programming language

FANF

written by
Konstantin Dimitrov

Revision 1408
August 2014

For comments and suggestions: knivd@me.com

Table of Contents

1. Introduction

FANF is a new general purpose language, completely designed from scratch by combining various features from a number of other modern programming languages.

The general syntax is inspired from the popular Forth, but while looking similar on the surface, FANF is actually a very different language (hence the name "**F**orth **A**like but **N**ot **F**orth"). FANF also natively supports multitasking and easy operations with text and binary data with all memory allocations done transparently to the user.

The basic set of FANF words is the minimum needed to serve as base on which other words can be defined thus expanding the scope of performed functions.

2. Data stack

FANF uses data stack for all operation. Consider the stack as a pile of paper notes. When you have something to record, you write the note and put it on the top of the pile. When taking from the pile, the last put there being on the top, is the first to be taken, then the one put before it, etc.

Here is a simple example how the stack works:

Initially the stack is empty.

Adding 5:

Index	Data
0	5

Adding the phrase "Hello world!":

Index	Data
0	Hello world!
1	5

Adding -2.7:

Index	Data
0	-2.7
1	Hello world!
2	5

Adding another phrase "My stack"

Index	Data
0	My stack
1	-2.7
2	Hello world!
3	5

At this moment we want to take something from the stack. Since the phrase "My stack" was last put there, it will be the first to go:

Reading ---> "My stack"
Stack after the operation:

Index	Data
0	-2.7
1	Hello world!
2	5

Reading ---> -2.7
Stack after the operation:

Index	Data
0	Hello world!
1	5

Reading ---> "Hello world!"
Stack after the operation:

Index	Data
0	5

Adding 99:

Index	Data
0	99
1	5

... and so on.

The stack is automatically maintained by the FANF Virtual Machine (FVM) and is completely transparent to the user.

3. Data types

There are only two data types in FANF: numbers and text. The term "text" represents any sequence of bytes (including bytes with value 0), so text can also store any binary data as well. There is no limit in the length of a text element.
Numbers can be any integer or real and the FANF compiler and FVM automatically determine the needed type.
FANF supports a few possible ways of representing a number:

12, *0.1*, *-6.77* are integer or real numbers (automatically determined by the compiler).

1.003e-02 is a real number presented in scientific format.

#22998 is a number in decimal format (preceded by a **#** character). The # character is optional and can be omitted.

$fa893 is a integer in hexadecimal format (preceded by a **$** character)

%100011 is a integer in binary format (preceded by a **%** character)

Text constants are enclosed in double quotes: "*This is text*" or "*Enter your age:* "

If a double quote character is needed in the text, it is preceded by a _ character: "*_"This is enclosed in double quotes_"*".

Similar to that, a _ character in the text constant must be preceded by another _ character.
Any byte value can be inserted into a text constant using one of the possible representations:

c will insert a byte with value the ASCII code of the printable character 'c' minus $30. Thus for example *:* will insert $0a because the : character has ASCII code $3a. *_0* will insert the actual value 0 because the character 0 has ASCII code $30.

_#ddd will insert a byte with decimal value 'ddd'. All the three digits must be present, which creates combinations from _#000 through _#255.

_$hh will insert a byte with hexadecimal value 'hh'. The two hexadecimal digits must be present, which creates combinations from _$00 through _$ff.

_%bbbbbbbb will insert a byte with binary value 'bbbbbbbb'. All the eight binary digits must be present, which creates combinations from _%00000000 through _%11111111.

Example: "*_$fa_$99$_00_$84*" is a four-byte binary constant with the four bytes $fa, $99, $00 and $84.

4. Syntax

FANF uses reverse notation (RPN) just like its ancestor - the Forth language. RPN although looking a bit strange in the beginning, is actually quite simple and allows the developer to change the logical order of operations according to the exact needs. Writing in reverse notation is closely connected to the stack – data is inserted in the stack and then the operation to it is performed.

Let's consider this (assuming the stack is empty):

12 27 +

The execution of it is as follows:
Step 1: the number 12 is inserted into the stack
Step 2: the number 27 is inserted into the stack and becomes the top element
Step 3: the operation **+** takes two numbers from the stack (27 and 12 in their respective order of taking), add them together and returns the result back to the stack.

After these steps in the stack will be only one number: the result of the adding operation – the number 39.

Lets consider a bit more complex example:

33 15 – 22 *

This is the equivalent of the equation (33-15)*22. The **–** operation will return its result 18, then the ***** operation will take the two currently remaining in the stack numbers 22 and 18 in their respective order, multiply them and return the overall result 396.

It is obvious that the order of inserting the parameters and their operations is a vital factor for the proper result calculation. Taking the previous example:

33 15 22 - *

although inserting the same numbers and operations, but in a slightly different order, would create a completely different result: 33*(15-22) = -231

A FANF program consists of a number of defined "*words*", which specify what should be done when the particular word is used. These words create a "*runtime library*" for FVM.

Definition of a word follows a strict and consistent pattern:

word:
    ..... other words and parameters .....
;

The definition starts with the new word itself followed by a : character without any spaces. Then follows the code of the word which consists of other words, numbers, text and operations. The word ends with a ; character.

Note that after the : character and before and after the ; character there must be spaces just as between any other words and parameters.

The ***space character*** is a universal delimiter in FANF.

Here is example of a simple new word definition:

SQR: 0.5 power ;

This creates a new word "*SQR*" which calculates square root from the number in the stack by applying the operation $SQR(X)=X^{0.5}$

Note that everything in FANF is case sensitive, thus "*SQR*", "*Sqr*" and "*sqr*" and three different words.

Now with the "*SQR*" word already defined, somewhere in the program later:

81 SQR

will leave the number 9 in the stack

5. Comments

Comments in a FANF program can be inserted in two ways:

```
` this is something that
can span in as many lines as needed `
```

Any text enclosed in ` characters will be ignored during compilation.

Alternatively, the form

```
`! this is just another comment
```

will insert the comment only until the end of the current line

6. Variables

While in many cases the data stack is all that is needed for a program for its operative data, sometimes it is much more convenient to store data somewhere and later refer to it by name. FANF does not separate data from code, thus a define word can be both executed and referred to as a data container. Normally every newly defined word has no data container. Such can be specified by including ***x data*** in the code, where 'x' is the number of data elements needed. As an example:

myword: 1 data *;*

This will define a new word "*myword*" which has a data container with one element.

Now in order to store something into "*myword*", we need to refer to it not for execution, but for storage:

27 @myword =

This translates as "store 27 into myword", or myword=27. Note the **@** character, which refers to the word's data container. If the @ reference is not followed by the = operation, the data container is returned in to stack:

@myword

will read the "*myword*" data container and store it in the stack.

Data containers may have more than one element:

myword: 25 data ;

will define myword as a 25-element data container. In order to access containers with more than one element, an index is required when referring to the word:

27 3 ! @myword =

The same can be expressed in the alternative form:

3 ! 27 @myword =

This translates as "store 27 into the 3rd element of myword". Note that element indexes always start from 0.

Note the word **!** Which has the purpose to tell the compiler which index of the variable will be used.

Respectively to read a multi-element data container:

3 ! @myword

will put the 3rd element of "*myword*" into the stack.

The number of elements of a word can be found by using the word **size**. In the example above:

@myword size

will return 25.
Note that within a word the size of the data container can be changed as many times as needed.
FANF also does not make any difference in the type of the data stored into the containers:

27 3 ! @myword =
"Hello!" 4 ! @myword =

will store the number 27 into the 3rd element, and the text "Hello" into the 4th element of "*myword*".

This concept makes very easy the definition and work with constants as well:

pi: 3.1415926 ;

Executing **pi** in the code will simply put the number into the stack. Another way of doing that is:

```
constants:
     2 data
     3.1415926 0 ! @:constant = ;
     2.7182818 1 ! @:constant = ;
; constants
```

This will define both *pi* and *e* as constants with indexes 0 and 1 respectively. Note the : characters, which define the scope. That will be explained in the following chapter.

Finally, the same example with constants can be made as an object with encapsulated provision of the needed constant:

```
constant:

     2 data
     3.1415926 0 ! @:constant = ;
     2.7182818 1 ! @:constant = ;

     get:
          dup
          "pi" == if
               drop 0 ! @:constant
          endif
          "e" == if
               1 ! @:constant
          else
               0     `! unknown parameter
          endif
     ;
;
```

This code defines a sub-word "*get*", which based on the input parameter will return the appropriate value:

```
"pi" constant:get
```

later used in our code will return the numeric value of *pi*. Using the new sub-word however is completely optional:

```
1 @constant
```

will return the same value of *pi*. What will be used is up to the developer's liking.

Dynamic changes in the size of a data container are also allowed:

```
myword:
     1 data     `! myword has data size 1
     ..... doing something .....
     5 data     `! now myword has data size 5
     ..... doing something else .....
     0 data     `! now myword has no data container
;
```

When a data container is increasing in size, its currently stored values are retained and only new elements are added. When decreasing the elements with indexes greater than the new size are destroyed.

7. *Scope of a word*

Each defined word in FANF has a certain scope for its code and data. A word may have its own set of "sub-words" and data containers:

```
myword:

     subword1: ..... some code or data ..... ;
     subword2: ..... some code or data ..... ;

     subword1  `! executing subword1

;
```

This code defines a new word "*myword*", which has two internal sub-words: "*subword1*" and "*subword2*". Those are local for myword. They can be seen from the outside world, but only by referring to their parent word first:

```
myword:subword1
```

will execute "*subword1*" from outside of myword. Note that the same name subword1 could be a completely different word in another word, eg. "*mynewword*".

The : character specifies the scope of a name. Each : takes one level up.

This example demonstrates types of referencing words in different scopes:

```
otherword: 1 data ;

myword:
     somedata: 1 data 55 * ; somedata

     mydata:
          mysubdata: 2 data ; mysubdata

          1 data

          3 1 ! @mysubdata =  `! referring to mysubdata which is a subword of mydata
          5 @:mydata =        `! referring to mydata's own container
          16 @:somedata =     `! referring to another word within the parent myword
          "November" @::otherword = ` referring to otherword which is one level
                                       up and outside of the parent word `
     ; mydata
;

"Index 0" 0 ! @myword:mydata:mysubdata = `! referring to subword's container
32 myword:somedata  `! executing a subword (in this case multiply a parameter by 55)
```

As it can be seen any word can serve as a data container and perform some function at the same time:

```
apple:
     cut: 1 data / @:cut = ;
;
```

In the example above the word "*cut*" is a sub-word of "*apple*" and stores the number of pieces the apple has been cut into, but also performs the operation of cutting itself. Hence in the code outside

```
8 apple:cut
```

will execute the operation of cutting and store the result. When at some point in the program we need to know how many pieces of apple do we have:

```
@apple:cut cout
```

will take the number from the data container and send it to the console.

Let's also consider the following piece of code:

```
myword: 1 data ; myword

myword1:

     myword: 1 data ; myword

     1 @myword =
     2 @:myword =
;

myword1
@myword1:myword cout
@myword cout
```

What would it print on the console: "12", "21" or "22"?

The first = statement refers to "*myword*". But "*myword*" is a sub-word in "*myword1*" hence takes priority over the global definition of "*myword*". Therefore the first statement loads the local "*myword*" data container with the value 1.

The second = statement refers to "*myword*" in the parent's scope, therefore it loads the global "*myword*" data container with 2.

Therefore the result after printing will be "12".

8. Multitasking

Normally when a word is called by its name, the execution goes to its code and after the execution of the word is completed, returns back to the caller and continues further. FANF allows two ways of executing words:

myword
~myword

The first line is the word "*myword*" executed in the normal way. The second line (note the leading **~** character) executes the word in parallel with the current thread. The execution continues immediately after the statement and a new parallel thread running "*myword*" is created.

When a new thread is created, a new local stack for it is created as well and its initial content cloned from the data stack of the parent. All data containers are global and available to all threads though so they can exchange data and messages through them.

A thread ends in the same way as a program – by using the **end** word or reaching end of the code in the executed word.

9. Error handling

Most of the errors are detected still during the compilation phase, some however can only occur during the process of execution (divide something to a parameter 0 from the stack for instance). When an error is detected the execution stops and the control is passed to a word with pre-defined name **ERROR** in the top scope. If such word does not exist, then an error message is sent to the console the FVM stops the execution.

The error handler has a fixed format:

ERROR: ..... error handling code ..... ;

Upon entering the word "*error*" the stack is empty with only the error code passed as parameter. There is nowhere to return after reaching the end of the error handler and the error message will be sent to the console and FVM stopped. If that needs to be avoided, the error handler must "restart" the whole program by executing its entry word.

10. Shell words

In general FANF does not make any difference between words typed directly in the command line, and those, actually used in the programs (called “atomic words”). There are a few exceptions though. Those words can only be executed from the command line and will trigger an error when used in code.

These words are part of the programming shell and helping the user to enter or edit FANF code.

Shell words are displayed in a separate group at the top, when the word “**words**” is executed.

11. FANF words

11.1 General execution control, branches and loops

x if
continue if x is not 0, otherwise jump to the relevant 'else' address
*x **if** ...(executed if x is not 0)... else ...(executed if x is 0)... endif*

else
mark the 'else' option for an 'if' section
*x if ...(executed if x is not 0)... **else** ...(executed if x is 0)... **endif***

endif
mark the end of an 'if' section
x if ...(executed if x is not 0)... else ...(executed if x is 0)... endif

do
mark the start of an 'repeat' section
***do** .... x repeat*

@x while
execute the section 'while ... repeat' if the variable @x is not 0
*@myvar **while** .... x repeat*

loop
execute the section 'loop ... repeat' exactly x times
*x **loop** .... repeat*

repeat
different formats according to the loop opening statement

*do ..... x **repeat***	*if x is not 0, return back to the relevant 'do'*
*@x while .... **repeat***	*unconditionally return to 'while'*
*x loop ..... **repeat***	*repeats the loop x times*

cont
continue the execution of the current 'do', 'while' or 'if' structure skipping all the remaining words before the closing statement

break
break the execution of the current 'do', 'while' or 'if' structure

end
prematurely exit the current word

endall
ends all active threads at once

run
run the last top level word

threads
return the number of currently active threads

maxthds
return the maximum number of supported simultaneous threads

11.2 Variables and stack
v @w = set an element of @w with value v
x ! load the variable index pointer with value v x **!** @w =   will set the x-th element of w with v x **!** @w   will return the x-th element of w in stack
@w size return the size of a data container
n data reserve n data cells associated with the current word
clear clear the entire data memory - stack for all processes and data containers
empty empty the stack for the current process
depth return the number of elements currently in the stack
drop remove the top stack element
dup duplicate the top stack element
n copy copy the n-th element in the stack to the top depth indexes start from 0, so '0 copy' has the same effect as 'dup' negative depth indexes have no effect
n swap swap the n-th element in the stack with the top (does not change the number of elements in the stack) depth indexes start from 0, so '0 swap' has no effect negative depth indexes have no effect
x type return the type of element x (top) = (0:number; 1:text)
x astext return x converted into text form if x is a number, it is first converted into its text representation no operation is done if x is already text
x asnum return x converted into number if x is text, it is first converted into a number (result 0 on error) no operation is done if x is already a number
x isnum return 1 if x can be converted into a valid number, 0 otherwise

11.3 Maths and logic
x not bitwise 'NOT' operation on an integer number
x y and bitwise 'AND' operation on integer numbers
x y or bitwise 'OR' operation on integer numbers
x y xor bitwise 'EXCLUSIVE OR' (XOR) operation on integer numbers
x y shl return x shifted bitwise y times left
x y shr return x shifted bitwise y times right
x ? return 1 if x contains any data of any type, 0 if x contains no data
x y == return 1 if x is the same as y, and 0 otherwise works with any data type
x y <> return 1 if x is not the same as y, and 0 otherwise works with any data type
x y > return 1 if x greater than y, and 0 otherwise works with numbers; for text compares the length; comparison of different data types always return 0
x y >= return 1 if x greater than or equal to y, and 0 otherwise works with numbers; for text compares the length; comparison of different data types always return 0
x y < return 1 if x smaller than y, and 0 otherwise works with numbers; for text compares the length; comparison of different data types always return 0
x y <= return 1 if x smaller than or equal to y, and 0 otherwise works with numbers; for text compares the length; comparison of different data types always return 0
x sign return 1 if x is 0 or a positive number return -1 if x is a negative number works with numbers only; for text types always returns 1
x y + return x+y

x y - return x-y
x y * return x multiplied y times
x y / return x divided by y
x y // 'modulo' operation; return the reminder of an integer division x/y by its nature the modulo operation is applicable on integer numbers only
x _+1 a numeric value on the top of the stack is increased by 1 text type data will generate an error
x _-1 a numeric value on the top of the stack is decreased by 1 text type data will generate an error
x abs return the absolute value of x
x round round the value of x to the nearest integer
x trim trim the value of x to greatest integer not greater than the value
x y power calculate and return x^y
x sin calculate the trigonometric function sin(x) the value of x is supplied in radians
x cos calculate the trigonometric function cos(x) the value of x is supplied in radians
x tan calculate the trigonometric function tan(x) the value of x is supplied in radians
x atan calculate the trigonometric function arctan(x) the value of x is supplied in radians
x rad convert x from degrees into radians
x deg convert x from radians into degrees
x exp calculate and return e^x
x ln calculate natural logarithm of x
x log calculate decimal logarithm of x

rnd
return a random number between 0 and 1, but not equal to 1

x rndseed
initialise the pseudo-random generator with seed value x

PI
return the pre-defined constant *pi*

E
return the pre-defined constant *e*

11.4 Work with text type data

x len
return the length of x in number of bytes
works with text type; for numbers the length is the number of characters as if the number if output to the console in text form

x y ++
concatenate x and y to produce the result xy
works with text type only

s b c cut
cut c characters starting from the b-th one (b starts from 0) from s
works with text data only

s b c delete
delete c characters starting from the b-th one (b starts from 0) from s
works with text data only

d s x insert
insert text s into d starting from position x
works with text data only

d s x replace
replace a fragment in d with the text s starting from position x
the length of d is increased if necessary
works with text data only

d s x scan
scan d starting from the x-position onwards and return the first occurrence of s or -1 if not found

[... **x***]* **f format**
format input parameters according to the format specifier(s) f and return the result in text form

format specifiers follow the pattern:
|*type*[*modifiers*][[=]*length*[.*fraction*[=]]]

type:
(decimal number), **$** (hexadecimal number), **%** (binary number)
***** (text); the * type MUST be followed by a fill character for the blank positions in the result
| (the | character itself)

modifiers:
< (left aligned), **>** (right aligned), **^** (centred)
+ (forced + sign for decimal numbers), **-** (reserved space for sign)
= (enable leading or trailing zeros in numbers)

fractional length is applicable to decimal numbers only
a format string f can contain text literals and any number of parameters to be taken from stack

11.5 Work with files and console

n open
open file with name n
this function returns TWO values in the stack:
(top) = current file length in bytes or -1 on failure
(top+1) = file handler or -1 on failure

h close
close file with handler h
if h has value 0, all currently open files will be closed at once

h eof
return 1 if the end of file has been reached, and 0 otherwise

h x seek
place the internal file position pointer at offset x from the beginning of the file
return the actual position of the pointer or -1 on failure or EOF

h pos
return the current file pointer position in the file; return -1 on failure or EOF

h x fout
(top element) ---> (file h)
text data is output as it is while numeric type is first converted into text and then output
will return the number of actually sent characters

h x fin
(x readings from file h) ---> (top element)
read exactly x characters from h into a text type element

x cout
(top element) ---> (console)
text data is output as it is while numeric type is first converted into text and then output

cin
(console as text input) ---> (top element)
read the console until a LF character; editing with backspace is allowed
will return a text type element

x nwcin
(x readings from console without waiting) ---> (top element)

read exactly x characters from the console into a text type element will not wait if the currently available characters in the console buffer are less than x, and will return what has been read so far

11.6 OS and hardware
x addr M8w write 8-bit unsigned x to physical memory address addr works with integer numbers
addr M8r read 8-bit unsigned from physical memory address addr works with integer numbers
x addr M16w write 16-bit unsigned x to physical memory address addr works with integer numbers
addr M16r read 16-bit unsigned from physical memory address addr works with integer numbers
x addr M32w write 32-bit unsigned x to physical memory address addr works with integer numbers
addr M32r read 32-bit unsigned from physical memory address addr works with integer numbers
free return the amount of currently free memory (program plus data) in bytes
x system pass the text x to the operating system for execution (not available in all systems)
x machine execute the text x as piece of machine code (not available in all systems)

11.7 FANF library and shell words
help basic help information (shell word)
s source compile (and execute if necessary) provided in s FANF source code
w isknown return 1 if the word w is an atomic or shell word, 2 if the word already known in the library, and 0 otherwise

w use use a defined word w passed by its text name
w forget remove the definitions of all defined words from w onwards
words list all words from the dictionary the words are listed as three separate groups – shell words, atomic words, and library words (shell word)
restore restore the library from non-volatile memory (shell word, not available in all systems)
store store the library into non-volatile memory (shell word, not available in all systems)
*[***w***]* **list** list a given word supplied by its name if the stack is empty, list the entire library
x s resrc replace source line starting from index x, given by the 'list' word with the new source s (shell word)
x s insrc insert a new source line s starting from index x, given by the 'list' word (shell word)
peek display all stack data NOTE: does not remove anything from the stack (shell word)
*[***w***]* **insp** inspect the data container(s) of a given word if no word name supplied – inspect all current data containers (shell word)

12. Platform-dependent words

(Microkite / PIC32MX170 port)

The Microkite module is designed and manufactured by Dimitech: **http://dimitech.com**

These words are only available in the relevant port for the hardware platform. They are not part of the generic FANF syntax and may not exist, or exist in a completely different form in another FANF port for a different hardware.

12.1 Miscellaneous control words

v p option
set shell/device option p with value v
The type of v (number or text) depends on the exact option

valid identifiers for p (case sensitive):
"**PageLines**" define the number of printed lines in terminal before pause (0: disable)
valid values are between 0 and 1000000000
"**ConsoleBaudrate**" define baudrate for the console (default is 38400); the protocol is fixed at 8N1
valid choices are: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200

u10tick
return the current value of the free running 10-microsecond system tick 32-bit incremental counter

f cpuclk
set CPU frequency

frequency:
1 1 MHz
4 4 MHz
8 8 MHz
12 12 MHz
24 24 MHz
32 32 MHz
40 40 MHz *(default at start)*
48 48 MHz *(with 50MHz revisions of PIC32MX170)*

NOTE: selecting new CPU frequency will only re-initialise the console
other clock-dependent peripherals such as UART, etc., will have to be re-initialised as well

12.2 Input/Output ports

n func[**modifiers**] **portcfg**
configure port n for function func
func is supplied as text, optional modifiers are part of the function text
ports are numbered from 0 to 32 (Microkite; 44-pin PIC32MX170)
PIC32MX170's raw port numbering can be found in Chapter 12A.

functions:
“**DIN**” digital input
“**DOUT**” digital output
“**AIN**” analogue input
“**PMO**” pattern modulated output
“**PWM**” pulse width modulated output

modifiers:
1 (pull-up), **0** (pull-down), ***** (open-drain output)
- (action triggered on 1-0 transition;
(in PMO once-off mode will set exit state high, instead of low)
= (action triggered on either 0-1 or 1-0 transition)
! (applicable to PMO only; specify once-off operation)

n whatcfg
return the possible configuration for a port as a 32-bit bitmask
each bit when raised tells that the port can be assigned to/with the relevant functionality; unused bits in the mask have values 0

bit	description
0	can be digital input
1	can be digital output
2	can be analogue input
3	can be pattern modulated output
4	can be pulse width modulated output
20	can enable an internal pull-up resistor
21	can enable an internal pull-down resistor
22	can enable an open-drain output
23	action triggered on 1-0 transition (by default it is 0-1) this bit also sets the exit state for once-off PMO events
24	action triggered on any transition
25	'once-off' PMO flag
31	assigned to a specific hardware (comms port, etc) this bit also tells that the port can't be used as normal I/O

n portrd
read port n and return its current digital or analogue value according to its configuration

behaviour depends on the port configuration:

DIN	the actual port logic value 0 or 1
DOUT	the last written to the port logic value 0 or 1
AIN	measured voltage on the port
PMO	PMO pattern value (1 behaviour bit + 31 pattern bits)
PWM	PWM value (0..32)

n v portwr
write integer value v into port n

behaviour depends on the port configuration:

DIN	set initial value in the port counter
DOUT	set the digital output with logic value 0 or 1
AIN	no effect
PMO	set new PMO pattern value in once-off operation the exit state will be 0 (or 1 in "**PMO-**" configuration); there are 32 valid pattern bits
PWM	set new PWM value (0..32)

n portfq
return the currently measured frequency [Hz] on a DIN port
every DIN port has the ability to measure low frequency in software with the level of accuracy significantly dropping with increasing the input frequency
the Microkite/PIC32MX170 port is able to measure low frequency input up to about 4kHz with sub-Hertz frequencies possible to measure

n portcn
return the current counter value on a DIN port
every DIN port has the ability in software to count transitions
counter works as per the triggering action specified in **portcfg** (default: 0-1 transition)

n porttlc
return the elapsed time since the last detected change on a DIN port
the returned time is in microseconds, which limits the maximum detected interval at about 4295 seconds

n v portdiv
set output divider v for PMO or PWM port n
the PMO and PWM outputs are normally clocked at 9.6kHz for 40MHz CPU clock and proportionally divided for different CPU clocks
value 0 has same effect as value 1; negative values are ignored

12.3 Communications

c x pout
(top element) ---> (communication port c)
text data is output as it is while numeric type is first converted into text and then output
will return the number of actually sent characters

supported communication ports:
1 hardware UART RX:(P14/B11), TX:(P15/B10)
5 hardware SPI MISO:(P0/B5), MOSI:(P1/A1), SCLK:(P2/B14)
8 hardware I^{2}C SDA:(P11/B9), SCL:(P12/B8)

c x pin
(x readings from communication port c) ---> (top element)
read exactly x characters from port c into a text type element

will not wait if the currently available for reading characters from d are less than x, and will return what has been read so far

supported communication ports:
1 hardware UART RX:(P14/B11), TX:(P15/B10)
5 hardware SPI MISO:(P0/B5), MOSI:(P1/A1), SCLK:(P2/B14)
8 hardware I^2C SDA:(P11/B9), SCL:(P12/B8)

c p b pconfig
configure communication port c with protocol p and baudrate b
the same word is used to close the communication port as well
executing with baudrate 0 release the assigned hardware port

communication port 1
hardware UART RX:(P14/B11), TX:(P15/B10)

protocol:
"8N1" 8 data bits, no parity, one stop bit

baudrate can be any number between 200 and 250000
NOTE: some baudrates may not be possible to be achieved accurately when the processor clock is very low

communication port 5
hardware SPI MISO:(P0/B5), MOSI:(P1/A1), SCLK:(P2/B14)

protocol:
0 SPI mode 0
1 SPI mode 1
2 SPI mode 2
3 SPI mode 3 *(recommended; automatically used with **mount**)*

baudrate can be any number between 50000 and 20000000
NOTE: that the actual maximum baudrate will depend on the current processor clock and cannot be higher than 1/2 of it, and higher than 20000000, whichever is a smaller number

for mSPI applications and to conform with the standard DTX pinout, **P3** (PIC32 port RC6) must be used for the slave select line

if file system on SD is used with the word **mount**, the SPI port is automatically initialised and port **P32** (PIC32 port RC3) is assigned to the slave select line for the SD card

communication port 8
hardware I^2C SDA:(P11/B9), SCL:(P12/B8)

protocol:
0 7-bit addressing
1 10-bit addressing

baudrate can be any number between 5000 and 500000
NOTE: that the actual maximum baudrate will depend on the current processor clock and cannot be higher than 1/8 of it, and higher than 500000, whichever is a smaller number

12.4 Storage

strgerr
every executed word that performs an operation with the storage, returns the result of its execution into an internal FANF variable, which can be read into the stack with this word
value 0 means no error

mount
mount a FAT16/FAT32 file system on SD card using the SPI channel
return 0 if successfully mounted, or a negative number error code if there was an error
the hardware is automatically initialised if necessary
MISO:(P0/B5), MOSI:(P1/A1), SCLK:(P2/B14), ~nSS:(P32/C3)

unmount
unmount already mounted SD card and release P32/RC3 port
the SPI hardware will remain active

makefs
initialise a new file system on the SD card
NOTE: when executed in code the **makefs** word will erase ALL information currently on the SD card without any further questions

*[***n***]* **dir**
show files and sub-directories starting from path n
if the stack is empty, will assume n as the current path

n mkdir
make directory with name n

n chdir
change the current path to directory with name n

n f fcopy
copy file with name n to f

n fdel
delete file with name n

n f fren
change a file name from n to f

12A. *Microkite / PIC32MX170 port numbering relation*

Microkite port	PIC32MX170 port (44pin)	Note
N/A	**RA4**	**FANF console Rx line**
N/A	**RB4**	**FANF console Tx line**
P0	RB5	DTX standard pinout MISO line
P1	RA1	DTX standard pinout MOSI line
P2	RB14	DTX standard pinout mSCK line
P3	RC6	DTX standard pinout ~mSS line
P4	RC7	
P5	RC8	
P6	RC9	
P7	RB12	
P8	RA9	
P9	RC4	
P10	RC5	external pull-up in Microkite
P11	RB9	external pull-up in Microkite
P12	RB8	external pull-up in Microkite
P13	RB7	external pull-up in Microkite
P14	RB11	
P15	RB10	
P16	RB3	
P17	RB6	
P18	RA3	
P19	RA2	
P20	RB0	
P21	RB1	
P22	RB2	
P23	RB15	
P24	RB13	
P25	RA0	
P26	RC0	
P27	RC1	
P28	RC2	
P29	RA8	
P30	RA10	PIC32 PGD line in Microkite
P31	RA7	~LED and PIC32 PGC in Microkite
P32	RC3	~uSD (plus LED) in Microkite