes

L = long register, D = double register, S = string register  
BV = byte variable, IV = int variable, LV = long int variable  
DV = double variable  
LI = array index

#### Register to array

```
move_i_a    L, IV, LI;  
move_l_a    L, LV, LI;  
move_d_a    D, DV, LI;  
move_b_a    L, BV, LI;  
move_s_a    S, SV, LI;
```

#### Array to register

```
move_a_i    IV, LI, L;  
move_a_l    LV, LI, L;  
move_a_d    DV, LI, D;  
move_a_b    BV, LI, L;  
move_a_s    SV, LI, S;
```

## 7.2 Strings

This creates a string with space for **12 chars**:

```
string s[13];
```

To copy some text to the string variable "s", we use "push\_s":

```
push_s      "Hello", s;
```

And add a string:

```
add_s       s, " world!", s;
```

Now the string "s" contains "Hello world!".

And converting it to uppercase:

```
ucase       s;
```

Right, the string "s" is now "HELLO WORLD!".

If we want to get a char from a string, we use "move\_p2s":

```
push_i      6, L0;  
move_p2s    s, L0, ch;
```

This copies the char at position **6** to string "ch". And the string "ch" contains now "W".

To copy a char to a string, we use "move\_s2p":

```
push_i      0, L0;  
push_s      "h", S0;  
move_s2p    S0, s, L0;
```

The string "s" is "hELLO WORLD!".

## String Arrays

This creates a string array with space for **5** strings with a length of **30 chars**:

```
string s_array[5][31];
```

To copy text to the string array, we use "move\_s\_a":

```
push_i      0, L0;
move_s_a    "foo bar", s_array, L0;
```

The "L0" register is the array index.

To copy from a string array to a string, we use "move\_a\_s":

```
push_i      0, L0;
move_a_s    s_array, L0, s;
```

This copies the string to the string register "s".

### 7.2.1 Opcodes

L = long register, D = double register

S = string register

LI = array index

move_s	S1, S2;	move string "S1" to "S2"
move_p2s	S1, L, S2;	move char at position "L" of "S1" to "S2"
move_s2p	S1, S2, L;	move string "S1" to position "L" of "S2"
move_s_a	S1, S2, LI;	move string "S1" to string array "S2"
move_a_s	S1, LI, S2;	move from string array "S1" to string "S2"
add_s	S1, S2, S3;	add string "S1" and "S2" to "S3"
strlen	S, L;	return string length to "L"
strleft	S1, L, S2;	move the left "L" chars of "S1" to "S2"
strright	S1, L, S2;	move the right "L" chars of "S1" to "S2"
ucase	S;	to uppercase
lcase	S;	to lowercase
char	L, S;	makes the string "S" from the ASCII-code of "L"
asc	S, L;	makes the ASCII-code "L" from the string "S"
eq_s	S1, S2, L;	equal
neq_s	S1, S2, L;	not equal

|----- set to "1" if true, "0" if false

## 8. Type conversion

L = long register, D = double register  
S = string register

val_l	S, L;	string to number
val_d	S, D;	
str_l	L, S;	number to string
str_d	D, S;	
2int	D, L;	double to int
2double	L, D;	int to double
char	L, S;	ASCII-code to string: 65 -> A
asc	S, L;	string to ASCII-code: A -> 65

## 9. Time functions

time; returns the current time to the following int variables:

_year	1900 - x	
_month	1 - 12	
_day	1 - 31	
_hour	0 - 23	
_min	0 - 59	
_sec	0 - 59	
_wday	1 - 6	weekday (1 = sunday ... 6 = saturday)
_yday	1 - 366	yearday

ton; timer on  
toff; timer off

The time between the "ton" and "toff" calls is stored in the lint variable "\_timer".  
The time is in "ticks". To get seconds, divide by "\_timertck" (ticks per second).

wait_s	L;	waits "L" seconds
wait_t	L;	waits "L" ticks (1/50 sec)

## 10. Internal variables

name	default	
_break	1	1 = break with Ctrl-C enabled, 0 = disabled
_timer		number of ticks between "ton" and "toff"
_timertck		ticks per second
membsize	4096	memory block size (bytes) Lx
vmbsize	1048576	vm swapfile size (bytes) Lx
vmcachesize	1024	array element cache Lx
vmuse	0	1 = virtual memory enabled, 0 = disabled Lx
stsize	4096	stack size (bytes) Lx
_intsize	2	size of int (short int) (bytes)
_lintsize	4	size of lint (long int) (bytes)
_doublesize	8	size of double (bytes)
_machine		host machine number: 1 = Amiga 2 = PC
_os		host OS number
_err_alloc	0	1 = memory error handling on, 0 = exit on error
_alloc		memory error code
_err_file	0	1 = file error handling on, 0 = exit on error
file		file error code Lx
sock		socket error code Lx
_year		1900 - x
_month		1 - 12
_day		1 - 31
_hour		0 - 23
_min		0 - 59
_sec		0 - 59
_wday		1 - 6 weekday (sunday - saturday)
_yday		1 - 366 yearday
_version		version number
_vmregs		number of registers
_language		language setting
_fnewline		newline string setting (fwrite_n, swrite_n)
_fendian		endianess setting (file read/write)
process		process error code Lx

# 11. Files

## 11.1 File handling

To work with a file, it must be opened first:

```
fopen      L0, name, mode;          (name, mode = string registers)
```

Register "L0" contains the file number, the name is "file", and the mode is read "r". Other modes are: "a" append and "w" write.

Close a file:

```
fclose     L0;
```

To read from a file:

```
fread_s    L0, string
```

As you may have guessed "string" is a string variable and "L0" is the file number.

Here is an example:

```
// txtsave.na

    push_s      "file to save text? ", S0;
    print_s     S0;
    input_s     S2;

    push_s      "w", S0;
    fopen       L0, S2, S0;

    push_i      1, L1;
    push_s      "Enter the text. Empty line to exit...", S0;
    print_s     S0;
    print_n     L1;

lab input;
    push_s      ": ", S0;
    print_s     S0;
    input_s     S1;
    fwrite_s    L0, S1;
    fwrite_n    L0, L1;
    push_s      "", S0;
    neq_s       S1, S0, L2;
    jmp_l       L2, input;

    fclose     L0;
    push_i      0, L0;
    exit       L0;
```

### Line feed

The "line feed" marks the end of a line in a text file. The two chars to mark this are:

```
CR (carriage return, ASCII code: 13)
LF (line feed,       ASCII code: 10)
```

The terms "carriage return" and "line feed" are from the good old typewriter age. Every operating system uses a different code:

```
DOS, Windows:      CRLF
Mac OS:           CR
Amiga OS, Unix, Linux: LF
?:               LF+CR (Yes! Even this weird thing seems to be around!)
```

If we read a line with "fread\_ls", nano takes care of all codes. It can handle all line feeds. Writing a line feed with "fwrite\_n" is different. We have to choose a code. This can be done with some code like this:

```
string cr[2];
string lf[2];

push_i      13, L0;
char        L0, cr;
push_i      10, L0;
char        L0, lf;

move_s      cr, _fnewline;
add_s       _fnewline, lf, _fnewline;
```

This sets "\_fnewline" to CRLF. The "fwrite\_n" opcode uses CRLF now. There is a default setting for "\_fnewline". It's the host code. On a Windows machine "\_fnewline" is set to CRLF, and so on.

## Binary files

There are two ways to store binary numbers in a file:

```
little endian
big endian
```

If we write the long int "76543" to a file, we get this hex code:

```
little endian:  FF 2A 01 00
big endian:    00 01 2A FF
```

The long int is four bytes long. Each number pair is one byte.

The difference is: "little endian" is the other way round as "big endian". We have to know the endianness of a binary file, to read and write numbers. Otherwise we would read and write false numbers. The endianness is set by the variable "\_fendian".

An example:

```
// file_endian.na
// endian file test
// shows how to use the endianness setting for binary files

#include <file.nah>
#include <math.nah>

#setreg_l    L0, null;
#setreg_l    L1, one;
#setreg_l    L2, two;
#setreg_l    L3, file;
#setreg_l    L4, write_b;
#setreg_l    L5, write_i;
#setreg_l    L6, write_l;
#setreg_l    L7, read_b;
#setreg_l    L8, read_i;
#setreg_l    L9, read_l;
#setreg_l    L10, endian;
#setreg_l    L11, check;
```

```

#setreg_l      L12, if;

#setreg_d      D0, write_d;
#setreg_d      D1, read_d;

#setreg_s      S0, files;
#setreg_s      S1, mode;
#setreg_s      S2, s;

push_s         "endian_big", files;
push_s         "wr", mode;

push_i         0, null;
push_i         1, one;
push_i         2, two;

push_b         255%, write_b;
push_i         3001I, write_i;
push_l         76543L, write_l;
push_d         m_pi, write_d;

move_l         null, file;

// set endianness with _fendian variable

push_i         endian_big, endian;
pull_i         endian, _fendian;

fopen          file, files, mode;

// write

fwrite_b       file, write_b;
fwrite_i       file, write_i;
fwrite_l       file, write_l;
fwrite_d       file, write_d;

// read

frewind        file;

fread_b        file, read_b;
fread_i        file, read_i;
fread_l        file, read_l;
fread_d        file, read_d;

// check data

move_l         one, check;
print_n        one;

neq_jsr_l      write_b, read_b, byte_error;
neq_jsr_l      write_i, read_i, int_error;
neq_jsr_l      write_l, read_l, long_error;
neq_d          write_d, read_d, if;
jsr_l          if, double_error;

eq_jsr_l       check, one, check_ok;
fclose         file;

```

```

    exit          null;

lab byte_error;
    move_l       null, check;
    push_s      "byte read/write error!", s;
    print_s     s;
    print_n     one;
    rts;

lab int_error;
    move_l       null, check;
    push_s      "int read/write error!", s;
    print_s     s;
    print_n     one;
    rts;

lab long_error;
    move_l       null, check;
    push_s      "long read/write error!", s;
    print_s     s;
    print_n     one;
    rts;

lab double_error;
    move_l       null, check;
    push_s      "double read/write error!", s;
    print_s     s;
    print_n     one;
    rts;

lab check_ok;
    push_s      "read/write check ok!", s;
    print_s     s;
    print_n     one;
    rts;

```

## 11.2 Opcodes

L = long register, D = double register  
 BV = byte variable, S = string register

### Open/close

```

fopen          L (file number), S (name), S (type);      opens a file

               types are: "r"   read
                           "w"   write
                           "a"   append
                           "rw"  read/write
                           "wr"  write/read
                           "ar"  append/read

fclose        L (file number);                          closes a file

```

## Read/write

<code>fread_b</code>	<code>L (file number), L;</code>	read byte
<code>fread_ab</code>	<code>L (file number), BV, L (length);</code>	read byte array
<code>fread_i</code>	<code>L (file number), L;</code>	read int
<code>fread_l</code>	<code>L (file number), L;</code>	read lint
<code>fread_d</code>	<code>L (file number), D;</code>	read double
<code>fread_s</code>	<code>L (file number), S, L (length);</code>	read string
<code>fread_ls</code>	<code>L (file number), S;</code>	read line
<code>fwrite_b</code>	<code>L (file number), L;</code>	write byte
<code>fwrite_ab</code>	<code>L (file number), BV, L (length);</code>	write byte array
<code>fwrite_i</code>	<code>L (file number), L;</code>	write int
<code>fwrite_l</code>	<code>L (file number), L;</code>	write lint
<code>fwrite_d</code>	<code>L (file number), D;</code>	write double
<code>fwrite_s</code>	<code>L (file number), S;</code>	write string
<code>fwrite_sl</code>	<code>L (file number), L;</code>	write lint as string
<code>fwrite_sd</code>	<code>L (file number), D;</code>	write double as string
<code>fwrite_n</code>	<code>L (file number), L;</code>	write "L" newlines
<code>fwrite_sp</code>	<code>L (file number), L;</code>	write "L" spaces

## Other

<code>fsetpos</code>	<code>L (file number), L;</code>	set stream position
<code>fgetpos</code>	<code>L (file number), L;</code>	get stream position
<code>frewind</code>	<code>L (file number);</code>	rewind stream
<code>fsize</code>	<code>L (file number), L;</code>	get file size in bytes
<code>fremove</code>	<code>L (file number), S (name);</code>	remove file
<code>frename</code>	<code>L (file number), S (old), S (new);</code>	rename file

## 11.3 Error codes

The default setting is to exit the program, if there is a "file error". This can be: a file can't be opened, or read... However in most cases you want a bit more control over this.

You can set the variable "`_err_file`" to "1" and switch on the error handling. Now every file operation returns a code to the register "`_file`".

The codes are defined in the "file.nah" include:

variable	code
-----	
<code>err_file_ok</code>	no error
<code>err_file_open</code>	can't open file
<code>err_file_close</code>	can't close file
<code>err_file_read</code>	can't read from file
<code>err_file_write</code>	can't write to file
<code>err_file_number</code>	file number not in legal range
<code>err_file_eof</code>	end of file reached while reading
<code>err_file_fpos</code>	wrong position in file

# 12. Memory

## 12.1 Organization

Variables are allocated in memory blocks. The default size is 4096 bytes per block. Up to 64 blocks can be used. Nano allocates the needed blocks automatically. If you need more memory, you can increase the blocksize with the “membsize” register:

```
#include <vm.nah>

push_i      8192, membsize;

// declare your variables:

int foo;
int bar;
```

You have to set “membsize” before you declare your variables.

Arrays are allocated as their own block. So “membsize” has no effect on them.

## 12.2 Virtual memory

If you need lots of memory and your machine has no MMU, then you can use the built in virtual memory driver.

**Note:** only arrays can be stored in VM!

You have to set the path, where the swapfile will be created:  
Set a “NANOTEMP” environment variable to the directory:

Amiga OS:

```
setenv NANOTEMP "T:vm_"
```

Nano adds the current time to the filename: “vm\_hhmmss”.

The “vmuse” register must be set to “1” to enable VM. The “vmbsize” register sets the swapfile size in bytes. The default is 1048576 bytes (1 MB). Just multiply with a factor to increase.

The “vmcachesize” register sets the cachesize for array elements. The default is 1024.

Example:

```
#include <vm.nah>

push_i      1, vmuse;                activate virtual memory

push_i      256, L0;
mul_l      vmbsize, L0, vmbsize;     set swapfile size to 256 MB

push_i      10000, vmcachesize;      cache for 10000 elements

stack_set;                            set global memory settings

// declare your arrays:

lint big[1000000];
```

## 12.3 Error codes

The default setting is to exit the program, if nano can't allocate memory. You can set the variable `"_err_alloc"` to `"1"` and switch on the error handling. Now nano returns an error code to the variable `"_alloc"` after each array allocation.

The error codes are defined in the `"memory.nah"` include:

variable	code
err_alloc_ok	allocation done
err_alloc_nomem	out of memory

# 13. TCP/IP Sockets

## 13.1 Introduction

Sockets are the standard interface for TCP/IP. They are used to send data through a network. Every computer on a network has an address like a phone number. To send data we have to know the right address and port number.

The port number is to identify the service. If you browse the web then you use 80, http. Downloading a file from a FTP server goes over port 21, and so on.

The port numbers are going from 0 - 65535. The numbers up to 1023 are reserved for standard services. So we should use numbers from 1024 upwards.

## 13.2 Client/Server

There are two kinds of sockets: client and server.

Now it's time for a little example. Let's say we have two computers in a network:

foo (192.168.1.1) and bar (192.168.1.2)

We want "foo" to be the server and "bar" is the client. Foo waits on port 2000 for an incoming message. The client asks for a message and sends it to the server. Here is the server:

```
// tcp server
//
// prints the client messages

#include <socket.nah>

#setreg_l      L0, null;
#setreg_l      L1, one;
#setreg_l      L2, two;
#setreg_l      L3, server;
#setreg_l      L4, _err_sock;
#setreg_l      L5, _err_sock_ok;
#setreg_l      L6, _err_addrinuse;
#setreg_l      L7, _err_nosys;
#setreg_l      L8, port;
#setreg_l      L9, server_open;
#setreg_l      L10, server_act_open;
#setreg_l      L11, len;
#setreg_l      L12, shutdown;
#setreg_l      L13, if;
#setreg_l      L14, args;
#setreg_l      L15, accept;

#setreg_s      S0, ip;
#setreg_s      S1, buf;
#setreg_s      S3, ip_arg;
#setreg_s      S4, sh;
#setreg_s      S5, lg;
#setreg_s      S6, s;

push_i        0, null;
push_i        1, one;
push_i        2, two;
```

```

push_i      err_sock_ok, _err_sock_ok;
push_i      err_addrinuse, _err_addrinuse;
push_i      err_nosys, _err_nosys;

move_l      null, server_open;
move_l      null, server_act_open;
push_s      "/shutdown", sh;
push_s      "/logout", lg;

// check arguments

argnum      args;
neq_jump_l  args, one, show_args;

argstr      null, ip_arg;

argstr      one, buf;
val_l      buf, port;

// open server

hostbyname  ip_arg, ip;
eq_jump_l  err_sock, _err_sock_ok, ip_ok;

hostbyaddr  ip_arg, ip;

lab ip_ok;
ssopen      server, ip, port;
jsr         check_err;
move_l      one, server_open;

wait_s      one;

lab wait_conn;
print_n     one;
push_s      "waiting...", s;
print_s     s;
print_n     two;

ssopenact   server, accept;
jsr         check_err;
move_l      one, server_act_open;

clientaddr  accept, buf;
push_s      "get message from: ", s;
print_s     s;
print_s     buf;
print_n     two;

lab read;
sread_l     accept, len;
jsr         check_err;

sread_s     accept, buf, len;
jsr         check_err;

print_s     buf;
print_n     one;

```

```

    eq_s      buf, lg, if;
    jmp_l     if, read_end;

    eq_s      buf, sh, shutdown;
    jmp_l     shutdown, read_end;

    jmp      read;

lab read_end;
    wait_s    one;
    print_n   one;
    push_s    "client logged out.", s;
    print_s   s;
    print_n   one;

    sscloseact accept;
    jsr       check_err;
    move_l    null, server_act_open;

    jmp_l     shutdown, shutdown;

    jmp      wait_conn;

lab shutdown;
    wait_s    one;
    push_s    "shutdown...", s;
    print_s   s;
    print_n   two;

    ssclose   server;
    jsr       check_err;
    move_l    null, server_open;

lab end;
    eq_jsr_l  server_act_open, one, close_act_server;
    eq_jsr_l  server_open, one, close_server;

    exit     null;

lab close_act_server;
    sscloseact accept;
    rts;

lab close_server;
    ssclose   server;
    rts;

lab check_err;
    neq_jmp_l err_sock, _err_sock_ok, error;
    rts;

lab error;
    push_s    "socket error: ", s;
    print_s   s;
    print_l   err_sock;

    eq_jsr_l  err_sock, _err_addrinuse, addrinuse;

    eq_jsr_l  err_sock, _err_nosys, no_tcp;

```

```

    print_n      one;
    jmp          end;

lab addrinuse;
    print_n      one;
    push_s       "address already in use!", s;
    print_s      s;
    print_n      one;
    rts;

lab no_tcp;
    print_n      one;
    push_s       "tcp stack not running!", s;
    print_s      s;
    print_n      one;
    rts;

lab show_args;
    push_s       "server <ip> <port>", s;
    print_s      s;
    print_n      one;
    jmp          end;

```

And here is the client program:

```

// tcp client
//
// sends messages

#include <socket.nah>

#setreg_l      L0, null;
#setreg_l      L1, one;
#setreg_l      L2, two;
#setreg_l      L3, client;
#setreg_l      L4, _errsock;
#setreg_l      L5, _errsock_ok;
#setreg_l      L6, _connrefused;
#setreg_l      L7, _errnosys;
#setreg_l      L8, port;
#setreg_l      L9, client_open;
#setreg_l      L10, len;
#setreg_l      L11, if;
#setreg_l      L12, args;

#setreg_s      S0, ip;
#setreg_s      S1, buf;
#setreg_s      S3, ip_arg;
#setreg_s      S4, sh;
#setreg_s      S5, lg;
#setreg_s      S6, s;

push_i         0, null;
push_i         1, one;
push_i         2, two;
push_i         err_sock_ok, _errsock_ok;
push_i         err_connrefused, _connrefused;

```

```

    push_i          err_nosys, _errnosys;

    move_l          null, client;
    move_l          null, client_open;
    push_s          "/shutdown", sh;
    push_s          "/logout", lg;

// check arguments

    argnum          args;
    neq_jmp_l       args, one, show_args;

    argstr          null, ip_arg;

    argstr          one, buf;
    val_l           buf, port;

// open client

    hostbyname      ip_arg, ip;
    eq_jmp_l        err_sock, _errsock_ok, ip_ok;

    hostbyaddr      ip_arg, ip;

lab ip_ok;
    scopen          client, ip, port;
    jsr             check_err;
    move_l          one, client_open;

// print commands

    print_n         two;
    push_s          "commands:", s;
    print_s         s;
    print_n         one;
    print_s         lg;
    push_s          " : logout", s;
    print_s         s;
    print_n         one;
    print_s         sh;
    push_s          " : shutdown server", s;
    print_s         s;
    print_n         two;

lab send;
    push_s          "message? ", s;
    print_s         s;
    input_s         buf;
    strlen          buf, len;

// check for empty string

    eq_jsr_l        len, null, empty_string;

    swrite_l        client, len;
    jsr             check_err;

    swrite_s        client, buf;

```

```

    jsr          check_err;

    eq_s        buf, lg, if;
    jmp_l      if, send_end;

    eq_s        buf, sh, if;
    jmp_l      if, send_end;

    jmp        send;

lab send_end;
    scclose    client;
    jsr        check_err;
    move_l     null, client_open;

lab end;
    eq_jsr_l   client_open, one, close_client;

    exit      null;

lab close_client;
    scclose    client;
    rts;

lab check_err;
    neq_jsr_l  err_sock, _errsock_ok, error;
    rts;

lab error;
    push_s     "socket error: ", s;
    print_s    s;
    print_l    err_sock;

    eq_jsr_l   err_sock, _connrefused, no_server;

    eq_jsr_l   err_sock, _errnosys, no_tcp;

    print_n    one;
    jmp        end;

lab empty_string;
    push_s     " ", buf;
    move_l     one, len;
    rts;

lab no_server;
    print_n    one;
    push_s     "server not found!", s;
    print_s    s;
    print_n    one;
    rts;

lab no_tcp;
    print_n    one;
    push_s     "tcp stack not running!", s;
    print_s    s;
    print_n    one;
    rts;

lab show_args;

```

```

push_s      "client <ip> <port>", s;
print_s     s;
print_n     one;
jmp         end;

```

### 13.3 Opcodes

L = long register, D = double register  
 BV = byte variable, S = string register

#### Open/close

ssopen	L (socket number), S (ip), L (port);	opens a server socket
ssopenact	L (socket number);	waits for clients
ssclosenact	L (socket number);	closes connection
ssclosen	L (socket number);	closes a server socket
scopen	L (socket number), S (ip), L (port);	opens a client socket
scclosen	L (socket number);	closes a client socket

#### Read/write

sread_b	L (socket number), L;	read byte
sread_ab	L (socket number), BV, L (length)	read byte array
sread_i	L (socket number), L;	read int
sread_l	L (socket number), L;	read lint
sread_d	L (socket number), D;	read double
sread_s	L (socket number), S, L (length);	read string
sread_ls	L (socket number), S;	read line
swrite_b	L (socket number), L;	write byte
swrite_ab	L (socket number), BV, L (length)	write byte array
swrite_i	L (socket number), L;	write int
swrite_l	L (socket number), L;	write lint
swrite_d	L (socket number), D;	write double
swrite_s	L (socket number), S;	write string
swrite_sl	L (socket number), L;	write lint as string
swrite_sd	L (socket number), D;	write double as string
swrite_n	L (socket number), L;	write "L" newlines
swrite_sp	L (socket number), L;	write "L" spaces

#### Other

hostname	S (name);	returns the local hostname
hostbyname	S (name), S (ip);	returns the ip
hostbyaddr	S (ip), S (name);	returns the name
clientaddr	L (socket number), S (ip);	returns the client ip on a server socket

## 13.4 Error codes

The socket opcodes return an error code to the register "err\_sock". Take a look at the examples "client.na" and "server.na" for more info.

The codes are defined in the "socket.nah" include.

## 14. Processes

### 14.1 Introduction

The process opcodes are for launching programs from nano. The new process is independent from the nano program. It runs asynchron. But your program can wait until the new process ends. So it will run synchron.

To launch a program we use "runpr":

```
push_s      "foobar", S0;
runpr       S0, process;
```

This launches the program "foobar". In the register "process" we get the process number. If we want to wait until the program "foobar" ends we can use "waitpr":

```
waitpr      process, retcode;
```

We must call it with the process number and get back the return code in the "retcode" register.

### 14.2 Opcodes

L = long register, S = string register

runpr	S (program name), L (process number);	launch program
runsh	S (program name), L (return code);	launch shell
waitpr	L (process number), L (return code);	wait until process ends

### 14.3 Error codes

The process opcodes return an error code to the variable "\_process". The error codes are defined in the "process.nah" include:

variable	code
err_process_ok	program launched
err_process_fail	can't launch process

# 15. Shell arguments

## 15.1 Introduction

To get the number of arguments we use "argnum". You can read the arguments with the "argstr" opcode.

The following example prints the shell arguments:

```
// args.na
// read shell arguments

#setreg_l      L0, null;
#setreg_l      L1, one;
#setreg_l      L2, no_args;
#setreg_l      L3, args;
#setreg_l      L4, i;

#setreg_s      S0, arg;
#setreg_s      S1, s;

push_i         0, null;
push_i         1, one;
push_i         -1, no_args;

// get number of arguments and jump to no_arguments if args = -1
argnum         args;
eq_jump_l      args, no_args, no_arguments;

push_s         "arguments: ", s;
print_s        s;
print_n        one;
move_l         null, i;

lab print_arguments;
argstr         i, arg;
print_s        arg;
print_n        one;

inc_l          i;
lseq_jump_l    i, args, print_arguments;

exit           null;

lab no_arguments;
push_s         "no arguments!", s;
print_s        s;
print_n        one;
exit           null;
```

## 15.2 Opcodes

L = long register, SV = string variable

argnum	L;	returns the number of shell arguments: -1 = no arguments, 0 = one argument...
argstr	L, SV;	moves the argument with index "L" to a string

## 16. Usage

### 16.1 The assembler nanao

To assemble program "foo.na":

```
$ nanao foo
```

#### Options:

```
-lines=      max source lines
-ops=        max opcodes
-vars=       max variables
-labs=       max labels
-objs=       max size of the output file (KB)
-s           strip debug info
```

### 16.2 The VM nano

To run program "foo.no":

```
$ nano foo
```

#### Options:

```
-q           quiet mode: no start messages
-stacks=     stacksize (KB)
```

#### Amiga OS note:

You have to increase the stack before running the programs.  
I use "stack 100000" and this works fine. Maybe you need to use a higher value.

### 16.3 Installation

Rename the binaries which are fitting to your machine to "nanao" and "nano".

#### Amiga OS

Example: nano is in "Work:nano":

Insert the following lines to your "user-startup" file:

```
path Work:nano add
setenv nanoinc "Work:nano/include/"
setenv nanoprogram "Work:nano/prog/"
setenv nanotemp "T:vm_"
```

#### Linux

Open a shell and "cd" to the nano directory.  
Create the nano directory in your homedir:

```
$ cp prog ~/nano/prog
```

Copy nanao and nano to "/usr/local/bin":

```
$ su
# cp nanoa /usr/local/bin
# cp nano /usr/local/bin
```

Copy the includes:

```
# cp -r include /usr/local/share/nano
```

Copy the manual:

```
# cp -r manual /usr/local/doc/nano
```

Set the env variables. I did this in the "~.bashrc" file:

```
#nano
export NANOPROG=/home/yourname/nano/prog/
export NANOINC=/usr/local/share/nano/include/
```

## Windows

Example: nano is in "C:\nano":

Insert the following lines to your "autoexec.bat" file:

```
set PATH=c:\nano\;%PATH%
set nanoinc=c:\nano\include\
set nanoprog=c:\nano\prog\
set nanotemp=%TEMP%\vm_
```

## 16.4 Compiling

You need the gcc c compiler.

### Amiga OS

You can find gcc on the Aminet archive: <http://aminet.net>

### Linux

Install gcc with your packet manager. Read the manual of your distribution how to do it.

### Windows

Install the MinGW tools from <http://www.mingw.org>

To compile nano, open a shell and do the following:

```
$ ./configure
$ make
$ su
# make install
```

### Porting nano

Here is a short list with the things that must be changed:

Makefile

```
CFLAGS          set the needed compiler options
LDFLAGS
```

include/

```
host.h
    Define a new machine and OS type.
    Set endianness.
    Set CLOCKS_PER_SEC if needed.
    PATH_SLASH_CONV set TRUE, if OS uses backslash in paths.
```

vm/

```
arch.h
    wait_sec    Use the delay functions of your OS.
    wait_tick

exe_socket.c   If your OS doesn't support BSD sockets, you have to
               change some stuff there.

exe_process.c  Process handling. This is platform dependent code.
```

# 17. Pointers

## Intro

With the “getaddress” and “pointer” opcodes, you can do indirect addressing of arrays.

```
// pointer.na pointer test

#setreg_l      L0, null;
#setreg_l      L1, one;
#setreg_l      L2, ind;
#setreg_l      L3, max;
#setreg_l      L4, address;
#setreg_l      L5, i;

int a[10];

push_i         0, null;
push_i         1, one;
push_i         0, ind;
push_i         9, max;

lab setarray;
  move_i_a     ind, a, ind;
  inc_l       ind;
  lseq_jump_l ind, max, setarray;

// get address of array a
getaddress    a, address;
int b[1];

move_l       null, ind;

// set address of a to variable b, at label readarray
pointer      address, b, readarray;

lab readarray;
  move_a_i    b, ind, i;
  print_l    i;
  print_n    one;
  inc_l     ind;
  lseq_jump_l ind, max, readarray;

  exit      null;
```

## Opcodes

L = long register, V = array variable

getaddress	var, L;	returns the address of the variable
pointer	L (address), V, label	sets the address to variable V at label
gettype	L (address), L (type)	returns the type of variable

## 18. Hash function

With “hash32\_ab” you can calculate a hash sum of a byte array:

```
hash32_ab  BV, length (L), hash (L);
```

## 19. Threads

With nano version 1.1.9 you can start multiple threads and run parts of your code parallel. This is implemented with pthreads (POSIX threads).

Here is an example “Hello world!” program with 4 threads:

```
// hello world threaded

#setreg_l    L0, null;
#setreg_l    L1, one;
#setreg_l    L2, two;
#setreg_l    L3, thread1;
#setreg_l    L4, thread2;
#setreg_l    L5, thread3;
#setreg_l    L6, thread4;
#setreg_l    L7, loop;
#setreg_l    L8, maxloop;
#setreg_l    L9, threadnum;

ston;

push_i      1, one;
thread_create thread1, hello;
print_s     "created thread: ";
print_l     thread1;
print_n     one;

thread_sync thread1;

thread_create thread2, hello;
print_s     "created thread: ";
print_l     thread2;
print_n     one;

thread_sync thread2;

thread_create thread3, hello;
print_s     "created thread: ";
print_l     thread3;
print_n     one;

thread_sync thread3;

thread_create thread4, hello;
print_s     "created thread: ";
print_l     thread4;
print_n     one;

thread_sync thread4;

// send string to thread1
```

```

    thpush_s      thread1, "Hello world! thread: ";
    thpush_sync   thread1;

// send string to thread2
    thpush_s      thread2, "Hello world! thread: ";
    thpush_sync   thread2;

// send string to thread3
    thpush_s      thread3, "Hello world! thread: ";
    thpush_sync   thread3;

// send string to thread4
    thpush_s      thread4, "Hello world! thread: ";
    thpush_sync   thread4;

// wait for all threads to be finished
    thread_join;
    push_i        0, null;
    exit          null;

lab hello;
    string s[256];

// get string from stack
    thpull_sync;
    stpull_s      s;

    push_i        1, loop;
    push_i        10, maxloop;
    push_i        0, null;
    push_i        1, one;

lab loop;
    print_s       s;
    thread_num    threadnum;
    print_l       threadnum;
    print_n       one;

    inc_lseq_jump_l  loop, maxloop, loop;

    exit          null;

```

With “thread\_create” you can start a new thread. It returns the thread handle in the long register and starts at the given label.

With “thread\_sync” we wait till the new thread is running. This is for time syncing the threads. This is done automatically by this opcode.

Then “thpush\_s” pushes the “Hello world!” string to the new thread stack. And “thpush\_sync” puts a synchronize object on top of the stack.

And at the called routine in the thread we take first the synchronize object (thpull\_sync) and then the string (normal “stpull\_s” opcode).

The “thpull\_sync” opcode waits for the synchronize object on top of the stack. If it gets the object, the thread is ready to pull the string from the stack.

And with “thread\_join” the main thread waits till the other threads are finished.

## Mutexes

You can protect variables with mutexes, this is done with the “var\_lock” and “var\_unlock” opcodes. If a variable is locked, other threads can't lock and access them. This protects them.

## Other

With “thread\_num” you can find out the current thread number.

And “thread\_state” returns the current state of a thread (running or not), see the threads.nah include file.

## Conclusion

With the thread opcodes it's possible to do a lot amazing things that were not possible before. It's just like opening a new door and see the possibilities that will come. It's a whole new world out there to explore and I just showed you the door to get in there...

## Opcodes

thread_create	threadnum (L), label;	start thread at the given label
thread_sync	threadnum (L);	waits for the thread start
thread_join;	wait till all other threads	are finished
thread_state	threadnum (L), state (L);	returns the state of a thread
thread_num	threadnum (L);	returns the current thread number
thpush_l	threadnum (L), L;	push the L register to thread
thpush_d	threadnum (L), D;	
thpush_s	threadnum (L), S_VAR;	
thpush_sync	threadnum (L);	push the sync object to thread
thpull_sync;		wait for the sync object

## 20. Appendix

There are some modifiers for number types:

**I** for an integer number.

**L** for a long integer number.

**D** for a double number.

**%** for a byte number.

And for number formats:

**B** for a binary number with ones and zeros only.

**&** for a hex number.

Lets see it in an example:

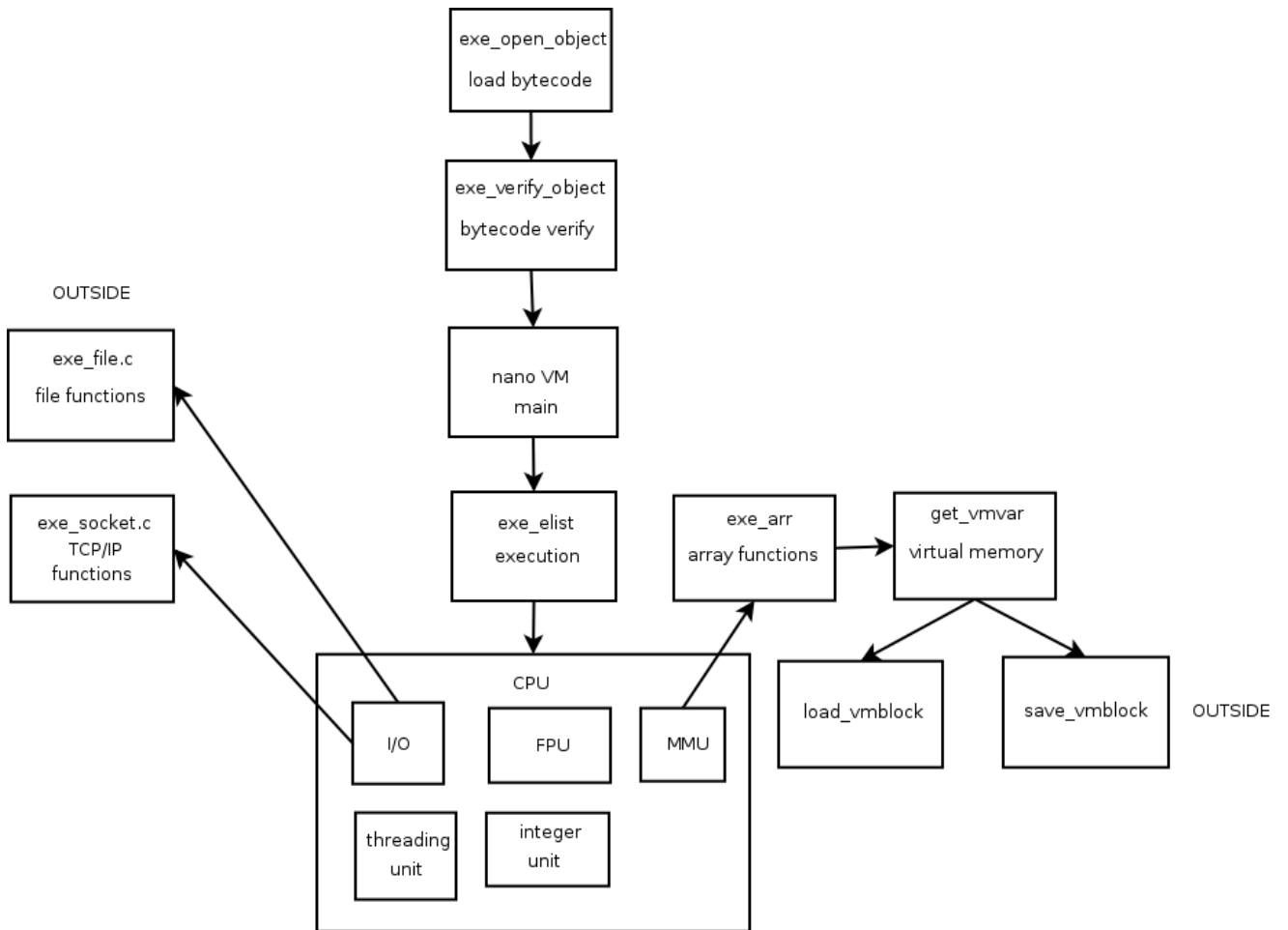
```
push_b      B10110111%, L0;
```

The B at the beginning marks the number as binary. The % sign at the end tells the assembler that is a byte number.

```
push_i      &FF, L0;
```

The & sign at the beginning marks the number as hex. The chars range from A to F. That numbers go from 10 to 15.

**New stack:** "stack\_set" has to be called before "ston" to set the global stack / memory settings registers. **This is important!**



**Nano VM internal design.**