まずはforth-dwcを理解したい

**ChatGPT:**

いいですね 👍  
まず「forth-dwc」を理解するために、全体像をまとめます。

**1. forth-dwc とは**

* **DWC = dword-code** の略
* Forth 系言語の極小実装で、
  + **VM (仮想マシン) が 250 行程度**
  + 実行ファイルは Windows で 17KB、Linux で 24KB 程度と非常に軽量。
* **3つのCファイル**で構成
  + dwc-vm.c … VMの中核 (inner/outer ループ)
  + dwc-vm.h … 定数・型・外部関数宣言
  + system.c … OS依存のI/OやREPL

**2. DWC の仕組み**

**命令の表現**

* **1命令 = 1 DWORD (32bit)**
  + 値が **32〜44以下 → プリミティブ** (VM内蔵命令)
  + 値が **0x4000\_0000 以上 → リテラル値** (上位ビットで判別)
  + それ以外 → 辞書に登録されたユーザ定義ワードのアドレス

**状態 (state)**

* **4種類の状態**を持ち、入力の解釈を切り替える
  + COMPILE : 入力をコンパイル
  + DEFINE : 新しいワードを辞書に追加
  + INTERPRET : 即時実行
  + COMMENT : コメント扱い
* 状態の切替は制御文字や : ; [ ] ( ) などで行う

**3. プリミティブ命令**

* **32個の基本プリミティブ** + **13個のシステムプリミティブ**
  + 例:
    - dup (スタック複製)
    - + (加算)
    - jmp (ジャンプ)
    - key / emit (I/O)
    - outer (C側の外部文字列をForth解釈)

**4. 実行の流れ**

1. main()（system.c）で dwcInit() → boot("boot.fth") 実行
2. boot.fth がロードされ、テキスト全体が outer() に渡る
3. outer() がテキストをトークンに分割し、inner() がVMで実行

**5. 特徴**

* **ColorForth 影響**  
  制御文字や (( )) コメントなど、独特の入力スタイル。
* **INLINE ワード**  
  マクロ的に展開できる。
* **Transient ワード** (t0〜t9)  
  辞書に残らない一時ワード。

**6. まとめると**

* forth-dwc は「**最小限のForth処理系をCで実装したVM**」
* **外部C関数 (emit, key, fOpen など) を呼べる**ので拡張が容易
* **outer() を介して文字列 → 辞書ワード列 → VM実行**というシンプルな構造

👉 質問ですが、次のステップとしては

* **プリミティブ一覧の動きを順番に理解したい**ですか？
* それとも **実際にForthコードを流し込んで遊んでみたい**ですか？

**あなた:**

forth-dwcを実行して、入力待ちになります。そのときに実行できることと実行できないことを教えて下さい。

**ChatGPT:**

思考時間: 1m 15s

了解！REPL（入力待ち状態）で「最低限“必ず”できること」と「そのままだと“できない/未定”のこと」をサクッと整理します。  
（※ forth‑dwc は超ミニマル。高機能は boot.fth 側の定義しだいです）

**いま実行できること（保証される最低限）**

**即時実行（INTERPRET）や定義**

* 状態切替の即時語がハードコード  
  :（DEFINEへ） / ;（定義終了しINTERPRETへ） / [（INTERPRETへ） / ]（COMPILEへ） / ( ) / (( ))（コメント）
* 新しい語の定義：  
  : STAR 42 emit ; など（emit は1文字出力）
* 行を一気に読み取り、トークンごとに解釈・実行（REPLは行入力→outer解釈）

**スタック＆制御**

* データスタック操作：dup drop swap over
* 算術：+ - \* /mod
* 比較/論理：< = > and or xor
* 制御用プリミティブ：jmp jmpz jmpnz、簡易ループ：for next  
  例）10個のアスタリスクを表示：  
  10 for 42 emit next

**メモリ/リターンスタック**

* メモリ：@ !（セル） / c@ c!（バイト）
* リターンスタック：>r r@ r>

**I/O/時間/ファイル**

* キー入力：key（1文字待ち受け） / ?key（押下有無 1/0）
* 出力：emit（1文字）/ ztype（ヌル終端文字列の出力、ポインタが要る）
* 時刻/待機：timer / ms
* ファイル：fopen fclose fread fwrite fseek

**その場で別の文字列を Forth として実行**

* outer（アドレスを渡すと、その文字列をその場で評価）  
  ※ REPL内では行を outer に渡して解釈しているのと同じノリ

**そのままだと “できない/未定” のこと（= boot.fth 次第）**

下の語は **VMのプリミティブに含まれません**。boot.fth で定義されていなければ使えません。  
必要なら自作して拡張する前提の設計です。

* 高水準制御構文：IF / ELSE / THEN / BEGIN / UNTIL / DO / LOOP など
* 数値出力・スタック表示：. .S など（emit/ztype しか無いので自作が必要）
* 文字列リテラル：." ..." や S" … "（未定義なら不可）
* 変数/配列/辞書系：VARIABLE/CONSTANT/CREATE/DOES>/ALLOT 等々
* 浮動小数点/例外/マルチタスク/語彙（VOCABULARY）などの高機能
* 標準Forthの即時語いろいろ（POSTPONE や IMMEDIATE など）
* システムコマンド実行（system）：ビルドや環境により使えない/危険な場合あり

**REPLのちょいメモ**

* プロンプト表示：  
  **COMPILE状態**なら ...、それ以外は ok が出ます（行単位で outer に渡されます）
* キー入力は key/?key を語として使う（REPL自体は行入力）
* 一時語 t0..t9 は辞書に残さず使い捨てできる（短い分解に便利）

**すぐ試せる“確実に動く”例**

（出力は文字だけに寄せています）

* 入力したキーをそのままエコー：  
  key dup emit drop
* 1秒待って ! を出す：  
  1000 ms 33 emit
* 10個の \* を出すループ：  
  10 for 42 emit next

もし「IF/ELSE」や「.（数値の表示）」を使いたければ、最小実装を足していくミニ辞書を一緒に作れます。  
まずは「表示用の ., .S」「数値→文字列」「簡単な IF/ELSE/THEN」から入れるのがおすすめです。

**あなた:**

回答をどうもありがとうございます。 では、forth-dwcで実行できないことを教えて下さい。

**ChatGPT:**

なるほど、「できないこと」を整理したいのですね。  
forth-dwc は **「VMのプリミティブ + boot.fthで定義される最小語」だけ**が存在するので、標準Forthの便利機能はほとんど欠けています。

ソース確認（dwc-vm.c の PRIMS 定義, README）から見える “できないこと” をリスト化します。

**forth-dwc でできないこと（boot.fthで定義しない限り）**

**1. 制御構文**

* IF … ELSE … THEN
* BEGIN … UNTIL / AGAIN
* DO … LOOP / +LOOP
* CASE … ENDCASE  
  → プリミティブに存在しない。ユーザが jmp/jmpz を組み合わせて定義する必要あり。

**2. 表示・デバッグ**

* 数値表示：.（スタックトップを10進で出す）
* スタック表示：.S
* 文字列リテラル：." hello" や S" abc"  
  → 出力系は emit と ztype のみ。

**3. メモリ管理・辞書操作**

* VARIABLE / CONSTANT / CREATE / ALLOT / DOES>
* 語彙管理（VOCABULARY）や FORGET  
  → 辞書操作は add-word と outer に限られる。

**4. 即時コンパイル系**

* IMMEDIATE
* POSTPONE
* コンパイル時の条件分岐（標準Forthの [IF] … [THEN] など）  
  → 状態遷移はハードコードされた : ; [ ] ( ) (( )) だけ。

**5. 数値表現・演算**

* 浮動小数点演算
* シフト演算（LSHIFT / RSHIFT）
* 論理NOT（0= 等）
* 文字コード変換系（CHAR など）  
  → 整数四則・比較・AND/OR/XORのみ。

**6. 環境系**

* マルチタスク
* 例外処理（CATCH/THROW）
* 標準入出力リダイレクトやファイル属性変更  
  → fopen/fclose/fread/fwrite/fseek 程度のCラッパしかない。

**7. 高水準ツール群**

* 辞書ブラウズ（WORDS）
* コメント表示やリスト化（SEE）
* アセンブラ／metaコンパイル  
  → 当然一切なし。

**言い換えると**

forth-dwc は **「スタック演算・メモリアクセス・ジャンプ・簡易ループ・キー入力/文字出力・ファイル入出力」しかできないVM**。  
それ以上の機能は **boot.fthや自作定義で積み上げる**設計です。

👉 次に進むなら：

* boot.fth の中身を見て「最低限どこまで標準Forthに近づけてあるか」を調べると、実際の「できないことリスト」が明確になります。

見てみますか？

**あなた:**

「forth-dwc は 「スタック演算・メモリアクセス・ジャンプ・簡易ループ・キー入力/文字出力・ファイル入出力」しかできないVM。 それ以上の機能は boot.fthや自作定義で積み上げる設計です。」 上記のこれらが教えていただいたことですが、これらは大切なことなので教えて下さい。 　１）boot.fthを使うか変更することで追加機能を実現する方法 　２）自作定義で積み上げることで追加機能を実現する方法

**ChatGPT:**

いい質問です！  
「追加機能」を入れる道は大きく **(1) boot.fth を整える** と **(2) REPLで自作ワードを積み上げる** の2本立て。どちらも、DWC が持っている最小プリミティブと内部変数をうまく使います。

根拠になる事実だけ先に：

* 使えるプリミティブ（dup + jmp jmpz ... key emit ztype fread ... など）は VM 側に固定で実装済み。
* VM は内部変数を「リテラル語」として公開しています（mem, mem-sz, base, state, (h) ここ=here, (vh) vhere, (l) last, >in 等）。初期化時に辞書へ登録されるため Forth から参照できます。
* 行入力は outer が解釈、: と ; はハードコードされた状態遷移で定義/終了します。

**1) boot.fth を使う／変更して追加機能を実現する**

目的：起動直後から “ふつうの Forth に近い便利語” が使える状態にする。  
方法：**最低限の「アセンブリ」に相当する語（`, here, allot, cr, . など）」を boot.fth に定義**しておきます。これで毎回の REPL を快適化できます。

**代表的な下回り語の実装例（boot.fth に入れる）**

（DWC の内部変数とプリミティブだけで作れます）

\ ===== cell と here の基本 =====

4 constant CELL

: here@ ( -- idx ) (h) @ ; \ here の “インデックス” 値を読む

: here! ( idx -- ) (h) ! ; \ here を書き換える

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

here@ CELL \* mem + \ 書き込み先アドレス = mem + here\*4

swap ! \ セル書き込み

1 (h) +! ; \ here を 1 進める（セル単位）

\ ===== 改行やスペース =====

: cr ( -- ) 10 emit ;

: space ( -- ) 32 emit ;

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

\ ===== 数値→10進表示（簡易版） =====

\ 負数対応、0 対応、内部に一時バッファを確保する簡易実装

create \_buf 32 allot

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

: u/mod10 ( u -- u' d ) 10 /mod ; \ /mod は (a b--r q) なので 10 で割る

: (u.) ( u -- addr ) \ u を 10進文字列にしてバッファ先頭のアドレスを返す

\_buf 31 + >r \ r: 末尾ポインタ

begin

u/mod10 \ u' d

48 + \ '0' を基準に

r@ c! r> 1- >r \ 1文字書いてポインタ戻す

dup 0=

until

drop r> 1+ \ 先頭位置（最後に書いた次）

;

: . ( n -- )

dup 0< if 45 emit abs then \ 負なら '-' を出す

(u.) ztype space ;

\ ===== 変数/定数（超簡易版） =====

\ vhere はデータ領域の先頭インデックス。last との間を使って単純確保

: allot ( n -- ) \ n セル確保（セル数で扱う）

(vh) @ swap + (vh) ! ;

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

: variable ( -- a ) \ セル1個の領域を確保して、そのアドレスを返す

(vh) @ cells mem + \ 実アドレスを算出

1 allot ;

: constant ( x -- ) \ 直後の名前に x を束縛

create , does> @ ;

ポイント

* **,（カンマ）** は「mem + here\*4 に書いて here を+1」。DWC の here は**セルのインデックス**で管理されているため、アドレスへ直すときに \* 4 しています。
* mem と (h) は VM が公開している**内部変数リテラル**（アドレス）なので @ ! +! が使えます。
* . （数値表示） は ztype で文字列を出すために、簡易の 10 進変換を自前で組んでいます。

**もう一歩（分岐構文を足したい）**

IF ELSE THEN のような**即時語（IMMEDIATE）を作るには、本来「コンパイル時にジャンプ先をパッチする」仕組みが要ります。  
DWC は ユーザ定義語に IMMEDIATE フラグを付ける操作語が未提供 なので、素朴な IF/ELSE/THEN を即時語として再現するのは難しめ**です（VM側には IMMED フラグはあるが Forth から触る API が無い）。

回避策：

* **コンビネータ方式**（高階風）：cond true-xt false-xt choose のように「実行時に分岐する語」を作る
* **“未解決ジャンプ” を手動でパッチ**：(mark) (resolve) のような語で , と @ ! を使って分岐先を書き換えるミニアセンブラを boot.fth に用意

（必要ならこの部分も書き起こします）

**2) 自作定義を積み上げて実現する（REPL で都度 : 定義）**

起動後に REPL で **都度 : … ; を使って足していく**方法です。  
boot.fth に入れた最小語があれば、REPL でも追加しやすくなります。

**すぐ使える小物の例（REPLで都度定義してOK）**

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

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

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

: <> ( a b -- f ) = 0 = ;

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

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

: key? ( -- f ) ?key ;

: type ( a -- ) ztype ; \ エイリアス

: emit? ( c -- ) emit ; \ エイリアス

: delay ( ms -- ) ms ;

: .s ( ... -- ... ) \ とりあえずTOSだけ覗く簡易版

(sp) @ (stk) + @ . ; \ 本格的な .S は別途作る

注意

* **即時語が作れない**ため、標準Forthの “構文糖” をそのまま再現するのは難しいです。
* それでも、**算術・比較・入出力・ファイル**はプリミティブで完備しているので、アプリを組むのは十分可能です。

**どっちを選ぶ？**

* **毎回の作業を楽にしたい** → 1) boot.fthを整える
* **試行錯誤で最小から育てたい** → 2) REPLで積む  
  実運用では、**まずREPLで試作 → うまくいったら boot.fth に移植** が王道です。

必要なら、上の断片をベースに **完成度の高い “minimal 標準ライブラリ（boot.fth）”** を一式作ります。  
「数値表示・スタック表示・基本変数・簡易分岐コンビネータ・ファイルユーティリティ」までをワンファイルにまとめる形で用意できますが、どこまで入れますか？