

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 CForth intention is to make a straightforward user oriented 32 bit Forth system, specifically to organize **electronic control of art objects and installations or similar simple process control**. No attempt is made to comply with established Forth standards.

CForth is completely programmed in C (GNU-C). The name is derived from the C-program language, which is unusual at Forth. There is no relationship with Mitch Bradley's CForth

No attempt is made to "program Forth by Forth". CForth uses several specific "doAll" runtime operators for loops, conditionals and VARCONs ="variable constants". At the expensive of firmware code length, everything is programmed on rather flat subroutine nesting. So I think, CForth is an acceptably fast Forth system, which is rather easy to understand and to manage.

Instead of classic Forth "Words" the termini "**Kernel Operator**" (built in the firmware) and "**User Token**" (user programmed mix thread of Kernel Operators and User Token) are used.

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 (pieces of ARM machine code),

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 primitive operator which is compiled by another operator and executed at runtime. Cannot be handled directly by the user, but knowledge about is essential for debugging. COMPILEONLY and IMMEDIATE are combined in some cases.

Special is the BACKGROUND attribute, which allows selected Kernel Operators (see Glossary) to be executed in a simple background (or say "debugging") process while a User Thread is executed. BACKGROUND and IMMEDIATE are combined in some cases Good for manipulation of VARCONs, for single step operation and break of endless loops (**ESC** key).

CTRL_B terminates long blocking delay (MS, DS) or hanging input (KEY, RX). **TAB triggers** single step. Wrong background terminal input causes an error message but does not abort the main process.

User Token features are listed in the "**UserHeader**" array of structures with 3 elements:

1. "**name**", ASCII string, max 15 chars (STM32F042 max 7 chars) plus null termination.
2. "**toCompile**" contains the absolute start address of the compiled token thread in the Token Code array **or the actual data value of a VARCON** ("variable constant").
3. "**attributes**" contains the index of the compiled thread in the Token Code array or the VARCON attribute. Double storage of thread address and thread index has advantages for token search and makes Kernel Operators REPLACE and RESTORE possible.

The **User Token Code is organized** as a linear sequence of token (i.e. references to code):

The complete user code for STM32G4xx and STM32L476 versions is stored in a single array of 32 bit words. The code for STM32F042 is stored in a single array of 8 bit bytes due to very limited memory size. (**new from release OCT 2021**, older versions were organized same way as STM32G4xx). Each "User Thread" occupies a defined part within this array.

Within this array **during compilation** references to "Kernel Operators" and "User Token" are added in linear sequence. Most runtime primitives ("do...") are followed by an additional entry which contains a literal, specific address or a string to be handled at runtime.

At runtime, thread entries are read, interpreted and executed in linear sequence. Each time, when the read entry points to a User Thread, execution jumps to that start index, executes this code and when the compiled ';' (semis) entry 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". The background process can be switched off completely, saves runtime. All this can be handled from the background process.

To get distinguished, Kernel Operators and User Token are coded differently in a User Thread:

STM32G4xx and STM32L476: each **Kernel Operator** is compiled in a User Thread with its index in the array of Kernel Operator code pointers (32 bit word with bits 31-16 = zero). But each **User Token** compiled in another User Thread is compiled with its **absolute start address** (i.e.SRAM address > 0x20000000), so easily distinguished from Kernel Operators, provides fast execution.

STM32F042: each Kernel Operator is compiled in a User Thread with its index in the array of Kernel Operator code pointers (byte with bit7=0). But each User Token is compiled as it's **index in the UserHeader structure** with additionally bit 7 set, so distinguished from Kernel Operators. So theoretically max 128 Kernel Ops and 128 User Token are possible. The UserHeader structure contains the element "toCompile", where the absolute User Code Thread start address is stored. So, for the gain of up to 3 times more Token Code memory a few additional CPU re-addressing operations are necessary for User Token. Calling Kernel Operators does not take additional runtime except 8 to 32 bit expansion of its index in the array of Kernel Operator code pointers.

STM32G4xx and L476 versions support **168 User Token and 3072 words of code space**.

STM32F042 supports **95 User Token and 1536 bytes of code space**. These limits are optimized to fit well into SRAM and Flash pages and good ratio of codelength per UserToken.

CONSTANTS and VARIABLES are combined together in the VARCON data type

("variable constant") which is re-valued with Kernel Operator REV and the according runtime primitive doREV, but puts its value on the stack like a CONSTANT when called by name. Provides best runtime speed and flexibility and modification from background process. Temporary data storage on the Return Stack is **not** supported (Standard Forth words R, >R, R>, R@ etc.).

All math operations are performed as 32bit signed integer.

No warning or error message, when result is out of 32 bit 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 native 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 has no native 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 (very simple process control) introduction of unsigned operations seems not necessary (exception "U<"). Though it would be rather easy to implement a set of float operations - due to the FPU of G4 and L4 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.

All CForth versions (except the small DMX capable) provide an RS-232 terminal interface parallel with the USB terminal. For hardware debugging, USART terminal is more practical.

CForth "Projects":

In accordance with the memory design of the STM32G4xx processors, up to 6 "**Projects**" are supported. The versions for STM32L476 have the same memory configuration. Due to restricted memory, the STM32F042 supports only 2 projects (one in Flash, experimental work in SRAM is possible independently). Actual CForth operation (interactive and compilation) is always stored and done in SRAM. "Compilation into Flash" is not supported.

By default Project #0, an "Empty Project" is loaded with default parameters into SRAM. This installs two VARCONs N1, N2 (for motor control), and four VARCONs P1....P4 **as User Token**, to be helpful for interactive process control. User Threads can be compiled and executed in the Empty Project, but are lost when a power cycle or reset is made.

Projects #1 to 5 (with DMX only 1, 2) are **loaded from Flash** to SRAM at CForth start or with operator "**LOAD**". Existing code in SRAM is overwritten then. If the corresponding Flash area is "virgin", an empty project is loaded instead. "Project#0" always loads an empty project. Deviating from this, the F042 version loads an empty project with operator EMPTY. LOAD and SAVE don't ask a project number there.

Any project in SRAM can be **burnt into Flash** (operator "**SAVE** ") as Project #1 to 5 (with DMX only 1 and 2, STM32F042 only one), assigned Flash area is overwritten then.

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, intended ColdStart or reset, the last saved project is automatically reloaded** from Flash into SRAM (or an empty project). Project#0 can be "saved" too. CForth keeps the empty project then - but actual global system parameters are stored.

You can **change the project number** which is automatically loaded at CForth start: Load this desired project (modify it or not) and restore it into Flash (another project number).

Glossary of Operators:

Operator names written **red** are **not or partially implemented for STM32F042**

Operator names written **green** are **ONLY** implemented in versions **with MIDI interface**.

Arithmetic Kernel Operators:

DROP	(x2 x1 -- x2)	
DUP	(x -- x x)	
SWAP	(x2 x1 – x1 x2)	
OVER	(x2 x1 – x2 x1 x2)	
ROT	(x3 x2 x1 -- x2 x1 x3)	rotate 3rd stack entry up
-ROT	(x3 x2 x1 – x1 x3 x2)	rotate TOS down

PICK	(xn x(n-1).x2 x1 n -- xn x(n-1)...x2 x1 xn)	keep n(=TOS) in mind and copy the (n+1)th element of stack on TOS 0 PICK performs DROP, 1 PICK performs DUP. 2 PICK performs OVER
ROLL for STM32F042 only with MIDI	(xn x(n-1)...x2 x1 n --- x(n-1) x2 x1 xn) eg(x3 x2 x1 3 - x2 x1 x3)	drop TOS, but keep n in mind. Rotate the new n-th stack element up to TOS. 0 ROLL and 1 ROLL simply drop TOS 3 ROLL performs ROT, sequence "4 ROLL 4 ROLL" performs 2SWAP
INJECT	(x(n+1) xn... x2 x1 n --- x1 x(n+1) xn... x2) eg(x3 x2 x1 2 - x1 x3 x2)	drop TOS, keep n in mind. Rotate new TOS(x1) down below the n-th stack element. 0 INJECT and 1 INJECT simply drop TOS 3 INJECT performs -ROT. "n PICK n INJECT" double a deep Stack Item
+	(x2 x1 -- x2+x1)	signed 32bit addition
1+	(x -- x+1)	add 1 to TOS
-	(x2 x1 -- x2-x1)	signed 32 bit subtraction
1-	(x -- x-1)	subtract 1 from TOS
ABS	(x -- abs(x))	
<<	(x2 x1 - x2<<x1)	shift range X1:0 ...31
>>	(x2 x1 - x2>>x1)	shift range X1:0 ...31 if X2(bit31) is set, "ONE"s are shifted in ! (preserve minus sign)
&	(x2 x1 -- x1&x2)	bitwise AND replaces standard Forth "AND"
 	(x2 x1 - x2 x1)	bitwise OR replaces standard Forth "OR"
XOR	(x2 x1 -- x2^x1)	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 bitN -- xmodified)	bitN of x is set to 1 (bitN 0..31)
BCLR	(x bitN -- xmodified)	bitN of x is cleared (set to 0) (bitN 0..31)
BTST	(x bitN -- bitvalue)	bitvalue = boolean value (=0 or 1) of bitN
RANDOM	(-- random)	a 32bit random number is generated with 1kHz update frequency

Comparison Operators

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

Memory Operators

VARCON	(x ---) IMMEDIATE	("Variable Constant") Syntax: 789 VARCON CONNY a global VARCON named CONNY is compiled with initial value 789. If CONNY is called by name, the actually stored value is put on TOS
REV	(x ---) IMMEDIATE BACKGRND	replace (re-value) value of named VARCON with value of TOS (next use and in previously compiled threads). Syntax: 234 REV CONNY

		May be applied from Background process for interactive control of thread flow (IF ., Z_BREAK) or modification of process parameters else.
W	(value addr ---)	replaces Standard Forth ! (typing is more easy) Not needed in standard CForth programs, but useful for code hacks.
R	(addr --- value)	replaces Standard Forth @ (typing is more easy) Not needed in standard CForth programs, but useful for code hacks.
ARRAY	(index --- value) (value -index ---)	32bit signed integer array, 1000 entries. Index 0 cannot be used positive TOS=index: read value from ARRAY[index] to TOS negative TOS=index: write NOS into ARRAY[-index]

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 Blocking may be ended with terminal input CTRL_B
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 .Max 78 bytes. 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
%	(---) IMMEDIATE BACKGRND	set system number base DECIMAL In this case hex numbers can be entered with preceeding 0x. Even 0x-... is accepted: 0x-FFFFFFFF = 1
\$	(---) IMMEDIATE BACKGRND	set number base HEXADECIMAL The number base can be changed by % and \$ 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?	(--- count)	the actual count of bytes in the serial I/O input buffer, = 0 if none. In MIDI mode (see SPLIT), count of MIDI messages is returned
RX	(--- byte or MDIword)	execution blocks until a serial I/O byte or a MIDI message is received, returned on TOS. Blocking may be ended with terminal input CTRL_B
TX	(byte ---)	the least 8 bit of TOS are sent via serial I/O as raw byte
NTX	(b1 b2 bn n ---)	the n stack items b1 bn are sent via serial I/O (least 8 bit as raw bytes). Order of transmission: b1 first, bn last. n+1 stack items are dropped.
TX""	(---)	send the text between "" as byte stream via terminal. Remark see P"

Time Operators

NOP	(---)	may be used as minimal blocking delay. Does nothing else
XNOP	(n ---)	variable short delay, NOP is executed n times
MS	(t ---)	blocking countdown of t milliseconds is started. blocking is ended immediately by terminal input CTRL_B (bin code 2)
DS	(t ---)	blocking countdown of t*100ms (deci-seconds) is started (calls MS) blocking is ended immediately by terminal input CTRL_B (bin code 2)
TIX	(t ---)	NON -blocking background countdown of t*100ms is started. See WAIT
WAIT	(--- t)	returns remaining countdown of TIX in 100ms units may be used to construct simple timelines (see IF, CASE, <CASE)
FREQ12 STM32F042: FREQ,PWM1,2 only in version	(divisor ---) BACKGRND	starts STM32 Timer3 with 48MHz clock, which is divided by a prescaler. "divisor" sets the prescaler rate to provide a variable input frequency for Timer3. The prescaler range is 16bit. With prescaled input, Timer3 produces a constant overflow each 2000 prescaled clock pulses.

without MIDI PWM2 is not in motor version		This results in base frequency of FREQ12,PWM1,PWM2,TIME,PULS . This way the FREQ12 frequency can be modified in the range between 24kHz and ca.0.3Hz . 24 FREQ12 sets the base freq to 1kHz .
PWM1 STM32F042: range only 0... 100	(high --) BACKGRND	The PWM1 signal output is available at PA6. If PWM1 is started when FREQ12 is not active, FREQ12 gets provisionally activated with 1kHz overflow frequency. The HIGH percentage of PWM1 is (high/2000)*100 0 PWM1 sets PA6 permanently LOW, 2000 PWM1 (or greater) sets PA6 permanently HIGH -1 PWM1 will deactivate the PWM output at PA6, - but not stop FREQ
PWM2	(high --) BACKGRND	The PWM2 signal output is available at PA7. Else remark see PWM1
TIME	(-- "SysTime")	When FREQ12 is active, TIME is upcounted at each Timer3 overflow. At moderate base frequencies, this can be used as "System Time" base.
PULS	(-- number)	When FREQ12 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. TIX and WAIT may be easier to use, but PULS is kept as a low-cost by-feature.
FREQ34	(divisor ---) BACKGRND	starts STM32 Timer2 with 48MHz clock, which is divided by a prescaler. "divisor" sets the prescaler rate to provide a variable input frequency for Timer3. The prescaler range is 16bit. With prescaled input, Timer23 produces a constant overflow each 2000 prescaled clock pulses. Both together results in the base frequency of FREQ34, PWM3,PWM4. This way the FREQ34 frequency can be modified in the range between 24kHz and ca.0.3Hz . 24 FREQ34 sets the base freq to 1kHz .
PWM3	(high --) BACKGRND	The PWM3 signal output is available at PA15. If PWM3 is started when FREQ34 is not active, FREQ34 gets provisionally activated with 1kHz overflow frequency. The HIGH percentage of PWM3 is (high/2000)*100 0 PWM3 sets PA15 permanently LOW, 2000 PWM3 (or greater) sets PA15 permanently HIGH -1 PWM3 will deactivate the PWM output at PA15, - but not stop FREQ
PWM4	(high --) BACKGRND	The PWM4 signal output is available at PA1. Else remark see PWM3

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: 0xAF OH sets PA15 or %X B3 OH sets PB3 hex/decimal conversion: 0xA0 is decimal 160, 0xAF is decimal 175 etc
OL	(portPin ---)	sets addressed port pin as push/pull OUTPUT LOW Syntax: 0xAF OL sets PA15 0xB3 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: 0xAF IPU configures PA15 0xB3 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: 0xAF IPD configures PA15 0xB3 IPD configures PB3
IZ	(portPin ---)	sets addressed port pin as INPUT with HIGH IMPEDANCE Syntax: 0xAF IZ configures PA15 0xB3 IZ configures PB3
RDIG	(portPin -- 0 1)	returns the digital level at port pin. Independent of actual configuration and user availability of this pin. Syntax: 0xAF RDIG returns level at PA15
RANA	(PinNo -- value)	reads the 12 bit level of PinNo 0xA0...0xA7(0xB0) or shortened 0..7(8). Each level is read 8 times and the result is averaged (oversampling). When called first, the A/D converter gets initialized. Once opened, the A/D converter stays active during the session (consumes little current).

INI-SPI SPI is not in STM32F042 MIDI version	(mode bits clock ---)	initializes an SPI master with bits=4 to 32 bit transfer 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 due to memory saving code. 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 /CS must be handled separately (if necessary) by user with any pin else. This way, multi-slave configurations are possible. Used I/O pins: STM32G431(Mini)SCK=PB3, MOSI=PA15, MISO=PB4 Else STM32G4xx and STM32L476: SCK=PB3, MOSI=PB5, MISO=PB4 STM32F042: SCK=PA1, MOSI=PA4, MISO=PA0 (changed Dec2020)
SPI	(TxWord -- RxWord)	One data word is transferred. If not initialized explicitly before , SPI gets configured with default or with user values stored in Flash project
INI-I2C	(Addr --)	Initializes I2C. Addr is the slave device address 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 (i.e 256 added), then I2C will work with 400kHz clock. Slave address and bus speed can be changed at any time between two transmit and receive commands. Only port init is performed. Nothing is transmitted by INI-I2C. This slave address is transmitted automatically by I-TX, I-RX1, I-RXAN. STM32G4xx (Veroboard and Mini): SDA=PB7, SCL=PB8 Nucleo Boards: SDA=PB9, SCL=PB8 (wiring is easier then) These pins are specified "5V tolerant", i.e. can work with 5V I2C devices Max safe layout: SDA,SCL pullUp to 5V:3.9kOhm +10kOhm to Ground PB8 BOOT0 must be disabled , see last page of HWDIY manual!
I-TX	(b1 b2 ... bn n --)	n lower stack bytes are transmitted via I2C in the order b1 first, bn last b1 may have special meaning, e.g.word address for EEPROM write
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-RXAN	(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.
MOT1 STM32F042: MOT is only in motor version	(speed --) BACKGRND	control a (stepper) motor: provides a pair of continuous pulses at PA1,PA3 (Standard+Standard DMX G431) or PA0,PA1 (Nucleo) PA4,PA7 (small DMX). (pin groups are selected for best HW useability) speed is entered in milliseconds per phase change, max 16384. speed = 0 stops rotation. Negative speed = backward rotation. Number of steps is limited by value of permanent "user" variable N1. N1 = 0: permanent rotation. For unipolar motors, the inverted phase and switch motor power off must be generated externally. See HwDiy manual N1 = -1: PA1,PA3 (or A0,PA1 or PA4,PA7) control a H-bridge. speed controls PWM1, negative speed reverses rotation.
MOT4	(speed --) BACKGRND	control a (stepper) motor: provides a 4 wire pulse at PA1, PA3, PA7, PB0 (Standard and Standard DMX G431) or PA0, PA1, PA4, PB0 (Nucleo) or PA4, PA7, PA0, PA1 (small DMX). Function else like MOT1. Well useable together with TB6612 power amp. PWM1 is recommended to power motor up and down (TB6612 PWMA and PWMB commonly). Special cases for Control of a "normal" commutated DC motor: N1 = -1: PA1,PA3 (or PA4,PA7) control a H-bridge. N1 = 2: PA7,PB0 (or PA0,PA1) control a H-bridge. speed controls PWM1, negative speed reverses rotation.

MOT2	(speed --) BACKGRND	control a (stepper) motor: all features like MOT1, but pulse output PB7,PB8 (G431) or PB8,PB9 (Nucleo). Use N2 . "speed"&power: PWM3
-------------	---------------------	---

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 bitN ---)	SETS bitN of the addressed peripheral register
REGBCLR	(peripheral_reg_addr bitN ---)	CLEARs(sets =0) bitN of the the addressed peripheral register
REGTST	(peripheral_reg_addr bitN -- bitLevel)	returns level of bitN of the addressed peripheral reg.

Structuring Operators

IF	(-- addr) IMMEDIATE, COMPILEONLY	compiles doIF. Starts compilation of a IF ... ELSE ...THEN conditional Essentially provides a placeholder for jump address and sends its code address via stack to ELSE or THEN!
CASE	(-- addr) IMMEDIATE, COMPILEONLY	To be used in a decision cascade like CASE...THEN CASE ...THEN ... and for more flexible use of single conditional branches. Works internally like IF , but compiles a different criterion : At runtime , the second stack parameter (NOS) is compared with TOS. If both are equal , the CASE code is executed, else the program flow branches behind ELSE or THEN. Attention: at runtime TOS (=case criterion) is deleted, the former NOS becomes TOS. So it can be evaluated in a subsequent CASE...THEN cascade. Must be DROPEd if not needed anymore . Can be combined with ELSE, but this may cause unwanted behaviour in a CASE cascade. Compiles doCASE, starts compilation of CASE .ELSE.THEN conditional
<CASE	(-- addr) IMMEDIATE, COMPILEONLY	To be used in a conditional like <CASE...else ...THEN DROP and for more flexible use of single conditional branches. Works internally like CASE , but compiles a different criterion : At runtime , the second stack parameter (NOS) is compared with TOS. If NOS is less than TOS , the CASE code is executed, else the program flow branches behind ELSE or THEN. See further remarks at CASE above, these are valid for >CASE, too
ELSE	(addr -- addr) IMMEDIATE, COMPILEONLY	compiles doELSE, the middle part of a IF ... ELSE ...THEN 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
THEN	(addr --) IMMEDIATE, COMPILEONLY	compiles termination of IF,or CASE,or <CASE..ELSE..THEN conditional Essentially it compiles the terminating code address into the placeholder behind doIF, or doCase, or do<CASE, (or doELSE if present).

Loops are handled quite differently from standard Forth implementations.

Because most technically interested people have some knowledge in C or Java (Script),

I have tried to organize loops more C-style for easier learning and remembering

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

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

DO ..ZBREAK .. AGAIN is functionally synonym with standardForth BEGIN..WHILE..REPEAT

FOR ... LOOP is functionally synonym with standard Forth 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

doZBRK checks TOS: if = 0, the loop is BREAKed, else execution continues linear

If doZBRK 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 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
ZBREAK	(---) IMMEDIATE, COMPILEONLY	compiles doZBRK
RETURN	(---) COMPILEONLY	At runtime, it terminates a User Thread immediately (like "C" return)

Runtime Operators (Runtime Primitives)

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 ST

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 F042: doLIT8B for number <256	(-- 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 when a VARCON is called by name during compilation In the compiled thread, the VARCON value address is compiled behind doVALUE (STM32F042: UserHeader[] index), but at runtime it puts the actual VARCON value on TOS.
doREV (replaces standard Forth doVAR)	(-- address)	compiled by REV In the compiled thread, the VARCON UserHeader[].toCOMPILE field address = value storage address is compiled behind doREV. (STM32F042: UserHeader[] index) At runtime, it writes TOS into this address, no extra W command needed
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 THEN if != 0, execution is performed until doELSE if present or continues linear
doCASE	(flag reference -- flag)	compiled by CASE At runtime, it compares 2nd stack element "flag" with TOS"reference", if equal , execution is performed until doELSE, or continues linear if unequal , a jump is initiated: behind doELSE if present, or to code address compiled by THEN Attention: at runtime TOS is deleted, "flag" is proceeded as new TOS
do<CASE	(flag reference -- flag)	compiled by <CASE At runtime, it compares 2nd stack element "flag" with TOS "reference", if "flag" is less than "reference" , execution is performed until doELSE if present, or continues linear if "flag" is greater than or equal "reference" , a jump is initiated: behind doELSE if present, or to code address compiled by THEN Attention: at runtime TOS is deleted, "flag" is proceeded as new TOS
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 THEN). if the flag of doIF == 0, execution is continued behind doELSE. Behaviour after doCASE or do<CASE effectively the same, see above. 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
doZBRK	(flag --)	compiled by ZBREAK 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
FS	(---)	"Flush Stack", deletes all items from the Data Stack. Removes garbage from the Data Stack. May be compiled - only do this when result is sure
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 TokenCode address, where the thread is compiled. For VARCONs, the actual value and its storage address of is sent. For other User Token, first the address in the TokenHeader struct is appended, next its code start address in the Token Code array. Because all SRAM of STM32 processors is in the range 0x2000xxxx, only the LOW 16BIT are sent via terminal in HEX
SEE	(---) IMMEDIATE BACKGRND	Syntax: SEE <User Token name> The thread is de-compiled as ASCII text. Additional info in paranthes: doREV value storage address, doVALUE followed by VARCON name. others like doIF, ...:conditional jump address. Use op MEM if needed. Because all SRAM of STM32 processors is in the range 0x2000xxxx, only the LOW 16BIT are sent via terminal in HEX
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. Use \$ or leading 0x. MEM may be compiled, but should be removed in final code
ST	(---) BACKGRND	enables single step execution of User Threads, actual token 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 MS and DS times, terminates hanging KEY,RX First push TAB, next CTRL_B. CTRL_R initates cold start (data loss !) ST may be compiled, but takes runtime if not needed.
TRACE	(--) BACKGRND	Same as ST, but output msg permanently repeated, to TAB trigger.
RUN	(---) BACKGRND	terminates single step mode. May be compiled too, but takes runtime if not needed. RUN is always default mode after CForth start.
NO-DBG	(---) BACKGRND	"No Debug": Turns the execution of the background process OFF, saves about 40% runtime. Care must be taken when an endless loop without BREAK or similar is executed. Only exit then without reset: compile DEBUG within the loop.

DEBUG	(---)		Compiled in UserThreads, it starts background process unconditionally Attention: while CForth is in No Debug mode, any terminal input of CTRL_D (bin 4) switches back to RUN mode. Take care if terminal input of CTRL_D is intended!
FORGET	(---)	IMMEDIATE	Syntax: FORGET <User Thread name> The named User Thread and all newer ones are deleted
ABORT	(---)	IMMEDIATE	Clears all stacks and resets all system parameters. Internally executed by the system after error messages.
REPLACE available in STM32F042 Low Power version	(---)	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 UserHeader[] <target name> is always the first search hit from TOP of UserHeader[]. Supplier must be newer than target. For a "retro" REPLACEMENT compile a new UserToken with the same code as the old target and use this as supplier. Other code hacks are possible with kernel operators MEM and W
RESTORE available in STM32F042 Low Power version	(---)	IMMEDIATE	Syntax: RESTORE <original name> Due to special encoding of the UserHeader[] 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. "AUTOEXE 0" doesn't start a User Thread. Stored AUTOEXE may cause deadlock , only resolvable by Flash erase! To avoid AUTOEXE, hold the RS-232 input at +3.3 or +5V during power on. (If RS-232 not implemented, hold USART RX pin (PA10 or PA3) at GND. Unmodified Nucleo: press User button. G431 Mini: PA3 grounded.
Q	(---)	IMMEDIATE BACKGRND	"Quest" sends a list of system parameters:SRAM array start addresses, project number, AUTOEXE thread, Number Base, Baudrate, SPI, PWM
SAVE	(---)	IMMEDIATE	Burns the actual User Threads and system config. as "Project" in Flash. Global settings are stored project independent in a separate Flash page. The project number (0 to 5)(DMX:0 to 2) is asked after SAVE is entered. STM32F042 saves only one project in Flash, doesn't ask a number As a precaution, operation must be confirmed with upper case 'Y'. Flash can be rewritten up to ca.10.000 times (specified by STM). Special CForth-DMX: the stored DMX levels are not saved (for this purpose use operator X-SAVE, see below). Fadetime is saved CForth-DMX supports only project no. 0-2. Remaining 48kB Flash is used for storage of 64 DMX "lighting scenes", shared by all projects
LOAD	(---)	IMMEDIATE	Loads a user stored "Project" from Flash. The project number (0 to 5) (DMX:0 to 23) is asked after LOAD is entered. LOAD doesn't change the project number loaded at PowerOn. As a precaution, operation must be confirmed with upper case 'Y'. Project#0 is "empty" with default parameters, overwrites existing code!

		<p>Projects #1-5 are loaded from Flash STM32F042 supports only one project in Flash and one Empty project. So SAVE and LOAD don't ask a number, instead the operator EMPTY loads the Empty project. If the Flash area of a Project is "virgin", an empty project is loaded. Global parameters are loaded project independent from Project number. Special CForth-DMX: the stored DMX levels are not reloaded (for this purpose use operator X-LOAD, see below). Fadetime is loaded.</p>
EMPTY	(---) IMMEDIATE	STM32F042 only. Starts an Empty Project Replaces LOAD(project #0)
// and (<SPACE>		<p>// 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.</p>

Following operators are not supported by unmodified Nucleo boards

VID and PID	(---) IMMEDIATE	<p>Specify the VID/PID used for USB communication. To avoid errors, enter VID or PID in HEX,4 digits w. 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 active after stored in Flash and system restart.</p>
--------------------	-------------------	---

Some following operators are ONLY supported by versions with MIDI, not by unmodified Nucleo boards:

<p>BAUD not supported by unmodified Nucleo boards else: supported by STM32G4xx with RS-232 supported by STM32F042 all versions</p>	(baudRate --) IMMEDIATE	<p>Specifies the USART baudRate (default=115200). For easy handling, only the first 2 digits of the Baud Rate are entered.No matter if hex or decimal. Eg- 38 or 0x38 selects38400Baud. 11=115200, 57=57600, 38=38400, 31=31250(MIDI), 19=19200, 96=9600 New entered baudRate is not active before stored in Flash and system restart. Exception: if SPLIT = 1 or 3, baudRate changes immediately. F042:save BAUD in Flash project, reload, start EMPTY with user BAUD The actual hardware for G4xx and F042 does not use an external TX inverter. Level is inverted at USART register level. For hardware with external inverter (old HW version or MAX232 etc) enter BAUD parameter with leading 1. Example: 157 or 0x157 selects 57600 Baud with external inverter</p>
CHANNEL	(channel --) IMMEDIATE	<p>Enter the MIDI channel for received MIDI messages (1...16), default=1. Special case 0 CHANNEL accepts any MIDI channel. Not relevant for transmitted MIDI messages The new entered MIDI channel is not active before the project is stored in Flash and system is restarted (or project reloaded).</p>
SPLIT	(type --) IMMEDIATE	<p>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 operated as USB-MIDI interface (USB MIDI works significantly different from virtual COM) Simplified for CForth versions without MIDI interface: type == 0. USB and USART both work parallel as Terminal (default) type == 1: only USB is Terminal type == 2 not implemented here</p>

Features of the MIDI interface:

Any MIDI message can be sent. Message type, values and sequence of messages is freely composed by the user, same way as RS-232 transmission. See operators TX, especially NTX for MIDI Channel Msgs, TX”

When serial I/O is in MIDI mode (see SPLIT), the support for **received MIDI messages** is organized as follows:

In a special background process, received MIDI bytes are extracted from the active receive buffer (USB or USART, depending on SPLIT configuration), **packed to a single data word** and stored in a cyclic MIDI buffer (first in – first out), max 256 entries (STM32F042 only max 16 entries). This packing operation is performed automatically by the terminal input handler "query()" and any time one of the kernel Operators RX? (before number of received messages is returned) and RX (is unblocked as soon as one or more messages are packed in MIDI buffer) is executed.

This buffer is read with RX? and RX on TOS, but 32 bit words are returned then instead of single bytes.

The received word is structurally HEX formatted, so check it visually on terminal with “PH”. Any kind of MIDI message **except SysEx messages** is supported (SysEx messages are automatically deleted from the receive buffer).

MIDI Channel Messages are filtered according to the configured MIDI channel 1...16.

If "0 CHANNEL" is configured, all MIDI Channel Messages are accepted.

Special MIDI Messages (MIDI Time Code, Song Position Pointer, Song Select, Tune Request and one byte Real Time Messages) are accepted. MIDI Running State is supported.

MIDI Messages are packed into words (and read on TOS by RX) **as follows:**

3 Byte messages: bits 24 - 31 = 0

bits 16 - 23 = Status byte

bits 8 - 15 = MIDI Data byte 1 (like Note Pitch, Controller No....)

bits 0 - 7 = MIDI Data byte 2 (like Velocity, Controller Value....)

2 Byte messages: bits 24 - 31 = 0

bits 16 - 23 = Status byte

bits 8 - 15 = 0

bits 0 - 7 = MIDI Data byte (like Program No....)

(the single data byte is packed into bits 0-7

for easier combined handling of 2 and 3 byte messages

and easier extraction of usually most application relevant byte)

1 Byte messages: bits 24 - 31 = 0

bits 16 - 23 = Status byte

bits 8 - 15 = 0

bits 0 - 7 = 0

Evaluation and handling of the packed MIDI Messages on TOS are freely done by the user. Check and extraction of the different MIDI bytes is made with AND and SHIFT operations.

Special Operators of the CForth - DMX versions

The DMX command set is an expansion provided with the G4xx Standard DMX and the Mini G431 DMX version. **Kernel Ops "XRGB" and "XNRGB" are added new September 2021.** I/O pins PA2 and PA3 (Standard DMX) or PB5 and PB6 (Mini) 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.** MiniDMX is exclusively served via USB virtual COM, RS-232 is too slow.

XS	(slot ---) BACKGRND	selects the DMX channel (DMX slang is "slot") for subsequent actions. Lowest channel number is 1, max slot is 512
XV	(level ---) BACKGRND	sets the DMX level at the currently selected DMX channel (0...255)
XRGB	(rgb level triple)	sets the color of a RGB lamp (3 consecutive DMX channels) Parameter: bits 0..7= addressed DMX channel = red bits 8-14= (addressed DMX channel + 1) = green bits 15-23= (addressed DMX channel +2) = blue (bits24-31 ignored)
XNRGB	(rgb level triple)	same as XRGB, but selected DMX level is pre-incremented by 3
XSET	(slot level ---)	combination of XV and XS: set DMX level at DMX channel. The DMX channel remains set for subsequent actions like XN, X=, XR
XN	(level ---)	pre-increments the selected DMX channel and sets the DMX level there
X=	(number ---)	copies the DMX level of the selected DMX channel into the following 'number' channels.
XFT	(fadetime ---)	sets the the time fading DMX level from actual 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 "fadetime" or 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	(---) BACKGRND	switches all DMX channel to zero level immediately ("Panic", no fade)
XFLASH	(scene time --)	during "time"(deci sec !!) set all DMX channels to Flash preset "scene"
XCYC	(cycle --)	set DMX cycle =number of transmitted channels (24 <= cycle <= 512)
X-BOOT	(scene --) IMMEDIATE	saves "scene" (0...96) permanently to be loaded at system start
X-SAVE	(scene --) IMMEDIATE	saves the active DMX transmit buffer as "lighting scene" (1...96) in Flash memory. This Flash area is separated from the CForth project storage. Lighting scenes in Flash are shared by all projects.
X-LOAD	(scene --)	loads DMX lighting scene(0...96) from Flash to active DMX transmit buffer, using the actual fadetime. Scene 0 is dark, all DMX channels=0
XL-PART	(scene slot number --)	copies 'number' DMX levels from Flash lighting 'scene' (0...96) 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 useful to copy short lighting patterns to other lamps
XR	(number --) IMMEDIATE BACKGRND	returns the levels of the number of DMX channels starting from selected DMX channel

Experimental feature of non-DMX versions (only STM32G4xx and F4769)

Four empty Kernel Operators MY1,MY2,MY3,MY4 are implemented, which can be filled by the user with its own C-code without need to care how to integrate it in the CForth system. Only simple code without attributes can be inserted. Parameter transfer via data stack. Other Kernel Operators can be called as C-subroutines. User specific interrupt handlers can be added, need not to be declared explicitly. Be careful not to conflict with interrupt handlers already used by CForth. Max total code length ca 3.3kBytes.

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

* 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