# **CForth** Operating Manual

 27 December 2020

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 Forth system, specifically to organize electronic control of art objects and installations or similar process control.

No attempt is made to comply with established Forth standards.

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

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

Instead of Forth "Words" the termini "**Kernel Operator**" and "**User Thread**" are used.
Data Stack, Return Stack and Thread Code are organized each as array of 32bit words

## Some CForth internals:

**Access to Kernel Operators** is organized in three synchronous arrays, which gives a small advantage compared with organisation in a structure of arrays.
One array contains the name words,
another one contains pointers to their code subroutines,
a third one contains their **attributes:**

**COMPILEONLY** can only be compiled in compiling mode
**IMMEDIATE** cannot be compiled, only be executed immediately even in compiling mode.
**RUNTIME** is a runtime operator which is compiled by another operator.and executed at runtime. Cannot be handled by the user directly. RUNTIME includes COMPILEONLY COMPILEONLY and IMMEDIATE are combined in some cases.
**Special is the BACKGROUND attribute**, which allows selected Operators (see Glossary) to be started in a simple background (or say “debugging”) process while a User Thread is executed. Good for manipulation of CONSTants and VARiables, for single step operation and break of endless loops. BACKGROUND and IMMEDIATE are combined in some cases. Attribute violation by the user causes an error message but does not abort the main process.

**Access to User Threads** is organized as an array of structures with following entries:
1. "name", name word, max 15 chars plus null termination
2. "toCompile" contains the absolute start address of the compiled thread in the Thread Code array or actual data value of CONST and VAR.
3. "attributes" contains the CONST or VAR attribute or the index of the compiled thread in the Thread Code array. Double storage of thread address and thread index has some advantages for token search. By means of Kernel Operators REV and VAL, CONST can be used similar like VAL. But CONST puts its value, VAR puts its address on stack when called.

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

The **Thread Code is organized** as follows:
The complete code is stored in a single array of 32 bit words. Each "User Thread" occupies a defined part within this array.

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

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

At any time, **execution can be switched to single step operation** with operator "ST" and return to normal execution is made with Operator "RUN". The Background Process can be switched off completely with Operator "NO-DBG", saves ca. 40% runtime. All this can be handled from the background process.
Vice versa: in "no-debug" mode, any **terminal input CTRL_D switches back** to RUN mode.

**About "token"**: each Kernel Operator is compiled in a User Code thread with its index in the array of Kernel Operator code pointers (max 8 bit). But each User Thread called by another User Thread is compiled in the calling User Code thread with its previously compiled absolute start address (i.e.SRAM address > 0x20000000, so easily distinguished from Kernel Operators)
It would have been possible to compile the start address of Kernel Operator subroutines directly, which would speed up the runtime a little bit. But I didn't because the actual solution makes the compiled code better user readable (Kernel Operator "MEM"). This concept implies some waste of project memory. The STM32G431 has plenty of this. The STM32F042 memory is rare anyway. Possible alternatives don't give a noticeable benefit.

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

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

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

As a new option (Dec.2020) some firmware versions are supplied without RS-232 interface, only with USB terminal. This releases some I/O pins and Flash code space (STM32F042).

## CForth "Projects":

In accordance with the memory design of the STM32G431KB processors, up to 6 "**Projects**" are supported. The versions for STM32L476 have the same memory configuration. Due to restricted memory, the STM32F042 supports only one project (one in Flash, experimental work in SRAM is possible independently). Actual CForth operation (interactive and compilation) is always done in SRAM. "Compilation into Flash" is not supported.

**By default Project #0, an "Empty Project"** is loaded with default parameters into SRAM. This installs two CONSTants P0, P1 and two VARiables V2, V3 **as user token**, to be helpful for interactive process control. FORGET them if not needed. 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 to 3) can be **loaded from Flash** to SRAM (operator "**LOAD** "). Existing code in SRAM is overwrittten 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.

Any project in SRAM can be **burnt into Flash** (operator "**SAVE** ") as Project #1 to 5 (with DMX only 1 to 3, STM32F042 only one), assigned Flash area is overwritten then. Project#0 can be saved too, CForth starts with an empty project 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**).

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 (same or another project number).

## Glossary of Operators:

Operator names written **red** are **not implemented for STM32F042**
Operator names written **green** are NOT implemented in versions **without RS-232 interface**, including unmodified Nucleo modules.

### Arithmetic Kernel Operators:

| | | |
|---|---|---|
| **DROP** | ( x1 x2 -- X1 ) | |
| **DUP** | ( x -- x x ) | |
| **2DUP** | ( x1 x2 -- x1 x2 x1 x2 ) | |
| **SWAP** | ( x1 x2 -- x2 x1 ) | |
| **2SWAP** | (x1 x2 x3 x4 –<br>x3 x4 x2 x1) | |
| **OVER** | ( x1 x2 -- x1 x2 x1 ) | |
| **PICK** | ( x-n ... X-2 x-1 x0 n --<br>x-n ... X-2 x-1 x0 x-n) | consume n and copy the n-th element of stack to the Top of Stack (TOS)<br>0 PICK is equivalent to DUP 1 PICK is equivalent to OVER |
| **ROT** | ( x1 x2 x3 -- x2 x3 x1 ) | rotate 3rd stack entry up |
| **-ROT** | ( x1 x2 x3 -- x3 x1 x2 ) | rotate TOS down |
| **+** | ( x1 x2 -- x1+x2 ) | signed 32bit addition |
| **1+** | ( x -- x+1 ) | add 1 to TOS |
| **-** | ( x1 x2 -- x1-x2 ) | signed 32 bitz subtraction |
| **1-** | ( x -- x-1 ) | subtract 1 from TOS |
| **ABS** | ( x -- abs(x) ) | |
| **<<** | ( x1 x2 -- x1<<x2) | shift range X2:0 ...31 |

| | | |
|---|---|---|
| **>>** | ( x1 x2 -- x1>>x2) | shift range X2:0 ...31<br>if X1(bit31) is set, "ONE"s are shifted in ! (preserve minus sign) |
| & | ( x1 x2 -- x1&x2) | bitwise AND replaces standard Forth "AND" |
| **\|** | ( x1 x2 -- x1\|x2) | bitwise OR replaces standard Forth "OR" |
| **XOR** | ( x1 x2 -- x1\|x2) | bitwise XOR |
| **NOT** | ( x ---xmodified )<br><one's complement of x> | bitwise 0/1 inversion |
| **NEG** | ( x -- -x )<br><two's complement of x> | change sign of TOS |
| ***** | ( factor factor -- factor*factor ) | signed 32 bit multipication<br>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<br>if (factor * factor > 32bit), operation is internally perforned in "float"<br>and reconverted to 32bit signed int. Small roundíng error possible.<br>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).<br>The remainder has always the same sign as the dividend |
| **BSET** | ( x bit# -- xmodified ) | bit# of x is set to 1 (bit# 0..31) |
| **BCLR** | ( x bit# -- xmodified ) | bit# of x is cleared (set to 0) (bit# 0..31)´ |
| **BTST** | ( x bit# -- bitvalue)) | bitvalue = boolean value (=0 or 1) of bit# |
| **RANDOM** | ( -- random) | a 32bit random number is generated with 1kHz update frequency |

## Comparison Operators

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

## Memory Operators

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

## Terminal Operators

| | | |
|---|---|---|
| **TERM?** | ( --- number ) | the actual number of bytes in the terminal input buffer, = 0 if none<br>USB has priority over USART |
| **KEY** | ( --- byte ) | execution blocks until a terminal byte is received, returned on TOS |
| **EMIT** | ( byte --- ) | the lowest 8 bit of TOS are sent via terminal as raw byte |
| **P** | ( number --- ) | TOS is sent as ASCII text using actual SYSTEM NUMBER base<br>The content of TOS is sent as 32 bit signed integer. |
| **PH** | ( number --- ) | TOS is sent as ASCII text HEX formatted<br>The content of TOS is sent as 32 bit unsiged integer. |
| **P" .... "** | ( --- ) | send the text between " ...." as byte stream via terminal<br>Between P" and the first text letter MUST be a SPACE which is not sent<br>Any "non-printable" byte can be sent if formatted as \hiNibbleLowNibble<br>Examples: \1B sends ESC, \22 sends " \5C sends \ \00 sends nullByte |
| **%D** | ( --- ) | set system number base DECIMAL In this case hex numbers can be entered with preceeding 0x. Even 0x–… is accepted: 0x-FFFFFFFF = 1 |
| **%X** | ( --- ) | set number base HEXADECIMAL<br>The number base can be changed by %X,%D token in compile mode, but it is reset to the system number base at compile end (;) |

## Serial I/O Operators

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

With the SPLIT operator (see below), USB and USART can get different functions.
By default, both work parallel as terminal.
But each of them can be configured as serial data I/O.

| | | |
|---|---|---|
| **RX?** | ( --- count ) | the actual count of bytes in the serial I/O input buffer, = 0 if none.<br>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 is released** with terminal input CTRL_B |
| **TX** | ( byte --- ) | the lowest 8 bit of TOS are sent via serial I/O as raw byte |
| **NTX** | ( b-n ... b-1 b0 n ) | the n stack items below n are sent via serial I/O<br>Only the lowest 8bit of each stack item are sent as raw bytes. |
| **TX" ...."** | ( --- ) | send the text between " ...." as byte stream via terminal. Remark see P" |

## 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 ms is started.<br>**blocking is finshed** immediately by terminal input CTRL_B (bin code 2) |
| **DS** | ( t --- ) | blocking countdown of t*100ms (deci-seconds) is started (see CTRL_B) |
| **TIX** | ( t --- ) | nonblocking backgound countdown of t*100ms is started. See WAIT |
| **WAIT** | ( --- t ) | returns remaining countdown of TIX in 100ms units<br>may be used to construct simple timelines |
| **FREQ**<br>**STM32F042:**<br>FREQ,PWM,<br>TIME,PULS<br>only in version<br>without RS-232 | (divisor --- ) BACKGRND | starts STM32 Timer3 with prescaler input clock frequency 48MHz.<br>"divisor" sets a prescaler to get the input clock frequency of Timer3. With this prescaled input, Timer 3 produces a constant overflow each 2000 prescaled clock pulses. Both together results in the **base frequency** of PWM1,2, TIME and PULS . E.g: 24 FREQ will cause a 1kHz base freq.<br>The prescaler has a 16bit range, so the base frequency can be modified in the range of ca 24kHz and ca.0.3Hz. |
| **PWM1** | ( high -- ) BACKGRND | The PWM1 signal output is available at PA6.<br>If PWM1 is started when FREQ is not active, FREQ gets provisionally activated with 1kHz overflow frequency.<br>The HIGH percentage of PWM1 is (high/2000)*100<br>0 PWM1 will set PA6 permanently LOW,<br>2000 PWM1 (or greater) will set PA6 permanently HIGH<br>-1 PWM1 will deactiavate 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 FREQ is active, TIME is upcounted each Timer3 overflow.<br>At moderate base frequencies, this can be used as "System Time" base. |
| **PULS** | ( -- number) | When FREQ is active, PULS is upcounted each Timer3 overflow,<br>but reset to zero after each read access. So PULSE can be used as a non-blocking action trigger with additional information if the readout came slower than one base frequency period. BLOCK and WAIT may be easier to use, but PULS is a low-cost by-feature. |

## Peripheral Operators

Each of the following operators configures the pin each time it is called at cost of additional runtime, but handling is quite easy this way.
User is responsible not to connect an output with another output

| | | |
|---|---|---|
| **OH** | ( portPin --- ) | sets addressed port pin as push/pull OUTPUT HIGH<br>Syntax: AF OH sets PA15 B3 OH sets PB3<br>portPin is entered as HEXbyte. A0 is decimal 160, AF is decimal 175 etc |
| **OL** | ( portPin --- ) | sets addressed port pin as push/pull OUTPUT LOW<br>Syntax: AF OL sets PA15 B3 OL sets PB3 |
| **DAO4** | ( value --- ) | configures PA4 as 12 bit DAC right aligned and sets value |
| **DAO5** | ( value --- ) | configures PA5 as 12 bit DAC right aligned and sets value |
| **IPU** | ( portPin --- ) | sets addressed port pin as INPUT with PULL-UP<br>Syntax: AF IPU configures PA15 B3 IPU configures PB3<br>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<br>Syntax: AF IPD configures PA15 B3 IPD configures PB3 |
| **IZ** | ( portPin --- ) | sets addressed port pin as INPUT with HIGH IMPEDANCE<br>Syntax: AF IZ configures PA15 B3 IZ configures PB3 |
| **RDIG** | ( portPin -- 0\|1 ) | returns the digital level at port pin<br>independent of actual configuration of this pin |
| **RANA** | ( portPin -- value ) | configures the A/D converter and reads 12 bit level<br>Supported pins: PA0...PA7 if not used by hardware else<br>Precision of STM32 A/D converters is bad, so each level is read 8 times and the result is averaged.<br>Once opened the A/D converter stays active during the session, but it is connected to a port only when a read is requested.<br>No matter when the port gets configured else. |
| **INI-SPI**<br>SPI is not in<br>**STM32F042**<br>**MIDI** version | ( mode bits clock --- ) | intializes an SPI master with bits=4 to 32 bit length<br>Except 16bit, the SPI is implemented very badly by STM<br>So an alternative was hand-coded with up to ca. 2MHz SPI clock, almost double fast with 144MHz system clock, somewhat slower on STM32F042.<br>**/CS must be handled separately** (if necessary) by user with any pin else. This way, multi-slave configurations are possible.<br>Motorola modes 0,1,2,3 are supported.<br>(sometimes referenced as (0,0) (1,0) (0,1) (1,1) )<br>clock = 0 is max. speed, else clock inserts a heuristic slow down.<br>Default: mode 0, 8 bit, clock 50 (ca 50kHz), max.clock= 0xFFFF=65535<br>**Used I/O pins:**STM32G431(Mini)SCK=PB3, MOSI=PA15, MISO=PB4<br>Else STM32G431 and STM32L476: SCK=PB3, MOSI=PB5, MISO=PB4<br>STM32F042: SCK=PA1, MOSI=PA4, MISO=PA0 (changed Dec2020) |
| **SPI** | ( TxWord -- RxWord ) | If not initialized explicitly before, SPI gets configured with default<br>or with user values stored in Flash project |
| **INI-I2C** | (SlaveAddr -- ) | Initializes I2C. addr is the slave address which is used in all subsequent I2C transmit and receive operations. Only 7 bit addressing is supported. Standard is 100kHz clock. If bit8 of addr is set (say: 256 added to SlaveAddr), then I2C will work with 400kHz clock.<br>Slave address and bus speed can be changed at any time between two |

| | | |
|---|---|---|
| | | transmit and receive commands.<br>STM32G431(Veroboard and Mini): SDA=PB7, SCL=PB8<br>Nucleo Boards: SDA=PB9, SCL=PB8 (wiring is easier then) |
| **I-TX** | (b1 b2 … bn n -- ) | n lower stack bytes are transmitted via I2C in the order b1 first, bn last |
| **I-RX1** | ( -- byte) | a receive command is sent to the slave, which will return 1 byte.<br>This command is useful for simple I/O expansions like PCF8574 or sequential read from memory chips like 24Cxx . |
| **I-RXA** | (RegAddr n –<br>b1 b2 … bn ) | a receive command which sends one device specific register address byte to the slave and orders a return of n bytes. This command is useful for more complex I2C peripherals like PCF8591 or random or page read from serial memory devices. |

Following operators manipulate MCU peripheral register directly, good knowledge of the STM32 Reference Manual is necessary.
Don't confuse these operators with W, R, BSET, BCLR, BTST! Both sets use different address spaces and context.

| | | |
|---|---|---|
| **REGWRITE** | (peripheral_regr_addr value -- ) | writes value into the addressed pheripheral register |
| **REGREAD** | (peripheral_reg_addr -- value) | reads the addressed pheripheral MCU register<br>and returns value |
| **REGBSET** | (peripheral_reg_addr bitNo -- ) | SETS bitNo of the addressed peripheral register |
| **REGBCLR** | (peripheral_reg_addr bitNo -- ) | CLEARS(sets =0) bitNo<br>of the the addressed peripheral register |
| **REGBTST** | (peripheral_reg_addr bitNo -- bitLevel) | returns level of bitNo of the addressed peripheral reg. |

## Structuring Operators

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

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

DO ... WHILE is functionally synonym with standard BEGIN ... UNTIL
DO ... AGAIN is functionally synonym with standard BEGIN ... AGAIN
DO ..ZBREAK .. AGAIN is functionally synonym with standard BEGIN..WHILE..REPEAT
FOR ... LOOP is functionally synonym with standard DO ... LOOP

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

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

Next the runtime token for BREAK handling is compiled (Return Stack adjustment):
RP-1 (behind WHILE and AGAIN) deletes 1 entry, RP-4 (behind LOOP) deletes 4 entries.
When the loop is terminated "normally", execution passes there too, and the BREAK address (plus 3 loop parameters of FOR...LOOP) is removed from ReturnStack.

The code body of any loop **may contain one or more of these runtime tokens**:
doBREAK terminates the loop unconditionally synonym with standard LEAVE
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 )<br>IMMEDIATE, COMPILEONLY | compiles doDO<br>starts compilation of a DO ... WHILE or DO ... AGAIN loop.<br>Essentially it provides a placeholder for the BREAK code address and sends its code address via stack to WHILE or AGAIN |
| **WHILE** | ( addr -- )<br>IMMEDIATE, COMPILEONLY | compiles doWHILE, finishes compilation of a DO ... WHILE loop<br>Essentially it compiles its code address for BREAK behind doDO - and compiles the loopback code adress provided by DO behind doWHILE. |
| **AGAIN** | ( addr -- )<br>IMMEDIATE, COMPILEONLY | compiles doAGAIN, finishes compilation of a DO ... AGAIN loop<br>It inserts its code address for BREAK compilation behind doDO - and compiles the loopback code adress provided by DO behind doAGAIN. |
| **FOR** | ( -- addr )<br>IMMEDIATE, COMPILEONLY | compiles doFOR, starts compilation of a FOR ... LOOP loop<br>The runtime behaviour is very similar to "C" programming:<br>doFOR expects 3 DataStack entries: startIndex, endIndex, Increment<br>Increment can be positive or negative and can be greater than 1.<br>For runtime details of the FOR...LOOP loop see doFOR and doLOOP.<br>Essentially FOR provides a placeholder for the BREAK code address and sends its code address via stack to LOOP. |
| **LOOP** | ( addr -- )<br>IMMEDIATE, COMPILEONLY | compiles doLOOP, finishes compilation of a FOR ... LOOP loop<br>Essentially it inserts its code address for BREAK compilation behind doFOR and compiles the loopback code adress provided by FOR behind doLOOP |

| | | |
|---|---|---|
| **I** | ( -- loopIndex )<br>COMPILEONLY | At runtime, it copies the actual loop index in a FOR...LOOP loop<br>from the second ReturnStack entry to data stack TOS. |
| **CONTI** | ( --- )<br>IMMEDIATE, COMPILEONLY | compiles doCONTI |
| **BREAK** | ( --- )<br>IMMEDIATE, COMPILEONLY | compiles doBREAK |
| **ZBREAK** | ( --- )<br>IMMEDIATE, COMPILEONLY | compiles doZBRK |
| **RETURN** | ( --- ) COMPILEONLY | At runtime, it terminates a User Thread immediately (like "C" return) |

## Runtime Operators

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

| | | |
|---|---|---|
| **doSTRING** | ( --- ) | compiled by P"<br>In the compiled thread it is followed by the bytes to be transmitted.<br>Differing from standard Forth, the byte sequence is 0 terminated.<br>Byte 00 is internally coded as 0x100. |
| **doIOSTR** | ( --- ) | compiled by TX", remark see doSTRING |
| **doLIT** | ( -- n ) | compiled by numbers.<br>In the compiled thread, doLIT puts the content of the next ThreadCode entry (=number value) on TOS at runtime. |
| **doVALUE**<br>(replaces standard Forth doCONST) | ( -- value ) | compiled by CONST (and by VAL)<br>In the compiled thread, the CONST value **address** is compiled behind doCONST,<br>but at runtime it puts the CONSTant value on TOS.<br>Referencing the CONST by address is a contribute to SEE and REV |
| **doADDR**<br>(replaces standard Forth doCONST) | ( -- address ) | compiled by VAR<br>In the compiled thread, the VAR value address is compiled behind doVAR.<br>At runtime, it puts this address on TOS. |
| **doIF** | ( flag -- ) | compiled by IF<br>takes a flag from TOS,<br>if == 0, a jump is initiated: behind doELSE if present,<br>or to code address compiled by THEN<br>if != 0, execution is performed until doELSE if present<br>or continues linear |
| **doCASE** | ( flag reference -- flag) | compiled by CASE<br>At runtime, it compares 2nd stack element “flag” with TOS”reference”,<br>if **equal**, execution is performed until doELSE if present,<br>or continues linear<br>if **unequal**, a jump is initiated: behind doELSE if present,<br>or to code address compiled by THEN<br>**Attention:** at runtime TOS is deleted, “flag” is proceeded as new TOS |
| **do<CASE** | ( flag reference -- flag) | compiled by <CASE<br>At runtime, it compares 2nd stack element “flag” with TOS ”reference”,<br>if “**flag” is less than “reference”**, execution is performed until doELSE if present, or continues linear<br>if “**flag” is greater than or equal ”reference”**, a jump is initiated: behind doELSE if present, or to code address compiled by THEN<br>**Attention:** at runtime TOS is deleted, “flag” is proceeded as new TOS |
| **doELSE** | ( --- ) | compiled by ELSE<br>if the flag of doIF != 0, doELSE manages a jump behind the code of the<br>behind the code of the ELSE part (compiled by THEN).<br>if the flag of doIF == 0, execution is continued behind doELSE.<br>Behaviour after doCASE or do<CASE effectively the same, see above.<br>At the end of the ELSE part, execution is continued linar. |

| | | |
|---|---|---|
| **doDO** | ( --- ) | compiled by DO. doDO puts the BREAK code address on ReturnStack |
| **doWHILE** | ( flag -- ) | compiled by WHILE, doWHILE takes the flag from TOS.<br>If == 0, execution goes forward via RP-1 for linear progression<br>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<br>At runtime it removes the top entry from the ReturnStack |
| **doFOR** | ( startIndex endInde<br>increment --- ) | compiled by FOR<br>3 items are copied from the DataStack to the ReturnStack:<br>TOS is the increment/decrement of loop index per loop<br>NOS is the loop end index for loop termination<br>3rd DataStack entry -> 2nd ReturnStack entry:start index=loop index<br>Furthermore the BREAK code address is put on top of the ReturnStack. |
| **doLOOP** | ( --- ) | compiled by LOOP<br>counts the 2nd ReturnStack entry up or down (depends on 4th entry), compares the 2nd entry with the 3rd one.<br>If compare is in range, execution loops back to behind doFOR<br>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. Uncoditional jump to (value from Return Stack – 2) |
| **doBREAK** | ( --- ) | compiled by BREAK. Uncoditional jump to value from Return Stack |
| **doZBRK** | ( flag -- ) | compiled by ZBREAK<br>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 [...].<br>For debugging, .S may be compiled, but should be removed in final code |
| **FS** | ( --- ) IMMEDIATE | "Flush Stack", deletes all items on the Data Stack.<br>Most times used to remove garbage from the Data Stack. |
| **OPS** | ( --- )<br>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** | ( --- )<br>IMMEDIATE BACKGRND | Sends a list of all actually compiled User Threads.<br>First the UserThread structure index of the thread is sent, next the name followed by the ThreadCode address, where the thread is compiled.<br>For CONSTants, the acual value is sent, too<br>For VARiables, the actual value and its storage address of is sent. |
| **SEE** | ( --- )<br>IMMEDIATE BACKGRND | Sytax: SEE <User Thread name><br>The compiled thread is sent as ASCII text. Additional info in parenthes:<br>doLIT and doVALUE: numerical value.<br>doADDR: value storage address<br>others: jump target code address. See op MEM.<br>Because all SRAM of STM32 processors is in the range 0x2000xxxx, **only the LOW 16BIT** are sent via terminal in HEX without leading 0x... |
| **MEM** | ( addr(HEX) -- )<br>IMMEDIATE BACKGRND | sends 48 words of SRAM as ASCII table starting from addr.<br>Because SRAM of the STM32 processors is in the range 0x2000xxxx, only the LOW 16BIT (4 hex nibbles) are entered.<br>MEM may be compiled, but should be removed in final code |
| **ST** | ( --- ) BACKGRND | **enables single step execution of User Threads**<br>actual information is shown similar to the output of SEE.<br>**Display**:token name,output, Data Stack, if context relevant:Return Stack<br>At thread runtime, TAB from terminal triggers execution of next token<br>Special cases:<br>'ESC' terminates thread execution immediately (escapes endless loop)<br>CTRL_B terminates long BLOCK times<br>To release a hanging BLOCK or KEY, first push TAB, next CTRL_B or the intended key. 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. |
| **NO-DBG** | ( --- ) BACKGRND | "No Debug": Turns the execution of the Background OFF, saves about 40% runtime.<br>Care must be taken when an endless loop without BREAK or similar is executed. Only exit then without reset: see DEBUG. |
| **DEBUG** | ( --- ) | Is only effective in "No Debug" mode, then it **switches unconditionally back** to RUN mode. May be compiled corresponding with NO_DBG<br>Independently, "No Debug" is leaved at any time with terminal CTRL_D. |
| **FORGET** | ( --- ) IMMEDIATE | Syntax: FORGET <User Thread name><br>The named User Thread and all newer ones are deleted |
| **ABORT** | ( --- ) IMMEDIATE | Clears all stacks and resets all system parameters.<br>Internally executed by the system after error messages. |
| **REPLACE**<br>NOT in<br>**STM32F042**<br>**MIDI** version | ( --- ) IMMEDIATE | Syntax: REPLACE <supplier name> <target name><br>Useful for experimental replacement of old token by new ones without new compilation. Only code between supplier and target is replaced.<br><supplier name> is always the first search hit from TOP of userToken[]<br><target name> is always the first hit searched from TOP of userToken[].<br>If supplier and target have the same name, the search for target is continued down towards bottom of userToken[].<br>This PROCEDURE WORKS ONLY WELL if supplier is the first hit and target is the first or second hit of search from TOP of userToken !! |
| **RESTORE**<br>NOT in<br>**STM32F042**<br>**MIDI** version | ( --- ) IMMEDIATE | Syntax: RESTORE <original name><br>Due to special encoding of the userToken[] structure,<br>the original token can be restored after REPLACE.<br>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.<br>Numbers have higher priority than thread names.<br>So avoid thread names which could be numbers (like DAC, ADC, BEE)<br>A User Thread with the same name as a Kernel Operator overrides the Kernel Operator (will be compiled instead in future threads).<br>Experimentally this can be used productive<br>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><br>The specified user thread is automatically executed at system start.<br>To get it available at system start, it must be stored to Flash and the system must start from Flash. "AUTOEXE 0" doesnt start a User Thread.<br>Stored AUTOEXE may cause **deadlock**, only resolvable by Flash erase!<br>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** | ( --- )<br>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.<br>Global settings are stored project independent in a separate Flash page.<br>The project number (0 to 5)(DMX:0 to 3) is asked after SAVE is entered.<br>STM32F042 saves only one project in Flash, doesn't ask a number<br>As a precaution, operation must be confirmed with upper case 'Y'.<br>Flash can be rewritten up to ca.10.000 times (specified by STM).<br>**Special CForth-DMX**: **the stored DMX levels are not saved**<br>(for this purpose use operator X-SAVE, see below). Fadetime is saved<br>**CForth-DMX** supports only project no. 0-3. Remaining 32kB Flash is used for storage of 64 DMX "lighting scenes", shared by all projects |

| | | |
|---|---|---|
| **LOAD** | ( --- ) IMMEDIATE | Loads a user stored "Project" from Flash.<br>The project number (0 to 5) (DMX:0 to 3) is asked after LOAD is entered.<br>As a precaution, operation must be confirmed with upper case 'Y'.<br>**Project#0 is "empty"** with default parameters, overwrites existing code!<br>**Projects #1-5 are loaded from Flash**<br>**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**.<br>If the Flash area of a Project is "virgin", an empty project is loaded.<br>Global parameters are loaded project independent from Project number.<br>**Special CForth-DMX**: **the stored DMX levels are not reloaded**<br>(for this purpose use operator X-LOAD, see below). Fadetime is loaded. |
| **EMPTY** | ( --- ) IMMEDIATE | **STM32F042 only**. Starts an Empty Project Replaces LOAD(project #0) |
| **//** and **(<SPACE>** | | // and (<SPACE> initiated comments are supported<br>They are catched and deleted directly in the "query()" terminal input handler, so they are not explicitly listed as operators.<br>Any comment is valid UNTO and ONLY UNTIL LINE END (default=CR)<br>) style comment termination is not supported. |

**Following operators are not supported by unmodified Nucleo boards**

| | | |
|---|---|---|
| **VID** and **PID** | ( --- ) IMMEDIATE | Specifiy the VID/PID used for USB communication.<br>To avoid errors, enter VID or PID in HEX,4 digits w. 0x + leading zeroes<br>Be careful not to shoot your USB access (appropriate Windows .inf file !)<br>By default, the VID/PID and Windows .inf provided by ST-LINK is used.<br>For non-evaluation and public use, a legal VID/PID must be configured.<br>New entered VID/PID is active after stored in Flash and system restart. |

**Following operators are NOT supported by versions without RS-232 and by unmodified Nucleo boards:**

| | | |
|---|---|---|
| **BAUD** | ( baudRate -- )<br>IMMEDIATE | Specifies the USART baudRate (default=115200).<br>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.<br>11=115200, 57=57600, 38=38400, 31=31250(MIDI), 19=19200, 96=9600<br>New entered baudRate is not active before stored in Flash and system restart. **Exception**: if SPLIT = 1 or 3, baudRate changes immediately.<br>F042:save BAUD in Flash project, reload, start EMPTY with user BAUD<br>**New** as of 27Dec20: the actual **hardware for G431 and F042 does not use an external TX inverter**. Level is inverted at USART register level.<br>For hardware with external inverter (old HW version or MAX232 etc) enter BAUD parameter with **leading 1**.<br>Example: 157 or 0x157 selects 57600 Baud with external inverter |
| **CHANNEL** | ( channel -- )<br>IMMEDIATE | Enter the MIDI channel for received MIDI messages (1...16), default=1.<br>**Special case 0 CHANNEL** accepts any MIDI channel.<br>Not relevant for transmitted MIDI messages<br>The new entered MIDI channel is not active before the project is stored in Flash and system is restarted (or project reLOADed). |
| **SPLIT** | ( type -- ) IMMEDIATE | Specifies the organisation of terminal versus serial I/O (default = 0)<br>type == 0: USB and USART work parallel as terminal<br>type == 1: USB is terminal, USART is serial I/O as COM port<br>type == 2: USART is terminal, USB is serial I/O as virtual COM port<br>type == 3: USB is terminal, USART is serial MIDI I/O<br>(for transmission, no difference between MIDI and COM,<br>but **received data are handled differently**. Any baud rate works in MIDI mode, so MIDI can be served via RS-232 interface, too.)<br>type == 4: USART is terminal, USB is operated **as USB-MIDI interface**<br>(USB MIDI works significantly different from virtual COM) |

## Features of the MIDI interface: (improved Dec2020)

**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 can be freely done by the user. Check and extraction of the different MIDI bytes is usually performed with AND and shift operations.

## Special Operators of the CForth - DMX versions

The DMX command set is an expansion provided with the G431 Standard DMX and the G431 Mini version.

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 in 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 "slot") for subsequent actions.<br>Lowest slot number is 1, max slot is 512 |
| **XV** | ( level --- ) BACKGRND | sets the DMX level at the currently selected DMX channel (0…255) |
| **XSET** | ( slot level --- ) | combination of XV and XS: set DMX level at DMX slot.<br>The **DMX channel remains set**<br>for subsequent actions like **XN**, **X=**, **XR,** etc |
| **XN** | ( level --- ) | increments the selected DMX channel andsets the DMX level there |
| **X=** | ( number --- ) | copies the DMX level of the selected DMX channel<br>to the following 'number' channels. |
| **XFT** | ( Fadetime --- ) | sets the the time during a changed DMX level is faded<br>from the actual value to the new final value.<br>Fadetime is entered in 1/10 sec steps (0 … 127=12.7 seconds)<br>Once triggered, the fade continues unitil finished,<br>even when settings of other channels are changed |
| **X+"** | ( --- ) | increases the DMX level of the selected DMX channel by one |
| **X-** | ( --- ) | decreases the DMX level of the selected DMX channel by one |
| **X0** | ( --- ) BACKGRND | switches all DMX channel to zero level immediately (no fade) |
| **XCYC** | ( cycle -- ) | set DMX cycle =number of transmitted channels (24 <= cycle <= 512) |
| **X-BOOT** | ( scene -- ) IMMEDIATE | saves "scene" (0…64) permanently to be loaded at system start |
| **X-SAVE** | ( scene -- ) IMMEDIATE | saves the active DMX transmit buffer as "lighting scene" (1…64)<br>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...64) 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…64)<br>**starting from DMX channel 'slot' of 'scene" stored in Flash**<br>into the active DMX transmit buffer<br>**starting there from the actually selected DMX channel** (e.g. XS)<br>This may be used to copy short lighting patterns to other lamps |
| **XR** | (number --- )<br>IMMEDIATE BACKGRND | sends 'number' DMX levels ASCII text formatted to the terminal,<br>starting from the actually selected DMX channel |

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