Fox プロジェクト

アマチアでも自作できる 32 ビット RISC ボード

暫定版

野田篤司

2001 年 10 月 7 日

はじめに

最新の 32 ビット RISC CPU を用いて，アマチュアがオリジナルコンピュータを設計し，手作りする方法を紹介する。コンピュータの自作といっても今流行の「自作 PC」のことではない。自分で回路を設計し，CPU やメモリを単体のパーツとして購入，はんだづけをしてコンピュータを作ることである。

Fox と名付けたこのコンピュータは，SH-3 を使った超小型の組み込み型ワンボードコンピュータである。Fox の設計・製作の過程の紹介を通じ，自分でコンピュータを作る方法と楽しさを伝えたい。

本文書の目的

本文書の最初の目的は、当然の事ながら、Fox 及び Fox プロジェクトの紹介と説明を行う事である。

Fox に限らないが、組込み目的のボード・コンピュータは、特にドキュメント類が整備されていない場合が多く、これが普及を阻害している。これは私自身にも当てはめて反省すべき点だ。

次の目的は、多分、CQ 出版で、出版されるであろう Fox プロジェクト関係の書籍のために文章を書き貯めておくことにある。[1]

対象となる読者

本文書の対象となる読者は以下のように想定できる。

- キットフォーム (未配線のプリント基板と全部品) を購入し、Fox を作ろうと考えている人、もしくは作っている人。
- Fox をロボット等のオリジナルな作品に使おうと考えている人、または、それを実行中の人。
- 自分自身で、完全に (CPU が SH-3 であろうが、無かろうが) オリジナルなコンピュータを白紙から設計し自作しようとする人。
- A 博士もしくは H 君もしくは M 子のファンの方々

最初の二つは、もっとも自然な本文書の読書対象者である。

それに対し、三つ目の「自分自身のオリジナル・コンピュータを作る人」は、 Fox のマニュアルとしての本ドキュメントの性格から外れたものであると思われるかもしれない。しかし、このような内容のドキュメントは、ほとんど無い。私自身が Fox を作る折に、そう言った文書を欲していた。Fox ではないにしても、新たなコンピュータを作ろうとする人も、 Fox プロジェクトの正当な後継者として、読者の対象とすることにしたい。

[1] まあ、本当に出版されるか、怪しいものだが

本文書の構成

本文書は次のような構成を持つ。

第 I 部 コンセプト編 Fox プロジェクトのコンセプトを判りやすく説明する。

ここでは、 A 博士もしくは H 君もしくは M 子の会話を通して、Fox プロジェクトの全体像を示している。

第 II 部 基本操作編 Fox を使ったアプリケーション・プログラムの作り方を説明する。

このとき、製作やホストコンピュータへの開発環境のインストールは既に済んでいるものとして、使い方やプログラミング、デバッグの仕方を示す。

第 III 部 製作・設定編 Fox キットの作り方と、ホスト・コンピュータにプログラミング開発環境のインストール方法を説明する。

Fox を使う前に、この第 III 部を実施しておく必要がある。

第 IV 部 設計・移植編 Fox の設計・過程と、eCos 及び NetBSD/sh3 の移植方法を説明する。

第 IV 部は、スクラッチからコンピュータを設計・製作したり、オペレーションシステムを移植しようとしている人の参考のために書いている。

Fox は、既に設計が完成しており、オペレーションシステムの移植も終っているので、 Fox ユーザーの全てが、この第 IV 部を読む必要は無い。

第 V 部 おわりに

付録 Fox の仕様等が記述されている。

本文書の内容

本文書は、現在、暫定版につき、特に図等が含まれていないなど、内容的に不充分のものであることを了承されたい。

本文書の著作権

本文書の使用条件

関連書籍・記事

現在までに、次の様な記事が出ているので、参考にしてもらいたい。

- インターフェース 1999 年 6 月号 小型 SH-3 ボード Fox の設計と製作 [1]
- インターフェース 1999 年 12 月号 SH-3 用コンパクトフラッシュ インタフェースの製作 [2]
- インターフェース 2000 年 4 月号 SH-3 ボードへの NetBSD の移植と実装 [3]
- インターフェース 2000 年 12 月号 Fox プリント基板と NetBSD/sh3 のインストール [5]

目 次

第I部

コンセプト編

第1章　Fox 一巡り

まず、 H 君と A 博士の会話を通じて、人通り、 Fox の紹介をしよう。
　登場人物は、

H 君 8 ビット・マイコンなら、使った事がある。ハードウェア指向。

A 博士 ちょっと、わざとらしい Fox の伝導師。

M 子 ソフトウェア指向、オブジェクト指向原理主義者で、UNIX 信仰者でもある。
　　反面、物事を抽象化しすぎる余り、現実を忘れることも多い。

である。
　の三人だ。
　Fox の製作と使い方は、次ぎの 4 つのステップから構成される。

- ステップ 1 『ハードウェアの製作と C によるプログラミング』
- ステップ 2 『リアルタイム OS』
- ステップ 3 『UNIX 互換 OS』
- ステップ 4 『ハードウェアの拡張』

　会話の中では、ストーリーの都合で、ステップを一気に 4 段階まで進めるが、実際には、ステップ・バイ・ステップで進めて行けば良い。また、無理にステップの順序を番号順に追う必要も無い。例えば、 Fox を作って、リアルタイム OS を使い、オリジナルのハードウェアを制御すると行った場合、ステップ 1 、 2 の順に進んだ後、 3 を飛ばして、 4 に進めば良い。
　逆に、特に OS も使わず、ハードウェアを拡張する必要も無ければ、ステップ 1 のままで留まっても、それなりの Fox を使うことができる筈だ。

1.1　ステップ 0 『準備』

Fox 登場

A 博士 「 H 君、何を悩んでいるのかね？ 」

H 君 「あっ、 A 博士。こんにちは。実は、今度、ちょっとしたロボットを作ろうと思っているのですが、コンピュータに PIC を使おうか、 H8 を使おうかで悩んでいたのですよ。

A 博士 「なるほど、そうか。それなら、 PIC や H8 も良いが、 もうちょっと上級を指向して、Fox と言うのも、どうだろう。」

H 君 「 Fox って何ですか？ 」

A 博士 「 SH-3 と言う 32 ビットの RISC CPU を使った、超小型のワンボード・コンピュータだ。PIC や H8 よりも、高機能・高性能で、 C のような高級言語を使えることはもちろん、リアルタイム OS や、 UNIX 互換の OS まで使える優れものだ。」

H 君 「32 ビット RICS に、UNIX 互換 OS ですって … 今回、作ろうとしているロボットって、ごく小さな物で、電源だって、乾電池を考えている程度のものですから、とてもそんな余裕はありませんよ。第一、予算オーバーです。」

A 博士 「そんな事は無いぞ。大きさで言えば、このポケット・ティッシュ程度のもんだし、消費電力も乾電池で十分なほどだ。

図 1.1: Fox

予算的にも、 PIC よりは高いかも知れないが、回路等の使用料は無料だし、ソフトウェア類は全てオープン・ソースだ。」

H 君 「それじゃ、 Fox に挑戦してみようかな？ 」

1.2　ステップ 1 『ハードウェアの製作と C によるプログラミング』

Fox を作る

A 博士 「これが、 Fox の全部品キットだ。」

図 1.2: Fox キット f

H 君 「えっ、完成品じゃ無いんですか !? それに、この小さな部品が沢山、とっても半田付けできそうにないですよ。」

A 博士 「始める前から、あきらめてはいかんな。それに、部品が多いと言っても、 UNIX 互換 OS が動くコンピュータにしては部品点数は少ない方だと思うが ···· 。」

H 君 「そんな事、言うなら、作ってみますよ。でも、何か、特殊な工具が必要なんでしょ ?」

A 博士 「そんな事は無い。秋葉原で、ごく普通に売っている半田ゴテで十分だよ。ただし、温度制御機能は必要だが、それでも、数千円で買えるぞ。半田だって、普通の物で十分だ。」

H 君 「じゃ、作ってみますよ ···· それにしても、小さい部品が多くって、目が痛くなって来たぞ。」

A 博士 「慣れていない者が、急にこんを詰めると疲れるぞ。ゆっくり、やった方が良い。それに急いでやると、ブリッジしたり、ろくな事は無い。

それから、コツと言う程のものでないが、 CPU とか EEPROM と言ったピン間隔が狭く、ピン数の多いものほどを、精神が集中している早いうちにに付けると良い。」

ROM ライタと RS232C インタフェース

H 君 「おはようございます。Fox 半田付け終りましたよ。ところで、どうやって動かすんですか ??」

A 博士 「あれ程、ゆっくりやれと言ったのに徹夜しおったな。」

H 君 「やり始めたら、区切りが付かないと止められないじゃ無いですか。それに、徹夜じゃ無いですよ。2 時間くらいは寝ましたから。」

A 博士 「とにかく、せっかく作ったんだから、テストしてみよう。まず、 Fox にプログラムを書き込まん事には動作テストもできないから、これから始めるか。」

H 君 「そう言えば、どうやって、 Fox の ROM にプログラムを書き込むんですか ? 小さな EEPROM が付いていましたが、半田付けしちゃいましたよ。」

A 博士「 Fox は、基板を組んだ後からでも、 ROM にプログラムを書くことができる。それには、専用の回路が必要だが、回路図は、これだ。ついでに、 RS232C インタフェースも作ったら、どうだろう。

Fox は、そのままでは何の外部インタフェースを持たないから、この RS232C インタフェースがあれば、シリアル・ケーブルでパソコンに接続できるんで、デバッグなどに便利だ。」

H 君「結構、作るものが多いですね。」

A 博士「だが、今度は部品も少ないし、デバイスのピンも標準ピッチだから、半田付けも簡単だろう。第一、 ROM ライタ・キット付属の FPGA には、既に回路が焼き込んであるから配線するだけで、 OK だ。」[1]

H 君「 FPGA って、何です？」

A 博士「いまどき、 FPGA も知らんのか。詳しくは、別の機会に説明するから、とりあえずは、簡単に作れるカスタム LSI の事だと思えば良いだろう。」

H 君「 判りました。とりあえず、 ROM ライタと RS232C インタフェースを作っちゃいますね。Fox に比べたら、ピン同士が開いているから、半田付けが楽だなあ。」

ホスト・コンピュータ

A 博士「 Fox にプログラムを焼き込むとなると、ホスト・コンピュータが必要だな。」

H 君「ホスト・コンピュータって、何です？」

A 博士「Fox で走らせるプログラムを開発するコンピュータの事だよ。Windows マシンや UNIX マシンの場合、プログラムは、コンピュータ自身に乗ったコンパイラで開発するのが普通だ。これをセルフ開発環境と言う。

ところが、H 君も、 8 ビット・マイコンを作った事があるなら、判るだろうが、組み込み用の小型コンピュータの場合、プログラムを開発するためのコンパイラを走らせる余裕は無いことが普通だ。だから、もっと大規模なコンピュータの上に、コンパイラなどの開発環境を乗せる。これをクロス開発環境と言う。

ホスト・コンピュータとは、クロス開発環境を乗せるコンピュータの事だ。Fox の場合、ホスト・コンピュータには、さっき作った ROM ライタを、プリンタ用のパラレル・ポートから制御して、 Fox の EEPROM に焼き込む機能も必要になる。」

[1] FPGA の書きこみ済みキットに付いては検討中

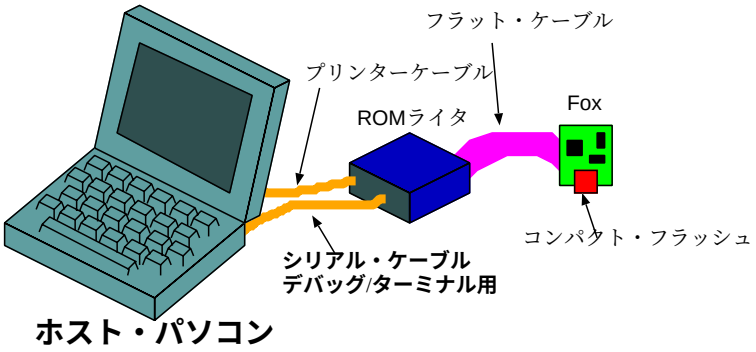

図 1.3: Fox のホスト・コンピュータ

H 君 「ええっ。Fox の他に、コンピュータが必要なんですか？僕は、 Windows 98 のデスクトップ・コンピュータを持っているのですが、それじゃ、駄目ですか？」

A 博士 「それで、十分だよ。ハードディスクに、数百メガの空き容量が必要だが。

ただ、理想を言えば、デスクトップよりもノート・パソコンの方が良いな。」

H 君 「なぜ、ノート・パソコンなのですか？」

A 博士 「理由は、二つある。

まず、一つ目の理由だが、 Fox を作ったときに判っただろうが、コンパクト・フラッシュ・インターフェースを標準装備している。このインターフェースを利用して、プログラムやデータをホスト・コンピュータと交換できる。

だが、デスクトップ・コンピュータの場合、多くは肝心のコンパクト・フラッシュに対して、読み書きするためのインターフェースを持っていない。

それに対して、ノート・パソコンは PC カード・スロットを持っているのが普通だ。PC カード・スロットがあれば、アダプタを介せば、コンパクト・フラッシュを使えるからな。」

H 君 「デスクトップ・コンピュータに、 PC カード・スロット・インターフェースを付ければ、その問題は解決するんじゃないですか。」

A 博士 「その通りだ。だが、気を付けて欲しいのは、もし新規に購入するなら、最近流行の USB 接続の PC カード・スロット・インターフェースは止めたほうが良い。できれば、 PCI か ISA 接続のものが良いだろう。

USB 接続のものでも、ホスト・コンピュータの OS に Windows を使っているうちは良いのだが、ステップ・アップして、 Linux や NetBSD を使うようになると、 USB 接続をサポートしていないからな。

H 君 「僕は Linux や NetBSD を使わないから、良いですよ。」

A 博士 「そんなことは無いぞ。なぜ、 Linux や NetBSD が必要になるかは、また、後で説明しよう。

それより、ホスト・コンピュータには、ノート・パソコンが理想だと言う二つ目の理由に戻ろう。

Fox の使い方は、ほとんどが、組込み用途だろう。プログラミングとかデバッグとかを考えた時、ホスト・コンピュータがデスクトップ・パソコンの場合、自由に動けない。だが、実際に使う場所で、プログラミングやデバッグできた方が効率的な事も多いだろう。

ノート・パソコンなら、自由にフィールドにでることができる。これは、ノート・パソコンが軽いと言うだけでなく、うまくシステムを構成すれば、 AC 100V の電源すら不要の、バッテリーだけのシステムも可能と言うことだ。

まさにフィールド、つまり、野に出ることが可能なわけだ。これが二つ目の理由だ。」

H 君 「ノート・パソコンですか。結構、高くつきそうだなあ。」

A 博士 「すまん、すまん。必ずしも、ノート・パソコンが必要な訳じゃなくて、あくまでも理想を言ったまでだ。

デスクトップ・パソコンだって、十分にホスト・コンピュータとして、使えるぞ。

それに、ノート・パソコンであろうとデスクトップ・パソコンであろうと、最新型の高性能なコンピュータである必要は無い。

むしろ、プリンター・ポートやシリアル・ポートが必要だから、古いコンピュータの方が良いかもしれん。」

動作チェック

A 博士 「ホスト・コンピュータの話が長くなったが、そろそろ、 H 君の作った Fox の動作テストを行おう。

取り合えず、今、 H 君が使っているデスクトップ・パソコンをホスト・コンピュータとして使ってみよう。

電源を入れる前に、 Fox の配線が正しいか、目視で確認しよう。」

H 君 「小さいから、見にくいですね。」

A 博士 「贅沢を言えば、実体顕微鏡が欲しいところだが、むし眼鏡で十分だ。

うーーん、何箇所か、ブリッジしているところがあるな。」

H 君 「えっ、嘘。」

A 博士 「ほれ、ここだ。

吸い取り線を当てて、半田ゴテを押しつければ、余計な半田を取り除くことができる。」

H 君 「そうですね。これで、どうでしょう。」

A 博士 「ちゃんと直っているようだ。このように、半田付けの後からでも、修正は可能だから、組み立ての時に余り神経質になる必要は無い。

さて、テスターで、電源がショートしていないことを確認して … 」

H 君 「流石に、それは大丈夫のようです。」

A 博士 「いよいよ、テストプログラムを書き込むぞ。プログラム自体は、 CD-ROM の中に入っているから、これを書き込むだけだ。」

H 君 「どうやって、 CD-ROM の中のプログラムを Fox に書き込むんですか。」

A 博士 「さっき、作った ROM ライタを使うのだ。こうやって、ホスト・コンピュータの CD-ROM ドライブに、 Fox キットに付属している CD-ROM を入れる。

ホスト・コンピュータと ROM ライタは、プリンター・ケーブルで接続する。

また、 ROM ライタと Fox は、フラット・ケーブルで接続するのだ。

最後に、 ROM ライタに電源を接続する。電源には、5V と 3.3V の二つが必要なところが、この ROM ライタの面倒な所と言えば、面倒な所だな。」

H 君 「 Fox 自体は、3.3V 単一電源ですからね。」

A 博士 「ホスト・コンピュータから、 ROM ライタをコントロールするプログラムも CD-ROM に入っている。

Windows のエクスプローラから、 ROMW.EXE を起動する。

プログラムから、動作テスト用のプログラム rom011.hex を開く。」

H 君 「プログラムは、 Hex 形式になっているんですね。」

A 博士 「ライトボタンを押せば、プログラムを Fox に書き込む。この時、プリンタ・ポートを双方向モードになっていなかったら、 BIOS 設定で、プリンタ・ポートを双方向モードに設定する必要がある。

設定があっていて、 Fox の配線ミスも無ければ、エラーも出ず、プログラムが書き込めるはずだ。」

H 君 「ちゃんと、書き込めたみたいですよ。」

A 博士 「念のため、ベリファイを取っておこう。」

H 君 「ベリファイでも、エラーが無いようですね。」

A 博士 「いよいよ、動作テストだ。

まず、電源を切って、 RS232C インタフェースに接続しなおそう。」

H 君 「何故、 RS232C インタフェースが必要なのです？ 」

A 博士 「 Fox 自体は、何も外部インタフェースを持っていないからだよ。だが、動作チェック・プログラムを走らせたら、その結果を表示する必要があるだろう。それに使うんだ。」

H 君 「それじゃ、 RS232C インタフェースとホスト・コンピュータをシリアル・ケーブルで接続しますよ。」

A 博士 「シリアル・ケーブルは、ストレート接続かな？ さて、ホスト・コンピュータ側のターミナル・ソフトを設定しよう。」

H 君 「ターミナル・ソフトって、ハイパー・ターミナルで良いですか？ 」

A 博士 「 Windows に最初から付いている奴だな。それでも良いが、フリーウェアの TeraTerm Pro と言うのがお奨めだ。今回はハイパー・ターミナルでも構わんが、後々役に立つから、ダウン・ロードしておいて、損は無い。

取り合えず、ハイパー・ターミナルを起動して、シリアル接続で、ビットレートは 9600 ボーだ。」

H 君 「じゃ、 Fox 側の 3.3V 電源を入れますよ。

ああっ、ターミナルに、 Fox のメッセージが出ましたよ。」

A 博士 「ちゃんと動いているって、事だな。」

H 君「でも、メモリのチェック中と言うメッセージで、止まっちゃいましたよ。壊れているのかな？」

A 博士「慌てちゃいかん。

今、 8M バイトあるメイン・メモリを全て、チェックしている最中だ。

長周期の疑似ランダム・データを作り、読み書きができるかチェックすると言う凝ったプログラムだから、ちょっと時間がかかるのだ。」

H 君「メモリのチェックが無事終了したって、メッセージが出ましたよ。」

A 博士「これにて、動作チェックは一件落着だな。」

プログラミング

H 君「一応、動作試験はできました。でも、 Fox にどうやってプログラミングするんですか？

こう言う組込み用コンピュータって、プログラムできて、初めて使いものになるわけですから。」

A 博士「実は、 Fox にプログラムする方法は、幾つかある。大きく分けて、

- OS 無しでプログラム
- リアルタイム OS を使う
- UNIX 互換の NetBSD を使う

の三つがある。

最初は、一番簡単な『 OS 無しでプログラム』する方法を使ってみよう。」

H 君「プログラムって、どんな言語を使うのですか？」

A 博士「取り敢えずは、 C かな。」

H 君「えっ、 C のコンパイラって、結構高いんですよね。」

A 博士「それは、 CPU を作っているメーカー製のコンパイラの話だ。

Fox のキットに付属している CD-ROM には、ちゃんとフリーのコンパイラが入っている。もちろん、 gcc だが、ちゃんと ROM 化も可能だ。」

H 君「gcc って、設定とか面倒じゃありませんか？」

A 博士「面倒が全く無いと言ったら、嘘になるが、それ程大変なものではない。

必要なファイル類は、全て、 CD-ROM に入っているから、 Windows 上にインストールするだけだ。ここ[2]を見ながら、指示通りに操作すれば、インストールは完了する。確かに手順は多いし、時間はかかるが、大した事は無い。」

H 君「うーーん、コリャ、結構時間が、かかりそう。」

A 博士「私は、お茶でも飲んでいるから、ゆっくりやりたまえ。」

[2]工事中

簡単なハードウェアの操作

A 博士 「どうかな？ gcc のインストールは済んだかな？」

H 君 「済みましたよ。でも、 gcc のインストールだけかと思ったら、 binutils もインストールするんですね。」

A 博士 「高級言語のコンパイラである gcc だけでは、不十分で、どうしても、 binutils に入っているアセンブラやリンカ、ローダーが必要になる。」

H 君 「クロス開発環境も整ったのですから、プログラムを作りたいと思うのですが。せっかくですから、ハードウェアの制御をしたいと思うのですが、どうでしょう？」

A 博士 「そうだな。いきなり、複雑なハードウェアの制御をしても難しいから、最初は、発光ダイード LED を点滅させることから、始めようか。

Fox には、入力も出力にも使えるポートが、8 ビットだけだが、標準で付いている。

この 8 ビットのポートは、 Fox と言うより、 SH-3 が、標準で持っているのだが、 Fox は、このポートを拡張バス用のコネクタに出しているんだ。」

H 君 「なるほど、そのポートに LED を接続して、プログラムで点滅させる制御をするんですね。」

A 博士 「そういうことだ。但し、ポートから出せる電流は、わずか 2mA だから、 LED に直列に電流制限用の 2k Ω の抵抗を付ける必要がある。LED を余り明るく光らせることはできないが、確認用には十分だろう。」

H 君 「なるほど … コネクタの CN2 の 19 ピンから 33 ピンに、一つ置きに出ていますね。抵抗を介して、 LED を半田付けするっと。簡単、簡単。」

A 博士 「あっと言う間に作りおったな。

後は、今度までの課題と言うことにしよう。

ヒントを与えておくと、 8 ビットのポートを使うためには、 3 つのレジスタに書き込みを行う必要があると言うことだ。一つは、ポートを入出力に使うことを許可するレジスタ、二つ目は、ポートの入力出力の方向を制御するレジスタ、最後は入出力のデータを読み書きするレジスタだ。」

機械語と コンパイラ

A 博士 「おはよう、 H 君。おや、その真っ赤な眼はどうしたんだね。」

H 君 「 A 博士、おはようございます。昨日は本当に徹夜してしまいましたよ。」

A 博士 「どうしたね。SH-3 のポートの使い方が判らなかったかね？ CPU のハードウェアマニュアルを読めば、すぐに判ると思うのだが。」

H 君 「いえ、それはすぐに判りました。SH-3 の『SH7708 ハードウェアマニュアル』は、付属の CD-ROM [3]にも入っていますし、最新版は、インターネットから、ダウンロードできますから。

昨日、博士が言っていた、3 つの制御用レジスタもすぐ判りました。」

A 博士 「それじゃ、全く問題ないじゃ無いか。何が悪かったのかね。

制御用レジスタのアドレスとか書き込む値が判らなかったのかね。」

[3]本当に付録 CD-ROM に収録できるかどうかは、著作権を持っている日立製作所に確認する必要がある。

H 君「アドレスも書き込むべき値もすぐに判りました。ところで、 SH-3 ってメモリマップド I/O だったんですね。

問題は、どうやって、書き込むかが判らなかったんですよ。

同じ付属の CD-ROM に SH-3 の『SH7700 シリーズ プログラミングマニュアル』[4]を徹夜で読んだんですが、どうしても理解できなかったんです。」

A 博士「SH-3 のプログラミング・マニュアルって、まさか、機械語でプログラムしようと思っているんじゃないだろうね。」

H 君「もちろん、機械語ですよ。機械語でコントロールするのが、ハードウェアを操作する基本ですからね。」

A 博士「いまどき、機械語じゃ無いだろう。」

H 君「あっ、もちろん、アセンブラは使っていますよ。昨日、インストールした binutils の中にアセンブラも入っていましたから、それを使おうと思ったのです。だから、正確には機械語ではなくてアセンブラ言語ですかね。

でも、とにかく、マニュアルを読んでも、決まった値を決まったアドレスに書き込む、そんな単純な命令がどうしても判らないんで、困っていたんです。」

A 博士「機械語も、アセンブラのニーモニックも基本的には、 1 対 1 で対応しているから、同じようなものだ。」

H 君「で、やっと作ったプログラムは、複雑怪奇なものになってしまうんですよ。

もっと、洗練された命令がある筈だと、一晩中、マニュアルを読んでいたのですが、どうしても見付からないので、困っていたんです。」

A 博士「そりゃそうだろう。複雑怪奇でも、自力で機械語でプログラムを作った方が凄いな。」

H 君「どうも、話が噛み合いませんね。

とにかく、一番、心配しているのは、 LED を点滅させるだけの単純なプログラムでさえ、こんな複雑怪奇なプログラムになるのなら、ちょっとでも難しい事をやらせようと思ったら、プログラムを作るので、気が狂いそうな事なんですよ。」

A 博士「そもそも、機械語でプログラムしようとした事自体が間違っていたのだよ。C 言語を使うべきだったね。

昨日、 LED の点滅プログラムと言う課題を出した時に示唆すべきだったが、 SH-3 のような RISC CPU を使うときは、機械語は使っては駄目で、 C 言語のようなコンパイルできるプログラム言語を使う必要があるんだ。」

H 君「なぜ、機械語じゃ駄目なんですか？ 今回のような単純なプログラムを組むときだったら、機械語でプログラムした方が、 C プログラムに特有の面倒な事を考えるまでも無く、ハードウェアを制御できると思うのですが。」

A 博士「君自身で経験した通りさ。単純な LED の点滅プログラムでさえ、機械語でプログラムすると、複雑怪奇なプログラムになってしまうから、機械語でプログラミングしたらいけないのさ。」

H 君「えっ、複雑怪奇なプログラムが正解なんですか？ 僕は、てっきり、もっと洗練された命令があると思って、徹夜でマニュアルを探したのですが。 」

---

[4]同様に、本当に付録 CD-ROM に収録できるかどうかは、著作権を持っている日立製作所に確認する必要がある。

A 博士 「逆に、君に質問しよう。

なぜ、もっと洗練された命令があると思ったのかね。」

H 君 「そりゃ、洗練された命令があれば、人間だって、コンピュータだって、命令の意味を理解し、実行するのが有利な筈ですから。」

A 博士 「本当にそうかな。」

H 君 「僕の作った複雑怪奇なプログラムは、ステップ数が多いんですよ。もっと洗練された命令があれば、ステップ数は半分にはなると思います。

半分のステップ数のプログラムなら実行時間は半分、つまり、処理速度は 2 倍になるわけですよね。」

A 博士 「なるほど、やっと君が何処で間違えたか判ったぞ。君の機械語プログラムの経験は、 8 ビットの CISC CPU の経験だけだったな。

つまり、元々、君の言うところの『洗練された機械語の命令系統』とは、『人間にとって判りやすい機械語の命令系統』と言う意味だな。

そして、次のような論法が通るわけだ。

- 洗練された命令系統なら、ステップ数が少なくなる。
- ステップ数が少なくなれば、実行速度が速くなる。
- つまり、洗練された機械語の命令系統を持てば、人間もコンピュータにも有利。

」

H 君 「そうですよ。その通りじゃないんですか。」

A 博士 「違うね。

君の考え方の結論は、『洗練された機械語の命令系統を持てば、人間もコンピュータにも有利』と言うことだが、現実には、人間とコンピュータの両方に有利な命令体系など存在しない。」

H 君 「本当ですか。」

A 博士 「君が、そのような誤った認識を持つことも仕方が無いことかもしれない。15 年ほど前まで、コンピュータの専門家でも、そう言った意見を持った人が多数派だったんだ。

実際は、洗練された機械語命令、言い換えれば、人間に判りやすい命令体系は、コンピュータに取っては複雑怪奇なものなのだよ。

君が今までに使ったことのある CISC タイプの CPU は、できる限り、人間に判りやすい命令体系にしていたんだ。

ところが、人間にとって判りやすい命令は、コンピュータに取っては複雑だから、コンピュータに取って判りやすい、もっと簡単な命令に翻訳しながら、実行していたんだ。」

H 君 「博士が言っているのは、大昔に流行った BASIC 等のインタープリタ言語の事じゃないんですか？ 」

A 博士 「違う、違う。機械語の話をしているんだ。でも、多くの CISC (Complex Instruction Set Computer) の場合、まさにインタープリタ言語と同じ様に、命令を翻訳しながら実行しているんだ。

ここで、逆の発想がある。コンピュータに取って判りやすい命令、つまり、実行するときに翻訳された結果として生成されるような、簡潔な命令だけで、最初から機械語でプログラムするようにしたらと言う発想だ。

これが、 RISC (Reduced Instruction Set Computer) だ。」

H 君「 Fox で使っている SH-3 は、 RISC CPU ですね。 」

A 博士「その通り。だから、 Fox に機械語でプログラムすると、コンピュータには良くても、人間にとっては複雑怪奇なプログラムになってしまうのさ。」

H 君「本当にあんな複雑怪奇なプログラムがコンピュータに良いんですか ？さっきも言いましたが、ステップ数が、ひどく増えるから、実行速度はむしろ低下すると思うのですが。 」

A 博士「その心配は要らない。元々、 CISC の時に翻訳した後の実際に実行する命令に近いわけだら、実行速度は落ちない。

例え、ステップ数が二倍になっても、ステップ毎の実行速度を 2 倍にすれば、実質上の実行速度は同じになるわけだ。」

H 君「実行速度が同じなら、ステップ数が増えるだけ、メモリを大喰いする分、不利だと思うのですが。」

A 博士「実際は実行速度は落ちるどころか、むしろ、速くなる。ステップ数が増えても、翻訳と言う処理が不要になる分、有利だからだ。

また、翻訳と言う機能自体が不要になるから、 CPU のチップから、これに必要な部分が要らなくなる事の効果も大きいな。同じ性能なら、機能が減った分、コストも消費電力も下がる。

逆に、翻訳に使っていた部分を、パイプライン処理とかメモリ・キャッシュに使うことで、同じコスト・消費電力なら、より高速な CPU を作る事ができる。

RISC CPU は、こう言ったメリットがあり、 SH-3 は、高性能を維持したまま、低消費電力と低コストに重点を置いて開発された CPU だと言うことができるな。」

H 君「処理速度の方は、判りました。

でも、ステップの増加によるメモリの大喰いはどうするんです。

それに、機械語で組んだプログラムの複雑怪奇な事。あんなプログラムが必要なら、本当に記が狂っちゃいますよ。」

M 子「じれったいわね。頭、腐ってんじゃないの ？

さっきから、博士が何度も言っているじゃない。プログラムを機械語で組むこと自体が、アナクロなの。

機械語なんて、コンパイラに任せりゃ良いのよ。」

M 子 登場

A 博士「おや、 M 子 君じゃないか。何時から、そこで話を聞いておったのかね。」

M 子「そこの腐った子が、 LED を制御するプログラムを機械語で書こうってあたりからです、博士。」

H 君「なんだよ。言ってくれるな。」

M 子「言うわよ。

まず、コンピュータに最適化された RISC の命令系統で、人間がプログラムしようとすること自体が、馬鹿なの。

そんなこと、コンパイラに任せちゃって、人は高級言語で抽象化されたアルゴリズムに専念してれば良いわけ。」

H 君 「そんなこと言ったって、ハードウェアを制御するには、微妙なコントロールとか、処理を速くするために、機械語でプログラムするのが、基本なんだぜ。」

M 子 「そんなの 処理速度も遅くて、メモリも録に乗っていない PIC とか H8 とか使って居る事の言い訳なの。

メモリが潤沢にあって、処理速度が十分速ければ、機械語でプログラムする必要なんか無いのよ。」

H 君 「博士、あんな事言ってますよ。」

A 博士 「H 君、残念ながら、 M 子君の方が、ほとんど正しい。

但し、 M 君、機械語によるプログラムは、完全に必要が無い訳ではないよ。どうしても、機械語で無いとプログラムできない処理は、今でも機械語でプログラムすることはあるのだ。」

M 子 「逆に言えば、コンパイラでできる処理は、全て、コンパイラでプログラムすべきと言うことね。

機械語を使って、処理速度を上げるとか、メモリを効率的に使うとか、そういう小手先の技が、重要なテクニックだったのは、CPU の処理速度が遅くてメモリとかコンパイラ自体が高価だった大昔の話よ。

今は、潤沢なメモリと高速な処理速度と、最適化コンパイラを用いて、人間は抽象化された問題の解決に専念することが、トータルとして、プログラムの生産性を上げることになるの。」

A 博士 「 Fox の場合、 C プログラムに初期化で必要な機械語のルーチン (crt0.s) は既に用意されているから、 C によるプログラムは、非常に簡単なのだ。

実際、 LED を点滅させるプログラムは、次の通りだ。

led.c

```
void main(void)
{
  int i;
  (*(short *)0xFFFFFF62) = 0x2aa9;  // ポートを入出力に使うことを許可
  (*(short *)0xFFFFFF76) = 0xffff;  // ポートを出力に設定。
  for(;;) {
    (*(char*)0xffffff78) = 0xff;    // ポート、全ビットに 1 を出力
    for(i = 0; i < 100000; i++);    // 適当に待つ
    (*(char*)0xffffff78) = 0x00;    // ポート、全ビットに 0 を出力
    for(i = 0; i < 100000; i++);    // 適当に待つ
  }
}
```

」

H 君 「えっ、たった、それだけで、良いんですか。本当に、それだけなら、昨日の徹夜は何だったんだろう。」

M 子 「全くの無駄 … ね。」

H 君「‥‥」

M 子「もしかしたら、あなた C でプログラムするの苦手なんじゃ無い？」

H 君「実は、そうなんだ。」

M 子「じゃ、私のカーニハン & リッチー [6] を貸して上げるわ。今度までに良く読んでおく事ね。」

H 君「げっ、カーニハン & リッチーかい？ もっと、やさしい本は無いの ?」

M 子「無いわ」

コンパクト・フラッシュ

数日後：

A 博士「H 君、おはよう。おや、素敵なノート・パソコンを持って来たね。」

H 君「おはようございます。このノート・パソコン、叔父さんに貰ったものなんです。
この間の『ホスト・コンピュータはノート・パソコンが理想』と言う話を叔父さんにしたら、古くて使わなくなったノート・パソコンをくれたんです。
CPU は 無印 Pentium の 100 MHz だし、OS は Windows 95 ですが、使い物になりますよ。
もう、クロス環境もインストールしています。」

A 博士「なるほど、ちゃんと、PC カード・インタフェースも付いているし、これならば、コンパクト・フラッシュも使えるな。」

H 君「そう言えば、以前、ホスト・コンピュータの話をしたときにコンパクト・フラッシュの話がありましたが、 Fox と、どういう関係があるんですか？」

A 博士「知っての通り、 Fox には、コンパクト・フラッシュ用のコネクタが用意されているね。これを利用して、プログラムを Fox に読ませることができるのだよ。

図 1.4: Fox コンパクト・フラッシュ付き

今までは、 ROM ライタを使って、 Fox のボード上にある EEPROM に直接プログラムを書いていたね。この方法だと、ホスト・コンピュータ以外に、 ROM ライタが必要だとか、そのための電源が居るとか、 ROM ライタ用のコネクタへの接続が面倒だとか、色々と問題もある。

この方法の問題点は、 EEPROM の容量である 128K バイトにプログラムの大きさが制限されている事と、 Fox には恒久的な記録媒体が無いことだね。

コンパクト・フラッシュを使えば、これらの問題が一気に解決する。Fox ボード上の EEPROM に、 IPL プログラムを焼いておけば、コンパクト・フラッシュにプログラムを書き込むだけで、起動時に RAM 上にロードして実行してくれるのだ。

Fox の RAM は、ROM と違って、容量が 8M バイトもあるから、プログラムの制限は大幅に改善されるぞ。」

H 君 「僕にとっては、プログラム領域が 128K もあれば、十分ですが、プログラムの焼き込みが楽になるのは、魅力的ですね。実際には、どうやるんのですか？」

A 博士 「たいして、難しくは無い。コンパイルするときに、スタートアドレスを、 0x0c000000 にして、バイナリモードで、セーブするだけだ。付属の CD-ROM にサンプル・プログラム用の Makefile が用意されているから、これをそのまま使うと早い。

できたファイルを、コンパクト・フラッシュのルート・ディレクトリに FOX.FOX と言うファイル名で、コピーするだけで、終り。

後は、コンパクト・フラッシュを Fox に付けて、起動すればおしまい。」

H 君「なんて、簡単なんだ。これじゃ、 ROM ライタなんか要らないじゃないですか。」

A 博士「コンパクト・フラッシュ自体がある程度、高価だから、デバッグ時には便利でも、実際に動作させるときには、 EEPROM の方が低コストと言うメリットがある。

だから、用途に合わせて、使い分ければ良いのだ。」

1.3　ステップ 2 と 3 への準備『 Fox と OS 』

Fox で使える オペレーション・システム

A 博士「ところで、そのノート・パソコンについているハード・ディスクは、どのくらいの容量があるのかね。」

H 君「750 M です。昔のパソコンですから、小さいですよ。プリ・インストールのワープロとか表計算ソフトを消して、やっと、開発環境をインストールできましたから。」

A 博士「うーーん、贅沢を言えば、ハード・ディスクは、買い替えて、もうちょっとだけ、大容量にしたいところだね。

そうすれば、パーティションを切って、 Linux とか NetBSD を入れることもできる。」

H 君「僕は、 UNIX 系は使いませんから、別に要りませんよ。

第一、このノート・パソコンは、 Fox のホスト・コンピュータ専用にしようと思っているですから、それ開発環境以外のソフトウェアを入れる気はありません。」

A 博士「いや、そういう意味じゃなくて、 Fox の上で、 リアルタイム OS とか UNIX 互換 OS を使おうと思ったら、ホスト・コンピュータ側にも、 Linux や NetBSD が必要になって来るんだよ。」

H 君「リアルタイム OS や、 UNIX 互換 OS ですか !? そう言えば、前にもそんな事言ってましたが、 OS なんて要らないですよ、僕は、ちょっとしたハードウェアを制御したいだけですから。」

M 子「まだ、そんな事言っているの !!

馬っ鹿じゃないかしら。」

H 君「うああ、また出た。」

OS を使うメリット

A 博士「 M 子君が、説明すると過激になるから、私から、説明しよう。

H 君のロボットに、何をさせるのか聞いていなかったから、ちょっとした例え話で説明するぞ。

あるところに、飲んだくれの親父と息子が居たとしよう。親父が、酒が切れたので、子どもに買いに行かせる。

子どもは、親父には逆らえないので、一升壜を持って、酒屋へでかける。」

H 君「まるで、落語ですね。

ところで、今時、一升壜に秤り売りしている酒屋なんて、ありませんよ。」

A 博士 「子どもは、歩きながら、駅前に酒屋があった筈だ、あそこへ行こうと考える。」

H 君 「駅前にあった酒屋さん、コンビニになっちゃいましたよ。」

A 博士 「そうか、あそこもフランチャイズされちゃったか。うまい酒が置いてあったんだが。

それは、ともかく、ここで重要なのは、子どもが歩きながら、何処へ行こうかと考えたと言う点なのだよ。歩くと言う行為は、普段、何気なくやっているが、実は、これは大変な制御なのだ。前後左右に傾く身体を立て直しながら、歩行を続ける。この時の制御に必要な処理速度と言うのは、一秒間に 10 回とか 100 回とかの高速性だろう。

それに比べて、何処の店に行こうかと考えること自体は、もっとゆっくりとしたレスポンス性で良い。たぶん、数秒から数十秒のオーダーだろう。

さて、目的となる酒屋が決まった後は、子どもは何処をどういったら、近道になるかを考えながら、歩く。大通りを行くより、八百屋の横の小路を通った方が早いってね。」

M 子 「最適ルート探索ね。」

H 君 「八百屋はスーパーマーケットになりましたよ。それに、あの小路は下水工事で通れません。」

A 博士 「そう、あの道は通れないのだよ。子どもは、八百屋、じゃなくてスーパーマーケットまで来て始めて、それを知る。

ここで、子どもは立ち止まったりはしないだろう。取り敢えず、駅の方向へ歩きながら、やっぱり、大通りを歩いて、三丁目の煙草屋の角を曲がろうって、考えるわけだ。」

H 君 「三丁目の角にあった煙草屋は 100 円ショップになっています。」

A 博士 「いちいち、うるさいな。

子どもは、三丁目の角に来て、右に曲がる。

この時、歩行と言う高速の制御、道を探すと確認すると言ったレスポンスの遅い処理の他に、道を曲がると言う中程度の処理を行う。」

H 君 「道を曲がるのは歩行と同じでしょう。」

A 博士 「要は考え方なのだが、曲がると言う行為である位置制御は、歩行よりも遅い制御と考えた方が良いと思うよ。

もちろん、見解の相違はあるかもしれないが、ここでは、単に三つの速度の異った処理を同時に行っていると言う事の例だと思ってくれ。」

H 君 「なんとなく判ります。」

A 博士 「三丁目の角を曲がると、すぐに酒屋さんだ。こうして、無事に子どもは酒屋にたどり着き、飲んだくれの親父の為に酒を買うことができるわけだ。」

H 君 「それは、無理ですよ。」

A 博士 「何故かね？」

H 君 「法律で、未成年に対する酒類の販売は禁じられています。この子に、酒を売ってくれないでしょう。」

マルチ・タスク

A 博士「話が、思いっきり脱線したが、私の言いたかったのは、速度の異る処理とかレベルの異る処理を同時に行う必要があることだ。

簡略化の為に省略したが、実際には、歩行すれば、手・足・腰と言った部分毎に細かい制御があるし、歩きながら自分の場所を確認する必要があったり、すれ違った人に挨拶したりと、もっともっと複雑な数多くの処理を同時に行う必要がある。

君が、作ろうとしているロボットも、程度の差はあれ、複数の処理を同時に行う必要があるのではないかね。」

H 君「そうかも知れません。」

A 博士「そこで、 OS の必要性が出て来る。複数の処理を、それぞれ別のプロセスに分けて、実行するんだ。

Windows や MAC でもやっているが、タイム・シェアリング・システムと言う奴だな。」

H 君「TSS マルチタスクですね。

でも、わざわざ、 OS を使うまでもなく、プログラムを、工夫すれば、処理時間の問題は解決すると思います。」

M 子「そんなこと言っているから、あなたはタコなのよ。プログラムで工夫と言ったて、限度があるわ。

簡単な内はともかく、ちょっと複雑な組合せになったら、すぐに破綻することなんか、判り切っているじゃない。」

A 博士「 M 子 君の言い方は乱暴だが、言っていることは、その通りだよ。

複数のプロセスを同時並行で処理することは簡単ではない。

自分で作ったプログラムの中で、この処理を行うことは不可能ではない。同時に動かす処理の数が少なく、組合せが単純な内は、それでも良いだろう。

でも、処理の数が増え、処理間の組合せが複雑になると、タスク管理が破綻することは無くても、そのメンテナンスに追われて、肝心のプログラムに手が回らなくなる。

だから、タスク管理は、出来合いの OS に任せた方が安心だし、効率的なのだよ。」

H 君「処理の数が増えると言うのは判りますが、組合せが複雑というのは、どういう意味です？」

M 子「判んない子ね。

例えば、さっきの例だったら、酒屋へ行く最適ルート探査のタスクから、位置制御のタスクへ『煙草屋の角を右に曲がる』と言う指令が伝わるはずよね。

これがプロセス間通信だわ。こんな通信、あなたはどうやって解決するつもりだったの。他にも、道を歩いて居て、向こうから歩いて来る人が、学校の先生だったら、どうする。

『挨拶をする』ってタスクを緊急に起動しなきゃならないでしょ。この『挨拶タスク』は定常に動いているわけじゃないから、本当に緊急的の時しか起動しないわけよ。

こう言った色々なタスク間の通信とか、起動指令のつながりの事を『複雑な組合せ』って呼んだの。」

A 博士 「いま、 M 子 君が言った、タスク間通信とかシグナル送信やセマフォ等の制御は、 OS の基本だ。色々と複雑な状況を考えると、これらの処理のプログラムは、かなり難しいと言って良いだろう。

H 君、君が自ら、これらの処理をプログラムすることができないとは言わないが、相当身を入れてプログラミングしないと、まともに動くものはできまい。それでは、そもそも君の目的としていたロボットの為のプログラムを作る時間が無くなってしまう。

だから、既成の OS に、こう言った複雑な処理は任せた方が良いのだ。」

H 君 「でも、複雑だとか、時間的にクリティカルな処理は、 OS を使わずにプログラムした方が最適化できると思うのですが。また、 OS って、結構、大きいでしょ。メモリがもったいない …· 」

M 子 「処理速度とかメモリとか、コンピュータのリソースを、小手先のテクニックで節約するのはアナクロなの。潤沢なリソースで、 OS を使えば良いじゃない。

メモリが足りなきゃ、メモリを足せば良いし、速度が間に合わなければ、もっと速い CPU に換えれば良いのよ。

PIC みたいに、プログラム領域 1k、データ・メモリ 68 バイト、H8 でも、プログラム領域 128k、データ・メモリ 4K バイトのような物ばっかり使って、せこせこしているから、貧乏人根性が染み着くのよ。」

A 博士 「 M 子 君の意見は過激だが、ある程度、的を獲ていることも事実だな。

実際、 Fox の ROM 128K バイト、RAM 8M バイトと言うボード・コンピュータとしては大容量メモリも、50 MIPS を超える高速処理も、その為にあるようなものだ。」

M 子 「そういう事。」

A 博士 「 OS が、マルチ・タスクを行う上で必要だと説明したが、 Fox で使える OS は、現時点では、

- リアルタイム OS 系の eCos
- UNIX 互換 OS 系の NetBSD

の二種類だ。[5]

大雑把に言うと、リアルタイム OS とは、反応速度の異る処理を同時に行う為の OS だ。また、 UNIX 互換 OS とは、知っての通り、極めて高度な処理を行うことに向いた OS だ。」

1.4 ステップ 2 『リアルタイム OS 』

リアルタイム性

A 博士 「リアルタイム OS は、処理に対して、時間的にシビアに管理していることが、特徴だ。

さっき話した子どもの場合、歩行に必要な制御は、非常に時間的に厳しい。ちょっと、反応が遅れただけで、転んでしまう。

例えば、『道が工事中だから、何処の道を歩こうか。』と夢中になって考えていても、右足を前に出し忘れることなど無いよな。

これは、人間の制御系が、極めて高度なリアルタイム性を実現しているおかげだ。

[5] Linux については、今後、移植の可能性が高い。

下手なタイムシェアリングシステムの場合、あるタスクに負荷がかかると、他のタスクの反応が遅くなってしまう事もある。こんな場合は、右足を出し忘れて、転んでしまう訳だな。

このように、どんなに他の処理が忙しくても、高速性を必要とする処理は、必ず要求される時間内に処理を行うように、タスク管理する OS が、リアルタイム OS なのだよ。」

H 君「つまり、処理速度の速い OS って言うことですね。」

M 子「全然、判ってないわけね、あなたは。」

H 君「なんだよ。結局、リアルタイム OS は処理が速いって、事じゃないのかよ。」

M 子「処理が速いか、遅いかは、 CPU の処理速度で決まる問題で、リアルタイム性とは関係ないわ。

リアルタイム性って言うのは、あくまでも、マルチタスクの切替えの問題。

必要なタスクを、必要な時間内に実行させると言うことが、リアルタイム性なの。」

A 博士「結局、 CPU の処理速度は、有限なリソースで、 OS が良かろうが悪かろうが、トータルとして、増えるものでも減るものでもない。

だから、処理を急ぐタスクに優先的に、リソースを割り振って、処理の時間を保証するものだ。

逆に、処理を急ぐタスクに処理時間を割り振る分、処理を急がないタスクは実行時間が減るのだ。

歩くと言う処理は、位置制御や、最適ルート探索よりも優先順位が高い。

ゆっくり歩いている内は、歩くと言う処理に余裕があるので、位置制御や最適ルート探索に割く時間もある。

ところが、何かにつまずいたりして、倒れそうになったとしよう。こう言った緊急時には、急に歩行と言う処理、と言うかバランスを取る処理にかかる負荷が増える。

だから、位置制御をしている暇が無いから、道の真中からずれたりする。ましてや、どうやったら近道か ‥ なんて考えている余裕は無いな。

こういう風に、タスクの時間管理を臨機応変に行う機能が、リアルタイム OS には必要なのだよ。」

H 君「そんな大変な機能を持つのなら、リアルタイム OS は、高価だったり、メモリを大喰いだったりしませんか。」

A 博士「リアルタイム OS は、一般に市販されているものは、非常に高価なのが普通だ。[6]

だが、 Fox で使われている eCos と言うリアルタイム OS は、ロイヤリティー・フリーのオープン・ソースな OS だ。Fox の付属の CD-ROM に、全ソース込みで収録されているから、安心だ。

また、 eCos のカーネル・サイズは小さいので、 Fox のボード上 EEPROM にも乗る大きさだ。」

H 君「なるほど、それなら、僕も、リアルタイム OS を使ってみようかな。」

[6] リアルタイム OS の場合、 OS の使用料自体は、それほど高価ではなく、コンパイラ等の開発環境が高価の場合が多い。家庭用の電化製品や自動車のエンジン制御、自動販売機等にリアルタイム OS が多用されていることからも、 OS 単体の使用料が安価であることは想像できる。だが、本文での内容では、 H 君が、自作ロボットに使うと言う設定なので、開発環境が必要であり、 A 博士の『リアルタイム OS は高価だ。』の言葉は、開発環境込みの価格を指している。

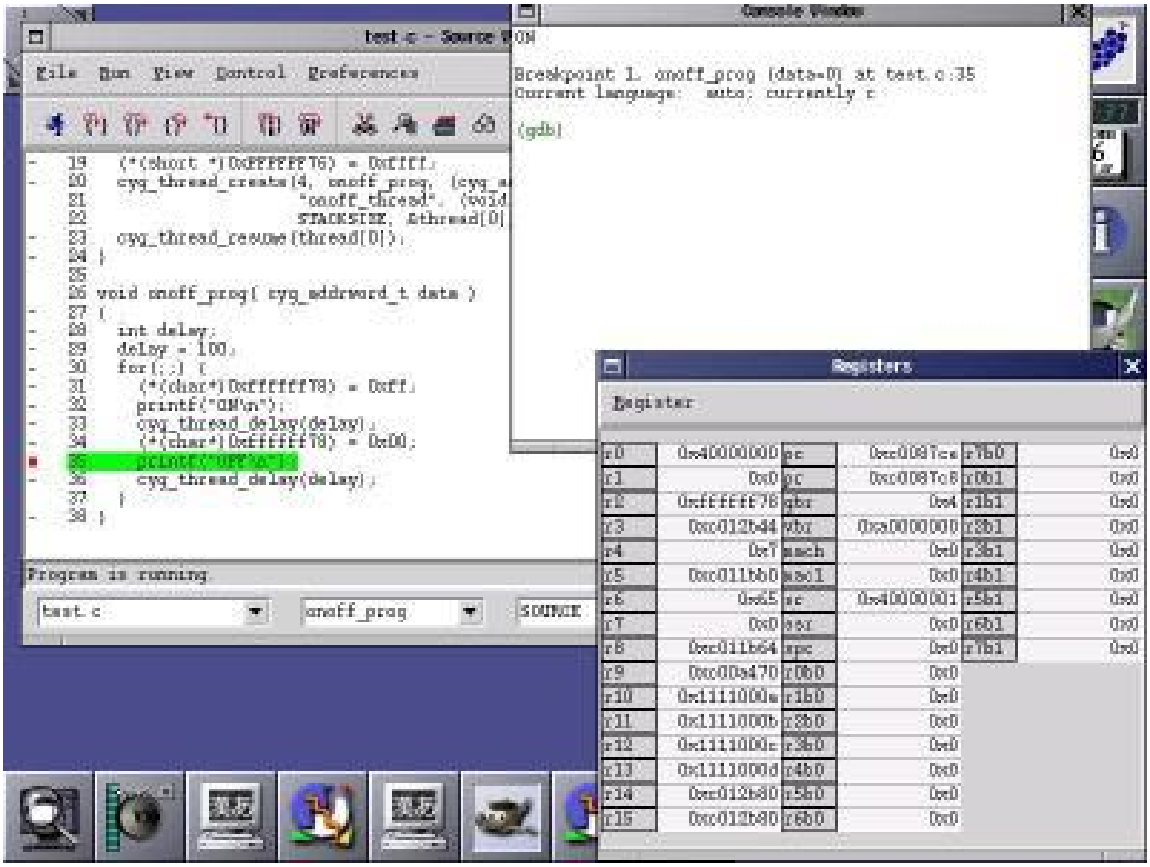

図 1.5: eCos でのリモートデバッグ中の画面

デジタル制御

H 君 「ところで、『処理を急ぐタスク』って、具体的に、どんなものがあるんだろう。

あっ、いや、怒らないで ····

観念的には判るんだけど、具体的なイメージが湧かないんだよ。」

M 子 「別に、私だって、年がら年中、怒っている訳じゃ無いわ。

そうね、『処理を急ぐタスク』と言えば、通信処理とか、画像取得とか、色々あるけど、やっぱり、イメージ的に判りやすい例は、デジタル制御かしら。

じゃ、『歩行』を例に取って、『リアルタイム OS を用いた組込み型コンピュータを使ったデジタル制御』を説明しようかしら。」

H 君 「人間の『歩行』って、デジタル制御なのかな ？ 」

M 子 「私は、生科学者じゃかいから、本物の人間の歩行制御がどう言ったプロセスで行われているかなんて、知らないわ。最前線の科学者だって、まだ、厳密な解を得てないのじゃないかしら。

私は単に『歩行』を『コンピュータでデジタル制御』するとしたら、どうかって、例を説明するだけ。人間と言うより、ロボットの歩行の制御ね。

あんまり、判り切ったところで、チャチャを入れないで！」

H 君「わかったよ。」

A 博士「一般的に『デジタル制御』と呼ばれているものは、次の二つを合わせた制御である場合が多いな。

- 量子化制御 (狭義のデジタル制御)
- 離散時間システム制御

1 つ目の『狭義のデジタル制御』は、判りやすい。制御の対象となる状態、歩行の例で言えば、身体の傾きや歩く速度等は、アナログ的な連続量を取るのが普通だ。だが、このアナログ量を、8 ビットなら 256 段階に、 12 ビットなら 4096 段階に分けて数値化し、処理を行う。これが、デジタル化であり量子化だ。」

H 君「それなら、判ります。一般常識ですね。」

M 子「次の『離散時間システム制御』は、一般的な常識じゃ無いわよ。

あなたも、コンピュータを使ったり、プログラムをした経験があるのなら、コンピュータの処理は、時間的に連続して行える訳じゃ無いことは理解しているわね。

単純な入力を出力するだけのルーチンだって、ループを回るのに必要な時間だけは時間的に離れているわ。これが、離散時間システムよ。

良く、デジタル・オーディオ等で、サンプリング周波数と言うのがあるけど、これが離散時間を表しているものなの。例えば、サンプリング周波数が 44.1 KHz なら、$\frac{1}{44100}$ 秒ずつの細かい時間ステップで処理を行っている訳ね。

正直言って、歩行のために処理の量は判らないけれど、多分、一秒間に 20 回から 100 回と言ったところかしら。つまり、時間ステップは、 10 ms から 50 ms 程度と言ったところね。」

H 君「なんとなく、判って来たよ。

つまり、その時間ステップと言う $\frac{1}{100}$ 秒から $\frac{1}{20}$ 秒と言う高速で処理をし続けるのが、デジタル制御なんだね。

そして、各処理は、 $\frac{1}{100}$ 秒から $\frac{1}{20}$ 秒 と言う時間内に終える必要があるわけで、その為にリアルタイム OS が有効なんだ。」

A 博士「そうだよ。その通りだよ、 H 君。」

M 子「漠然とした概念は、その通りだけれど、本当にちゃんと理解しているのかしら。

あなたのイメージするデジタル制御と言うのは、どう言ったものかしら？ ちょっと、図に書いてみてくれる？ 」

H 君「こんなものかな ?」

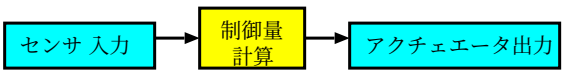

図 1.6: H 君のデジタル制御のイメージ その 1

M 子 「これじゃ、離散時間との関係が良く判らないわ。ちゃんと、処理時間がイメージできる図に書き換えて。」

H 君 「これで、どう ?

これなら、各処理が、いかに処理時間にリアルタイム性が必要か、判るね。」

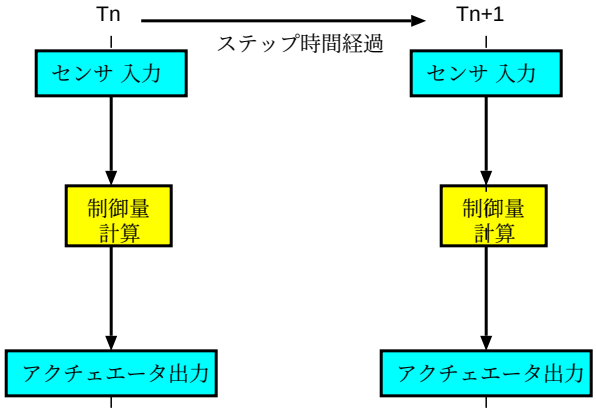

図 1.7: H 君のデジタル制御のイメージ その 2

M 子 「本当に判るの ?

じゃ、図 1.7 における『制御量計算』の処理に許される最大時間って、どのくらいなの?」

H 君 「ステップ時間じゃないのかな ? 」

M 子 「違うわ。

あなたの図の通りなら、 Tn の時間にセンサーから得たデータから計算した制御量は、全く同時刻にアクチュエータに出力される。

つまり、処理に許される時間は零。処理速度は無限大を要求しているのよ。」

H 君 「えっ。いや、多少は遅れても良いんだよ。

だから、処理に対する要求は『できるだけ速く』ってところかな ? 」

M 子 「もう一つ聞くけど、ステップ時間はどうやって決めるの ? 」

H 君 「えっ ?? ステップ時間は、短ければ短い程、良いんだよね。結局、 CPU の処理能力に依存するのかな ?

だから、 CPU の処理能力の許す限り『できるだけ短く』って事かな ? 」

M 子 「ナンセンス !!

『できるだけ速く』とか『できるだけ短く』なんて、仕様でも要求でも設計でも、何でも無いわ !

そう言うのは、『我がまま』って言うのよ。」

H 君 「そんな事、言ったって ···· 」

A 博士　「まあ、 H 君の場合は、今日初めて、離散時間システムを知ったわけだし、その程度は仕方が無いことだろう。

問題なのは、現実に制御設計しているような人の中にも、離散時間システムに対して、同じような勘違いをしている人とか、機器の要求仕様とか設計仕様に『できるだけ速く』的な記述や、実現不可能な程の高い要求を平気で出すような人がいることだ。

それでは、 M 子 君、デジタル制御のイメージ図を書き直してみてごらん。」

M 子　「それじゃ、書き直すわよ …　こんなものかしら。

先に言っとくけれど、これは、あくまでも一例で、他にも色々と制御の方法はあるからね。」

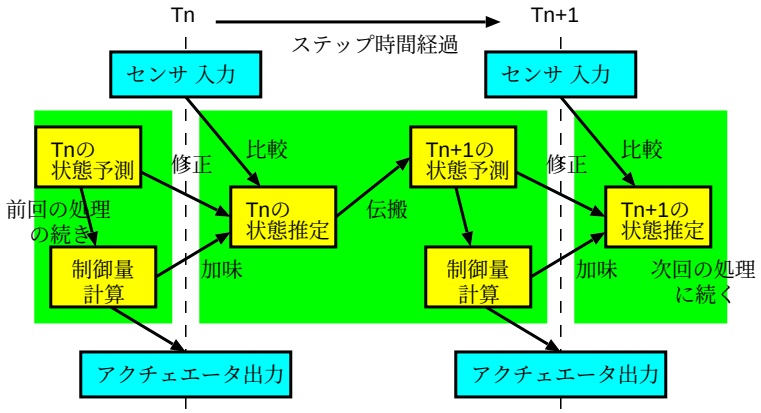

図 1.8: デジタル制御

H 君　「僕のより、ずいぶん、複雑だね。」

M 子　「一番重要な事は、矢印が全て、時間軸に対して、逆向も垂直にもなっていないことよ。

Tn にセンサーから得た情報は、 Tn+1 の時のアクチュエータの制御出力に使われるの。

処理に許される時間制限は、ステップ時間だと言うことが、この図から一目瞭然ね。」

H 君　「 Tn+1 の状態を予測するんだ！

これが、噂に聞いた『予測制御』って奴なのかい ?」

M 子　「『予測制御』って程、大層なものじゃないわ。

ごく短い時間なら、ものの動きの予測は可能だわ。ここでの『状態』は、さっきも言ったように身体の傾き等だから、物理法則から予測は可能ね。

Tn の時点での正確な『状態』が判っていれば、 Tn+1 の時点の『状態』は予測できるわけ。」

H 君　「でも、予測不可能な事もあるだろう。例えば、『石につまずいたとか』さ。」

M 子　「身体には、それなりの大きさと重さがあるわ。そのために慣性があるから、予想外の事があっても、短時間なら大きく変化することは無いわ。

逆に言えば、予想ができる範囲が、時間ステップの上限になるわけ。

これは、制御の対象によって左右されるわ。

制御対象の慣性が大きければ予想できる時間は長くなるし、慣性が小さければ予想できる時間は短くなるの。このような制御対象の慣性の事を『時定数』と呼ぶの。

時定数の大きい制御対象の時定数が大きさによって、時間ステップの上限が決まるのよ。」

H 君 「なるほどね。もう一つ、気になることがあるんだけど。

『 Tn の状態推定』ってあるけど、これは何 ?

一つ前のところの『状態予測』と『制御量』と、センサからの取得データから、『推定』しているようだけど、そんな面倒な事せずに、センサで、その時点の状態を測定してしまえば良いと思うのだけど。」

M 子 「理由は、二つあるわ。

- 全ての状態を測定するだけのセンサが無い
- センサ自体に測定誤差がある

例えば、人間の三半規管なんか考えた場合、あのセンサだけで、一瞬にして、身体の傾きとか動きとかの全ての『状態』を正確に測定できると思う ?

できるわけ無いわね。

でも、一瞬一瞬の測定値が部分的なもので、その上、誤差を含んでいても、ある程度の間、継続してデータを取り続ければ、必要な『状態』を十分な精度で推し量ることができるわ。これが、『 Tn の状態推定』なの。」

H 君 「大体、判って来たよ。

つまり、制御対象によって、時定数が決まる。時定数で制御対象の予想できる時間が決まるから、時間ステップも決まる。

決められた時間ステップの中で、『センサーからのデータ取得』『状態の推定』『状態の予測』『制御量の計算』や『アクチュエータへの出力』と言った処理が必要となる。

この時間的な制約を守るために、リアルタイム OS が役に立つんだ。」

M 子 「気が早いわね。

今、あなたが理解していると言った『処理』の中に時間的制約が異なる処理が混在しているわ。

図 1.8 の緑の部分の『状態の推定』『状態の予測』『制御量の計算』の処理の時間的制約は、確かに時間ステップが上限だわ。

でも、『センサーからのデータ取得』と『アクチュエータへの出力』は、極めて速い処理が必要よ。『センサーやアクチュエータへの入出力』に時間ステップに値する程のずれがあってはならないし、また、処理に必要な時間は、そんなに大きくないわ。だって、センサから取ったデータをバッファに記録して、予め用意しておいた制御量を出力するだけだもの。

だから、時間ステップが、数十 ms 程度の時で、『状態の推定』『状態の予測』『制御量の計算』の処理の時間的制約も同じ程度の時も、『センサーやアクチュエータへの入出力』に対する時間制約は数百 μ s と言った速度が必要になるの。

こう言った時間的制約が、桁違いに異なる処理を一つのタスクにしては、うまくリアルタイム性が保証できないわ。

だから、超高速の『入出力』タスクと、それ以外の『処理』タスクの二つに分ける必要があるの。」

A 博士 「M 子 君の『入出力』タスクは、『デバイス・ドライバ』として、プログラムされる時も多いね。」

リアルタイム OS を用いたデジタル制御

M 子「今さら、言う必要も無いでしょうけど、 CPU が一つのコンピュータは、本当は一つの処理だけしか、行うことができ無いわ。だから、マルチタスクと言っても、実際は、時間で分割しているだけ。

リアルタイム OS は、図 1.9 のように、整理したタスクを時間軸上に配置する事で、マルチタスクを実現することができるの。」

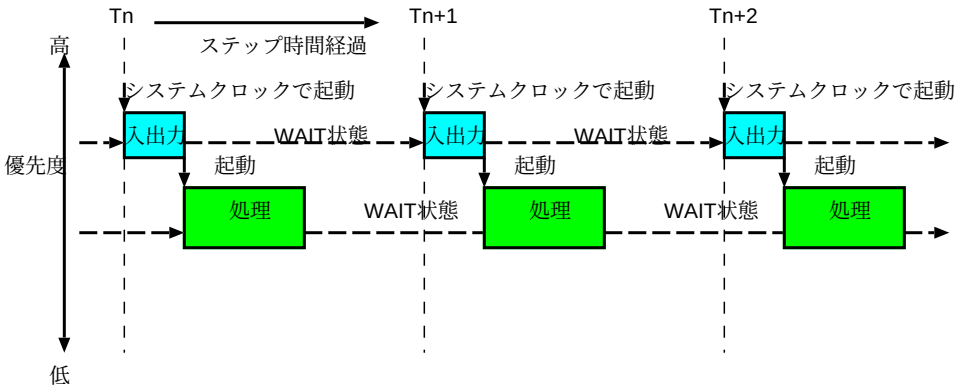

図 1.9: リアルタイム OS によるデジタル制御 その 1

M 子「図の中で、『システムクロックによる起動』とかタスク間で『起動』信号を送る事や、 WAIT 状態から実行状態への移行と言った処理は全て、リアルタイム OS が持っている機能よ。」

H 君「なるほど、この図を見れば、リアルタイム OS の持つ特徴が理解できるね。」

M 子「こんなの序の口だわ。たった一つの対象しか、制御していないから、リアルタイム OS なんて使わなくても、制御できるもの。

実際には、制御対象が複数あることも多いわ。そして、制御対象によって、時定数が異り、時間ステップがまちまちな場合も多いわね。

そうなって、初めて、リアルタイム OS の本領が発揮できるのよ。

次の図を見て。」

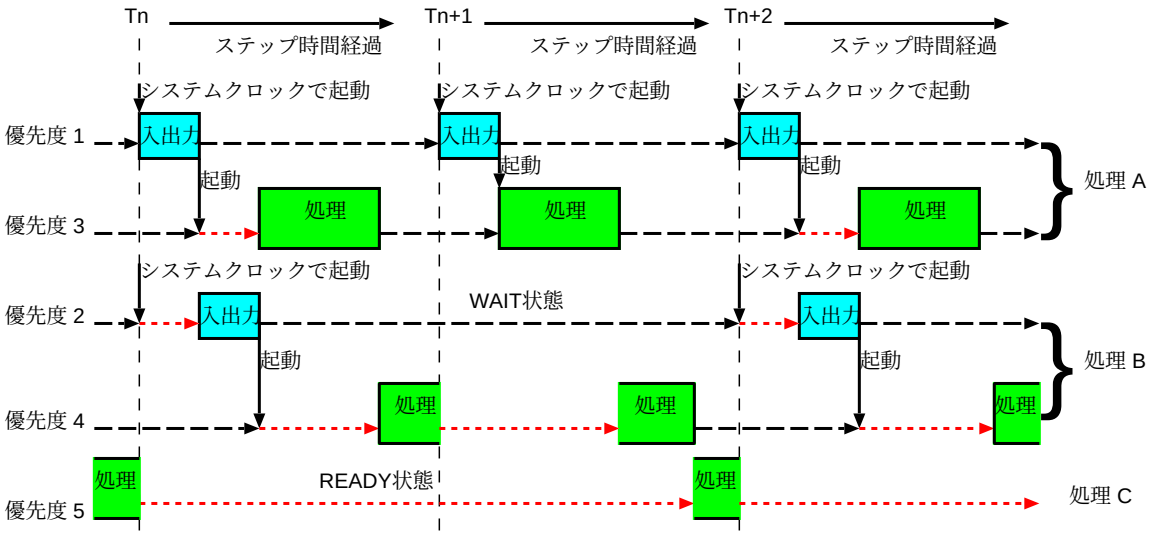

図 1.10: リアルタイム OS によるデジタル制御 その 2

M 子 「この図は、二つの対象を同時に制御している例よ。制御 A は、時定数が小さく制御の時間ステップも短いわ。制御 B は、逆に時定数が大きくて、制御の時間ステップも長い。

この二つの制御を同時に行いつつ、全く別の処理 C と言うタスクも同時に処理しているの。

赤い破線で書いた『READY 状態』とは、本当はタスクは実行したいのに、より、優先度の高いタスクが走っているので、それが終るまで待っている状態なの。」

H 君 「わあ、複雑だなあ。」

M 子 「もっともっと、制御する対象が増えても、リアルタイム OS は、必要な処理に実行時間を割り振るわ。

そこで初めて、リアルタイム OS の有効性が出て来るのね。」

A 博士 「M 子 君は、デジタル制御を主眼にリアルタイム OS の特性を説明したね。当然のことながら、時間的にクリティカルな処理を必要とするのは、デジタル制御だけではなく、通信処理等、多種多彩なものがある。

その全てに、リアルタイム OS が有効な事は言うまでも無いだろう。」

1.5 ステップ 3 『 UNIX 互換 OS 』

NetBSD

H 君 「もう一つの UNIX 互換 OS と言うのはどうなんですか ?」

A 博士 「NetBSD は、フリーの UNIX としては、 Linux と FreeBSD に次いで使われている OS だ。一般的には、普及活動の盛んな Linux や FreeBSD に比べると、普及されていない感があるが、それは、あくまでもインテルの CPU を使ったパーソナル・コンピュータでの事で、多種の CPU をサポートしている NetBSD は、色々な コンピュータで使われている。

UNIX 互換の OS の特徴は、

- メモリの保護管理
- 豊富なアプリケーション・ソフトウェア

だな。片や、リアルタイム性は乏しい。

Fox の場合、NetBSD/sh3 を実装して、コンパクト・フラッシュをハードディスク代わりに使えるから、大容量のデータを扱えるぞ。

ただし、ホストコンピュータとして、やはり、 NetBSD の乗ったコンピュータが必要になるが。」

H 君 「メモリの保護管理と言うのはどういう意味ですか？」

A 博士 「 OS を使わない状態や、リアルタイム OS では、メモリは一続きのフラットな状態で使われている。だから、マルチ・タスクで並行処理しているときでも、タスク間でメモリが自由に読み書きできてしまう。

だが、 UNIX 互換 OS では、CPU の MMU[7] 機能を使って、タスク毎に別のメモリ空間が割り当てられるので、メモリの干渉は起こりえないのだ。」

H 君 「メモリが、別空間にあるって ?」

A 博士 「タスクであるアプリケーション・プログラムから、メモリをアクセスするとき、 MMU でアドレス変換する。つまり、実際のハードウェアとして存在するメモリのアドレスとは無関係のアドレスにあるメモリを使う事ができる。

だから、複数のアプリケーション・プログラムを同じメモリのアドレスに入れ込む事も可能だ。

アプリケーション・プログラム自体は、それぞれが自分にメモリが割り当てられているように使えるのだが、実際には、別々のメモリに分かれている。それぞれのアプリケーションは、どんなアドレスをアクセスしても、自分に割り当てられたメモリにしかアクセスできない。

だから、他のアプリケーションのメモリを侵すこともない。

これがメモリ保護だ。」

H 君 「アプリケーション・プログラム同士で通信を行う時などはどうします？」

A 博士 「OS の本体であるカーネルは、全てのメモリ領域にアクセスできる。アプリケーション同士が連絡を取り合うときなどは、一度、カーネルを通して他のカーネルに連絡を送る。

このようにカーネルとのインターフェースを取る処理が、API アプリケーション・インターフェース だ。アプリケーションに取って、カーネルや他のアプリケーションとのインターフェースを取る方法は、API だけに限られている。」

H 君 「結構、面倒ですね。メモリ保護って、何かメリットあるのですか？」

M 子 「馬鹿ね。

アプリケーションに取ってのメモリ・アドレスが、実際のメモリと無関係と言うことは、実際に存在しないメモリでも使えるって事よ。つまり、仮想メモリね。

また、アプリケーションに取って、唯一のインターフェースが、 API だって事は、それだけ気にしてプログラムを組めば良いって事！

プログラムが大きくなったり、処理が複雑になったり、アプリケーション同士の関係が複雑になれば、こう言った機能が必要でしょ！

また、インターネット上にアプリケーション・ソフトウェアが豊富にある理由も API だけ気にすれば、プログラムが作れることが理由ね。」

[7] メモリ・マネジメント・ユニット

H 君 「プログラムが大きく複雑になったからって、自分でメモリくらい管理できると思うんだがなあ。

それに、豊富なアプリケーションったて、僕のロボットで、ワープロから表計算ソフトを動かそうなんて思ってないしね。」

M 子 「誰も、あなたのロボットで X-Window を動かして、XEmacs して、キーバインドをカスタマイズしなさいなんて、言わないわよ。[8]

あなたが、自分でメモリを管理できる範囲でのプログラムの大きさとか複雑さって言うのは、私が言っている大きさとか複雑さとはレベルが違うのよ。

どうせ、今まで使っていた H8 の ROM 容量の 128 K バイトと RAM の 4K バイトが限界でしょ。無意識の内に、自分で限界を作っちゃっているから、いつまで経っても飛躍がないのよ。

Fox で、 NetBSD を使えば、主メモリの 8M だけじゃなくて、仮想メモリも使えればアプリケーション・プログラムで使えるメモリは無限大よ。[9]

コンパクト・フラッシュをハードディスク代わりに使えるのなら、 100M 200M のデータだって扱えるわ。」

A 博士 「コンパクト・フラッシュのコネクタに接続できるハードディスクなら最大 1G の容量があるぞ。[10]」

H 君 「 1G のデータなんて、僕一人のプログラムで使い切れる訳無いじゃないですか。」

M 子 「馬鹿ね、プログラムにしろ、データにしろ、一人だけで作ろうなんてと思わないの。

大量のデータを扱うのなら、データベース・アプリケーションを使えば良いでしょ。インターネットを探せば、フリーのソフトが一杯あるわ。」

H 君 「また、『豊富なアプリケーション・ソフト』って、話が出たね。でも、僕のロボットにはディスプレーもキーボードも無いんだから、どんなに豊富にアプリケーションが有っても使いようが無いと思うんだけど。」

M 子 「どうやって、説明しようかしら、このタコには。

そうね。例えば、あなたのロボットにチェスをさせるようにできないかしら。

ディスプレー上にチェスの盤面を表示するのではなくて、本当のチェスの盤と駒を使って、人間相手にチェスを指すのよ。

もちろん、画像認識で、現在の盤面を判断し、自分の手を考えて、マニピュレーターで、駒を動かすの。」

H 君 「無茶苦茶言うなあ。

そんな、国立研究所や大企業じゃなきゃできないような研究テーマ、僕にできるはず無いじゃないか。」

M 子 「本当にそうかしら。

例えば、 Fox を使って、現在の盤面を、画像認識して、判断することは可能かしら。」

A 博士 「CMOS カメラから、 Fox に画像を取り込む事ができるぞ。[11]」

---

[8] 実際、どうして、そう言った UNIX の紹介記事が多いんだろう ?

[9] 現状の Fox 用 NetBSD/sh3-1.5 では、バグがあって、まだ、仮想メモリのハードディスクへのスワップは行えない。

[10] IBM 製のマイクロ・ドライブに関しては、まだ、動作確認していない。ドキュメントを見る限りは OK だと思うのだが。

[11] CMOS カメラを、 Fox に接続する方法については、現在、開発中

H 君「画像が取り込めるのなら盤面上の駒が、白か黒か位の判断程度なら可能かな。

でも、同じ白なら白の駒でも、それがクイーンか、キングかの区別が付く程の画像認識プログラムを作るのは無理だよ。」

M 子「あなたが、盤面上の駒が白か黒かを判断するプログラムが作れるのなら、私は、直前の盤面と比較して差分を出すプログラムを作るわ。差が有るところが、駒が動いたところね。

最初の盤面は決まった物だし、駒の動きもルールで決まっているから、盤面をずっと追って、動いた駒をトレースして行けば、どの駒がクイーンか、キングかって事も推定可能よ。」

H 君「なるほど。でも、僕のプログラムから、君のプログラムにどうやって、情報を渡す？

面倒な手続きが必要じゃないかな。」

M 子「標準出力に文字列でアウトプットすれば良いじゃない。最初のますが白い駒で、次ぎは黒 … って。」

H 君「標準出力に文字列って？

まさか、例の printf を使うんじゃ無いだろうね。」

M 子「そうよ。

printf の何処が悪いの？」

H 君「だから、さっきから何度も言っている様に、僕のロボットにはディスプレーもキーボードも付ける予定は無いんだよ。

ディスプレーも何にも無いところに、文字列をプリントしたって、無意味だろう。」

M 子「 UNIX の世界では、標準出力は、標準出力なの。

その先に、 CRT ディスプレイがあろうが、 LCD だろうが、 X 上の kterm だろうが、はたまた、 telnet で接続されたインターネット空間だろうが、関係ないのよ。

あなたが、標準出力で出した、チェス盤の情報は、リダイレクトして、私のプログラムに取り込むわ。この程度のプログラムなら、 Perl で十分ね。盤面の比較して差分を取り出すのは、 diff を使えば良いし。駒の動きをトレースするところだけが、面倒だけど、プログラム自体は簡単よ。

ところで、駒を動かすマジックハンドと言うか、マニピュレータは作れるかしら。」

H 君「動かすべき駒と動かす場所さえ指定してくれれば、後は機械やハードの問題だから、不可能では無いと思う。簡単ではないけど。」

M 子「そう、これで大体の部分は、できたわね。」

H 君「肝心のチェスを指す思考部分が、まだだよ。

こう言ったプログラムを作ると、それだけで何年もかかりそうだよ。」

M 子「鈍いわね。

世界中のインターネット上に、 UNIX で動く、対戦型のチェス・プログラムはゴロゴロしているわ。

だから、フリーでオープンソースのものを持って来て、乗せちゃえば良いの。

誰が作った チェスプログラム でも、CUI の物なら、標準入出力を使って居るはずだから、リダイレクトして、私のプログラムから、指し手とかを出し入れすれば良いでしょ。

GUI の物でも、クライアント・サーバー形式の物なら、思考部と表示部が分かれているから、思考部だけを使えば良いわけ。それも無理なら、ソースから手を入れなくちゃ行け無いけど、まず、そこまでする必要は無いわ。」

H 君 「なるほど、できそうだね。

うん、さっきまで、実際にチェス盤と駒を使って、対戦試合のできるロボットなんて、できっこないと思っていたけど、できそうな気がして来た。

いや、そんなに簡単では無い筈なんだが、不可能じゃ無いような気がする。」

M 子 「それが、飛躍ってもんよ。

小さなメモリとか OS 無しのプログラミングに縛られて居たら、そんな飛躍はできないわ。

要は、さっき、博士が言っていた API よ。標準入出力もクライアント・サーバー方式での通信方式も、みんな API で規定されているの。だから、それだけ判って居れば、プログラム同士のインターフェースは簡単なのよ。

インターネットで使われている TCP/IP も API の一つよ。TCP/IP で、インターネットに接続できたら、それこそ、世界中のコンピュータとインターフェースが取れて、自由に通信ができるわ。」

A 博士 「今の例は、 OS によって、飛躍的な事が可能であると言う事を示すと共に、複数人によるプログラムの開発を示唆しているね。

ハードウェアの苦手な M 子 君だけだったら、チェス・ロボットは作れないし、 H 君だけでも不可能だ。

それが、 OS の API を用いることで、分業してプログラムを作ることができるようになる。API さえ知っていれば、ハードウェアの詳細とか、アルゴリズムの中身を知らなくても、プログラム同士でインターフェースが取れるからね。

だから、ソフトウェアの強い M 子 君と、ハードウェアの強い H 君が、協力して、各々一人ではとてもできないような事ができるようになる。

今のチェス・ロボットには、M 子 君と H 君だけでは無く、もう一人の開発者が居たよね。フリーの対戦チェス・プログラムの作者さ。

そのプログラマーは、チェス・ロボットの詳細どころか、その存在も、 Fox の事も何にも知らなくったって、開発に加わっているんだ。

一人でできないことでも、何人か集まれば、解決できることも多い。その為には、 OS は不可欠だと思うな。」

1.6 ステップ 4 『ハードウェアの拡張』

FPGA

A 博士 「おはよう、 H 君、 M 子 君。

おや、 H 君、何をしているのかね。」

H 君 「 Fox に、入出力ポートを追加しているんですよ、もともとの 8 ビットだけじゃ、足りませんから。

それで、こうやって、74HC244 とか 74HC373 とかを使って、入出力ポートにしているのです。」

A 博士 「 74HC シリーズは、 3.3V でも使えるからな。

でも、いまどき、ディスクリートで、デジタル回路を組むのも考え物だな。」

M 子 「そうよ、アナクロなの。」

H 君「うわあ、また。僕、何回、アナクロって言われたかな？

今回は、ハードウェアの話題だよ。ソフトウェアの時はともかく、ハードウェアについては余り言われたくないな。」

A 博士「まあ、ともかく、話を戻すと、君が作っているのは、単純なパラレル・ポートの様だが、それで良いのかね。」

H 君「そうですよ。モーターの ON/OFF の制御と、センサの ON/OFF 確認するだけですから。」

A 博士「その程度の回路規模なら、ディスクリートで組んでも良いかも知れないな。

でも、将来的には、 モーターの制御を PWM にしたり、回転数をカウントしたり、と回路が大規模に成って行くんだろう？ 」

H 君「そうですね。

今から、カウンター回路とかタイマー回路をどうしようか、考えているところです。」

A 博士「それなら、やはり、ディスクリートで回路を組むべきではないな。

FPGA とか CPLD を使うべきだろう。」

H 君「 FPGA って、何ですか？ 」

A 博士「自分で、カスタム・メイドできるゲートアレイだよ。」

H 君「ゲートアレイですって。

開発には、何千万もかかるって、聞いたことがありますよ。」

A 博士「それは、製造工程で、回路を書き込むタイプの物だろう。

私が言っているのは、 ROM と同じように、 製造後にユーザーが自由に書き換えられるタイプの物だ。

昔は、 FPGA 自体も、開発環境も高かったが、今は安くなったぞ。

Fox の ROM ライタにも、 FPGA を使っている。ROM ライタに使っている XC95108 は、秋葉原で、3000 円以下で手に入るものだ。ライタに必要な物は、 Windows パソコンと、プリンターケーブル、それに 74HC125 2 つで自作できるものだ。また、EEPROM タイプだから、何度でも簡単に書き換えができる。

それに開発環境は、インターネットから、無料でダウンロードできる。数年前まで、それこそ、何百万もしたものだ。」

H 君「 FPGA が、 3000 円って、ゲート IC なら、数十円で買えるのに、ちっとも安くないですよ。」

A 博士「 FPGA は、ゲート IC の 数百個分の働きをするから、高くは無いよ。

その上、細かい配線が必要ではないから、大規模回路を組むときには、圧倒的に有利だ。」

H 君「 FPGA の中身の回路って、どうやって作るんですか？ 」

A 博士「無料でダウンロードできる開発環境の中にも、論理回路入力で設計できる CAD も入っているから、これを使うかな。君なら、論理記号が判るから、ちょうど良いかも。」

M 子「博士、お言葉ですが、ゲートアレイの入門者に論理回路入力から教えるのは、今や罪悪ではないでしょうか？ 」

A 博士 「うーーん、言うね。

私も、どうしようか、迷っていたんだが、やっぱり、いまさら、論理回路や論理記号の時代でも無いか。」

H 君 「どういう意味です。論理回路とか論理記号が時代遅れなのですか?」

M 子 「そうよ。アナクロなの。

小規模な回路の設計ならともかく、ある程度大規模のデジタル回路設計なら、ハードウェア記述言語 HDL を使うべきなのよ。」

H 君 「君はてっきり、ハードウェアは苦手だと思っていたけど。」

M 子 「 HDL は別よ。ちゃんと抽象化されているもの。

回路図って、実際の配線と対応しているわよね。

現実の配線通りの実体配線図から始まって、回路記号を使っても、内容的には 1 対 1 に対応していた。論理記号を使った回路で多少は抽象化されたけれども、やはり、現実の回路と関係があるわね。

でも、ゲートアレイの内部回路は、論理記号を使った回路とはまるで違った物に成るわ。たとえ、機能的には同じでも。

だから、ディスクリートの時代なら、ともかく、ゲートアレイを使うようになったら、本当は内部回路と異なるのに、論理記号を使って、回路設計することは無意味ね。」

H 君 「ゲートアレイの中身の回路って、論理記号を使った回路図と違うのかい。」

M 子 「そうよ。でも、ちゃんと設計された通りの機能を持つ様にコンパイラが処理するから大丈夫。

ゲートアレイの内部で、実際は、どのように回路が実現されているかは、ゲートアレイのメーカーや種類で異なるわ。でも、そのことをユーザーは気にしなくても良い仕組みに成っているの。

つまり、 CPU が、Pentium であっても、 PowerPC であっても、 SH-3 であっても、 C で書いたプログラムなら、コンパイルしてどれでも使えるのとおなじことね。

だから、元々、現実の電気配線を元にした回路図と言う概念そのものの、意味が薄れるわけ。

それなら、むしろ、回路に必要とされる機能を抽象的に記述するだけに特化した方が、回路設計に良いことに成るわ。

それが、 HDL よ。」

A 博士 「確かに、その通りなんだが、 H 君の様に既に論理記号による回路設計をマスターした者には、ゲートアレイの設計も、回路図入力の方が入り易いかと思ったんだが。」

M 子 「でも、いずれは、論理記号による同期回路の設計手法とかで苦しまなきゃ行けないわ。

私に言わせれば、同期回路も非同期回路も、ごちゃごちゃになる論理記号による回路設計なんて、抽象化が中途半端なだけのものに過ぎないわ。だから、ゲートアレイの入門者に論理回路入力から教えるのは罪悪だと思うわけ。

その点、最初から、 HDL で入門しておけば、抽象的な設計概念が、自然と覚えられるから、良いわけ。」

H 君 「もう、二人だけで話を進めて …

そもそも、 HDL ってどんなものなの ?」

M 子「そうね。例えば、 10 ビットのアップダウンカウンターを HDL で記述したら、こうなるわ。

```
signal D: std_logic_vector (9 downto 0);

begin
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
            if (UPDOWN = '0' ) then
                D <= D - 1;
            else
                D <= D + ;
            endif;
        endif;
    end process;
end behavioral;
```

これと同じものを、論理記号で作ったら大変な規模になるでしょ。それも同期回路で。」

H 君「本当に、たった、これだけでアップダウンカウンターなるのかい。
ぐうのねも出ないな。」

M 子「そうよ。実際は、多少のヘッダ読み込みや、入出力ピンの定義が加わるけど、機能の記述部分は上の通りよ。

それに、ディスクリートなら、回路を作る手間も大変でしょう。

その点、 FPGA なら、回路書き込みに一分もかからない。もし、回路がうまく動かなくても、書き換え可能だから、作業の効率は、ものすごく向上するわ。

その上、 FPGA なら、処理は、猛烈に高速だし、並列処理だし、。

例えば、

```
begin
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
            処理 1 ;
            処理 2 ;
            処理 3 ;
        endif;
    end process;
end behavioral;
```

なんて、プログラムなら、上の処理 1、処理 2、処理 3 は本当に同時に行われるのよ。

OS の様に、本当は、シーケンシャルに実行されているなんて事は無し。

クロックにもよるけど、処理 1、処理 2、処理 3 が、同時に 数十 ns 以内に実行できるわけ。

HDL は究極の並列処理プログラムじゃないかしら。ノイマン・ボトルネックも無いし。」

H 君 「FPGA で何でもプログラムできるのなら、 CPU なんて要らなくなるのかな。」

M 子 「現実には、 FPGA は、まだまだ、容量不足で、何でもできるとまでは行かないわ。

だから、 複雑な処理は CPU で実行し、超高速が必要な処理は FPGA と言う切り分けが行われるわ。

でも、PFGA の容量が、遥かに大きくなって、全ての処理が、ナノセック・オーダーで並行に実行できる時代も遠くないかもよ。」

A 博士 「まあ、そうなった未来の為にも、今から、 HDL を勉強しておいて、損は無いだろうね。」

ライントレース・ロボット

A 博士 「ところで、 H 君、元々、君の作ろうとしているロボットって、どう言ったものだい ?」

H 君 「ライントレース・ロボットです。

床に黒い線を書いて、それを辿って走るロボットです。

本格的なものは、それなりに大変なんで、最初は、市販の模型用モーターとギアボックスを使った簡単な奴を試作しようと思ったんですよ。」

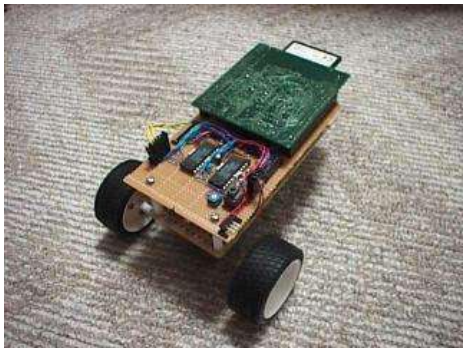

図 1.11: ライントレース・ロボット

M 子 「ライントレース・ロボット !?

こんなのに、 RISC CPU も リアルタイム OS も UNIX 互換 OS も FPGA も必要無いんじゃ無いかしら。」

H 君 「何だよ、いまさら。

だから、最初から、 PIC か H8 を使おうって言ってたんだ。」

A 博士 「いやいや、確かに、ライントレース・ロボットに Fox は、強力過ぎるかも知れんが、そう言った常識論に因われていると何時まで経っても発展しない。

それこそ、M 子 君が良く言うように、飛躍が無いだろう。」

M 子「飛躍ねえ。

そうかも知れないわ。

このライントレース・ロボットが飛躍を生む、かも知れないわね。」

1.7　色々な疑問

M 子「おはよう。

あれっ、浮かない顔して、どうしたの。」

H 君「いやね、教えてもらってから、つらつら 考えたんだけれど、色々と疑問が湧いて来てね。」

M 子「どう言った疑問なの？

判る範囲で教えてあげるわ。」

A 博士「私も協力しよう。」

『生産性』と『飛躍』

H 君「君は、良く『トータルとしての生産性』とか『飛躍』と言う言葉を使うよね。

あれ、どう言った意味なんだろう。」

M 子「コンパイラとか OS の使用を説明した時に、そう言った言葉を使ったわ。

コンパイラや OS を使った方が使わないよりも、プログラムを作る生産性が上がり、より先進的な的なプログラムを作れると言う単純な意味よ。『トータル』と断ったのは、コンパイラや OS を使い始める時のインストールや言語仕様や使い方を勉強すると言った初期の作業量を差し引いても、総合的には生産性が上がると言うこと。

どうして、こんな簡単な意味、理解できないの？」

H 君「言葉自体の意味は判ってるよ。そうじゃなくて、僕みたいなアマチュアに『生産性』や『飛躍』の意味があるかって事なんだ。

僕が、プロのプログラマーだったり、エンジニアだったら、当然、『生産性』は必要だし、他の人より一歩先んじると言う意味で『飛躍』が必要な理由も判るよ。

でも、僕はアマチュアで、趣味で物を作ったり、プログラムしているんだ。

だから、『生産性が高い』事も、『飛躍』も重要じゃ無いと思うんだ。」

M 子「例え、あなたが個人的な趣味で、物作りやプログラミングをしていたとしても、たった一人で、できる事は作業量の限界があるわ。

有限な作業量で、より多くを作りプログラムするためには『生産性』が高い方が良いんじゃないかしら。

それに『生産性』が上がれば、時間的に余裕ができて、より『飛躍』した発想ができると思うんだけど。」

H 君「そうかな。

僕の場合、物作りやプログラムと言った行為や過程、そのものが楽しいんだ。

だから、物作りやプログラムに時間がかからない『生産性の高さ』は、むしろ悪い方に働くとすら思えて来るんだよ。」

M 子 「そこまで言われると、『あなたが何をもって楽しいとか、面白い』と言った主観的な好き嫌いに依存する事だから、私が云々する問題では無いわね。

でも、少なくとも私は、個人的な趣味の為の物作りとかプログラミングにこそ、『生産性』は重要だと思うし、それに付随する『飛躍』は更に重要だと思うの。

プロフェッショナルとして、物作りなりプログラムなりするのなら、どんな場合でも最低限、『お金儲け』として理由付けができるわ。

でも、個人的に趣味として物作りなりプログラミングする時には、その行為が『作る』から『創る』に昇華しないと意味を持たないと思うわ。」

H 君 「どうして、そう思えるのかな。」

M 子 「逆に、あなたに聞くけど、あなたが苦労して作った物やプログラムしたものが、他の誰かが、とっくの昔に同じような物を作ったりプログラムをしたりした物だったら、楽しいかしら？

苦労して自分で工夫して作ったり、プログラムした物ほど、できた後で、過去に類似するものが、もっと完成度の高い物で作られていると判ったら悔しいと思わない？

悔しいと思わないのなら、何を言っても無駄かも知れないけど、でも、やっぱり同じ苦労するのなら、世界で唯一の物を創り出す事に労力を使った方が、より面白いと思うのよ。」

H 君 「そりゃ、世界で唯一の物とかを創れりゃ、面白いとは思うけど、僕は、まだ知らない事も多いし、良いアイデアが浮かぶとも思えない。

第一、世界で唯一のアイデアが浮かんだって、それが役に立つものだとは限らないし。」

M 子 「チャレンジをする前から、弱音を吐かないの！そう言うのを負け犬根性って言うのよ。

誰も、最初から、あなたが世界で唯一の素晴らしいアイデアを思い付くなんて思ってないわ。

作っている内に、何か新しいところとか、不便なところを直す工夫とか、そうやって、ユニークなアイデアに結び付くんでしょ。

そうやって、あなたが思い付いたアイデアも必ずしも役に立つものとは限らない。でも、『世界で唯一の役に立たないアイデア』ってのも、それはそれで存在意義のあるものだと思うわ。」

H 君 「そんな存在意義じゃ、有っても無くても同じような気がする。」

M 子 「私が言いたいのは、どうせアイデアを思い付くなら、誰も思い付いていないユニークな物が良いって事よ。

だから、既にコンパイラとか OS で解決されている苦労をするよりも、もっと誰も挑戦していない分野で苦労した方が、良いと思うの。

実際にハードウェアを作って、プログラミングをすると苦労するわ。苦労すれば、必ず、工夫するアイデアが出て来るものよ。

誰も挑戦していない分野の苦労に対するアイデアなら、常に世界唯一の物だし、万一、役立つアイデアなら、それは凄いことよ。」

H 君 「僕が、『役立つアイデア』を思い付く可能性は、『万一』なのかい？」

M 子 「あなたが、世界で唯一の役に立つアイデアを生涯で一つでも思い付いて、それが歴史に残れば、後生の人は、それを『天才』って呼ぶわよ。」

A 博士 「M 子 君の意見は、コンパイラや OS と言った既に解決されている工夫は取り入れて、新たな分野に挑戦するべきだと言うものだね。

私は、必ずしも『コンパイラや OS』と言った分野に挑戦することは無意味だとは思わない。例えば、 Linux の生みの親である Linus Torvalds 氏も、既に UNIX を始めとする OS と言うものがありながら、敢えて、自分で OS を作ると言う試みを行って、新しい OS を作ったよね。

同じように、既存のコンパイラや OS を使わずに、自分で工夫すれば、新たな道を発見できる事もあると思う。

だが、 コンパイラや OS の自作は、極めて難易度の高いものだ。H 君のように、元々の目的が『ロボットの制御』と言った別の目的があるときは、止めておいた方が良いね。どちらも中途半端になってしまうだろう。

コンパイラや OS の自作を目的にするならば、それだけに目的を絞った方が良いだろう。」

『マルチタスク OS 』と『分散並列処理』

H 君 「他にも疑問に思ったことがあるんだ。

前に、コンピュータで処理すべき事項が多いから、マルチタスクを行うために、リアルタイム OS や UNIX 互換 OS を使うと言ったね。」

M 子 「 OS の使う理由は、『マルチタスク』だけじゃ無いけど、確かにそれに類することは言ったわ。」

H 君 「でね、僕は思い付いたんだけれども、処理すべきタスクが多ければ、タスクの数だけ、コンピュータを使えば良いんじゃないかって事。

これなら、一つのコンピュータで、一つの処理しか実行しないから、マルチタスク OS なんて使わなくて良いと思うんだ。

良いアイデアでしょ !? 」

M 子 「あなたのアイデアらしいわね。詰めが甘いと言うか ····

そのアイデアは、半分正解で、半分間違いよ。」

H 君 「どういう意味かい ?」

M 子 「まず、前半の『処理すべきタスクが多ければ、タスクの数だけ、コンピュータを使えば良い』と言うところは、その通りだわ。

もちろん、寸法・電力・コストと言ったリソースとのトレード・オフが必要だけれど、タスクの分だけ、コンピュータの数を増やすと言うのは、単純だけれど、有効な解決方法の一つだと思うわ。」

H 君 「そうだろう。」

M 子 「こう言ったアイデアは、『分散並列処理』と言うべきね。

問題は、あなたのアイデアの後半の部分。『一つのコンピュータで、一つの処理しか実行しないから、マルチタスク OS は不要』と言う部分。

これは、まるっきり、逆だわ。『分散並列処理』にこそ、『マルチタスク OS 』が有効なの。」

H 君 「どうしてさ ?

一つの処理しか、実行しないコンピュータに『マルチタスク OS 』が必要なのかい ?」

M 子 「あなたは、各コンピュータが実行するのは、目的となる処理のたった一つのタスクだと思っているけれど、それに付随する通信タスクを忘れているわ。」

H 君 「確かに通信タスクの事は忘れていたけれど、もう一つタスクが増えるだけだろう。二つ位のタスクなら、『マルチタスク OS 』が無くても何とかならないかな？」

M 子 「各々のコンピュータを『ノード』と呼ばせて貰うけど、各ノード間の通信って、どのくらい複雑になるか、考えた事がある？

通信が単純なのは、ノード数が 2 つまでよ。それなら、 2 ノード間の一つしか通信がないもの。

でも、ノード数が多くなるとノード間通信は爆発的に増えるわ。

1 対 1 の通信を基本とした場合、理想的な通信を確保するには、 n ノードの場合、$\frac{n(n-1)}{2}$ の接続が必要になるわ。

とても理想通りには、ハードウェア的な通信網を確保できないから、ソフトウェアで対処するのが普通だけれども、1 対 1 の通信ならば、少なくとも一つのノードは 2 つ以上の通信に対応していなければならないわ。

例えば、ノードを直線的に並べるわけ。最初と最後のノードは通信する相手は一つだけれども、それ以外は、前後二つのノードが相手になるわね。こう言った接続方法の場合、通信する相手が、隣接していなければ、パケットにして、宛先をヘッダーに入れておくの。

パケットを受け取ったノードは自分宛のパケットなら受け取るけど、違った場合は、次のノードに渡してしまう。こうやって、バケツ・リレーを繰り返して、目的のノードまで、運ぶの。」

H 君 「複雑な処理なんだね。」

M 子 「通信の方法には、他にも 1 対 n とか、 n 対 n とか色々な接続方法があって、ノードの配列も沢山の種類があるわ。

結局、これは『ネットワーク』なのよ。

パケットを通信する手順、つまりプロトコルだけれども、色々と種類があるわ。TCP/IP は、プロトコルでも最も有名な物の一つね。

プロトコルを一から作るのは大変な事よ。そして、そのプロトコルを実行するのも大変な事なの。

だから、『ネットワーク通信』を行うことに『マルチタスク OS 』を使う意味が出て来るわ。

マルチタスク OS なら、

- 既存の実績あるプロトコルを OS レベルでサポート
- 複雑な通信の処理をマルチタスクで実行

と言う 2 点で有利なわけよ。

判った？」

H 君 「判りました。」

『リアルタイム OS のリアルタイム性』って ??

H 君 「まだ、疑問があるんだ。

以前、リアルタイム OS の時に、デジタル制御の例を出して説明したよね。

あの時、制御量の計算をする処理の時間的上限が、時間ステップだって話だと思ったんだけど。」

M 子 「その通りよ。何か、文句あるの？」

H 君「いや、その『時間的制約』を守るために『リアルタイム OS 』の持つ『リアルタイム性』が、有効だって話だと思ったんだ。

だけど、そのリアルタイム性って、 『OS の機能・性能』なのかな、それとも、『制御量を計算するプログラムを作る上での制約』なのかな ?」

M 子「その両方よ。

いいっ ? まず、処理が必要となる時に、タスクを起動しないと処理そのものが始まらないわ。

この『処理を要求された時間内にスタートさせる』と言うリアルタイム性は、 OS がサポートする機能・性能だわ。

でも、肝心の『制御量を計算するタスク』の計算自体が複雑で、とても『時間ステップ』内に終らないような代物じゃ、話にならないわよね。少なくとも、『時間ステップ』内に計算が終了するように工夫しなければならないわ。

このように、リアルタイム性を保つ為には、ユーザーが書くプログラムにもリアルタイム性が要求されるの。」

H 君「リアルタイム OS って、使う為には、ちゃんとリアルタイム性を考えたプログラミングが必要なんだ。

でも、どうやったら、リアルタイム性を保つプログラミングができるんだろう。」

M 子「現実には、 CPU の処理能力とか、他の走っているタスクによっても、プログラムに対するリアルタイム性の要求は代わってくるものなの。

だから、リアルタイム OS のプログラムは、ハードウェアに依存するプログラムにならざるを得ないわね。」

H 君「ずいぶん、 UNIX 風のプログラムとは違うんだねえ。」

『リアルタイム OS 』と『UNIX 互換 OS』の両方の長所を取った OS

H 君「その話で思い出したんだけれども、『リアルタイム OS は、メモリ保護が無い』で、片や『UNIX 互換 OS には、リアルタイム性が無い』って話だったよね。

両方の良いところを取った OS って、無いんだろうか ? 」

M 子「難しい問題ね。

今、あなたが言った『メモリの保護』って言うのは、単なる問題の一部に過ぎないけれども、『リアルタイム OS が目指しているもの』と『UNIX 互換 OS が目指しているもの』が、如何に異っているかを、端的に示しているわね。

リアルタイム OS は『タスクの処理がリアルタイム性を持つこと』を第一優先しているし、UNIX 互換 OS は『アプリケーション・プログラムが汎用的に使えること』を優先していると言えるわね。

『アプリケーション・プログラムが汎用的に使える』為には、『アプリケーションが使うメモリの仮想化』と言う抽象化は必要だし、その為には『メモリ保護』は不可欠だわ。

でも、『メモリ保護』は、タスク切替えの時に、余計なオーバーヘッドを生む事になるから、リアルタイム性としては、マイナス要素になるわ。だから、大多数のリアルタイム OS は、『メモリ保護機能』を持たないの。」

H 君「OS としての目的が違うんだから、機能が違うのは当然か。」

M 子 「そうよ。その上、さっき、話したように、『リアルタイム OS での為のプログラム』と『UNIX 互換 OS の為のプログラム』の間でも、 プログラムの作り方自体が、全く違うわ。

『リアルタイム OS の為のプログラム』は、リアルタイム性を保つ為に、ハードウェアべったりで処理速度を速めなければならない。

片や、『UNIX 互換 OS の為のプログラム』なら、『チェス・ロボットの存在すら知り得ないプログラマーが作ったプログラムでさえ使えるほど、汎用化』されている。全く、ハードウェアを意識しないでプログラムされている事が判るわね。」

H 君 「でも、『汎用化されているプログラム』を『リアルタイム性が必要なシステム』で使う必要性もあると思うんだが。」

M 子 「確かにあなたの言うところにも、一理あると思うわ。

実際に、『リアルタイム性の必要なタスク』と、『汎用化されたプログラム』を分けて、並行的に処理する OS の研究も進められているわ。

RT-Linux とか RT-Mach は、 UNIX にリアルタイム性を付加する為の試みだわ。完全なリアルタイム性が実現された訳じゃないけど興味深い試みだと思うわ。

まだ、 Fox に移植されたわけでは無いけど、今後が楽しみね。」

『オープン・ソース』って ?

H 君 「『チェス・ロボット』の話なんだけど、あれにも疑問があるんだ。」

M 子 「なに ? 何か、技術的に問題でもあるの ??? 」

H 君 「何にも問題は無いよ、少なくとも技術的には。

僕が気にしているのは、法律的と言うか、道義的と言うか、道徳的と言うか、そう言った問題なんだ。」

M 子 「何の事 ? 」

H 君 「つまり、『チェス・ロボット』は、仮の話でも、もし、ああ言った開発が可能なら、『チェス・プログラム』を作った人に、失礼にならないかって事さ。

『チェス・プログラム』を作った人は、『チェス・ロボット』の事を知りもしないわけだよね。想像すら、してないだろう。

元々、『チェス・プログラム』の作者は、プログラムが、 UNIX の走るパソコンとかワークステーション上で、人間相手にゲームする事を想定して作られたに違いないんだ。

それを、僕が『チェス・ロボット』に使って良いんだろうか ?

もっと、悪い表現を使うなら、僕は『他人の褌で相撲を取る』行為を行っていることになるよね。そんな事をして良いんだろうか ? 」

M 子 「そんな下品な表現、レディの前で、しないで ! 」

H 君 「『レディ』だったのかい ? 」

M 子 「レディに決まってるでしょ。何処に目を付けてるの !

とにかく、あなたの心配は、 99 % 、無用のものだわ。」

H 君「そうかな。

今まで、僕が使ったアプリケーション・ソフトウェアって、必ず、使用目的やインストールできるパソコンとか、事細かに規定されていたよ。

だから、規定されてもいない使い方ができるとは思えないんだ。」

M 子「あなたが言っているのは、『商用』のアプリケーション・ソフトウェアでしょ。

私が、言ったのは『フリーウェア』とか『オープン・ソース』ってものよ。

『商用アプリケーション・ソフトウェア』の場合、ユーザーは使用する権利を、お金を出して買うわね。その代償として、プログラムの作り手は、品質保証する必要があるわね。品質保証をするためには、プログラムの作り手は、想定された使用条件の範囲を規定する必要があるわね。」

H 君「なんで、『品質保証』のために、『使用条件の範囲を規定』する必要があるかな？」

M 子「馬鹿ね。

『品質保証』のためには、試験が必要でしょ。プログラムの場合、デバッグと言っても良いわ。試験とか、デバッグをやる為には、想定された使用条件の下で行う必要があるでしょ。

簡単に言えば、『デバッグした使用条件なら OK で、それ以外は駄目』って事ね。」

H 君「判ってしまえば、単純な事だね。」

M 子「商用アプリケーションは、こう言った理由から使用条件が厳しいのよ。もっとも、使用条件の規定は、『品質保証』だけじゃなくて、違法コピー禁止と言った財産権のからみも多いけど。

一方、『フリーウェア』とか『オープン・ソース』と言ったソフトウェアには、そもそも使用するためにお金を払う必要は無いわ。だから、『品質保証』も無いから、使用者責任でプログラムを使う必要があるの。

でも、『品質保証』が無いことは、逆に『使用条件の規定』も無くなるわけね。

作者の想定外の使い方をしても、その結果が良かれ悪しかれ、責任を使用者が取る限りは問題は無いのよ。」

H 君「なるほど。プログラムの作者の想定に有ってようが、いまいが、使用条件の規定そのものが無いんだから、どうやって使っても、自由って事だね。」

M 子「あなたは、いつも気が早すぎるの !!

私が、今言ったのはネガティブな面が無い、つまり、想定外の使用でも、法的な根拠が無いから問題が無いって、だけのこと。

世の中、法律をおかしていなければ、何をしても良いってもんじゃ無いわよね。行為そのものが、道徳的に認められるかって言う問題も付随するでしょ。」

H 君「じゃ、法律的には問題なくても、プログラムの作者の想定以外の使い方は、道徳的な問題が有るって事かい？」

M 子「ノン、ノン。逆、逆。

想定外の使い方は、ネガティブな面よりも、ポジティブな面が強いわ。」

H 君「どう言う意味だい？」

M 子 「プログラムの作者は、むしろユーザーが想定外の使い方をすることを望んでいる可能性が高いって事。

しつこいようだけど、これは、あくまでも『フリーウェア』とか『オープン・ソース』の話であって、『商用アプリケーション』の話では無い事に注意してね。

『フリーウェア』、特に『オープン・ソース』の作者は、自分のプログラムが広く使われる事を望んでいるわ。これは、『使用条件書』にも何処にも書いていないけれど、実際に『フリーウェア』とか『オープン・ソース』のプログラムを作り配布している人に会ってみれば判るけど、彼らは、広くプログラムが使われるなら、想定した使用条件とか環境に束縛される必要は無いと感じているわ。むしろ、想定外の使用例を報告すると喜びすらするわ。」

H 君 「そんなものなのかい？」

M 子 「もちろん、想定外の使用に対して、喜ぶ喜ばないは、主観的な問題だから、『フリーウェア』や『オープン・ソース』の作者によっては、喜ばない人も居るかも知れない。でも、喜ばない人は小数派だと思うし、『フリーウェア』や『オープン・ソース』にする事を決めた段階で、そう言った可能性は、ある程度、覚悟しているはずよ。

大多数の場合、想定外の使用が喜ばれるのだから、そう言った使い方をしたのなら、是非、そのことを報告すると良いわね。

例えば、『チェス・ロボット』を本当に作ったら、そのことをレポートにして、メールなどで送ると喜ばれると思うわ。

プログラムを、作者の想定外の使い方をして悪いかどうか悩むくらいなら、そう言った報告が作者に対する感謝になると思った方が良いわね。

でも、もっと、『フリーウェア』や『オープン・ソース』の作者に報いる方法があるわ。」

H 君 「それは、何なんだい？」

M 子 「それは、あなたが作った『チェス・ロボット』の事を、公表することよ。

できれば、詳細の説明とか、プログラムのソースコードまで付けて、『フリー』で公開するの。

もちろん、作った方法とかプログラムとかの知的所有権は、あなたにあるのだから、誰も公開を強要はできないわ。

でも、そうすることで、『チェス・ロボット』に触発されて、新たな発想のロボットとか、応用例が生まれるかも知れない。

そう言った連鎖を元々の『フリーウェア』や『オープン・ソース』の作者も期待しているのよ。」

H 君 「そんなもんなのかな？」

M 子 「あなた自身が『フリー』とか『オープン』の供給者になるわけでしょ。

考えてもみなさい。

あなたの『チェス・ロボット』に触発された人とか、更に、その触発された人に触発された人 … その連鎖の末に、作られた『ロボット』が、火星とか木星、いえ、太陽系圏外を探査するロボットを作ったなら、それは、あなた自身が作ったと同然とは言えないまでも、少なくとも連鎖の中に、あなたが有効な役割を果している事だけは間違い無いわ。」

H 君 「そうだね。

そうなったら、僕自身も、勝ったも同然だね。」

第2章　Fox 概要

2.1　Fox のコンセプト

Fox の基本コンセプトは「だれもが安く簡単に手作りできるコンピュータ」である。Fox の特徴を以下に示す。

- SH-3 を用いた超小型ワンボードコンピュータ
- 小型軽量，低消費電力，高性能
- 回路図および使用料はフリー (無料)
- メモリ容量：ROM 128K バイト，RAM 8M バイト
- 外部記憶として、コンパクトフラッシュを使用
- 開発環境 (C/C++) もフリー
- リアルタイム OS (eCos) が使用可
- UNIX (NetBSD/sh3) が使用可
- 入手が容易な部品で構成
- 特殊な工具や設備を必要としない配線
- 組み込み状態でも ROM に書き込める ROM ライタ

SH-3 は日立製作所の開発した 32 ビット RISC CPU であり，DRAM コントローラや通信インターフェースなど，コンピュータを作る上で必要な周辺をオンチップで持っている。そのため，水晶とメモリなど，ごく少数の部品を接続するだけでコンピュータシステムを構成できる。

今回、プリント基板を製作した。コンパクトフラッシュを装着した状態で、ポケットティッシュ並の大きさである (写真 1)。以前、クレジット・カード大を目指すと言っていたが、やや大きくなってしまったが、コンパクトフラッシュインターフェースを標準装備する事で、NetBSD/sh3 を使えるようにした事から、多少の大型化は仕方ない事であろう。

Fox は「だれもが安く簡単に手作りできるコンピュータ」をコンセプトにした 32 ビット RISC の高性能 CPU である SH-3 を用いた、組み込み用途を主な目的とした小型の自作コンピュータである。

2.2　フリーの意味 •• 自己責任

Fox は回路図の使用料もプログラム環境の使用も無料である。最後に紹介するプリント基板と部品類の頒布も、実費のみで行う予定である。このように、Fox においてはロイヤリティーは無い。

その代わりに使用時における保証もサポートも存在しない。つまり、使用者の自己責任において使って欲しい。これは、Linux 等のフリーソフトとかオープン・ソフトと同じ考えである。

－ メモ －
Fox は、フリーである。
その代わりに、自己責任である。

2.3 何故 フリーか ?

Fox は、回路図の使用料無料、 ROM ライトプログラムだけでなく、回路 CAD やプログラム開発環境等できる限り、フリーなシステムを目指している。もちろん、実際に Fox を作るとなると、構成する部品の購入などにお金がかかる。

しかし、通常なら高価な C コンパイラ等に余計な金がかからない様にしている。

では、何故、これほどフリーにこだわっているか？

GNU のように大上段に構えた高尚な主義主張を持っている訳ではないが、私の持っている主な理由は、以下の 3 点である。

- 自作派の為
- 情報交換の為
- 私の為

自作派の為

私が、コンピュータの自作を始めたのは、相当昔の話しであるが、当時から比べると、部品等は格段に性能が向上し、価格は低下している。しかし、コンピュータの自作を行なう環境は、昔より良くなっているかと言うとそうでもない。

以下に列記するようにコンピュータの自作は、当時よりむしろ悪くなっているのだ。

動機付け 昔は、自分のコンピュータを所有するためには、自作しか方法が無かった。これは非常に大きな動機である。

現在では、コンピュータを所有するために自作は必要ではなく、むしろ、購入した方がメリット多い。

情報過多による情報不足 毎年どころか、毎月のように進歩するコンピュータ業界のため、本当に必要な情報を選択する事が困難になっている。

高性能・高密度・高速 あまりにも、高性能化したコンピュータ (主に CPU) が自作では使いこなせなくなっている。特に、高密度化するために半田付けすら困難になり、また、高速化がバス設計を困難にしている。

関連するソフトウェアの高価格化 その昔のハッカーには、「騎士道精神」のようなところがあり、必要となる開発ソフトは、只 (フリー) で入手する事が可能であった。

ところが、現在では、ほとんどのソフトウェアは、有料で且つ高価になっている。コンピュータのソフトウェアに金儲け主義を導入したのは、ビル・ゲイツに違いないのだが、 C/C++ コンパイラやリアルタイム OS 等は、非常に高価で、数万から数十万、物によっては数百万のイニシャルコストが必要である。とても個人の手の出るところではない。

これらをすべてでは無いが、補うのが、私の目的である。
外部インターフェースから内部タイマーに至るまで、ほぼ全ての機能をオンチップで持つ SH-3 をコアに使う事で、回路を簡略化している。その上、回路図を公表し、動作確認した結果を示す。
さらに、GCC を始めとするフリーウェアを用いて、開発環境を如何に構築するかの情報も取り入れるつもりである。

情報交換の為

Fox プロジェクトが、実際に進み、多くの人が、SH-3 を使ったコンピュータを作るようになると、自然に情報が出ると思う。
そんな情報が集まり、皆が使えるようになれば良いなと願っている。

私の為

私とて、聖人ではないので、Fox をフリーにする事が全て他人の為だけではない。
正直に白状すると、下心があるのだ。(もちろん、金儲けではない)
その多くは、情報交換が目的である。
私がある程度、 Fox の枠組みを決めれば、それに対する応用 (アプリケーション) の情報が集まるのではないかと期待している。既に安井吏氏が、 SH-3 用に BSD を移植しようとしているように、例えば、 Hurd とか RT-Mach を SH-3 に移植しようとする人が現れることを期待しているのだ。
さらに、色々な人に使ってもらうと言う事は、言ってみれば「無料でデバック試験してもらっている」ような物である。ですから、バグレポートはお願いする。
最後に、 Fox が『恒星間　鮭の卵』に使えるようなコンピュータに進化する事が、究極の目的である。

― コラム ： ホスト・コンピュータの最小構成 ―

Fox を作り、使う上で、ホスト・コンピュータが必要な事は、本文を読んで判ったと思うが、はたして、どの程度のコンピュータが必要なのであろうか？ ホスト・コンピュータの要求如何により、 Fox にかかるコストが左右される事から、この件が気がかりな方も多いだろう。

そこで、拾いもののノート・パソコンで、どこまでホスト・コンピュータとして使いものになるか、挑戦してみた。

対象となったノート・パソコンのスペックは、以下の通りである。

表 2.1: 最小構成 ホスト・コンピュータのスペック

項目	種類/性能	備考
形式	Panasonic AL-N1	B5 ノート 1996 年製らしい
CPU	Pentium	100 M Hz か 150 M Hz の様だが、不明
主メモリ	32 M バイト	48 M バイト以上あれば、もっと楽だったんだが
HDD	770 M バイト	1.5 G 以上 なら、マルチブート可
OS	Windows 98 と NetBSD	元々は Windows 95

と、まあ、こんな感じだが、一応、 Fox の開発環境に対して、必要な事は一通り可能である。

難点があるとしたら、以下の点だ。

- Windows の Cygwin 上で gcc を構築するには、メモリ不足で、処理速度も遅い。
- バッテリーが、劣化しているので、せっかく B5 ノート・パソコンなのに、モバイルできない。

第II部

基本操作編

第 II 部の説明

第 II 部では、Fox を使ったアプリケーション・プログラムの作り方を説明する。

このとき、製作やホストコンピュータへの開発環境のインストールは既に済んでいるものとして、使い方やプログラミング、デバッグの仕方を示す。

従って、実際に、ここで説明するアプリケーション・プログラムを作る前に、最初に Fox 自体を製作・試験し、ホストコンピュータにプログラミング開発環境をインストールする必要がある。

Fox 自体の製作・試験は、第 III 部 の 第 4 章を参考にしてもらいたい。

アプリケーション・プログラムの開発環境

Fox は、自分自身でアセンブラやコンパイラ等を持たず、プログラムを開発するためのホスト・コンピュータが必要である。[1]

第 III 部 第 5 章 に開発環境のインストール方法を示す。[2]

[1] 現状ではクロス環境のみだが、近い将来、NetBSD/sh3 が、数百メガ以上の第容量ハードディスク上で動くことが確認できれば、セルフ環境でカーネルの再構築を行うことも可能になるだろう。

[2] 現状は未完成

第3章　最も簡単なアプリケーションの作り方

この章では、 Fox で動作するアプリケーションの作り方を、簡単な例を用いて説明する。

3.1　LED の点滅回路

Fox の様なワンボード・コンピュータは、主に組込み用途で使われることが多い。従って、使い方のサンプルとしては、極く簡単な入出力装置に対するコントロールのプログラミング方法が適当だと思われる。

SH-3 には、オンチップにシリアルインターフェースと 8 ビットのパラレルインターフェースが標準で付いている。[1]

図 3.1 に、 LED の点滅回路を示す。

回路の説明をしておくと、 CPU から出ているパラレルポートの上位 8 ビットに LED を接続し、下位 8 ビットにプッシュスイッチを接続している。

SH-3 の場合、ポートから出せる電流は、わずか 2mA であり、、 LED に直列に電流制限用の 2k Ω の抵抗を付ける必要がある。

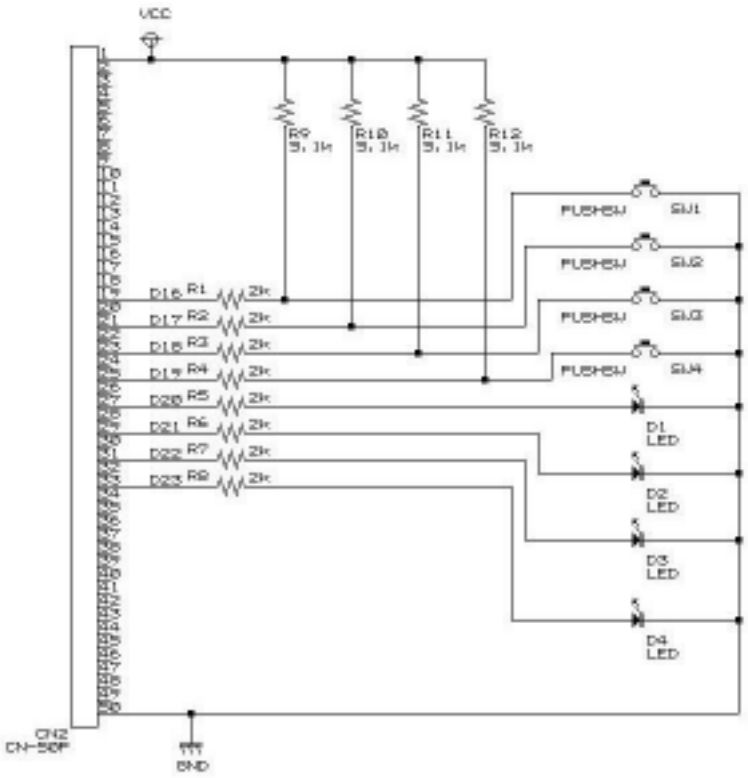

図 3.1: LED の点滅回路

[1] 正確に言うと、 SH-3 のデータバスは 8 、 16 、 32 の可変長になっており、データバスを 8 ビット、または 16 ビットで使うときに、上位 16 ビットの内、 8 ビットをパラレルポートとして使用することが可能になっているだけだ。Fox の場合、オンボードのメモリと、コンパクトフラッシュインタフェースだけの場合、 8 ビットと 16 ビットモードのアクセスしか使わないので、 8 ビットをパラレルポートとして、使用することが可能である。なお、 Fox の拡張バスを使用して、基板外に 32 ビットアクセスするメモリや IO を接続することも可能だが、この場合は、当然、 8 ビットをパラレルポートとして使用することは不可能である。

以下、未完成

第III部

製作・設定編

第4章　Fox 量産型キットの組み立て

図 4.2 にプリント基板版の Fox の回路図を示す。

図 4.2 に示したように、プリント基板版の Fox は、CPU と ROM、RAM の他はリセット波形整形用のロジック IC があるだけの至ってシンプルな構成である。単体では何の入出力も持たないため、何らかの拡張ボードを作って、インターフェースを持たせる必要がある。もっとも、シリアル・インターフェースに関しては、単に電圧コンバートするだけで、通常の RS-232C とインターフェースできる。

初期起動時のプログラムは、ROM に書かれる。この ROM には EEPROM を使っており、Fox の基板完成後でも、外部から ISP(イン・システム・プログラム) 可能である。ただし、それ相応のインタフェースが必要であり、この点に付いては後述する。

Fox の基板上には、コンパクト・フラッシュ用のコネクタが標準で付いている。このコンパクト・フラッシュ用コネクタは、SH-3 のバスに直結しているもので、以下のような機能限定がある。

1. 活線挿抜には未対応
2. 自動認識無し
3. 3.3V 対応のカードのみ
4. CF のメモリカード領域だけ
5. IRQ 無し

図 4.1: Fox 外観写真

つまり、コンパクト・フラッシュの一般的な IO カードは接続できない。主にメモリカードを接続して、ハードディスク代わりに使う事を目的にしている。

Fox 基板上の ROM に IPL を入れておき、メモリカード上のアプリケーション・プログラムをロードするのが、主な使い方である。最後に紹介する NetBSD/sh3 の場合は、同様に NetBSD/sh3 のカーネルを IPL でロードして起動し、OS 起動後も、コンパクト・フラッシュをハードディスク代わりにマウントする事ができる。

4.1　プリント基板と部品

4.2　配線・組み立て

Fox は、未配線のプリント基板とパーツだけで頒布する。従って、ユーザーが半田付けして組み立てる必要がある。

Fox で使っている部品は、ほとんどが表面実装の小型高密度実装のものである。最も配線が難しいのが CPU であり、0.5 ミリ・ピッチの QFP である。次に同じく 0.5 ミリ・ピッチの ROM が続く。さらに抵抗やコンデンサーの多くはチップ部品であり、一つ一つの半田付けは、さほど難しく

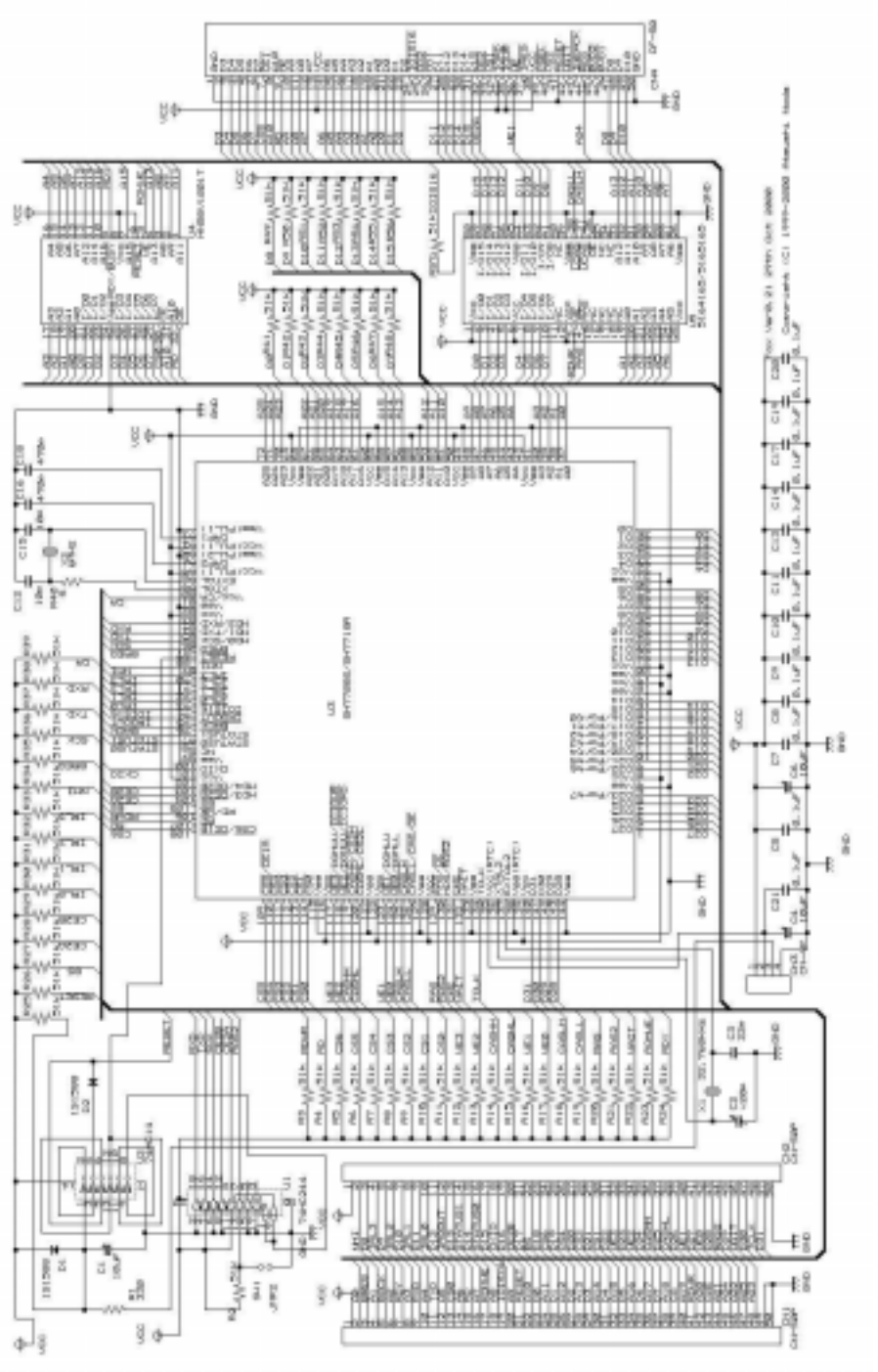

図 4.2: Fox 基板回路図

は無くとも数が多いので大変である。

これらの部品はアマチュアでは半田付け不可能と思われているが、実は普通の半田ごてで半田付けできる。半田ごてや半田に数万円以上のコストをかけると良いと言う話も聞いたのだが、私はアマチュアが簡単にできる事を示したかったので、秋葉原で店頭販売している中で最も安い温度制御付きの半田ごてを用いた。確か 3800 円で購入したものである。半田ごては、最小限、温度制御が無いと駄目だが、このような安物では設定温度を変更できないが問題は無い。半田も店頭で安売りしているものを使っている。また、チップ部品を半田付けするために先の尖ったピンセットも必要だ。と言っても私が使っているものは 800 円の安物だが。

さて、実際の半田付けのコツだが、ほとんど最初の位置合わせで決まると言っても過言ではない。如何に基板のパターンに部品をピッタリと合せるかが勝負の分かれ目となる。

CPU は四辺の四方向共にピンが伸びている QFP パッケージなので、全てのピンがパターンに合うように位置を決める。ROM や RAM は二方向だけなので少しは簡単である。

最初に、対角線上の 2 つのピンだけを半田付けし、ずれが有った時は修正する。もし、位置の修正が上手く行かないときは思い切って、パーツを外し、パターンの半田を吸い取って、最初からやり直したほうが良いだろう。ある程度、半田付けを進めてからの位置修正は不可能だと思ったほうが良い。

対角 2 ピンだけの状態で、位置が完全に合い、全てのピンがパターンに一致していることを確認したら、他のピンを半田付けする。接続している対角線と異なる対角線上のピンから半田付けを始める。このとき半田の量は多すぎて、ブリッジする程度でも構わない。ブリッジを気にせずに全てのピンをパターンにしっかり半田付けした後、半田を吸い取り線で吸い取っていく。この時、ブリッジしない程度に半田を吸い取ること。あまり吸い取りすぎるとパターンとピンが離れてしまうので注意が必要だ。

ピン間隔が狭いとは言え、パターン上でのブリッジは肉眼でも、虫眼鏡程度でも確認できる。実は、部品のピンの根元、パッケージの裏側で半田がブリッジする事があり、これが最も難しいところである。これを避けるには、最初に半田を盛るとき、プリント基板のパターンとピンのところには多目でも良いが、ピンの根元には半田が流れないように注意することが必要である。万一、ピンの根元でパッケージの裏側でブリッジしても半田を吸い取り線で吸い取る事は可能なので、余り神経質になる必要は無いが、単に発見が難しいだけである。

部品を付ける順番は自由だが、私の場合、CPU、ROM、RAM と言ったように難しい順番に付けて行く。その後、74HC244、74HC14 を付け、チップ抵抗、コンデンサーやクリスタルを付けた後、最後にコネクタを付ける。なお、裏面はコスト低減のため、シルク印刷をしていないが、パターンは全てパスコンの 0.1 μ F のコンデンサーである。

プロが半田付けする場合はともかく、アマチュアの場合は、CPU や ROM、RAM と言った部品毎に休憩を入れるべきだ。続けてやると集中力が持続しなし、細かい作業であるため、視力も低下する。私の場合、続けて半田付けした為に、一番簡単な筈の RAM で接続不良を起こしてしまった。

ゆっくりと、神経を集中して半田付けする事をお勧めする。

なお、ジャンパーである SW1 は、CPU のエンディアンの設定である。オープンでリトル・エンディアン、クローズでビック・エンディアンに設定できる。以降の説明ではビック・エンディアンでプログラムを作っているので、SW1 はクローズにしておこう。

― メモ ―

Fox の試作プリント基板である FOX3019 については、リセットタイミングの設計が甘く、ダイオード D2 をジャンパーでショートさせる必要があるので、注意する必要がある。

4.3　ROM ライタ

SH-3 はプログラム用の ROM を内蔵していない。そのため、Fox では起動時のプログラムを置くための ROM を用意している。この ROM は小型化のため高密度実装で基板上に半田付けされている。従って、IC ソケットを使うって ROM を挿抜し、一般的な ROM ライターでプログラムを書き込む事は不可能である。

Fox では、オンボードの ROM にプログラムを書き込む ROM ライターが必要になる。ROM ライターの回路図を図 4.3 に示す。なお、今回、頒布予定の Fox の基板と部品には ROM ライターの部品は含まれていないが、入手はごく簡単な部品だけで構成されている。

ROM にプログラミングする時、アドレスやデータ等のバスが CPU と衝突しないように、BREQ(バス・リクエスト) を使って、CPU を止めている。その後、ホストとなるパソコンのプリンタ・ポートからアドレス／データ／コントロール信号を制御して ROM にプログラムをする。

この制御は、ディスクリートで組むと、かなり大規模な回路になることと、パソコンのプリンタ・ポートの信号レベルが 5V に対して、Fox 内部の信号レベルが 3.3V になる事が難しい。

そこで、Xilinx 社の XC95108 と言う FPGA(正確には CPLD) を使っている。この XC95108 の信号レベルが、3.3V と 5V を同時に使用できると言う優れものである。

アマチュアが FPGA(CPLD) を使うとなると、

- 回路入力等のツールが高価
- FPGA(CPLD) へのプログラマーが高価

と言う事で尻込みしがちである。

ここでは、Fox の精神で、フリーで FPGA(CPLD) をプログラムする方法を紹介しよう。用意するものは、パソコンとインターネットと XC95108 自体の他は、74HC125 が僅か 2 個である。

XC95108 に書き込む回路は VHDL で記述し、このファイル名は ROMW.VHD である。また、ピン配置も ROMW.UCF と言う名前のファイルに作る。

FPGA(CPLD) にプログラムする回路の記述には、VHDL 等の記述言語以外にスケマティック (論理記号を使う通常の回路 CAD 的な入力) を使う方法もあるが、VHDL や Verilog HDL と言ったのハードウェア記述言語を使う方が先進的であり、また今後の応用も期待できるので、この際、勉強しておこう。

次に、VHDL で記述した回路を実際に FPGA(CPLD) に書き込みするデータに変換するコンパイル作業が必要である。このために必要なコンパイル・ツールは Xilinx のホームページから無料で使う事ができる。

http://www.xilinx.co.jp/

上記の Xilinx の URL で、WebFITTER と言うページでホームページ上のフォームから、VHDL のソースをアップロードし、コンパイル作業をオンラインで行う。Xilinx では、WebPACK と言うスタンドアロンで使用可能なツールもインターネットからダウンロードできる。

VHDL ソースをコンパイルすると、Design.jed と言うファイルが出力される。これを XC95108 にプログラムをする。プログラムに必要なツールも Xilix のホームページからダウンロードできる。また、書き込みに必要なハードも僅か 74HC125 が 2 個で作る事ができる。

実は、XC95108 にプログラムする為に必要な回路は、図 4.4 に示している。図 4.4 の CN1 をパソコンのプリンタ・ポートに接続し、CN3 ヘッダを CN5 コネクタに接続すると XC95108 にプログラムできる。

また、CN3 ヘッダを CN4 コネクタに接続すると Fox にプログラムできる。

XC95108 に対するプログラムの注意点はパソコンのプリンタ・ポートに接続するケーブルである。ここにしっかりしたものを使わないとノイズで書き込みエラーが多発する。

なお、私の経験では、XC95108 の電源投入時に、ごく稀に出力端子から、ロー・レベルが出ることがある。誤動作を避けるために、ROM の WE 端子に対する出力に 74HC125 を用いて、電源投入時に出力をハイ・インピーダンスにしている。

また、ROM ライターは RS232C への電圧レベル・コンバータを内臓しており、Fox の SH-3 が持つシリアル・インターフェースをパソコンと接続する機能もある。

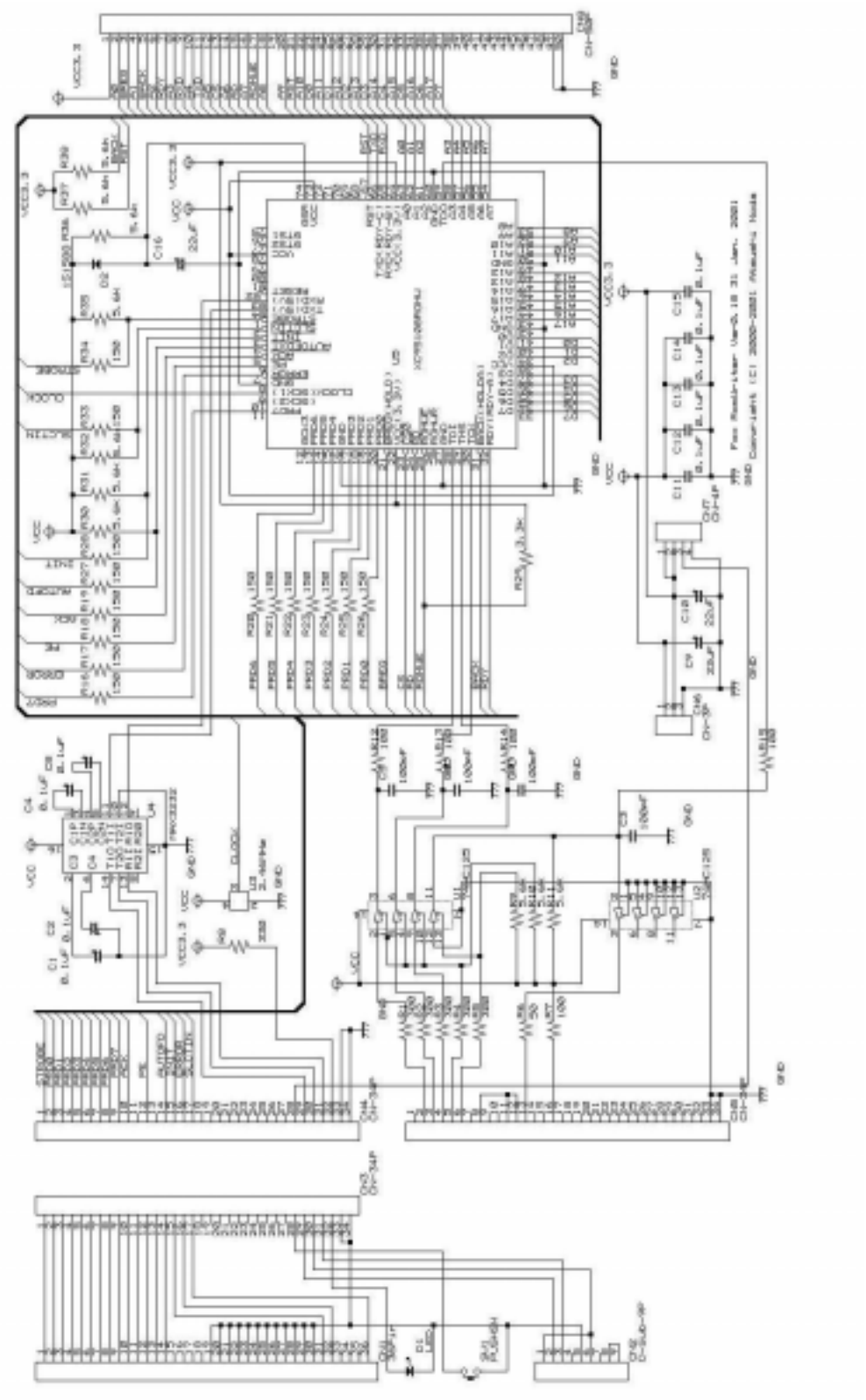

図 4.3: Fox 用 ROM ライター

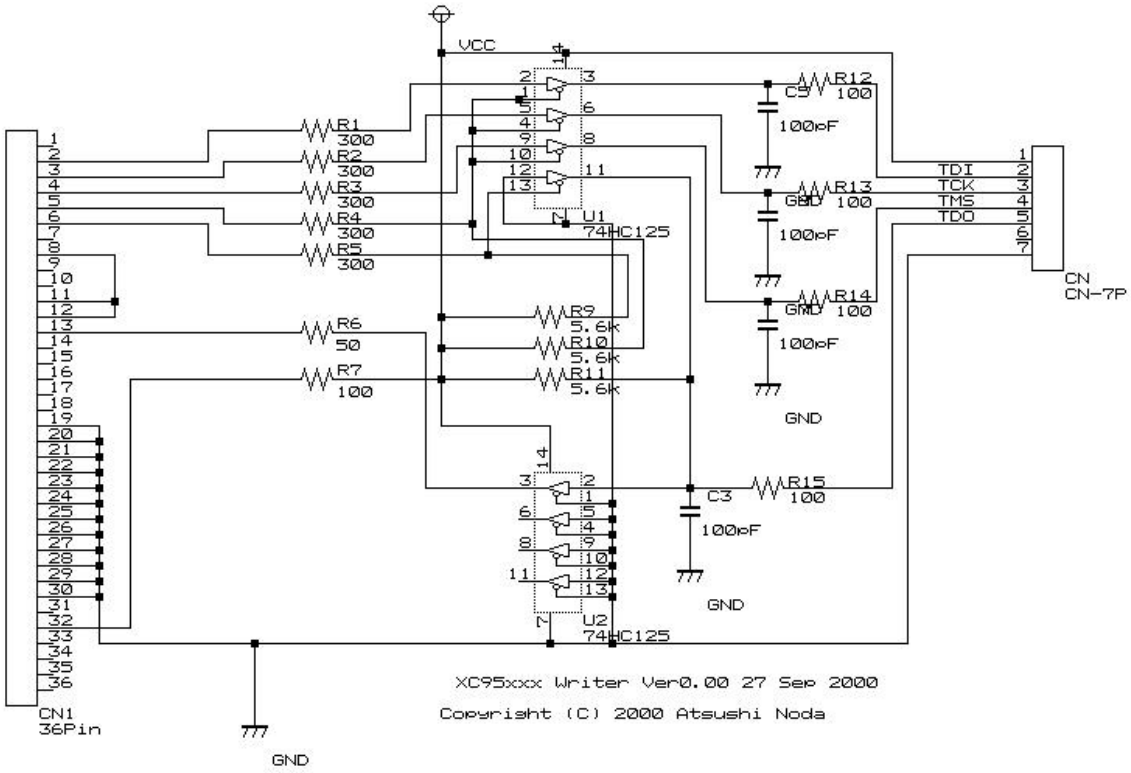

図 4.4: FPGA ライター

4.4 プログラミング

Fox のROM に対する書き込みは、Windows95/98 上のプログラムによって、上記の ROM ライターをコントロールする。この時、プリンタ･ポートを直接操作するので、残念ながら、WindowsNT/2000 では、ライタ・プログラムは動作しない。また、プリンタ・ポートは双方向モードで動作する必要があり、予め BIOS 等で設定しておく。Dynabook 等、ごく一部のパソコンにプリンタ・ポートを双方向モードに設定できない機種があるため、注意が必要だ。

Windows95/98 上で、ROMW.EXE を動作させ、書き込む対象の HEX ファイルを読み込ませて、書き込むだけで、Fox の ROM にプログラミングができる。

4.5 動作チェック

Fox 及び ROM ライタが完成したら、動作チェックである。

まず、ROM ライタを単体で動作するか確認した後、Fox を接続する。この時、コネクタなどの接続ミスに注意すること。

最初に、test.hex を書き込む。

ちゃんと書き込めれば、リセットをかけて、動作させる。ホストのパソコンでターミナル・エミュレータを起動しておく。回線の速度は 9600bps である。Fox の配線に問題が無ければ、起動メッセージを出した後、内部メモリのチェックを行う。少し時間がかかるが、メモリに異常が無ければ、その旨のメッセージを出力してプログラムを終了する。

この後、コンパクト・フラッシュからプログラムをロードする IPL を書き込んでおく。

4.6 インターフェース・ボード

Fox を動作させるために、一々ROM ライタを接続するのは大変である。そこで、RS-232C へのレベル・コンバータを 3.3V 単一電源で行う MAX3232 を用いたインタフェース・ボードの回路を図に示す。

私の場合、このインタフェース・ボードを Fox に重なるようにコンパクトに作った。

その他に、DCDC コンバータで単三電池 2 本から 3.3V を供給しており、このような小さな自作システムで NetBSD/sh3 と言った本格的なオペレーション・システムを動作させている。

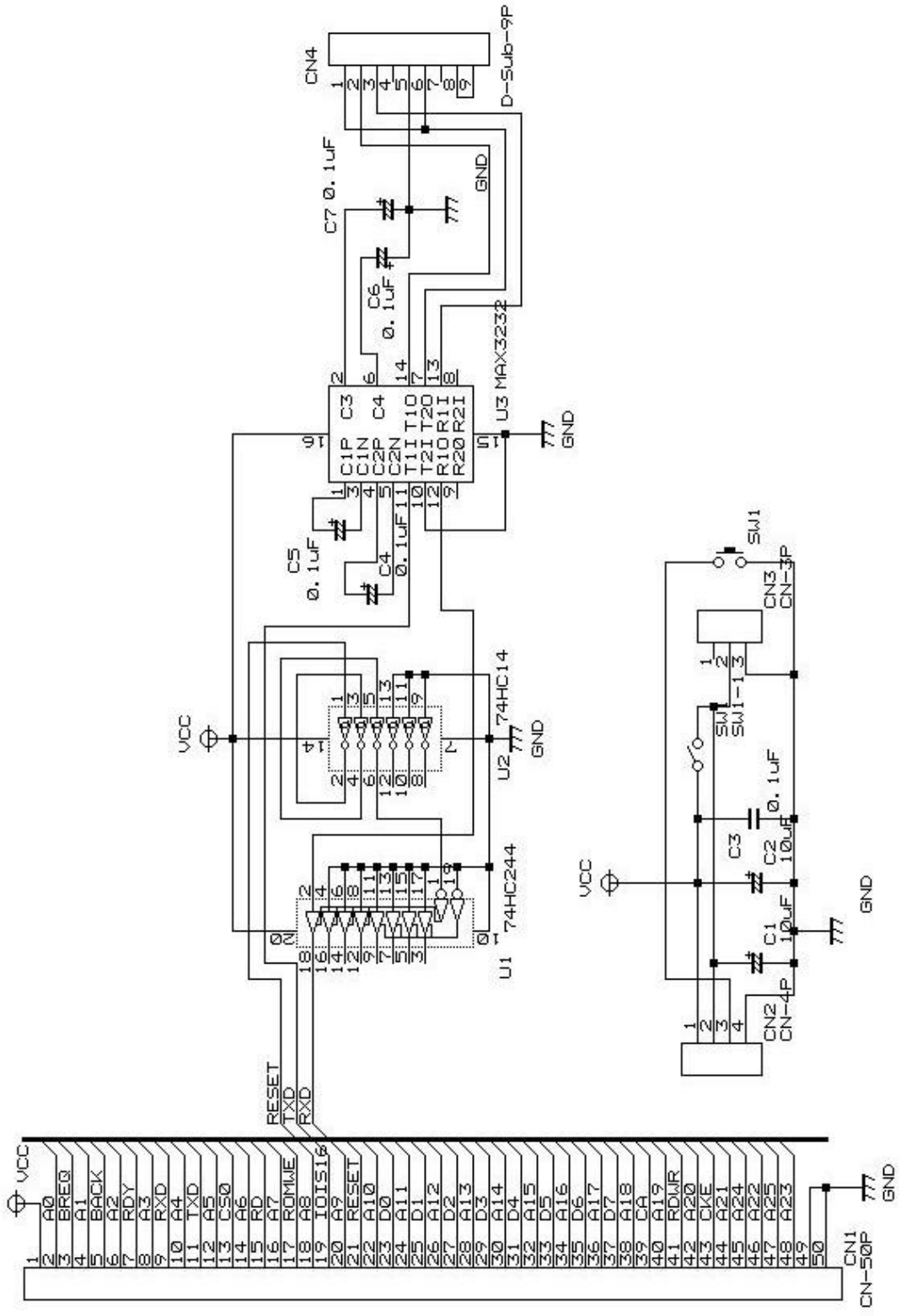

図 4.5: RS232C インターフェース

第5章　ホスト・コンピュータに開発環境を、インストール

未完成

第IV部

設計・移植編

未完成

第Ⅴ部

おわりに

未完成

付 録A　仕様

表 A.1: メモリマップ

領域号	エリア	アドレス
P0 領域	エリア 0	0x00000000 - 0x0001ffff EEPROM 128k バイト 8 bit 幅
		0x00020000 - 0x03ffff EEPROM のイメージ
	エリア 1	0x04000000 - 0x07ffffff 未実装
	エリア 2	0x08000000 - 0x0bffffff 未実装
	エリア 3	0x0c000000 - 0x0c7fffff EDO-RAM 8M バイト 16 bit 幅
		0x0c800000 - 0x0fffffff EDO-RAM のイメージ
	エリア 4	0x10000000 - 0x13ffffff 未実装
	エリア 5	0x14000000 - 0x17ffffff 基板上 コンパクトフラッシュ
	エリア 6	0x18000000 - 0x1bffffff 未実装
	エリア 7	0x1c000000 - 0x1fffffff 未実装
P1 領域	0x80000000 - 0x9fffffff P0 領域のイメージ	
P2 領域	0xa0000000 - 0xbfffffff P0 領域のイメージ	
P3 領域	0xc0000000 - 0xdfffffff P0 領域のイメージ	
P4 領域	エリア 0-6	0xe0000000 - 0xfbffffff P0 領域のイメージ
	エリア 7	0xfc000000 - 0xffffffff コントローラ類や CPU 内蔵周辺機器

表 A.2: 領域の説明

領域	MMU	キャッシュ	備考
P0 U0 領域	アドレス変換	可 (コピーバック ライトスルー)	ユーザーモードでは、ここのみ
P1 領域	固定アドレス	可 (ライトスルー)	OS 等のシステムプログラム
P2 領域	固定アドレス	不可	CPU 外の 周辺機器制御
P3 領域	アドレス変換	可 (コピーバック ライトスルー)	
P4 領域	固定アドレス	不可	CPU 内の 周辺機器制御

表 A.3: 部品リスト

部品番号	部品種	型式	個数
-	基板	-	1
U1	ロジック IC(SOP)	74HC244	1
U2	ロジック IC(SOP)	74HC14	1
U3	SH-3S CPU	SH7708S	1
U4	128K byte EEPROM	HN58V1001F	1
U5	8M byte EDO-RAM	HM5164165	1
X1	クリスタル	32.768kHz	1
X2	クリスタル	6MHz	1
CN1-2	コネクタ	50 ピン	2
CN3	コネクタ	4 ピン	1
CN4	コネクタ	CF コネクタ	1
C1,4,6	電解コンデンサ	10uF	3
C2,C3	チップ・コンデンサ	22p	2
C5,7-11,13,14,17,19-20	チップ・コンデンサ	0.1uF	11
C12,15	チップ・コンデンサ	10p	2
C16,18	チップ・コンデンサ	470p	2
D1-2	ダイオード	1S1588	2
R1	チップ抵抗	330 Ω	1
R2-39,41-57	チップ抵抗	51k Ω	55
R40	チップ抵抗	10 Ω	1

表 A.4: Fox3 バス (1/2) : CN1

信号名	I/O	ピン番号	ピン番号	I/O	信号名
VCC(3.3V)	-	1	2	O	A0
$\overline{BREQ}$	I	3	4	O	A1
$\overline{BACK}$	O	5	6	O	A2
RDY	O	7	8	O	A3
RXD	I	9	10	O	A4
TXD	O	11	12	O	A5
$\overline{CS0}$	I/O	13	14	O	A6
$\overline{RD}$	I/O	15	16	O	A7
$\overline{ROMWE}$	I	17	18	O	A8
$\overline{IOIS16}$	I	19	20	O	A9
$\overline{RESET}$	I/O	21	22	O	A10
D0	I/O	23	24	O	A11
D1	I/O	25	26	O	A12
D2	I/O	27	28	O	A13
D3	I/O	29	30	O	A14
D4	I/O	31	32	O	A15
D5	I/O	33	34	O	A16
D6	I/O	35	36	O	A17
D7	I/O	37	38	O	A18
CA	I	39	40	O	A19
RD/$\overline{WR}$	O	41	42	O	A20
CKE	O	43	44	O	A21
A24	O	45	46	O	A22
A25	O	47	48	O	A23
GND	-	49	50	-	GND

表 A.5: Fox3 バス (2/2) : CN2

信号名	I/O	ピン番号	ピン番号	I/O	信号名
VCC(3.3V)	-	1	2	I	NMI
D8	I/O	3	4	I	$\overline{IRL3}$
D9	I/O	5	6	I	$\overline{IRL2}$
D10	I/O	7	8	I	$\overline{IRL1}$
D11	I/O	9	10	I	$\overline{IRL0}$
D12	I/O	11	12	O	$\overline{IRQOUT}$
D13	I/O	13	14	O	STATUS1
D14	I/O	15	16	O	STATUS0
D15	I/O	17	18	O	CKIO
D16	I/O	19	20	O	$\overline{CE2B}$
D17	I/O	21	22	O	$\overline{BS}$
D18	I/O	23	24	O	$\overline{CS6}$
D19	I/O	25	26	O	$\overline{CS4}$
D20	I/O	27	28	O	$\overline{CS2}$
D21	I/O	29	30	O	$\overline{CS1}$
D22	I/O	31	32	O	$\overline{WE3}$
D23	I/O	33	34	O	$\overline{WE2}$
D24	I/O	35	36	O	$\overline{CASHH}$
D25	I/O	37	38	O	$\overline{CASHL}$
D26	I/O	39	40	O	$\overline{WE1}$
D27	I/O	41	42	O	$\overline{WE0}$
D28	I/O	43	44	O	$\overline{RAS2}$
D29	I/O	45	46	I	$\overline{WAIT}$
D30	I/O	47	48	O	TCLK
D31	I/O	49	50	-	GND

凡例： O:CPU 周辺 I:周辺から CPU I/O:双方向

表 A.6: 電源用コネクタ CN3

ピン番号	信号名
1	+3.3V 電源
2	+3.3V RTC 用電源
3	$\overline{RESET}$
4	GND

表 A.7: XC95108 ピン機能

ピン	機能	ピン	機能	ピン	機能	ピン	機能
1	STROBE	22	VCC(3.3V)	43	A17	64	VCC(3.3V)
2	SLCTIN	23	CS0	44	A16	65	RXD
3	INIT	24	RD	45	A15	66	TXD
4	AUTOFDTXT	25	$\overline{ROMWE}$	46	A14	67	
5	ACK	26	$\overline{ROMWR}$	47	A13	68	
6	PE	27	GND	48	A12	69	
7	ERROR	28	TDI	49	GND	70	
8	GND	29	TMS	50	A11	71	
9	CLOCK(GCK1)	30	TCK	51	A10	72	
10	(GCK2)	31	$\overline{BACK}$	52	A9	73	VCC(5V)
11	PRD7	32	RDY	53	A8	74	GSR
12	(GCK3)	33	D7	54	A7	75	
13	PRD6	34	D6	55	A6	76	GTS1
14	PRD5	35	D5	56	A5	77	GTS2
15	PRD4	36	D4	57	A4	78	VCC(5V)
16	GND	37	D3	58	A3	79	
17	PRD3	38	VCC(5V)	59	TDO	80	
18	PRD2	39	D2	60	GND	81	
19	PRD1	40	D1	61	A2	82	
20	PRD0	41	D0	62	A1	83	RXD(5V)
21	$\overline{BREQ}$	42	GND	63	A0	84	TXD(5V)

関連図書

[1] 野田篤司、小型 SH-3 ボード Fox の設計と製作、インターフェース 1999 年 6 月号

[2] 野田篤司、SH-3 用コンパクトフラッシュ インタフェースの製作、インターフェース 1999 年 12 月号

[3] 野田篤司、SH-3 ボードへの NetBSD の移植と実装、インターフェース 2000 年 4 月号

[4] 柏谷春樹、eCos の詳細と SH-3 ボードへの移植、インターフェース 2000 年 10 月号

[5] 野田篤司、Fox プリント基板と NetBSD/sh3 のインストール、インターフェース 2000 年 12 月号

[6] B. W. カーニハン / D. M. リッチー 著、石田晴久 訳プログラミング言語 C 第 2 版 (訳書訂正版)、共立出版株式会社、 ISBN 4-320-02692-6