

# **P2XCForth and CCForth for P2 and RP Pico**

## Contents

1	Overview.....	3
2	Comparison to Taqoz.....	6
3	Setup for P2.....	9
4	Setup for RPI Pico.....	10
5	How it works.....	11
5.1	Stacks.....	11
5.2	Coding and the Inner Interpreter of CCForth.....	11
5.3	Coding and the Inner Interpreter of P2XCForth.....	11
5.4	The Dictionary.....	12
5.5	The Outer Interpreter.....	13
5.6	The User Area.....	13
6	Concept for Online- Help.....	15
7	Defining a new Primitive Word.....	16
8	Startup Definitions of Compound Words – Block File System.....	17
9	Compound Words in First Blocks 2...5.....	18
10	Some Useful Words.....	19
11	Multitasking.....	20
11.1	Defining a User Area.....	20
11.2	A single Background Task in the Main COG 0.....	20
11.3	A Task in another COG.....	20
11.4	Cooperative Tasks with “pause” in main COG 0.....	20
12	Local Variables.....	22
12.1	Using Standard Names.....	22
12.2	Using Named Locals.....	22
13	Macros.....	23
14	The Editor FED for CCForth and XCForth.....	24
15	Some Remarks about XCForth.....	25
16	ToDo.....	27
17	Wordlist of CCForth.....	28

# 1 Overview

So why would we need another Forth?

There have been some main **reasons** for me to create CCForth and then P2X-CCForth:

- We already have got Taqoz, which is a very impressive Forth, hand-optimized for P2, compact and fast. But it lives in it's own world, completely separate from what has evolved over the last years. For example, it might be interesting to combine a Forth with an USB-driver or HDMI output, or any other code from Obex. Forth can be seen as a fast scripting language. And/or a mighty communication protocol.
- It would be nice to have the same Forth, or nearly the same one for different microcontrollers. This CCForth is written in C, so we can hope, that it will be not too difficult to port it to other 32bit controllers. To see if this is true, PicoCCForth.ino is included.
- This Forth provides some structural means, which make the implementation of some advanced features easy: cooperative multitasking, a help system, local variables, a nice source code editor.
- Last not least it has been said, that at some point, every Forth friend will want to do their own. So, this is my first take. From Scratch. 😊

After CCForth was somewhat completed, it became clear, that it's benefits come with the price of significantly lower speed in comparison to Taqoz. So the question was, if the special XBYTE mechanism of P2 could be used to go the other direction and make a Forth for P2, which is even faster than Taqoz. XBYTE is a mechanism for an assembler program sitting in COG or LUT RAM. It uses the fast STREAMER cache to read an instruction code byte from HUB RAM, looks up where to find the right assembler routine in COG or LUT and executes it. Then this cycle is repeated. XBYTE is fast. I think, that during the development of P2, XBYTE came relatively late, when Taqoz and it's word-code mechanism had already been fixed. CCForth uses a 32bit-code, and the inner interpreter is executed from C so the XBYTE code mechanism is a different world.

When I had a more close look into XBYTE, I discovered, that using Flex there could be a way to combine a small XBYTE machine executing a core wordset with a mechanism to execute words written in C. So the hybrid P2XCCForth was born.

## Core Features of P2CCForth and of P2XCCForth:

- 32bit.
- Can access all HUB memory of P2
- Strings are handled C-style, ending with NULL.
- Can run "pause" style cooperative multitasking.
- Can run words in other cogs on P2.
- Uses a FAT file system on SD card to host a block file system. Also boots from SD card via \_BOOT\_P2.BIX.
- Uses large 32kByte blocks. The tiny 1k blocks of the past have been a main reason, that code was squeezed with scarce comments and was therefore

not well readable. With the immense storage room of a SD card, there is no longer any need for this.

- Provides named local variables. Too many swap rot dup -rot drops have been the second reason for bad readability, so local variables come handy and still keep the code reentrant.
- Uses the dictionary as main database. There is a field for each word, which can hold a link to the source for fast access to the definition. Also there is a field which holds information about the type of the word. For example “repeat” is tagged as ISLOOP. These loop words are printed in red.
- You are free to add new words not only on the Forth side but also as “primitives”, written in C or it’s inline assembler, or even calling SPIN routines.

### **P2XCForth:**

- In P2XCForth there is a second level of inner interpreter, the XBYTE machine, written in assembler. This is loaded into the first half of LUT, where it stays. XBYTE is started from the inner interpreter and it will run a small Forth of core words. It has the capabilities of conditional and unconditional branching, looping and calling other Forth words. For example the Byte Sieve Benchmark is executed completely within this inner-inner XBYTE interpreter. However, when it encounters a TRAP instruction, the XBYTE machine stores it’s status, quits to the calling inner interpreter, written in C, which will execute the TRAP-code instruction (another byte code) and then restart the XBYTE machine to go on.
- So with P2XCForth we have 3 types of instructions. XBYTE-words, TRAP-words and compound words, written in Forth. Actually most TRAP-words are identical to their counterparts in CCForth. The encoding of words in the dictionary has to be different though.
- TRAP-words are written in FlexC, can use inline assembler and can also call routines written in SPIN2 or FlexBasic.

There is a lot of heritage in P2CCForth. I have read lots of papers about Forth and have been inspired by a lot of good ideas. A main influence has been Dr. Tings books describing F83 or his own cForth. [Forth-books - Dr.Tings Collection](#)<sup>15</sup> They have encouraged me to do the project.

This text is neither intended to be a booklet to learn Forth nor to be a glossary of P2CCForth. To learn Forth just grab a tutorial or a book. For example, have a look at [Simple Forth](#). A glossary is available in the system online.

This text contains a comparison to Taqoz. Also it intends to describe the internals of CCForth and P2XCForth. This might be interesting, even if you don’t intend to use P2\*CForth.

At the time of writing, P2CCForth has a much lower number of words in comparison to Taqoz. I wanted to have a strong backbone of core elements. As a first application, I have begun to port FED, my editor from Taqoz to P2CCForth. At the moment FED begins to be sufficiently attractive to be used.

When I had a working version of P2CCForth with FED, I wanted to try to port it to a different processor. I choose RPI Pico, because it makes sense to use a simpler lower cost board sometimes. I also choose to implement this in the Arduino environment, to bring this more easily over to other machines later. During the port I learned, that the ARM M0+ processor can only access aligned 32bit longs. So I had to add alignment and also I had to go to a 32 bit Flags fields from 8 bits only.

I want to express, that I am very thankful for all the open sources and for all the help I get! Especially for Kiss boards, for FlexC, for the helpful people at the Parallax Forum!

Have Fun!

Christof Eberspächer

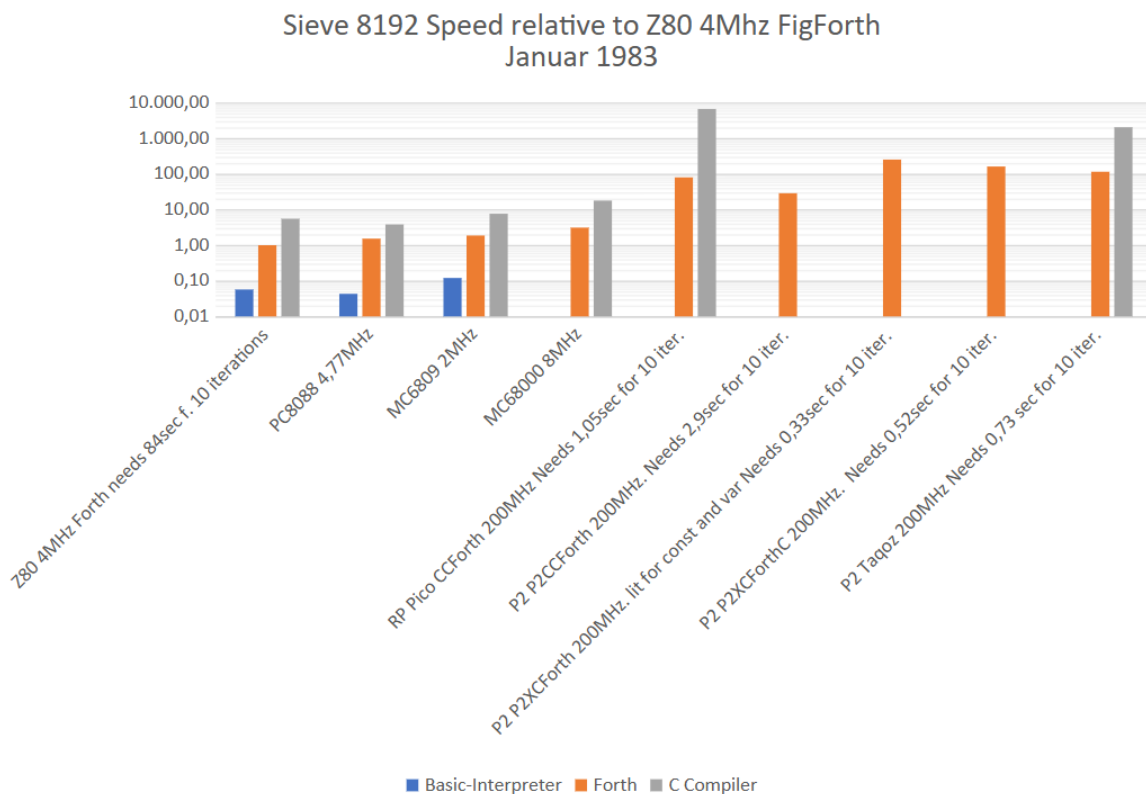
## 2 Comparison to Taqoz

Taqoz is a hand-optimized Forth written entirely in assembler and taking into account the specific architecture of P2. It is written to be fast and also to be compact.

	Taqoz	(P2)CCForth, <b>P2XCForth</b>
Written in	assembler	C; <b>XBYTE: assembler, TRAPs: C</b>
Usage of COG and LUT	Forth Registers and Core words reside in COG Stacks reside in LUT	Forth registers are register variables in COG FlexC uses COG for Fast Cache of little loops. TOS and NOS and the stackpointers are register variables. Stacks are in HUB Code is in HUB <b>XBYTE machine in LUT</b>
pause-style co-operative multitasking	As stacks reside in LUT and many registers in COG are used, it is slow to swap tasks. Therefore “pause” is not provided.	Included. For each task there is a “user area” in HUB, which holds the stacks and to which the Forth registers are saved, when the task is swapped.
Forth codes	Taqoz uses 16bit word codes. This leads to compact code. Code can also include some data. For example small literals can be included in a code word and also short relative jumps. This leads to restrictions of jump distances. Because 16bit codes are used as addresses, a paging mechanism has to be used.	Code is 32bit and straight forward. “Primitives” written in C are coded as positive numbers. Negative numbers indicate compound words. The inner interpreter is simple. <b>XBYTE: 8bit</b> <b>TRAP: 8bit</b> <b>compound words start with XBYTE wcall.</b>
Usage of SD-card	A simplified FAT is given. It can hold the bootfile and give access to other files which must have consecutive sectors. An opened file can be accessed as virtual storage.	FAT is fully compatible. It holds the bootfile and a single file named “blocks.blk” which holds all the blocks. Blocks are large to avoid frequent writes.

One disadvantage of P2CCForth in comparison to Taqoz is, that it is about 3.8 times slower than Taqoz. The following picture shows a comparison of the Byte sieve benchmark. Why do we compare to these historic machines? The reason is,

that you can see, that while P2CCForth is well slower than Taqoz or FlexC, it is also blazing fast in comparison to Forths of the time, when Forth was popular and considered fast.



You may conclude, that working with P2CCForth is not spoilt because of speed problems. 😊

P2XCForth beats Taqoz in this test, which does not contain any TRAPs in it's loops. The faster variant does not use the constant "size" and the field "flags" directly but here their addresses are compiled as literals.

It seems, that FlexC can take full advantage in this comparison of caching the innermost loop. The diagram also shows, that the old claim "nearly as fast as assembler" was never true, while a huge advantage against interpreted BASIC is visible. There have been two advantages against compilers: Until Ram for ram-disks became affordable, you had to wait much longer (minutes) to compile. And until Turbo Pascal, compilers have been very expensive.

The Forth benchmark code for Sieve is shown in the next picture.

```

\ Block 10 Prime Benchmark
forget size

8192 constant size ( original: 8190 1024 )
variable flags    size allot

: do-prime \ calculate primes Byte benchmark one iteration
  cnt@
  flags size 1 fill ( set array )
  0 ( 0 count )
  size 0 do
    flags i + c@ if
      i dup + 3 +
      ( dup . )
      dup i +
      begin dup size <
        while
          0 over flags + c!
          over +
          repeat
        drop drop 1+
      then
    loop
    .
  cnt@ swap - dup . 200 / .
;
( do-prime )

\ do-prime 1899 51165904 255829us 0.26s f. 1 Durchlauf
\ €

```

Figure 1 Byte Sieve Benchmark. P2CCForth coloured listing.



### 3 Setup for P2

P2CCForth is compiled with FlexC, from 7.1.0.

I use a P2 kiss board, which has a 25MHz resonator, with SD card and led at P57.

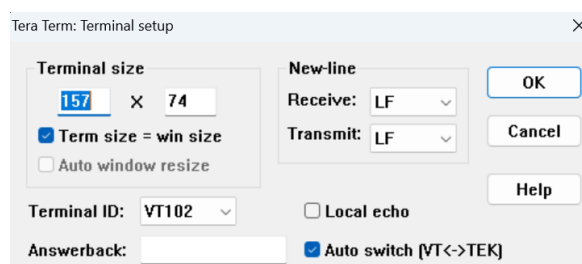
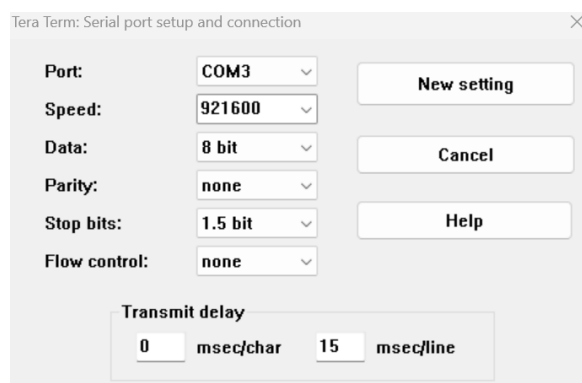
SD card is FAT32 and contains file block.blk. To get started, there must be any file with this name. This file will then be extended, when you flush blocks.

The SD card also holds \_BOOT\_P2.BIX . Unfortunately at the moment the booting does only work after power cycle.

PC Tera Term is used as Terminal, LF is used to mark line end.

Words are written in lower case. For longer word names I like to structure them like "doLineUpdate". At the moment P2CCForth is fully case sensitive.

Before FlexC 7.1.0, settings HAVE to be set to 1 stopbit, otherwise the serial input routine of FlexC loses sync and destroys chars, if you try to paste text.



To build up the blocks, we first copy the contents of block2.fth to P2. Then we use mkBlocks to initialise the empty blocks. Then "2 gu" which starts gettext for block 2. Depending on your version of Teraterm you will have to signal the end of the text pressing ESC. GlosBasA.txt goes into block 1.

In block 5 the right pin number has to be set for the blinking led of your board.

## 4 Setup for RPI Pico

The setup for RPI Pico is very similar to the one with P2. So here we focus on differences. The source file has to be renamed to \*.ino and to be compiled with the Arduino environment for RPI Pico with the option 1MB Sketch+1MB FS to give room for the file system.

**Raspberry Pi Pico/RP2040/RP2350** by Earle F. Philhower, III

4.5.1 installed

Boards included in this package: iLabs Challenger 2350 BConnect, Newsan Archi, Raspberry Pi Pico 2, SparkFun IoT Node LoRaWAN, Pimoroni PGA2350, Seeed XIAO RP2040, Generic RP2350, Amken Revelop, DFRobot Beetle RP2040, Seeed XIAO RP2350, Generic RP2040, Amken BunnyBoard, Viyalab Mizu RP2040, Adafruit Feather...

[More info](#)

```
#define pico
```

```
//#define P2
```

It turns out, that LittleFS in the flash chip seems to have difficulties to handle a big file for all the blocks with seek commands. Therefore there is one file for each block. Block size of 32k is big, if you have only 1MB of space available. We only have a low number of blocks here.

As serial connection we use the builtin USB serial with Teraterm.

After compilation and upload we use the “format” command to clear the file system.

For the time being only one core is used in RPI Pico. But you still have Cooperative Multitasking!

Be aware, that RPI pico needs aligned addresses for 32bit accesses like @ or ! ! It will simply crash otherwise.

## 5 How it works

COG 0 is used as main COG. COG1 is used to buffer the serial console terminal input in a 4kB ring buffer. This allows to copy and past a block of text in TeraTerm.

### 5.1 Stacks

There are three stacks in P2CCForth.

The parameter stack holds data.

The return stack holds return addresses for subroutine calls. Do..loop also uses the return stack and it can be used as a temporary storage.

Third stack is the auxiliary stack, which is used for frames of 5 local variables a,b,c,d,e. With loc and endloc the frame is moved by 5 longs.

### 5.2 Coding and the Inner Interpreter of CCForth

If a word is written in C as a function, it is called a “primitive”. Each primitive has got it’s code number, defined in an enum statement.

If a word is built from primitives, it is called a compound word. It’s code is the negated address of it’s code field.

A compound word is a row of primitive-codes and compound-address-codes ending with “exit”.

The Inner Interpreter is the Forth virtual processor. It has an instruction pointer IP which points to the next code position. It’s contents is read into W. If W is negative, then we know, that  $\text{abs}(W)$  is the code field address of a compound word. So we push IP to the return stack and load  $\text{IP}=\text{abs}(W)$ .

If W is positive a long SWITCH CASE structure decides, which primitive routine must be executed. FlexC compiles this to a fast indirect jump.

If the primitive word “exit” is found, then IP is popped from the return stack. This way execution will continue at the calling word.

### 5.3 Coding and the Inner Interpreter of P2XCForth

The interpreter of P2XCForth consists of two levels: The Inner Interpreter and the XBYTE machine.

The XBYTE loop written in assembler and loaded permanently in LUT memory loads, interpretes and executes XBYTE-Instructions (8bit). If it encounters a TRAP-XBYTE \$00 instruction, it loads the next byte as TRAP-code (8bit), preserves it’s status and returns to the Inner Interpreter. The Inner Interpreter, written in C, interprets und executes TRAP-codes given to it by the XBYTE loop. After each TRAP, the XBYTE loop is resumed. Only exeption is a zero TRAP-code. In this case the Inner Interpreter returns to the Outer Interpreter.

Examples:

	Word example	XBYTE	
Simple XBYTE	1+	\$05	XBYTE will execute «1+»
literal	lit	\$03, \$12345678	XBYTE will push the number \$12345678 onto the parameter stack
Call a word	wcall	\$26, \$aaaaaaaa	XBYTE will save the IP on the return-stack and set IP=\$aaaaaaaa, the code-field address of the word
Exit from word	wexit	\$25	pop(IP) from return stack, if not zero. If zero then TRAP text to end the interpreters.
Trap-code	emit	\$00=TRAP, \$02=trap-code	XBYTE will return the trapcode \$02 for «emit» to the Inner Interpreter, which executes it

## 5.4 The Dictionary

The dictionary is the single central database of P2CCForth. While in Taqoz we have two structures, here we only have one. The database is made as a linked list of entries.

The entry for each word has the following structure in ascending order:

- 1 Long 32bit **Link Field** points to previous link field
- 1 Long **Source Field** contains Blocknumber + Lineposition
- 1 Long, LSB: 8bit **Flags**
  - 1 bit 128 immediate flag
  - lower Bits type for color
    - enum { ISNORMAL, ISLOOP, ISSTRUCT, ISDATA, ISIO, ISASS };
- 5 Longs = 19+1 zero bytes **namefield**
- Start of **Codefield**
- **Parameter Field** variable length, the Parameter Field can contain program code or data or both. For CCForth Codefield and Parameterfield are a sequence of longs. In XCForth they are a mixture of bytes and longs.

To be able to use the same XBYTE instruction for constants, variables and fields created by create, in P2XCForth these use the same structure in the Parameter Field. The data address can be fetched with “dataddr” at an offset of 6 bytes.

A fixed length for the name field was chosen for several reasons: With this we do not need to calculate the code field address if we know the linkfield address. The second reason is, that named local variables can be implemented rather easily: The name of a local must be defined, when the compilation of the routine, where it shall be used, has already begun. The new variable name can be patched instead of the standard name of the variable. So the compiler can then find the new name immediately.

Originally for P2 the flags field was only 8 bits. To have aligned addresses for Arm M0+, this was expanded to 32 bit.

## **5.5 The Outer Interpreter**

The outer interpreter translates text input into Forth codes searching the dictionary from the newest entry to the oldest. If the outer interpreter is not in compile state, then the code found is executed directly. If it is in compile state, normal codes are written into the dictionary forming a new word. Only words, tagged as IMMEDIATE, will be executed directly in compile state. If a word is not found, it's text will be translated to a number. Hex numbers start with \$ . The text can come from the serial connection to a terminal or from a text buffer. The compiler state is held in the COG register variable status.

## **5.6 The User Area**

Together with COG Register Variables, user areas are used to keep tasks separate from each other. Every task has it's own 1kByte HUB ram space. The stacks reside all the time in this user area. COG registers are saved in the user area, when tasks are swapped via “pause”. You need to have a separate user area for both types of tasks: Running in a separate COG or running as cooperative task in the main COG. The COG Register variable UStart, which is accessible via UP@ and UP!, points to the start of the area of the active task.

- 0 Task-Link links to next USER Area, used by “pause”
- 1 taskState, 1 for ready, 0 for suspended
- 2 EABORT (register)
- 3 status (register)
- 4 SP (register)
- 5 RP (register)
- 6 AuxP (register)
- 7 IP (register)
- 8 W (register)
- 9 23 User Registers

- 9 variable for switch....case...break
- 32 64 Stack
- 98 64 Rack
- 162 50 Aux Stack for locals
- 212 End

The name “user” area stems from a time, when Forth systems have been used for multi users.

## 6 Concept for Online- Help

When a Forth system grows, I find it difficult to remember all the words. Therefore, a concept for a help function at your fingertips is nice to have. It is some work to maintain a glossary, so it would be nice to generate it from the actual sources.

The concept for Online Help of P2CCForth consists of the following elements:

1. **First line of a word definition:** When source code is written, the first line of a word definition starting with : directly after LF contains the name of the word, a stack diagram and a comment. This can be used as the relevant information for a glossary. For “primitive” words, written in C, in block 1 information is given in lines of the same format.
2. **A source field in the directory** can be filled with the block number and the position in the block, where to quickly find the first line of the definition.
3. **Different help words** can look up -via dictionary- and display the first line of a word definition or the complete definition.

scanDefs

will fill the source fields in the dictionary from the open block.

This information can be used then:

gwords	print glossary for the words in the dictionary.
prtDef ( < word> -- )	print first line of definition of a specific word.
prtSource ( < word> -- )	print source until ; in color

^W will in the editor FED display the glossary entry of the word under the cursor.

^G can be used to jump to the word's definition, if this is in the open block.

## 7 Defining a new Primitive Word

Primitives are words, written in C. To include a new one, P2\*CForth has to be compiled newly.

The new word has to be introduced at more than one places:

1. A code and a name for it has to be defined. This is done in the ENUM statement at the beginning of the source file. For P2XCForth in the TRAPs. The order there is arbitrary. Some names like “while” cannot be used, because FlexC will become confused.
2. A function void \_name(void) has to be written, which will perform the desired action. It's header is defined via the macro PRIMI, when the word name can be used to name the function. For words like “+!” the macro PRIMS is used. Actually PRIMI and PRIMS do two things, besides forming the header of a C function they also define the name of the word.
3. In the inner interpreter void inner(void) the macro CAS(name) will insert a case ... brake structure for the new primitive.
4. The routine void compPrimitives() will build up the dictionary using the macro COP(name,flags).

If a word is not implemented for a machine, an empty dummy word is given instead.



## **8      Startup Definitions of Compound Words – Block File System**

As the outer interpreter can evaluate text not only from the terminal, but also from a buffer of chars, this feature can be used to form the first compound words from a `const char builtinWords[]=".....";`

This is used to construct the block file system.

`bBuf ( -- address )` give the address of the (single) block buffer

`load ( blocknumber -- )` evaluate block

`block ( blocknr -- buffer_address )` flush if updated and then load new block into `bBuf`

`update` mark block buffer to be written

`flush` write block back if updated

## 9      **Compound Words in First Blocks 2...5**

At the moment the blocks have the following structure:

1 \ block 1 glossary of builtins glosbasA.txt <https://esp32forth.forth2020.org/documents>

2 \ block 2 Blocks words

3 \ block 3 Named locals, switch...case...break

4 \ Block 4 Help

5 \ Block 5 Cooperative Multitasking

7 \ Block 7 Fed1A readblock, saveblock, print

8 \ Block 8 Fed4A.fth

9 \ Block 9 Fed5A

10 \ Block 10 Prime Benchmark

2 load will compile blocks 2...5 and then load the editor FED from blocks 7...9.

## 10 Some Useful Words

see <name>	decompile a word
dump4 ( address -- )	dump 4 lines (16 longs)
evaluate ( string_address -- )	evaluate words in string
: mystring s" This is Text" ;	define a string constant, will leave address
print\$ ( address -- )	print null terminated string
conkey ( -- char )	get char from console terminal, 0 for none
words, cword, lwords	show contents of dictionary
colBuf ( -- )	display bBuf in color
index ( -- )	display first lines of blocks
gettext ( -- )	sample text from Console into bBuf, end with ESC

To use gettext, first load the old block and clear it "15 block drop wipe gettext" .  
After ESC: "update flush"

scanSource will fill the source fields in the dictionary from the first blocks, where to find the source. This information can be used then:

gwords	print glossary of first lines.
prtDef ( < word> -- )	print first line of definition
prtSource ( < word> -- )	print source until ; in color

' word trace!	Set the IP limit to activate tracing, (use singletask first!)
untrace	to switch off tracing

## 11 Multitasking

### 11.1 Defining a User Area

Cooperative Tasks and Tasks in own COGs both need their own User Area.

### 11.2 A single Background Task in the Main COG 0

While the outer interpreter is waiting for a line input ending with lf, it can do one polled task in background. This has to be a state machine which is called and completely done. The mechanism is the same as in Taqoz. We write the code-fieldaddress of the task into a variable called background.

```
57 constant ledPin
: simpleBlink
  getms 9 >> 1 and ledPin pin! ;
' simpleBlink background !
```

To stop it: 0 background !

### 11.3 A Task in another COG

COG 0 is used as main COG, COG1 is used to buffer the serial console terminal input. Here we use a separate COG 2 to run an endless task.

```
57 constant ledPin
: simpleBlink
  getms 9 >> 1 and ledPin pin! ;

: blinkCog begin simpleBlink again ;
create user1 1000 allot
2 user1 ' blinkCog cogstart \ ( COGnr userArea instruction -- )
```

For the time being, for RPI pico only one core is supported.

### 11.4 Cooperative Tasks with “pause” in main COG 0

The mechanism of cooperative tasks brings the advantage, that the programmer sets the point in time, when a task switch can occur. This can be relevant, if shared resources are to be used.

For a cooperative task, we need to have a dedicated user area. Create an endless task, which has at least one pause command. Then this task has to be inserted in the linked list of tasks. From this point, the task will be activated in a round robin scheme, when another task reaches it's pause command.

The pause mechanism can be inserted as background task, with “multitask”. Unlike a simple background task, cooperative multitasking preserves the state of that task, as it will go on from the position after any pause, it had reached. Swapping the task needs less than 10  $\mu$ s.

```
57 constant ledPin
create blinkerT 1024 allot \ create user area
500 value bFreq#

: blinkTask \ example for multitasker
  begin
    getms bFreq# / 1 and ledPin pin!
    pause
  again
;

blinkerT ' blinkTask addTask \ insert task into linked list
multitask
```

## 12 Local Variables

### 12.1 Using Standard Names

There are 5 local variables a, b, c, d, e, which are stored in a frame on the auxiliary stack.

a> copy the contents of a onto the parameter stack

>a move the contents of TOS to a, consuming TOS

+>a add TOS to a, consuming TOS

Normally at the beginning of word definition we need to prepare space for the locals

loc move stack frame for locals

loc\_a move stack frame for locals with one parameter, which will be moved from TOS to a.

At the end of the routine we must release the space and move back the stack frame using "endloc".

### 12.2 Using Named Locals

The same 5 local variables can have individual names.

The brace starts the definition of named variables. The ones before the comma will be filled from the parameter stack. The ones following the comma will not be filled. Between -- and the closing brace, text is comment.

The semicolon takes care to execute endloc, but if you want to exit otherwise, you have to insert it manually to move back the stack frame.

To access the local variable "name#", 3 different names are created. 1. "name#", 2. "to\_name#" and 3. "+to\_name#".

```
: calcCheck { last# , check# -- endvalue } \ checksum of first
bytes
  0 to_check# \ clear checksum
  begin
    last# c@
    +to_check# \ add byte to check#
    -1 +to_last# \ count down
  last# 0= until
  check# \ bring the sum to TOS
; forgetLocals \ end word and forget the local-names.
100 calcCheck .
```

## 13 Macros

A macro inserts text into the outer interpreter. This can be used during compilation, if the macro is marked as immediate. The compiler will then inline the text of the macro into the new definition. The following example will be faster because no jump or return/exit has to be executed. Especially useful have been macros for the definition of “case” and “break”.

```
: >= s" < 0=" eval$ ; immediate \ defining a macro
```

```
: mtst \ test for the macro
  0
  begin
    1+ dup .
    dup 5 >= until \ using the macro
    drop
;
0 0 0 0 > mtst
mtst 1 2 3 4 5
0 0 0 0 > see mtst
see mtst: 1bacd
          1baea      11  lit
                        0
          1baf2      4e  1+
          1baf6       8  dup
          1bafa       e  .
          1bafe       8  dup
          1bb02      11  lit
                        5
          1bb0a      55  <
          1bb0e      57  0=
          1bb12      52  0branch
                        -36
          1bb1a       5  drop
          1bb1e       2  exit
```

## 14 The Editor FED for CCForth and XCForth

FED is a modified port from FED for Taqoz. See Blocks 7...9. Some key elements:

- Works with 32kB blocks as fixed file size
- Uses it's own fedBuf\$ to hold the text in RAM
- Has fixed line length of 99+LF, therefore 326 lines per file.
- Is somewhat line orientated. ^X and ^V work for whole lines. Inserting chars will not spill over into next line.
- Uses the dictionary as fast database for coloured printing. Unknown words are yellow.
- Has syntax highlight.
- No mouse, but ^G will offer all word-definitions to Goto.
- Is intended to be used with Teraterm with a large Terminal Window, see chapter Setup. Copy and Paste of Teraterm can be used as long as the 4kB input buffer of P2CCForth is big enough.

### Starting fed:

eFed ( blocknr -- ) \ start fed with block, line lengths will be expanded first, sets fedBlock#

nFed ( blocknr -- ) \ start fed with blocknumber, sets fedBlock#

fed \ forth editor restarts editing the block in fedBlock#

^G fill source fields from actual block file, which has to be loaded first, then display line numbers of words to Goto. ESC to leave.

^K Mark a line

^N Jump to 4 marks, set with ^K, in round robin fashion.

^W display glossary entry of word under cursor. Displays first lines of word definitions. Uses source fields and therefore might need ^G first.

^R will shut off line numbers and right margins. Good for copy and paste.

^X cut line into cut-buffer

^V insert line from cut-buffer above actual line

^Q Quit editor, will ask, if file shall be written to SD

After leaving FED, reload will load fedBlock#.



## 15 Some Remarks about XCForth

At the time of writing P2XCForth is still quite new to me. The code of the XBYTE machine is far from polished but seems to work. I do not like the concept of instruction skipping in the XBYTE mechanism, because I think it makes the source code rather obscure.

I was astonished to find, that it is possible to load permanently a XBYTE machine into a COG, that is also executing FlexC or SPIN2. The machine together with it's code table is loaded into the first 256 longs of LUT, where it can be found by the calling routine. This space is reserved in FlexProp for cog-code routines, so these cannot be used in parallel. 256 longs is not too much, at the moment 188 longs are used for 43 core words. Enough for example to run the Sieve Benchmark completely in XBYTE without any TRAPs. I am wondering, what I should squeeze into the XBYTE machine. Value-type variables? Local variables? ?

The XBYTE machine is written in a SPIN2 file. Unfortunately the assembler there does only support a small number of register variables PR0....PR7. So the rest of the register variables are only available for the Inner Interpreter and not for the XBYTE machine.

The XBYTE machine gets it's speed from 3 elements:

- It runs from LUT memory, so the other COGs do not interfere memory access. – The general rule for speed of P2 is to avoid HUB access.
- It uses the micro-cache of the Streamer to read the XBYTE-codes (and inline data) from HUB memory
- The XBYTE mechanism does in hardware a lookup of the XBYTE-code to start the right routine.

For TRAP instructions the XBYTE machine has to be stalled and restarted and the micro-cache has to be refilled. So TRAP instructions come with much overhead. However when a routine for a TRAP instruction is longer, then the benefits of FlexC come into action like optimised compiled code and automatic caching of loops in COG memory.

Some timing examples:

	Time needed
Fastest XBYTE like «1+»	8 cycles
Stack operation: dup, drop	16 cycles
Empty loop: 1000 0 do loop	72 cycles per loop
Call and exit an empty word	70 cycles
Trap «j»	167 cycles
Trap multiply «*»	280 cycles

## 16    **ToDo**

- At the moment there is no does> .
- Some more IO
- Find bugs
- ...

## 17 Wordlist of CCForth

```
0 0 0 0 > gwords
```

```
gwords
```

```
prt2000          9 prt2000

eFed             9 eFed ( blocknr -- ) \ start fed with block, will be expanded first, sets
                fedBlock#

nFed            9 nFed ( blocknr -- ) \ start fed with blocknumber, sets fedBlock#

fed            9 fed \ forth editor starts editing the block in fedBlock#

list           9 list ( block -- ) \ list block in color

reload         9 reload \ load fedBlock#

findLastLine   9 findLastLine \ find last line of fedBuf$ to maxLines#

fedKey         9 fedKey \ react to key#

sedesc         9 sedesc \ ESC combinations

esc5b         9 esc5b ( code -- ) \ ^[[n cursor movement

a>A           9 a>A ( lower -- upper ) \ convert case of char

insCutLine     9 insCutLine \ insert a line from cutBuf$

cutLine        9 cutLine \ delete actual line into cutBuf$

cutBuf$

insBlankLine   9 insBlankLine \ above

newBlankLine   9 newBlankLine \ under cursor

delChar        9 delChar ( -- ) \ delete char

bsChar         9 bsChar ( -- ) \ backspace

insChar        9 insChar ( -- ) \ insert key#

key#

lineUpdate     9 lineUpdate \ mark to be updated

doLineUpdate   9 doLineUpdate \ print actual line newly

centerAddr     9 centerAddr ( addr ) \ if near borders center on screen

Forth_Editor5

prtCurGlos     8 prtCurGlos { , addr# } \ print glossary entry of word under cursor

fedFillDefs    8 fedFillDefs \ fill source field in dict from fedBuf$ and show defs

getPosNum      8 getPosNum { , k# num# -- number } \ get a positive number from keyboard
                0 for none

fedWords       8 fedWords loc \ print words from dict for this file

setHold        8 setHold \ set marker and also blockMarker#
```

```

jumpHold          8 jumpHold \ jump to next marker round robin

centerAddr1       8 centerAddr1 ( addr ) \ center line on screen

cur2Add           8 cur2Add ( -- addr ) \ cursor to addresse

findEnd           8 findEnd ( -- ) \ actCol# to last char of line

addrRange         8 addrRange ( addr -- addr ) \ limit to text buffer

Forth_Editor4

listFile          7 listFile \ mark screen update to be done

doListFile        7 doListFile { , i# } \ print the whole sceen newly

listLine          7 listLine ( line -- ) \ number relative to startList#

.decs             7 .decs { n# dig# -- } \ print n# right justified

prtLineHigh       7 prtLineHigh loc_a ( linestart$ -- ) \ display colored until CR

prtComment2       7 prtComment2 loc_a ( pointer -- new pointer ) \ prt comments green

fedExpand         7 fedExpand ( block -- ) { , inch# col# line# p# } \ read text into bBuf
                  and expand into fedBuf$

fillLine          7 fillLine \ fill the actual line in fedBuf$ to lWidth# with BLANKS

fedBlock          7 fedBlock ( block -- addr ) \ load directly into fedBuf$

fedUpdate         7 fedUpdate \ mark as updated

fedFlush          7 fedFlush \ write back block if updated

updateLiFlag

key#

listFiFlag

actFedBlock

fedUpdated

fedBlock#

blockMarker#

actCol#

maxLines#

actLine#

lastLine#

startList#

iHold#

setHold#

holdBuf

marginsFlag#

fedHeight#

fedStartLine#

```

```

lwidth#

fedBuf$

magenta          7 magenta 5 fcolor ; \ set color

erline           7 erline \ Clear entire line ^[[2K

nload            7 nload ( block -- ) \ force new load from SD card

gul              7 gul ( block -- ) \ gettext update load

Forth_Editor

counterTask      5 counterTask \ example for multitasker

cnt1

blinkTask        5 blinkTask \ example for multitasker

bFreq#

blinkerT

ledPin

unlinkTask       5 unlinkTask { task# , link# } \ remove Task from link list

prtTasks         5 prtTasks { , link# -- } \ print Tasks and their status

multitask        5 multitask \ start cooperative multitasking

(pause)

singletask       5 singletask \ stop cooperative multitasking

addTask          5 addTask { ua# cfa# , unext# -- } \ add task into linked list at last be-
fore main task

Multitasking_Lib

prtSource        4 prtSource ( < word> -- ) \ print source in color

colSource        4 colSource loc ( cfa -- ) \ display source in color

gwords          4 gwords loc \ words with glossary

length$         4 length$ loc_a ( addr1 -- n ) \ { give length of NULL terminated string

dots            4 dots ( n -- ) \ emit n dots

spaces          4 spaces ( n -- ) \ emit n spaces

prtDef          4 prtDef ( < word> -- ) \ print definition

prtDefLine      4 prtDefLine ( cfa -- ) \ print from dictionary

scanSource      4 scanSource \ fills source fields from first buffers

scanDefs       4 scanDefs loc \ fills source field from actual buffer

cmpWord$        4 cmpWord$ { w1# w2# , c1# c2# flag# -- flag# } \ compare limited by
whitespace

findLineStart   4 findLineStart { addr# -- new_addr } \ scans until pos after CR, 0
for end

printLine       4 printLine { address# , char# -- } \ print to line end

Help_Lib

```

```

casetest          3 casetest ( val -- ) \ a test for switch....case...break
break             3 break ( -- ) \ case...break exits!
<case>            3 <case> ( value1 value2 -- ) \ case....break compare within switchvar
case              3 case ( value -- ) \ case....break compare with switchvar
switch            3 switch ( value -- ) \ save in switchvar
eval$             3 eval$ ( stringaddr -- ) \ save inP evaluate restore inP
within            3 within { x# lo# hi# -- flag } \ true if n is within range of low and high
                  inclusive
locTest           3 locTest { l1# l2# , l3# -- } \ a test for named locals
;                 3 ; \ new version with locals
isstruct!         3 isstruct! \ mark as structure word
{                 3 { \ begin locals Syntax { local1 local2 ... , nonInitLocal ... -- comment } \
forgetLocals      3 forgetLocals \ patch back the standard names
getWord           3 getWord ( -- address ) \ get next word of the input stream into lineBuf$
append$           3 append$ ( source dest -- ) \ append null terminated string
cpy$              3 cpy$ loc_ab ( source dest -- ) \ copy null terminated string
nextName$
cells             3 cells 4 * ;
NamedLocals_Lib
prtDefs           2 prtDefs loc \ prints 1st lines of buffer
MODULE            2 MODULE ( addr1 addr2 -- ) ! ( Ueberspringen der Def zwischen INTERNAL und
                  EXTERNAL ) ;
EXTERNAL          2 EXTERNAL ( -- addr ) here@ ( Start of visible definitions. Adresse des
                  aktuellen Linkfields ) ;
INTERNAL          2 INTERNAL ( -- addr ) context @ ( Start of hidden/local definitions ) ;
colBuf            2 colBuf loc ( -- ) \ display bBuf in color
prtComment        2 prtComment loc_a ( pointer -- new pointer ) \ prt comments green
                  from /_ or ( _ to lf )
copyword          2 copyword loc_ab ( source dest -- new_source )
cwords            2 cwords loc \ print words in color
prtCWord          2 prtCWord loc_a ( lfa -- ) \ print word in color
colTab
index             2 index ( -- ) \ display all first lines of blocks
lineBuf$
space             2 space 32 emit ;
cr                2 cr 10 emit ;
prtBuf            2 prtBuf ( -- ) \ display bBuf

```

```

mkBlocks          2 mkBlocks ( CLEARS BLOCKS)

bPoint

page              2 page ( -- ) \ { clear page

at-xy              2 at-xy ( col row -- ) \ { position cursor at col row, starting with 0 0

.3d               2 .3d ( n -- ) \ { print 2 digits without blank

blue

yellow

green

red               2 red 1 fcolor ; : green 2 fcolor ; : yellow 3 fcolor ; : blue 4 fcolor ;

fcolor            2 fcolor ( col ) 27 emit 91 emit 51 emit 48 + emit 109 emit ;

normal

bold              2 bold 1 scolor ; : normal 0 scolor ;

scolor            2 scolor ( color -- ) 27 emit 91 emit 48 + emit 109 emit ;

-->              2 --> \ load next block too

untrace           2 untrace 2000000 trace! ; \ stop tracing

clearstack        2 clearstack \ clear parameter stack

++                2 ++ ( addr ) \ add one to variable

<>               2 <> = 0= ;

wkey              2 wkey ( -- char ) ( wait for char )

?dup              2 ?dup dup if dup then ;

gu                2 gu ( block -- ) ( gettext update load )

wipe              2 wipe ( clear the actual block buffer )

MarkBlock1

gu

wipe

load              1 load ( blocknumber -- ) evaluate block

block              1 block ( blocknr -- buffer_address ) flush if updated and load new block
into bBuf

update            1 update mark block buffer

flush             1 flush write block back if updated

maxBlocks#

updated

dump4             1 dump4 ( address -- ) dump 4 lines

fibo46

fibo

cogstart          1 cogstart ( COGnr userArea instruction -- ) start word in new cog use
with ' word

```



```

getms          1 getms ( -- milliseconds )
getus          1 getus ( -- microseconds )
pin!           1 pin! ( flag pinnumber -- )
pin@           1 pin@ ( pinnumber -- flag )
bye            1 bye leave P2CCForth
cnt@           1 cnt@ ( -- counter )
cmove>         1 cmove> ( source dest cnt -- ) move chars starting at source+cnt-1
cmove          1 cmove ( source dest cnt -- ) move chars starting at source
trace!        1 trace! ( value -- ) set trace flag 0 for all
up@           1 up@ ( -- address ) get user area address
pause         1 pause \ pause active task and switch to next, if ready
background    1 background ( -- varaddress ) variable holds cfa of background task, 0
               for nothing
>>           1 >> ( n1 n2 -- n1>>n2 ) shift right
<<           1 << ( n1 n2 -- n1<<n2 ) shift left
rdepth        1 rdepth ( -- rdepth) find depth of return stack
depth         1 depth ( -- depth ) find depth of parameter stack
]             1 ] switch to compile
[             1 [ switch to execute
word          1 word ( -- addr_of_wordBuf ) get next word into wordbuf and return address
inP           1 inP ( -- address_of_input_pointer )
context       1 context ( -- address ) start of link list of words
lineCnt       1 lineCnt ( -- varaddress )
actBlock      1 actBlock ( -- address ) variable containing the actual block
bBuf          1 bBuf ( -- address ) get address of block buffer
bSize         1 bSize ( -- size ) constant 32768
dir           1 dir print directory
forget        1 forget forget following wordname
gettext       1 gettext get text into bBuf until ESC
conkey        1 conkey ( -- char ) get char from console terminal, 0 for none
format        1 format \ format file system deletes all blocks
fwriteBlock   1 fwriteBlock ( textbuf startpos length -- )
freadBlock    1 freadBlock ( textbuf startpos length -- length)
lastfree      1 lastfree ( -- varaddress )
."            1 ." (--) print a string terminated by ''
print$        1 print$ ( address -- ) print null terminated string

```

s"	1 s" \ ( -- address ) Usage : mystring s" My String" ;
getstring	1 getstring \ internal
fill	1 fill ( startaddress number byte -- ) \ fill buffer with n bytes
c@	1 c@ (adr -- c) readc(1 byte) fromadr
c!	1 c! (c adr --) storec(1 byte) toadr
mod	1 mod (n1 n2 -- n3 ) remainder ofn1 / n2 (sign ofn1 )
max	1 max (n1 n2 -- n3 ) leave greater of two numbers
min	1 min (n1 n2 -- n3 ) leave lesser of two numbers logical
xor	1 xor (x1 x2 -- x3 ) bitwise boolean xor
and	1 and (x1 x2 -- x3 ) bitwise boolean and
or	1 or (x1 x2 -- x3 ) bitwise boolean or
j	1 j (--n)get outer loop count
i	1 i (--n) get current loop count
+loop	1 +loop (n--) addnto loop count, terminate if end
doplusloop	
loop	1 loop (--) increment loop count, terminate if end
doloop	
do	1 do (n1 n2 --) counted loop structure do...loop (n2 = count start,n1 = count
end)	
repeat	1 repeat (--) jump back to begin in a while loop
while	1 while (flag--) exit loop whenflag= false (begin..while..repeat)
again	
else	1 else (--) false condition of an if structure
then	1 then (--) end of an if conditional structure
if	1 if (flag--) conditional structure if..(else)..then
until	1 until (flag--) loop untilflag= true (begin..until)
begin	1 begin (--) begin a while or until loop
negate	1 negate (n1 -- n2 )n2 = -n1 (two's complement)
0=	1 0= (n -- flag) true ifn= 0
>	1 > (n1 n2 -- flag) true if n1 >n
<	1 < (n1 n2 -- flag) true if n1 < n2
=	1 = (n1 n2 -- flag) true if n1 =n2
immediate	1 immediate \ set immediate bit in flags of last word
0branch	
branch	
+	1 +! ( value address -- ) \ add value to contents at address

```

1-          1 1- (n1 -- n2 )n2 = n1- 1
1+          1 1+ (n1 -- n2 )n2 = n1 + 1
*/          1 */ (n1 n2 n3 -- n4 )n4 = n1 * n2 / n3
/           1 / (n1 n2 -- n3 )n4 = n1 / n2
*           1 * (n1 n2 -- n3 )n3 = n1 * n2

allot       1 allot ( n -- ) reserve n bytes in dictionary
(
\
endloc      1 endloc release locals

loc_abcde
loc_abcd
loc_abc
loc_ab
loc_a       1 loc_a move stack frame for locals with one parameter in a
loc         1 loc move stack frame for locals

e>
+>e
>e
d>
+>d
>d
c>
+>c
>c
b>
+>b
>b
a>
+>a
>a
+to
to
value       1 value <name> ( number -- ) define new value type variable
dovalue
constant
doconst

```

```

create
dcreate
variable
dovar
evaluate          1 evaluate ( string_address -- ) evaluate words in string
fnext             1 fnext end of fast for..next
ffor              1 ffor ( number -- ) start of fast for loop
dofnext
;
:                 1 : \ start word definition
c,                1 c, \ ( byte -- ) compile a byte into dictionary at here
,                 1 , \ ( long -- ) compile long into dictionary at here
nip               1 nip ( a b -- b ) dump NOS
see               1 see <word> decompile word
prtword
execute           1 execute (adr--) execute word with compilationadr
'                 1 ' ( <word> -- cfa ) \ get code field address of following word
find$             1 find$ ( string_address -- ) get link field address
words
lwords            1 lwords list words
over              1 over (x1 x2 -- x1 x2 x1 ) copy second on stack to top
rot               1 rot (x1 x2 x3 -- x2 x3 x1 ) rotate 3rdcell to top
swap              1 swap (x1 x2 -- x2 x1 ) exchange the two top cells
r@                1 r@ (-- x) (r: x -- x) copy from return stack
r>                1 r> (-- x) (r: x --) retrieve from return stack
>r                1 >r (x --) (r: -- x) move tos to return stack
lit
exit              1 exit ( -- ) exit current word execution
-                 1 - (n1 n2 -- n3 )n3 = n1- n2
+                 1 + (n1 n2 -- n3 )n3 = n1 + n2
.                 1 . ( a -- ) print a decimal
emit              1 emit (c--) print ascii character
here@              1 here@ ( -- compilation_address )
@                 1 @ (adr -- x) readx(2 bytes) fromadr
!                 1 ! (x adr --) storex2 bytes) toadr
2dup

```

```
dup          1 dup (x -- x x) duplicate tos
2drop
drop         1 drop (x --) discard tos (top of stack)
dump16       1 dump16 ( address -- ) dump 16 bytes
prtS4        1 prtS4 print stacks
nop
RDepth: 1 Depth: 0
$0 $0 $0 $0
```