

This manual is written with the assumption, that the reader has coarse knowledge of Forth programming. No general instructions how to operate with Forth are provided.

The intention is to make a straightforward user oriented Forth system, specifically to organize electronic control of art objects and installations or similar process control.

No attempt is made to comply with established Forth standards.

No attempt is made to "program Forth by Forth". At the expensive of code length everything is programmed on rather flat subroutine nesting.

Where standard Forth systems compile sniblets of assembler code to manage runtime behaviour I have introduced several specific "doAll" runtime operators for loops, conditionals, variables and constants. So I think, I have got an acceptably fast and easy to understand and to manage Forth system.

Instead of Forth "Words" I use the termini "**Kernel Operator**" and "**User Thread**".

Data Stack, Return Stack and Thread Code are organized each as array of 32bit words

Some CForth internals:

Access to Kernel Operators is organized in three synchronous arrays, which gives a small advantage compared with organisation in a structure of arrays.

One array contains the name words,

another one contains pointers to their code subroutines,

a third one contains their **attributes**:

COMPILEONLY can only be compiled in compiling mode

IMMEDIATE cannot be compiled, only be executed immediately even in compiling mode.

RUNTIME is a runtime operator which is compiled by another operator and executed at runtime. Cannot be handled by the user directly. RUNTIME includes COMPILEONLY

COMPILEONLY and IMMEDIATE are combined in some cases.

Special is the BACKGROUND attribute, which allows selected Operators (see

Glossary) to be started in a simple background process while a User Thread is executed.

Good for manipulation of CONSTants and VARIables, for single step operation and break of endless loops. BACKGROUND and IMMEDIATE are combined in some cases.

Attribute violation by the user causes an error message.

Access to User Threads is organized as an array of structures with following entries:

1. "name", name word, max 15 chars plus null termination
2. "toCompile" contains the absolute start address of the compiled thread in the Thread Code array or actual data value of CONST and VAR.
3. "attributes" contains CONST or VAR attribute plus the index of the compiled thread in the Thread Code array. Double storage of thread address and thread index has some advantages for token search.

To keep it simple, User Threads can never be made COMPILEONLY, IMMEDIATE, RUNTIME or BACKGROUND.

The **Thread Code is organized** as follows:

The complete code is stored in a single array of 32 bit words. Each "User Thread" occupies a defined part within this array.

Beginning from the thread start index within this array (index stored in User Thread structure "attributes") during compilation "token" are added in linear sequence. Where necessary a token is followed by one or more additional words containing a literal, an address to be put on TOS or a string to be sent at runtime.

During runtime, "token" are read, interpreted and executed. Each time, when the read "token" points to a User Thread, the instruction pointer jumps to that start index and when the ';' token is reached, back to the original (index+1).

At any time, **execution can be switched to single step operation** with operator "ST" and return to normal execution is made with Operator "RUN". Both can be handled from the background process.

About "token": each Kernel Operator is compiled there with its index in the array of code pointers (max 12 bit). Each User Thread is compiled there with its absolute start address in the Thread Code array (i.e. SRAM address > 0x20000000, so easily distinguished from Kernel Operators)
It would have been possible to compile the start address of Kernel Operator subroutines directly, which would speed up the runtime a little bit. But I didn't because the actual solution makes the compiled code better user readable (Kernel Operator "MEM") and compiled projects (when stored in Flash) are quite stable against moderate changes of C-source code.

All **operations are performed** as 32bit signed integer.

No warning or error message, when result is out of range.

Two Exceptions (not implemented for STM32F042):

--- * (multiplication): First factors are converted into float and checked if 32bit int overflow will happen. In this case, execution is terminated with an error message.

Else multiplication is performed 32bit signed int. (STM32F042 no float support, no error msg!)

--- */ (combined multiplication/division): First factors are converted into float and checked if multiplication will exceed 32bit int range. If not, the operation is performed with 32 bit integer. If overflow would happen, the operation is performed in "float". If this result fits into 32bit int, the result is reconverted into 32 bit signed integer (may cause small rounding error). **(STM32F042 no float support, performed int only, no error msg, risk of overflow!)**

Else operation is terminated with an error message.

In accordance with the intended range of use (simple process control) introduction of unsigned operations seems not necessary. Though it would be rather easy to implement a set of float operations due to the FPU of L4 and G4 processors, this seems not to make sense for the intended range of use.

All text input is case insensitive (internally converted to upper case) except user strings to be sent via terminal or serial I/O.

CForth "Projects":

In accordance with the memory design of the STM32G431KB processors, up to 6 "**Projects**" are supported. The versions for STM32L476 have the same memory configuration. Due to

restricted memory, the STM32F042 supports only one project (one in Flash, experimental work in SRAM is possible independently). Actual CForth operation (interactive and compilation) is always done in SRAM. No "compilation into Flash" is supported.

By default (if nothing else is loaded from Flash), **Project #0, an "Empty Project"** with default parameters is loaded into SRAM. This installs two **CONSTants** P0, P1 and two **VARIABLES** V2, V3 **as user token**. Sometimes helpful for interactive process control. **FORGET** them if not needed. User Threads can be compiled and executed here, but are lost when a power cycle or reset is performed.

Projects #1 to 5 (with DMX only 1 to 3) can be loaded from Flash to SRAM (operator "LOAD-F"). Existing code in SRAM is overwritten then. If the corresponding Flash area is "virgin", an empty project is loaded automatically. "Project#0" loads an empty project (=default settings)

Any active project in SRAM can be burnt into Flash (operator "SAVE-F") as Project #1 to 5 (with DMX only 1 to 3), existing Flash area is overwritten then. Project#0 can be saved too, but then only global system parameters are stored, next CForth start loads an empty project#0.

Together with the project specific code, some global system parameters (as active project number, USB Vid/Pid, baud rate) are stored in a separate Flash area, which is shared by all Projects. Particularly the active project number is stored there. So after power cycle or reset, the last saved project is automatically reloaded from Flash into SRAM (or an empty project).

How to **change the project number** automatically reloaded at CForth start: Simply load the desired project (modify it or not) and restore it into Flash (possibly as another project).

Glossary of Operators:

Operator names written **red**, are **not implemented for STM32F042**

Operator names written **green**, are implemented for STM32F042 / **SPI version**

Operator names written **blue**, are implemented for STM32F042 / **MIDI version**

Arithmetic Kernel Operators:

NOP	(---)	
DROP	(x1 x2 -- X1)	
DUP	(x -- x x)	
2DUP	(x1 x2 -- x1 x2 x1 x2)	
SWAP	(x1 x2 -- x2 x1)	
2SWAP	(x1 x2 x3 x4 -- x3 x4 x2 x1)	
OVER	(x1 x2 -- x1 x2 x1)	
PICK	(x-n ... X-2 x-1 x0 n -- x-n ... X-2 x-1 x0 x-n)	consume n and copy the n-th element of stack to the Top of Stack (TOS) 0 PICK is equivalent to DUP 1 PICK is equivalent to OVER
ROT	(x1 x2 x3 -- x2 x3 x1)	rotate 3rd stack entry up
-ROT	(x1 x2 x3 -- x3 x1 x2)	rotate TOS down
+	(x1 x2 -- x1+x2)	signed 32bit addition
1+	(x -- x+1)	add 1 to TOS
-	(x1 x2 -- x1-x2)	signed 32 bitz subtraction
1-	(x -- x-1)	subtract 1 from TOS
ABS	(x -- abs(x))	
<<	(x1 x2 -- x1<<x2)	shift range X2:0 ...31
>>	(x1 x2 -- x1>>x2)	shift range X2:0 ...31 if X1(bit31) is set, "ONE"s are shifted in ! (preserve minus sign)
&	(x1 x2 -- x1&x2)	bitwise AND replaces standard Forth "AND"
 	(x1 x2 -- x1 x2)	bitwise OR replaces standard Forth "OR"

XOR	(x1 x2 -- x1 x2)	bitwise XOR
NOT	(x ---xmodified) <one's complement of x>	bitwise 0/1 inversion
NEG	(x -- -x) <two's complement of x>	change sign of TOS
*	(factor factor -- factor*factor)	signed 32 bit multiplication Error message if result is greater than 32bit signed integer
/	(dividend divisor ---- dividend/divisor)	signed 32 bit division
*/	(factor factor divisor -- (factor*factor)/divisor)	first multiply factor*factor, then divide the intermediate result by divisor if (factor * factor > 32bit), operation is internally performed in "float" and reconverted to 32bit signed int. Small rounding error possible. Error message if result is greater than 32bit signed integer
MOD	(dividend divisor -- remainder)	The remainder has always the same sign as the dividend
/MOD	(dividend divisor -- remainder quotient)	result is: quotient on TOS, remainder on NOS(=2nd stack element). The remainder has always the same sign as the dividend
BSET	(x bit# -- xmodified)	bit# of x is set to 1 (bit# 0..31)
BCLR	(x bit# -- xmodified)	bit# of x is cleared (set to 0) (bit# 0..31)
BTST	(x bit# -- bitvalue))	bitvalue = boolean value (=0 or 1) of bit#
RANDOM	(-- random)	a 32bit unsigned random number is generated with 1kHz update freq.

Comparison Operators

=	(x1 x2 -- 1 0)	1 if x1 == x2 x1 and x2 are removed from stack
!=	(x1 x2 -- 1 0)	1 if x1 != x2 x1 and x2 are removed from stack
0=	(x -- 1 0)	1 if x == 0 x is removed from stack
>	(x1 x2 -- 1 0)	1 if x1 > x2 x1 and x2 are removed from stack
>=	(x1 x2 -- 1 0)	1 if x1 >= x2 x1 and x2 are removed from stack
<	(x1 x2 -- 1 0)	1 if x1 < x2 x1 and x2 are removed from stack
<=	(x1 x2 -- 1 0)	1 if x1 <= x2 x1 and x2 are removed from stack

Memory Operators

CONST	(x ---) IMMEDIATE	Syntax: 234 CONST CONNY a global CONSTant named CONNY is compiled with value 234. When CONNY is called, the VALUE is put on TOS
VAR	(x ---) IMMEDIATE	Syntax: 789 VAR WILLI a global VARIable named WILLI is compiled with initial value 789. When WILLI is called, the data storage ADDRESS is put on TOS
REV	(x ---) IMMEDIATE BACKGRND	replace value of named CONST (next use and in compiled threads). Syntax: 234 REV CONNY May be applied to VARs too, especially from Background process. To be used for interactive control of thread flow (IF ., Z_BREAK). Useful for experimental parameter variations.
VAL	(-- x) IMMEDIATE BACKGRND	returns the actual value of named CONST or VAR. May be compiled too, puts value of a VARIable directly on TOS then, as known from CONSTants. Syntax: VAL CONNY. Saves minimal runtime. But the biggest advantage is the possibility to check the actual value of a variable from the Background process.
W	(value addr ---)	replaces Standard Forth ! Syntax: 456 WILLI W write value 456 into parameter address of VAR
R	(addr --- value)	replaces Standard Forth @ Syntax: WILLI R read value from parameter address of VAR

Terminal Operators

KEY?	(--- number)	the actual number of bytes in the terminal input buffer, = 0 if none USB has priority over USART
-------------	----------------	---

KEY	(--- byte)	execution blocks until a terminal byte is received, returned on TOS
EMIT	(byte ---)	the lowest 8 bit of TOS are sent via terminal as raw byte
P	(number ---)	TOS is sent as ASCII text using actual SYSTEM NUMBER base The content of TOS is sent as 32 bit signed integer.
PH	(number ---)	TOS is sent as ASCII text HEX formatted The content of TOS is sent as 32 bit unsigned integer.
P" "	(---)	send the text between " " as byte stream via terminal Between P" and the first text letter MUST be a SPACE which is not sent Any "non-printable" byte can be sent if formatted as \hiNibbleLowNibble Examples: \1B sends ESC, \22 sends " \5C sends \ \00 sends nullByte
%D	(---)	set system number base DECIMAL In this case hex numbers can be entered with preceding 0x. Even 0x-... is accepted: 0x-FFFFFFFF = 1
%X	(---)	set number base HEXADECIMAL The number base can be changed by %X,%D token in compile mode, but it is reset to the system number base at compile end (;

Serial I/O Operators

(not supported by unmodified Nucleo boards and the special "CForth DMX Mini" version)

With the SPLIT operator (see below), USB and USART can get different functions.

By default, both work parallel as terminal.

But each of them can be configured as serial data I/O.

RX?	(--- number)	the actual number of bytes in the serial I/O input buffer, = 0 if none. Works only if the serial I/O is COM mode, not in MIDI mode (see SPLIT)
RX	(--- byte)	execution blocks until a serial I/O byte is received, returned on TOS
TX	(byte ---)	the lowest 8 bit of TOS are sent via serial I/O as raw byte
NTX	(b-n ... b-1 b0 n)	the n stack items below n are sent via serial I/O Only the lowest 8bit of each stack item are sent as raw bytes.
TX""	(---)	send the text between " " as byte stream via terminal. Remark see P" above

Time Operators

XNOP	(n ---)	variable short delay, NOP is executed n times
MS	(t ---)	background countdown of t ms is started, see WAIT and BLOCK
TIX	(t ---)	background countdown of t*100ms is started, see WAIT and BLOCK
WAIT	(--- t)	returns remaining countdown of MS or TIX in 100ms units may be used to construct simple timelines
BLOCK	(---)	after MS or TIX execution is blocked until countdown = 0. Can be terminated from background process with CTRL_B, then first send TAB, next CTRL_B
FREQ	(divisor ---) BACKGRND	starts STM32 Timer3 with prescaler input clock frequency 48MHz. "divisor" sets a prescaler to get the input clock frequency of Timer3. With this prescaled input, Timer 3 produces a constant overflow each 2000 prescaled clock pulses. Both together results in the base frequency of PWM1,2, TIME and PULS . E.g: 24 FREQ will cause a 1kHz base freq. The prescaler has a 16bit range, so the base frequency can be modified in the range of ca 24kHz and ca.0.3Hz.
PWM1	(high --) BACKGRND	The PWM1 signal output is available at PA6. If PWM1 is started when FREQ is not active, FREQ gets provisionally activated with 1kHz overflow frequency. The HIGH percentage of PWM1 is (high/2000)*100 0 PWM1 will set PA6 permanently LOW, 2000 PWM1 (or greater) will set PA6 permanently HIGH -1 PWM1 will deactivate the PWM output at PA6, reconfigure PA6 as output with pull-down - but not stop FREQ
PWM2	(high --) BACKGRND	The PWM2 signal output is available at PA7. Else remark see PWM1
TIME	(-- "SysTime")	When FREQ is active, TIME is upcounted each Timer3 overflow. At moderate base frequencies, this can be used as "System Time" base.

PULS	(-- number)	When FREQ is active, PULS is upcounted each Timer3 overflow, but reset to zero after each read access. So PULSE can be used as a non-blocking action trigger with additional information if the readout came slower than one base frequency period. BLOCK and WAIT may be easier to use, but PULS is a low-cost by-feature.
-------------	--------------	---

Peripheral Operators

Each of the following operators configures the pin each time it is called at cost of additional runtime, but handling is quite easy this way.

User is responsible not to connect an output with another output

OH	(portPin ---)	sets addressed port pin as push/pull OUTPUT HIGH Syntax: AF OH sets PA15 B3 OH sets PB3 portPin is entered as HEXbyte. A0 is decimal 160, AF is decimal 175 etc
OL	(portPin ---)	sets addressed port pin as push/pull OUTPUT LOW Syntax: AF OL sets PA15 B3 OL sets PB3
DAO4	(value ---)	configures PA4 as 12 bit DAC right aligned and sets value
DAO5	(value ---)	configures PA5 as 12 bit DAC right aligned and sets value
IPU	(portPin ---)	sets addressed port pin as INPUT with PULL-UP Syntax: AF IPU configures PA15 B3 IPU configures PB3 portPin is entered as HEXbyte. A0 is decimal 160, B0 is decimal 176 etc
IPD	(portPin ---)	sets addressed port pin as INPUT with PULL-DOWN Syntax: AF IPD configures PA15 B3 IPD configures PB3
IZ	(portPin ---)	sets addressed port pin as INPUT with HIGH IMPEDANCE Syntax: AF IZ configures PA15 B3 IZ configures PB3
RDIG	(portPin -- 0 1)	returns the digital level at port pin independent of actual configuration of this pin
RANA	(portPin -- value)	configures the A/D converter and reads 12 bit level Supported pins: PA0...PA7 if not used by hardware else Precision of STM32 A/D converters is bad, so each level is read 8 times and the result is averaged. Once opened the A/D converter stays active during the session, but it is connected to a port only when a read is requested. No matter when the port gets configured else.
INI-SPI	(mode bits clock ---)	initializes an SPI master with bits=4 to 32 bit length Except 16bit, the SPI is implemented very badly by STM So an alternative was hand-coded with up to ca. 2MHz SPI clock, almost double fast with 144MHz system clock, somewhat slower on STM32F042. /CS must be handled separately (if necessary) by user with any pin else. This way, multi-slave configurations are possible. Motorola modes 0,1,2,3 are supported. (sometimes referenced as (0,0) (1,0) (0,1) (1,1)) clock = 0 is max. speed, else clock inserts a heuristic slow down. Default: mode 0, 8 bit, clock 50 (ca 50kHz), max.clock= 0xFFFF=65535 Used I/O pins: STM32G431(Mini)SCK=PB3, MOSI=PA15, MISO=PB4 STM32G431(Vero), STM32L476: SCK=PB3, MOSI=PB5, MISO=PB4 STM32F042: SCK=PA5, MOSI=PA7, MISO=PA6 (changed 11Aug2020)
SPI	(TxWord -- RxWord)	If not initialized explicitly before, SPI gets configured with default or with user values stored in Flash project
INI-I2C	(SlaveAddr --)	Initializes I2C. addr is the slave address which is used in all subsequent I2C transmit and receive operations. Only 7 bit addressing is supported. Standard is 100kHz clock. If bit8 of addr is set (say: 256 added to SlaveAddr), then I2C will work with 400kHz clock. Slave address and bus speed can be changed at any time between two transmit and receive commands. STM32G431(Veroboard and Mini): SDA=PB7, SCL=PB8 Nucleo Boards: SDA=PB9, SCL=PB8 (wiring is easier then)

I-TX	(b1 b2 ... bn n --)	n lower stack bytes are transmitted via I2C in the order b1 first, bn last
I-RX1	(-- byte)	a receive command is sent to the slave, which will return 1 byte. This command is useful for simple I/O expansions like PCF8574 or sequential read from memory chips like 24Cxx .
I-RXA	(RegAddr n – b1 b2 ... bn)	a receive command which sends one device specific register address byte to the slave and orders a return of n bytes. This command is useful for more complex I2C peripherals like PCF8591 or random or page read from serial memory devices.

Following operators manipulate MCU peripheral register directly, good knowledge of the STM32 Reference Manual is necessary.

Don't confuse these operators with W, R, BSET, BCLR, BTST! Both sets use different address spaces and context.

REGWRITE	(peripheral_regr_addr value --)	writes value into the addressed peripheral register
REGREAD	(peripheral_reg_addr -- value)	reads the addressed peripheral MCU register and returns value
REGBSET	(peripheral_reg_addr bitNo --)	SETS bitNo of the addressed peripheral register
REGBCLR	(peripheral_reg_addr bitNo --)	CLEARs(sets =0) bitNo of the the addressed peripheral register
REGBTST	(peripheral_reg_addr bitNo -- bitLevel)	returns level of bitNo of the addressed peripheral reg.

Structuring Operators

IF	(-- addr) IMMEDIATE, COMPILEONLY	compiles doIF. Starts compilation of a IF ... ELSE ... ENDIF conditional Essentially provides a placeholder for jump address and sends its code address via stack to ELSE or ENDIF
ELSE	(addr -- addr) IMMEDIATE, COMPILEONLY	compiles doELSE compiles middle part of a IF ... ELSE ... ENDIF conditional Essentially compiles its code address into the placeholder behind doIF and provides a placeholder for jump address. The stack entry with the code address provided by IF is replaced by a stack entry that contains the code address of doELSE
ENDIF	(addr --) IMMEDIATE, COMPILEONLY	compiles termination of a IF ... ELSE ... ENDIF conditional Essentially it compiles the terminating code address into the placeholder behind doIF (or doELSE if present).

Loops are handled quite different from standard Forth implementations.

Because most technically interested people have some knowledge in C or Java today, I have tried to organize loops more C-style for easier learning and remembering

DO ... WHILE is functionally synonym with standard BEGIN ... UNTIL

DO ... AGAIN is functionally synonym with standard BEGIN ... AGAIN

DO ..Z-BREAK .. AGAIN is functionally synonym with standard BEGIN..WHILE..REPEAT

FOR ... LOOP is functionally synonym with standard DO ... LOOP

The essential task of runtime ops doDO and doFOR is to put the code address for loop BREAK on the Return stack.

The **end of all these kind of loops** is compiled as follows:

Behind the type specific runtime token (doWHILE, doAGAIN, doLOOP), first the loopback address gets compiled (back to behind DO or FOR).

Next the runtime token for BREAK handling is compiled (Return Stack adjustment):

RP-1 (behind WHILE and AGAIN) deletes 1 entry, RP-4 (behind LOOP) deletes 4 entries.

When the loop is terminated "normally", execution passes there too, and the BREAK address (plus 3 loop parameters of FOR...LOOP) is removed from ReturnStack.

The code body of any loop **may contain one or more of these runtime tokens:**

doBREAK terminates the loop unconditionally synonym with standard LEAVE

doZ-BRK checks TOS: if = 0, the loop is BREAKed, else execution continues linear

If doZ-BRK is used several times in a loop, different criteria may be used.

doCONTI jumps unconditionally to the loop end runtime token:

When used in a DO..WHILE loop, **jump is made to directly to doWHILE**

over the loop termination check compiled ahead of doWHILE.

doWHILE always checks TOS: if =0:termination, else loop is continued.

So, this criterion must be placed on TOS before execution of doCONTI.

If doCONTI is used several times in a loop, different criteria may be used.

When used in DO..AGAIN, jump is made to doAGAIN, which recycles to behind doDO

When used in FOR..LOOP, jump is made to doLOOP,

which checks and updates the loop index (2nd Return Stack entry).

If the loop end is not yet reached, code execution is looped back to behind doFOR.

Else execution continues linear behind doLoop via RP-4.

Internally, doCONTI uses the Return stack entry the same way as BREAK,

but simply initiates a jump which is 2 ThreadCode fields shorter.

Operator specific descriptions:

DO	(-- addr) IMMEDIATE, COMPILEONLY	compiles doDO starts compilation of a DO ... WHILE or DO ... AGAIN loop. Essentially it provides a placeholder for the BREAK code address and sends its code address via stack to WHILE or AGAIN
WHILE	(addr --) IMMEDIATE, COMPILEONLY	compiles doWHILE, finishes compilation of a DO ... WHILE loop Essentially it compiles its code address for BREAK behind doDO and compiles the loopback code address provided by DO behind doWHILE.
AGAIN	(addr --) IMMEDIATE, COMPILEONLY	compiles doAGAIN, finishes compilation of a DO ... AGAIN loop Essentially it inserts its code address for BREAK compilation behind doDO and compiles the loopback code address provided by DO behind doAGAIN.
FOR	(-- addr) IMMEDIATE, COMPILEONLY	compiles doFOR, starts compilation of a FOR ... LOOP loop The runtime behaviour is very similar to "C" programming: doFOR expects 3 DataStack entries: startIndex, endIndex, Increment Increment can be positive or negative and can be greater than 1. For runtime details of the FOR...LOOP loop see doFOR and doLOOP. Essentially FOR provides a placeholder for the BREAK code address and sends its code address via stack to LOOP.
LOOP	(addr --) IMMEDIATE, COMPILEONLY	compiles doLOOP, finishes compilation of a FOR ... LOOP loop Essentially it inserts its code address for BREAK compilation behind doFOR and compiles the loopback code address provided by FOR behind doLOOP
I	(-- loopIndex) COMPILEONLY	At runtime, it copies the actual loop index in a FOR...LOOP loop from the second ReturnStack entry to data stack TOS.
CONTI	(---) IMMEDIATE, COMPILEONLY	compiles doCONTI
BREAK	(---) IMMEDIATE, COMPILEONLY	compiles doBREAK
Z-BREAK	(---) IMMEDIATE, COMPILEONLY	compiles doZ-BRK
RETURN	(---) COMPILEONLY	At runtime, it terminates a User Thread immediately (like "C" return)

Runtime Operators

are compiled by other operators and control the runtime behavior.

All runtime operators have the RUNTIME attribute.

No direct user handling or access, but essential for code interpretation with SEE and STEP

doSTRING	(---)	compiled by P" In the compiled thread it is followed by the bytes to be transmitted. Differing from standard Forth, the byte sequence is 0 terminated. Byte 00 is internally coded as 0x100.
doIOSTR	(---)	compiled by TX", remark see doSTRING

doLIT	(-- n)	compiled by numbers. In the compiled thread, doLIT puts the content of the next ThreadCode entry (=number value) on TOS at runtime.
doVALUE (replaces standard Forth doCONST)	(-- value)	compiled by CONST (and by VAL) In the compiled thread, the CONST value address is compiled behind doCONST, but at runtime it puts the CONSTANT value on TOS. Referencing the CONST by address is a contribute to SEE and REV
doADDR (replaces standard Forth doCONST)	(-- address)	compiled by VAR In the compiled thread, the VAR value address is compiled behind doVAR. At runtime, it puts this address on TOS.
doIF	(flag --)	compiled by IF takes a flag from TOS, if == 0, a jump is initiated: behind doELSE if present, or to code address compiled by ENDIF if != 0, execution is performed until doELSE. or continues linear (if no doELSE is present)
doELSE	(---)	compiled by ELSE if the flag of doIF != 0, doELSE manages a jump behind the code of the behind the code of the ELSE part (compiled by ENDIF). if the flag of doIF == 0, execution is continued behind doELSE. At the end of the ELSE part, execution is continued linear.
doDO	(---)	compiled by DO. doDO puts the BREAK code address on ReturnStack
doWHILE	(flag --)	compiled by WHILE, doWHILE takes the flag from TOS. If == 0, execution goes forward via RP-1 for linear progression If != 0, execution loops back to behind doDO
doAGAIN	(---)	compiled by AGAIN, execution loops always back to behind doDO
RP-1	(---)	compiled by WHILE and AGAIN At runtime it removes the top entry from the ReturnStack
doFOR	(startIndex endIndex increment ---)	compiled by FOR 3 items are copied from the DataStack to the ReturnStack: TOS is the increment/decrement of loop index per loop NOS is the loop end index for loop termination 3rd DataStack entry -> 2nd ReturnStack entry: start index=loop index Furthermore the BREAK code address is put on top of the ReturnStack.
doLOOP	(---)	compiled by LOOP counts the 2nd ReturnStack entry up or down (depends on 4th entry), compares the 2nd entry with the 3rd one. If compare is in range, execution loops back to behind doFOR else execution goes forward via RP-4 for linear progression.
RP-4	(---)	compiled by LOOP. It removes the 4 top entries from the Return Stack
doCONTI	(---)	compiled by CONTI. Unconditional jump to (value from Return Stack - 2)
doBREAK	(---)	compiled by BREAK. Unconditional jump to value from Return Stack
doZ-BRK	(flag --)	compiled by Z-BREAK if (flag == 0) linear loop progression else jump to value on Return Stack.

Compiler Operators

.S	(---) BACKGROUND	sends all entries of the DataStack, TOS last, enclosed by [...]. For debugging, .S may be compiled, but should be removed in final code
OPS	(---) IMMEDIATE BACKGRND	Sends a list of all Kernel Operator names followed by their index in the operators array (HEX), which is compiled in User Threads for this operator. Useful information for debugging.
USER	(---) IMMEDIATE BACKGRND	Sends a list of all actually compiled User Threads. First the UserThread structure index of the thread is sent, next the name followed by the ThreadCode address, where the thread is compiled. For CONSTANTS, the actual value is sent, too For VARIABLES, the actual value and its storage address of is sent.

SEE	(---) IMMEDIATE BACKGRND	Syntax: SEE <User Thread name> The compiled thread is sent as ASCII text. Additional info in parentheses: doLIT and doVALUE: numerical value. doADDR: value storage address others: jump target code address. See op MEM. Because all SRAM of STM32 processors is in the range 0x2000xxxx, only the LOW 16BIT are sent via terminal in HEX without leading 0x...
ST	(---) BACKGRND	enables single step execution of User Threads actual information is shown similar to the output of SEE. Display: token name, output, Data Stack, if context relevant: Return Stack At thread runtime, TAB from terminal triggers execution of next token Special cases: 'ESC' terminates thread execution immediately (escapes endless loop) CTRL_B terminates long BLOCK times To release a hanging BLOCK or KEY, first push TAB, next CTRL_B or the intended key. ST may be compiled, should be removed in final code
RUN	(---) BACKGRND	terminates single step mode. May be compiled too, but should be removed in final code
MEM	(addr(HEX) --) IMMEDIATE BACKGRND	sends 48 words of SRAM as ASCII table starting from addr. Because SRAM of the STM32 processors is in the range 0x2000xxxx, only the LOW 16BIT (4 hex nibbles) are entered. MEM may be compiled, but should be removed in final code
FORGET	(---) IMMEDIATE	Syntax: FORGET <User Thread name> The named User Thread and all newer ones are deleted
FS	(---) IMMEDIATE	"Flush Stack", deletes all items on the Data Stack. Most times used to remove garbage from the Data Stack.
ABORT	(---) IMMEDIATE	Clears all stacks and resets all system parameters. Internally executed by the system after error messages.
REPLACE	(---) IMMEDIATE	Syntax: REPLACE <supplier name> <target name> Useful for experimental replacement of old token by new ones without new compilation. Only code between supplier and target is replaced. <supplier name> is always the first search hit from TOP of userToken[] <target name> is always the first hit searched from TOP of userToken[]. If supplier and target have the same name, the search for target is continued down towards bottom of userToken[]. This PROCEDURE WORKS ONLY WELL if supplier is the first hit and target is the first or second hit of search from TOP of userToken !!
RESTORE	(---) IMMEDIATE	Syntax: RESTORE <original name> Due to special encoding of the userToken[] structure, the original token can be restored after REPLACE. Only code between original and former supplier is restored.
: (colon)	(---) IMMEDIATE	Starts a new compilation, followed by the name of the new thread and the code in ASCII text. All names must be separated by a SPACE. Numbers have higher priority than thread names. So avoid thread names which could be numbers (like DAC, ADC, BEE) A User Thread with the same name as a Kernel Operator overrides the Kernel Operator (will be compiled instead in future threads). Experimentally this can be used productive for modification of Kernel behaviour, but else will cause problems.
; (semis)	(---) IMMEDIATE	Terminates every compilation, i.e must be the last thread token.

System Operators

AUTOEXE	(---) IMMEDIATE	Syntax: AUTOEXE <User Thread name> The specified user thread is automatically executed at system start. To get it available at system start, it must be stored to Flash and the system must start from Flash. See LOAD-F and SAVE-F. AUTOEXE 0 deactivates autoexe function. Ignore error message!
----------------	------------------------	---

Q	(---) IMMEDIATE BACKGRND	"Quest" sends a list of system parameters:SRAM array start addresses, project number, AUTOEXE thread, Number Base, Baudrate, SPI, PWM
SAVE-F	(---) IMMEDIATE	Burns the actual User Threads and system config. as "Project" in Flash The project number (0 to 5) is asked after SAVE-F is entered. As precaution, operation must be confirmed with upper case 'Y'. Flash can be rewritten up to 10000 times. CForth-DMX supports only project no. 0-4. Remaining 16kB Flash is used for storage of 32 DMX "lighting scenes", shared by all projects
LOAD-F	(---) IMMEDIATE	Loads a user stored "Project" from FlashThe project number is asked after LAOD-F is entered. As precaution, operation must be confirmed with upper case 'Y'. 6 "Projects" are supported (STM32F042 only Project#0 and 1) Project#0 is "empty" with default parameters, overwrites existing code! Projects #1-5 are loaded from Flash If the Flash area of a Project is "virgin", an empty project is loaded. Global system settings are loaded project independent from extra Flash page. STM32F042 code stores only one project in Flash (+1 in SRAM) Special CForth/DMX: the stored DMX levels are not reloaded (for this purpose use operator XL_ALL, see below). Fadetime is loaded.
EMPTY	(---) IMMEDIATE	STM32F042 only. Replaces LOAD-F(project #0)
// and (<SPACE>)		// and (<SPACE> initiated comments are supported They are caught and deleted directly in the "query()" terminal input handler, so they are not explicitly listed as operators. Any comment is valid UNTO and ONLY UNTIL LINE END (default=CR)) style comment termination is not supported.

Following operators are not supported by unmodified Nucleo boards and special "CForth DMX Mini":

VID and PID	(---) IMMEDIATE	Specify the VID/PID used for USB communication. To avoid errors, enter VID or PID in HEX,4 digits with 0x + leading zeroes Be careful not to shoot your USB access (appropriate Windows .inf file !) By default, the VID/PID and Windows .inf provided by ST-LINK is used. For non-evaluation and public use, a legal VID/PID must be configured. New entered VID/PID is not active before stored in Flash and system restart.
BAUD	(baudRate --) IMMEDIATE	Specifies the USART baudRate (default=115200). For easy handling, only the first 2 digits of the baudRate are entered: 11=115200, 57=57600, 38=38400, 31=31250(MIDI), 19=19200, 96=9600 23=23400 may not be supported by hardware, make a test. New entered baudRate is not active before stored in Flash and system restart. Exception: if SPLIT = 1 or 3, baudRate changes immediately
CHANNEL	(channel --) IMMEDIATE	Enter the MIDI channel for received MIDI messages (1...16), default=1. Not relevant for transmitted MIDI messages New entered channel is not active before stored in Flash and system restart.
SPLIT	(type --) IMMEDIATE	Specifies the organisation of terminal versus serial I/O (default = 0) type == 0: USB and USART work parallel as terminal type == 1: USB is terminal, USART is serial I/O as COM port type == 2: USART is terminal, USB is serial I/O as virtual COM port type == 3: USB is terminal, USART is serial MIDI I/O (for transmission, no difference between MIDI and COM, but received data are handled differently. Any baud rate works in MIDI mode, so MIDI can be served via RS-232 interface, too.) type == 4: USART is terminal, USB is serial I/O as MIDI interface (USB MIDI works significantly different from virtual COM)

For most users probably MIDI is not an important feature. It is added because some of the hardware described at my website already have a MIDI interface. It may be useful when CForth is used in media art context.

Features of the MIDI interface:

While **any MIDI message can be sent**,

the support for **received MIDI messages** is rather poor:

--- each PROGRAM CHANGE message triggers the user thread whose index (if programmed) in the user thread structure is the same as the MIDI program number of the message.

Attention: Program numbers here are counted **starting from 0**.

Most MIDI tools need input +1 higher. I.e. commonly used program no.1 technically will send a PROGRAM CHANGE with data byte 0.

--- each CONTROL CHANGE message

--- triggers the user thread whose index (if programmed) in the user thread structure is the same as the controller number (0...127) of the message. (check with "USER")

Additionally the controller value (second data byte 0...127) is put on the DataStack.

--- **if the addressed user Token is a CONST or VAR**, it's value is updated

with the second data byte of the CONTROL CHANGE message (unfortunately max 7 bit).

Special Operators of the CForth - DMX versions

The DMX command set is an expansion of the standard STM32G431 version.

I/O pins PA2 and PA3 (Veroboard assembly) or PB5 and PB6 (Mini version) are reserved for DMX output and are not available as peripherals.

As a special option the **MiniDMX** protocol is implemented (tested here with DMXControl 3), which is activated at system start when a jumper is placed between PA14 and Ground (programmer connector). **In this case, no CForth is started.** Usually MiniDMX is served via USB, then a lot of garbage is sent via USART/ RS-232 out.

XS	(slot ---)	selects the DMX channel (DMX slang "slot") for subsequent actions Due to limited SRAM, only 186 DMX channels are accessible!
XV	(value---)	sets the DMX level at the currently selected DMX channel (0...255)
XN	(byte ---)	increments the selected DMX channel and sets the DMX level there
X=	(number ---)	copies the DMX level of the selected DMX channel to the following 'number' channels.
XFT	(Fadetime ---)	sets the the time during a changed DMX level is faded from the actual value to the new final value. Fadetime is entered in 1/10 sec steps (0 ... 127=12.7 seconds) Once triggered, the fade continues until finished, even when settings of other channels are changed
X+"	(---)	increases the DMX level of the selected DMX channel by one
X-	(---)	decreases the DMX level of the selected DMX channel by one
X0	(---)	switches all DMX channel to zero level immediately (no fade)
XCYC	(cycle --)	set DMX cycle =number of transmitted channels (24 <= cycle <= 512)
X-BOOT	(scene --) IMMEDIATE	saves "scene" (0...64) permanently to be loaded at system start
X-SAVE	(scene --) IMMEDIATE	saves the active DMX transmit buffer as "lighting scene" (1...64) in Flash memory. Lighting scenes in Flash are shared by all projects.
X-LOAD	(scene --)	loads DMX lighting scene(0...64) from Flash to active DMX transmit buffer, using the actual fadetime. scene 0 is dark, all DMX channels=0
XL-PART	(scene number slot --)	copies 'number' DMX levels from Flash lighting 'scene' (0...64) starting from DMX channel 'slot' of 'scene' stored in Flash into the active DMX transmit buffer starting there from the actually selected DMX channel (e.g. XS) This may be used to copy short lighting patterns to other lamps
XR	(number ---) IMMEDIATE BACKGRND	sends 'number' DMX levels ASCII text formatted to the terminal, starting from the actually selected DMX channel!

contact: wschemmert@t-online.de, <www.midi-and-more.de>

* Right of technical modifications reserved. Provided 'as is' - without any warranty. Any responsibility is excluded.

* This description is for information only. No product specification or useability is assured in juridical sense.

* Trademarks and product names cited in this text are property of their respective owners