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.

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

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 "program Forth by Forth" nor comply with any common Forth standard. CForth uses several specific "doAll" runtime operators for loops, conditionals and VARCONs ="variable constants". At the expensive of firmware code length, most 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. The Forth community claims Forth to be an alive program language, so I feel free to introduce some new naming and handling concept. Instead of classic Forth "Words", the termini "**Kernel Operator**" (built in the firmware) and "**User Token**" or "**User Thread**" (=compiled mix 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** (= 0 if none assigned)**:**

COMPILEONLY can only be compiled in compiling mode
IMMEDIATE cannot be compiled, only be executed immediately even in compiling mode.
RUNTIME is a runtime prmitive 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 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. "t**oCompile**" 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 Code is organized** as a linear sequence (called thread) of Kernel Operators and pre-existing User Token (i.e.references to code). Each User Token points to a User Thread.

The complete user code for STM32G4xx, STM32 F411 and STM32L476 processors is stored in an array of 32 bit words. Each "User Thread" occupies a defined part within this array. These CPU versions versions support **168 User Token and 3072 words of code space**.

To get distinguished, Kernel Operators and User Token are coded differently in a User Thread:
STM32G4xx ,F411, L476: 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.

The code for STM32F042 (and STM32F030, STM32L031 Low Power) is stored in a single array of 8 bit bytes due to very limited memory size. 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.

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 (VARCOM !!) is compiled as it's **index it 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.

Within the user code 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.

CONSTANTs **and VARIABLE**s **are combined together in the VARCON data type ("variable constant")** which is re-valued with Kernel Operator W, and puts its value on the stack like a conventional Forth CONSTANT (corresponding runtime primitive doVAR) when called by name. (Example:23 VARCON CONNY, 45 W CONNY, CONNY .S). Takes more time during compilation, **but at token runtime, it provides best speed** and flexibility (even compared with classic ! and @) and allows modification from background process. **Temporary data storage on the Return Stack is not supported** (traditional Forth words R, >R, R>, R@).

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,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 - using the FPU of G4 and L4 processors - it would be rather easy to inplement a set of float operations, 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 and STM32L476 processors, up to 5 "**Projects**" in Flash are supported. The STM32F411 processors only support 4 "Projects", because they have only 7 big Flash segments. Due to restricted memory, the **STM32F042 supports only 1 project in Flash**. Project#0 "Empty" in SRAM is possible independently. Actual CForth operation (interactive and compilation) is always stored and done **in SRAM**. Runtime "compilation to-/read from Flash" isn't supported.

By default Project #0, an "Empty Project" is loaded with default parameters into SRAM. This installs four generally available VARCONs V1....V4 and up to three VARCONs NSTP1, NSTP2, NSTP3 (for motor control) **as User Token**, to be helpful for interactive process control. User Threads can be compiled and executed in the Empty Project, but **the Empty Project is lost when a power cycle or reset is made**. So it is recommended, to **save the Empty Project as a Flash Project before critical oerations** are made or to evaluate the project on a text editor and load in via terminal to be reconstructed easily in case of crash.

Any project in SRAM can be **burnt into Flash** (operator "**SAVE** ") as Project #1 to max #5 ((STM32G4xx with DMX only Project 1, 2 - STM32F411 with DMX supports 1,2,3 and STM32F042 only one), assigned Flash area is overwritten then. Project#0 can be "saved" too. CForth saves some global parameters in Flash and starts it at power-on or reset (doesn't work with STM32F042)

Projects #1 to max #5 are **loaded from Flash** to SRAM at CForth start or with operator "**LOAD** ".Existing code in SRAM is overwrittten then.

If the corresponding project Flash area is "virgin", an Empty project is loaded instead. "Project#0" always loads an Empty project

Together with the project specific code, some **global system parameters** (as active USB Vid/Pid, baud rate, number base) 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**). Deviating from this, the F042 version always loads – if saved – the Flash project, else an empty project with global default parameters is laoded.

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 (with another project number).

Glossary of Operators:

Operator names written **red** are **not implemented at STM32F042**
Operator names written **dark red** are **not implemented in all HW versions. Check with "OPS"** or read below
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	(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 peforms 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))	changes negative TOS value into it's two's complement.
SHIFT	X1>0:(x2 x1 – x2<<x1) X1<0:(x2 x1 – x2>>x1)	If X1 >= 0: X2 (former TOS) is shifted by X1left, TOS then If X1 < 0: X2 (former TOS) is shifted by X1 right, TOS then
AND	(x2 x1 -- x1&x2)	bitwise AND replaces standard Forth "AND"
OR	(x2 x1 – x21|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 (one's complement)
+/-	(x -- -x) <two's complement of x>	change sign of TOS (two's complement)
*****	(factor factor -- factor*factor)	signed 32 bit multipication 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 perforned in "float" and reconverted to 32bit signed int. Small roundíng 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 (former and final TOS) is set to 1 (bitN 0..31)
BCLR	(x bitN -- xmodified)	bitN of x (former and final TOS) is cleared (set to 0) (bitN 0..31)
BTST	(x bitN -- bitvalue))	bitvalue = boolean value (=0 or 1) of bitN (x=former and final TOS)
RANDOM	(-- random)	a 32bit random number is generated with min.10 Hz 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 ("Variable Constant")	(x ---) IMMEDIATE	if a VARCON is called by name,the **actually stored value** is put on TOS Example : 789 VARCON CONNY → A VARCON named CONNY gets compiled with inital value 789. Read of VARCON simply CONNY ,S, e.g. **If read in code, their Header address is compiled,**no runtime operator
W changed Feb'25 before: 'REV'	(x ---) IMMEDIATE BACKGRND	replace (re-value) value of named VARCON with value of TOS (next use **and in prevoiusly compiled threads**). **Syntax: 234 W CONNY** May be applied from Background process for interactive control of thread flow (IF ., or WHILE) or modification of process parameters else.
ARRAY	(index --- value) (value –index ---)	32bit signed integer array,1000 entries(F042LP:200,L031LP:600).**No index 0** positive TOS=index: read value from ARRAY[index] to TOS negative TOS=index: write value (=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
.	(number ---)	TOS is sent as ASCII text using actual SYSTEM NUMBER base The content of TOS is sent as 32 bit signed integer.
.H	(number ---)	TOS is sent as ASCII text HEX formatted The content of TOS is sent as 32 bit **unsigned** integer.
.D	(number ---)	TOS is sent as ASCII text DECIMAL formatted The content of TOS is sent as 32 bit **signed** integer.

." ...."	(---)	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
DZ	(---) 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
HX	(---) 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 by 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

XNOP	(n ---)	may be used as a short blocking delay.variable short delay. A minimal C-code loopmis 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 backgound countdown of t*100ms is started. See WAIT
TIME	(--- 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 without MIDI PWM2 is not in motor version	(divisor ---) BACKGRND	starts CPU timer with peripheral bus clock, which is divided by a prescaler. "divisor" sets the prescaler rate to provide a **variable input frequency** for the Timer. This prescaler range is 1…65535 (16bit). **0 FREQ12 stops** the timer (and PWM if active, too) With prescaled input, the Timer produces a **constant overflow each 2000** prescaled clock pulses (**LowPwr versions each 100** clock pulses) This generates the **base frequency** of FREQ12, PWM1, PWM2 . This way the FREQ12 frequency can be modified in the range between: **24kHz down to ca. 0.4 Hz**. (**ST32F411**: 42kHz down to ca 0.5Hz) **Low Power versions**: 9.216kHz down to ca. 0.15 Hz (**48Mhz CPU clock**: 30kHz) **STM32L031**: 10.48 Khz down to ca 0.15 Hz If "divisor FREQ12"=max.FREQ12 is chosen, resulting Freq is ca. 1kHz, example: 24 PWM12 or Low Power: 9 FREQ12 sets Freq to ca 1kHz
PWM1 **STM32F042:** range only 0… 100	(high --) BACKGRND	The PWM1 signal output is available at PA6. (**STM32L031** lowPwr:PB1) If PWM1 is started when FREQ12 is not active, FREQ12 gets provisionally activated with ca. 1kHz overflow frequency. The HIGH percentage of PWM1 is (high/2000)*100 (LowPwr: high/100) 0 PWM1 sets PA6 permanently LOW, 2000 PWM1 (**LoPwr** 100 PWM1)or greater sets PA6 permanently HIGH **-1 PWM1 deactiavates** PWM output at PA6, but does **not** stop FREQ12
PWM2	(high --) BACKGRND	The PWM2 signal output is available at PA7. (STM32L031 lowPwr:PA0) STM23F042: PWM2 is only available as alternative to MIDI and MOTOR Details else same as PWM1

FREQ34 **(only G4xx standard) or FREQ3** **(G4xx lowPwr and Nucleo versions and F411 BlackPill)**	(divisor ---) BACKGRND	starts CPU timer with peripheral bus clock, which is divided by a prescaler. "divisor" sets the prescaler rate to provide a **variable input frequency** for the Timer. This prescaler range is 1…65535 (16bit). With prescaled input, the Timer produces a **constant overflow each 2000** prescaled clock pulses (**LowPwr versions each 100** clock pulses) This generates the **base frequency** of FREQ3, PWM3, (PWM4) . Details else same as FREQ12
PWM3	(high --) BACKGRND	The PWM3 signal output is available at PB3 (changed Jan.24) Details else same as PWM1
PWM4	(high --) BACKGRND	The PWM4 signal output is available at PA5 (changed Jan.24) Exclusively at **G4xx standard** version. Details else same as PWM1

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 Exclusively available at STM32G4xx, and STM32L476
DAO5	(value ---)	configures PA5 as 12 bit DAC right aligned and sets value Exclusively available at STM32G4xx, and STM32L476
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
RDIN	(portPin -- 0|1)	returns the digital level at port pin. Independent of actual configuration and user availabilty of this pin. Syntax: 0xAF RDIG returns level at PA15
RADC	(PinNo -- value)	reads 12 bit ADC level of **all CPUs** PinNo 0xA0..0xA7(short code 0…7) **G4xx** plus 0xB0(short 8). **F411,L476** plus 0xB0,0xB1(short code 8,9) **Nucleo** versions extra plus 0xC0…0xC5, no short code When called first, the A/D converter gets initalized. **Configure the pin as input HiZ before or make a dummy RADC for level stabilisation.** Once opened, the ADC stays active during the session (**takes current** !) **-1 RADC deactivates** the ADC, **saves remarkably current**
INI-SPI SPI is not in **STM32F042 MIDI** version	(mode bits clock ---)	intializes 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, MISO=PB4, MOSI=PA15 **STM32F042 TSSOP20**: SCK=PA1, MISO=PA0, MOSI=PA4 **all versions else**: SCK=PB3, MISO=PB4, MOSI=PB5
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 (except LowPower). 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 ITX.., IRX.., IADRXN. **STM32G4 LQFP32**: SDA=PB7, SCL=PB8, **else** SDA=PB9, SCL=PB8 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 !
ITX1	(byte --)	1 byte is transmitted to the addressed I2C slave
ITXN	(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
IRX1	(-- 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 .
IRXN	(n – b1 b2 b3 … bn)	a receive command is sent to the addressed I2C slave, which will return n bytes. TOS is the last received byte. It can be used for **sequential read from memory** chips like 24Cxx .
IADRXN	(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 **random or page read from serial memory devices**.and other more complex I2C peripherals
MOT1 or **MOT** "LowPower" versions: only MOT STM32F042: MOT is only in "motor" version	(speed --) BACKGRND	By default, **the motor driver is configured for"normal" commutated DC motor, e.g.** TB6612 power amp**:** **NSTP1(NSTP) = -1:** PA1/PA4(**G4xx Standard,Mini**), PA13/PA4(**G4xx Micro**), PA0,PA1(**else**) control a H-bridge. (pin groups are selected for best HW useability, modified Jan.24) **speed** controls motor speed (PWM1), negative speed reverses rotation. (if TB6612 is iused: PWMA and PWMB commonly) **control a stepper motor:** provides a two wire pulse sequence at the same pin group as used for DC motor (see above) **speed** is entered in milliseconds per phase change, max 16384. speed = 0 stops rotation. **Negative speed = counter clockw. rotation**. steps number is limited by value of "user" VARCON NSTP1(NSTP). NSTP1(NSTP) = 0: permanent rotation. For unipolar motors, the **inverted phase** and switch motor power off must be generated externally. See HwDiy manual Recommended **initial startup sequence for stepper motors**: First set to stepper mode: 0 W NSTP1. Next set PWM1 to max level. Then enter step duration in millieconds, e.g. 10 MOT1. Finally enter number of steps, e.g. 200 W NSTP1 (=1 turn if 1.8 deg/step) After rotation is finished, only number of steps has to be repeated.
MOT2	(speed --) BACKGRND	By default, **the motor driver is configured for a DC motor: NSTP2 = -1:** PB7,PB8 .(Nucleo boards PB8,PB9) control a H-bridge. **speed** controls motor speed (PWM2), negative speed reverses rotation. **control a stepper motor:** provides a two wire pulse sequence at the same pin group as used for DC motor (see above) Function else like MOT1.
MOT3	(speed --) BACKGRND	By default, **the motor driver is configured for a DC motor: NSTP3 = -1:** PB4, PA15 control a H-bridge. (modified Jan.24) **speed** controls motor speed (PWM3), negative speed reverses rotation. **control a stepper motor:** provides a two wire pulse sequence at the same pin group as used for DC motor (see above) Function else like MOT1.

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.

REGW	(value peripheral_regr_addr ---)	writes value into the addressed pheripheral register

REGR	(peripheral_reg_addr --- value)	reads the addressed pheripheral MCU register and returns value
REGBSET	(bitN peripheral_reg_addr ---)	SETS bitN of the addressed peripheral register(N=0..31)
REGBCLR	(bitN peripheral_reg_addr ---)	CLEARS(sets =0) bitN (N=0...31) of the the addressed peripheral register
REGBTST	(bitN peripheral_reg_addr -- 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 THENI
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. A**t 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
>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 greater 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, do<CASE, 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 organizise loops more C-style for easier learning and remembering

DO ... WHILE is functionally synonym with standard Forth BEGIN ... UNTIL
DO ... WHILE is functionally synonym with standard Forth BEGIN ... UNTIL, but inverted criterion
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 and CONTI on the Return stack.

The **end of all these kind of loops** is compiled as follows:
Behind the type specific runtime token (doUNTIL, doWHILE, doAGAIN, doLOOP),
first the loopback address gets compiled (back to behind DO+1 or FOR+1 code word) as parameter code word.

Next the runtime token for BREAK handling (Return Stack adjustment) is compiled automatically:
RP-1 (behind UNTIL, 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**:
BREAK terminates the loop unconditionally synonym with standard Forth LEAVE
ZBREAK checks TOS: if = 0,the loop is BREAKed, else execution continues linear
If ZBREAK is used several times in a loop, different criteria may be used.
CONTI performs a special check of the loop-end runtime token:
In a DO..UNTIL, DO....WHILE, DO....AGAIN loop**, jump is made directly to behind DO parameter word**
(effectively performs doAGAIN)
When used in FOR..LOOP, jump is made to doLOOP,
which updates and checks the loop index (2nd Return Stack entry).
If the loop end is not yet reached, code execution is looped back to behind doFOR parameter word.
Else execution continues linear behind doLoop via RP-4.
Internally CONTI uses the Return stack entry similarly as BREAK,
but copies th loopback address from the parameter word of doLOOP (stored at address (Return Stack-1)).

Operator specific descriptions:

DO	(-- addr) IMMEDIATE, COMPILEONLY	compiles doDO, starts compilation of a DO...UNTIL,WHILE,AGAIN loop. Essentially it provides a placeholder for the BREAK code address and sends its code address via stack to WHILE or AGAIN
UNTIL	(addr --) IMMEDIATE, COMPILEONLY	compiles doUNTIL, finishes compilation of a DO ... UNTIL loop Essentially it compiles its code address for BREAK behind doDO - and compiles the loopback code adress provided by DO one code word behind doUNTIL and compiles RP-1 two code words behind doUNTIL
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 adress provided by DO one code word behind doWHILE and compiles RP-1 two code words 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 adress provided by DO behind doAGAIN. and compiles RP-1 two code words behind doWHILE
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 adress provided by FOR into the code word behind doLOOP and compiles RP-4 next word behind
I	(-- loopIndex) COMPILEONLY	At runtime, it returns the actual loop index in a FOR...LOOP loop from the second ReturnStack entry on data stack TOS.
CONTI	(---) COMPILEONLY	Jump to (value stored at (Return Stack – 1)) FOR..LOOP:checks index
BREAK	(---) COMPILEONLY	Unconditional jump to value from Return Stack
ZBREAK	(---) COMPILEONLY	if (flag == 0) linear loop progression else jump to value on Return Stack.
RETURN	(---) COMPILEONLY	At runtime, it terminates a User Thread immediately (like "C" return). But it returns only one Return Stack level up. To leave a User Thread from inside a loop, RP-1 or RP-4 must be placed ahead of RETURN
RP-1	(---) COMPILEONLY	automatically compiled by UNTIL, WHILE and AGAIN At runtime it removes the top entry from the ReturnStack Compiled ahead of RETURN to return out of a loop ReturnStack level
RP-4	(---) COMPILEONLY	automatically compiled by LOOP. At runtime it removes the 4 top entries from the Return Stack Compiled ahead of RETURN to return out of a loop ReturnStack level

Runtime Operators (Runtime Primitives)

are compiled by other operators and control the runtime behavior.
All runtime operatores have the RUNTIME attribute.
No direct user handling or access, but essential for code interpreation with SEE and ST

doSTR	(---)	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 doSTR
doLIT	(-- n)	compiled by numbers. In the compiled thread, doLIT puts the content of the next ThreadCode entry (=number value) on TOS at runtime.
doLIT8	(-- value)	**used with:STM32F042,STM32F030,STM32L031** compiled for small numbers >= 255 to save Flash memory. Else runtime behaviour like doLIT.
doWVA (replaces standard Forth doVAR)	(-- address)	compiled by W In the compiled thread, the VARCON UserHeader[].toCOMPILE field address = value storage address is compiled behind doWVA. (STM32F042: UserHeader[] index) **At runtime, it writes TOS into this address**, no 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 An appropriately placed DROP may be needed.
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
do>CASE	(flag reference -- flag)	compiled by >CASE At runtime, it compares 2nd stack element “flag” with TOS ”reference”, if “**flag” is dreater than “reference”**, execution is performed until doELSE if present, or continues linear if “**flag” is less 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, do<CASE, do<CASE is effectively the same. At the end of the ELSE part, execution is continued linar.
doDO	(---)	compiled by DO. doDO puts the BREAK code address on ReturnStack
doUNTIL	(flag --)	compiled by UNTIL, doUNTIL takes the flag from TOS. If != 0, execution goes forward via RP-1 for linear progression If == 0, execution loops back to behind doDO
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
doFOR	(startIndex endInde 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. compare means: first the index is counted up or down. if the new index is equal or beyond the loop end index, loop terminates, i.e. execution goes forward via RP-4 for linear progression. else execution loops back to behind doFOR parameter

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	Sytax: SEE <User Token name> The thread is de-compiled as ASCII text. Additional info in parenthes: doWVA 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 128 bytes (32words) 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: new Jan.24: **CTRL_R triggers execution until next ;** (i.e. end of actual User Thread, avoids deep stepping through called functions) '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. ST may be compiled, but takes runtime if not needed.
RUN	(---) BACKGRND	**terminates single step mode**. May be compiled too, but takes runtime if not needed. **RUN is always default mode** after CForth start.
NOBACK	(---) BACKGRND	"No Background Priocess": Turns the execution of the background process OFF, saves about 10-30% runtime. Care must be taken when an endless loop without BREAK or similar is executed. Only exit then without reset: compile BACKOP within the loop.
BACKOP	(---)	Compiled in UserThreads, it starts background process unconditionally **Attention:** while CForth is in NOBACK 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	(---) IMMEDIATE	Syntax: REPLACE <supplier name> <target name> <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.** Works on new User Threads which are created after REPLACE, too. Useful for experimental replacement of old token by new ones without new compilation. All code newer than "target" is replaced. **STM32G4xx, L476: r**eplaces SRAM words "target code address" with "supplier code address plus bit19 set" in complete SRAM project code **STM32F042:** only UserHeader[] structure is modified, no code change Other code hacks are possible with kernel operators MEM and W
RESTORE	(---) IMMEDIATE	Syntax: RESTORE <original name> Due to special encoding of the UserHeader[] structure and bit19 set in the replaced code, the original token can be restored with REPLACE. All REPLACEd code with bit19 set is restored to original "target code address". Original "supplier" code is not changed.
: (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 behavíour, 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” deletes stored 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.
?	(---) IMMEDIATE BACKGRND	Sends a list of system parameters as: SRAM array start addresses, project number, AUTOEXE thread, Number Base, Baudrate, SPI, PWM
SAVE	(projNum ---) IMMEDIATE	Burns the actual SRAM user code as project #"projNum" into Flash You are asked to confirm your choice with upper case 'Y'. Max number of projects: (STM32G4xx, L476: 0 to 5 - DMX:0 to 2, STM32F411: 0 to 4 - DMX:0 to 3)). STM32F042: only 1 project in Flash. **CForth LowPwr**: STM32G4xx: 5 projects, STM32F042,L031: 2 projects STM32F030(4kB SRAM): 3 projects with max 63 UserToken + 1kB code Project#0 is "empty" in SRAM, not saved permanently at power-off. Flash can be rewritten up to ca.10.000 times (specified by STM). **Global parameters** (number base, baud rate, VID/PID) are stored in the Flash project and additionally in a separate Flash page. So an Empty Project uses the last saved globals (STM32F042,F030,L031: from Project#1) **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 less projects in Flash. Released Flash area is used for storage of 96 DMX "lighting scenes", shared by all projects
LOAD	(projNum ---) IMMEDIATE	Loads user stored project #"projNum" from Flash into SRAM. You are asked to confirm your choice with upper case 'Y'. **LOAD doesn't change the project number loaded at next PowerOn.** **Project#0 is "empty"** with default- or pre-stored global parameters **Projects #1-5 are loaded from Flash** **STM32F042 supports only 1 project in Flash and 1 Empty project.** If the Flash area of a **project is "virgin", an "empty"** project is loaded. **CForth-DMX special**: **the stored DMX levels are not reloaded** (for this purpose use operator X-LOAD, see below). Fadetime is loaded.

// and **(plus <SPACE>**		// and (plus <SPACE> initiated comments are supported They are catched 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) Termination of comment by) is not supported.

Following operators are not supported by unmodified Nucleo boards and CForth LowPower

VIDPID	(VID PID ---) IMMEDIATE	Specifiy the VID/PID used for USB communication. First VID then PID are entered as stack parameters each **in HEX, 4 digits with leading zeroes** New entered VID/PID is active after **stored in Flash** and system restart. **STM32F042**: can only be stored as Flash project, but will be used then together with EMPTY project after system reset. 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.

These operators are ONLY supported by versions with RS232 or MIDI,not by unmodified Nucleo boards:

BAUD **not** supported by unmodified Nucleo boards **else:** supported by STM32G4xx with RS-232 supported by STM32F042 all versions	Where	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 changes immediately (modified April2025) F042, F030, L031:save BAUD in Flash Project#1, EMPTY reloads this. **T**he actual **hardware for G4xx, L476, F042 and lowPower F030, L031 does not use an external USART TX inverter**. **Exception**: proposed PCBs for these firmware use external TX inverter: LQFP48 Black Pill compatible "Grey Pill" and STM32F042 "Mini" Firnware for STM32F411 **always implies an external inverter** If applicable, in some source code EXTERN_TXINVERT can be chosen
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 operated **as USB-MIDI interface** (USB MIDI works significantly different from virtual COM) **Special cases for STM32F042 "MINI, MICRO, NUCLEO**:(new April'25) type == 11: transparent data transfer USB-COM ←> USART-RS232 this is a 1:1 byte stream, differing from type 1 return to CForth: type 1x CTRL_T(dec20) at terminal type == 13: transparent data transfer USB-COM ←> USART-MIDI this is a 1:1 byte stream, differing from type 3 return to CForth: type 3x CTRL_T(dec20) at terminal **Simpified** 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**

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"

If serial I/O is in MIDI mode (see SPLIT), the support for **received MIDI messages** is organized as follows:
In a background process, received MIDI bytes are extracted from the receive buffer (USB or USART, depending on SPLIT config.), **packed to a single data word** and stored in a cyclic MIDI buffer (first in – first out). This operation is performed automatically by the terminal input handler and at any time if 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, 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 ".H".
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.

Configure STM32F042 PCBs as USB to MIDI/COM interfaces (new April 2024)

This option is available only for the small TSSOP20 MINI, MICRO and NUCLEO-USART" boards.
A jumper is placed on the programmer socket, checked only once during systems start.

--- **no jumper: CForth is started**. It is essential that at startup no externally connected peripherals pull down PA13 or PA14. After startup, this is not relevant anymore.

--- **jumper PA14 to GND (NUCLEO: J1 set):** board is started as bidirectional **USB-COM to USART-RS-232** interface.
USART baudrate as configured for CForth (default 115200 or Flash preset)

--- **jumper PA13 to GND:** board is started as bidirectional **USB-MIDI to USART based MIDI** interface. USART baudrate always 31250 baud = MIDI

--- **jumper PA14 and PA13 to GND:** board is started as bidirectional **USB-COM to USART based MIDI** interface. USART baudrate always 31250 baud = MIDI. This option may be useful to operate MIDI devices with byte-stream based software - like terminal e.g.

Special Operators of the CForth - DMX versions

The DMX command set is provided exclusively with selected CPUs and firmware packets.
The reserved DMX output pins are not available as peripherals

STM32G4xx Standard DMX:
I/O pins PA2 and PA0 are reserved for DMX

STM32G4xx Mini DMX:
I/O pins PB5 and PB6 are reserved for DMX

STM32G4xx LQFP48 DMX "Grey Pill":
I/O pins PB10 and PB0 are reserved for DMX

STM32G4xx LQFP48 DMX "Blue Pill compatible" (not published)**:**
I/O pins PB6 and PB0 are reserved for DMX

STM32F411 DMX (Black Pill and Nucleo STLINK_CUTOFF):
I/O pins PA2 and PB0 are reserved for DMX

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 unitil 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
XBUS new 2025	(0,1 --) BACKGRND	0 XBUS deactivates DMX bus (saves runtime during DMX lowLevel act.) 1 XBUS (default) DMXbus is active

Appendix: pin configurations of special features

SPI

STM32G4 (LQFP32,LQFP48)	SCK:PB3, MOSI:PB5, MISO:PB4
STM32G4LowPwr	SCK:PB3, MOSI:PB5, MISO:PB4
STM32F411	SCK:PB3, MOSI:PB5, MISO:PB4

STM32L476	SCK:PB3, MOSI:PB5, MISO:PB4
STM32F042 (TSSOP)	**SCK:PA0, MOSI:PA5, MISO:PA4**
STM32F042(Nucleo32)	**SCK:PA11,** MOSI:PB5, MISO:PB4
STM32F042(LQFP)	SCK:PB3, MOSI:PB5, MISO:PB4
STM32F042LowPwr (TSSOP)	**SCK:PA0 MOSI:PA5, MISO:PA4**
STM32F042LowPwr (LQFP)	SCK:PB3, MOSI:PB5, MISO:PB4
STM32L031 (TSSOP)	**SCK:PC15, MOSI:PA5, MISO:PA4**
STM32L031 (LQFP)	SCK:PB3, MOSI:PB5, MISO:PB4
STM32F030	SCK:PB3, MOSI:PB5, MISO:PB4

PWM

STM32G4	PWM1:PA6 PWM2:PA7 PWM3:PB3 (not@Nucleo)PWM4:PA5
STM32G4LowPwr	PWM1:PA6 PWM2:PA7 PWM3:PB3
STM32F411	PWM1:PA6 PWM2:PA7 PWM3:PB3
STM32L476	PWM1:PA6 PWM2:PA7 PWM3:PB3
STM32F042 (TSSOP)	PWM1:PA6 PWM2:PA7
STM32F042(LQFP)	PWM1:PA6 PWM2:PA7
STM32F042LowPwr (TSSOP)	PWM1:PA6 PWM2:PA7
STM32F042LowPwr (LQFP)	PWM1:PA6 PWM2:PA7
STM32L031 (TSSOP)	**PWM1:PA2 PWM2:PA0** (HW restriction)
STM32L031 (LQFP)	**PWM1:PA2 PWM2:PA0** (HW restriction)
STM32F030	PWM1:PA6 PWM2:PA7

MOT

STM32G4(Standard LQFP32)	**MOT1: PA3,PA1** MOT2:PB8, PB7 MOT3:PA15, PA4
STM32G4 (MiniDMX)	**MOT1:Micro:PA13,PA14 else:**PA0,PA1 MOT2:PB8, PB7
STM32G4(LQFP48 and Nucleo)	MOT1: PA0,PA1 MOT2:PB8, PB7 MOT3:PA15, PA4
STM32G4LowPwr	PA0, PA1
STM32F411	MOT1:PA0, PA1 **MOT2:PB8, PB9** MOT3:PA15, PA4
STM32L476	MOT1:PA0, PA1 **MOT2:PB8, PB9** MOT3:PA15, PA4
STM32F042 (TSSOP)	PA0, PA1
STM32F042(LQFP)	PA0, PA1
STM32F042LowPwr (TSSOP)	**PA7, PB1** (optimized for HW layout)
STM32F042LowPwr (LQFP)	PA0, PA1
STM32L031 (TSSOP))	PA0, PA1
STM32L031 (LQFP))	PA0, PA1
STM32F030	PA0, PA1

I2C

STM32G4	SCL:PB8, SDA:PB7
Nucleo STM32G4	SCL:PB8, SDA:PB7
STM32G4LoPo	SCL:PB8, SDA:PB7
STM32F411	**BlackPill:**SCL:PB8, SDA:PB7 **Nucleo:**SCL:PB8, **SDA:PB9** (optimized for HW layout)
STM32L476	SCL:PB8, **SDA:PB9** (optimized for HW layout)

DMX

STM32G4 (LQFP32)	Standard:PA2, PA0(Break sw.), MINI&Nucleo:PB6,PA15(Break sw.)
Black/GreyPill (LQFP48)	PA2, PB0(Break switch)

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