1 Introduction....2
1.1 System States....2
1.2 Word Types....3
1.3 Stack Comments....4
2 Stack....5
2.1 Data Stack....5
2.2 Return Stack....5
3 Memory....6
Data Memory....6
3.2 Program Memory....8
3.3 Data Stack Memory....8
4 Branching....9
4.1 Primitives....9
4.2 Compiler Structures....9
4.3 Traps....11
5 Integer Arithmetic....12
5.1 Unary....12
5.2 Binary....12
5.3 Shift....14
5.4 Comparison....16
6 Floating Point....17
6.1 Conversion....17
6.2 Arithmetic....18
6.3 Comparison....18
6.4 Transcendental Functions....19
6.5 Scaling....19
7 Registers....21
7.1 Status (-1)....21
7.2 Dsp (-2)....22
7.3 Rsp (-3)....22
7.4 Int-reg (-4)....22
7.5 Flag-reg (-5)....23
7.6 Version-reg (-6)....23
7.7 Debug-reg (-7)....23
7.8 Ctrl-reg (-8)....23
7.9 Time-reg (-9)....24
8 Multitasker....25
8.1 Task Scheduling....25
8.2 Task Control....27
8.3 Semaphors....30
8.4 Catch and Throw....30
9 Cross Compiler....31
9.1 Executables....31
9.2 Compilers....32
9.3 Defining Words....33
9.4 Create New Defining Words....33
9.5 Object Code Output Files....34
9.6 Libraries....34
9.7 Optimizations....35
10 Debugger....36
10.1 Configuring the RS232 interface....36
10.2 Entering and Exiting the Debugger....36

10.3 Displaying Information 37
10.4 Debugging Commands..... 38
10.5 Single Stepping..... 39
11 microForth Coding Standard 41
11.1 Readability..... 41
11.2 Understandability 43
12 Index.....45

1 Introduction

The microCore (μCore) architecture is a hardware implementation of the Forth virtual machine. This document is a glossary of microForth (μForth), its assembler, operating system, and debugger.

The development environment is an interactive system consisting of a host computer running the cross compiler, and a target system, for which the code is generated, loaded into, and debugged on the target via a 2-wire umbilical UART link.

1.1 System States

A stand alone Forth system has two states:

1. In interpret state, it will immediately execute Forth words in the input stream, which can come from a terminal or an included file.
2. In compile state, a new word with a unique name will be created in so called "colon-definitions". In that case, Forth words in the input stream will not be executed. Instead calls to the words will be compiled in program memory. A special set of words marked "immediate" will be executed nevertheless. Using this mechanism, e.g. conditional constructs like `IF .. ELSE .. THEN` are realized.

μForth, a symbiotic host/target environment, adds considerable complexity having a total of four states:

1. Interpret state on the host executing host words or those target words, which are executable during compilation. This can e.g. be used on the fly to compute literal values for the target on the host. In state 1, `exec?` will return true.
2. Compile state on the host, compiling host words, which may produce target code later on. During target compilation, new macros, new target defining words, or target state display routines can be added to the cross compiler using `Macro:` and `Host:`. In state 2, `comp?` will return true.
3. Compile state on the target, which produces code for the target's program memory. With the exception of the context, this is identical to state 2 and `comp?` will also return true.
4. Debug state executing target words on the target system, controlled by the host. In state 4, `dbg?` will return true. The debugger commands can only be executed in state 4

1.2 Word Types

Each word is marked regarding its implementation and/or context:

C	Compiler that can be used when defining a new target word (colon-definition). It produces target code or alters the compiler's state.
const	A target constant.
ext	Only present when extended := true in **architecture_pkg.vhd**
float	Only present when with_float := true in **architecture_pkg.vhd**
H	Word that can be executed in the host context as target compilation support.
I	μCore instruction, executed in one or two cycles
Mn	μCore macro, which will be expanded in-line, consisting of n instructions. The disassembler recognizes macros and displays them as a single line of instructions.
T	Word that can be executed in the target context creating data objects, displaying information or changing the compiler's state.
W	Forth word, conventionally called subroutine. Words are implicitly called when compiled inside colon definitions.
|	Separator for alternatives.

Words are categorized w.r.t. functionality:

- Stack (data, return)
- Memory (data, program)
- Branches (call, exit, traps)
- Arithmetic (unary, binary, shift, comparison)
- Floating Point
- Registers
- Multitasker
- Cross compiler
- Debugger

1.3 Stack Comments

Each instruction, macro, or word has a stack effect, which is shown in symbolic form right next to each word. Data stack comments are surrounded by "(" and ")". Return stack comments by "(R: " and ")". Stack items are characterized by abbreviations to indicate their semantics:

--	Used to mark the moment of execution in stack comments: Arguments to the left of **--** are the inputs, those to the right are the foutputs.
addr	Data memory address
d	2s-complement double number occupying two stack elements. The most significant word is on top of the least significant word.
exp	2s-complement exponent. Its size in number of bits used is defined by `exp_width` in **architecture_pkg.vhd**.
flag **tf, ff**	0 = false, any other number = true for input arguments, all bits set for output arguments. tf ::= true flag, ff ::= false flag
float	Floating point number, `data_width` wide. The mantissa takes the more significant bits, the exponent the less significant bits. Meaningful floating point arithmetic is possible for data_widths >= 24 and exp_widths >= 6.
lit	Literal value accumulated in TOS, preceding an opcode
man	2s-complement mantissa..
mask	A bit mask. Usually only one single bit will be set.
n	2s-complement number.
op	8 bit instruction.
paddr	Program memory address.
u	Unsigned number
ud	Unsigned double number. The most significant word is on top of the least significant word.
xt	Execution Token. A program memory address that can be **execute**d.

Therefore, `<name>( addr -- n )` means: word `<name>` needs an address as input argument, consumes it, and leaves a signed number as output. If there are two or more arguments, the arguments further to the right are more "on top" of stack than the ones to the left.

Only those words, which are not ANSI Standard Forth are explained. Standard words are just listet to show their presence.

2 Stack

2.1 Data Stack

clear **(... --) W, monitor.fs**
Empties the data stack.

depth **(-- u) W, monitor.fs**
u is the number of items on the data stack before executing depth.

drop **(n --) I, opcodes.fs**

nip **(n1 n2 -- n2) M2 | ext I, opcodes.fs**
Removes the second item from the stack.

dup **(n -- n n) I, opcodes.fs**

?dup **(n -- n n | 0) I, opcodes.fs**

over **(n1 n2 -- n1 n2 n1) I, opcodes.fs**

swap **(n1 n2 -- n2 n1) I, opcodes.fs**

rot **(n1 n2 n3 -- n2 n3 n1) I, opcodes.fs**

-rot **(n1 n2 n3 -- n3 n1 n2) I, opcodes.fs**

2drop **(d --) M2, opcodes.fs**

2dup **(d -- d d) M2, opcodes.fs**

2over **(d1 d2 -- d1 d2 d1) W, forth_lib.fs**

2swap **(d1 d2 -- d2 d1) W, forth_lib.fs**

2.2 Return Stack

rclear **(R: ... --) W**
Empties the return stack. This word must be used with care, because there will no longer be a return address on the return stack for an EXIT.

r@ **(-- n) (R: n -- n) I, opcodes.fs**

r> **(-- n) (R: n --) I, opcodes.fs**

>r **(n --) (R: -- n) I, opcodes.fs**

rdrop **(--) (R: n --) M2 | ext I, opcodes.fs**

local **(offset -- addr) I, opcodes.fs**
Converts an offset into the return stack into its equivalent data memory address.

3 Memory

μCore has three independent memory areas:

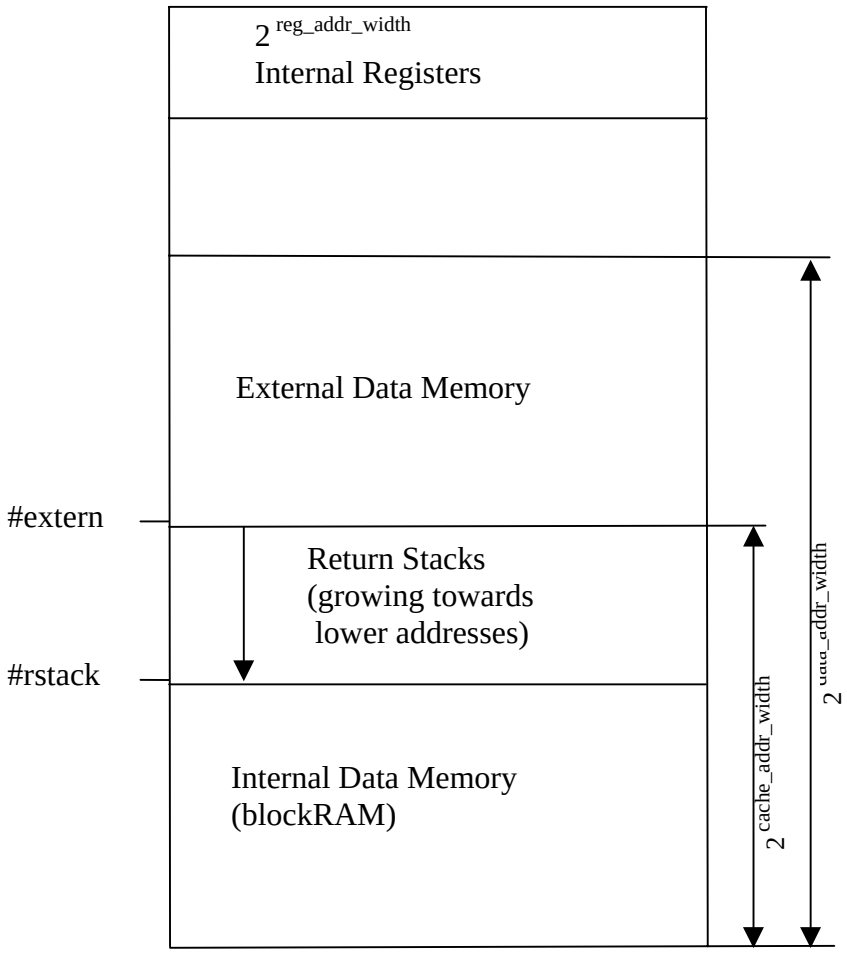
- Data memory and return stack, data_width wide.
- Program memory - 8 bits wide
- Data stack memory - data_width wide.

3.1 Data Memory

The **Internal Data Memory** is located from 0 on upwards composed of the FPGA's blockRAM. The return stack is located at the upper end of this area. (Note: There may be several return stacks, one for each task.)

Above the return stack up to $2^{data\_addr\_width}$-1 is the **External Data Memory** area for off the shelf asynchronous SRAM devices.

Internal Registers are memory mapped and they reside at the "upper end" of the data space, i.e. the register addresses are "small" negative numbers when they are interpreted as 2's complement numbers. All memory access operators can be used for register access as well.



Data Memory

`ld` **`( addr -- n addr ) I`, opcodes.fs**
Load the content of the memory cell or register at `addr` into NOS. `addr` remains in TOS.

`@` **`( addr -- n ) M2 | ext I`, opcodes.fs**

`st` **`( n addr -- addr ) I`, opcodes.fs**
Store the content of NOS into the memory cell or register at `addr`. `addr` remains in TOS, the content of NOS is dropped.

! **(n addr --) M2, opcodes.fs**

2@ **(addr -- d) W, forth_lib.fs**

2! **(d addr --) W, forth_lib.fs**

lld **(u -- n rsp+u) M2, opcodes.fs**

Local load. u is an index into the return stack, which resides in the data memory. At first, the effective address as sum of u and RSP, the return stack pointer is computed and a load is performed from the effective address. The effective address remains in TOS, the content of the effective address is pushed into NOS.

l@ **(u -- n) M3 | ext M2, opcodes.fs**

Local fetch. Macro lld drop.

lst **(n u -- rsp+u) M2, opcodes.fs**

Local store. u is an index into the return stack, which resides in the data memory. At first, the effective address as sum of u and RSP, the return stack pointer is computed. n, the content of NOS is stored at the effective address, NOS is popped and the effective address remains in TOS.

l! **(n u --) M3, opcodes.fs**

Local store. Macro lst drop.

+st **(n addr -- addr) ext I, opcodes.fs**

Adds n to the content of the memory cell at addr. This is an indivisible two cycle read-modify-write operation.

+! **(n addr --) ext M2, opcodes.fs | W, forth_lib.fs**

Add n to the content of the memory cell at addr. Macro +st drop. This is an indivisible two cycle read-modify-write operation.

inc **(addr --) W, forth_lib.fs**

The memory cell at addr is incemented by one as an indivisible read-modify-write operation.

dec **(addr --) W, forth_lib.fs**

The memory cell at addr is decremented by one as an indivisible read-modify-write operation.

on **(addr --) W, forth_lib.fs**

The value -1 (all bits set) is written to the memory cell at addr.

off **(addr --) W, forth_lib.fs**

The value 0 (all bits reset) is written to the memory cell at addr.

erase **(addr len --) W, forth_lib.fs**

fill **(addr quantity pattern --) W, forth_lib.fs**

move **(addr1 addr2 u --) W, forth_lib.fs**

An intelligent cmove, which does not overwrite itself moving u memory cells from addr1 to addr2.

`initialization ( -- )` `W`, microcross.fs

`end` automatically creates this word at the very end of the compiled code. When executed, it will initialize data memory. The phrase `call initialization` compiles a forward call to the initialization code. Usually used at the beginning of the boot routine.

`initialized ( -- )` `T`, microcross.fs

Data memory assignments using `,` or `!` or `2!` interpretively following `initialized` will be included in the `initialization`.

`uninitialized ( -- )` `T`, microcross.fs

Data memory allocations that follow `uninitialized` up to the next `initialized` will not be initialized. Data memory can be allocated using e.g. `allot` or `Variable`.

3.2 Program Memory

`p@ ( paddr -- op )` `M2`, opcodes.fs

`op` is the content of programm memory at `paddr`. It will only be available when constant `with_prog_rw` is set to true in **architecture_pkg.vhd**.

`p! ( op paddr -- )` `M2`, opcodes.fs

`op` will be stored in program memory at `paddr`. This word will only be available when constant `with_prog_rw` is set to true in **architecture_pkg.vhd**.

3.3 Data Stack Memory

Data stack memory can not be accessed by software directly. Apart from using the data stack operators, the data stack pointer register (**DSP**) can be redirected.

4 Branching

These instructions are presented in three categories:

1. Primitives are instructions that modify program flow.
2. Compiler Structures use the primitives for control flow compiler constructs.
3. Traps are hardware or software single cycle calls to fixed addresses.

4.1 Primitives

The primitives are single cycle instructions, which modify program flow. Usually, they are not directly used by the programmer, who uses the compiler control structures of the following chapter instead. Those structures compile the primitives in a user friendly way.

`branch ( lit | addr -- ) I`, opcodes.fs
When preceded by `lit` instructions, a relative branch is performed. When preceded by an address, an absolute branch to `addr` is performed.

`0=branch ( flag lit -- ) I`, opcodes.fs
Conditional branch. Performs a relative or absolute `branch` when flag is zero.

`tor-branch ( lit -- ) ( R: u -- u-1 ) I`, opcodes.fs
Conditional branch. Performs a relative or absolute `branch` when the number `u` in TOR is non-zero. `u` will be decremented when the branch is taken. When it is zero the branch is not taken and `u` will be discarded.

`jsr ( lit | addr -- ) ( R: -- addr' ) I`, opcodes.fs
When preceded by `lit` instructions, a relative call is performed. When preceded by an address, an absolute call to `addr` is performed. `Jsr` pushes the content of the program counter (pc) on the return stack. Pc points to the instruction just after the `jsr` instruction.

`exit ( -- ) ( R: addr -- ) I`, opcodes.fs
Subroutine return. Branches to the address on the return stack and pops the return stack. Compiled by `;`.

`nz-exit ( flag -- ) (R: addr -- ) ext I`, opcodes.fs
Conditional exit. Execution continues at `addr` when `f` is non zero. Otherwise, execution continues at the next sequential instruction. Both `f` and `addr` are dropped off the stacks.

`iret ( status -- ) ( R: addr -- ) I`, opcodes.fs
Interrupt return. In addition to `exit`, `iret` also restores the `Status` register from the data stack.

4.2 Compiler Structures

These target code compilers modify program flow using the primitives above.

`execute ( xt -- ) M2`, opcodes.fs
Performs a subroutine call to execution token `xt`.

`IF ( flag -- ) C`, microcross.fs

`ELSE ( -- ) C`, microcross.fs

THEN (--) C, microcross.fs

BEGIN (--) C, microcross.fs

WHILE (flag --) C, microcross.fs

It may be used multiple times after BEGIN. All WHILEs will be resolved by the closing REPEAT.

REPEAT (--) C, microcross.fs

BEGIN ... REPEAT constitutes and endless loop.

UNTIL (flag --) C, microcross.fs

?EXIT (flag --) (R: addr --) ext I | M3, microcross.fs

Conditional exit. Performs an exit when the flag is non-zero using the nz-exit primitive. The flag will be dropped. The return stack will only be popped when exit is actually executed.

FOR (u --) (R: -- u) C, microcross.fs

Start of a counted loop. Used in the form FOR ... NEXT. The loop index u will be pushed on the return stack. It may be retrieved inside the loop using r@. The loop will be executed u+1 times. When leaving the loop prematurely with an EXIT, do not forget to discard the loop index from the return stack with rdrop before executing EXIT.

?FOR (u --) (R: -- u) C, microcross.fs

Start of a counted loop. Used in the form ?FOR ... NEXT. The loop index u will be pushed on the return stack. It may be retrieved inside the loop using r@. The loop will be executed u times. If u is zero the loop will not be executed at all. When leaving the loop prematurely with an EXIT, do not forget to discard the loop index from the return stack with rdrop before executing EXIT.

NEXT (--) (R: u -- u-1) C, microcross.fs

The closing word for FOR and ?FOR. NEXT compiles the tor-branch primitive and therefore, it branches back to FOR or ?FOR until the loop index u on the return stack is zero. While branching the loop index is decremented by one. If the loop index is zero, execution continues after NEXT and the loop index is popped off the return stack. Please note that FOR ... NEXT loops may be nested as well as interrupted, because the loop index is kept on the return stack.

DO (n1 n2 --) (R: -- n3 n4) C, forth_lib.fs

?DO (n1 n2 --) (R: -- n3 n4) C, forth_lib.fs

LOOP (--) (R: n1 n2 -- n1 n2) (R: n1 0 --) C, forth_lib.fs

I (-- n1) (R: n2 n3 -- n2 n3) ext I, opcodes.fs | W, forth_lib.fs

bounds (start quan -- limit start) M4, forth_lib.fs

Label (<name> --) T C, microcross.fs

Compiles <name> into the dictionary as a constant, which holds the current program memory address. If <name> is the destination of a preceeding GOTO, ?GOTO, or CALL, pending forward references will be resolved. Label may be used inside and outside of colon definitions.

?GOTO (<label> --) C, microcross.fs
Compiles a conditional branch to `<label>`. `<label>` may be a label or colon definition. `?GOTO` supports forward referencing, i.e. `<label>` may be defined later on.

GOTO (<label> --) C, microcross.fs
Compiles an unconditional branch to `<label>`. `<label>` may be a label or colon definition. `GOTO` supports forward referencing, i.e. `<label>` may be defined later on.

CALL (<label> --) C, microcross.fs
Compiles an unconditional call to `<label>`. `<label>` may be a label or colon definition. `CALL` supports forward referencing, i.e. `<label>` may be defined later on.

ADDR (<label> --) C, microcross.fs
Compiles a literal with the program memory address of `<label>`. `ADDR` supports forward referencing, i.e. `<label>` may be defined later on.

4.3 Traps

trap-addr (trap# -- paddr) H T, microcross.fs
Convert **trap#** into its corresponding trap address in program memory.

TRAP: (trap# <name> --) T, microcross.fs
Defines `<name>` as trap `trap#`. The code between `TRAP:` and the closing `;` will actually be compiled at the trap address (see: `trap-addr`) and therefore, this definition defines the processor's behaviour for the specific trap. 2^{trap_width} instructions are set aside for each trap in **architecture_pkg.vhd**. Longer trap service routines have to compile a branch to the actual trap server. For the sake of readability, traps are usually defined after the boot routine in the load file.

#reset (-- trap#) Const, microcross.fs
Reset vector `trap#` defined by the hardware.

#isr (-- trap#) Const, microcross.fs
Interrupt service routine `trap#` defined by the hardware.

#psr (-- trap#) Const, microcross.fs
Pause service routine `trap#` defined by the hardware.

#break (-- trap#) Const, microcross.fs
Break point service routine `trap#` used by the debugger.

#does> (-- trap#) Const, microcross.fs
`Does>` runtime primitive `trap#` used for user defining words.

#data! (-- trap#) Const, microcross.fs
Data memory initialization primitive `trap#` used in the automatically generated `Initialization` code.

5 Integer Arithmetic

noop (--) | (lit -- n) I, opcodes.fs

No-operation. Noop is used to convert a literal value, which has been accumulated by a sequence of literal instructions into a proper number on the data stack. This is important e.g. when a branch should branch to an absolute address: When the address is the result of a sequence of literal instructions, `noop` must precede the `branch` instruction or else a relative branch would be taken.

5.1 Unary

not | invert (n1 -- n2) I, opcodes.fs

Bitwise logical not.

0= (n -- flag) I, opcodes.fs

0<> (n -- flag) M2, opcodes.fs

0< (n1 -- flag) I, opcodes.fs

negate (n -- -n) M2, opcodes.fs

abs (n -- +n) W, forth_lib.fs

true (-- tf) M1, opcodes.fs

false (-- ff) M1, opcodes.fs

1+ (n -- n+1) M2, opcodes.fs

1- (n -- n-1) M2, opcodes.fs

sqrt (u -- rem root) ext W, forth_lib.fs

Square root of `u`. `u = root * root + rem`. For a high level version of `sqrt` see **forth_lib.fs**.

Double precision

extend (n -- d) M2, opcodes.fs

Sign extend `n`.

d0= (d -- flag) M2, opcodes.fs

dnegate (d1 -- d2) W, forth_lib.fs

dabs (d -- +d) W, forth_lib.fs

5.2 Binary

Binary operations as well as complex math may lead to a result, which can not be represented in the chosen data word width. This is an overflow condition and the overflow status flag will be set accordingly. Usually, the result of an operation that produces the overflow is grossly misleading from a mathematical point of view. To this end, the `+sat` and `u+sat` instructions have been added to the word set.

+ **(n1 n2 -- n1+n2) I, opcodes.fs**

`2dup +` will be optimized into a single instruction.

+c **(n1 n2 -- n1+n2+carry) I, opcodes.fs**

Add with carry. Used for extended precision arithmetic. `2dup +c` will be optimized into a single instruction.

+sat **(n1 n2 -- n3) ext I, opcodes.fs | W, forth_lib.fs**

Add with saturation arithmetic. `n3` is the result of `n1+n2` when no over- or underflow occurs. In case of an overflow, `n3` is the largest positive number ($7FFF for 16 bits), and in case of an underflow, it is the smallest negative number ($8000 for 16 bits). `+sat` can be used to keep control loops from oscillating during over- or underflow conditions.

u+sat **(u1 n -- u2) W, forth_lib.fs**

Unsigned add with saturation arithmetic. `u2` is the result of `u1+n` when no over- or underflow occurs. In case of an overflow, `u2` is the largest unsigned number ($FFFF for 16 bits), and in case of an underflow it is zero. `u+sat` can be used to keep control loops from oscillating during over- or underflow conditions.

- **(n1 n2 -- n1-n2) I, opcodes.fs**

`swap -`, `2dup -` as well as `2dup swap -` will be optimized into a single instruction.

and **(u1 u2 -- u3) I, opcodes.fs**

`2dup and` will be optimized into a single instruction.

or **(u1 u2 -- u3) I, opcodes.fs**

`2dup or` will be optimized into a single instruction.

xor **(u1 u2 -- u3) I, opcodes.fs**

`2dup xor` will be optimized into a single instruction.

***** **(n1 n2 -- n3) M2, opcodes.fs | W, forth_lib.fs**

When the product `n3` can not be respresented as a single number, the overflow status flag will be set.

/mod **(n u -- urem quot) W, forth_lib.fs**

/ **(n u -- quot) W, forth_lib.fs**

mod **(n u -- urem) W, forth_lib.fs**

u/mod **(u1 u2 -- urem uquot) W, forth_lib.fs**

u/ **(u1 u2 -- uquot) W, forth_lib.fs**

umod **(u1 u2 -- urem) W, forth_lib.fs**

***/mod** **(n1 n2 u -- urem n3) W, forth_lib.fs**

***/** **(n1 n2 u -- n3) W, forth_lib.fs**

Double precision

d+ **(d1 d2 -- d3) W, forth_lib.fs**

d- **(d1 d2 -- d3) W, forth_lib.fs**

m+ (d n -- d) W, forth_lib.fs

m* (n1 n2 -- d) I, opcodes.fs | W, forth_lib.fs

um* (u1 u2 -- ud) I, opcodes.fs | W, forth_lib.fs

m/mod (d u -- rem quot) W, forth_lib.fs

um/mod (ud u -- urem uquot) W, forth_lib.fs

ud* (ud u -- udprod) W, forth_lib.fs

ud/mod (ud u -- urem udquot) W, forth_lib.fs

5.3 Shift

2* (u1 -- u2) M2, opcodes.fs

Logical shift left.

2/ (n1 -- n2) M2, opcodes.fs

Arithmetic shift right.

u2/ (u1 -- u2) M2, opcodes.fs

Logical shift right.

shift (u1 n -- u2) I | M2, opcodes.fs

A general purpose logical shift operation. `u1` will be shifted left or right depending on `n`, resulting in `u2`. If `n` is 0 nothing happens. If `n` is >0, `u1` will be shifted `n` times to the left. If `n` is <0, `u1` will be logically shifted `n` times to the right.
When `|n| >= data_width`, `u2` will be 0.
The carry status flag will be set to the last bit shifted out of `u1`.
If a hardware multiplier is not available, it takes `n` cycles to execute.
If a hardware multiplier is available, it takes two cycles to execute.

ashift (n1 n2 -- n3) I | M2, opcodes.fs

A general purpose arithmetic shift operation. `n1` will be shifted left or right depending on `n2`, resulting in `n3`. If `n2` is 0 nothing happens. If `n2` is >0, `n1` will be shifted `n2` times to the left. If `n2` is <0, `n1` will be arithmetically shifted `|n2|` times to the right, i.e. the sign bit will be shifted in from the left.
The carry status flag will be set to the last bit shifted out of `n1`.
If a hardware multiplier is not available, it takes `n` cycles to execute.
If a hardware multiplier is available, it takes two cycles to execute.

2 (u -- 2**u) W, forth_lib.fs**

Returns the exponentiation of 2 by `u`. Used to convert e.g. number of address bits into a corresponding address range or a bit number into a bit mask.

pack (char u1 -- u2) W, forth_lib.fs

Shift `u1` 8 bits to the left and insert `char` at the vacated position. Equivalent to the phrase `$100 * or`.

unpack (u1 -- char u2) W, forth_lib.fs

`u2` is the result of logically shifting `u1` 8 bits to the righ. `char` are the 8 least significant bits of `u1`. Equivalent to the phrase `0 $100 um/mod`.

With Multiplier

The following words will only be available when a single cycle hardware multiplier is available (`with_mult := true` in **architecture_pkg.vhd**).

`mshift` (u1 n -- u2 u3) I, opcodes.fs

Logical barrel shift. When `n` is positive, shift `u1 n` positions to the left. When it is negative, logically shift `u1 n` positions to the right. `u2` is the shift result, `u3` are the bits, which have been shifted out of `u1`, and the carry status flag is set to the last bit shifted out. When `n` **i**s zero, `u2` equals `u1` and `u3` is zero.

`mashift` (n1 n2 -- n3 n4) I, opcodes.fs

Arithmetic barrel shift. When `n2` is positive, shift `n1 n2` positions to the left. When it is negative, arithmetically shift `n1 |n2|` positions to the right. `n3` is the shift result, `n4` are the bits, which have been shifted out of `n1`, and the carry status flag is set to the last bit shifted out. When `n2` **i**s zero, `n3` equals `n1` and `n4` is zero.

`rotate` (u1 n -- u2) M2, opcodes.fs

Cryptographic bit rotate instruction. Macro `mshift or`.

Without Multiplier

The following words will only be available when no single cycle hardware multiplier is available (`with_mult := false` in **architecture_pkg.vhd**).

`c2/` (u1 -- u2) I, opcodes.fs

`u1` is shifted one position to the right. The MSB of `u2` is set to the carry status flag, and the carry status flag is set to the LSB of `u1`. `c2/` is used to realize multi-precision shift right operations when no single cycle multiplier is available.

`c2*` (u1 -- u2) I, opcodes.fs

`u1` is shifted one position to the left. The LSB of `u2` is set to the carry status flag, and the carry status flag is set to the MSB of `u1`. `c2*` is used to realize mult-precision shift left operations when no single cycle multiplier is available.

Double precision

`d2*` (ud1 -- ud2) W, forth_lib.fs

`ud1` is shifted one position to the left, shifting zero into the LSB of `ud2`. The carry status flag is set to the MSB of `ud1`.

`ud2/` (ud1 -- ud2) W, forth_lib.fs

`ud1` is logically shifted one position to the right, shifting zero into the MSB of `ud2`.The carry status flag is set to the LSB of `ud1`.

`d2/` (d1 -- d2) W, forth_lib.fs

`d1` is arithmetically shifted one position to the left, shifting the MSB of `d1` into the MSB of `d2`. The carry status flag is set to the LSB of `d1`.

dshift (ud1 n -- ud2) W, forth_lib.fs

Double precision logical shift operation. ud1 will be shifted left or right depending on n, resulting in ud2. If n is 0 nothing happens. If n is >0, ud1 will be shifted n times to the left. If n is <0, ud1 will be logically shifted |n| times to the right.
The carry status flag will be set to the last bit shifted out of ud1.

dashift (d1 n -- d2) W, forth_lib.fs

Double precision arithmetic shift operation. d1 will be shifted left or right depending on n, resulting in d2. If n is 0 nothing happens. If n is >0, d1 will be shifted n times to the left. If n is <0, d1 will be arithmetically shifted |n| times to the right, i.e. the sign bit will be shifted in from the left. The carry status flag will be set to the last bit shifted out of d1.

5.4 Comparison

= (n1 n2 -- flag) M2, opcodes.fs

/= (n1 n2 -- flag) M3, opcodes.fs

Flag is true, when n1 is not equal to n2.

< (n1 n2 -- flag) I, opcodes.fs

> (n1 n2 -- flag) M2, opcodes.fs

>= (n1 n2 -- flag) M2, opcodes.fs

<= (n1 n2 -- flag) M3, opcodes.fs

u< (n1 n2 -- flag) W, forth_lib.fs

u> (n1 n2 -- flag) W, forth_lib.fs

max (n1 n2 -- max) W, forth_lib.fs

min (n1 n2 -- min) W, forth_lib.fs

umin (u1 u2 -- umax) W, forth_lib.fs

umax (u1 u2 -- umin) W, forth_lib.fs

within (n [low [high -- flag) W, forth_lib.fs

case? (n1 n2 -- n1 ff | tf) W, forth_lib.fs

Comparison operator. When n1 equals n2, case? leaves true and discards the input arguments. When n1 and n2 are not equal, false is returned and only n2 is discarded. It is used in the following form for simple case statements:

```
...
<put value on the data stack>
1 case? IF  ... EXIT THEN
2 case? IF  ... EXIT THEN
...
```

Double Precision

d= (d1 d2 -- flag) W, forth_lib.fs

6 Floating Point

Floating point numbers (reals) are `data_width` wide and therefore, they fit on μCore's data stack. The exponent is `exp_width` wide.

The MSB of a real is its sign bit. The mantissa is a 2's complement number whose leading digit after the sign has been dropped, because in a normalized number this is always the complement of the sign bit.

The floating point instructions are only synthesized when `with_float := true` in **architecture_pkg.vhd**. The basic floating point words are defined in **float_lib.fs** and they are compiled both into the target system as well as into the host system and therefore, floating point operations can be executed during target compilation e.g. in order to compute floating point constants.

6.1 Conversion

`normalize ( man1 exp1 -- man2 exp2 ) float I`, opcodes.fs
: `man1` is shifted to the left until its MSB-1 bit is different from its MSB, the signbit. For each shift, `exp1` is decremented by one. `normalize` is an auto-repeat instruction, which can be interrupted any time.

`>float ( man exp -- r ) float M2`, opcodes.fs
: Converts 2s-complement numbers `man` and `exp` into normalized real `r`.

`float> ( r -- man exp ) float I`, opcodes.fs
: Converts real `r` into 2s-complement numbers `man` and `exp`.

`float ( n -- r ) M3`, float_lib.fs
: Converts signed integer `n` into real `r`.

`integer ( r -- n ) W`, float_lib.fs
: Converts the integer part of real `r` into signed integer `n`.

`>ieee ( r -- ieee ) H`, float_lib.fs
: Only available when data_width = 32. Converts μCore's real into an IEEE-754 standard floating point number. This conversion function is only available on the host.

`ieee> ( ieee -- r ) H`, float_lib.fs
: Only available when data_width = 32. Converts an IEEE-754 standard floating point number into μCore's real. This conversion function is only available on the host.

6.2 Arithmetic

Primitives

`*.` **`( n1 x -- n2 ) float I`, opcodes.fs | `W`, forth_lib.fs**

If a hardware multiplier is available it executes in one cycle.
It is a multiply primitive for polynomial approximations using the Horner scheme. `n1` is the result accumulated so far, `x` is the functions unsigned x value and `n2` are the rounded most significant bits of the double precision product `n1 * x`.

`log2` **`( u1 -- u2 ) float I`, forth_lib.fs**

Only available when a hardware multiplier is present (`with_mult := true` in **architecture_pkg.vhd**). Fractional logarithm dualis. `1 > u1 >= 0` and `1 > u2 >= 0`. It takes data_width+3 cycles to execute.

`int.frac` **`( r -- n1 u2 )` `W`, float_lib.fs**

Real `r` is split up into `n1`, its integer, and `u2`, its fractional part. Used for defining trascendental functions using a polynomial approximation for the fractional part.

Operators

`f+` **`( r1 r2 -- r3 )` `W`, float_lib.fs**

`f-` **`( r1 r2 -- r3 )` `W`, float_lib.fs**

`f*` **`( r1 r2 -- r3 )` `W`, float_lib.fs**

`f/` **`( r1 r2 -- r3 )` `W`, float_lib.fs**

`1/f` **`( r1 -- r2 )` `W`, float_lib.fs**

`f2*` **`( r1 -- r2 )` `W`, float_lib.fs**

`f2/` **`( r1 -- r2 )` `W`, float_lib.fs**

`fnegate` **`( r1 -- r2 )` `W`, float_lib.fs**

`fabs` **`( r1 -- r2 )` `W`, float_lib.fs**

6.3 Comparison

`f0=` **`( r1 -- flag )` `W`, float_lib.fs**

`f<` **`( r1 r2 -- flag )` `W`, float_lib.fs**

`f<=` **`( r1 r2 -- flag )` `W`, float_lib.fs**

`f>` **`( r1 r2 -- flag )` `W`, float_lib.fs**

`f>=` **`( r1 r2 -- flag )` `W`, float_lib.fs**

6.4 Transcendental Functions

`fsin` **`( r1 -- r2 )`** **W, f_sin_cos.fs**
`r2 = sine(r1). r1` in radian.

`fcos` **`( r1 -- r2 )`** **W, f_sin_cos.fs**
`r2 = cosine(r1). r1` in radian.

`degree` **`( fdeg -- frad )`** **W, f_sin_cos.fs**
scales real `fdeg` in degree converting into `frad` in radian

`fpi/2` **`( -- r )`** **`const`, f_sin_cos.fs**
This constant has been derived from the fractional expression 104348 / 33215.

`flog2` **`( r1 -- r2 )`** **W, f_exp_log.fs**

`fexp2` **`( r1 -- r2 )`** **W, f_exp_log.fs**

`fln` **`( r1 -- r2 )`** **W, f_exp_log.fs**

`fexp` **`( r1 -- r2 )`** **W, f_exp_log.fs**

6.5 Scaling

In μForth, integers and scaling is used for floating point number input and output. This simplifies the floating point package considerably, because no numerical character string processing is needed.

`fscale` **`( r1 n -- r2 )`** **W, float_lib.fs**
When `n > 0`, `r1` is multiplied with `n`. When `n < 0`, `r1` is divided by `|n|`.

`micro` **`( r1 -- r2 )`** **W, float_lib.fs**
`r2 = r1 / 1,000,000.`

`milli` **`( r1 -- r2 )`** **W, float_lib.fs**
`r2 = r1 / 1,000.`

`kilo` **`( r1 -- r2 )`** **W, float_lib.fs**
`r2 = r1 * 1,000`

`mega` **`( r1 -- r2 )`** **W, float_lib.fs**
`r2 = r1 * 1,000,000.`

Example

In a laser control loop, an NTC resistor measures a laser's temperature and we want to compute both ways: Given the NTC's resistance in Ohm, we want to know the temperature in centideg Celsius. And when centideg Celsius are given, we want to know the NTC's resistance in Ohm. A simple pair of formulas for this is:

$$T = \frac{B}{\ln\left(R/r_\infty\right)} \qquad R = r_\infty e^{B/T}$$

And this is the code that solves the problem when **float_lib.fs** has been included as library:

```
&3892  float Constant B-factor   \ NTC material constant
-&298  float Constant -T0        \ 25 Celsius in Kelvin * -1
&10000 float Constant R0         \ Reference resistance in Ohm at 0 Celsius
&27300       Constant 0_degC     \ 0 centideg Celsius in centideg Kelvin

B-factor -T0 f/ R0 fln f+ fexp Constant R_lim

: T>R   ( degC*100 -- Ohm )  \ R = ...
   0_degC + float  -&100 fscale  B-factor swap f/   fexp   R_lim f*   integer
;
: R>T   ( Ohm -- degC*100 )  \ T = ...
   float R_lim f/   fln   B-factor swap f/   &100 fscale integer 0_degC -
;
```

We can see the use of scaling for the temperature, which will be computed in centideg, i.e. 27300 centideg = 273 deg.

In the `T>R` routine, we have centideg Celsius on the stack. Adding `0_degC` converts this into centideg Kelvin, which we then convert into a real. In the next step we scale by -100, i.e. we divide by 100 and as a result, we have the real number converted into deg Kelvin, from which our formula will directly compute equivalent Ohms, which are finally convert back into an integer.

In the `R>T` routine, we have Ohms on the stack, which eventually results in a real number that constitutes the equivalent temperature in deg Kelvin. It is scaled by 100, i.e. we multiply with 100 to convert into centideg Kelvin, which we then convert into an integer. Finally, we convert into centideg Celsius by subracting integer constant `0_degC`.

7 Registers

Registers are memory mapped and they reside at the upper end of the data address space. If the register addresses are represented as signed numbers, they are located from addresses min_registers (-1) downto max_registers (e.g. -9). Therefore, a register can most often be accessed in three cycles: a single `lit` (-64 .. -1) as address followed by `@` (read) or `!` (write).

In VHDL, register indices are defined in **architecture_pkg.vhd** as negative integer constants. They are imported into μForth in **constants.fs**.

7.1 Status (-1)

The processor's `Status` register holds flag bits, which together with the program counter `pc` constitute the processor state. On an interrupt, `Status` will be pushed on the data stack and `pc` will be pushed on the return stack automatically. The `Status` register may be accessed using `@` and `!`. It is composed of the following flags:

Bit-mask	Name	Access	Description
1	`#c`	r/w	The carry flag reflects the result of the most recent `+`, `+c`, `-`, `2*`, `2/`, `u2/`, `shift`, and `ashift` instructions. When subtracting, it is the complement of the borrow bit.
2	`#ovfl`	r/w	The OVerFLow flag reflects the result of the most recent `+`, `+c`, `-`, and `/` instruction.
4	`#ie`	r/w	Interrupt Enable flag. When IE is not set, interrupts will be disabled. On reset, it is set to zero.
8	`#iis`	r/w	The Interrupt-In-Service flag is set when jumping to the interrupt service address. It will be reset when IRET (Interrupt-RETurn) restores the status register at the end of the interrupt service routine . When IIS is set, interrupts are disabled.
$10	`#lit`	r	The LITeral flag keeps the most significant bit of the previous instruction.
$20	`#neg`	r	The NEGative flag reflects the content of the most significant bit of TOS or of NOS when #lit = 1.
$40	`#zero`	r	The ZERO flag reflects the content of TOS or of NOS when #lit = 1
$80	`#sign_div`	r/w	DIVisor sign bit set at the start of signed division
$100	`#sign_den`	r/w	diviDENt sign bit set at the start of signed division
$200	`#unfl`	r/w	floating point underflow

`st-set ( mask -- ) I`, opcodes.fs

Sets the writable `status` flags depending on `mask`. E.g. `#c st_set` will set the `#c` flag to '1'.

`st-reset ( mask -- ) M2`, opcodes.fs

Resets the writable `status` flags depending on `mask`. E.g. `#c st_reset` will reset the `#c` flag to '0'. `st_reset` is equivalent to `not st_set`.

ei (--) M2, opcodes.fs
Enables all interrupts by setting the `#ie` flag of `status` to '1'.

di (--) M2, opcodes.fs
Disables all interrupts by setting the `#ie` flag of `status` to '0'.

ovfl? (-- flag) I, opcodes.fs
Pushes the `#ovfl` flag of `status` on the stack.

carry? (-- flag) I, opcodes.fs
Pushes the `#c` flag of `status` on the stack.

7.2 Dsp (-2)

The **D**ata **S**tack **P**ointer points to the third item below TOS and NOS in the current task's data stack memory area. `Dsp` may be accessed using `@` and `!`. When `Dsp` is set to a new stack location or another task's stack memory area, the phrase `Dsp ! drop` must be used in order to fill both TOS and NOS with the new stack's top items. (Please note that macro `!` already does the first drop pulling the value at the new stack location into NOS.)

7.3 Rsp (-3)

The **R**eturn **S**tack **P**ointer points to the second item below TOR in the current task's return stack data memory area. `Rsp` may be accessed using `@` and `!`. When `Rsp` is set to a new stack location or another task's return stack memory area, the phrase `Rsp ! rdrop` must be used in order to fill TOR with the new return stack's top item.

7.4 Int-reg (-4)

`Int-reg` holds `interrupts` number of bits of `Flag-reg` (see below), which may raise an interrupt. Each flag in `Int-reg` represents an interrupt source.
`<bitmask> Int-reg !` is used to enable a specific interrupt (see `int-enable`**).**
`<bitmask> not Int-reg !` is used to disable a specific interrupt (see `int-disable`**).**
`Int-reg @` returns those interrupts, which are both enabled and raised. This is usually the first phrase of the interrupt service routine. Please note that the interrupt flags are not reset by reading them. Instead, each individual interrupt source has to take care of resetting its respective interrupt flag.

intflags (-- u) M2, opcodes.fs
Macro `Int-reg @`.

int-enable (mask --) M4, opcodes.fs
Enable those interrupts in the interrupt enable register, whose bits are set in `mask`.

int-disable (mask --) M4, opcodes.fs
Disable those interrupts in the interrupt enable register, whose bits are set in `mask`.

7.5 Flag-reg (-5)

`Flag-reg` holds a number of application specific flag bits, which may be accessed using `@` and, when they are used as a semaphors, `!` as well.
0 up to `interrupts-1` least significant bits are defined as interrupt flags (see constant INT_REG and constant interrupts in **architecture_pkg.vhd**). `Flag-reg @` will return the actual status of all flag bits, even if some interrupt flags have been disabled (see `Int-reg`).

`flags` **`( -- u )` M3, opcodes.fs**
Macro `Flag-reg @`.

`flag?` **`( mask -- flag ) ext I`, opcodes.fs `| W`, forth_lib.fs**
Macro `Flag-reg @ and`.

`pass` **`( mask -- )` M3, opcodes.fs**
Used to lock a semaphor bit in `Flag-reg`. `mask pass` will set the bit addressed by `mask` to '1' when it was not set in `Flag-reg`. When it was already set, a pause will be executed until it will have been reset to '0' by another task.

`release` **`( mask -- )` M3, opcodes.fs**
Forces a semaphor bit in `Flag-reg` to '0'.

7.6 Version-reg (-6)

`Version-reg @` returns the version number defined by VHDL constant `version` in **architecture_pkg.vhd**.

`.version` **`( -- )` T, opcodes.fs**
Displays in decimal both μCore's `version` number and `data_width` as defined in **architecture_pkg.vhd**.

7.7 Debug-reg (-7)

The `debug-reg` allows to communicate with the host system via the umbilical link.

`>host` **`( n -- )` M3, opcodes.fs**
transmits data to the host. When the umbilical transmitter is busy, a pause is raised.

`host>` **`( -- n )` M3, opcodes.fs**
receives data from the host. When no data is available, a pause is raised.

7.8 Ctrl-reg (-8)

`Ctrl-reg` is a register that holds several application specific output control bits. `Ctrl-reg @` returns the current setting of all bits. `mask Ctrl-reg !` sets or resets individual control bits.
When the MSB of `mask` is '0', bits, which are set to '1' will be set, and bits set to '0' will not be modified.
When the MSB of `mask` is '1', bits, which are set to '0' will be reset, and bits set to '1' will not be modified.

`ctrl` **`( -- addr )` M2, opcodes.fs**
Alias for `Ctrl-reg. 1 ctrl !` will set the least significant bit to '1'.

-ctrl (mask1 -- mask2 addr) M2, opcodes.fs
Macro `not Ctrl-reg. 1 -ctrl !` will reset the least significant bit to '0'.

ctrl? (mask -- flag) W, forth_lib.fs
```
: ctrl?   Ctrl-reg @ and ;
```

7.9 Time-reg (-9)

`Time-reg` is a wrap around counter that increments at the constant rate of `ticks_per_ms`. It is the time keeping element of μCore.
The longest possible delay equals `2^data_width-1 / ticks_per_ms` [msec].

time (-- ticks) M2, opcodes.fs
Macro `Time-reg @`.

time? (ticks -- flag) I, opcodes.fs
Returns a true flag when `ticks` is less or equal to `time`. Returns a false flag otherwise.

ms (u1 -- u2) W, forth_lib.fs
Converts milliseconds into timer ticks. See: `#ticks_per_ms` in **architecture_pkg.vhd**.

sec (seconds -- ticks) W, forth_lib.fs
Converts seconds into timer ticks. See: `#ticks_per_ms` in **architecture_pkg.vhd**.

ahead (ticks1 -- ticks2) W, forth_lib.fs
`ticks2` is the sum of `ticks1` plus the content of `Time-reg`.

continue (ticks --) W, forth_lib.fs
Wait until `Time-reg` has passed `ticks`. In a multi-tasking system, `pause` would be executed until `Time-reg` has passed `ticks`. (Note: The comparison between `Time-reg` and `ticks` is made in the 2's complement number circle. Therefore, the maximum number for `ticks` is $2^{data\_width-1}$.)

sleep (ticks --) W, forth_lib.fs
Wait until `ticks` have elapsed.

elapsed (time -- ticks) W, forth_lib.fs
Computes the number of ticks elapsed since `time` was pushed on the stack and the current content of `Time-reg`. Used for performance measurement in the form
`time <some application code> elapsed .`

8 Multitasker

μCore's multitasker is a co-operative rather than a pre-emptive one, because:

- It is simple.
- Task switches can only happen at clearly defined moments rather than 'between any two instructions'. Therefore, the multitasker never has to disable interrupts.
- It has a moderate overhead in terms of processing power consumed.

Its disadvantage is the necessity that every task in the system must give other tasks a chance to run or else the entire system will not be balanced. Because μCore is used for real-time control under the control of one programmer or one programming team, this prerequisite can be easily met. The pre-emptive approach is the better solution when you can not predict, which programs will be running on a system.

μCore's multitasker has been designed for real time control. It consists of prioritized and time sharing tasks as well as semaphores for resource locking and process synchronization. Its use in several complex embedded applications has proven its appropriateness.

8.1 Task Scheduling

At the heart of the multitasking system is the 'task list', a linked list of TaskControlBlocks (TCBs). It is anchored at `Priority`. Every TCB starts with the `exec` field that points to a code sequence, which characterizes a task's state. Followed by the `link` field that points to the next TCB in the list. These are the possible code sequences for `exec`:

exec	**description**
`do_priority`	This is performance measurement code for `Priority`. The following fields after `Priority` will be updated: 2: `Prioflag` will be set to true. 3: `time` of the previous update. 4: Longest dead time in `ticks` encountered so far. 5: Task address that produced the longest dead time. 6: PC when the longest dead time was encountered. The trigger for a watchdog counter will be executed here when one is present.
`go_next`	The task is sleeping, go to the next task in the list.
`do_wake`	The task is ready to run and it will wake up. If another task had been running, its context will be saved first.
`do_robin`	This is a code sequence for `RoundRobin`, the anchor of the time sharing part of the task list. `Prioflag` will be set to false and RoundRobin's `link` will be moved one position forward in the circular time sharing list.
`do_time`	Check the time field: When `Time-reg` >= time field the task wakes up.
`do_poll`	Execute the code the poll field points to. When it returns true, the task wakes up, when it returns false it continues sleeping.
`do_poll_tmax`	Polling with timeout, checking both the poll and the time fields.

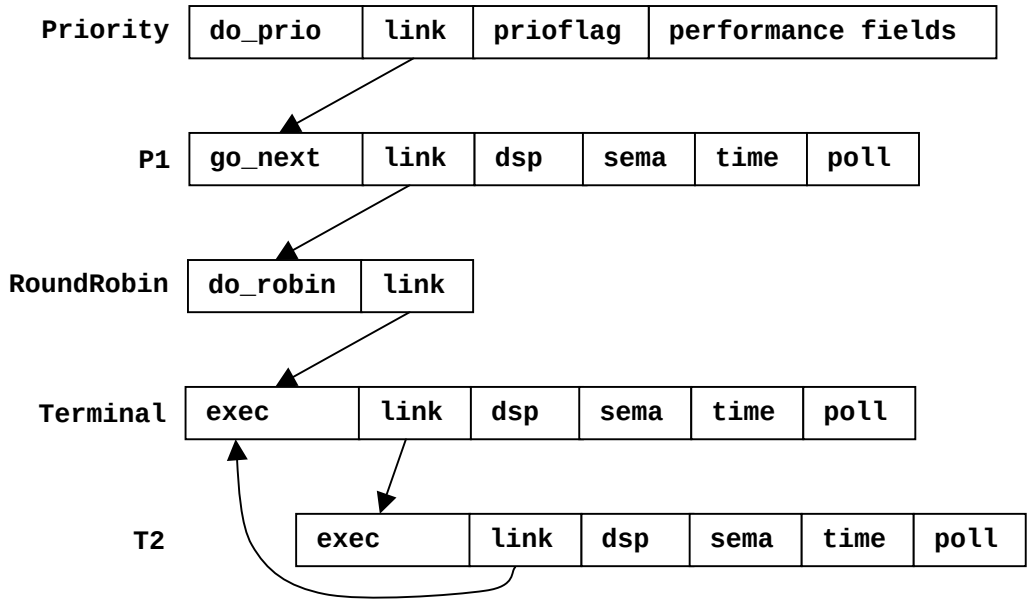
Prioritized tasks are anchored at `Priority` and the last prioritized task points to the task stub `RoundRobin`, which is the anchor of the circular list of time sharing tasks.

Task scheduling consists of fetching the content of the exec field and executing it, initially starting at `Priority`.

After executing the code sequence

```
init
Task P1
Priority P1 schedule
Task T2
Terminal T2 schedule
```

the following task list will have been constructed:



After booting the scheduler starts executing the task list at `Priority`. After doing performance measurement [1], `P1` is probed. Its exec field points to `go_next` and therefore, `P1` is not ready to run. As a consequence, the scheduler arrives at `RoundRobin`, executing `do_robin`.

`do_robin` rewrites `RoundRobin`'s link field to point to the next task in the circular list, and executes the next task's exec field. Should none of the timesharing tasks be ready to run, the scheduler will eventually reach the task whose TCB address equals `RoundRobin`'s link address. This means that no task of the timesharing tasks was ready to run and scheduling continues at `Priority`. This mechanism gives each time sharing task in the circular list an equal opportunity to run.

[1] At **`Priority`**, fields are set aside for performance measurement. These are the **task**, which ran exclusively for the longest time before relinquishing control so far. The **time** it took (in **`Time-reg`** ticks). And the **PC** of said task when it relinquished control. Using this data in the software engineering process makes it easy to find code sequences, which need to execute more `pause`s, because they grab the processor for too long. Mostly, these sequences turn out to be loops doing list processing or extended math computations. See the **`tasks`** command that displays the multitasker's system state. Performance measurement may be disabled by setting PERFORMANCE_MEASUREMENT to False in **task_lib.fs**.

When a task gives up control, all time sharing tasks continue scheduling at `Priority`, whereas prioritized tasks continue at the following task due to variable `Prioflag`.

The `Priority` tasks are part of the real time system, whereas the `RoundRobin` tasks share the remaining processor power equally.

The sema field holds a semaphore's address a task may be waiting for. Releasing a semaphore wakes all tasks, which are waiting for it.

The time field holds the `Time-reg` value when a task should wake up using `do_time` in its exec field.

The poll field points at a polling routine, which returns true or false. It will be executed by `do_poll`. When true, the task will wake up. When false, it continues sleeping. The polling mechanism is a higher performance alternative to waking up a task first and have it do the polling itself, immediately going to sleep again when not ready yet.

`Do_poll_tmax` in the exec field combines both polling and time delay. The task wakes up when either the polling condition has been met or the timeout has been reached. After waking up, a flag is pushed on the stack. When true, the polling condition was met. When false, the timeout was reached instead.

If μCore runs at 25 Mhz, a full task switch (putting the current task to sleep and waking up another task) takes about 7 μsec.

8.2 Task Control

`Taskvariable ( <name> -- ) | <name> ( -- index ) T`, task_lib.fs
is a defining word, which creates an offset constant, whose value is set automatically. Subsequently defined task variables have consecutive offsets. They are used as named indexes into the TCB (TaskControlBlock). The following task variables are used by the multitasking system itself:

```
Taskvariable #t_exec     \ current task execution routine
Taskvariable #t_link     \ link to the next task descriptor block
Taskvariable #t_dsp      \ saved data stack address
Taskvariable #t_sema     \ set to semaphore address when waiting
Taskvariable #t_poll     \ waiting on specific polling routine
Taskvariable #t_catch    \ holds address of the previous catch frame
```

Additional ones may be defined as needed by the application.

`Task ( <name> -- ) | <name> ( -- task ) T`, task_lib.fs
creates a dictionary entry and allocates space for a TaskControlBlock. The TCB holds the task variables, which are local to every task. Please note that the size of the TCB depends on the number of task variables defined when the task is created. Usually all task variables are defined prior to creating the first task.

`Priority ( -- addr ) Const`, task_lib.fs
is the anchor of the task list. It points to the beginning of the prioritized tasks.

`init-performance ( -- ) W`, task_lib.fs
re-initializes the performance measurement fields to zero. This word is only present when PERFORMANCE_MEASUREMENT is set to true in **task_lib.fs**.

RoundRobin (-- addr) Const, task_lib.fs
is the end of the prioritized tasks and the anchor of the time sharing tasks.

Terminal (-- task) Const, task_lib.fs
is the predefined system task, which runs the debugger. It communicates through the serial two wire umbilical link with the host. It is linked into the task list after `RoundRobin` and therefore, it is a time sharing task.

pause (--) W, task_lib.fs
relinquishes control of the currently running task and marks it as ready to run again by writing `do_wake` into its exec field. If it is a prioritized task, the next task in the task list will be probed for execution readiness. If it is a time sharing task, the search for the next task ready to run will start at `Priority`.

halt (--) W, task_lib.fs
relinquishes control of the currently running task without marking it as ready for execution. Please note that the exec field is always set to `go_next` when a task wakes up. This allows an interrupt to overwrite the exec field with `do_wake` while a task is running.

message (n --) W, task_lib.fs
Error messages are negative numbers. They are globally defined in **messages.fs** so that error numbers can be easily interpreted. When `message` is executed in `Terminal`, a text string or numerical error message will be displayed on the host. When it is executed in another task, the task will `halt` and write the error number into its `poll` field. The following errors have been defined in the multitasker:

all_tasks_busy in schedule
not_my_semaphore in unlock
task_not_linked in cancel.

Command `.tasks` displays all error strings/numbers written into `poll` fields.

wake (task --) W, task_lib.fs
writes `do_wake` into the exec field of `task`. Therefore, `task` will run as soon as the scheduler looks at it. `Wake` is executed by a task to wake up another task.

stop (task --) , W, task_lib.fs
writes `go_next` into the exec field of `task`. Therefore, `task` will not run any more. `Stop` is executed by a task to halt another task.

init (--) W, task_lib.fs
has to be executed at the end of the boot phase. It initializes system parameters, which can not be initialized during compilation. Application specific initializations can be chained by defining `: init ( -- ) init <more user code> ;` i.e. at first all previously defined init routines will be executed followed by `<more user code>`. This can be repeated multiple times.

activate (task xt --) W, task_lib.fs
wakes `task` to execute the code at `xt` and initializes its the stacks.

deactivate (task --) W, task_lib.fs
deactivates `task`, setting its exec field to `go_next`. Semaphores owned by `task` will be released. The task remains in the task list.

schedule (task newtask --) W, task_lib.fs

links `newtask` into the task list after `task`. `Task` must be part of the task list or it may be the address of `Priority` or `RoundRobin`. If `newtask` is supposed to be prioritized, `task` must be the task with a higher priority or `Priority` itself. If `task` is a time sharing task or `RoundRobin`, `newtask` will become a time sharing task itself. If no more stack memory space is available, the **all_tasks_busy** error will be produced.

spawn (task newtask xt --) W, task_lib.fs

is `schedule` followed by `activate`.

cancel (task --) W, task_lib.fs

deactivates and removes `task` from the task list. Thereafter, `task` does no longer consume any processing power. When `task` is not part of the task list, a **task_not_linked** error message will be produced.

poll (xt --) W, task_lib.fs

writes a poll routine's execution token `xt` into the active task's poll field, sets the exec field to `do_poll` and halts. Subsequently, the poll routine is executed by `do_poll` and the task wakes up, as soon as the poll routine returns true.

poll-tmax (xt delay -- flag) W, task_lib.fs

writes a poll routine's execution token `xt` into the active task's poll field, writes a wakeup time derived from `delay` into the time field, sets the exec field to `do_poll_tmax`, and halts. Subsequently, `do_poll_tmax` executes the poll routine and checks the time. When either one of these conditions is true, the task wakes up. When the poll routine became true, a true flag is left on the stack. When the time limit had been reached, a false flag is left on the stack.

.tasks (--) T, task_lib.fs

can only be executed interactively while debugging the target via the umbilical link. It displays the current state of the multitasking system.

8.3 Semaphors

`Semaphore ( <name> -- ) | <name> ( -- sema_addr )` T, task_lib.fs

creates semaphore `<name>`, allocates three data memory cells for housekeeping, and links the new semaphor into the semaphor list, anchored at variable `Sema-link`.

The 1st cell is used to store the TCB address of the task owning the semaphore.
The 2nd cell is used as a counter.
The 3rd cell is a pointer to the next semaphore defined.

All semaphores are linked in a list anchored at `Sema-link`.
`<name> link-semaphore` (see below) adds semaphore `<name>` to the list.

`lock ( addr -- )` W, task_lib.fs

locks the semaphore at `addr`. When the semaphore is owned by another task, `addr` is written into the sema field of the active task, and the task is halted.
It wakes up again when the semaphore is unlocked by the other task. Therefore, semaphores can be used for mutual exclusion. Each time a task executes `lock` successfully, the semaphore's count field is incremented by one.

`unlock ( sema_addr -- )` W, task_lib.fs

decrements semaphore's count field. When the count becomes zero, the semaphore is released and the task list is searched for tasks, which are waiting for this very semaphore. If the semaphore was not owned by the active task, the **not_my_semaphore** error message will be produced.

`wait ( sema_addr -- )` W, task_lib.fs

wait for the semaphore's count field to become greater than zero. This way, a task may wait for an interrupt or another task to execute `signal` (see below) on the same semaphore. When the count field is greater than zero, the count field is decremented by one and execution continues.

`signal ( sema_addr -- )` W, task_lib.fs

increments the semaphore's count field by one. It wakes the task that may be waiting for the semaphore. `Signal` can be used in an interrupt service routine to wake a task that executed `wait` on the same semaphore.

`.semas ( -- )` T, task_lib.fs

can only be executed interactively via the umbilial link. It displays the current state of all semaphores on the host.

8.4 Catch and Throw

`Catch` and `throw` behave according to the Forth-2012 standard. The "exception frame" is placed on the return stack. They will only be compiled when `CATCH/THROW` has been set to True in **task_lib.fs**.

9 Cross Compiler

9.1 Executables

These words can only be executed on the target or on the host. Some words are immediate and therefore, they will also be executed while in compilation mode.

((--) T C, microcross.fs

\ (--) T C, microcross.fs

\\ (--) T, microcross.fs
Skip the remaining source code until the end of the file.

(* (--) T, extensions.fs
Multiline comment. Skip the source code until a matching `*)` or the end of the file.

H (--) T C, microcross.fs
sets context to Forth.

T (--) T C, microcross.fs
sets context to Target.

2 (u -- 2**u) W, forth_lib.fs**

2// (u -- log2_u) T, forth_lib.fs

include (<file> --) T, microcross.fs

library (<file_lib> --) T, library.fs
is similar to `include` but it operates on files that have been prepared as libraries using `~`. As a convention, library file names have **_lib** appended at end as e.g. in **forth_lib.fs**.
When a library file is loaded using `library`, each word definition produces a word entry in the dictionary that points to the file location just after the `~` that started a library group. But no code will be produced in the target system.
When one of those words is used later on - either in a word definition or in a command line while debugging - the entire library group will be loaded producing code in the target system.

code-origin (addr --) T, microcross.fs
Set the target's program memory pointer to `addr`.

code@ (-- addr) T, microcross.fs
Retrieve the current value of the target's program memory pointer.

data-origin (addr --) T, microcross.fs
Set the data memory pointer to `addr`.

here (-- addr) T, messages.fs | D, debugger.fs
can be compiled into `:`-definitions on the target while debugging and returns the current value of the target's data memory pointer.

noexit (--) T, microcross.fs
removes the latest `EXIT` compiled by `;`.

immediate (--) T, microcross.fs

[IF] (flag --) T C, microcross.fs

[NOTIF] (flag --) T C, microcross.fs

[ELSE] (--) T C, microcross.fs

[THEN] (--) T C, microcross.fs

.((--) T, microcross.fs

' (<name> -- addr) T, microcross.fs

h' (<name> -- xt) T, microcross.fs
Even if in the target context, returns execution token `xt` of a word in the host system.

t' (<name> -- xt) T, microcross.fs
Even if in the host context, returns the execution token of a word in the target system.

char (<c> -- n) T, microcross.fs
leaves the numeric value of ASCII character `c` on the stack.

macros (--) T, microcross.fs
displays the names of all macros.

new (--) T, microcross.fs
Initializes the cross compiler. See one of the `load_xxxx.fs` files for its use.

end (--) T, microcross.fs
Completes target compilation of an application. Code for data memory initialization is generated starting at `Label initialization` at the end of the application code.

Host (--) T, microcross.fs
Sets the host context.

Target (--) H, microcross.fs
Sets the target context.

9.2 Compilers

[(--) C, microcross.fs

] (--) T, microcross.fs

lit, (n --) H, microcross.fs
Compile the signed number `n` as a sequence of literal instructions. Often used to define macros.

Literal (n --) C, microcross.fs

[char] (<c> --) C, microcross.fs
compiles the numeric value of ASCII character `c` as a literal value in the target code.

['] (<name> --) C, microcross.fs

[t'] (<name> -- xt) H, microcross.fs
Compiles the code address of a target word into a cross compiler directive. See: `Host:`

recurse (--) C, microcross.fs

9.3 Defining Words

: (<name> --) T, microcross.fs

; (--) C, microcross.fs

Host: (<name> --) T, microcross.fs

Creates compiler word for the target context, which will be executed on the host. Use `T` and `H` to select the context for words to be compiled. `Host:` itself up to the closing `;` does not produce any code in the target system. Example:

```
Host: DO ( n1 n2 -- ) ( R: -- n3 n4 ) ?comp T (do >r FOR H ;
```

Macro: (<name> --) T, microcross.fs

Creates a macro definition on the host that can be used in the target context. The macro definition itself produces no code in the target system. It produces code when executed in the target context. Example:

```
Macro: time ( -- time ) T Time-reg @ H ;
```

Init: (<name> --) T, microcross.fs

Creates a colon definition that will be automatically included in the initialization code. Several Init: definitions may be made and they will be executed in the order in which they have been loaded.

Constant (<name> n --) T, microcross.fs

2constant (<name> d --) T, microcross.fs

Create (<name> --) T, microcross.fs

Variable (<name> --) T, microcross.fs

Version (<name> flag --) T, microcross.fs

creates an immediate constant, which returns a flag later on to be used as a compiler directive ahead of `[IF]`.

Register (<name> addr --) T, microcross.fs

This is a constant used for register addresses. Register definitions get included in the variables list for debugging purposes.

Bit (<name> u --) T, microcross.fs

Creates constant `<name>` with a value of `2**u`. It is used to create named bitmasks (see **constants.fs**).

Message# (n <name> --) H, microcross.fs

Creates message number constants both in the host and in the target system.
Negative message numbers are errors, message numbers >= $4000 are warnings, message numbers below $4000 are used by the target to trigger specific host support functions. E.g. `#dot` ($101) is used to display the top item of the target's data stack as a signed number on the host's display.

9.4 Create New Defining Words

Create (<name> --) T C, microcross.fs

Does> **(--) C, microcross.fs**

allot **(n --) T, messages.fs | D, debugger.fs**
can be compiled into :-definitions on the target while debugging.

, **(n --) T, microcross.fs**

@ **(addr -- n) T, microcross.fs**

2@ **(addr -- d) T, forth_lib.fs**

! **(n addr --) T, microcross.fs**

2! **(d addr --) T, forth_lib.fs**

9.5 Object Code Output Files

Bin-file (<name> --) H, images.fs
Writes a binary image of the compiled target code to file <name>.

Boot-file (<name> --) H, images.fs
Produces VHDL ROM source file <name> when e.g. loading bootload.fs. The latter produces bootload.vhd as μCore's boot loader that is synthesized statically into the design. At the end, a branch to address 0 (the #reset trap) will finish boot loading and start execution of the code that has been transferred into the program memory by the boot loader from a nonvolatile memory of the target system.

CRC-file (<name> --) H, images.fs
Produces file <name> that can be stored in a nonvolatile memory to be transferred into program memory by the boot loader. It consists of a 16 bit length field, the binary image of the compiled target code, and a 16 bit CRC-CCITT.

MEM-file (<name> --) H, images.fs
Produces target's program memory image file <name>, which can be used during VHDL simulation to initialize program memory (see the entity definitions for internal RAMs in **functions.vhd**). It can also be used to initialize e.g. Xilinx blockRAMs in the FPGA configuration file.

VHDL-file (<name> --) H, images.fs
Produces a VHDL ROM source file <name> of the compiled target code. It can be used to predefine the program memory during simulation.

9.6 Libraries

Three libraries have been prepared for μForth_2200, namely: **forth_lib.fs**, **float_lib.fs**, and **task_lib.fs**. When these files are loaded using library instead of include, no code will be produced in the target system at first but only word definitions pointing to the source code file and location on the host system instead.

Later, when one of those words is used inside a target word definition, the current definition will be abandoned, the library word(s) will be loaded instead, and thereafter, compilation of the abandoned definition resumes. This is a recursive process. When a library word needs to compile another library word, the current library definition will be abandoned and the missing word loaded.

Eventually, all library words that are needed by an application will have been loaded. Therefore, optimal code without any dead code will always be produced in a single compilation run.

Please note that library files can be loaded using `include` as well producing target code for all word definitions in the file as usual.

`~` **`( -- ) T,` library.fs "tilde"**

`~` marks the beginning of a library group. It ends when either another `~` is found or the end of file is reached. Whenever a library file is loaded using `include`, `~` will be treated as a noop. Look at e.g. **forth_lib.fs** for its use.

`library` **`( <file> -- ) T,` library.fs**

is similar to `include` but it operates on files that have been prepared as libraries using `~`. As a convention, library file names have **_lib** appended at end as e.g. in **forth_lib.fs**.
When a library file is loaded using `library`, each word definition produces a word entry in the dictionary that points to the file location just after the `~` that started a library group. But no code will be produced in the target system.
When one of those words is used later on - either in a word definition or in a command line while debugging - the entire library group will be loaded producing code in the target system.

9.7 Optimizations

The cross compiler performs peephole optimizations to fully utilize the capabilities of the μCore instruction set.

1. Tail-call: When an `EXIT` follows a `JSR`, the `JSR` will be replaced by `BRANCH` and the `EXIT` will be omitted.
2. Arithmetic: `OVER OVER <binop>` or `2DUP <binop>` will be optimized, as well as `SWAP -`.
3. Stack operations:

syntax	compiled code
`swap over`	`tuck`
`over swap`	`under`
`swap drop`	`nip`
`swap swap`	`<nothing>`
`r> >r`	`<nothing>` This is an optimizer for `FOR DO ?FOR ?DO`

`|` **`( -- ) C, microcross.fs`**

Break optimization. Resets the peephole optimizer so that no optimizations will be done to the left of `|`.

10 Debugger

μCore uses an FPGA's blockRAMs as program memory. A 2-wire RS232 umbilical link is used to upload programs and interactively control the processor. The debugger works in such a way that μCore is halted using the CLK_EN signal while the content of the program memory is altered. Afterwards, CLK_EN is asserted again and μCore continues execution where it was halted without loss of state. Therefore, breakpoints can be shifted through the executable code under control of the debugger on the host in order to realize a single-stepping debugger.

Debugger words can only be executed interactively when the target system is connected to the host via the umbilical. Then the debugger command wordlist will always be searched before any other wordlist. Therefore, when a `:`-definition is made in the target system that has a name identical to a debugger command, it can not be executed interactively, because the debugger word will always be found first.

In the debugger, `:`-definitions can be made interactively. They will be compiled and transferred into the target's program memory by `;` and therefore, they can immediately be executed. This is a handy tool for short debug loops when the target system's peripherals are explored.

10.1 Configuring the RS232 interface

In the software directory, different versions for the RS232 host interface are available for Windows, Linux and MacOS. The actual code needed and the host's port name are defined in **umbilical.fs**. The target system's baudrate is defined by VHDL constant `umbilical_rate` in **architecture_pkg.vhd**.

`Umbilical:` **`( <name> u -- ) H`, rs232_xxx.fs**

`<name>` is an operating system dependent characterstring for the device to use as umbilical link and `u` is its baudrate. In a Windows environment, `<name>` may be `COM3`, in Linux, it may be `/dev/ttyUSB0`.

10.2 Entering and Exiting the Debugger

`boot-image ( -- ) H`, debugger.fs

transfers the compiled object code into the target's program memory and resets μCore.

`handshake ( -- ) H`, debugger.fs

synchronise host and target system by exchanging magical numbers over the umbilical link.

`dbg ( -- ) H`, debugger.fs

reenters the debugger and the target system.

`run ( -- ) H`, debugger.fs

transfers the cross compiled target's program memory image into μCore's program memory, resets μCore and enters the debugger on the target system.

```
: run ( -- ) boot-image handshake dbg ;
```

`connect ( -- ) H`, debugger.fs

wakes the `Terminal` task and enters the debugger on the target system. This command must be used to connect a running target system with a host that has compiled code that is identical to the running system for inspection and debugging.

bye **(--) T**, **debugger.fs**
leaves the debugger on the target system returning to the host system's Forth.

break **(--) T**, **debugger.fs**
leaves the debugger on the target system returning to the host system's Forth. In addition the `Terminal` task is put to sleep. The `break` command will be automatically executed when a break condition is detected on the umbilical RS232 interface. Use `connect` to reconnect to the target.

debugger **(--) W**, **debugger.fs**
enters the single step debugger without modifying μCore's state. This is usually what the `#break` trap executes.

debugService **(--) W**, **debugger.fs**
Usually executed at the end of the boot routine. Both the data- and the return-stacks are emptied. Then the host and target systems exchange magical numbers across the umbilical. In the end, both host and target are synchronized and the target waits to receive an address to call. After execution has finished, 0 is returned to the host if everything was ok. Positive numbers are warnings, negative numbers are errors. Warnings and errors may display descriptive text. See **messages.fs**.

10.3 Displaying Information

binary **(--) T, microcross.fs**
set the number base to binary. Numbers preceded by **%** are always interpreted as binary numbers independent of the number base, e.g. %11010010.

decimal **(--) T, microcross.fs**
set the number base to decimal. Numbers preceded by **&** are always interpreted as decimal numbers independent of the number base, e.g. &1205.

hex **(--) T, microcross.fs**
sets the number base to hexadecimal. Numbers preceded by **$** are always interpreted as hexadecimal numbers independent of the number base, e.g. $3AD2.

. **(n --) T**, **debugger.fs**
prints the top item of the target's stack as a signed number. `.` can be compiled into `:`-definitions while debugging.

.r **(n u --) T**, **debugger.fs**
displays `n` on the target's stack as a right justified, signed number, `u` characters wide. `.r` can be compiled into `:`-definitions while debugging.

u. **(u --) T**, **debugger.fs**
displays `u` as an unsigned number. `u.` can be compiled into `:`-definitions while debugging.

d. **(d --) T**, **debugger.fs**
prints the two top items of the target's stack as a signed double number. `d.` can be compiled into `:`-definitions while debugging.

ud. **(d --) T**, **debugger.fs**
prints the two top items of the target's stack as an unsigned double number. `ud.` can be compiled into `:`-definitions while debugging.

cr **(--) T**, **debugger.fs**
can be compiled into `:`-definitions while debugging.

emit **(c --) T**, **debugger.fs**
can be compiled into `:`-definitions while debugging.

signed **(--) H T**, **microcross.fs**
display the numbers of `.s` and `dump` as singed numbers.

unsigned **(--) H T**, **microcross.fs**
display the numbers of `.s` and `dump` as unsinged numbers.

dump **(addr u --) T**, **debugger.fs**
displays the content of `u` number of data memory cells starting at `addr` as `signed` or `unsigned` numbers (see `signed` and `unsigned` above).

udump **(addr u --) T**, **debugger.fs**
displays the content of `u` number of data memory cells starting at `addr` as unsigned numbers.

.s **(--) T**, **debugger.fs**
non-destructively print all items on the target's stack as signed or unsigned numbers (see `signed` and `unsigned` above).

disasm **(addr --) H T**, **debugger.fs**
disassembles instructions starting at program memory address `addr`. The command will be finished by <cr>, any other key will disassemble the next instruction.

show **(<name> --) H T**, **debugger.fs**
disassembles instructions starting at `:`-definition `<name>`. The command will be finished by <cr>, any other key will disassemble the next instruction.

Umbilical (-- addr) H, **debugger.fs**
a variable in the host system, which may be true (Umbilical on) or false (Umbilical off). When on, the serial data communication across the umbilical link will be displayed.

10.4 Debugging Commands

When communicating with the target system via the umbilical interactively, a number of commands are available, which are always looked up first in the dictionary.

? **(--) T**, **debugger.fs**
displays the debugger commands. Aliases are: `commands` and `help`.

' **(<name> -- xt) T**, **debugger.fs**
returns `<name>`'s execution token.

breakpoints (--) T, **debugger.fs**
displays all currently set breakpoints.

watch (<name> --) T, **debugger.fs**
activates the watch function on variable or memory mapped register <name>. After executing an input command line, the content of <name> will be displayed.

#watch (n <name> --) T, **debugger.fs**
similar to watch. n consecutive data memory cells will be displayed starting at the address that <name> points to.

unwatch (<name> --) T, **debugger.fs**
removes <name> from the watch list.

unwatch-all (--) T, **debugger.fs**
empties the watch list.

download (addr u <file> --) H T, **debugger.fs**
transfer u number of data memory cells starting at addr to <file> on the host. The content of each cell is stored as a hexadecimal ASCII string followed by <cr>. The download command will only access the target's data memory when it is not used by the running system.

upload (addr <file> --) H T, **debugger.fs**
transfer <file> on the host to the target's data memory starting at addr. The upload command will only access the target's data memory when it is not used by the running system.

times (n --) T, **debugger.fs**
reinterprets the current input line n times. If n is zero, the input line will be reinterpreted until hitting <cr>.
E.g. cr 1 . 3 times will produce the following output:

```
1
1
1 ok
```

10.5 Single Stepping

The single step debugger works by inserting the break instruction into the program memory at appropriate places. When the break instruction is executed, execution of the program is halted, the next instruction is disassembled, the content of the data stack is displayed, and the system waits for keyboard input.

Hitting <cr> on an empty input line will push the breakpoint one instruction forward and execute the instruction that had been disassembled. Instead of hitting <cr>, the data stack may be modified, any :-definition may be executed, or a single stepping instruction may be executed to modify the single stepping flow.

debug (<name> --) T, **debugger.fs**
inserts a breakpoint at the beginning of :-definition <name>. When <name> will later be executed as part of a test program, the single step debugger will be invoked. Debug lays down a permanent breakpoint, which will be in effect until removed by unbug (see below).

unbug (<name> --) T, **debugger.fs**
removes a breakpoint at :-definition <name>.

unbug-all (--) T, **debugger.fs**
removes all breakpoints.

`trace` **`( <name> -- ) T`**, **debugger.fs**
inserts a breakpoint into `<name>` and executes it immediately. This is a transient breakpoint, which will only be activated once.

`end-trace ( -- ) T`, **debugger.fs**
terminates single step debugging and executes the remainder of the current program without further interruption.

`abort` **`( -- ) T`**, **debugger.fs**
immediately finish single step debugging without executing the remainder of the current program.

`breakpoint ( addr -- ) T`, **debugger.fs**
install a breakpoint at `addr`.

`after` **`( -- ) T`**, **debugger.fs**
used when the next instruction is a branch backwards that may have been compiled by `UNTIL, REPEAT, LOOP, NEXT,` or `GOTO`. `after` places the next breakpoint after the branch instruction and therefore, the remainder of the loop will be executed without interruption until the loop finishes.

`jump` **`( -- ) T`**, **debugger.fs**
jump over the current instruction that has been displayed. This in combination with appropriate data stack manipulation may be used to navigate around a program bug while single stepping.

`nest` **`( -- ) T`**, **debugger.fs**
used when the next instruction will be a call. Continues single step debugging into the called program.

`unnest` **`( -- ) T`**, **debugger.fs**
terminates single stepping on the current call level executing the current program level uninterrupted until after the next EXIT instruction.

11 microForth Coding Standard

The purpose of the coding standard is to aid in creating reliable code. To this end it helps when the code is written in such a way that it is easy to read and understand by the initial programmer himself, his colleagues, and by system engineers as well.

11.1 Readability

Readability not only deals with the layout of the source code on the display. The code's structure should be easy to recognize. Identifier names should be descriptive w.r.t. their functionality and easy to remember.

Place all (sub)words, which are needed to define a word of the next higher level as close together as possible. Ideally, one higher level word and its subwords can be seen on one editor screen whithout having to scroll. Because scrolling diverts the thought process.

Forth is unconventional w.r.t. its input stream parser. Forth's parser works like a human reader: Syntactic elements are separated by whitespace. This makes Forth easily extensible.

11.1.1 Character Case

Forth is NOT case sensitive. ThisWord and thisword are the same to the Forth interpreter.

All words, which modify program flow (e.g. `IF ... THEN` or `EXIT`) are written in upper case.

All defining words, and words which leave an address on the stack (e.g. words defined by `Variable`) are nouns and therefore, their first character is written upper case German style.

11.1.2 Special Characters

Only whitespace (space, tab, cr, lf) are reserved characters. Hence, all special characters (and most ASCII control characters) may be used for word names. This opens up a whole new dimension of descriptive possibilities compared to other languages. Especially the parenthesis (and) are noteworthy: Because Forth uses postfix notation, explicitly operating on the stack, the parenthesis don't have to be used to overwrite operator precedence rules. Instead, they serve the same purpose as in human reading: They enclose comments.

In µForth, these special characters are used (`ccc` stands for 'any character string', `num` for 'any numerical string' according to Base):

Character	**Semantic**
`ccc!`	a word, which stores into memory
`ccc"`	a word that compiles a text string up to the closing "
`#ccc`	a constant or a bitmask, e.g. `#signbit`, `#c`
`$num`	hexadecimal number, irrespective of `Base`
`%num`	binary number, irrespective of `Base`
`&num`	decimal number, irrespective of `Base`
`ccc'`	a word, which consumes a name and returns its execution token
`ccc,`	a word, which compiles something into memory advancing the memory pointer
`.ccc`	words, which display something, e.g. `.version`.

`Ccc:`	words that begin a `:`-definition of some sort, e.g. `Macro:` or `Host:`
`<ccc`	move something "backwart", e.g. `<mark`
`>ccc`	move something to or "forward", e.g. `>r`, `>mark`
`ccc>`	move something from, e.g. `r>`
`ccc?`	a word, which leaves a flag on the stack
`?ccc`	a word which tests a condition.
`ccc@`	a word, which fetches from memory

11.1.3 Compound Names

Traditionally, compound names use either capital letters or the `_` character to separate the constituents of a compound name. The preferred separator in Forth is the `-` character (e.g. `trap-addr`), which can be used for this purpose, because Forth's syntactical separator is whitespace only.

In μForth, `-` is used for Forth words, `_` is used for names that have been imported from μCore's VHDL code.

11.1.4 Spacing

Forth words (procedures) do not need parameter lists when they are called, because the stack is used for parameter passing. A word expects elements on the stack of an expected type and in the expected position (stack allocation). Therefore, Forth tends to be written more horizontally than most other programming languages. This has the benefit that more code fits on a screen and therefore, more code can be seen at once without scrolling.

Phrases are sequences of words, which produce an intermediate result. Two phrases are separated by three spaces or a <cr>. Words within a phrase are always separated by only one single space.

11.1.5 Indentation

Each nesting level should be indented at least by an additional three spaces from the previous one. NEVER use tabs instead of spaces, because different editors will interpret the tabs differently and therefore, your text layout may look quite different.

If your nesting level reaches three you should consider to factor out the inner nesting level as a separate word, which just does get called.

When computing the flag value for an `IF`, `WHILE`, or `UNTIL`, it may precede the `IF` on the same line if it is not longer than three words. Otherwise, place the `IF` on the next line.

Example from **forth_lib.fs**:

```
: move    ( addr-from addr-to ucount -- )
   >r 2dup u< IF ( cmove> )
      r@ + swap r@ +  r> ?FOR  1- ld -rot  swap 1- st  swap  NEXT
   ELSE          ( cmove )
       1 - swap  1 -  r> ?FOR  1+ ld -rot  swap 1+ st  swap  NEXT
   THEN  2drop
;
```

11.1.6 Literals

It is good practice to write down literals with a leading base indicator independent from the system variable Base. Use capital letters A - F for hexdecimal numbers. Examples:

```
%10001101   -%111
&29         -&29
$43FC       -$43CA
```

For the sake of information hiding, never use numerical literals in colon definitions. Instead, define a constant that can bear semantic information in its name as an added benefit.

11.2 Understandability

Write the code in such a way that you do not need to twist your brain unnecessarily in order to understand it.

11.2.1 Factoring

Understandability deals with the program logic itself and how it is being factored, i.e. a word on a higher level should be broken down into a sequence of sub-words on a lower level, which together read as natural as possible. If done well, the sequence of word-names, which are executed sequentially, already shed light on the functionality that it performs as self documenting code.

Getting the factoring "right" has a major impact on understandability.

As a rule: When a word needs more than three input arguments, you should re-think your factoring.

If a word gets longer than five lines, you should probably break it up into sub-words.

If possible, place a word and its sub-words closely together. Ideally, they are placed on one screen so they can all be seen without having to scroll.

11.2.2 Using EXIT

`IF ... ELSE ... THEN` structures, especially nested ones, tend to be difficult to think through. Very often, a conditional clause can leave the word using `EXIT` before the words final `;`. Example:

```
: shift ( n1 quan -- n2 )
   dup 0< IF  abs rshift  EXIT THEN  lshift
;
```

When `EXIT` precedes `THEN`, processing of this very condition is finished and you don't have to think about it any longer.

Of course, you have to make sure that the number of output arguments are the same for every `EXIT` and the final `;` with the exception of the few words like `?dup`, which have different numbers of input or output arguments depending on a condition. These types of words tend to be difficult to understand when they are used!

11.2.3 Comments

Don't comment the obvious!

Use comments to the right of a code line if its functionality is obscure. When the stack juggling becomes challenging, make a list of arguments, which are on the stack after the last word on the line has been executed. You will appreciate this when you have to modify the code at a later time. An example from **task_lib.fs**:

```
: do-wake ( lfa -- )
   ['] go-next swap 1- st >r          \ mark task as not ready to run
   TCB   dup r@ -                     \ is the new task different from
                                      \ the running task?
   IF  Rsp @ Dsp @ rot #t-dsp + !     \ save RSP on stack and DSP in TCB
       r@ Tptr !                      \ switch task
       #t-dsp r@ + @ >dstack          \ restore dsp, fill TOS, NOS
       >rstack EXIT                   \ restore rsp, fill TOR
   THEN
   rdrop drop                         \ if myself, nothing else to do
;
```

Precede a functional block, i.e. a word and its sub-words, with an explanatory block. Example:

```
\ ------------------------------------------------------------------
\ peep-hole optimizer
\ The following words are used throughout the remaining code for
\ peep-hole optimizations.
\ ------------------------------------------------------------------
```

12 Index

-	13	/	13	>=	16	binary	37
'	32, 38	/=	16	>float	17	Bin-file	34
!	7, 34	/mod	13	>host	23	Bit	33
"	31	:	33	>ieee	17	Boot-file	34
#break	11	;	33	>r	5	boot-image	36
#c	21	?	38	0<	12	bounds	10
#data!	11	?DO	10	0<>	12	branch	9
#does>	11	?dup	5	0=	12	break	37
#ie	21	?EXIT	10	0=branch	9	breakpoint	40
#iis	21	?FOR	10	1-	12	breakpoints	38
#isr	11	?GOTO	11	1/f	18	bye	37
#lit	21	@	6, 34	1+	12	c2*	15
#neg	21	[	32	2!	7, 34	c2/	15
#ovfl	21	[']	32	2*	14	CALL	11
#psr	11	[char]	32	2**	14, 31	cancel	29
#reset	11	[ELSE]	32	2/	14	carry?	22
#sign_den	21	[IF]	31	2//	31	case?	16
#sign_div	21	[NOTIF]	32	2@	7, 34	Catch	30
#unfl	21	[t']	32	2constant	33	char	32
#watch	39	[THEN]	32	2drop	5	clear	5
#zero	21	\	31	2dup	5	code@	31
(	31	]	32	2over	5	code-origin	31
(*	31	|	35	2swap	5	connect	36
*	13	~	35	abort	40	Constant	33
*.	18	+	13	abs	12	continue	24
*/	13	+!	7	activate	28	cr	38
*/mod	13	+c	13	ADDR	11	CRC-file	34
,	34	+sat	13	after	40	Create	33
.	37	+st	7	ahead	24	ctrl	23
.(	32	<	16	allot	34	-ctrl	24
.r	37	<=	16	and	13	ctrl?	24
.s	38	=	16	ashift	14	d-	13
.version	23	>	16	BEGIN	10	d.	37

d+	13	elapsed	24	fpi/2	19	kilo	19
d=	16	ELSE	9	fscale	19	l!	7
d0=	12	emit	38	fsin	19	l@	7
d2*	15	end	32	go_next	25	Label	10
d2/	15	end-trace	40	GOTO	11	ld	6
dabs	12	erase	7	H	31	library	31, 35
dashift	16	execute	9	h'	32	lit,	32
data-origin	31	exit	9	halt	28	Literal	32
dbg	36	extend	12	handshake	36	lld	7
deactivate	28	f-	18	here	31	local	5
debug	39	f*	18	hex	37	lock	30
debugger	37	f/	18	Host	32	log2	18
debugService	37	f+	18	Host:	33	LOOP	10
dec	7	f<	18	host>	23	lst	7
decimal	37	f<=	18	I	10	m*	14
degree	19	f>	18	ieee>	17	m/mod	14
depth	5	f>=	18	IF	9	m+	14
di	22	f0=	18	immediate	31	Macro:	33
disasm	38	f2*	18	inc	7	macros	32
dnegate	12	f2/	18	include	31	mashift	15
DO	10	fabs	18	init	28	max	16
do_poll	25	false	12	Init:	33	mega	19
do_poll_tmax	25	fcos	19	initialization	8	MEM-file	34
do_priority	25	fexp	19	initialized	8	message	28
do_robin	25	fexp2	19	init-performance	27	Message#	33
do_time	25	fill	7	int.frac	18	micro	19
do_wake	25	flag?	23	int-disable	22	milli	19
Does>	34	flags	23	integer	17	min	16
download	39	fln	19	int-enable	22	mod	13
drop	5	float	17	intflags	22	move	7
dshift	16	float>	17	invert	12	ms	24
dump	38	flog2	19	iret	9	mshift	15
dup	5	fnegate	18	jsr	9	negate	12
ei	22	FOR	10	jump	40	nest	40

new	32	REPEAT	10	Taskvariable	27	Umbilical	38
NEXT	10	rot	5	Terminal	28	Umbilical:	36
nip	5	-rot	5	THEN	10	umin	16
noexit	31	rotate	15	throw	30	umod	13
noop	12	RoundRobin	28	time	24	unbug	39
normalize	17	run	36	time?	24	unbug-all	39
not	12	schedule	29	times	39	uninitialized	8
nz-exit	9	sec	24	tor-branch	9	unlock	30
off	7	Semaphore	30	trace	40	unnest	40
on	7	semas	30	TRAP:	11	unpack	14
or	13	shift	14	trap-addr	11	unsigned	38
over	5	show	38	true	12	UNTIL	10
ovfl?	22	signal	30	u.	37	unwatch	39
p!	8	signed	38	u/	13	unwatch-all	39
p@	8	sleep	24	u/mod	13	upload	39
pack	14	spawn	29	u+sat	13	Variable	33
pass	23	sqrt	12	u<	16	Version	33
pause	28	st	6	u>	16	VHDL-file	34
poll	29	stop	28	u2/	14	wait	30
poll-tmax	29	st-reset	21	ud*	14	wake	28
Priority	27	st-set	21	ud.	38	watch	39
r@	5	swap	5	ud/mod	14	WHILE	10
r>	5	T	31	ud2/	15	within	16
rclear	5	t'	32	udump	38	xor	13
rdrop	5	Target	32	um*	14		
Register	33	Task	27	um/mod	14		
release	23	tasks	29	umax	16		