TACHYON Forth Model

Whitepaper presentation
Peter Jakacki - October 2021
Tachyon Forth Sourceforge

The purpose of this document is to help those who are familiar with Forth to understand and appreciate the differences and design decisions that went into the Tachyon Forth model. However it also elaborates upon those features too and the use thereof.

`.:.:--TACHYON--:.:.`

TABLE OF CONTENTS

INTRO

Primarily designed for the Parallax multicore MCU - the P1 - and now also for the P2 which also has a version embedded in silicon called TAQOZ (Tachyon O/S).
- Versions being developed for RISC-V and RP2040 (ARM M0+)
- Previous "standard" Forth implementations were nowhere near fast or compact enough (for starters).

Since I use Forth for commercial and industrial control I needed a Forth that is fast and solid along with features that support development etc. Most Forths seem to be written by "programmers" rather than makers and as such lack features needed for the real world in devices that may need to continue running 24/7/365 without any downtime or glitches. The Propeller is a very versatile and flexible MCU but presented some challenges, both with the multicore cog/hub architecture, and with the existing tools. However it needed to be fast and also easy to compile the kernel from source as well. The Propeller lacked stack addressing modes and so to make it fast the stack had to be in fixed registers etc.

FEATURES:
- Optimized for turn-key embedded systems on modern 32-bit MCUs
- 3 STACKS - DATA, RETURN, & LOOP STACK
- 3 MEMORY SPACES - CODE, DATA, DICTIONARY
- WORDCODE - Fast & compact 16-bit hybrid address code (easy to modify)
- Removable headers and movable dictionary
- Data is not intermixed with code or dictionary
- COMPEX - Interactive non-interpretive Compile & Execute
  M2M - low-latency Machine to Machine scripting at compiled code speeds
- Flexible numeric input and output

Why not ANSI? Unlike a desktop or embedded PC which come with great resources and standard x86 type ISA, the embedded world is a very different landscape and processors are chosen based on price + capability + availability + power requirements + size etc. One thing practically all microcontroller chips have in common is that they don't have gigabytes of memory etc. In fact, many operate well below 100MHz and may have only a few kilobytes of RAM. But they are small, cheap, low-power, and nimble for the task. No one would design a keyboard with a huge power hungry and expensive CPU for instance, they instead use tiny sub $1 chips that draw mere milliwatts. The fact is that ANSI might be well suited for PCs while Forth itself is well suited for embedded MCUs, more so in the 21st century than originally. An MCU needs to do its job, quickly and efficiently, so therefore a machine type Forth is used with adherence to standard Forths where possible and practical.

COMPEX CLI - Compile then Execute Command Line Interface

Problem: The Forth command line is what makes Forth interactive, but the trouble is if I type 10 HIGH 10 LOW to generate a pulse on my embedded system during testing for instance, then that pulse could end up being many milliseconds long rather than a microsecond or so. Why? Because Forth assembles the text input into a text input buffer (TIB) until the enter key is pressed and then (and only then) it parses each word from the beginning dutifully searching the dictionary for a match and if found it will then execute it before going on to interpret the next word.

Processing of numbers suffers greatly because the dictionary has to be searched each time before it even tries to convert it to a number on the stack, and this takes the longest time. The result of all this is that there are long delays between the execution of each word or number.

None of this is really a problem on PCs or systems where we are not talking directly to I/O and where we are not concerned with real-time latencies so to us the response seems "instant". But in embedded use or for when one Forth machine talks to another Forth machine, it is far from instant and more than likely unsuitable for real-time control.

The other problem is since traditional Forth does not allow "compile time only" words on the command line (and even needs special interpret words) and if I wanted my test to repeat many times in a loop then I would have to create a new word, run that word, and then forget that word. That is so messy, slow, and unnecessary as if the Forth interactive command line is somehow a very slow, jerky, and crippled version of the Forth itself.

Solution: Rather than finding each word and executing it, we could compile it as if the text line were a colon definition, and then when the line is entered we could append a ; EXIT before executing the code from where we started. This would run at full speed and would always allow any compile-time words to be used. Once it has finished executing the code pointer can be restored to where it was. It's as simple as that and really just requires a shadow code pointer that is used to reset HERE.

Compex interactive compile and execute

```
TAQOZ# 'A' 26 ADO I 'Q' = IF SPACE THEN I EMIT LOOP --- ABCDEFGHIJKLMNOP QRSTUVWXYZ ok
TAQOZ# LAP 26 HIGH 26 LOW LAP .LAP --- 178 cycles= 523ns @340MHz ok
TAQOZ# HERE .L --- $0000_B72C ok
TAQOZ# LAP 1,000,000 0 DO LOOP LAP .LAP --- 32,000,227 cycles= 94,118,314ns @340MHz ok
TAQOZ# HERE .L --- $0000_B72C ok
```

(ADO is the same as DO except it takes a start and cnt - equivalent to BOUNDS DO)

Still haven't grokked it yet?
A little further explanation to help you grasp the concept….
You know that when you start a colon definition that every word is compiled except of course immediate ones such as IF etc. The code is compiled via the code pointer at HERE. Imagine now that your Forth interpreter was stuck in this mode except that it copies HERE into START at the start of the line and when the end of the line is reached this is what it does:

- Append an EXIT
- Now execute this code from START.
- Reload HERE from START

It is as simple as that and only needs explaining because we are used to the slow interpret mode but I'm sure if Chuck had thought of this back when, then this wouldn't need any explaining at all.

What shall I call this method? It can be called many things but naming is necessary only because Forth traditionally used a crippled interpret mode. Compile all text is really just Forth though.
***COMPEX** perhaps? COMPile and EXecute. So Tachyon uses a "compex cli" (I'm running with that)*
*BTW, Forth is normally referred to as a REPL (read–eval–print loop) but it doesn't always print, especially on embedded system, but it always executes. Tachyon would be termed a **RCEL** (read-compile-execute-loop) whereas traditional Forth is a REEL.*

Word by word
Another thing that helps is to compile word by word as text is being entered on the line so that when the line is done it is ready to execute without any further delays. With this approach the response IS almost instant from a machine-to-machine (M2M) perspective, and so now scripting can be used to control Forth machines for real-time control purpose. Another advantage is that there is no need for a TIB and so a "line" can be as long as it needs to be. After all, it's not text, it's code.

Is it a number?
On top of this too is that rather than searching the dictionary immediately we have a word, we instead check to see if the word could be a number. Does it have at least one decimal digit? Does it have a valid prefix/suffix? These checks and others are thousands of times faster than an exhaustive dictionary search, so if they are simple and make sense then it wouldn't make any sense to not do it that way.

STACKS x3

- DATA
- RETURN
- LOOP & BRANCH
- + machine stack

DATA STACK

Tachyon, at least on the Propeller has the top 4 data elements in fixed registers - assembly instructions can operate directly on parameters. There are no pick or roll words so as to encourage simple, clean, and efficient stack use, although 3rd and 4th etc are provided.

One of the greatest criticisms of Forth, even by Forth programmers, is trying to keep track of and juggling the stack. The solution really is not writing Forth, but thinking Forth. Just keep it simple and factor factor factor (sensibly).

P2 listing of some primitives which can operate directly on the stack registers

```
                  ' + ( n1 n2 -- n3 ) Add top two stack items together and replace with result
0097c 0c2 f1104624 PLUS         add     a,b wc
00980 0c3 fd800090              jmp     #\NIP
                  ' C@  ( caddr -- byte ) Fetch a byte from hub memory : 100ns @320MHz
00ac0 113 0ac04623 CFETCH _ret_ rdbyte  a,a
                  ' 1+
00960 0bb 01044601 INC  _ret_   add     a,#1
```

RETURN STACK

The return stack is dedicated to return addresses although >R R> etc are "tolerated". IMO traditional Forth return stacks are a weak point since they also hold other parameters other than return addresses and it is messy to access these parameters. If for instance the stack was not properly popped before returning, then Forth could wrongly pop data as a code address to return to.

Why paint Forth into a corner when it is just as easy to leave the return stack for return addresses and have a loop stack for loops?

LOOP STACK

The loop stack holds index and limit plus a branch address which is the IP (the Forth PC) at the time DO executes. Like the data stack these top elements are in fixed registers for fast operations.

- There is no DO and (DO) - only DO. Same for LOOP etc.
- DO pushes index, limit, and the current IP onto the LOOP stack.
- LOOP uses the branch address stored in a fixed register on the loop stack so there is no need to calculate the branch address at compile time nor use any further code memory.

Reflecting on DO LOOPs we can see that LOOP has to go back to just after DO and so really there is no need for a compile-time calculation, only a simple run-time push that only happens before the loop, not each loop. One wonders why this method was not used from the beginning.

P2 listing of LOOP

```
00bb4 150 f1044e01 LOOP          add      index,#1                ' increment index
00bb8 151 f2585027               cmps     limit,index wcz
00bbc 152 1603f029       if_a    mov      PTRA,loopip             ' Branch to DO (PTRA is the IP)
00bc0 153 1d64002d       if_a    ret
00bc4 154 f1842e01 UNLOOP        sub      lpptr,#1                ' pop loop index'
```

Access index and limit or leave externally from within the loop.

Example of accessing index from outside the loop word (also compile on-the-go)

```
TAQOZ# : .index I . SPACE ; ---  ok
TAQOZ# 10 FOR .index NEXT --- 0  1  2  3  4  5  6  7  8  9   ok
```

MEMORY SPACES x3

- CODE - assembly and threaded
- WORD - Dictionary of headers and code pointers
- DATA - Variables and buffers

CODE SPACE

Traditionally when variables are mixed in with code it is possible to inadvertently write over code when writing to variables incorrectly, especially since they are untyped. Also when headers are mixed in with code it is then not possible to remove a header and reclaim memory etc.

Since Tachyon headers and variables are placed elsewhere in memory and there is no indirect CFA then **code** can be contiguous and also fall-through to the next code word too. This is just like how assembly code in the kernel can fall through from one routine to another.

Example of wordcode that isn't terminated by a ; but falls through.

```
: MEGA  1000 *
: KILO  1000 * ;
```

Examine the wordcode produced by a fall through (0BFC6 to 0BFC8).

```
TAQOZ# SEE MEGA
1B5B4: pub MEGA
0BFC4: 23E8     1000
0BFC6: 1164     *
1B5AD: pub KILO
0BFC8: 23E8     1000
0BFCA: 1164     *
0BFCC: 0065     ;
     ( 10  bytes )
```

WORD SPACE

Because headers are all packed in together one after another the **dictionary** doesn't need any link fields. The dictionary builds down towards code which builds up. In the event of running out of room in-between, the dictionary is simply block moved upwards and the pointer readjusted.

- No link fields required
- Resizable and relocatable
- Private headers can be automatically removed

Dictionary header fields.

ATR+CNT(1)	NAME(1..31)	ADDRESS(2)	PAGE(1)
04	KILO	$BFC8	
04	MEGA	$BFC4	

Each header is comprised of count+flags(1),Name(1..31),address(2) then the previous entry.

```
TAQOZ# @WORDS $10 DUMP ---
1B5AD: 04 4B 49 4C  4F C8 BF 04  4D 45 47 41  C4 BF 01 4B     '.KILO...MEGA...K'
```

If the count+flag field is zero then this indicates a special control field along with the next byte. The end of the dictionary is indicated by a 00 00. NFA' finds the header address of a word and CFA>NFA can find the header address from a code address.

The dictionary can be moved and entries deleted to reclaim memory from "headerless" code. As well as the colon definition there is pub pri and pre where pri sets an attribute in the header which RECLAIM can use to determine which headers to remove. Use pre to indicate a preemptive or immediate definition.

```
TAQOZ# pub COLON ; ---  ok
TAQOZ# pri PRIVATE ; ---  ok
TAQOZ# pre PREEMPTIVE ; ---  ok
TAQOZ# NFA' PREEMPTIVE $20 DUMP ---
1B998: 8A 50 52 45  45 4D 50 54  49 56 45 30  B7 47 50 52     '.PREEMPTIVE0.GPR'
1B9A8: 49 56 41 54  45 2E B7 05  43 4F 4C 4F  4E 2C B7 06     'IVATE...COLON,..' ok
```

Aliases can easily be created for existing words and words can even be renamed even if this involves resizing the dictionary.

```
TAQOZ# ALIAS COLON PUBLIC ---  ok
TAQOZ# NFA' PUBLIC 10 DUMP ---
1B98F: 06 50 55 42  4C 49 43 2C  B7 8A 50 52  45 45 4D 50     '.PUBLIC,..PREEMP' ok
```

Aliases are also used to provide alternative names for common Forth symbols. Symbols such as . and ' and even \ etc are fine for interactive use but get lost in reading source code, so aliases of these words stand-out in source code and are not mistaken for other operators.

Header Attributes

b7 -	Private - maked for automatic stripping and reclamation if needed
b6 -	Preemptive immediate - typically compile-time words but also for retrieving further strings etc
b5 -	Paging - indicates 3 bytes for code address field.
b4..b0	Count 1..31

* No smudge attribute - only a simple ignore of the latest name while compiling a new definition.

Names are limited to 31 characters although most sensible names rarely go over 16 characters. It's mainly those long underscored names that might need more than 31 but then that's not very Forth-like and the code really needs to be factored better. For example, `turn_second_green_led_on` is a very poor way of writing and naming Forth code that is begging for a some thinking Forth to factor that and turn it into `on 2 green led` for instance so the words form a new language and a new way of describing things with meaningful words.

DATA SPACE

A variable In Forth is a cell that is interspersed with headers and code etc, so the storage area is like a strange little island surrounded by a sea of code. Also, to specify another size other than a cell requires some kind of kludgy +/- ALLOT. In embedded systems though it is not possible to have a variable in Flash and so depending upon whether it is compiled to RAM or Flash changes how the variable needs to be compiled.

There is no need for this at all. Variables do not have to be where the code is, and it is actually much much better that they are not.

Variables in Tachyon are specified as they would be in assembly or most languages as to size etc using common expressions such as byte/word/long/double as well as the plural of these etc. Groups of data variables can be treated as an array since data is contiguous. This means whole structures can be erased in one hit if needed. So the variable is actually an address constant that points to the area in data space.

Example of specifying "variables" in data space

```
       *** FAT32 BOOT RECORD ***

3    bytes   fat32           --- jump code +nop
8    bytes   oemname         --- MSWIN4.1
     word    b/s             --- 0200 = 512B (bytes/sector)
     byte    s/c             --- 40 = 32kB clusters (sectors/cluster)
     word    rsvd            --- 32 reserved sectors from boot record until first fat table
     byte    fats            --- 02
2    res                     --- Maximum Root Directory Entries (non-FAT32)
```

Snipping the output of .VARS reveals the address of those variables.

```
$0000_2257 @ fats
$0000_2255 @ rsvd
$0000_2254 @ s/c
$0000_2252 @ b/s
$0000_224A @ oemname
$0000_2247 @ fat32
```

Sometimes though we want a pre-initialized variable or a variable constant but there various ways of handling that. For instance, on the Propeller all code is loaded into RAM so variables are actually remembered when the code is backed up, especially on the P1 with its limited 32kB RAM and the way it boots. With the P2 it is simply a matter of specifying that range or placing special variables in an area alongside the code. Variables associated with the extended kernel are already in the special code area whose address codes are reserved for encoded wordcodes.

THREADING MODEL

You may be familiar with Forth threading models such as Indirect, Direct, Subroutine, and Token threaded code, but Tachyon uses a form of threading that is memory map dependent and does not require a code header but is inferred by the address range. Each address token is a 16-bit wide "wordcode" and can be a direct address to assembly code, or a direct address to threaded code, or even as a encoded token that can be a literal or relative conditional branch etc. Also, even though the address is only 16-bits it can still address higher code by using page select codes before a call, otherwise all addresses are assumed to be in the default code page. This provides a balance between speed and code compactness since most kernel and extensions and even large applications can fit into the default 64k code page since the code is so compact (no code headers, no dictionary etc).

WORDCODE

16-bit hybrid threaded wordcode

- Lower address range are direct calls to assembly code primitives
- Block of 2kB addresses reserved for encoded functions:
    - Short literals (10-bit including down to -8)
    - Conditional relative branches (IF ELSE WHILE UNTIL REPEAT)
    - Task registers (I/O hooks, pad, base etc)
    - Memory reused for data space (see .VARS above)
- Implicit threaded code above this block (no CFA field - just wordcode)
- Compact and easy to patch consistent code.

So wordcode looks a bit like a standard 16-bit Forth address except there is no CFA to determine if a routine is assembly or threaded etc. Instead, the wordcode interpreter will assume it is assembly if it is in a certain region, and threaded if it is in another. There's also a region in between where the wordcode is not an address, but an encoded literal or branch etc. The wordcode interpreter is optimized so that assembly code has minimal overhead and is directly called.

Q: Since user code would always be interpreted as threaded, how then do we write assembly words?
A: It is simply a matter of having a single ASM wordcode that calls the code that follows. This is added automatically to any CODE definition which also opens the assembly vocabulary and allows for a listing during interactive coding.

Main part of the wordcode address interpreter

```
doCALL call      x                      ' STEP 4 - call assembly primitive
doNEXT rdword x,PTRA++          ' STEP 1 - read a wordcode
doCODE cmpr   x,coded wc        ' STEP 2 - low range address is assembly code call
if_nc  jmp     #\doCALL         ' STEP 3 - either a call or something else.
       cmp     x,threaded wc    ' STEP 4 - either threaded code or an encoded token
if_nc  jmp     #\ENTER                  ' STEP 5 - Enter threaded code
```

Threaded addresses are always on an 16-bit boundary so the lsb is not used for addressing. Instead, the lsb indicates a jump rather than a call so this can also save an EXIT. This is handled automatically during compilation when a ; is encountered.

Example of wordcodes and using the jump lsb on threaded calls

```
TAQOZ# : .CIRC  ( radius -- ) DUP IF 2* 355,000,000 113 */ THEN PRINT ; ---  ok
TAQOZ# 25 .CIRC --- 157079646 ok
TAQOZ# SEE .CIRC
1B599: pub .CIRC
0BFE6: 009C     DUP
0BFE8: 2406     IF $BFF6
0BFEA: 00F0       2*
```

```
0BFEC: 007C       := 355000000    $1528_DEC0
0BFF2: 2071       113
0BFF4: 119C       */
              THEN
0BFF6: 357D     PRINT ;
     ( 18  bytes )
```

Addressing beyond 16-bit 64k boundary
Page codes are used for any reference to code outside of the primary 64k bank but otherwise primitives etc are still only 16-bit whereas paged calls are effectively 32-bits but with very little impact on performance.

Use of PAGE codes for >64kB code space references

```
TAQOZ# SEE C2START
196FD: pri C2START
11604: 0055     PAGE1
11606: 1590     C2D
11608: 0182     FLOAT
1160A: 0172     L
1160C: 0171     H
1160E: 0065     ;
     ( 12  bytes )
TAQOZ# ' C2D .L --- $0001_1590 ok
```

Paged references only incur an extra 25% cycles to call

```
TAQOZ# SEE DUMMY
1950A: DUMMY
11CB6: 0065     ;
     ( 2  bytes )
TAQOZ# LAP DUMMY LAP .LAP --- 81 cycles= 253ns @320MHz ok
TAQOZ# SEE NOP
1F702: pub NOP
02A3A: 0065     ;
     ( 2  bytes )
TAQOZ# LAP NOP LAP .LAP --- 65 cycles= 203ns @320MHz ok
```

So far I haven't really needed any page code stuff as wordcode is so very compact even "fully loaded" and code has its own codespace that has only ever exceeded 64k by manually changing the code-pointer for testing purposes just to confirm it works as intended.

Why not bytecode? Isn't that more compact?
My early versions of Tachyon were written as a bytecode model which looks really compact initially. Vector table overhead was needed though for threaded calls so 2 bytecodes were always required anyway as they also are even for an 8-bit literal. Once the code grows it gets even messier and slower, even having to go to 3-byte calls within the first 64k whereas wordcode is far more efficient overall and actually faster and more compact and cleaner with fixed width codes.

STRINGS

Tachyon uses null-terminated strings since these only require a single address and simplify the stack. There is a fast LEN$ instruction which returns the length. Many string operations can be effected by manipulating the address and the termination.

BOOT

The kernel will check for a vector (REG $E0) which the basic extensions and FLAT32 filesystem (hierarchical FAT32 with linear addressable unfragmented files) will have set to the BOOT routine. While this vector can be overwritten by the user, it provides basic boot checks and mounting etc. BOOT also calls INITS which is designed to allow user code to insert calls to various modules that are added which are called in order that they are set.

INITS with media module loaded

```
TAQOZ# see INITS
1ED5F: pub INITS
047DE: B6CC     !MEDIA
047E0: 2A3A     NOP
047E2: 2A3A     NOP
047E4: 2A3A     NOP
047E6: 2A3A     NOP
047E8: 2A3A     NOP
047EA: 2A3A     NOP
047EC: 2A3A     NOP
047EE: 0065     ;
     ( 18  bytes )
```

BASIC USER FEATURES:

- Assignable CLI control key shortcuts - not just CR and BS
- Number prefix modes #10 $DEAD_BEEF %1101 &192.168.0.1 ^A 'A'
- Mix any symbols in with numbers (1,000 12:45:59 etc)
- Numbers are preprocessed prior to searching the dictionary.
- DUMP can handle various memories, files, and devices in different formats (byte/word/long/char).
- TRACE execution with window control.
- CAP wordcode usage counters provide statistics
- LAP time execution time in cycles/us etc
- Single word comments end with an underscore_. Ready?_
- --- comment separator between user input and response.
- Super fast zero delay serial source code compile (no handshakes)

*Compact_comments_*

```
TAQOZ# TRY_THIS_ 'A' 26 ADO I EMIT LOOP --- ABCDEFGHIJKLMNOPQRSTUVWXYZ ok
```

I use compact comments a bit like labels at the start of a line, especially within CASE statements.

Assignable control key vector list

```
TAQOZ# .CTRLS ---
$02 ^B ~MBR
$03 ^C 03B42
$04 ^D ~DEBUG
$06 ^F ~FLASH
$07 ^G 02A54
$08 ^H 03B24
$09 ^I 03B18
$0B ^K 03B40
$0C ^L CLS
$0D ^M 03B3C
$0E ^N COLD
$10 ^P 03B46
$11 ^Q 03B48
$12 ^R ~RXCAP
$13 ^S 03B08
$14 ^T 03B52
$15 ^U ~USAGE
$16 ^V .VER
```

```
$17 ^W ~QWORDS
$18 ^X 03B3C
$19 ^Y ~WORDS
$1A ^Z REBOOT
$1B ^[ 03B14
$1C ^\ CRLF
$1D ^] ~SAFE
$1F ^_ DEBUG ok
```

Using control keys to perform actions on the command line really makes sense. We use control keys all the time with PC apps, why not with Forth? Do I really need to type .S<cr> just to check the stack? No, just type ^Q for instance or ^S to clear it. Has the dictionary been corrupted and Forth doesn't recognize any words, so rather than reset we can type ^D to print out a debug listing or ^W for a quick words listing etc.

So some of these are preset for kernel use but the user can change these or add new functions that are application specific. ^C is very handy when we just want to reset and Tachyon also detects some of these repeated sequences in the serial receive interrupt routine. So for instance , if Forth was totally locked up somehow and not responding, I could type four ^Cs which would be detected by the independent serial ISR and perform a reset. Even ^T for TRACE is detected this way just so I can see what Forth is trying to do instruction by instruction.

TIMING CODE

LAP is a timing tool that does nothing more than save the previous LAP into a second register and latch the current system ticks into the first LAP register. A LAP@ will find the difference between these two values and then also perform a LAP LAP on itself to compensate for overheads. Finally .LAP uses LAP@ and formats the result. Insert LAP at the start and end of the code and use .LAP to report on the timing.

LAP timing

```
TAQOZ# LAP 1,000,000 FOR NEXT LAP .LAP --- 32,000,139 cycles= 100,000,434ns @320MHz ok
```

TRACE FUNCTION

This is the ultimate debug, being able to trace Forth executing each instruction and viewing the stack. So many Doh moments when you run the trace and see what is actually happening, and then you go "D'oh!" Each line represents the instruction that is about to be executed and the stack right at that point "before" it executes.

Trace function

```
TAQOZ# TRACE 1 8 FOR 2* NEXT UNTRACE ---
0BD8C : 2001  $001
0BD8E : 2008  $008              1(00000001 )
0BD90 : 1124  FOR               2(00000008 00000001 )
0BD92 : 00F0  2*                1(00000001 )
0BD94 : 015D  NEXT              1(00000002 )
0BD92 : 00F0  2*                1(00000002 )
0BD94 : 015D  NEXT              1(00000004 )
0BD92 : 00F0  2*                1(00000004 )
0BD94 : 015D  NEXT              1(00000008 )
0BD92 : 00F0  2*                1(00000008 )
0BD94 : 015D  NEXT              1(00000010 )
0BD92 : 00F0  2*                1(00000010 )
0BD94 : 015D  NEXT              1(00000020 )
0BD92 : 00F0  2*                1(00000020 )
0BD94 : 015D  NEXT              1(00000040 )
0BD92 : 00F0  2*                1(00000040 )
0BD94 : 015D  NEXT              1(00000080 )
0BD92 : 00F0  2*                1(00000080 )
0BD94 : 015D  NEXT              1(00000100 )
0BD96 : 0430  UNTRACE           1(00000100 ) ok
```

Redirectable DUMP

```
TAQOZ# 0 $20 DUMP ---
00000: D4 15 80 FD  50 32 20 20  20 20 20 20  03 64 00 00     '....P2      .d..'
00010: 00 2D 31 01  00 D0 12 13  FB 3F 4D 01  00 10 0E 00     '.-1......?M.....' ok
TAQOZ# FOPEN TAQOZ.WAV Opened @ $00C0_3B96 ---  ok
TAQOZ# 0 $20 SD DUMP ---
00000: 52 49 46 46  24 24 79 00  57 41 56 45  66 6D 74 20     'RIFF$$y.WAVEfmt '
00010: 10 00 00 00  01 00 01 00  44 AC 00 00  88 58 01 00     '........D....X..' ok
```

Another extremely useful aid is being able to dump files and device registers the same way we dump memory. It is very simple really since DUMP must use fetch words anyway, all we do is allow for redirection via modifier words. After it is done we reset back to normal memory - mainly to prevent confusion. The SD word in this example redirects the dump fetch words to SDC@ for instance which fetches a byte from an open file or starting sector as if it were up to 4GB of virtual memory. Other modifier words are used for serial Flash, I2C chips etc.

TEXT INPUT/OUTPUT

Tachyon uses a non-blocking KEY word that immediately returns with 0 (false) if there is no data. In fact it does this for many functions using a non-zero both as a flag and as a result. This is no problem for text based input but a real null has $100 added to it so that it is still seen as a character before being stripped back to 8-bits. Not having a KEY? simplify drivers that can be used as input via the ukey task vector which btw will always use the serial input if the vector is zero.

Likewise, EMIT is non-blocking as buffering transmit characters doesn't make any sense at higher baud rates since it is actually faster to send the character directly whether that is via hardware or bit-bashed than it is to buffer it and update pointers and then have an ISR having to process that. Unnecessary double handling with its own overheads. I typically run Tachyon at 921600 mainly for compatibility with most terminals but also I have run it up to 8Mbd. Even at 921600 it takes less than 11us to bit-bash a character. Just like KEY, if the uemit task vector is zero it will default to the serial transmit.

KEYPOLL

Whenever KEY is called and returns with a null result it will check the keypoll vector and execute it if it is set. This is handy for simple low-priority polling in the background. Like INITS, POLLS uses a table of wordcode vectors that the user can setup.

+POLL ('code --)
-POLL ('code --)
!POLLS

NUMBER INPUT and BASE

The use of BASE in Tachyon is available although deprecated. The reason is that really there are only three bases that we are normally interested in, decimal, hex, and binary. Therefore the default base is decimal while hex and binary numbers are prefixed or suffixed to indicate otherwise. Therefore there is no confusion about numbers such as 10 which will always be decimal whereas $10 is hex and %10 is binary. However decimal numbers can be prefixed with # or suffixed with a d if necessary, especially so if a port number such as P31 is used in which case it could be entered as #P31 rather than just 31.

Characters are also supported and so there is no need for CHAR and [CHAR} for instance, just simply enclose the character in single quotes or prefix the character with a caret if it is a control.

BASE	PREFIX	SUFFIX
DECIMAL	`#1,234`	`1234d`
HEX	`$DEAD_BEEF`	`0CAFEBABEh`

BINARY	`%1001_0100`	`11001001b`
IP	`&192.168.0.240`	
ASCII	`'A'`	
CONTROL	`^A`	

Number and character formats

```
TAQOZ# 'A' EMIT --- A ok
TAQOZ# ^P .BYTE --- 10 ok
TAQOZ# &192.168.0.240 .LONG --- C0A8_00F0 ok
TAQOZ# 1,234,567 . --- 1234567  ok
TAQOZ# #P31 . --- 31  ok
TAQOZ# 11/10/21 . --- 111021  ok
TAQOZ# 11:10:59 . --- 723515  ok
TAQOZ# 11:10:59 .LONG --- 000B_0A3B ok
TAQOZ# $DEAD_BEEF .LONG --- DEAD_BEEF ok
TAQOZ# 12345678901234567890. D. --- 12345678901234567890 ok
```

The & method of encoding dot separated decimal numbers into bytes is useful for IP notation as well as for encoding up to 4 pins into a single long. This pin method is used to specify device connections easily.
 Perhaps I could also use 5-bit pin numbers for up to 6 pins in a long such as @25.26.27.28.29.30

PRINT FORMAT

String printing supports embedded \ escape sequences such as:

\n	Newline = $0D $0A
\r	CR (return to left margin) = $0D
\t	Tab = $09
\f	Form Feed or CLS = $0C
\[	Escape [= $1B $5B
\e	Escape = $1B
\'	Double quote = $22
\sp	Eight spaces
\$nn	Hex value
\A-Z	6 to 32 spaces

Examples of Print formatting using \ sequences

```
TAQOZ# PRINT" \t\$22hello world\'" ---  "hello world" ok
TAQOZ# PRINT" Hello\ZWorld" --- Hello                               World ok
TAQOZ# 4 FOR PRINT" \nHello World" NEXT ---
Hello World
Hello World
Hello World
Hello World ok
```

PRINT AS

Normally in Forth to print a string and print a number from the stack requires multiple operations or awkward pictured output with <# #> words. Tachyon also has a "print as" operator .AS" which can format a number and intermix symbols using command characters etc. One thing to note is that commands as well as characters are processed right to left the same way that digits are built up. Any character that is not a command is processed literally.

.AS" Format string commands:

#	Convert one digit (default is decimal)
~	Toggle leading zero suppression

\	pad leading zeros with spaces
$|	convert digits to hex base
*|	Convert all remaining digits
n|	Convert n digits (4| = 4 digits)

Examples of Print formatting using .AS"

```
TAQOZ# 123 .AS" #.##us" --- 1.23us ok
TAQOZ# 46 .AS" fibo(*|) = " --- fibo(46) =  ok
TAQOZ# 253 .AS" @ ##.#%" --- @ 25.3% ok
TAQOZ# 12 .as" ####" --- 0012 ok
TAQOZ# 12 .as" 4|" --- 0012 ok
TAQOZ# 12 .as" 4|\" ---   12 ok
TAQOZ# 12 .as" ####~" --- 12 ok
TAQOZ# 12 .as" ####\" ---   12 ok
TAQOZ# 1234 .as" ##:##$|" --- 04:D2 ok
```

Maybe I can rename .AS" to PRINTF" perhaps if it's not too confusing.

TIMERS
TIMEOUT (ms dvar --)
TIMEOUT? (dvar -- flg)
ALARM

RTC
I2C RTC devices are supported and many are very similar varying only with certain bitfields although the time and data and normally in the same register position. So these devices are supported but mainly the RV3028-C7 since it is built into the P2D2 and features extremely low standby current of 40na. Since the standby current is so low, it is backed up with a 11,000uF super-cap suitable for several "off power".

Tachyon supports TIME@ TIME! DATE@ DATE! in standard decimal formats as well as many other RTC words for access and printing. Since reading the time or data involves previous processor cycles over I2C which takes around 277us, there is also a QDATE@ and a QTIME@ which reads a soft RTC that has already been synchronized with the hardware RTC. The soft RTC actually reads from the system tick counter and subtracts an snch offset then returns this time in seconds which is converted to HH:MM:SS as needed.

Examples of RTC access

```
TAQOZ# .FDT --- 2021/10/23 SAT 12:17:53 ok
TAQOZ# QTIME@ . --- 121755  ok
TAQOZ# QDATE@ . --- 211023  ok
TAQOZ# DAY@ . --- 6  ok
```

Devices such as the RTC chip can also be accessed with the RTC@ and RTC! and thus makes it suitable as a dump source.

```
TAQOZ# 0 $40 RTC DUMP ---
00000: 18 18 12 06  23 10 21 80  80 80 00 00  00 00 31 00     '....#.!.......1.'
00010: 00 00 00 00  00 00 00 00  00 00 00 E9  70 73 61 06     '............psa.'
00020: 00 00 00 00  00 00 17 00  33 00 00 46  62 13 16 17     '........3..Fb...'
00030: FF 00 00 00  00 C0 FF B4  99 19 00 00  00 00 00 00     '................' ok
```

EXTENDED FEATURES:

- FAST SOURCE CODE LOAD & COMPILE
- DECOMPILER
- ASSEMBLER
- FAT32 FILE SYSTEM
- VIDEO TEXT and GRAPHICS
- WAVE PLAYER
- BMP VIEWER
- BMV VIDEO PLAYER
- LOGIC ANALYZER

FILES

FAT32 files with 8.3 names (or 11 characters) are supported and treated as virtual memory up to 4GB in size. When a file is opened its directory entry is buffered and the starting sector of the file is looked up in the FAT via the starting cluster. This starting sector is unique and is the handle for the file.

An assumption is made that the files are not fragmented so that they can be addressed using the starting sector as an offset into a 4GB virtual memory. This assumption seems to hold true for SD cards unlike the constantly changing system files on a PC.

- Virtual memory file/sector addressing
- Multiblock mode reads and writes
- 7.5Mbyte/sec read speeds @320MHz (80MHz SPI bus)

DISK TOOLS

Various tools include reporting on SD card information and speeds etc as well as a flexible FAT32 formatter that works with any size card (tested 128GB).

Example of accessing a file as memory

```
TAQOZ# FOPEN MINION.BMP Opened @ $00C0_3716 ---  ok
TAQOZ# 0 $80 SD DUMP ---
00000: 42 4D 36 88  03 00 00 00  00 00 36 04  00 00 28 00     'BM6.......6...(.'
00010: 00 00 80 02  00 00 68 01  00 00 01 00  08 00 00 00     '......h.........'
00020: 00 00 00 00  00 00 12 0B  00 00 12 0B  00 00 00 00     '................'
00030: 00 00 00 00  00 00 06 05  0C 00 BD 82  34 00 48 84     '............4.H.'
00040: AA 00 C5 C3  AB 00 1D 28  76 00 99 99  9D 00 4F C3     '.......(v.....O.'
00050: EC 00 3A 43  4E 00 8C A3  AA 00 3F 57  78 00 AF C5     '..:CN.....?Wx...'
00060: D7 00 38 56  AD 00 12 15  41 00 5E 6C  78 00 C2 E4     '..8V....A.^lx...'
00070: EA 00 92 84  6A 00 46 83  DA 00 DB A6  6C 00 9C A2     '....j.F.....l...' ok
TAQOZ# $12 SD@ . --- 640  ok
TAQOZ# $16 SD@ . --- 360  ok
```

ASSEMBLER

Accepts standard format mnemonics. Generates an interactive listing.

Example of interactive assembler listing -

```
TAQOZ# code VCH8A
0B72E FD64_2828         setq    #20              ' copy into cogmod memory
0B732 FF00_005B         rdlong  @COGMOD,#PC 12 + ' read longs into cog
0B736 FB07_A33E
```

```
0B73A FD80_01D1         jmp      #\@COGMOD        ' and run it there
                       ' this section is loaded and run in cog memory '
0B73E F603_F024         mov     PTRA,b
0B742 F600_4823         mov     b,a
0B746 F044_4808         shr     b,#8             ' b = pen, a = paper'
0B74A FF00_0006         rdlong  r2,#_hp          ' _hpixels
0B74E FB04_25BC
0B752 F184_2406         sub     r2,#6
0B756 F604_2207         mov     r1,#7
0B75A FAC0_1E25  .l1    rdbyte  xx,c             ' read in next font char
0B75E FCDC_0605         rep     #3,#5
0B762 F054_1E01         shr     xx,#1 wc
0B766 CC44_4961   if_c  wrbyte  b,ptra++         ' PEN PIXEL
0B76A 3C44_4761   if_nc wrbyte  a,ptra++         ' PAPER PIXEL
0B76E FC44_4761         wrbyte  a,ptra++         ' + BLANK COLUMN
0B772 F103_F012         add     PTRA,r2          ' next line
0B776 F104_4A01         add     c,#1             ' next font byte
0B77A 0B6C_23F7   _ret_ djnz    r1,#l1
---   ok               end
```

LINKS

- Tachyon Forth Sourceforge
- TAQOZ ROM GLOSSARY
- A Bit Bashers Guide to the Parallax P2 - Using TAQOZ ROM Forth
- MORE LINKS