

# 1. FORTH が生まれた世界

---

1970 年代のミニコン上のプログラミング環境を感じられるプロセスを、今風の自作ワンボードマイコンと母艦 PC の組み合わせの上に実現し、68000 機械語・アセンブリ言語で FORTH のワードを整えてゆきます。

私にとって FORTH の魅力は、言語としての側面よりも、実行環境としての側面にある。

- サブルーチンのアドレスを 2 重ポインタでたどって呼び出してゆく、シンプルな「インタプリタ」。
- 機械語サブルーチンではあるが、部品化して組み合わせるための仕組みである「単語」「辞書」「コンパイル」
- これらの単純な仕組みの組み合わせで「アプリケーション志向」でアプリケーションにくみ上げてゆくプログラミングスタイル。

UNIX でプログラミングに目覚め「小さな部品を組み合わせで問題を解いてゆく」スタイルがしみついている、今でもシェルスクリプトでバッククォート、sed を日常的に使う私にとって、もうひとつの「部品を組み合わせで問題を解いてゆく」仕組みとして魅力的である。UNIX が無い時代の「部品箱」アプローチはとても興味深い。

FORTH を発明した Charles Moore は、機械語サブルーチンの束とインタプリタを引っ提げて、顧客の求めるミニコンに開発環境を一から移植し、その上にアプリケーションを組んだ。この作業を数 10 種類のミニコンに対して繰り返す中で、FORTH 言語が形作られていった。

ならば、私も彼のたどった道をたどろう。新しい CPU に対して、アセンブリ言語を使って機械語サブルーチンの一つ一つ作ってゆく。最終的に FORTH 言語が動くようにする。彼が数 10 回たどった道だが、私がさらに 1 回たどってもいいじゃないか。このシンプルな開発環境・実行環境を体で味わいたい。ということだ。

- サブルーチン先頭のアドレスの列をソフトウェア部品として実行する。実行は 2 重ポインタの参照なので実行コードへのジャンプは高速である。
- プログラムはテキストで表現されているが、アドレスの列に変換した状態でメモリ上に展開しているので高速実行できる。
- テキストを解釈する際は空白で区切られた「単語」単位に区切り、単語に対応する部品のアドレスを見つけ出し実行する。

幾つかの機械語部品以外は、すべて数 10 個のアドレスの列(ポインタの配列)なのでコンパクトであり、当時の限られたメモリ容量でもある程度の規模のアプリケーションを組むことが出来た。

数キロバイトのメモリ、ハードディスクと低速の端末、機械語によるプログラミングで腕を磨いた Moore は、自分独自の開発環境を作り上げてゆく、さまざまなマシンの仕事を受け手元のソフトウェア部品をインタプリタで組み合わせ仕事を行なう中で彼が作り出したものが FORTH という開発環境だった。60 年後半から 70 年代前半にはコンパイラやライブラリは提供されているがコード品質はよくわからない状況で、自分の手元にある機械語サブルーチンの束を組み合わせ顧客要望に応える。新たな CPU の仕事を受けるとアセンブラを作るところから始め、用意した機械語ルーチン群を「移植」してゆく。数 10 種類のコンピュータに対してそれを繰り返す中で FORTH という仕組みが出来上がっていった。

- アドレス解決済のサブルーチン呼び出しの列を順次呼び出してゆく実行方法(内部インタプリタ)
- 入力プログラムを単語に区切って、単語を名前-コードのペアのエントリの片方向リスト(辞書)で検索して見つかったコードのアドレスを実行する(外部インタプリタ)
- 単語の列で構成されたモジュールを、辞書で検索してコードのアドレスをメモリ上に順次並べ、そのリストを新しい辞書エントリとする(コンパイル)

という処理をベースとして、

FORTH として最も初期の記述は(Moore74)"FORTH: A NEW WAY TO PROGRAM A MINI-COMPUTER"にある。RAM 8k バイト、ハードディスクと端末のつながったミニコンピュータ、フロントパネルのスイッチを操作して、数 10 バイトのブートローダを 1 バイト 1 バイト手打ちで入力して、紙テープからコンパイラを読み込ませ実行する環境。おそらく、FORTH 実行環境は紙テープに打ち込んでおき、手打ちブートローダから読み込んで RUN していたのでしょう。

FORTH 実行環境は 2k バイトだとのこと。現在の我々の住まいの 1000 分の 1、ひょっとすると 100 万分の一かもしれない。アプリケーションも数キロバイト、データ領域も数キロバイトの規模感である。現在を生きる我々がこの規模感の世界で何か役に立つ・実用的なものを開発すること自体に意味がないことは、私自身も承知している。だが、Moore はこの環境で天文台向けアプリケーション(おそらく、望遠鏡を天空の一点に向けながら写真を撮像するようなものなのだろう)を開発し、当時の天文台の人気を博したというのだ。小さな部品の組み合わせ、言語ではあるが、既にある部品とアプリケーション用の部品を追加で開発し組み合わせる。2kB のメモリでも動作する「インタプリタ」と「コンパイラ」なら、我々の世代なら VTL や Tiny BASIC を思い浮かべるだろう。だが、FORTH には「ソース読み込み・解釈部」自体も幾つかの部品の組み合わせでできており、ここをカスタマイズすることもできる。例えば、単語の文字並びの定義も変えることができる。あるアプリケーションの機能を単語として定義してそろえてゆくだけでなく、実際にアプリケーションを実行する際の「入力」の構文も専用化することができる。この辺りは他の言語・実行環境ではなかなかできるものではない。

Moore は「ジョン・マッカーシから Lisp を学んだ」ことがあると言う。1960 年に学士取得後にスタンフォードで 2 年間数学を学んだ時のことらしい。Lisp のインタラクティブ性、(S 式の範囲ではあるが)パーザも自作できる環境についても知っていたと推測する。

IBM704 で Fortran II を使用(1958 年)。このプログラムをアセンブラに圧縮し、衛星軌道を決定(1959 年)。一方、スタンフォード線形加速器センターで電子ビームのステアリングを最適化するためにバロウズ B5500 用の Algol を習得(1962 年)。Charles H Moore and Associates として、タイムシェアリングサービスをサポートするために Fortran-Algol トランスレータを作成しました(1964 年)。また、最初のミニコンピュータでリアルタイムのガスクロマトグラフをプログラミングした(1965 年)。モハスコ社でオーダーエントリネットワークをプログラムするために Cobol を習得した(1968)。[PROGRAMMING A PROBLEM ORIENTED LANGUAGE, Moore70b]

FORTH を発明するまでにこれだけの経験を積んでいた。Fortran/Algol 言語に熟知し、最終的にアセンブリ言語で書くことを躊躇しない(というか、アセンブリ言語で書かざるをえなかった)し、実際にアセンブリ言語で書いて仕事できたのだろう。

## FORTH の誕生

[Evolution-FORTH]には Moore が FORTH を誕生させた経緯について説明がある。

1960 年に、スミソニアン天文台で天文関係の計算を行うプログラムを開発していた。彼自身が開発・蓄積したプログラムの「構成管理」のためのインタプリタを自分で開発したというのが、のちの Forth につながっているらしい。

「カードトレイ 2 枚を埋め尽くす」規模に成長した彼のプログラムの再コンパイルの手間と時間を最小にするために、プログラムを制御するカードを読み取る簡単なインタプリタを開発した。これにより、彼は再コンパイルすることなく、複数の衛星のために異なる方程式を構成することができた。

このインタプリタには、現代の Forth に受け継がれているいくつかのコマンドとコンセプトがあり、主にスペースで区切られた「単語」を読むコマンドと、数字を外部形式から内部形式に変換するコマンド、それに IF ... ELSE 構成がある。

彼は、自由形式の入力が、特定の列にフォーマットするという、より一般的な Fortran のやり方、これは列間違いにより果てなく再実行を繰り返す結果になるやり方よりも効率的で(より小さく、より速いコード)、信頼できるものであることを発見しました。

その背景には、既存のプログラムに対する不信と、

## FORTH の発展

スタンフォード時代に、スタンフォード線形加速器センターでプログラムを書き、2 マイル電子加速器のビームステアリングを最適化したという。

そのプログラムを制御するために、彼は自分のインタプリタを拡張し、パラメータ渡しのためのプッシュダウンスタック、明示的に値を取得・保存できる変数、算術・比較演算子、手続きを定義・解釈する機能などを管理できるようにしたものを使用した。

1965 年からはフリーのプログラム書きとして様々な仕事をこなした。都度、インタプリタを活用していた。

60 年代終盤にはミニコンピュータが登場し、それに伴ってテレタイプ端末が登場したが、ムーアはこの端末用に演算子を追加して文字の入出力を管理するようになった。

1968 年、モハスコインダストリーズ社に入社し、そこで、IBM1130 ミニコンピュータ+2250 グラフィックディスプレイ用のコンピュータグラフィックスプログラムを開発した。

2250 用のコードを生成するために、プログラムにクロスアセンブラを追加し、さらにプリミティブエディタとソース管理ツールも追加しました。このシステムは、IBM のソフトウェアが静的な 2D 画像しか描けなかった時代に、アニメーションの 3D 画像を描くことができた。また、遊びで、初期のビデオゲーム「Spacewar」を書いたり、「Algol Chess」プログラムを、初めて「FORTH」と呼ばれる新しい言語に変換したりもした。彼は、FORTH が非常にシンプルになったことに感動した。

このころ、彼が育ててきたインタプリタ(もはや、実行環境といってもいいだろう)は Forth という名前を持つことになった。

ムーア氏は、2250 をプログラミングするための Forth ベースの 1130 環境が、1130 のソフトウェアが開発された Fortran 環境よりも優れていることに気づき、1130 コンパイラに拡張した。その結果、ループコマンド、ソースを 1024 バイトのブロックに分けて管理する

概念とその管理ツールなど、現在 Forth で認識されているコンパイラの機能のほとんどが追加された。

最も重要なのは、辞書が追加されたことだ。手続きは名前を持つようになり、インタプリタは名前のリンクリストを検索して一致するものを探しました。これはスタンフォード大学のコンパイラから学んだことで、1980 年代まで Forth で広く使われていた。

1130 システム上で、辞書・コンパイラが確立してゆく。また、リターンスタックも入った。

最後に、ルーチンをネストするための簡単なメカニズムを提供するために、「リターンスタック」と呼ばれる第 2 のスタックが追加された。リターンアドレス用に予約されたスタックを持つことの利点は、呼び出しの前後に「バランス」を取る必要がなく、もう 1 つのスタックをパラメータ受け渡しに自由に使えることであった。

モハスコ社でのプロジェクトと中断、この時代に「タスクを定義し管理する仕組み」「現在使われている仕組みに似たディスクブロックバッファ」が導入されている。

Forth に関する最初の論文はモハスコ社で書かれた [Moore,1970a]。1970 年、モハスコ社は、新しい Univac 1108 で受注システムのための専用線ネットワークを処理するという野心的なプロジェクトを Moore 氏に任せた。彼は、Forth を 1108 に移植し、トランザクション処理を行う COBOL モジュールとのインタフェースを確保した。1108 の Forth は、アセンブラでコード化されていた。入出力のメッセージをバッファリングし、各行を処理するタスク間で CPU を共有する。また、入力を解釈し、適切な COBOL モジュールを実行した。このバージョンの Forth には、タスクを定義し管理する仕組みや、現在使われている仕組みに似たディスクブロックバッファを効率的に管理する仕組みが追加されていた。

このころ、[PROGRAMMING A PROBLEM ORIENTED LANGUAGE, Moore, 70b]を書いた。

しかし、不況のあおりを受けて、モハスコ社は 1108 プロジェクトの完成を待たずに中止を決定した。Moore は直ちにその旨を伝え、怒りの詩を書き、出版されることのなかった Forth に関する本 [Moore, 1970b]を書きました。それは、Forth ソフトウェアの開発方法を説明し、シンプルさと革新性を奨励するものでした。

私がこの本を書こうと思ったきっかけである。1970 年、万国博覧会が万博と呼ばれていたころ、アメリカ東海岸のできごとである。C 言語もない(C 言語誕生は 1972 年である)、コンピュータ付属のコンパイラ・ライブラリが玉石混交だった(ドラゴンブック第 1 版の発売が 1986 年であった。その前 10 年程度は研究テーマとなるくらいで、市井のプログラム書きがコンパイラ理論など知っているはずがない、そんな時代につくられたツールの品質たるやどんなものなのだったろうか)ころ、「人が作ったソフトウェアは信用できない、自分の手で自分のアプリケーションに必要な機能を構築するのだ」という意思に基づいて生まれたソフトウェア構築環境、それが Forth だった。

## 2. Forth の「基本原則」

---

Moore にとって、Forth の誕生は、当時のソフトウェアツールに対する不満に対するためのものだった。

アセンブラはコンパイラとスーパーバイザを記述するための言語、スーパーバイザはジョブ制御のための言語、コンパイラはアプリケーションプログラムのための言語、アプリケーションプログラムはその入力のための言語を定義しているのです。ユーザはこれらすべての言語を知らないかもしれませんが、知ることもできないかもしれません。しかし、これらの言語は、ユーザとコンピュータの間に立ち、ユーザができることとそのコストに制約を課しているのである。

この膨大な言語の階層を作るには、人と機械の膨大な時間が必要であり、保守するにも同様に大きな労力を必要とします。これらのプログラムを文書化するのにも、その文書を読むのにも膨大なコストがかかる。そして、これだけ努力しても、プログラムはバグだらけで、使いづらく、誰も満足しないのである。

アセンブリ言語、ジョブ制御言語、アプリケーションプログラムのための言語、これらの言語を覚えて使うことがユーザに金と時間を使わせている。そのうえ、出来上がったプログラムはバグだらけで使いづらく誰も満足していない。

「この膨大な言語の階層」を Forth という一層だけの構造に置き換え、最小限のアプリケーションインタフェースとプログラマとのインタフェースを提供する。これが Moore の考える Forth であった。

これは、「膨大な言語の階層」を当たり前としてそのうえで暮らす私には新鮮な考えだった。というか、古い・古すぎる、現代とのギャップが大きすぎる。

一方で、Forth の言語処理系、特にインタプリタの実装は面白かった。40 年前にプログラム書きを始めた当時の 8bit 「パソコン」「マイコン」を使っていたときの経験から、メモリ 16kByte でも十分実用になる単純さ・効率の良さは納得感のあるものだった。いくつかの Forth 入門書を読んで感じる特徴「浮動小数点数を導入せず、整数×整数÷整数の演算子を導入する(これは 1 未満の大きさの桁数を処理系任せにせず自分で管理することを意味すると考えた)、コンパイラ自身も書き換えることができる、その機能を使って「アプリケーション固有の言語を作れ」というアピール。

Evolution of Forth や、PROGRAMMING PROBLEM-ORIENTED LANGUAGE という Forth の歴史書を読んで、この奇妙な言語は当時のコンピュータ環境に直面した Moore が自分なりに取り組んだ結果・回答なのだと分かった。

その結果、Moore74b には、私から見て一見みょうちくりんな主張が連発している。ここでは、それを説明し、私なりの解釈を添える。

### Keep it Simple!

Moore 師匠の「基本原則」が Keep it Simple! である。この「基本原則」は古今を通じてプログラム書きの中で受け継がれているもので、この原則自体にみょうちくりんなところはない。

プログラムに追加する機能の数が増えれば増えるほど、プログラムの複雑さは指数関数的に増していく。プログラムの内部的な整合性はもちろんのこと、これらの機能間の互換性

を維持する問題は、簡単に手に負えなくなる。基本原則を適用すれば、これを回避することができます。基本原則を無視したオペレーティングシステムをご存じかもしれませんね。

これは良いですね。最後の一文を除くと納得感あります。

基本原則を適用するのは非常に難しいことです。内外のあらゆる圧力が、プログラムに機能を追加しようと謀っているのです。結局のところ、半ダースの命令しか必要ないのだから、なぜそうしないのか？唯一の対抗力は「基本原理」であり、これを無視すれば対抗力は存在しない。

使われるかもしれないコードをプログラムに入れてはいけない。拡張機能をぶら下げるようなフックを置いてはいけない。あなたがやりたいと思うことは無限であり、それぞれが実現する確率は0である。もし後で拡張機能が必要になったら、後でコーディングすればいいのです - そしておそらく、今やるよりも良い仕事ができるはずです。そして、もし他の誰かがその拡張機能を追加したら、その人はあなたが残したフックに気づくでしょうか？あなたのプログラムのこの点を文書化しますか？

「使われるかもしれないコードをプログラムに入れてはいけない」この一文は重い。将来の拡張性、汎用性を常に意識しながらコードを書いてきた私だが、それは止めろと言われている。ちょっと待て！と言いたい気分だ。

「もし後で拡張機能が必要になったら、後で(そのときに)コーディングすればいいのです」これは、「あらかじめ要件を定め機能を定め作る」というアプローチと全く適合しない。プログラムを開発する過程で必要性に気づいた機能はその時点で作ればよいのだから、という意味なのだろうから。

だが、時間と締め切りの制約がないプログラミングだと、むしろこのアプローチの方がすすがしい。遠い昔、時間と締め切りの制約のないプログラミング、プログラミングの中にプログラミングを忘れる経験をしたことがあるなら、この言葉の意味も実感できるかもしれない。

この辺りを読んだ時点で、Forth というものは単純に言語処理系・単純なインタプリタとして見てしまっただけでいい、もっと深いものがあるのではないかという気になっていた。

## Do It Yourself!

この「なんでも自分でやれ」は結構恐ろしいことを言っている。Moore78の最初に書かれている基本原則の系「Do It Yourself!」には「標準的なライブラリを使うべきという通例に従わずに、自分でサブルーチンを書きなさい」とある。産まれたときから libc を当たり前のように使ってきた私としては「なんじゃこりゃー」である。

Moore 師匠曰く、

自分でサブルーチンを書く前に、書き方を知らなければならない。これは、現実的には、以前に書いたことがあるということであり、そのため、始めるのが難しいのです。しかし、一度試してみてください。同じサブルーチンを何台ものコンピュータと言語で何十回も書いていくうちに、かなり上手になるはずだ

ということなのだ。最初は難しいが、何度も書いていけばそれで書けるようになるというのだろう。そういう経験ならば私にもある。さすがに、アセンブリ言語の世界でそれを実践したことはないが。

有言実行、Moore 師匠は「これを驚くほど忠実に実行した」のだそう。



ムーアは、これを驚くほど忠実に実行した。1970年代を通じて、彼は18種類のCPUにForthを実装し(表1)、必ずそれぞれに独自のアセンブラ、独自のディスクおよびターミナルドライバ、さらには独自の乗除算サブルーチン(多くのマシンで必要とされていた)を書きました。これらの関数についてメーカー提供のルーチンがある場合、彼はそれを読んでアイデアを得ましたが、そのまま使うことはありませんでした。Forthがこれらのリソースをどのように使うかを正確に把握し、フックや一般性を省き、また、熟練と経験によって(彼は、ほとんどの乗除算サブルーチンは、これまで書いたことがなく、これからも書くことがない人が書いたと推測しています)、彼のバージョンは常に小さく、速く、通常は著しく速くなりました。

CPU 18種類を渡り歩き、それぞれに自作アセンブラ、自作のディスクドライバ、自作のターミナルドライバ、自作の乗除算サブルーチンを用意したというのだ。もちろんアセンブリ言語で記述した。

C言語が使える現在の我々は、全て自作でそろえるにしてもC言語で書いたモジュールを使いまわすでしょう。事実シリアルI/OやSDcard/FatFSルーチンはそうしている。誰かが十分効率的な(コード効率がいい、くそくなコードを吐く)C言語処理系をそのCPUのために用意してくれることを疑うことはない。また、GCCのCPUサポート数の多さもそれを後押ししている。何年も、10何年もの蓄積の上になりたっている高品質のソフトウェアの上で私たちは生活することができている。

しかしながら1970年代、万国博覧会が万博と呼ばれていたころ、C言語もGCCもなかった。ドライバも乗除算ルーチンも全てアセンブリ言語で書くしかない。一方でCPU付属のアセンブラも乗除算ルーチンも結構怪しいものだったのだろう。乗除算ルーチンも結構ノウハウの塊であり、正しく計算できるレベルからがりがりにチューンして高速化を図った結果のコードとは性能差は歴然だっただろう。

数回自分でやっているうちに、ディスクドライバもターミナルドライバも乗除算ルーチンも何度か書いて勘所はわかっている。一方でメーカ提供のアレはあそこかこことか配慮がないなあ、怪しいコードや。これなら自分でやった方が速くてまともなものができる。あたりがMoore師匠のお考えではなかったのか。

Forthという言語処理系ができた時代の環境・バックグラウンドは現在とだいぶ違うらしい。18種類のCPUの全てに対して、「全部自分でやった」結果、研ぎ澄まされた処理系がForthではないのか？そんな気がするのだ。

アセンブリ言語だけしか使えない環境下でディスクI/OもシリアルI/Oも天文アプリケーションも動くようにする。全部がForth言語処理系の中にある。なるほど、面白い。

## 3. Forth 処理系

---

この章では、Forth 処理系を簡単に説明する。Forth は「言語」なのだが、文法や意味(Syntax and Semantics)という枠組みで説明したくない。というか、文法と意味という枠組みに対応するプロセッサは存在しない。入力として FORTH「言語」で書かれたプログラムがどのようにして解釈実行されるのか、という実行系を説明のメインに置く。

Forth 処理系は、トークナイザ、パーザという枠組みでソースコードを解釈しない。ソースコードを受け取った Forth 処理系は、ソースコードからワードを一つきりだし、それを実行するだけである。「IF ... ELSE ... THEN」「DO ... WHILE」などの制御構造は利用できるが、それは、IF というワード、ELSE というワード、THEN というワードがそれぞれの処理を行うだけのことで、IF ... ELSE ... THEN という構造自体を切り出して処理するわけではない。

「文法を満たさないプログラム」を Forth 処理系は撥ねない。文法エラーだから処理しませんにならないのだ。THEN で終端されていない IF 文であってもそのまま実行して訳の分からないことになるだけである。プログラムをざっくり書いて文法チェックをコンパイラに任せている私には到底ついていけない厳しさなのだ。こんな言語処理系の「文法」を理解して意味があるのだろうか。

この本のテーマは、Moore 師匠の旅路をたどることにある。具体的には「19 番目の新規の CPU の上で処理系を動かさねばならなくなった Moore 師匠に代わって私が処理系を『実装』する」ので、実装対象のプログラムである処理系のイメージを読者の皆様にも伝えておきたい。ここでそれを行う。

### Forth 処理系の哲学

Moore74 の最初、「Philosophy」の章に処理系の背景と概要が書いてある。

時は 70 年代初頭、ミニコンピュータが台頭してきていたが、その開発環境は貧弱なものだった。高級言語を効率的に実行することができず、アセンブリ言語でプログラミングすることになる。手練れのプログラマーと多大な労力が必要であり、コスト、開発期間、品質に不満を持たれていた。Forth はこの課題に対する一つの方向を示している、そう Moore 師匠は言うのだ。

FORTH は、コンピュータとの効果的なコミュニケーションという問題に対する新しいアプローチです。FORTH は、動詞、名詞、定義語を要素とする英語風の言語である。動詞はコンピュータの一連の操作を引き起こし、名詞は操作される対象であり、定義語は以前に定義された言葉の観点から、あるいは機械命令の観点から、新しい言葉を定義させる。

言語としては、動詞、名詞、定義語を持つと言っている。定義語は Forth ならではの特徴あるワードであり、変数定義、定数定義の単語は標準的に提供される。「定義語を構成する機能を持つワード」が提供されるので、ユーザが自分で自分用の定義語を作ることができる。

FORTH は、プログラマーが自分の特定の問題を記述するために拡張できる基本的な語彙を提供する。基本語とは、語彙を構成し、並べ替え、テストするために必要な語であり、簡単で便利なものである。このように FORTH は、通常ならば、アセンブリ言語、コンパイラ言語、ジョブコントローラ言語、アプリケーション言語を必要とする範囲をこれ一つでカバーする言語を提供します。



アセンブリ言語、コンパイラ言語、ジョブコントローラ言語、アプリケーション言語をまとめてカバーする単一の言語。ワードを対話的に実行・定義できるので、今の言葉で言えば統合開発環境なのである。Lisp 処理系や、Unix シェルと同列に置かれる機能を持っていると思う。

FORTH は多くのレベルで有用です。プログラマにとっては、コアに常駐するアセンブラとコンパイラを提供し、ソースから再コンパイルすることでオブジェクトプログラムをロードする必要がありません。このような機能は、通常、非常に大きなメモリを持つ大型コンピュータにしかないものです。また、プログラマには、プログラムの作成と修正を容易にするテキストエディタが用意されています。また、独立した(複数のプログラムを持つ)タスクを簡単に記述する方法を提供します。

常駐するアセンブラ、コンパイラにより、ワードの定義・実行という開発サイクルを短時間で回すことができる。開発サイクルのスピードを見れば、現代の我々がソースコード修正-コンパイル-実行で享受するサイクルのスピードに近いサイクルで回せるということなのだろう。短時間で回すことができることは、今も昔も重要なことなのだろうと思う。

テキストエディタも提供されている。テキストバッファ上の固定長の行の並びから、行番号を指定して追加、削除、入替を行うワードが提供されており、Unix の ed のようなラインエディタである。テキストバッファは 64 文字×16 行を一単位とする。この単位でハードディスクに読み書きする機能も提供される。

ユーザにとっては、要求の厳しいリアルタイムアプリケーションも扱える効率的な語彙と、込み入った状況を診断する際に特に有用な広範な語彙を提供してくれます。さらに、必要に応じてデータブロックをディスクからコアへ、またディスクへ自動的に移動させるディスクルーチンにより、コアが提供するよりもはるかに多くのメモリに簡単にアクセスできるようになります。

ディスクアクセスは、1024 バイト単位のブロック番号を指定してディスクバッファに読み書きするワードが提供されている。ユーザは読み込んだデータの参照(ブロック内へのメモリアクセスを使用する)、書き換え(メモリ書き換えに加え、そのバッファが dirty になったことを保持する)、書き出しが行える。ユーザが明示的に書き出すほかに、新たなブロックを読み込むバッファがなくなると、既に埋まっているバッファを開放し(dirty ビットが立っている場合は裏でディスクへの書き込みを行う)、空のバッファを確保する。

今回の旅路では、「アセンブラ」「テキストエディタ」「ディスクアクセス」機能は扱わない。ホストマシンの組み合わせ、ホストマシン側でのアセンブリ言語コード開発を行うためである。はっきり言って「ずる」だが、さすがにラインエディタとファイルシステムの無いブロック番号管理のディスク上でのソースコード管理はご容赦頂きたい。

## Forth 処理系の概要

Moore74b によると、Forth には重要な要素が 5 つあるという。それらは、

- 辞書(Dictionary)
- スタック
- インタプリタ
- アセンブラ
- 二次記憶(Secondary Memory)

だそう。ここでは、Moore74b に従ってこの 5 つを説明してゆく。

Forth は基本的にボキャブラリ(ワードの集合)であり、そのボキャブラリの中で何が「ワード」を構成しているのかを理解することが重要です。

ワードとは、空白で区切られた任意の文字列のことです。ワードを構成できない特別な文字や、ワードを開始できない文字はありません。したがって、算術演算子を表す文字や、句読点に似た文字も、空白で区切られていればワードとなり得る。例えば、次のようなものがワードです。

```
FORTH re-start + IF, ? */. 2@ 3.14 1-0
```

## 辞書(Dictionary)

Forth 言語は辞書によって構成されています。これは、プログラムが使用するメモリのほとんどすべてを占めます。その機能は、ボキャブラリの中の各ワードの定義を提供することである。これには、動詞が行うべき操作も含まれる。また、この辞書には、最も頻繁に参照される名詞も含まれている。

Forth 処理系は辞書の中のワードを見つける機能、ワードを追加する機能、コンテキストにより辞書の一部分を選択する機能を持っている。

ワードは「定義語」によって辞書に追加されるが、その中でも特によく使われるのは3つである。1つは;**(セミコロン)**であり、端末からのメッセージの中で遭遇したとき、その直後のワードを辞書に登録する。それ以前に定義された単語を用いたこのワードの定義もまた、辞書に登録される。たとえば

```
: APPLE MEDIUM SIZE ROUND RED FRUIT;
```

は、APPLE の定義です。

もう一つの定義語は**CODE**である。これは、その次のワードが機械語命令によって定義されることを示すもので、この機械語命令も辞書に登録されます。例については後述します。

3つ目の定義語は**INTEGER**で、これも同様に次のワードを辞書に登録する。しかし、この場合、ワードは名詞であり、その値のために辞書の位置が確保される。たとえば

```
100 INTEGER DATA
```

は名詞 DATA を定義し、その値を 100 に設定します。

## スタック

プッシュダウン式のスタックは、ユーザとユーザが使うワード、およびあるワードと別のワードの間のデータのやり取り(communication)を提供します。例えば、ユーザは数字(あるいは他のオブ

ジェクト)をスタックに置き、次にあるワードを打ち込むことで、スタック上で作用して望みの結果を得ます。例えば、次のように

```
12 2400 * 45 / .
```

と入力すると、12 と 2400 という数字をスタックに置き、それらを掛け合わせ (ワード\*)、45 をスタックの一番上に置き、前の結果を 45 で割って(/)、その結果をタイプ(.)します。

このスタック(データスタック)とは別にリターンスタックというものもある。

## インタプリタ

FORTHには2つのインタプリタがある。高レベルのインタプリタと低レベルのインタプリタである。

高レベルのインタプリタは、端末から単語を読み、辞書を検索し、見つけたエントリを実行する。

低レベルのインタプリタは、他のワードにより定義されたワードを実行する。先ほどのAPPLEの定義がその例になる。低レベルのインタプリタはAPPLEを見ると、MEDIUM、SIZE、ROUND、RED、FRUIT、';'の順に実行することになる。

これらの単語はすでにコンパイルされているので、この作業は極めて単純である。つまり、APPLEという単語を定義するときに、その定義に含まれる各単語を辞書で検索し、その結果の辞書アドレスをAPPLEのエントリに配置してある。APPLEが実行されたとき、インタプリタはこれらのアドレスをたどって必要なコードを実行すればよい。APPLEを定義するテキストは保存されない。

低レベルインタプリタには、いくつかの重要な特性がある。まず、高速であること。実際、いくつかのコンピュータでは、ワードそのものが意味するコードに加えて、追加で実行される命令数はワード一つにつきわずか2命令でしかない。第二に、定義がコンパクトになる。定義で使われる各ワードがコアで占める場所は1箇所だけである。あるワードを他のワードで定義する場合、その定義を解釈するコンピュータに依存しないためである。

その結果、FORTHのボキャブラリのほとんどは低レベルのインタープリタによって定義され、解釈されることになる。高レベルのインタプリタ自体も、別のワードを用いた定義である。

## アセンブラ

アセンブラについては、Moore74bの記述をそのまま示しておく。

CODEという定義語を使って、プログラマは指定された機械語命令を実行させる語を定義することができる。このような定義は、I/Oの実行、算術演算の実装、その他機械に依存する処理を行うのに必要である。

これは、FORTHの重要な特徴である。FORTHは、コンピュータに依存したコードを、管理しやすい大きさに分割し、特定のインターフェイスの規則で明示することを可能にします。アプリケーションを別のコンピュータに移植するには、CODE語だけを再コード化すればよく、他の語とはコンピュータに依存しない形で相互作用します。

アセンブラが解釈する言葉の中には、命令ニーモニックがあります。各ニーモニックは、対応する機械語命令をアセンブルするように定義されている。例えば、次のようなフレーズ

```
X LDA,
```

は、LDA 命令を X の適切なアドレスでアセンブルし、辞書に置く。

このアセンブラは非常にコンパクトです。その主なコストは、ニーモニックが占める辞書のスペースで、通常 250 コアロケーションです。プッシュダウン・スタックは、従来のアセンブラで必要だったシンボルテーブルを大幅に削減する。これは、コアの位置に名前を付ける必要がないためです。動詞は、名前を付けた場所からパラメータを取り出すのではなく、スタック上でパラメータを見つけます。同様に、条件付きジャンプは、後で説明する方法で、アセンブル時にプッシュダウンスタックを使用します。ただし、変数とロケーションには名前を付けることができ、そのような名前は通常通り辞書で見つけることができます。

機械語で記述したサブルーチンを含むワードは必要不可欠なので、処理系全てをメモリ上に載せるためには、オンコアアセンブラが必要というのは理解できる。が、ニーモニック辞書が 250 ワードで収まり、オペランドの代わりにスタック中の位置を指す、条件付きジャンプの飛び先をリターンスタックを使って保持しておく。我々が普通に知っているアセンブリ言語とだいぶ異なるのではないかという気がする。

Moore74b では 5 大項目の一つとして扱われるアセンブラであるが、その後の Forth の展開の中では重要でなくなったせいか、あまり見かけることはなくなった。いくつかの歴史的文献をあさっているが、アセンブラの具体的実装をまだ見つけれられていない。私自身もよくわかってないので、本書ではこれ以上触れることはしない。残念だが。

## 2 次記憶

同じく、Moore74b での説明を記す。

FORTH の最後の重要な要素は、ブロック-1024 バイトの二次記憶装置です。2 つのコアバッファが提供され、ブロックは必要に応じて読み込まれます。ブロックがコアで変更されると、そのバッファが再利用されるときに、ディスク上のイメージが自動的に置き換えられます。プログラマは、自分のデータがいつでもコアにあると考えることができるため、非常に小さなコストで、非常に大きなサービスを提供することができます。

ブロックは、語彙を定義するテキストを格納するために使用されます。このブロックは、ユーザが要求したときにコアにコンパイルされる。編集がキャブラリは、ブロックを 64 文字で 16 行にフォーマットします。これにより、ユーザは自分のソースコードを修正したり、再コンパイルしたりすることができる。

また、ブロックはデータの保存にも使われる。小さなレコードを 1 つのブロックにまとめたり、大きなレコードを複数のブロックに分散して格納したりすることが容易にできる。ブロックはコンピュータに関係なく同じ形式なので、アプリケーションをあるコンピュータから別のコンピュータに移すのも簡単です。

- 1 ブロック 1024 バイトであること。

- バッファ 2 つをキャッシュとして使う。ライトバック。
- テキストバッファとして使うときは 64 文字×16 行、固定長行数。
- テキストバッファ上のテキストを編集するコマンド群を持つ。

Starting Forth には、ディスクアクセスと編集コマンドの説明がある。さすがにファイルシステムがない裸のディスクアクセスと、ラインエディタでプログラムを書く気にはなれないので、この 2 つについてはこれ以上追いかけない。

## Forth 処理系を簡単に説明する

Forth 処理系全体は、一言で言うと、ターミナル/ファイルから入力されるテキスト列を解釈し、実行するものといえる。解釈と実行は以下のような単純なループである。

- 次の空白で区切られた文字列を切り出し、
- それが数値であれば、データスタックトップにその数値を置き、
- それが数値でなければ、その文字列に対応する実行コードを探し出して呼び出す。
- 終われば次の文字列の切り出しを始める  
(次の文字列が無ければ実行を停止する)

Forth には制御構造が当然存在するが、それらは、制御構造を構成するワードの実行により実現される。IF が出てくると、IF 文の最後までソースコードを舐めて切り出すのではなく、IF の場所で行うべき処理(スタックトップの「判定式」の値により条件ジャンプ)を行うだけである。変数を定義する単語(**VARIABLE**)の実行では、「次の文字列を切り出し辞書エントリを作り、その実行

- **空白で区切られた文字列** : Forth で文法らしいところがあるとすれば、この「空白で区切られた文字列」である。空白かそれ以外かだけなので、記号も数字もわけへだてなく文字列として扱われる。
- **数値であれば** : 先頭が数字から始まる文字列は数値とみなされ、数値に変換される。通常は整数に変換される。正の数負の数も問題ない。普通の Forth では浮動小数点数は使用できない。
- **データスタックトップ** : Forth では演算対象の数値(オペランド)、サブルーチンの引数は全て Forth 処理系が管理するスタックの上にプッシュされる。このスタックをデータスタックという。例えば加算サブルーチンは、スタックトップの数値とその次の数値の和を計算し、スタック上の数値 2 つを消し、和の数値をスタックトップに置くのである。Forth がスタックマシンといわれるゆえんである。
- **文字列に対応する実行コードを探し出して** : この実行コードが「サブルーチン」になる。数値以外の文字列は**ワード**とみなされ、ワードに対応する実行コードのペアを保持するデータベース上で検索される。このデータベースが**辞書(Dictionary)**である。辞書中にそのワードのエントリが存在すれば、その中にあるサブルーチン呼び出す。  
サブルーチンには 2 種類ある。一つは、機械語で構成されるもの、もう一つは、辞書のエントリのアドレスの列を実行するものである。後者が Forth 処理系のカナメの一つとなる。

単純な例だと、コード

```
1 2 + .
```

である。これは、

- 整数 1 をスタックトップに置き
- 整数 2 をその上に置き、
- それら 2 つの和を取りスタックトップに置いて  
(+に対応するサブルーチンを辞書から探し出して実行する)
- スタックトップの値(3)を印字する  
(.に対応するサブルーチンを辞書から探し出して実行する)

という一連の処理を実行する。

ワード+は「和を計算する」サブルーチンであり、ワード. は、「スタックトップの値を印字する」サブルーチンである。

辞書の中に、ワード+のエントリとワード.のエントリがあり、それぞれ上記のサブルーチンの機械語列を格納している。

## ワードを定義する

Forth 言語でワードを定義することができる。定義したワードは辞書に追加され、以後にワードとして実行することができる。

ワードを定義するときは、実行したいワード列を:(コロン)と;(セミコロン)で囲む。例えば、

```
: +1 1 + ;
```

である。何のことやらという感じであるが、これは「スタックトップの値を 1 増やす」ワード+1 を定義しているのである。

定義されたワード、ここではワード+1 のエントリの中には、実行コードとして、

- 数値 1 をスタックトップに置くワードエントリのアドレス
- ワードエントリ+のアドレス

の 2 アドレスを持つ。このように、ワードを定義すると**ワードのアドレスの列**を含むエントリが生成され、辞書に追加される。ワード+1 が実行されると、その中のアドレス列を手繰りながらワードを順次実行してゆく。

## スタックを介したオペランド渡し

ワードがサブルーチンにより実行される、その引数はスタック上に置かれる。数値演算の単項演算子はスタックトップの値 1 つをオペランド(処理対象)とする、二項演算子はスタックトップとその下の値の 2 つをオペランドとする。

通常の数式



```
2 + (3 * 4)
```

は、スタック操作を用いて記述すると、

```
2 3 4 * +
```

となり、演算子の結合関係を記述するための括弧が不要になる。これにかぎらず、任意の数式を括弧なしで記述することができ、スタック操作に基づく処理で任意の数式を表現できる。

Fortran はじめ他の言語が数式を数式のまま記述して与え、言語処理系に解釈させることと異なり、Forth は数式のまま与えることは選ばず、プログラマに数式を逆ポーランド記法に変換させ、実行エンジンをスタックオペレーションだけの単純化させることを選んでいる。処理系を単純にすることを志向しているのではないか、そんな気がする。

## 内部インタプリタ: ワードアドレス列の実行

先ほど、ワードの定義により、ワードアドレスの列を含むエントリができることを説明した。あるワードを打ち込むと、そのワードが実行される。この処理は先に説明したとおりだが、その中で、文字列に対するコードを探し出して実行する、という記述があった、文字列をワードとして辞書を検索して見つけたエントリの中にアドレス列が格納されている場合、それはどのようにして実行されるのか？

アドレス列の実行のために、インストラクションポインタ(IP)というアドレスレジスタを用意し、そこにアドレス列の先頭のアドレスを格納する。すると

1. IP が指す先のメモリに格納された値を取り出し、
2. それがワードの実行ルーチンを指すとしてサブルーチンコールする。
3. サブルーチンの処理が終わると IP をインクリメントする。

の処理をアドレス列の末尾に至るまで繰り返すだけである。このままでは末尾で零れ落ちてしまうので、アドレス列の最後のアドレスが指す先は、「アドレス列の実行を完了し、そのワードを呼び出したもと(それも別のワードのアドレス列の途中である)に戻り、次のアドレスを実行する」となる。

ワードの呼び出し元に戻るために、IP の戻り番地を保持するスタックを用意する。これをリターンスタックという。

## 仮想 Forth 実行マシン

Forth 処理系は、

- ソースコードを読み込み単語に区切り辞書を引き、見つけたワードを実行する。
- ワードの実行において、ワード中のコード(機械語コードまたはワードアドレスの列)を実行する。

の2つの処理を行う。前者を外側インタプリタ(outer interpreter)、後者を内側インタプリタ(inner interpreter)と呼ぶ。内側インタプリタは、「データスタック、リターンスタック、インストラクションポインタ」を用いて処理する。

内側インタプリタを実行する仮想的なCPUを考えることができる。

- データスタックポインタ
- リターンスタックポインタ
- インストラクションポインタ

## 大体の旅程

- 端末からの入力、出力
- 行入力
- ワードの定義、辞書を作る
- 低レベルインタプリタ
- 高レベルインタプリタ
- 定義語 `:`, `:`
- 定義語 `VARIABLE`, `CONSTANT`
- `IF ... ELSE ... THEN`
- `DO ... WHILE`
- 算術演算(整数加減乗除)

Moore70 で規定される機械語ワードをそろえたところで旅の一区切りとしようか。

## 4. 実行環境の整備

---

70 年当時のミニコン環境を整えます。機械語/アセンブラでの開発がメインで、私個人として経験のない 68000 を選びました。

今風の開発環境として、SBC(Single Board Computer)とエミュレータソフトウェアを用意しました。SBC は、68008 CPU, 128k SRAM, PIC18F CPU からなる基板です。エミュレータは [Musashi](#) をベースにしました。68000 ファミリーを FPU/MMU まで備えたなかなかのエミュレータだと思います。

### 70 年代の実行環境・開発環境

Moore がたどった道をたどるために、70 年当時のミニコン環境に近づけます。CPU とメモリとスイッチパネル、数 MByte のハードディスクと、タイプライタ端末、紙テープリーダー・パンチャです。

ミニコン自体は 19 インチラック 1 本に CPU/メモリとフロントパネルが刺さっています。磁気テープ装置やハードディスクも含めてラック数本でコンピュータ 1 台を構成していたようです。オペレータの一人は、**タイプライタ端末**の前に座ってコンピュータを操作するふりをしています。

図 2-1. [PDP-11 ミニコンピュータ全景](#)(PDP-11 の前でポーズを取る Dennis Ritchie と Ken Thompson)



**フロントパネル:** コンピュータを操作するためのスイッチが並んでいます。CPU を止めておくスイッチ、CPU 停止中に RAM 上に 1 バイト 1 バイトデータを書き込んでゆくためのスイッチがならんでいます。

図 2-2. [ミニコン PDP-11 のフロントパネル](#)(実際には、レプリカ PiDP-11 のパネルです)



当時のミニコンピュータは、電源投入直後にメモリ上にはなにもなく、フロントパネルのスイッチを操作して、0番地から1バイトずつIPLを書き込み、それを実行して実行したいプログラムを紙テープから読み込むというのが使い始めの手続きとなっていました。

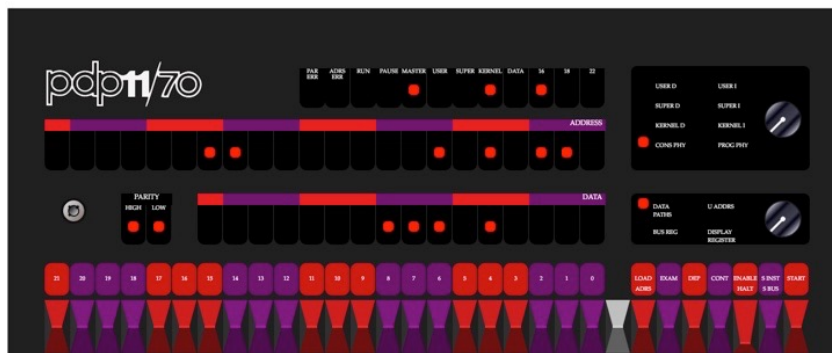
具体的にどんな感じで操作するのかは、以下のYouTubeムービーがよい感じを出しています。

[Programming a PDP-11 Through the Front Panel by BitDanz Blog](#)

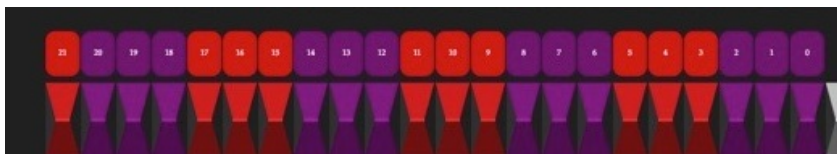
ムービーを今すぐ見られない方々のために、いかにざっと説明を書いておきます。

- 機械語(8進数データ)をプリントアウトしておく。
- 先頭アドレス(01000)を下側のスイッチをON/OFFして2進数で設定する。  
下側のスイッチ(スイッチレジスタ)は数値(アドレス、データ)を入力するためのもので、3個一組で色分けされており、8進数3ビットままとりのダンプデータの入力が分かりやすくなっています。
- LOAD ADRSスイッチを引き、スイッチの値をアドレスレジスタに設定する。
- スイッチレジスタのスイッチを操作し、データを設定する。
- DEPスイッチを引き、先ほど設定したデータの値をメモリに書き込む。
- (アドレスレジスタの値が自動的にインクリメントされるので)再度スイッチレジスタに次のデータを設定し、DEPスイッチを引くと、2番目のデータが格納される。

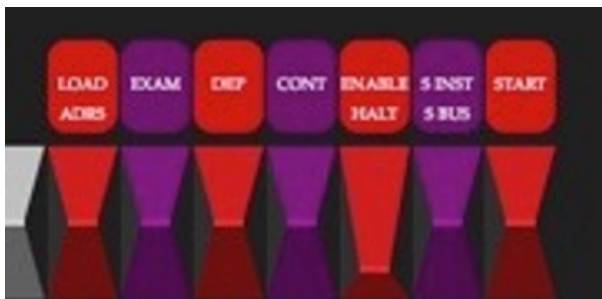
こうして、1ワード1ワードメモリに打ち込んでゆくのです。



左下のスイッチレジスタ(18個)で8進数を入力します。



右下のコントロールスイッチを使い、アドレス設定、データ書き込みを行います。



各スイッチの説明は以下の通りです。

スイッチ名	説明
LOAD ADRS	スイッチレジスタの内容をバスアドレスレジスタに書き込む。書き込んだ結果がアドレスレジスタ(18bit LED)に表示される。この値を使い、EXAM, DEP, START スイッチを押しそれぞれの動作を行う。
EXAM (Examine)	バスアドレスがさすメモリの内容をデータレジスタ(16bit LED)に表示する。再度このスイッチを引くと、次のアドレスの内容を表示する。つまり、バスアドレスが自動的にインクリメントされる。
DEP (Deposit)	スイッチレジスタの内容をメモリに書き込む。書き込む位置はバスアドレスである。
CONT (Continue)	先ほど停止した続きで実行再開する。
ENABLE/ HALT	ENABLE: CPU にプログラム実行させる。 HALT: CPU を停止する。スイッチを引くとシングルステップ実行する。
START	システムリセット後、プログラムを実行する。

これで数キロバイト打ち込むのは耐えられないので、「紙テープからプログラムをロードするプログラム」「ハードディスク先頭からプログラムをメモリにロードするプログラム」(IPL)だけを打ち込み、Fortran コンパイラやアセンブラをロードして実行させていたと思われます。

また、ブートローダを ROM で持たせることもできたようで、80 年代の PDP-11 のマニュアルを見ると、テンキー(実際には 8 個だけだが)でアドレス・データが入力できるコンソールと、それもなく実行開始・停止だけができるコンソールも出てきています。後者はブートローダを ROM で持つことが前提になっていることがわかります。

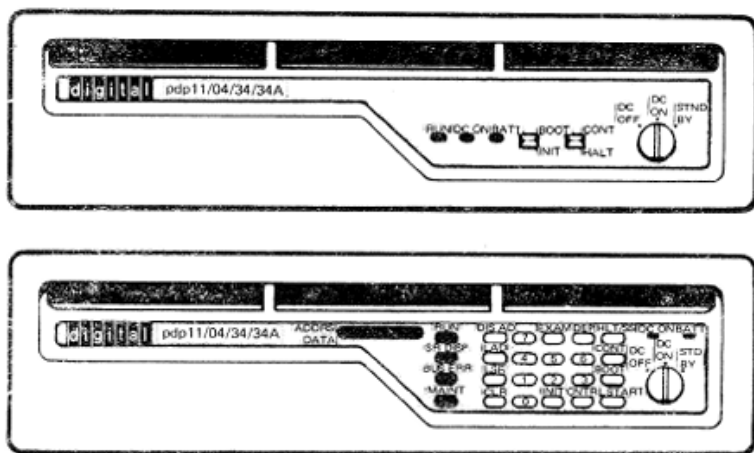


図 2-5. 80 年代の PDP-11 のフロントパネル(上が通常型パネル、下がプログラマー用パネル)

## 今回の実行環境

これだけ盛り上げておいて恐縮ですが、今回の実行環境ではスイッチパチパチのパネルは用意していません。大昔に自作コンピュータを操作したときの経験を思い出すと、数十バイトの IPL ロードでさえ「やってられねえ」感じからです。申し訳ありません。

SBC 版では、起動直後にブートロードモードに入り、シリアルターミナル(Teraterm 等)のテキストアップロード機能を用いてダンプデータファイルをアップロードします。

SBC 内部の PIC のファームウェアがダンプデータファイルをメモリ上に展開し、スタートコマンドを入力するとプログラムを実行します。

エミュレータ版は、Linux コマンドラインプロンプトでダンプデータファイルを引数として与えると、エミュレータ起動時に読み取りメモリ上に展開し、プログラムの実行を開始します。

アセンブル・リンクはいずれにおいても Linux プロンプトから、Linux コマンドのアセンブラ・リンクを起動します。70 年代の環境では、

- ブートローダを手で打ち込む。
- 紙テープのアセンブラプログラムを読み込み実行する。
- アセンブラソースコードを紙テープから読み込み、メモリ上に展開する。
- ターゲットプログラムを実行する。

と思われます。かなり面倒なことが想像できます。そういう観点からは今回の「Linux コマンドでアセンブルする」はかなりのズルかも知れません。

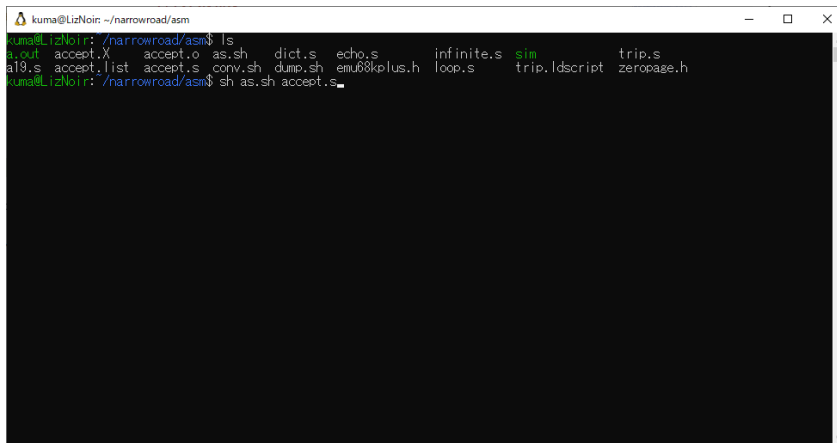
一方、実行開始後は打つ手はありません。デバッグなしでのデバッグとなります。ブレークポイントとシングルステップまで許容してもいいかもしれません。



## 具体的な手順

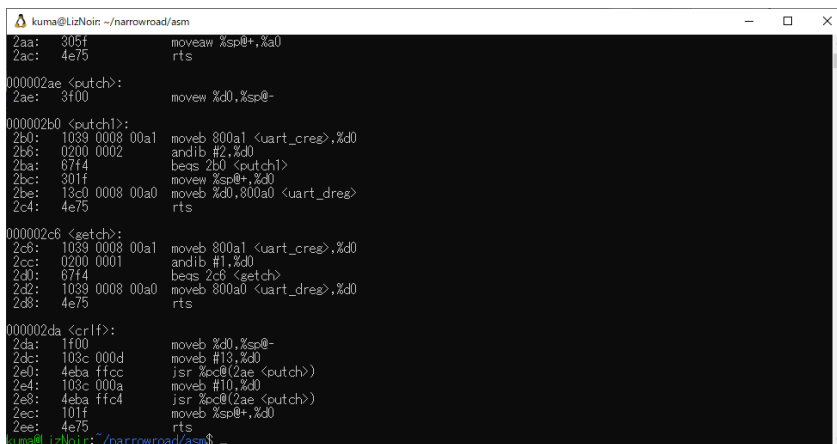
### 1. ソースコードをアセンブルする。

`sh as.sh [source-file]`を実行します。以下ではソースコード `accept.s` をアセンブルしています。



```
kuma@LizNoir: ~/narrowroad/asm
kuma@LizNoir: ~/narrowroad/asm$ ls
a.out  accept.X  accept.o  as.sh  dict.s  echo.s  infinite.s  sim  trip.s
a19.s  accept.list  accept.s  conv.sh  dump.sh  emu68kplus.h  loop.s  trip.ldscript  zeropage.h
kuma@LizNoir: ~/narrowroad/asm$ sh as.sh accept.s
```

アセンブルが完了すると画面にリストファイルが出力されます。



```
kuma@LizNoir: ~/narrowroad/asm
2aa: 305f      moveaw %sp@+, %a0
2ac: 4e75      rts
000002ae <putch>:
2ae: 3f00      movew %d0, %sp@-
000002b0 <putch>:
2b0: 1039 0008 00a1  moveb 800a1 <uart_creg>, %d0
2b6: 0200 0002  andib #2, %d0
2ba: 67f4      beq 2b0 <putch>
2bc: 301f      movew %sp@+, %d0
2be: 13c0 0008 00a0  moveb %d0, 800a0 <uart_dreg>
2c4: 4e75      rts
000002c6 <getch>:
2c6: 1039 0008 00a1  moveb 800a1 <uart_creg>, %d0
2cc: 0200 0001  andib #1, %d0
2d0: 67f4      beq 2c6 <getch>
2d2: 1039 0008 00a0  moveb 800a0 <uart_dreg>, %d0
2d8: 4e75      rts
000002da <crLf>:
2da: 1f00      moveb %d0, %sp@-
2dc: 103c 000d  moveb #10, %d0
2e0: 4eba ffc0  jsr %pc@(2ae <putch>)
2e4: 103c 000a  moveb #10, %d0
2e8: 4eba ffc4  jsr %pc@(2ae <putch>)
2ec: 101f      moveb %sp@+, %d0
2ee: 4e75      rts
kuma@LizNoir: ~/narrowroad/asm$
```

### 2. ダンプファイルに変換する。

`sh dump.sh > [dump-file]`を実行します。以下ではダンプファイル `accept.X` を生成しています。

```
kuma@LizNoir: ~/narrowroad/asm
2aa: 305f      moveaw %sp@+,%a0
2ac: 4e75      rts

000002ae <putch>:
2ae: 3f00      movew %d0,%sp@-

000002b0 <putch1>:
2b0: 1039 0008 00a1  moveb 800a1 <uart_creg>,%d0
2b6: 0200 0002      andib #2,%d0
2ba: 67f4      beqsb 2b0 <putch1>
2bc: 301f      movew %sp@+,%d0
2be: 13c0 0008 00a0  moveb %d0,800a0 <uart_dreg>
2c4: 4e75      rts

000002c6 <getch>:
2c6: 1039 0008 00a1  moveb 800a1 <uart_creg>,%d0
2cc: 0200 0001      andib #1,%d0
2d0: 67f4      beqsb 2c6 <getch>
2d2: 1039 0008 00a0  moveb 800a0 <uart_dreg>,%d0
2d8: 4e75      rts

000002da <crLf>:
2da: 1f00      moveb %d0,%sp@-
2dc: 103c 000d      moveb %13,%d0
2e0: 4eba ffc0      jsr %pc@(<2ae <putch>)>
2e4: 103c 000a      moveb %10,%d0
2e8: 4eba ffc4      jsr %pc@(<2ae <putch>)>
2ec: 101f      moveb %sp@+,%d0
2ee: 4e75      rts
kuma@LizNoir:~/narrowroad/asm$ sh dump.sh > accept.X
```

## アセンブラとリンカ

68000 用として、binutils を 68000 用にビルドして使っています。

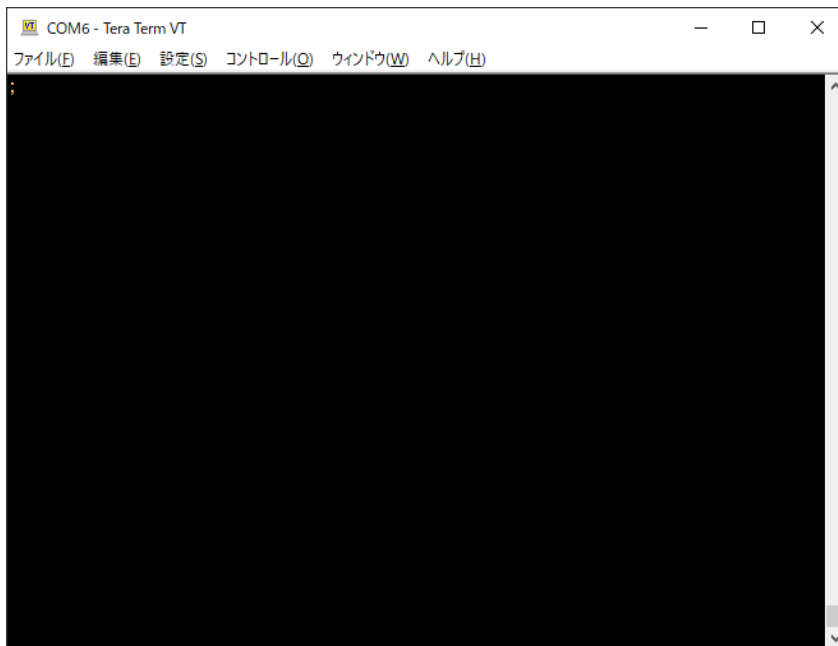
スクリプト `as.sh` では、アセンブラ `m68k-elf-as` を実行後、リンカ `m68k-elf-ld` で未解決シンボルの解決を行っています。

スクリプト `dump.sh` では `m68k-elf-objdump` コマンドを使ってダンプファイルを生成後、`sed` により PIC ファームウェアがロード可能な形式に変換しています。

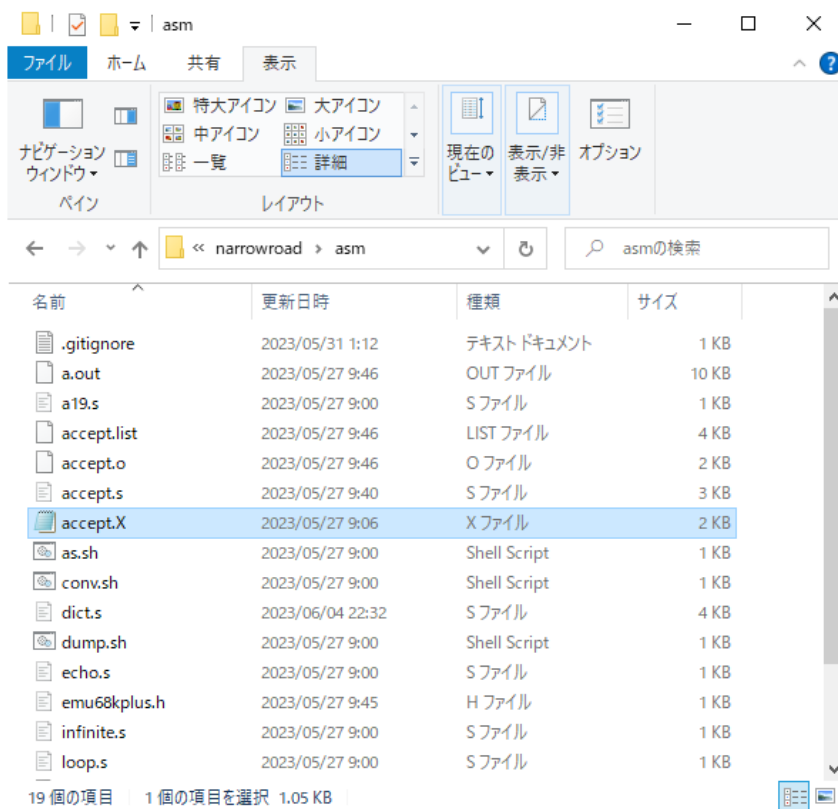
```
kuma@LizNoir: /narrowroad/asm$ cat as.sh
#!/bin/sh
o=`basename -s .s "$1"`
m68k-elf-as -o ${b}.o "$1" &&
m68k-elf-ld -T trip.ldscript ${b}.o &&
( m68k-elf-objdump -D a.out | tee ${b}.list )
kuma@LizNoir: /narrowroad/asm$ cat dump.sh
#!/bin/sh
m68k-elf-objdump -D a.out | sed -n '
/^ *([0-9A-Fa-f][0-9A-Fa-f]*)*:/[
s/^([ ]*) *([ ]*) *$/$1 $2/
s/^ *([0-9A-Fa-f][0-9A-Fa-f]*)*: */=$1 /
s/ *$//
s/ */ /g
p
```

## 3. 実行(SBC 版): ダンプファイルをドラッグアンドドロップする

SBC(emu68k\_plus)を起動する。Teraterm に ; だけのプロンプトが表示される。



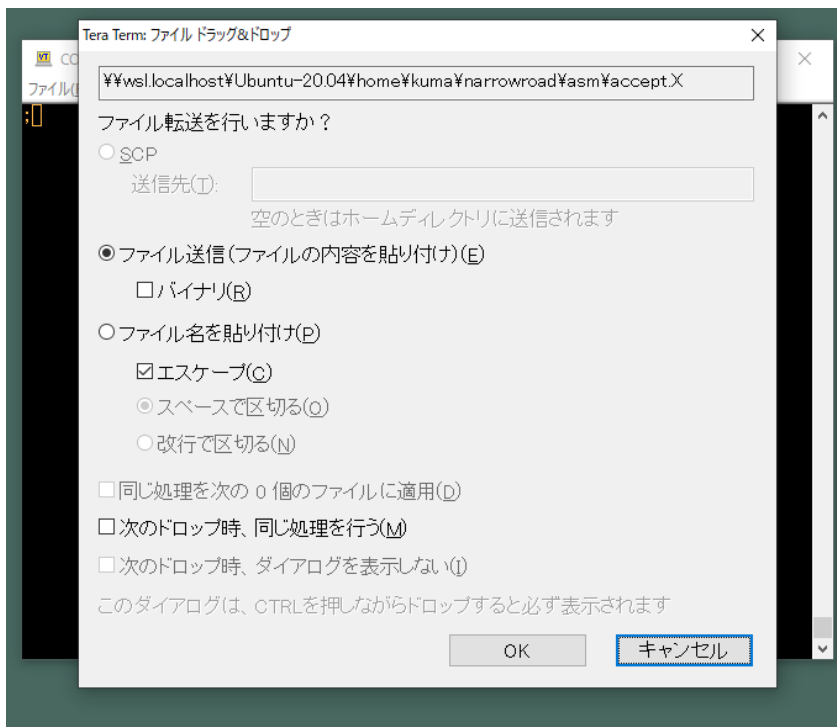
この状態で、Windows のエクスプローラからダンプファイル **accept.X** を掴んで、Teraterm にドラッグアンドドロップする。



WSL のファイルを Windows エクスプローラで開くには、`\\wsl$\\Ubuntu-20.04\\home\\kuma\\...\\accept.X`のように、`\\wsl$\\Ubuntu-20.04\\`を先頭に付けてエクスプローラを開けるとよいようである。

すると、以下のダイアログを出してくるので、OKを押す。アップロードが始まる。

8k BASIC でも数秒で終わる。さすがに 115200bps は速い。



アップロードが終わっても Teraterm 画面には何も変化がない。アップロード中は読み込みと RAM への展開に全力を注ぎ、エコーバックは一切行わないのだ。エコーバックを行わないことにより、PIC ファームウェアは 115200bps で突っ込まれても十分処理できる。

以下の画面はダンプコマンドを叩いたあとの状態です。ダンプコマンドは 1 文字 `!` (びっくりマーク) です。

```
COM6 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) ヘルプ(H)
:
0000 0000 F000 0000 0200 5441 4575 1545 5445
0010 7361 6D70 6C65 2073 7472 696E 6701 0203
0020 0000 5544 4D55 4545 1455 5455 5155 4501
0030 4551 4555 4555 4545 4451 04C5 4555 55C5
0040 5551 5444 4154 4045 4545 5544 0545 5545
0050 C555 5D51 4551 1545 4545 5545 5557 0551
0060 4141 4545 5C54 4545 4414 5504 55D5 5455
0070 5145 4465 6045 5515 4540 5455 D445 4144
0080 5555 7504 4456 5455 5544 5555 44D5 5554
0090 545D 5555 D444 5D55 5556 4541 4445 5554
00A0 45FF D544 5555 5544 5555 545D 5545 5554
00B0 5545 4155 4445 45C4 5044 0545 4545 4540
00C0 5554 5D45 5545 5455 5454 4504 5555 4754
00D0 5545 44D4 44D5 4544 5544 D055 5444 C444
00E0 5404 4555 144D 1445 4445 5445 3545 4445
00F0 5054 5555 5554 5545 5544 5755 4544 5554
0100 0455 5554 1455 45C4 5555 5555 5455 4545
0110 4544 4D74 4554 5655 5545 5144 4455 5455
0120 4545 5544 41C4 5444 1575 7545 C055 4555
0130 5554 5554 5545 4554 5445 4541 4505 5D50
0140 144D 5445 5515 0455 5445 4445 5551 554C
0150 5154 4344 5555 5545 4151 4445 5154 1555
0160 5444 4145 4575 5405 0554 5445 4515 4545
0170 4454 455D 5455 4504 5444 144D 4404 5455
```

プログラム実行開始は...(ピリオド3個連打)です。`start ss = 0, bp = 0000`と表示され68008のリセットが解除されます。上記の表示は見た目だけでブレークポイントが使えるわけではありません。

```
COM6 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) ヘルプ(H)
0180 4565 4545 5155 4556 4554 5514 4454 5044
0190 5445 5754 55E5 5455 5544 D445 5654 5446
01A0 6551 5544 44D5 4555 5444 5441 4555 5554
01B0 4455 5455 5555 4444 0445 5544 4544 D555
01C0 5555 7445 5455 5145 5454 5544 5544 5455
01D0 4441 5555 4545 D515 C545 7575 4447 5755
01E0 C745 4545 5445 5545 5444 5554 7545 4405
01F0 5555 4554 5545 4545 4554 5555 5454 5554
0200 207C 0000 0F00 7240 4EBA 0010 4EBA 00CC
0210 4EBA 007C 4EBA 00C4 60E6 3F09 3F02 3248
0220 3401 323C 0000 B441 6F00 005A 4EBA 0098
0230 0C00 00D0 6700 004E 0C00 000A 6700 0046
0240 0C00 0008 6700 001A 0C00 007F 6700 0012
0250 3F00 4EBA 005A 301F 1380 1800 5241 60C6
0260 0C41 0000 6F00 FFC0 5341 103C 0008 4EBA
0270 003E 103C 0020 4EBA 0036 103C 0008 4EBA
0280 002E 60A2 3001 341F 325F 4E75 60FE 3F08
0290 3F01 3F00 3200 0641 FFFF 6D00 000A 1018
02A0 4EBA 000C 60F0 301F 321F 305F 4E75 3F00
02B0 1039 0008 00A1 0200 0002 67F4 301F 13C0
02C0 0008 00A0 4E75 1039 0008 00A1 0200 0001
02D0 67F4 1039 0008 00A0 4E75 1F00 103C 000D
02E0 4EBA FFCC 103C 000A 4EBA FFC4 101F 4E75
start ss = 0, bp = 0000
```

今回の例で用いた `accept.X` は入行ルーチンです。通常文字を受信・エコーバックするとともにバッファに貯めます。`^H` により直前の文字を抹消することができます。行編集コマンドはこれだけです。リターンを押すと入力完了です。このテストプログラムでは入力して得た文字列をそのままダンプしています。よって以下のように同じ表示が2行続くことになります。

```
COM6 - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(C) ウィンドウ(W) ヘルプ(H)
01A0 6551 5544 44D5 4555 5444 5441 4555 5554
01B0 4455 5455 5555 4444 0445 5544 4544 D555
01C0 5555 7445 5455 5145 5454 5544 5544 5455
01D0 4441 5555 4545 D515 C545 7575 4447 5755
01E0 C745 4545 5445 5545 5444 5554 7545 4405
01F0 5555 4554 5545 4545 4554 5555 5454 5554
0200 207C 0000 0F00 7240 4EBA 0010 4EBA 00CC
0210 4EBA 007C 4EBA 00C4 60E6 3F09 3F02 3248
0220 3401 323C 0000 B441 6F00 005A 4EBA 0098
0230 0C00 000D 6700 004E 0C00 000A 6700 0046
0240 0C00 0008 6700 001A 0C00 007F 6700 0012
0250 3F00 4EBA 005A 301F 1380 1800 5241 60C6
0260 0C41 0000 6F00 FFC0 5341 103C 0008 4EBA
0270 003E 103C 0020 4EBA 0036 103C 0008 4EBA
0280 002E 60A2 3001 341F 325F 4E75 60FE 3F08
0290 3F01 3F00 3200 0641 FFFF 6D00 000A 1018
02A0 4EBA 000C 60F0 301F 321F 305F 4E75 3F00
02B0 1039 0008 00A1 0200 0002 67F4 301F 13C0
02C0 0008 00A0 4E75 1039 0008 00A1 0200 0001
02D0 67F4 1039 0008 00A0 4E75 1F00 103C 000D
02E0 4EBA FFCC 103C 000A 4EBA FFC4 101F 4E75
start ss = 0, bp = 0000
line input
line input
```



## 5. 旅立ち

[PROGRAMMING A PROBLEM ORIENTED LANGUAGE, Moore, 70b](#) は、Moore 師匠が 1970 年に書いた文書である。自前の環境を 10 年使い続けて、開発環境とアプリケーション構築に関する彼の考えを説明している。この文書は FORTH という開発環境そのものの説明だが、この 9 章は、新しいマシンを前にしてこの環境をどのようにして構築してゆくか、のあらすじをざっと説明している。

移植ではない、構築なのだ。別のアーキテクチャの CPU をまたいで生きてゆかねばならないのだから、共通のディスクアクセス機構、共通の OS、共通のライブラリを前提として、元のソースコードにちょいちょいと手直しだけして「移植」することは期待できない。C コンパイラなんてもちろん存在しない。下手するとアセンブラさえ使えない環境、機械語をばちばち打ち込んで実行するだけの環境に、自分の手元にある環境一式を「構築」してゆく。これまで頼りにしていたものすべてから切り離され、生きてゆかねばならない、そんな気持ちがするかも。「月は地獄だ」「Dr. STONE」「サバイバル」そんな作品を思い出しながら、彼のたどった道を自分の足で歩いてみたい。そんな気がするのだ。

以下では、Moore, 70b の 9 章の文言を引用しながら、私が歩いた道筋を説明してゆく。よろしければ、ご自分で好きな CPU を選んで実行環境をつくっていただいても構わない。

### まず練習だ

さて、あなたは今、コンピュータと向き合っている。どうするんだ？ まず練習です。コンピュータを起動したら、無限ループを実行するように割り込み位置を初期化しなさい。いいかい？ それからループを修正してメモリをクリアするようにします。いいですか？ おそらく多くのことを学んだことでしょう。(9.1 章)

BIOS はおろか、ブートローダもなにもない。1 バイト目からフロントパネルのスイッチをばちばちして入力して RUN スイッチを押すしかない、そんなコンピュータを前にして、最初にやることは、コンピュータに慣れること。

Moore 師匠当時のコンピュータはフロントパネルにアドレスバス・データバス状態を示す LED があり、その点滅やシングルステップで実行を見ることができたのでした。

実は、私も 1981 年に自分で「スイッチばちばち」のコンピュータをくみ上げて使い始めていました。が、これが結構つらいのです。結局 PC8801 を買ってからはそちらにひよってしまっていたのでした。なので、今回もスイッチばちばちから始めるというのはやめておきます。代わりにシングルステップを使うことにします。

EMU68kplus には割り込み機能はありません(今のところは)。よって、単純な無限ループを組みます。

```
.org      0
loop:
```

```
bra      loop
/* end */
```

アップロード用ファイルは、

```
=0 6000 fffe
```

## 辞書(ディクショナリ)

さて、これからが本番です。まだ使うことはできないにしても、辞書の構築を開始します。ここでエントリの形式を選びます。可変長エントリは必須ですが、それでも、ワードサイズとレイアウトはあなたが決めることができます。

既にあるライブラリを頼まず、一から手組みで「オレのシステム」をくみ上げてゆくこの取り組みで、データ・ルーチンに名前を付け呼び出せるようにする仕組みの基本が「辞書(ディクショナリ)」です。ロジカルには、「名前とバイト列のペアをエントリとする」「定義した順にエントリをリストにつなぎ、検索は最新のエントリから最初のエントリにさかのぼってゆく」がその定義で、登録・検索も単純なルーチンで実現できます。

エントリには、定義されるワード、実行されるコード、次のエントリへのリンク、パラメータの4つのフィールドがあります。それぞれについて説明する。

ワードの形式はワード入力ルーチンとともに決定されなければならない。ワードのサイズは固定で、NEXTで定義されたサイズより小さくてもよいが、ハードウェアのワードサイズの倍数でなければならない。しかし、より洗練されたアプリケーションは、出力メッセージを構築するために辞書のワードを使用する。その場合、ワードを切り詰めないことが重要であり、その場合、ワードフィールドは可変長でなければならない。このフィールドのサイズをマークするために、文字カウントではなく、空白文字を使用する必要がある。可変エントリ内で可変ワードフィールドを扱うには、ワードは一方方向に(後方に)、パラメータは他方向に(前方に)伸びる必要があります。固定または可変ワードサイズのどちらを選ぶかは、基本原則の適用が必要です。

コードフィールドには、テーブルや他の省略形へのインデックスではなく、ルーチンのアドレスを入れるべきです。プログラムの効率は、3.9で説明するように、エントリが特定された後、コードにたどり着くまでの時間に強く依存します。しかし、プログラムのサイズが小さいと、このアドレスはハードウェアアドレスフィールドより少ないスペースに収めることができます。

リンクフィールドも同様に、ハードウェアで指定されたものより小さくてもよい。これは、現在のエントリからの距離ではなく、次のエントリの絶対位置を含むべきである。

パラメータフィールドは、通常4種類の情報を含む。

- 定数または変数で、サイズは可変です。数値の性質は、実行されるコードによって決定されます。

- 配列 - 数値が格納されるスペースです。配列のサイズはパラメータであるか、または実行されるコードに含まれているかもしれません。
- 定義：仮想のコンピュータ命令を表す辞書の項目の配列。3.9を参照してください。
- 機械語命令：プログラムによってコンパイルされたコードで、このエントリが実行される際にこのコードが実行される。このようなデータは、おそらくワード境界でアラインされなければならないが、他はその必要がない。(3.6.1. エントリの形式)

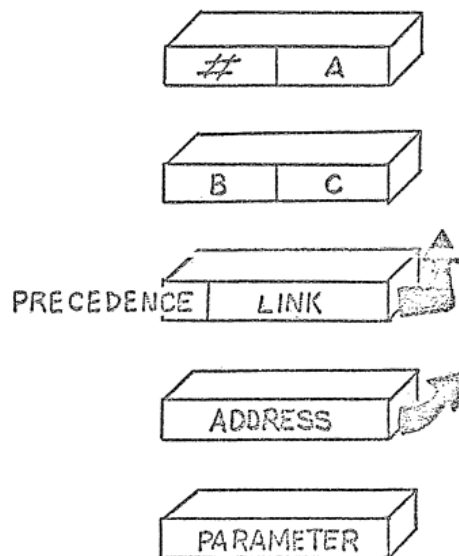


図 4. 辞書エントリの形式

例えば、単語"ABC"のエントリは以下のようになります。

```

1324: 03          ; number of name
1325: 41          ; 1st char of name
1326: 42          ; 2nd char of name
1327: 43          ; 3rd char of name
1328: 0340        ; PRECEDENCE and LINK
132A: 0551        ; machine code of the word, "ABC"
132B:

```

最初の1バイトは名前の文字列の長さです。Forthでは、文字列はヌル文字終端ではなくて、銭湯1バイトに長さを置き、2バイト目から名前を表す文字を順に並べます。

次の要素はPRECEDENCE and LINKです。この要素はワードアライメントされて置かれます。この例では2バイトアライメントです。

PRECEDENCEは、このワードの属性を表すビットです。通常、2ビットを占めます。LINKは次のワードの先頭を指すポインタです。この例では、1ワード(16bit)の上位2ビットをPRECEDENCEが占め、下位14ビットをアドレスを収容します。

LINKは、ワードの単方向リストを順にたどる際に用います。絶対アドレスを用いてもよいし、リンク位置からの相対オフセットでもかまいません。絶対アドレスを用いると、辞書サイズの上限が16kBになりますが、相対オフセットを用いると、1 エントリ 16kB 上限になります。あとで分かってくるのですが、1 エントリ 16kB も使うことは普通はないのと(特殊な巨大配列を使う場合ぐらい)、単方向リストをたどる処理はプログラムの実行時間に影響を与えない(「コンパイル」時間に効いてくる)ので、相対オフセットで1 エントリ手繰るたびにポインタ加算が余分に増えても問題ないでしょう。

コンパイラ(といっても入力文字列からワードを一つ一つ切り出し、辞書を引いてその名前を持つエントリを検索し、得たエントリのアドレスをメモリ上に並べてゆく)ができるまでは、「手で辞書をコンパイル」します。具体的には、「名前」「属性」「名前の列」を順に並べ、辞書エントリを描くアセンブラプログラムを書きます。

通常のやり方は、マクロアセンブラのマクロ機能を用いて辞書エントリを生成し、本体は「名前の列」のエントリシンボルを書き出して、アセンブラに掛けてオブジェクトを生成し、辞書とします。

ここでは、簡単なテキスト処理プログラムにより、辞書エントリを生成し、アセンブラに掛けるコードを生成させるようにします。名前の前方参照を解決する必要があるので、そこはアセンブラの機能を利用するというわけです。

エントリのデータフィールドには、他のエントリのアドレスが並びますが、アドレスでない要素も並べられるようにします。以下の要素です。

- **定数をスタックに置く要素(Literal):** 機械語のイミディエイト命令のように、次の要素を数値とみなしスタックに置きます。ソースコードで定数を書くと、辞書エントリとしてコンパイルするときにこの要素が必要になるのです。
- **ブランチ(スキップ)命令:** 直後の要素をアドレスとみなし、そこまでジャンプすることを指示する要素です。IF-ELSE-THEN, WHILE 等の制御構造を実現するときに、辞書エントリ内で要素をスキップしたり戻ったりすることが必要になるのですが、それを実現します。無条件ブランチと、条件付きブランチの2種類を用意します。条件付きブランチは、「スタックトップの値がゼロならジャンプする」の一つだけとします。大なり、小なりは、ブランチ命令の前に演算を並べてスタックトップにゼロ/非ゼロの結果を返すことで実現します。

ここまでくると、**call, bra, load immediate** の3種類の命令だけからなる特殊なCPUが実行する機械語の世界である。そのCPUは、スタックポインタ2つ、アキュムレータ1個を持ち、データスタックとリターンスタックを管理し、スタック上のデータとアキュムレータの間で演算するという風になる。文献によっては、Forth Virtual Machine と呼ぶものもある。

## SAVE, LOAD, DUMP

ここでワードの実行の話になるのかと思えば、Moore 師匠の話はいくつかのサービスルーチンをアセンブラで書く方向に向かいます。

辞書エントリの実行はどうなるねん。確かにサービスルーチンは必要だし、CPUに慣れるというなら、よく知ったサブルーチンを新しいCPUの機械語で書くというのもわかる。ということで、辞書の実行は後のお楽しみということで取っておきます。

最初のエントリはSAVEで、これはプログラムをディスクに保存します。コントローループがないので、手動でジャンプしなければなりませんが、少なくとも、多くの作業をやり直すことは最小限に抑えられます。2番目の項目はLOADで、ディスクからプログラム

を再ロードします。ハードウェアのロードボタンがあるかもしれませんが、それと互換性を持ってプログラムを保存できるのであれば、それはそれで結構です。そうでなければ、ロードカードにパンチして初期ロードを提供するのがよいでしょう。しかし、コアから再スタートできるのは常に便利なことです。

SAVE, LOAD は、ハードディスクとメモリの間でデータの読み書きを行うサブルーチンです。

いやあんた、EMU68kplus には外部ストレージがないやんか。SAVE も LOAD もないやろう。ということで少し困りました。

ここで想定している操作は、

- 以前作ったプログラムを RAM 上に LOAD する。
- スイッチぱちぱちで追加入力、ルーチン呼び出し、シングルステップやレジスタウォッチを使いデバッグする。
- 作業が一区切りしたら RAM イメージをディスクに SAVE する。

だと思います。ここでは、

- 以前保存しておいたバイナリイメージを母艦 PC から RAM 上にアップロードする。
- ルーチン呼び出し、シングルステップを使いデバッグ。
- 但しプログラム修正はアセンブラソースコードを修正して再アセンブルする。
- 修正プログラムのデバッグは、再アセンブル結果を RAM 上にアップロードする。

とします。EMU68kplus 上でプログラム変更しないので、モニタの機能だけで作業を進めることができます。

外部ストレージがまだないことと、バイナリを手で修正してデバッグというのも辛いので、ひよってしまいました。すみません。

3 番目のエントリは DUMP で、これはコアをプリンタにダンプします。スイッチで見るとよりずっと速いので、それほど速くなくてもよいでしょう。このルーチンはおそらく自明なものではありませんが、12 命令以上かかることはないはずです。ほんの少し延期してもいいかもしれません。

せめてこれぐらいはちゃんとやろうと想い、DUMP ルーチンを作りました。

```
/*
 * dodump
 * hex dump a region of RAM storage
 * %a0: begin address
 * %d0: count
 */
dodump:
    move.w    %a1, -(%a7)        /* push %a1 */
    move.w    %d1, -(%a7)        /* push %d1 */
    move.w    %d0, %d1           /* %d1: loop counter */
    move.w    %a0, %a1           /* %a1: address pointer */
    move.w    %a0, %d0
```

```

and.w  #0xfffe,%d0      /* address should be even */
move.w  %d0,%a1
and.w  #0xffff0,%d0      /* %d0: actual start address */
/* type initial address */
move.w  %d0,(dbg_port+2)
jsr     (puthex4)
move.b  #' : ',%d0
jsr     (putch)
jsr     (bl)              /* type a blank */
/* check skip words */
dodump1:
/* prefix five spaces */
move.b  %d0,(dbg_port)
move.w  %a1,%d0
and.w  #0xf,%d0          /* %d1 = %a1 & 0xf, skip count */
dodump2:
beq.b   dodump3          /* if zero, end of five spaces */
/* five spaces */
jsr     (bl)
jsr     (bl)
jsr     (bl)
jsr     (bl)
jsr     (bl)
sub.w   #2,%d0
bge.b   dodump2
dodump3:
/* check loop counter */
and.w   %d1,%d1          /* check loop counter */
beq.b   dodump4
/* word dump loop */
move.w  %a1,%d0
and.w   #0xf,%d0
bne.b   dodump5          /* skip typing address */
/* type address */
move.w  %a1,%d0
jsr     (puthex4)
move.b  #' : ',%d0
jsr     (putch)
jsr     (bl)
dodump5:
/* put word */
move.w  (%a1),%d0
jsr     (puthex4)
jsr     (bl)
add.w   #2,%a1

```



```

    /* check eol */
    move.w  %a1,%d0
    and.w   #0xf,%d0
    bne.b   dodump6      /* to tail check */
    /* put crlf */
    jsr     (crlf)
dodump6:
    sub.w   #2,%d1
    bge.b   dodump3
dodump4:
    /* all dump over, closing process */
    move.w  %a1,%d0
    and.b   #0xf,%d0
    beq.w   dodumpx
    /* do crlf if address %15 != 0 */
    jsr     (crlf)
dodumpx:
    /* pop registers */
    move.w  (%a7)+,%d1      /* pop %d1 */
    move.w  (%a7)+,%a1      /* pop %a1 */
    rts

```

このどこが12命令やねん。うーん。たぶん、もともと Moore 師匠が想定していたのは、アドレスもなく単に16進変換して出力するだけなんでしょうねえ。ここで私は頑張りました。結果は以下の通りです。だいたいよく見る見慣れた形式にしています。

```
COM4:115200baud - kuma@chinachu:~ VT
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) ヘルプ(H)
;start ss = 0, bp = 0000
123480102: 00 W M1
|80102 00 W OUT: 00
80103: 80 W M1
|80103 80 W OUT: 80
0080: 80100: 3A W M1
|80100 3A W OUT: 3a
0080: 307C 0080 303C 0024 4EBA 0064 60FE 3F00
0090: 1039 0008 00A1 0200 0002 67F4 301F 13CD
00A0: 0008 00A0 4E75 1039 0008 00A1 0200 0001
00B0: 67F4 1039
```

# Appendix.1 参考文献

---

## **Rather et al.93: The Evolution of Forth,**

ACM SIGPLAN History of Programming Languages Conference (HOPL II, 1993 年)で発表され、ACM SIGPLAN Notices, Volume 28, No. 3, March 1993 で発刊された。Moore がのちにForth と呼ばれる言語環境を構築していった歴史が描かれている。

## **Moore, 1970b: Moore, C.H., Programming a Problem-oriented Language.**

Amsterdam, NY: Mohasco Industries Inc. (internal pub.) 1970.