First, I want to understand forth-dwc.

**ChatGPT:**

Sure.👍  
 To understand "forth-dwc," let's first summarize the overall picture.

**1. What is forth-dwc?**

* **DWC stands for "dword-code**."
* It is an extremely minimal implementation of a Forth-like language,
  + **with a VM (virtual machine) of approximately 250 lines.**
  + The executable file is approximately 17 KB on Windows and 24 KB on Linux, making it extremely lightweight.
* Composed of **three C files**
  + dwc-vm.c … Core of the VM (inner/outer loops)
  + dwc-vm.h … Constants, types, and external function declarations
  + system.c … OS-dependent I/O and REPL

**2. How DWC works**

**Command representation**

* **1 instruction = 1 DWORD (32 bits)**
  + Value is **32–44 → Primitive** (VM-built-in instruction)
  + Value is **0x4000\_0000 or higher → Literal value** (determined by upper bits)
  + Otherwise → Address of a user-defined word registered in the dictionary

**State (state)**

* Has **four states** and switches the interpretation of input
  + COMPILE: Compiles input
  + DEFINE: Adds a new word to the dictionary
  + INTERPRET: Execute immediately
  + COMMENT: Treats input as a comment
* State switching is performed using control characters such as : ; [ ] ( )

**3. Primitive commands**

* **32 basic primitives** +  **13 system primitives**
  + Example
    - dup (stack duplicate)
    - + (addition)
    - jmp (jump)
    - key / emit (I/O)
    - outer (interpret C-side external string in Forth)

**4. Execution flow**

1. main() (system.c) calls dwcInit() → executes boot("boot.fth")
2. boot.fth is loaded, and the entire text is passed to outer()
3. outer() splits the text into tokens, and inner() executes them in the VM

**5. Features**

* **ColorForth influence**  
   Unique input style, including control characters and (( )) comments.
* **INLINE words**  
   Can be expanded macroscopically.
* **Transient words** (t0 to t9)  
   Temporary words that are not stored in the dictionary.

**6. Summary**

* forth-dwc is a "**VM that implements a minimal Forth interpreter in C.**"
* It can call **external C functions (such as emit, key, and fOpen),** making it easy to extend.
* A simple structure where **strings are converted to dictionary word sequences and then** executed by the **VM via outer()**

👉 Question: As the next step,

* Would you like to understand **the behavior of the primitive list step by step**?
* Or would you like to **actually feed in some Forth code and experiment**?

**You:**

Run forth-dwc and wait for input. Please tell me what can and cannot be executed at that point.

**ChatGPT:**

Thinking time: 1m 15s

Got it! I'll quickly list the "absolute minimum" things you can do and the things you can't do (or are undefined) in the REPL (input waiting state).  
 (※ forth-dwc is extremely minimal. Advanced features depend on the definitions in boot.fth.)

**What can be executed now (minimum guarantee)**

**Immediate execution (INTERPRET) and definitions**

* State transition keywords are hard-coded  
   : (Go to DEFINE) / ; (End definition and go to INTERPRET) / [ (Go to INTERPRET) / ] (Go to COMPILE) / ( ) / (( )) (Comment)
* Defining new words:  
   : STAR 42 emit ; etc. (emit outputs one character)
* Read the entire line at once, interpret and execute each token (REPL reads line input → outer interpretation)

**Stack & control**

* Data stack operations: dup drop swap over
* Arithmetic: + - \* /mod
* Comparison/logic: &lt; = &gt; and or xor
* Control primitives: jmp, jmpz, jmpnz, simple loop: for next  
   Example: Display 10 asterisks:  
   10 for 42 emit next

**Memory/Return Stack**

* Memory: @ ! (cell) / c@ c! (byte)
* Return stack: >r r@ >r

**I/O/Time/File**

* Key input: key (waits for 1 character) / ?key (press status 1/0)
* Output: emit (1 character) / ztype (outputs a null-terminated string, requires a pointer)
* Time/Wait: timer / ms
* File: fopen fclose fread fwrite fseek

**Execute another string as Forth on the spot**

* outer (passes an address and evaluates the string at that location)  
   ※ In the REPL, this works the same way as passing a line to outer for interpretation

**If left as is, it means "cannot be done/undefined" (depends on boot.fth)**

The terms below **are not included in the VM primitives**. They cannot be used unless defined in boot.fth.  
 The design assumes that you will create your own extensions if needed.

* High-level control structures: IF / ELSE / THEN / BEGIN / UNTIL / DO / LOOP, etc.
* Numeric output and stack display: . .S, etc. (custom implementation required as only emit/ztype are available)
* String literals: "." ... or S" … " (not allowed if undefined)
* Variables/arrays/dictionaries: VARIABLE/CONSTANT/CREATE/DOES&gt;/ALLOT, etc.
* Floating-point/exceptions/multitasking/vocabulary (VOCABULARY) and other advanced features
* Various standard Forth immediate words (POSTPONE, IMMEDIATE, etc.)
* System command execution (system): May not be available or may be dangerous depending on the build or environment

**REPL quick notes**

* Prompt display:  
   If in COMPILE mode, ...; otherwise, "ok" is displayed (passed to outer on a per-line basis)
* Key input uses key/?key as a word (REPL itself is line input)
* Temporary words t0..t9 are not stored in the dictionary and can be discarded (useful for short decompositions)

**A "guaranteed to work" example you can try right away**

(Output is limited to text only)

* Echo the entered key as-is:  
   key dup emit drop
* Wait 1 second and output "!":  
   1000 ms 33 emit
* Loop to output 10 asterisks:  
   10 for 42 emit next

If you want to use "IF/ELSE" or ". (displaying numbers)," you can create a mini dictionary by adding the minimum implementation.  
 We recommend starting with "display-only ., .S," "number to string," and "simple IF/ELSE/THEN."

**You:**

Thank you for your response. Please let me know what cannot be executed in forth-dwc.

**ChatGPT:**

I see, you want to list what cannot be done.  
 forth-dwc **only** has **"VM primitives + the minimal vocabulary defined in boot.fth**,**"** so it lacks most of the convenient features of standard Forth.

I will list the "things that cannot be done" as seen in the source code (dwc-vm.c's PRIMS definition, README).

**Things that cannot be done in forth-dwc (unless defined in boot.fth)**

**1. Control structures**

* IF ... ELSE ... THEN
* BEGIN ... UNTIL / AGAIN
* DO ... LOOP / +LOOP
* CASE ... ENDCASE  
   → Does not exist as a primitive. Users must define it by combining jmp/jmpz.

**2. Display/Debug**

* Numeric display: . (Displays the stack top in decimal)
* Stack display: .S
* String literals: ." hello" or S" abc"  
   → Output is limited to emit and ztype.

**3. Memory Management and Dictionary Operations**

* VARIABLE / CONSTANT / CREATE / ALLOT / DOES&gt;
* Vocabulary management (VOCABULARY) and FORGET  
   → Dictionary operations are limited to add-word and outer.

**4. Immediate Compilation System**

* IMMEDIATE
* POSTPONE
* Compile-time conditional branching (such as standard Forth's [IF] ... [THEN])  
   → State transitions are limited to hard-coded : ; [ ] ( ) (( )).

**5. Numeric representation and arithmetic**

* Floating-point arithmetic
* Shift operations (LSHIFT / RSHIFT)
* Logical NOT (0=, etc.)
* Character code conversion (CHAR, etc.)  
   → Only integer arithmetic, comparison, and AND/OR/XOR operations.

**6. Environment-related**

* Multitasking
* Exception handling (CATCH/THROW)
* Standard input/output redirection and file attribute modification  
  → Only basic C wrappers such as fopen/fclose/fread/fwrite/fseek are available.

**7. High-level tools**

* Dictionary browsing (WORDS)
* Comment display and listing (SEE)
* Assembler/meta-compiler  
   → None of these are available.

**In other words**

forth-dwc is **a VM that can only perform "stack operations, memory access, jumps, simple loops, key input/character output, and file I/O**.**"**  
 Any additional functionality must be **built upon using boot.fth or custom definitions**.

👉 If you want to proceed further:

* Examine the contents of boot.fth to determine how closely it aligns with standard Forth, which will clarify the actual "list of things that cannot be done."

Would you like to take a look?

**You:**

"forth-dwc is a VM that only supports stack operations, memory access, jumps, simple loops, key input/character output, and file I/O. Additional functionality is implemented by building on top of boot.fth or custom definitions." The above is what I was told, but these are important points, so please explain them to me. 　1) Methods to add additional functionality by using or modifying boot.fth 2) Methods to add additional functionality by building up custom definitions

**ChatGPT:**

Great question!  
 The path to adding "additional functionality" is twofold: **(1) refining boot.fth** and **(2) building custom words using the REPL**. Both approaches make effective use of the minimal primitives and internal variables provided by DWC.

Here are the key facts first:

* The available primitives (dup + jmp, jmpz, ..., key, emit, ztype, fread, etc.) are fixed and implemented on the VM side.
* The VM exposes internal variables as "literal words" (mem, mem-sz, base, state, (h) here, (vh) vhere, (l) last, &gt;in, etc.). These are registered in a dictionary during initialization and can be referenced from Forth.
* Line input is interpreted by outer, and : and ; define/terminate state transitions via hardcoded state transitions.

**1) Use boot.fth or modify it to add additional features.**

Purpose: To enable the use of "convenient words similar to standard Forth" immediately after startup.  
 Method: Define **the minimum set of "assembly" equivalent words (`, here, allot, cr, ., etc.) in boot.fth**. This will make each REPL session more comfortable.

**Example implementation of representative low-level words (to be included in boot.fth)**

(Can be implemented using only DWC's internal variables and primitives)

\ ===== Basic cell and here =====

4 constant CELL

: here@ ( -- idx ) (h) @ ; \ Read the "index" value of here

: here! ( idx -- ) (h) ! ; \ Overwrite here

: , ( x -- ) \ code[here++] = x

here@ CELL \* mem + \ Write destination address = mem + here\*4

swap ! \ Write cell

1 (h) +! ; \ Increment here by 1 (cell unit)

\ ===== Line breaks and spaces =====

: cr ( -- ) 10 emit ;

: space ( -- ) 32 emit ;

: spaces ( n -- ) 0 do 32 emit loop ;

\ ===== Convert numbers to decimal (simplified version) =====

\ Negative number support, zero support, simple implementation with internal temporary buffer

create \_buf 32 allot

: abs ( n -- u ) dup 0&lt; if negate then ;

: u/mod10 ( u -- u' d ) 10 /mod ; \ /mod is (a b--r q), so divide by 10

: (u.) ( u -- addr ) \ Convert u to a decimal string and return the address of the buffer's start

\_buf 31 + &gt;r \ r: end pointer

begin

u/mod10 \ u' d

48 + \ Using '0' as the reference

r@ c! r&gt; 1- &gt;r \ Write one character and return the pointer

dup 0=

until

drop r&gt; 1+ \ Start position (next to last written)

;

: . ( n -- )

dup 0&lt; if 45 emit abs then \ If negative, output '-'

(u.) ztype space ;

\ ===== Variables/constants (super simplified version) =====

\ vhere is the starting index of the data area. Use the space between last and vhere for simple allocation

: allot ( n -- ) \ Allocate n cells (handled by cell count)

(vh) @ swap + (vh) ! ;

: cells ( n -- n\*4 ) CELL \* ;

: variable ( -- a ) \ Allocates a single cell and returns its address

(vh) @ cells mem + \ Calculate the actual address

1 allot ;

: constant ( x -- ) \ Bind x to the name immediately following

create , does&gt; @ ;

point

* **, (comma)** means "mem + here\*4, where here is incremented by 1." In DWC, here is managed as a cell index, so we multiply by 4 to convert it to an address.
* mem and (h) are **internal variable literals** (addresses) **exposed by** the VM, so @ ! +! can be used.
* . (numeric display) performs a simple decimal conversion to output as a string using ztype.

**One more step (adding branch syntax)**

**To create immediate words** like IF ELSE THEN**, a mechanism to patch the jump destination at compile time is required.  
 DWC does not provide an operation word to add an IMMEDIATE flag to user-defined words, so** it is challenging **to reproduce simple IF/ELSE/THEN as immediate words** (although the VM has an IMMED flag, there is no API to access it from Forth).

Workaround:

* **Combinator approach** (high-order style): Create words that branch at runtime, such as cond true-xt false-xt choose
* **Manually patch "unresolved jumps"**: Prepare a mini-assembler in boot.fth that uses words like (mark) (resolve) and replaces the branch targets using , and @ !

(If necessary, I can rewrite this section as well)

**2) Implement using custom definitions (define on the fly in the REPL)**

This method involves adding words **one at a time using : … ;** in the REPL after startup.  
 If you include the minimum vocabulary in boot.fth, it becomes easier to add them in the REPL as well.

**Examples of useful small items (can be defined on the fly in the REPL)**

: 2dup ( a b -- a b a b ) over over ;

: 2drop ( a b -- ) drop drop ;

: =0 ( n -- f ) 0 = ;

: &lt;&gt; ( a b -- f ) = 0 = ;

: max ( a b -- m ) 2dup &lt; if swap then drop ;

: min ( a b -- m ) 2dup &gt; if swap then drop ;

: key? ( -- f ) ?key ;

: type ( a -- ) ztype ; \ Alias

: emit? ( c -- ) emit ; \ Alias

: delay ( ms -- ) ms ;

: .s ( ... -- ... ) \ Simple version that only checks TOS

(sp) @ (stk) + @ . ; \ Create a full .S file separately

Note

* Since **immediate words** cannot be created, it is difficult to reproduce standard Forth "syntax sugar" as-is.
* Nevertheless, **arithmetic, comparison, input/output, and** file **operations** are fully implemented as primitives, so it is sufficient for building applications.

**Which one to choose?**

* **Want to make each task easier** → 1) Set up boot.fth
* **Want to start small and iterate** → 2) Build up using the REPL  
   In actual production, the standard approach is to **first prototype in the REPL and then port it to boot.fth if it works well**.

If needed, you can create a **complete "minimal standard library (boot.fth)"** based on the above fragments.  
 You can prepare a single file that includes "numerical display, stack display, basic variables, simple branching combinators, and file utilities," but how much should be included?