

Visual Synthesizer := $\frac{Sonitus}{Lucis \times Colorum}$

Iori yoneji

2011 年 1 月 29 日

0.1 アブストラクト

この論文は、

- 実際に自分がオープンソフトウェア/ハードウェア、オープンノウハウについての公開を通して、これらの活動への、読者の参加容易性の向上

を図るとともに、

- オープンソフトウェア/ハードウェア、オープンノウハウによる恩恵の享受可能性
- パーソナルファブリケーションとオープンソースをとりまく社会とビジネスモデルの現状と今後の予測
- 本研究で開発したハードウェア、およびソフトウェアの新規性、および課題
- ソースコードを実際に読み、書くことによる、ソフトウェアとハードウェアの製作についての現状の開発環境の問題点と実地でのプロセスの実例

を明らかにし、以上のことから、90億総ソースコード・リテラシィの時代になる事を主張している。

0.2 キーワード

電気電子, コンピュータ科学, ARM, 組み込み, PLD, CPLD, C 言語, Verilog, ハードウェア, ソフトウェア, 開発, ものづくり, オープンソース, ヴァージョン管理, ソースコードリテラシィ, SNS

0.3 お品書き

今回の製作対象の、外から見た仕様について

- 学生が買える、廉価なカメラモジュールで、室内等の明るさや色などを元にパラメータを作成する。
- パラメータを元にスピーカから音が鳴る。

0.4 コンタクトなど

連絡をいれるには:ioriveur@ioriveurlabs.dnsalias.org
ここに今回の成果物などが公開されている:ioriveur@github
この著者のブログはこちら:否定的なものは怠惰の中では繁殖する

ここで製作物に関する動画が公開されている <http://www.youtube.com/user/iorivr>

0.5 ライセンス



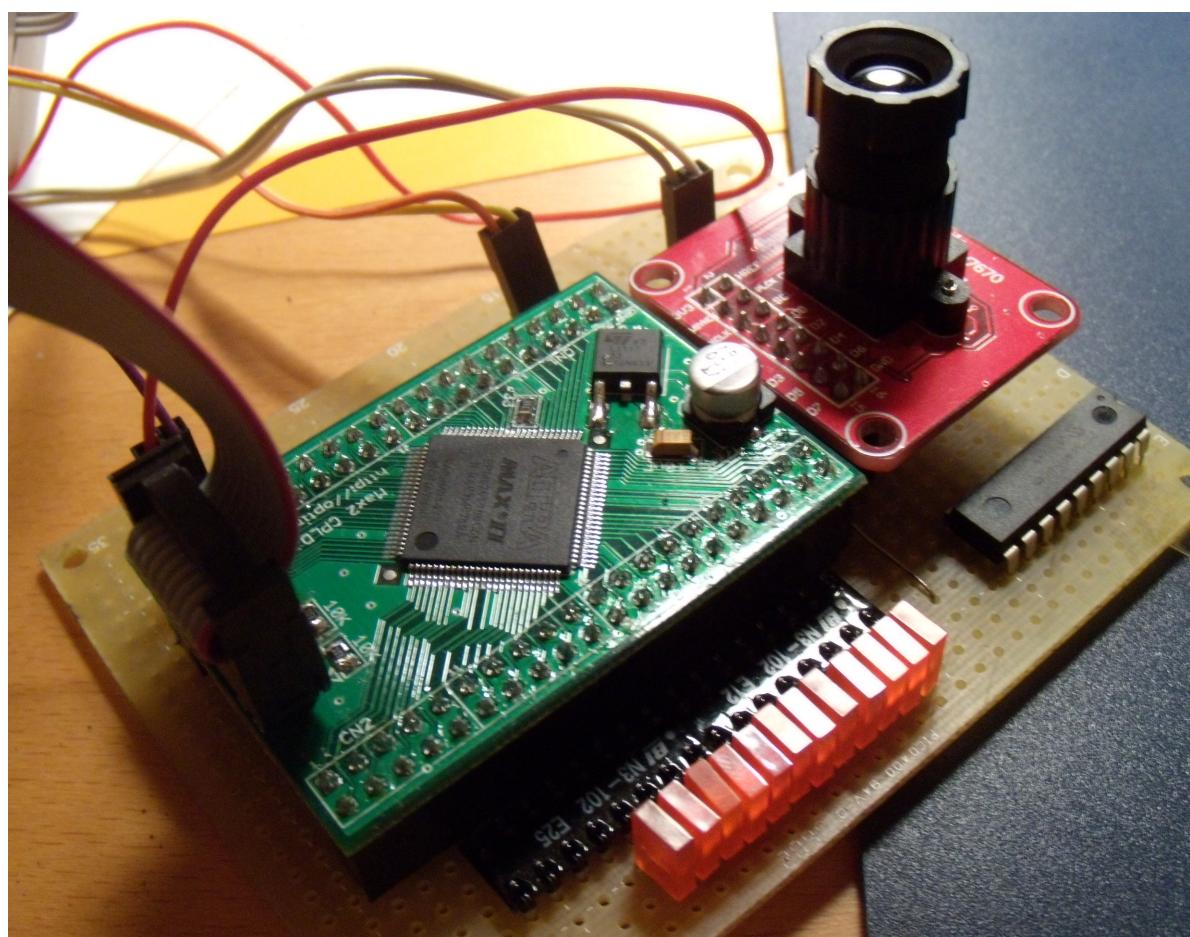
This Artwork by Iori Yoneji is licensed under a Creative Commons 表示 - 改変禁止
2.1 日本 License.

目次

0.1	アブストラクト	2
0.2	キーワード	2
0.3	お品書き	2
0.4	コンタクトなど	3
0.5	ライセンス	3
第Ⅰ部 導入		9
第1章	哲学	11
1.1	前口上 -お目汚し-	11
1.2	実用性 -音楽的要素がなくても-	12
第2章	内部仕様について	19
2.1	最初期の仕様	19
2.2	現在の仕様	21
第3章	開発環境	23
3.1	オープンソースデスクトップ環境において	23
3.2	Windows 環境において	27
3.3	git と github について	27
第Ⅱ部 開発		31
第4章	Verilog を書く	33
4.1	モジュールの構成	33
4.2	clkdivider(分周器)	34
4.3	データ抽出レイヤ	34
4.4	データ転送モジュール	39
4.5	シミュレーション	39
第5章	I2C セットアップ関数の開発	45
5.1	データ送出関数	45
5.2	シーケンシャルデータ受信関数	46
5.3	OV7670 全リード関数	47
5.4	OV7670 バイトライト関数	47

第 6 章	SCCB での OV7670 コンフィギュレーション	49
6.1	OV7670 の注意点	49
6.2	OV7670 用のレジスタセットアップデータの整理	49
第 7 章	独自シリアル転送プロトコル	55
7.1	トランスマッタ	55
7.2	レシーバ	57
第 8 章	SSG ドライバの製作と研究	59
8.1	SSG というは	59
8.2	ハードウェア層での結線	59
8.3	ソフトウェア層での工夫	60
8.4	制御速度に関する実験	62
第 9 章	ソースコードを読む技術	65
9.1	関数値渡しの謎	65
9.2	返り値の研究	70
第 III 部	社会	75
第 10 章	この論文に使われている技術	77
10.1	ヴァージョン管理	77
10.2	pLATEX	78
第 11 章	ノウハウの共有	81
11.1	ブログ	81
11.2	SNS	82
11.3	フォーラム	83
11.4	man	83
第 12 章	社会に於けるオープンソースの立場	85
12.1	ビジネスモデル	85
12.2	ホスティングサービス	85
12.3	Arduino	86
12.4	幅広い層の参画	86
第 13 章	90 億総ソースコード・リテラシーの時代へ	89
13.1	PC 価格の低下	89
13.2	ツールの成熟、コミュニティの発達	89

図2 OV7670とMAX II



第Ⅰ部

導入

第 1 章

哲学

1.1 前口上 -お目汚し-

過去に様々な形において、音楽と映像の関連づけが行われてきた。

映画中の音楽であったり…といった面もあるが、それはむしろ視聴者への心理的な関連付けを両者が並列して行っているということである。

ここで私が言いたいのは、むしろビジュアライザやイコライザの類についてである。

つらつら長く書いてしまいそうで怖い。意味の希薄な前口上を冗長に書いて見たところでなにもない。当然だ。つまるところ、と結論を急いでしまおう。詰まるところ、私の言いたいのは、iTunes や Rhythmbox のようなビジュアライザのような音楽のデータの(ごく一部をランダムに用いた)画像の自動生成システムを、なんの価値もなく、ただいたずらに PC に負担をかけるばかりで、あきるほど見てきた。しかし、今までにおいて、その逆のハードウェアを作ってみようという試みは寡聞にして聞かない。

音楽は、

- 作業中、考え中、読書中の類のときの BGM として。またトーク番組の BGM として
- その音楽を楽しむ。
- その曲を覚えてなにか自分で歌って踊って演奏してみたいなことをする。

といった用途が考えられる訳だけれども、とにもかくにも、再生中のディスプレイを凝視することなんて(むしろそのウィンドウが開かれていることだって)、普通に考えて、ないことなのだ。

ところが、である。なにか外を向いてみる、とかその類の、目で見ているもの、顔を向けている対象に特定の意味と意志をフォーカスしていないときには、その風景が音楽化(soundlize) されても、困らない。…んじゃないかなあと考えたわけである。

崩して言ってみる。

飽きているからかも知れないけれど、音楽が(機械的に)映像化されているのをみても、何とも思わないか、ムッとする。遅い PC を使って作業しているとなおさらだ。

けれど、外を見たときとかみたいに、Not Watch nor Look, But see な状況の時、それが音楽化されてみたら、ちょっと楽しいかもしれない。

そういうことを企んだのだった。

1.2 実用性 -音楽的要素がなくても-

音楽的要素が無くしても、視覚情報を音声に変換するという試みは、例えば後天的に目の見えなくなってしまった人への、わずかな手助けとなる可能性を秘めている。

音を位置と色で対応付けることができれば、ある程度の視覚補助になろう。もし、その目的で特化させたものを作るに至れば(今回の開発では到底無理そうだが)、日常的な交通の用に供することすら出来るかもしれない。

1.2.1 こういった、直接の役に立たない物作りの土壤

物作りとは、役に立つと万人が思うものを作ることのみであろうか。

ある日、東京工業大学の大岡山キャンパスで開かれた Make: Tokyo Meeting に行ってきました訳だが、半球のボールにたくさんつまみがついていて、大人の科学の付録のシンセとマイコンの接続をわざわざ信号でなくてサーボモータでつまみを動かすことで音作りを視覚化したり、テスラコイルで音楽を再生してみたり、そういうた、電気、光、音の融合したような分野での趣味の工作と研究の成果が大小様々に配置されることは製作者(高専生、大学生、院生を中心として高校生も含む)がそばにいて、お互い、または見学者と話しているといった雰囲気であった。それに触発もされて、私はこうしてこの文章を書いている。

幸い、先述のものを個人で、実現している例はまだない様なので良かった。

Make:や、筑波大学主催の、産学共同で行われた prosume といった企画は、どういう目的なのだろうか。

そのままでは、役に立たないかも知れないものを作る必要がある場所があるのだろうか。

引用開始***参考 1

PROSUME 2010 は、個人でもここまでできる！をコンセプトとした、クリエーターのためのイベントです。電子工作、機械工作、科学実験、クラフト、音響、映像パフォーマンスなど、基本的に個人やサークル等で開発、作成したものであれば何でも出展可能です。またこれらクリエーターを応援してくれる企業の出展も募集します。

Prosume(プロシューム)とは、produce(生産) + consume(消費)を組み合せた造語です。1980年にAlvin Toffler氏の「第三の波」の中で、生産者であり、消費者であるという意味の、Prosumerが提唱されたのが最初とされています。消費者の個別のニーズに応じた製品は、消費者自身が作り出すというもので、それを可能にするのが従来技術の低コスト化と、身近になった最新技術です。キーワードは、オープンソース、ラピッド・プロトタイピング、そしてパーソナル・ファブリケーションです。

これは、prosumeに実際に出展した人との会話を通して得た、私の解釈であるが、”産業

資本主義の末期的構造”として一般に問題提起されている構造-すなわち生産者（企業）というブラックボックスから与えられたものを「欲しい！」という動機のみで消費し、”需要が供給を生み出しているのではなく、「開発された量産製品」の存在が需要を生み出している”に対する一つの答えだと思っている。

具体的には、こんなものがあったら良いなというものを作り、そこから情報やアイディアを（消費者として、かつ製作者として）シェアすることにより、他人の作った物より斬新なもの、より複雑なものを目指すことが容易になり、PROSUME にあげられた”従来技術の低コスト化”と、情報技術の高度な発達によってそれが加速されたということである。

もし、一般的「消費者であり生産者ではない」人への応用が可能になるものをそこから「純粋な生産ブラックボックス」である企業が見いだし、量産し、人類の生活をわずかでも向上できれば、”Prosumer”として本望なのではないか。

1.2.2 キーワードとしての オープンソース ラピット・プロトタイピング パーソナル・ファブリケーション

「第三の波」が 1980 年に書かれていることは非常に衝撃的である。なぜか？それは、上にあげた様々な革命がほとんど無名の赤ん坊でしかなかった時代だからだ。

オープンソースとは一体何か

参考 2 として、最も普及しているオープンソースライセンス及び哲学のうちの特に歴史があるものとして、FreeBSD ライセンスと GPL(GNU General Public License) を示す。
The MIT License

Copyright (c) *year* *copyright holders*.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH

THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

GNU GENERAL PUBLIC LICENSE

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; a) The work must carry prominent notices stating that you modified it, and giving a relevant date. b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its chapters, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

参考3にしめすのは sourceforge 社(オープンソース文化のハブとして活躍し続けるプログラム開発支援やテクノロジ系ニュースコミュニティのホスト会社として世界的に有名)にホストされている OpenSourceGroupJapan による非公式訳である。

The MIT License

Copyright (c) [year] [copyright holders]

以下に定める条件に従い、本ソフトウェアおよび関連文書のファイル（以下「ソフトウェア」）の複製を取得するすべての人に対し、ソフトウェアを無制限に扱うことを無償で許可します。これには、ソフトウェアの複製を使用、複写、変更、結合、掲載、頒布、サブライセンス、および/または販売する権利、およびソフトウェアを提供する相手に同じことを許可する権利も無制限に含まれます。上記の著作権表示および本許諾表示を、ソフトウェアのすべての複製または重要な部分に記載するものとします。

ソフトウェアは「現状のまま」で、明示であるか暗黙であるかを問わず、何らの保証もなく提供されます。ここでいう保証とは、商品性、特定の目的への適合性、および権利非侵害についての保証も含みますが、それに限定されるものではありません。作者または著作権者は、契約行為、不法行為、またはそれ以外であろうと、ソフトウェアに起因または関連し、あるいはソフトウェアの使用またはその他の扱いによって生じる一切の請求、損害、その他の義務について何らの責任も負わないものとします。

入手した (GPL でライセンスされた) ソースコードの (一字一句正しい) 原本をどんな形でも配布できる。そのとき、GPL ライセンスされたソフトウェアであることを、明らかに分かるように示さなければならない a) 作品には、あなたが作品を改変したということと、改変に関連した日時を記述した告知を目立つように載せなければならない。b) 作品には、それが本許諾書と、下記第 7 項に従って追加された条件すべての下で公開されていることを記述した告知を目立つように載せなければならない。この条件は、上記第 4 項における「告知をすべてそのまま保全」するための条項を改変する。c) 作品の全部分を、総体として、コピーを所有するに至った人全員に、本許諾書の下でライセンスしなければならない。そこで、本許諾書は、本許諾書第 7 項に基づく適用可能な追加的条項のすべてとともに、作品全体に、すなわちその全部分に、それらがどのようにパッケージされているかに関わらず 適用されることになる。本許諾書は、これ以外のやり方には作品をライセンスする許可を与えないが、あなたが本許諾書以外で別途許可を得ていた場合 には、それによって得られた許可まで無効とするものではない。d) 改変された作品が対話的なユーザインターフェースを有する場合、それらのインターフェースは『適切な法的告知』を表示しなければならない。ただし、『プログラム』に元々『適切な法的告知』を表示しない対話的なインターフェースがある場合、あなたの作品で表示するようにする必要はない。

ここで特徴的なのはコピーされ、改変された work(作品) も GPL でライセンスされることであるが、オープンソースして重要なのはそこではない。

オープンソースの全体的な一般論として、だれかが先行して何かをオープンソースとして作った場合、それをいかなる形でも利用でき、その成果をオープンにすれば誰かに利用してもらえるのだ。

これは開発期間の短縮及びモチベーションの向上につながる。

また、ソースコードを作品と呼んでいるのも、Prosume や Make:的な感性に置ける、芸術としての技術的作品という側面を強く押し出しているとも言えよう。

しかし、この GPL というライセンスすらも、1989 年になって初版が策定されるのだ。 Alvin Toffler 氏すごい。

オープンソースとソースコード文学に関しての議論だけ進めてたくさんもの論文がかけよう。しかし、ここでは割愛させていただく。

少なくとも、ここで述べたことが、”Prosumer” という「第三の波」を生み出す土壤になっていることは明らかである。

ラピット・プロトタイピング

rapid prototyping とは高速な試作である。企業単位でしか買えないような大仰な装置を使って試作品を時間をかけてデバッグして世に送り出してるようでは、個人では当然できない上に、大企業が圧倒的に有利で、また、設計時間も長くなってしまう。しかし、計

算機科学の驚くべき進化によって、またはコンテンツ産業の貪欲な大規模化によって、たかだかウェブサイトを閲覧する程度のつもりで個人が購入したコンピュータが一昔前のスーパコンピュータの様な圧倒的な速度を誇っている。このことは、設計工程に関して、廉価なPCで、短期間でCADを使用して、アナログ回路、論理回路、コンピュータソフトウェア、機械、模型など多くのものをシミュレートし、設計できる様にした。このことは個人の休日の工作のできる幅を押し広げつづけている。

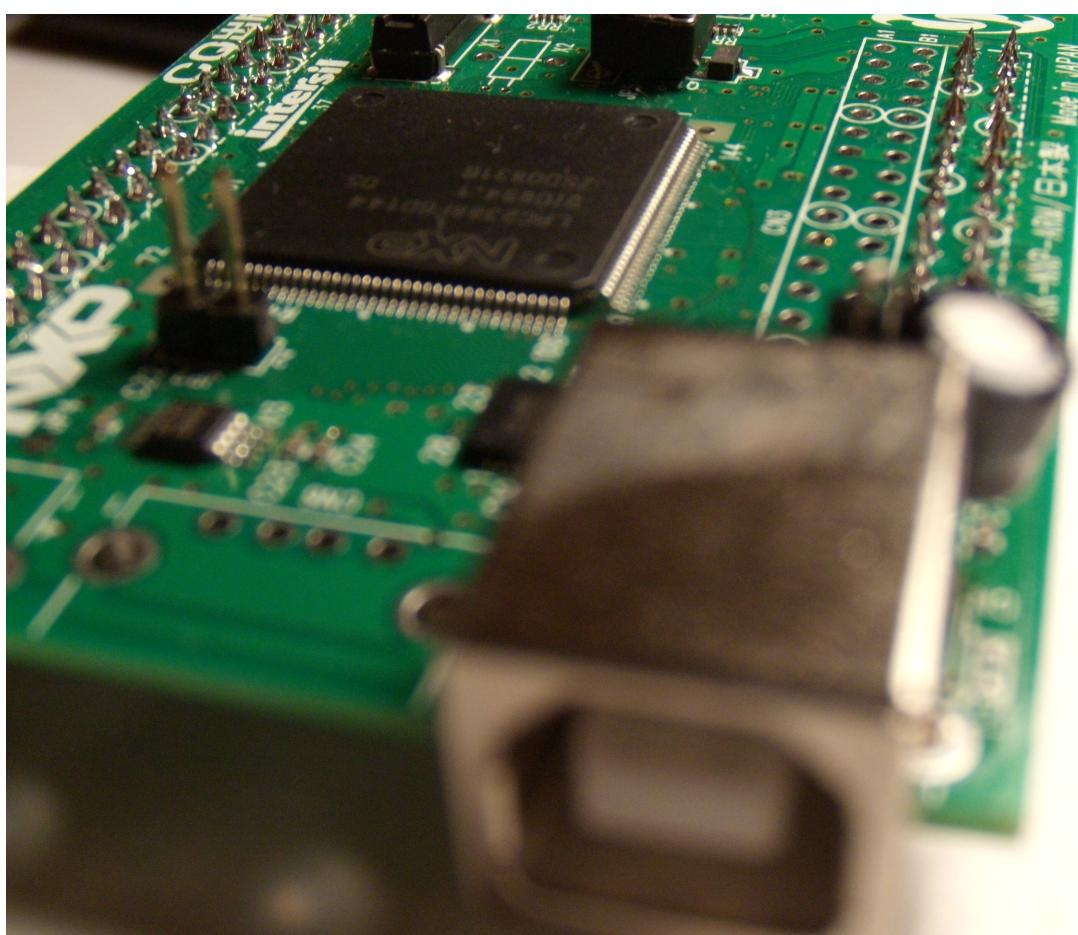
パーソナルファブリケーション

この単語は個人でのものづくりをさしている。これは、上で述べてきたことが一般化し、個人の製作能力を飛躍的に上昇させ、様々なことを個人にとって可能にしてきたという事の当然の帰結として、出てきた概念である。個人のファブリケーションの持つ可能性が無視できなくなってきたとき、”Prosumer”の先駆者が自然に発生し、あとから「第三の波」がやってきたといって過言ではないだろう。

当然の事ながらインターネットの高速化、リッチソフトウェア化、利用人口の増大は、その「波」を明らかに加速させている。

今回の純粋な技術的なヒントの半分ほどはインターネットを介して入ってきてる。特に、半導体のデータシートを個人が入手できるというのは革命である。

図 1.1 接写ターゲット基板



第2章

内部仕様について

2.1 最初期の仕様

以下に全体構成を示す。(中心となるチップ / メーカ / ベンダ)

- カメラ (OV7670 / Omnipixel / aitendo) 2980 円
- マイクロプロセッサ (LPC2388 / NXP semiconductor / CQ 出版) 1980 円 (本込み)
- CPLD (EPM570T100C5 / Altera / optimize) 1600 円 (未実装)
- SRAM (CY7C1041DV33 / Cypress / 秋月電子通商) 500 円

上記の構成の理由について述べる。

2.1.1 カメラ -目玉-

映像を電気信号に変える素子。

2.1.2 プロセッサ -大脳-

ワンチップ・コンピュータ。プログラムを実行でき、Flash EPROM がプログラム領域なので、いくらでも書き換えができる。

2.1.3 CPLD -脊髄脳髄その他-

プログラマブルなロジック IC 群。プロセッサとおおきくちがうのは、”手順”をプログラムするのではなく、回路そのものを組み込むことである。動作のシミュレーションができる。クロックに同期しない動作が可能があるので、低速であっても、メモリ操作などができる。後述の HBE や LBE を素早く制御するのに、上記のプロセッサでは役不足である。

2.1.4 SRAM -海馬。中期記憶-

SDRAM はコマンドがある。RAM に一度カメラからのコンスタントな情報をプールするのは CPLD になるので、実質制御は CPLD になろう。すると、ロジック IC でしか処理出来ない速度で、SRAM 以上の複雑なコマンドを送るのは、初めての RAM 制御と

しては酷であると判断した。

よって SRAM を使用する。

SDRAM は価格が比較的安く、入手性も高いが、今回は簡単に実現できる事を優先させていただく。

さらに、なぜこのチップを選択したかというと、実はカメラのデータアウトプットと深い関係がある。

カメラのデータバス幅は 8bit である。しかし、1pixel のデータは 16bit、1 データ集合を 2 回に分けて送信してくるのである。

そこで、16bit 幅の普通の RAM で簡単に制御しようとすると、実際のカメラからのデータの 2 倍のデータ容量が必要となる。

必要となる RAM 容量 =

アドレス長 × 1 アドレスに於けるデータ幅 = $vetical \times horizontal \times 2 \times 8bit$

16bit 幅の RAM を同じ方法で (1 回受信し 8 シフトして次回受信分と足すという大変な作業無しで) 扱うときの必要とされる RAM 容量 =

アドレス長 × 1 アドレスに於けるデータ幅 = $vetical \times horizontal \times 2 \times 16bit$

となり、2 倍の RAM 容量が必要なのは自明だ。

しかもこの方式では、プロセッサが RAM から値をとるという簡単な行為のために 2cycle も浪費しなければならない。残念である上に、連結処理に (Green が上位 bits と下位 bits に分かれているため) 時間を割く必要がある。手間も割く。しかし、この SRAM のデータシートを読むと 16bit の SRAM でありながら、下位 bit、上位 bit のゲートの開け閉めが任意に行えるため、上位 bit に関する値だけ取り込みたい! といったあと、下位 bit だけまた同じアドレスに取り込みたい! といったわがままが効くのだ。

しかも、上位も下位も開けておけば 16bit いっぺんに読み込み書き込み可能なのである。

It's fits into just my needs!! 引用開始。資料 2*****

To write to the device, take Chip Enable (\overline{CE}) and Write Enable (\overline{WE}) inputs LOW. If Byte Low Enable (\overline{BLE}) is LOW, then data from IO pins (IO0 through IO7) is written into the location specified on the address pins (A0 through A15). If Byte High Enable (\overline{BHE}) is LOW, then data from IO pins (IO8 through IO15) is written into the location specified on the address pins (A0 through A15).

To read from the device, take Chip Enable (\overline{CE}) and OutputEnable (\overline{OE}) LOW while forcing the Write Enable (\overline{WE}) HIGH. If Byte Low Enable (\overline{BLE}) is LOW, then data from the memory location specified by the address pins appear on IO0 to IO7. If Byte High Enable (\overline{BHE}) is LOW, then data from memory appears on IO8 to IO15. See the "Truth Table" on page 9 for

a complete description of read and write modes.

引用終了。また、文中に参照しろ！と書いてある Truth Table(真理表)

Truth Table

\overline{CE}	\overline{WE}	\overline{OE}	\overline{BHE}	\overline{BLE}	Input/Output	Mode
H	X	X	X	X	HighZ	Deselect
L	X	X	H	H	HighZ	PowerDown
L	H	L	L	L	Data Out (IO0-IO15)	Output Disabled
L	H	L	H	L	Data Out (IO0-IO7); IO8-IO15 in High Z	Read
L	H	L	L	H	Data Out (IO8-IO15); IO0-IO7 in HighZ	Read
L	H	H	L	L	HighZ	Output Disabled
L	H	H	H	L	HighZ	Output Disabled
L	H	H	L	H	HighZ	Output Disabled
L	L	X	L	L	Data In(IO0-IO15)	Write
L	L	X	H	L	Data In(IO0-IO7); IO8-IO15 in High Z	Write
L	L	X	L	H	Data In(IO8-IO15); IO0-IO7 in High Z	Write

2.2 現在の仕様

2.2.1 SRAM の破棄

SRAM を使用することに関して以下のデメリットがあった:

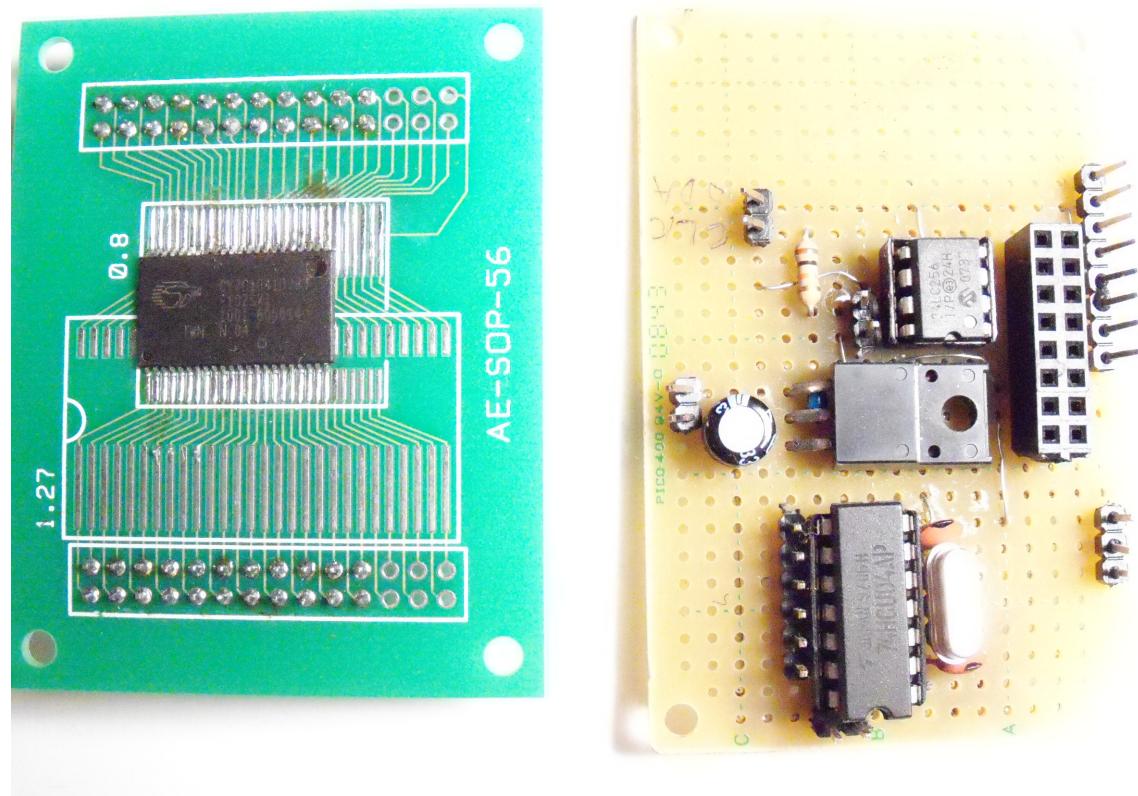
- SRAM 自体のサイズは小さいが、変換基板が無用に大きく、配置が手間である
- ピン数が多いために、ユニバーサル基板上に配置し、ピンを通じてコードで配線をすると、安定性が著しく低下する
- データ入出力層が SRAM に対してマイコン及び CPLD につながることとなり、短絡防止の保護回路を別に必要とする

また、後述する通り、作成に伴って SRAM を必要としなくても（信号データを一度 RAM 展開しなくても）データ抽出ができるようになったため、いらなくなってしまった。

2.2.2 音源デバイス

どんな音源が簡単に使えるかというのを念頭に探した。すると、音楽理論などに明るくない僕が低コストかつ簡単に扱える音源は PSG だと気づいた。これは古典的な PC やゲーム機に搭載されていた簡易な音源 LSI である。込み入ったレジスタいじりをせずにすむ。

図 2.1 左が使用を中止した SRAM, 右は後述する I^2C のテスト用 EEPROM(24LC256) と発振回路



第3章

開発環境

この開発環境の整備が、ファストプロトタイピングや、とくにパーソナルファブリケーションの大きなネックとなる。これが改善されなければ、今後の組み込みにおける、オープンソース環境での開発において大きな損失を招くであろう。

以下に、どのような開発環境の整備が簡易にでき、また、何が難しいか、不可能かを述べる。

3.1 オープンソースデスクトップ環境において

このサブセクション内で、一般に、ソフトウェアと言った場合、オープンソースであることとする。

また、デスクトップ環境とは、いわゆる PC での環境のことを指し、ここではホストと同意義とする。

『ホスト』は組み込み開発の『ターゲット』に対する用語とし、組み込み開発の母艦というニュアンスを含む。

プロプライエタリ(私企業や個人、政府が保有し、OSS でない)ソフトウェアの場合は配布企業とプロプライエタリであることを明示する。

オープンソースデスクトップ環境(以降 OSSDE)として、以下の 3 つの OS を対象とした。

- Debian GNU/Linux(Debian project)
- OpenSuSE (OpenSuSE project)
- OpenBSD (OpenBSD)

また、オープンソースではないデスクトップ環境(以降 non-OSSDE)として、以下の 2 つの OS を対象とした。

- Windows 7 リリース候補版(マイクロソフト)
- Windows XP professional(マイクロソフト)

OSSDE として使用した OS はいずれも UNIX 互換プラットフォームであり、強力なコンソールを備える（その他にも十分に便利な X windows system を備える）。対して、windows では、コマンドプロンプトの機能は貧弱であるといわざるを得ない。

これは Cygwin(後述) などといったソフトウェアを利用することにより補完されるものの、パッケージ管理などといった大きな OSSDE 側のアドヴァンテージを解消することはできない。

パッケージ管理システムとは、UNIX 互換の環境などにおいて存在する機能で、OS(正確には開発/配布者) ごとに存在する。

たとえば Debian の場合、dpkg というソフトウェアがパッケージを管理し、インストール、依存関係の確認、リムーブ、バージョン管理を行う。パッケージとは、ソフトウェアについて、作者による定義ファイルやバイナリファイル、場合によってはソースコードなどインストールに必要な情報が入った圧縮ファイルである。一般にパッケージ管理システムはこれだけの機能しかないわけではない。

Debian では apt というソフトウェア群が、あるソフトウェアの導入に伴って、依存関係の洗い出しを行い、必要なソフトウェアを自動でインストールし、不要になれば自動的に削除する。

これにより、さまざまなもの（開発が私企業でなく、個人が参加しているため本当にさまざまな）ライブラリが存在するにもかかわらず、使用者はライブラリを意識することなくただ目的のソフトウェアにありつけるし、開発者としても依存するライブラリとそのバージョンについて定義しておけば、利用者は必ずこのライブラリを持っているとみなせるので、自分が開発したいソフトウェアのみに専念することができる。

今回の組み込みに対する開発では、ホスト側にとっては一利用者に過ぎないことになるが、これらの OSSDE の特徴は、まさに prosume という風潮に適したシステムだと言える。

また、強力なコンソールを備えている事は、開発において、自動化したい部分をスクリプト化することで能率をあげることもでき、また、Windowsを利用してマウス操作でディレクトリの移動やソフトウェアの起動を行うよりも素早く、なおかつ開発に関係の無いことに患わされることが無いことを意味する。

これについてはさらに後述する。

まず、開発環境の導入の最初の一歩はインストールである。開発環境のインストールについて、比較する。

3.1.1 ARMマイコンのコンパイラ

一般に、ARMマイコンにおいて、もっとも安定しているコンパイラは arm-none-eabi であるとされている。

これは CodeSourcery 社によって開発されているもので、GUI(グラフィカルユーザインターフェース: ウィンドウとアイコンによって提供される環境)を備えた製品と、それに対するサポートを販売しているが、コンソールで扱うツールキットに関しては無料で提供している。このツールキットは、フロントエンドが GNU toolchain であり、これは非常に有名かつ有用なツールキットであり、オープンソースソフトウェアである。

また、ほかの ARM コンパイラとして、以下のものと比較対象にした。自分のソースコードにもっとも馴染んだものが arm-none-eabi であり、大きな性能差を感じたわけではないことをここに記しておく。

- arm-elf-eabi ビルドに失敗。
- arm-eabi
- arm-elf

上2つは、Linux 上でのみ確認できた。また、arm-elf については OpenBSD のみで試してみた。

3.1.2 ARMマイコン LPCxxxx の書き込み

PC上で動作するソフトウェアと違い、組み込みでは、書き込まなければ動かない。そこで、書き込みソフトウェアが必要となる。

このソフトウェアとして LPC21ISP を使用した。これもオープンソースなソフトウェアである。また、成熟していてもはや直すところが無いが、英語の情報が多いため、作成者への恩返しとして、とりあえずプロジェクトの概要などの日本語に翻訳する作業を行った。

このような翻訳ボランティアも一種のオープンソース活動である。

3.1.3 AlteraCPLD の開発環境

Altera の開発環境、Quartus II は windows,Linux どちらでも動作する。過去のバージョンでは Solaris で動作するものもあるようだ。

逆に言えば、マイコンと違って、CPLD の開発環境は完全なブラックボックスであるので、windows のみのサポートでは不満があったのだと思われる。

Quartus II にかぎらず、Altera や Xilinx などの、PLD メーカがサポートしている Linux ディストリビューションは、Redhat 系と言われるもの

中でも、RedHatEnterpriseLinux(RHEL) と CentOS(RHEL からサポートとロゴなどの著作権物を除いた OSS な OS)、および OpenSuSE のみである。

ところが、Debian で動作させることができ一般に可能であり、使う上で、特に問題があるわけではない。唯一ライブラリの問題があるが、古いライブラリを持ってきて展開すれば使用できる。

また、OpenSuSE では、書き込み機である USB-Blaster の認識が不安定で、一般的な問題ではないようだが、私が使った上では、まったく使い物にならなかった。むしろ、サポート対象ではないはずの Debian での使用の方が快適である。それには以下のようにする。

```
#echo /dev/usbfs /proc/bus/usb usbfs devmode=0666 0  
0 >> /etc/fstab
```

3.1.4 その他

Make

UNIX 系 OS でのコマンドラインによる快適な操作と相まって、Make というのはすばらしいソフトウェアである。

Make というのは、いくつかのソースファイルがあった場合に、差分のみコンパイルすることができて、高速にソフトウェアをビルドできる。また、オプションによって動作を変えることが出来るため、役に立つ。

git

git というのはバージョン管理システムである。Apple の TimeCapsule などは有名だが、同じように、一里塚過ぎたら(過ぎなくともかまわないが)Commit することによって、あらたなバージョンとして登録される。もし、前のバージョンに戻したい場合も、またブランチ(枝分かれ)する場合も、破壊的なことを一切せずに要求をこなすことができる。

これは OSS にとって非常に大きな意味を持つ。なぜなら OSS では、ある人が創っているソフトウェアを元に、別のソフトウェアを派生させる、ということが日常的に行われる。そのためにソースコードを公開しているといって過言ではない。バグフィックスであれば元のソフトウェアに取り込まれるかもしれないし、別の方針をこれから歩んでいこうということもある。たとえば便利さやリリースの速さを優先したり、セキュリティを優先したりするためにブランチしたプロジェクトなど枚挙に暇がない。

他にも、たとえば操作ミスや思い直した事があって論文の身に何かあっ

ても、落ち着いて過去のバージョンのファイルを取り出すことができる。

3.2 Windows 環境において

3.2.1 ARM の開発環境

- IAR workspace(nonOSS:IAR systems)

IAR systems の製品であるが、ライブラリの名前が嫌で、しかも見た目がすきじゃなかった。使い辛い。
使用可能な規模に制限がある。

3.2.2 AlteraCPLD の開発環境

Quartus II をなんら問題なく使うことができるが、フォントが汚い。これは windows に起因する問題で修正ができる。

このように、windows で使う開発環境として問題になることはほとんどなく、素晴らしいように見えるが、git を使うにも、ちゃんとしたコマンドラインを使うにも、不思議なほど手間がかかる。

3.3 git と github について

git というのは、分散バージョン管理システムである。つまり、そのリポジトリが存在する場所が 1箇所に限らないということである。それではバージョン間で矛盾しないのか？それは、中央サーバにただ一つのマスタリポジトリを置くことによって解決される。

つまり、git にはサーバにリポジトリをアップ(以降 push とよぶ)する機能がついている。そこで、無料で利用できる、データの損失の心配の無いクラウドに push できればこれほど安心なことはない。

さて、その要求を満たすサービスがある。それは github という。ではどうやって使うのかというと、rsa 暗号鍵を用意した後、サイトにアクセスし、リポジトリ作成をリクエストする。

その後、以下のようにコマンドを実行する。

```
Debian% cd 論文
Debian% gedit textest.tex&
[1] 16628
Debian% ls
Makepdf  textest.aux  textest.dvi  textest.log  textest.out  textest.pdf  textest.tex
```

新しいリポジトリの作成

プロジェクト名

説明 (optional)

メモ
注意: すでにGitHub上にある別の! プッシュしたい場合は、ここで新 fork してください。

ホームページのURL (optional)

このリポジトリにアクセスできるのは誰ですか? (あとから変更することもできます)

誰でも [\(公開リポジトリについてさらに学ぶ\)](#)

[プランをアップグレードすれば、非公開のリポジトリも作れるようになりますよ!](#)

[Blog](#) [サポート](#) [研修](#) [求人情報](#) [ショップ](#) [連絡先](#) [API](#) [ステータス](#)
 © 2011 GitHub Inc. All rights reserved. [利用規約](#) [プライバシーポリシー](#) [セキュリティ](#)

Powered by the
 Cloud Computing

日本語 English Deutsch Français Português (BR) Русский 中文 その他すべての言語を見る ➔

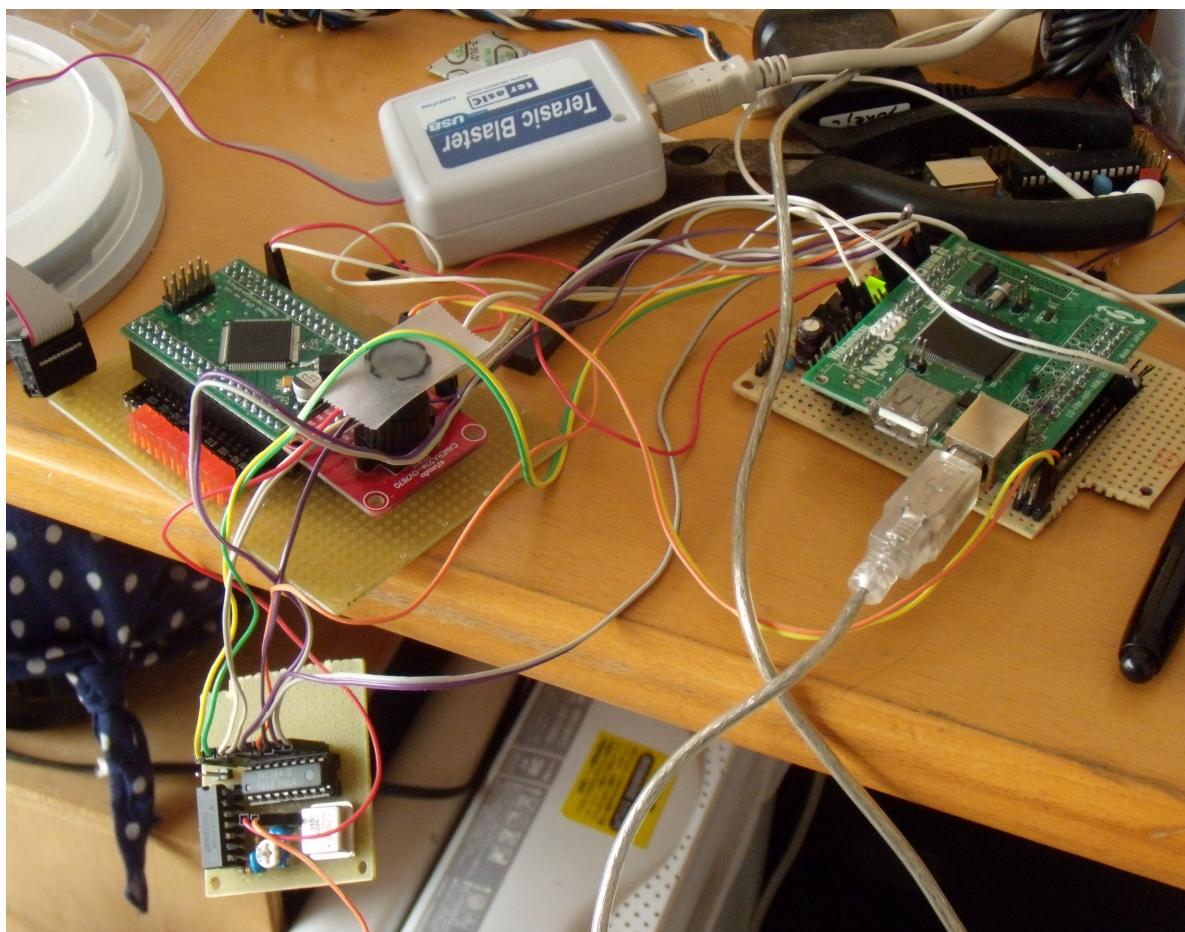
```
textest.tex~  textest.toc
Debian% git init
Initialized empty Git repository in /home/ioriveur/論文/.git/
Debian% git add .
Debian% git commit -a
[master (root-commit) 10c4ed0] First Commit
 8 files changed, 950 insertions(+), 0 deletions(-)
  create mode 100755 Makepdf
  create mode 100644 textest.aux
  create mode 100644 textest.dvi
  create mode 100644 textest.log
  create mode 100644 textest.out
  create mode 100644 textest.pdf
  create mode 100644 textest.tex
  create mode 100644 textest.tex~
  create mode 100644 textest.toc
Debian% git remote add origin git@github.com:iori-yja/Report.git
Debian% git push origin master
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (10/10), done.
```

```
Writing objects: 100% (11/11), 110.86 KiB, done.  
Total 11 (delta 2), reused 0 (delta 0)  
To git@github.com:iori-yja/Report.git  
 * [new branch] master -> master
```

気がつくべきなのは、これは、既に公開され、オープンソースソフトウェアとなったということだ。

ソフトウェアを創れるならば、OSSとして育っていくことはまったくコストでは無いということが分かっていただけだと思う。

図 3.1 混迷を極める開発室



第Ⅱ部

開発

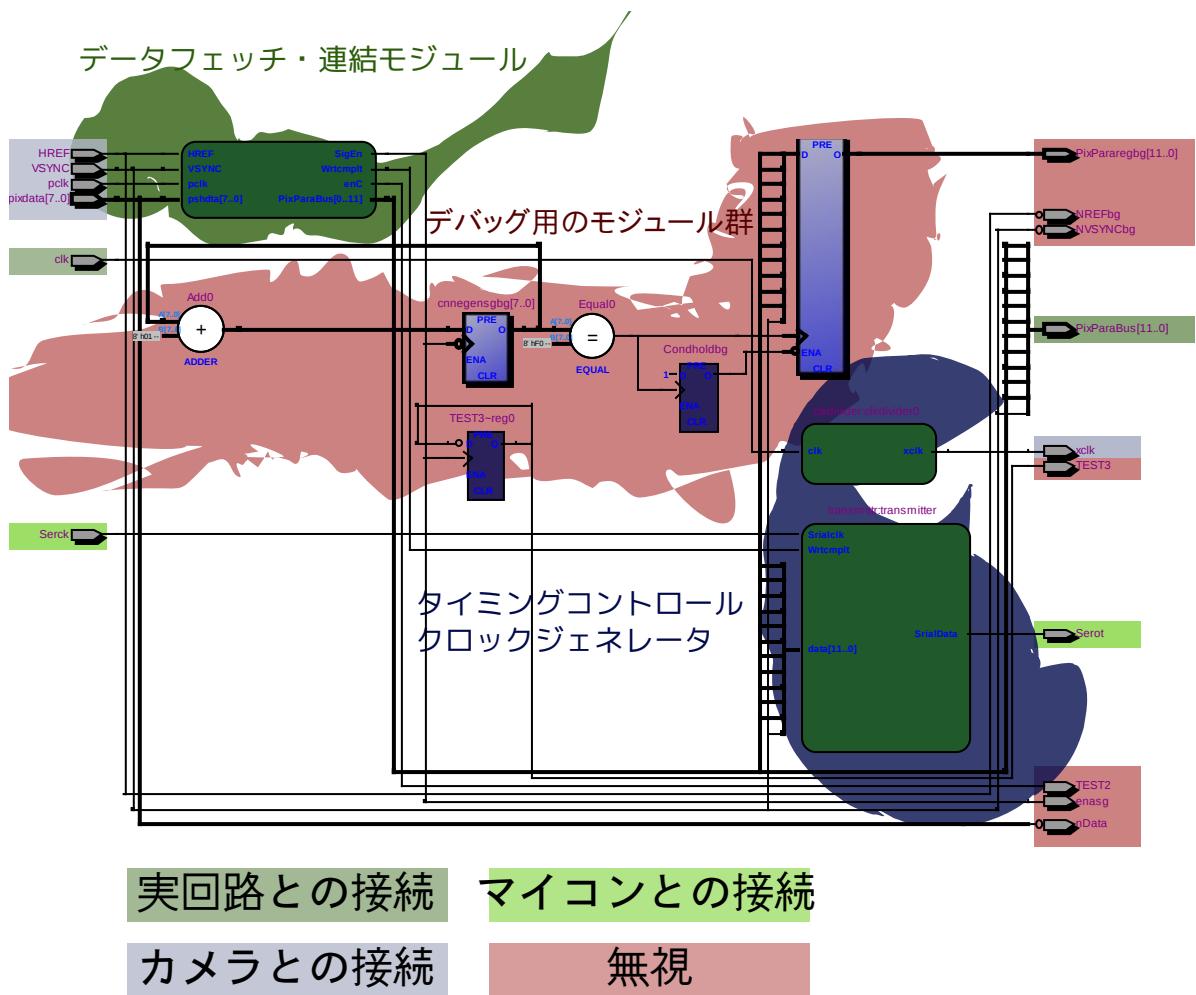
第4章

Verilogを書く

4.1 モジュールの構成

このモジュールの全体像を示す。

図 4.1 モジュールの全体図:拡大できます



4.2 clkdivider(分周器)

まず初めに、もっとも簡単なモジュールを作成する。

今回、MAX II の基板に搭載した水晶発信器が 40MHz である。今回の用途ではカメラの動作周波数は動作定格内であれば遅い方がより信号バスがノイズに強くなるため、10MHz に分周してカメラのクロックソースに入力する。分周するメリットは、さらに、分周することによって、クロックのデューティ比を 50% に近くすることが可能である。

つまり、水晶発信器が必ずしも理想的な出力をしているとは限らないが、少なくとも 1 サイクルの長さはほとんどかわっていない、とするとき、分周器のクロック出力が H になっている時間と L になっている時間は、クロック入力のデューティ比に関わらず、どちらも入力 1 サイクル分であるから、出力波形は高速動作をするプロセサのクロックソースとして適したものとなる。

リスト 4.1 分周器

```

1 module clkdivider (input clk, output xclk);
2   reg [1:0] t_count = 0;
3   always@ (posedge clk) t_count[0] <= ~t_count[0];
4   always@ (posedge t_count[0]) t_count[1] <= ~t_count[1];
5   assign xclk = t_count[1];
6 endmodule

```

今回は、2 つのレジスタを設け、入力クロックの立ち上がりエッジに、`t_count[0]` を反転させる。すると、`t_count` が立ち上るのは 2 入力クロックサイクルに 1 回となる。

`t_count[1]` は `t_count[0]` が立ち上がりクロックになる時に反転するため、結果として 4 入力サイクルにつき 1 サイクルで `blink` することがわかる。よってこのモジュールは 4:1 分周器になっている。

4.3 データ抽出レイヤ

4.3.1 ピクセル選択モジュール

リスト 4.2 特定のピクセルのサイクルのときに立ち上がる、一部省略

```

1 module TimingManager(
2   input VSYNC,
3   input HREF,
4   input pclk,
5   output reg Sig_En,

```

```

6     output enC);
7 reg [8:0] line = 0;
8 reg [8:0] foo = 0;
9 always @(posedge HREF or posedge VSYNC)begin
10   if(HREF == 1'b1)line <= line +1'b1;
11   else line = 8'h00;
12 end
13 assign enC = (line==9'h30 ) ? 1'b1 : 1'b0;
14
15 always @(posedge pclk)begin
16   if ( HREF&enC ) foo <= foo + 1'b1;
17   else foo <= 7'b0000000;
18   if(foo[0]&foo[8]&HREF)begin
19     Sig_En<=1'b1;
20     foo <= 7'b0000000;
21   end
22   else begin
23     Sig_En<=1'b0;
24   end
25 end
26 endmodule

```

このモジュールでは、カメラが生成する制御信号をすべて取り扱っていて、ほかのモジュールと分け合っていたりしないため、タイミング系で不具合が発生した場合はここに問題がある、と比較的迅速に問題の特定が出来る。まず、簡単に OV7670 の制御信号について説明する。

CMOS カメラ OV7670 モジュールの外に出ている制御信号は

- VSYNC
- HREF
- pclk

のたった 3 つである。また、カメラ内に止まっている、普段引き出されないものに HSYNC があるが、基本的に使わずにすむようだ。

フレーム (1 つの画像) の最初の行の最初のピクセルを送出する前に、VSYNC が立ち上がり、3 行分の時間ホールドされる。その間、有効なデータは流れきていないが、なんの値が出ているのかは未だ不明であり、タダの不定値である。

その後、VSYNC が立ち下がり、実データが送出される条件が一つ揃うようになる。

次に、17 フレーム分の時間、制御信号のバスに変化は無い。その後、HREF が上がると、ようやく実データが送出され始める。1 行がおわると、一度 HREF が立ち下がり、144px 分やすむ。

その間、データシートによると、実データはキューに入れられて待っているらしい。

まず、このソースの目につくのは、`pclk`立ち上がりエッジで起動する以下のカウンタ

リスト 4.3 ピクセルごとのカウンタ

```

1 always @ (posedge pclk) begin
2     if ( HREF&&enC ) foo <= foo + 1'b1;
3     else foo <= 7'b0000000;
4     if(foo[0]&foo[8]&HREF) begin
5         Sig_En<=1'b1;
6         foo <= 7'b0000000;
7     end
8     else begin
9         Sig_En<=1'b0;
10    end
11 end

```

であろう。しかし、このカウンタは、中の `if` 文の通り、`HREF` と `enC` が H でなければ動かない。また、`if` を満たさない場合は 0 にリセットされる。`HREF` が L のとき、無効なデータがやってくる。だからこのデータを手っ取り早く見ないようにするためにには、カウンタをリセットして、0 のときに少なくともデータを見ない様にすればいい。

ただ、`enC` とはなんだろうか。これは、

リスト 4.4 行のカウンタ

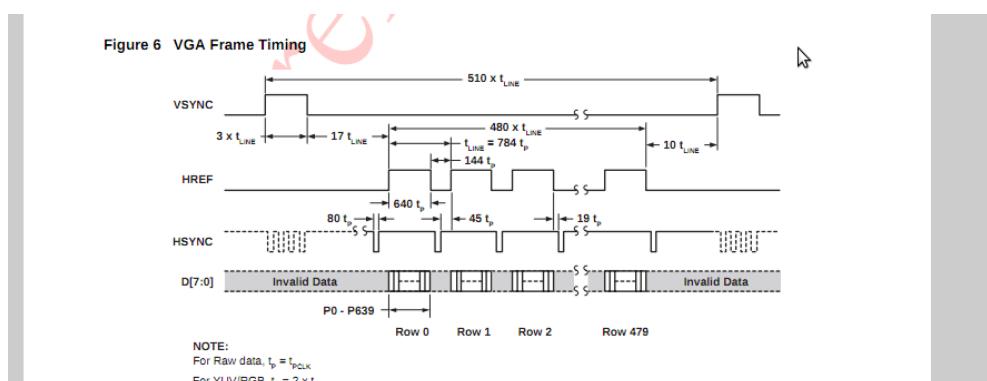
```

1 always @ (posedge HREF or posedge VSYNC) begin
2     if(HREF == 1'b1)line <= line +1'b1;
3     else line = 8'h00;
4 end
5 assign enC = (line==9'h30) ? 1'b1 : 1'b0;

```

でアサインされているワイヤが `enC` であり、`line`(行数)の値によってドライプされる。つまり、特定の行の時のみピクセルカウンタが起動される様

図 4.2 OV7670 のデータシートによるタイミングチャート



になっているわけだ。

実を言うと、この設計にたどり着くまで、もっと別の設計でやろうとしていた。最初は一つの always でやっていたが、implicit latch を生成しますよ！っていうワーニングや、always@文で呼び起こされたのに何も値を変えないことがあります、といったワーニングが出ていて、調べても何の事だかよく分かっていなかった。

もし、この設計を思いついでいなかったら、ずっと気持ち悪いままだったが、これによってワーニングが消えたどころか、使用するロジックエレメントの数もグッと減ったため、きっと裏でなにかあって、always@文が何のレジスタの値も変えない可能性があると、無駄なラッチを生成することがある様だ。

ということがわかったが、具体的なバッドノウハウもリゾルブノウハウもほとんど出回っていないという状態であるため、今後のワーニングメッセージの改善や、教本、ノウハウ集として有用なブログなどの登場を期待する必要があると感じた。

このためには、半導体ベンダ側の努力のみならず、企業のような、ノウハウが閉鎖的にこもりがちな顧客だけ、という状態から脱して、それこそ手芸やバレエのごとく、幅広い一般人が FPGA や CPLD などを使うようになる必要も、当然あるだろう。どれくらい未来になるか分からないが、IT がもっと陳腐化すればいずれ来る未来であると思う。

4.3.2 ピクセルデータフェッチ、連結モジュール

リスト 4.5 データコミッタ

```

1 module Dtacutcmmit(
2   input SigEn,
3   input [7:0] pshdta,
4   output WrCmplt,
5   output reg [11:0] popdta
6 );
7 assign WrCmplt = ~SigEn;
8 reg [3:0] redpx;
9 always @(posedge SigEn)begin
10   redpx [3:0] <= pshdta [3:0];
11 end
12 always @(negedge SigEn)begin
13   popdta [11:0] <= { redpx [3:0], pshdta [7:0] };
14 end
15 endmodule

```

このモジュールは、SigEn の両エッジで働く。

そもそも、カメラのデータが 2 クロックサイクルに渡って分割して送信されるため、特定の瞬間のバスデータをフェッチすれば良いと言う訳ではないため、少々複雑になる見込みであった。ところが現状、大分シンプルにまとまっている。

SigEn と言うのはピクセル選択モジュールである TimingManager のものと同じである。

これは、上で述べた変更によって pclk の立ち上がりエッジでのみ動作するようになったため、Sig_En のエッジ動作時には、ピクセルデータが流れている pshdta は必ず保証されている。これは上のタイミングチャートを見てほしい。pclk 立ち下がりエッジのときにデータ値が変化し、逆に立ち上がりエッジの前約 15ms 以上と後 8ms が保証されている。(ただし 24MHz 動作時、今回はさらに 2.4 倍遅い)

4.4 データ転送モジュール

これについては、独自シリアル転送プロトコルの項で後述する。

4.5 シミュレーション

4.5.1 シミュレータの意義

つぎに、モジュールが書けたら、マイコンでは実機で動かすことが一般的だが、PLD ではシミュレータにかけることが多い。

理由としては、まずマイコンよりもさらに可観測性が低いため、予期せぬ動作をした場合にデバッグが非常に大きな仕事となってしまうということ、C 言語でマイコンに対して書いたものとくらべ、人間にとって可読性が高いものではないため、まったく想定外のことを書いてしまっている可能性がより高いということ、シミュレーションを行う上では、マイコンの、特にペリフェラルでの動作と比べて Verilog の方が可視性があるということなどが挙げられるはずだ。

シミュレータは今回、Modelsim-altera starter edition(nonOSS:MentorGraphics 社) を用いた。

シミュレーションモデルも HDL を使って表記する。

論文の構成上、他のモジュールより後に書いたが、すべてのモジュールを書き終えてから回路検証をするのではなく、一つのモジュールを書きながらテストモジュールを書いて逐一懸賞をしていくことが何よりも大切だと思われる。

この方法であれば、検証対象モジュールとシミュレーションモデル、双方とも、バグを書いた時点でそうと分かる上、比較的見つけることが容易であり、最も効率がいいと思われる。

また、ModelSim でのコマンドは tcl であるため、セミコロンで連接することにより一気にビルドから波形を追加してシミュレーションまで終えることができる。

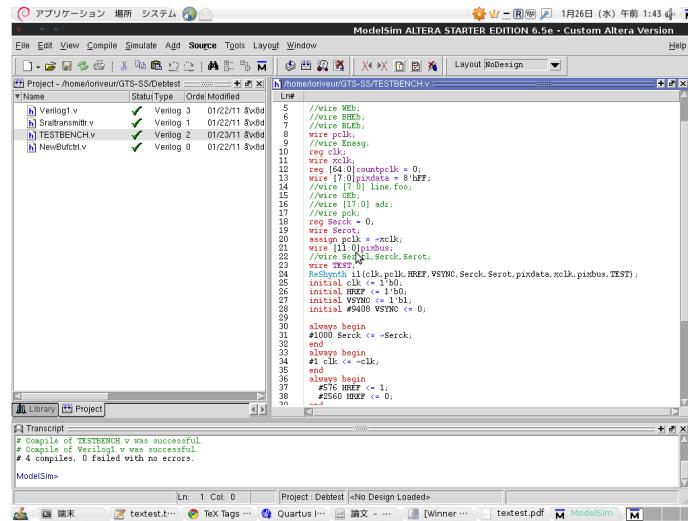
リスト 4.6 一気にシミュレーションまで行う 1 行スクリプト

```
1 vsim work.TEST;add wave -r /*;run 1600000;
```

もし、add wave しなければ、シミュレーションが行われた後もタイミングチャートをみることは出来ない。また、ModelSim の GUI は貧弱で、それ

のみに頼るには不足している印象を受けた。

図4.3 ソースコードを開いたModelSim Altera Starter Edition

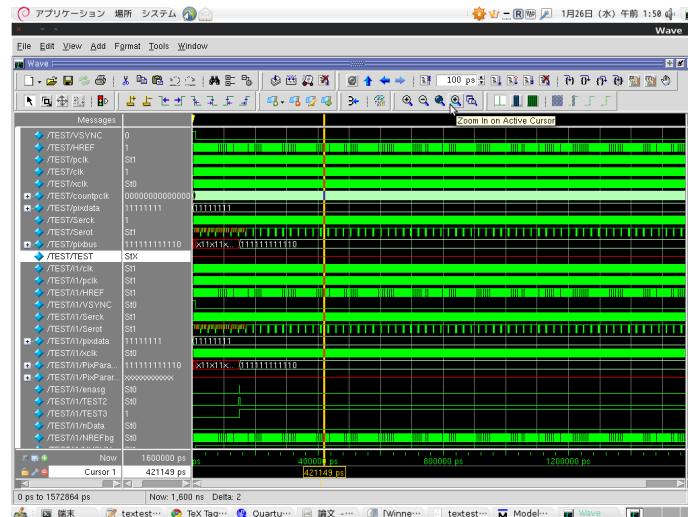


```

ModelSim ALTERA STARTER EDITION 6.5e - Custom Altera Version
File Edit View Compile Simulate Add Source Tools Layout Window
Project - home/tomeur/GTS-SS-Debtest
Project - home/tomeur/GTS-SS/TESTBENCH.v
Name Status Type Order Modified
[h] Verilog.v ✓ Verilog 3 01/22/11 0x0d
[h] SlaveTransmitter.v ✓ Verilog 1 01/22/11 0x0d
[h] TESTBENCH.v ✓ Verilog 2 01/23/11 0x0d
[h] NewBufct.v ✓ Verilog 0 01/22/11 0x0d
Line
5 //wire WEB;
6 //wire HED;
7 //wire LD;
8 wire pclk;
9 //wire Enasg;
10 wire psel;
11 wire nclk;
12 wire [15:0]countpclk = 0;
13 wire [7:0]line_foo;
14 wire [15:0]line;
15 wire [15:0]adr;
16 wire [17:0]addr;
17 reg Serck_0;
18 wire Serck;
19 assign Serck = ~clk;
20 assign Serck_o = ~clk;
21 wire [11:0]pixbus;
22 //wire Sel1_Serck.Serck;
23 wire [11:0]pixdata;
24 RegShift #1(CLK,pclk,HREF,VSYNC,Serck,Serck,pixdata,nclk,pixbus,TEST);
25 initial CLK <= 1'b0;
26 initial VSYNC <= 1'b0;
27 initial WSTRG <= 1'b1;
28 initial #4000 WSTRG <= 0;
29
30 always begin
31 #2500 Serck <- ~Serck;
32 end
33 always begin
34 #1 CLK;
35 end
36 always begin
37 #576 HREF <- 1;
38 #2560 HREF <- 0;
39 end

```

図4.4 シュミレーションを終え、タイミングチャートを表示するModelSim Altera Starter Edition



4.5.2 シミュレーションモデル

他の、実回路で動かすための Verilog と違い、シミュレーションではさまざまな便利な命令を使うことが出来るが、私が思うに、これらを一まとめにして Verilog と呼んでいる現状が、現在の簡易的では無いが実装されるべき構文の、実回路での論理合成可能性がいまだに上がらない今まである元凶であるように見える。

ちなみに、このシミュレーションモデルでは、コメントアウトした部分も、わざと掲載している。それは、消してしまうのではなく、コメントアウトすることによって、また後で検証対象モジュールのバスを観測したくなつたときにすぐ戻せるようにするという技を明示的に示すためだ。

リスト 4.7 回路検証のためのモジュール. 実回路には組み込まれない

```

1 module TEST;
2 reg VSYNC;
3 reg HREF;
4 //wire WEb; これら辺は外付けの SRAMを使う予定だったころの名残りが残っている
5 //wire BHEb;
6 //wire BLEb; always @(posedge pclk)begin
7   if ( HREF&enC ) foo <= foo + 1'b1;
8   else foo <= 7'b0000000;
9   if(foo[0]&foo[8]&HREF)begin
10     Sig_En<=1'b1;
11     foo <= 7'b0000000;
12   end
13   else begin
14     Sig_En<=1'b0;
15   end
16 end
17 wire pclk;
18 //wire Enasg; これはピクセル選択モジュールのイネーブルシグナルの観測用
19 reg clk; //40MHzのマスタクロック
20 wire xclk;
21 reg [64:0]countpclk = 0;
22 wire [7:0]pixdata = 8'h46;
23 //wire [7:0] line,foo;バスの観測用
24 //wire CEb;
25 //wire [17:0] adr;
26 //wire pck;
27 reg Serck = 0;
28 wire Serot;
29 assign pclk = ~xclk;
30 wire [11:0]pixbus;
31 //wire Serctl,Serck,Serot;
32 wire TEST;
33 ReShynth i1(clk,pclk,HREF,VSYNC,Serck,Serot,pixdata,xclk,pixbus,TEST);

```

```

34 initial clk <= 1'b0;
35 initial HREF <= 1'b0;
36 initial VSYNC <= 1'b1;
37 initial #9408 VSYNC <= 0;
38
39 always begin
40 #1000 Serck <= ~Serck;
41 end
42 always begin
43 #1 clk <= ~clk; //マスタクロック生成
44 end
45 always begin
46 #576 HREF <= 1;
47 #2560 HREF <= 0;
48 end
49 always begin
50 #1589952 VSYNC <= 1;
51 #9408 VSYNC <= 0;
52 end
53 always @ (negedge pclk)
54 if (VSYNC) countpclk <= 0;
55 else
56 countpclk <= countpclk + 1;
57 always @ (pixbus) $display ($time, "pix:%d countpclk is %d", pixbus, countpclk);
58 endmodule

```

このテスト用モジュールは、主に

- クロックソースの流し込み(マスタクロック、シリアルクロック、ピクセルクロック)
- カメラの制御信号の生成、流し込み
- バスデータの仮想的な提供
- 特定の観測しているワイヤに変化があった場合のモニタリング

を行っている。

たとえば

```

1 always begin
2 #576 HREF <= 1;
3 #2560 HREF <= 0;
4 end
5 always begin
6 #1589952 VSYNC <= 1;
7 #9408 VSYNC <= 0;
8 end

```

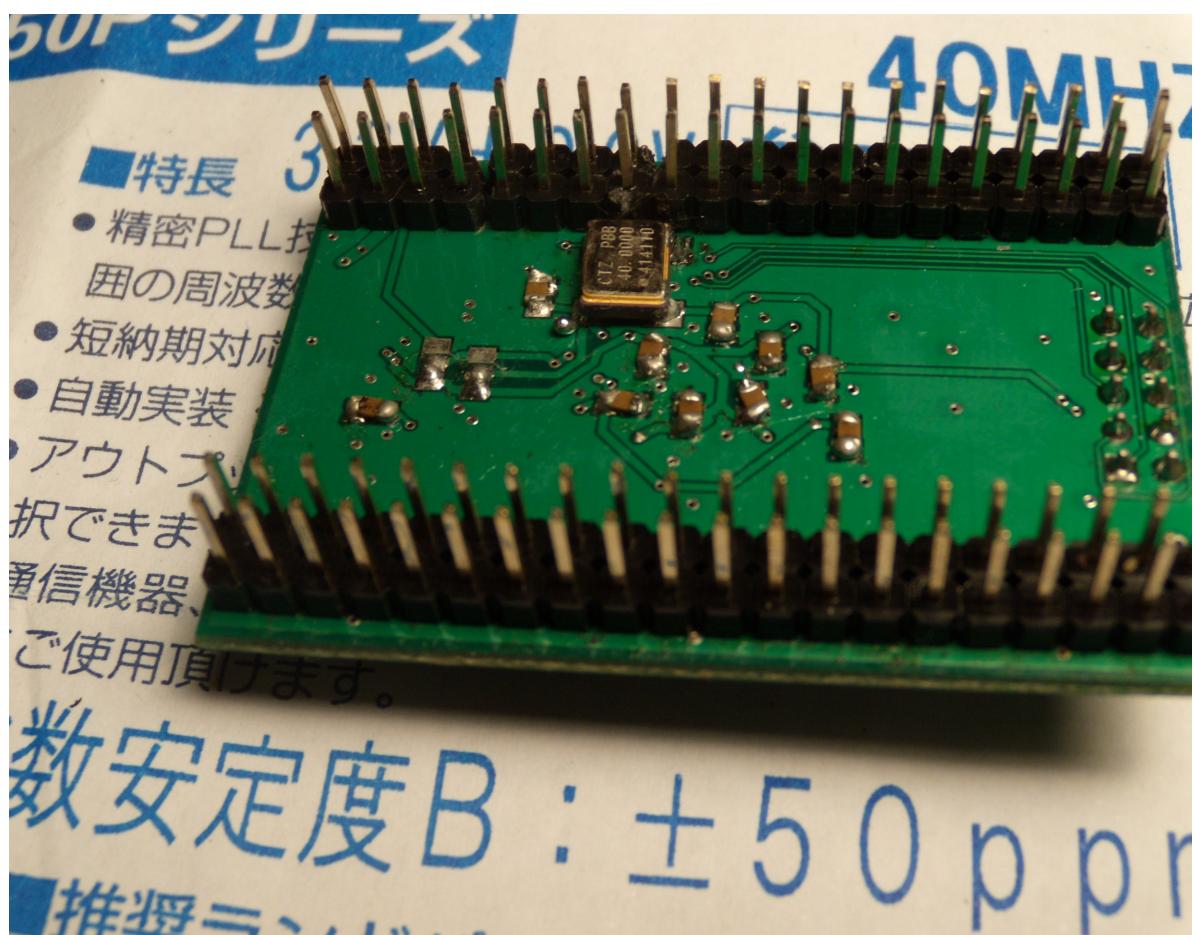
こここの下りでは、カメラから送られてくる制御信号を提供しているが、これはカメラのデータシート（

[http://aitendo2.sakura.ne.jp/aitendo_data/product_img2/product_img/camera/CAMERA30W-V7670/OV7670_DS_\(1_4\).pdf](http://aitendo2.sakura.ne.jp/aitendo_data/product_img2/product_img/camera/CAMERA30W-V7670/OV7670_DS_(1_4).pdf)

) を参考することによって比較的早くできあがっていた。

逆に、テストモジュールを書きづらいのは、仕様が決まってない自分の回路にたいしてであったりするため、悔れない。

図 4.5 CPLD ボードの裏側の水晶発振子とそのデータシート



第5章

I²C セットアップ関数の開発

I²C ドライバが必要だったため、作成した。割り込みを使わない *I²C* で必要となる動作が、2番ポートにおいてすべてカヴァーされているため、ライブラリとして公開する。

ただ、このコードを書いたのがずっと昔で、今回だましだまし使っていたので、とても汚いコードとなっている。

そんな汚いコードを公開するのか？という疑問もあるが、そもそもなにもないよりマシであるという考え方もある。

5.1 データ送出関数

```

1 void i2csender(int Continue, unsigned int Data, int Keta){
2 //int i2cStatus;
3     if(Continue==0){
4         I22DAT = Data;
5         I22CONSET |= 0x04;
6         I22CONCLR = 0x08;
7         while(I22STAT!=0x18 && I22STAT!=0x20);
8         FIO2PIN1 = 0x2;
9         if(I22STAT==0x20){
10             printf("No such device%4x\n",Data);
11             i2cErr = 0x22;
12             return;
13         }
14     }else {
15         for(;Keta != 0;Keta-=8){
16             I22DAT = (Data & 0xFF);
17             I22CONCLR = 0x28;
18             while(I22STAT!=0x28);
19             Data = Data >> 8;
20         }
21     }
22 }
```

5.1.1 動作

この関数の動作は、ライトサイクルとして最初であった場合には、まず割り込みをクリアし、マスタモードに入らないようにセットアップし、また、”データ”としてアドレスを送出させるところから始まる。その後、デバイスから返答があるか、どこからも返答が無いと分かるまで待ち、終了処理を行う。

ライトサイクルとして最初では無かった場合は、int 値を、下 8bit ずつ送出する。

5.1.2 問題点

これを FreeRTOS 上で動作させると、初回呼び出し時の 3 回に 2 回、しかも周期的に OS 全体を落とすという大きな問題を抱えている。

5.2 シーケンシャルデータ受信関数

```

1 int i2creader(int size, int Adr, int registernumber){
2     char Loopy;
3     upper:
4         I22DAT = (Adr + 1);
5         I22CONSET |= 0x04;
6         I22CONCLR = 0x08;
7         printf("Status is %x\n", I22STAT);
8         while(I22STAT != 0x40 && I22STAT != 0x48) printf(".");
9         if(I22STAT == 0x48){
10             printf("no acknowlege(%x)\n", I22STAT);
11             i2crestart(2);
12             goto upper;
13         }
14         while(registernumber != (size+1)){
15             I22CONSET |= 0x04;
16             I22CONCLR = 0x28;
17             while(I22STAT != 0x50 );
18             vTaskDelay(10 / portTICK_RATE_MS);
19             printf("Adress%x,\tData%x", registernumber++, I22DAT);
20         }
21         printf("done.....");
22         fflush(stdout);
23         I22CONCLR = 0x0C;
24     }

```

これは、SCCB レジスタに対して、シーケンシャルリードを 0 番地から最大値である 0xca までを一期に読み取る関数である。あまり美しいコード

にはなっていない。

5.2.1 バイトリード

サイズを 1 にすれば、バイトリードとなる。本質的に、シーケンシャルリードとバイトリードに違いは無いのだ。

5.3 OV7670 全リード関数

OV7670 の SCCB レジスタをリードする関数である。

```

1 void SCCBread(int subadr, int size, int NumByte){
2     i2cstart(2);
3     // vTaskDelay(300 / portTICK_RATE_MS);
4     i2cErr = 0;          //i want to read! So, start.
5     i2csender(0,0x42,8); //Hey, ?
6     if(i2cErr==0x22){
7         printf("fatal");
8         i2cstop(2);
9         return;
10    }
11    i2csender(1,subadr,8);
12    // printf("Adress read request at%4x OK\n\n", subadr); //0x03byte?
13    i2crestart(2);
14    i2creader(size,0x42,NumByte);
15    i2cstop(2);
16    printf("Debu331g\n");
17 }
```

一つ目のコメントアウトは、安全性のために最初いれておいたものだが、無くても動くらしいと分かったためコメントアウトした。i2cstart(*intport*) 関数はそのままなのでここでは割愛する。

二つ目のコメントアウトは、はずすと冗長な出力となる。

これを見るとわかるとおり、リクエストの送信とデータの受信を一括して行う。

5.4 OV7670 バイトライト関数

```

1 int SCCBwrite(int subadr, int size, int Data ){
2     i2cstart(2);
3     i2cErr = 0;
4     i2csender(0,0x42,8); //Hey, ?
5     if(i2cErr==0x22){
6         printf("fatal");
7         i2cstop(2);
8         return -1;
}
```

```
9      }
10     i2csender(1,subadr,8); //3byte?
11     i2csender(1,Data,size);
12     i2cstop(2);
13     return 0;
14 }
```

1 バイト書くのは非常に簡単だ。

通信スタートが成立した場合、サブアドレスとデータを順繕りに送信して
通信をとじればいい。

第 6 章

SCCB での OV7670 コンフィギュレーション

6.1 OV7670 の注意点

SCCB というのは、*I²C* というシリアル通信のスーパセットで、実質 *I²C* と同じ方法で通信が出来るとされている。ただし、実際にやってみるといくつか違いがあった。

まず、*I²C* 接続の EEPROM での実験をしめす。今回使用したのは、microchip 社製 24LC256 である。

これに、バイトライト、バイトリード、シーケンシャルリード、シーケンシャルライトを試したが、バイトライトの後に wait をはさむ必要が無く、シーケンシャルライトも問題なく行えた。

これに対して、OV7670 では、書き込みの後に wait を必要としていてなおかつ、データシートには記述が無いものの、実験によるとシーケンシャルライトは出来ないようだ。このことから、完全な *I²C* メモリインターフェースを積んでいるというよりは、あくまで *I²C* インタフェースをつかった通信が出来ますよ、という程度に考えておくべきだ。

また、ただの RAM のつもりでヴェリファイプログラムを書くべきではない。理由は後述する。

6.2 OV7670 用のレジスタセットアップデータの整理

<http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/drivers/media/video/ov7670.c> ここに、OV7670 用の linux ドライバがあるが、非常に不思議な事に、データシートには always "0" などとかかれた場所や reserved 領域に書き込んだりしている。ともかく、これをどれほど信用すべきなのかんがえあぐねているところに、FPGA で遊んでみるさんが既に試していた。

リスト 6.1 ov7670_set_up.ttl

```

1 ; OV7670<81>@set up macro
2 ;
3 ; delay set
4 pause_ms =0
5 ; delay for debug
6 ;pause_ms =2000
7 ;
8 ; ADR=0x12 WDATA=0x04
9 sendln 'W1204'
10 sendln
11 mpause pause_ms
12 ;
13 ; ADR=0x40 WDATA=0xd0
14 sendln 'W40D0'
15 sendln
16 mpause pause_ms
17 ;
18 ; ADR=0x8c WDATA=0x02
19 sendln 'W8C02'
20 sendln
21 mpause pause_ms
22 sendln 'W703a'
23 sendln
24 mpause pause_ms
25 sendln 'W7135'
26 sendln
27 mpause pause_ms
28 sendln 'W7211'
29 sendln
30 mpause pause_ms
31 sendln 'W73f0'
32 sendln
33 mpause pause_ms
34 sendln 'Wa202'
35 sendln
36 mpause pause_ms
37 sendln 'W1500'
38 sendln
39 mpause pause_ms
40 sendln 'W7A20'
41 :
42 :
43 :
44 :

```

これは Teraterm かなんかのスクリプトで、このままでは使えないで、vim で整形し、そのまま SCCB レジスタに書き込む関数にした。

```

1 SCCBByteWrite(0x1204);
2 vTaskDelay(2 / portTICK_RATE_MS);
3 SCCBByteWrite(0x40D0);
4 vTaskDelay(2 / portTICK_RATE_MS);
5 SCCBByteWrite(0x8C02);
6 vTaskDelay(2 / portTICK_RATE_MS);
7 SCCBByteWrite(0x703a);
8 vTaskDelay(2 / portTICK_RATE_MS);
9 SCCBByteWrite(0x7211);
10 vTaskDelay(2 / portTICK_RATE_MS);
11 SCCBByteWrite(0x40D0);
12 vTaskDelay(2 / portTICK_RATE_MS);
13 SCCBByteWrite(0x8C02);
14 :
15 :
16 :
17 :
18 :

```

しかし、その後、関数内に定数を置いてしまうことによる、保守性、可読性、コード密度の低さなどの問題があり、これを解決するために分離することとなった：

リスト 6.2 i2c.c の vi2c タスクの冒頭

```

1 printf("Setup begin[>  ");
2 fflush(stdout);
3 for(regpointer=0;regpointer!=129;){
4     if(SCCBByteWrite(OV7670[regpointer])==-1)continue;
5     regpointer++;
6     switch (regpointer%4){
7         case 0:
8             Loopy='|';
9             break;
10        case 1:
11            Loopy='/';
12            break;
13        case 2:
14            Loopy='-';
15            break;
16        case 3:
17            Loopy='\\';
18            break;
19    }
20    printf("\b\b\b=>%c",Loopy);
21    fflush(stdout);
22    vTaskDelay(10 / portTICK_RATE_MS);
23 }
24 printf("\rData transmition End!\n");
25 fflush(stdout);

```

とし、

リスト 6.3 OV7670userconfig.h の抜粋

```

1 const int OV7670 []={
2 0x1204,
3 0x40D0,
4 0x8C02,
5 0x703a,
6 0x7211,
7 :
8 :
9 :
10 :
```

というヘッダファイルを設けた。これにより、ループを回して書き込むようになったというのと、設定ファイルを別ファイル扱いにできたので、修正が容易になった。

また、これにより、二次的なメリットが得られた。

```

1 Hello, world!1:0
2 203PDIRRegister Reset
3 Setup begin
4 [=====]> \
```

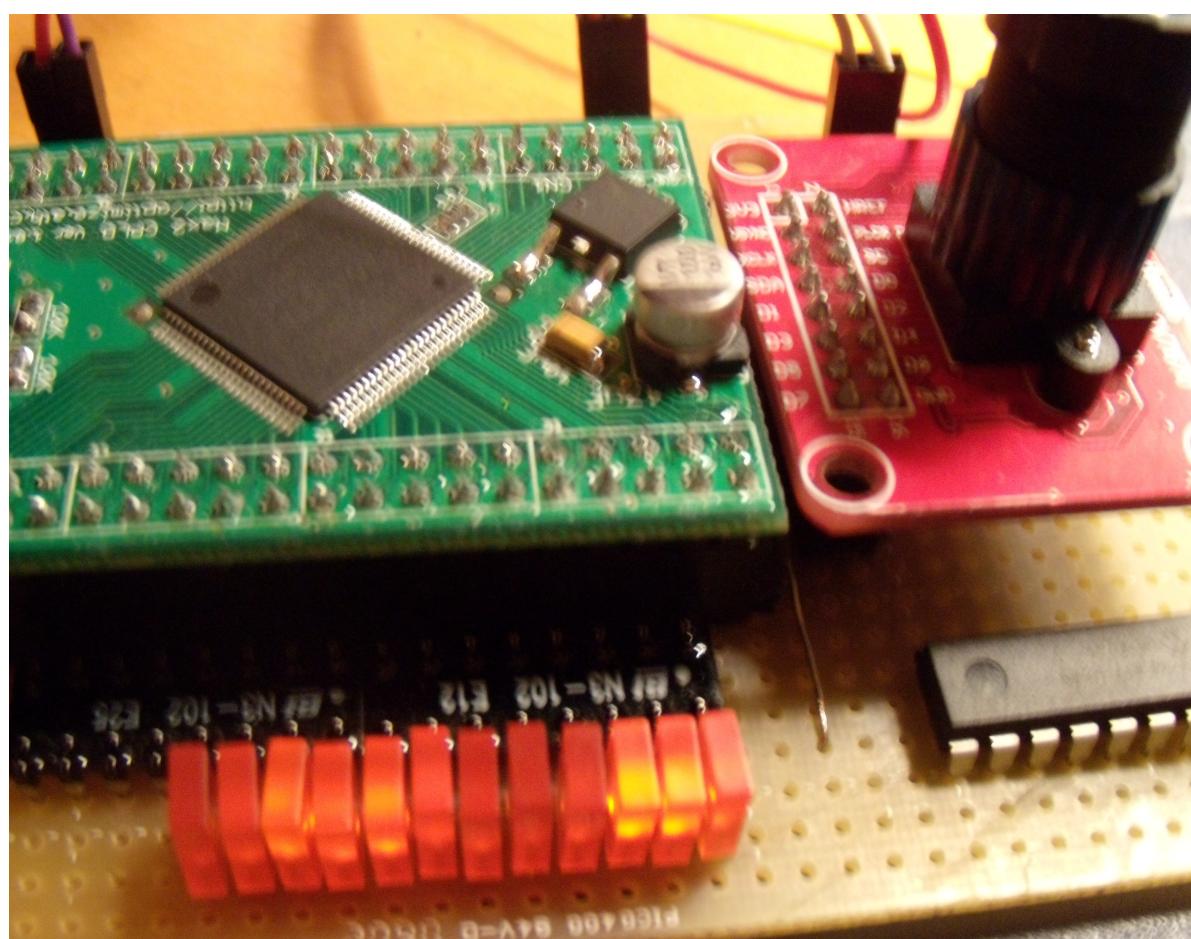
今までのメッセージはデバッグを目的とした冗長な出力がなされていたため、上の様なメッセージとした。簡潔であり、また十分な情報量である。

ひとつのノウハウとしては、バグが存在する間は冗長な出力が好ましいが、安定して動かせるようになったら静かな方が好ましい。

また、ウェリファイは今回行っていない。なぜなら、制御用レジスタにまざって、動的に変化するレジスタ、例えばアイリス値などがあり、また、述べた通り不思議な事だが、なぜかそのようなレジスタにも値を書き込むようになっている。このようなレジスタがすべて分かれれば良いものの、Reserved や Always zero とかかれたレジスタにも値を書き込んでいるため、これらがまた、動的に変化するレジスタであったらデバッグの難度が非常に高いと言わざるを得ず、また、すでに十分に動いていると言う現状から、個人では把握できないものとして考えるべきと言える。

メーカーはデータシートなど、デバイスを使用するに当たってどうしても必要な情報は大口顧客などにのみ提供するのではなく、万人が使えるものにするよう努めるべきである。

図 6.1 カメラ ⇌ CPLD で動いている様子



第 7 章

独自シリアル転送プロトコル

必要なロジックエレメントの数を最小限にとどめるため、できるだけシンプルなシリアル送信プロトコルが必要だったが、探してもなかなか出てこなかつたため作ることとした。

目標は

- 非常に小さいこと
- デバッグが簡単であること
- 実装が難しくないこと

とさだめた。

7.1 トランスマッタ

```

1 module transmitr (
2   input Srialclk,
3   input Wrtcmplt,
4   input [11:0] data,
5   output SrialData
6 );
7 reg [11:0] Datareg;
8 reg Endflg;
9 wire Endtoken;
10 assign Endtoken = Endflg & ~Srialclk;
11 assign SrialData = Endtoken ? ~Datareg[0] : Datareg[0];
12 reg [3:0] ShftCount = 0;
13 always@( posedge Srialclk )begin
14   Datareg[11:0] <= { 1'b0, Datareg[11:1] };
15   ShftCount <= ShftCount + 4'h1;
16   if ( ShftCount == 4'hB )begin
17     Endflg <= 1'b1;
18     if( Wrtcmplt )begin
19       ShftCount <= 0;
20       end
21     end
22   else if ( ShftCount == 4'h0 )begin

```

```

23      Endflg  <= 0;
24      Datareg [11:0]  <= data [11:0];
25      end
26  end
27
28 endmodule

```

7.1.1 何が起きているか

まず、カメラ制御層から特定のピクセルが検出された瞬間を考えてみよう。

その時はまだ、上位バイトのみがバスに流れているため、その時フェッチするのではなく、わかりやすさのため、フェッチモジュールで、12bit すべてのデータを用意し終えた後に立ち上がる Wrtempplt を用意して、それを接続して制御する。

また、たくさんリクエストして、同じデータが重複して送信されても問題が無いため、キュー構造を取る必要が無く、よって制御信号もサイクル終了トーカン以外必要なかった。

サイクル終了トーカンとは、12bit のデータのどこが切れ目か分かるような合図の事とする。これを、データ末尾に制御用コンディションを載せることで表現した。

Endtoken は

```

1 assign Endtoken = Endflg & ~Srialclk;
2 assign SrialData = Endtoken ? ~Datareg [0] : Datareg [0];

```

この様に用いられている。シリアルクロックが L で Endflg(カウンタが 11 を指しているときに発行される) が H のときに成立する。

そして、これが H のときに、シリアル出力が反転する。シリアルクロックがたち下がるときにかならず変化し、また、他のコンディションでは立ち下がりエッジでの変化は行われないので、確実に検出される。

7.1.2 公開と README

別の切り口で後述するが、もしこの様な独自プロトコルなどを作成した場合は README(規模と場合によってはデータシートやチュートリアルマニュアルまで) を作成することは不可欠である。

なぜなら、まだそれが人気になる前であれば、そのプロトコルに熟知している人は全世界であなたしかいないからだ。

もし、他の人が、自分が新しく公開したものを使おうとしても、それがいく

ら素晴らしいものであっても、使い方がわからなければ、せっかく頑張つて創って公開した自分の成果物が人に使ってもらえる機会を失してしまうことになる。使ってみてそれがそもそもつまらないものだ、と思われる以前に、ぱっと見てどんなものか分からなければ、気に留めてすらもらえないかもしれない。それは悲しいことだ。

だからちゃんと README をつくって、なおかつブログで紹介するといい。しばらくマイブームと言うか熱が覚めてしまうまで、連載ができるし、体裁を整えたサイトも非常に有意だが、体裁や整合性にはばかり気を使ってしまって、開発者の意欲を削ぐような事があっては良くない、と思う。

7.2 レシーバ

```
1 void getISSI( void )
2 {
3     int bitshift=0;
4     int il=0;
5     int prv;
6     int crr;
7     vTaskDelay( 1 / portTICK_RATE_MS );
8     printf("start ISSI connection\n");
9
10    while(1){
11        il++;
12        FIO2SET = 2;
13        prv=(FIO2PIN&6)>>2;assign Endtoken = Endflg & ~Srialclk;
14        assign SrialData = Endtoken ? ~Datareg[0] : Datareg[0];
15        printf("up\t%x\n",prv);
16        bitshift = (bitshift<<1)+prv;
17        vTaskDelay(1 / portTICK_RATE_MS );
18        FIO2CLR = 2;
19        printf("prv=%x",prv);
20        crr=(FIO0PIN&6)>>2;
21        printf("Cr=%d",crr);
22        if(crr&2!=prv&2){
23            printf("-%d\t%d\n",crr,il);
24            break;
25        }
26        vTaskDelay(1 / portTICK_RATE_MS );
27        printf("Delaying%d",0.01 / portTICK_RATE_MS );
28    }
29 }
```

実はこのモジュールは完全にうまく動いているとは言えない。

既に提出しなければいけないため、そのように現状を報告しておく。

動いた時には、既に述べたブログ等で紹介するであろうから、もし良かっ

たらみてみてほしい。

図7.1 オシロスコープのプローブ



第8章

SSG ドライバの製作と研究

まず、前提として、FIO4 ポートにパラレルバスをつないだものとする。また、YMZ294 のデータシートはネット上で手に入れるることは出来ない。

8.1 SSG というのは

YMZ294 というのは、MSX の SOUND コマンドと同一のレジスタ操作で発声できる。この音源を PSG という。ヤマハはこれを政治的な理由により SSG と呼ぶがほとんど同じものだ。

PSG のレジスタについては、AY-3-8910 が元となっている。

わたしは H-1 MSX マシンの手引書を読んで学習したが、手に入らない場合は姉妹品 YMZ284 のデータシートをみるとよいとおもわれる。違いは、284 は 294 と比べ、不要だった 2 つの端子が削られている。これによって、動作クロックが 4MHz にしばられる。

逆に言えば、ほかには特に違いが無い。

8.2 ハードウェア層での結線

YMZ294 のパッケージは 18pin で、ピン順序は /WR(write enable) /CS A0(address mode) VDD(5V) SoundOut GND M 4/8(clk select) /IC(reset) /TEST で、のこりは D[7:0] となっている。

秋月で買うといかつい水晶発振器がついてくる。これは 4MHz であり、4/8 は Hi レベルに設定することで 2 分周して 2MHz を生成し、これが LSI 内の fsc として扱われる。このことは秋月のデータシートに明示しておらず、誤解を生みやすいので注意すること。

また、/TEST, 4/8, /IC は内部に 60-260-600k というあまり精度の良くないプルアップ抵抗が入っており、結線しなくてもデフォルトとしてプ

ルアップされ、使えるようになっている。

さらに、その他のピンに於ける Hi レベルが 2.2V からであるので、2.8V や 3.3V で動作するマイコンや FPGA からも容易にアクセスできる。

これは大きな利点である。

/CS は、普通プルアップしちゃ放しでもとりあえずつかえるチップ (ex.SRAM) が多いように思うが、この場合、一回アクセスするごとに上げ下げせねばならないようで、読み込むわけでもないのに /WR も上げ下げせねばならない。ところが、/CS と /WR はショートしても問題無いようだ。

サンプルプログラムも同時に同じアクセスを行っているではないか。これらは隣のピンであり、実際にくっつけて使用しても問題ない。

/WR すら操作せずにうごかしたいとおもったが、それはできなかった。情けないことに、この後僕はとんでもなくしょうもないことに 2 日間も悩むことになった。

それはビットオーダである。こういう単純なミスをなくすためにも、データシートの熟読は欠かせない。ビットオーダというのは、どっちが桁が小さい方で、どっちが桁が大きい方か、ということだ。

8.3 ソフトウェア層での工夫

8.3.1 SOUND 文と同じ機能

まず、原理的な方を書いてみよう。この関数は MSX-BASIC に於ける PSG 発声の”SOUND 文”と同じ文法で使用する。また、機能も等しい。

```

1 void ymzwriteln(int value,int adr){
2     FIO4PIN=0x00;
3     FIO4PIN=adr;
4     FIO4PIN=(0x200+adr);
5     FIO4PIN=(0x100+value);
6     FIO4PIN=(0x300+value);
7 }
```

まず、データシートを見ると、それぞれたくさんステップを踏まないと書き込めないように見える。

しかし、これは単純化できる。タイミング要求のところを読むと分かる。

ようは、A0 も WR も L にし、Data を書き込む。これは 1 サイクルで出来る。

その後 A0 が Low のままにしつつ、/WR を Hi レベルにするとレジスタ選

図 8.1 タイミングチャート

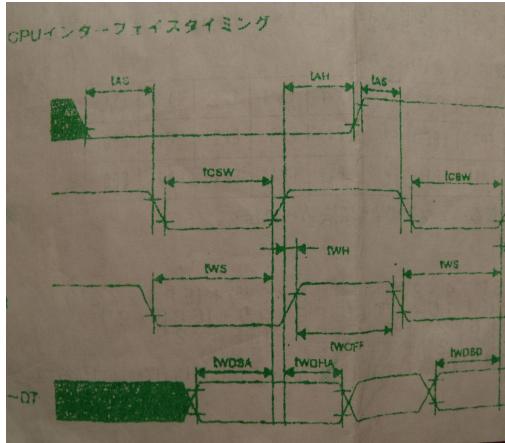


図 8.2 タイミング要求

項目	記号	最小	標準
マスタークロック周波数	f _{ck}	4 or 6	
マスタークロックデューティ	D	40	
リセットパルス幅	UCK	3	
アドレスセットアップ時間	T _{AS}	0	
アドレスホールド時間	T _{WH}	5	
チップセレクトパルス幅	T _{CSW}	30	
ライトパルスセットアップ時間	T _{WS}	10	
ライトパルスホールド時間	T _{WLH}	0	
ライトデータセットアップ時間(アドレス)	T _{WDWA}	10	
ライトデータセットアップ時間(データ)	T _{WDWD}	10	
ライトデータホールド時間(アドレス)	T _{WDRA}	10	
ライトデータホールド時間(データ)	T _{WDWD}	10	
ライトパルスオフ時間	T _{WDFF}	40	

択を行う。

これはブロック図をみれば雰囲気がつかめる。その後 A0 が上がり、/WR が落ちると D[7:0] を同じくフェッチし、/WR が上がることでそのレジスタにデータを書く。これがまさに指定した、一意に定まる場所にデータを書き込むということだ。

8.3.2 更なる改善を求めて

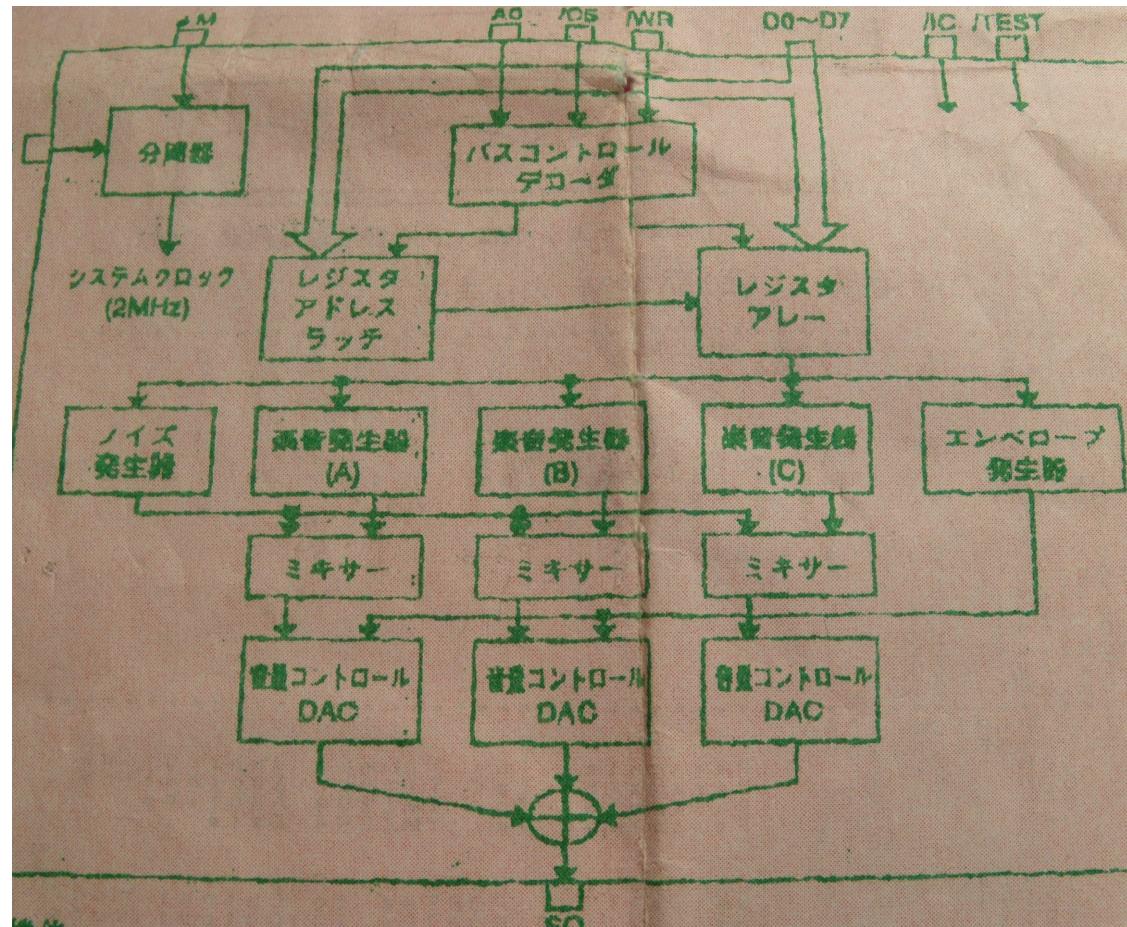
PSG の楽音周波数とエンベロープ周波数は 2 つのレジスタにまたがって書き込まれる。

いちいち楽音やエンベロープに関して、この低レイヤ部分が見えていて、意識して書かなければならないとする不都合である。

```

1 void ymzwriteln(int freq,int adr){
2     int highbyte;
3     FIO4PIN=0x00;
4     FIO4PIN=adr;
5     FIO4PIN=0x200+adr;
6     FIO4PIN=0x100+(freq&0xFF);
7     FIO4PIN=0x300+(freq&0xFF);
8     FIO4PIN=adr+1;
9     FIO4PIN=0x201+adr;
10    highbyte=(freq>>8)&0xFF;
11    FIO4PIN=0x100+highbyte;
12    FIO4PIN=0x300+(freq>>8);
13 }
```

図 8.3 YMZ294 ブロック図 (ところでこの干渉縞は一体...)



2つのレジスタにまたがっている値も、一つのレジスタの用に書き込めたら便利だ。

8.4 制御速度に関する実験

この函数、Delay をはさんでいない。しかしマイコンのクロックは 72MHz である。LPC2388 の GPIO は CPU バスに直接つながっているため、CPU 速度で動く。

しかし、データシートにはこう書いてある：ライトパルスオフ時間 t_{WOFF} 40ns 以上

あれ？おかしい。こういうときはアセンブリを読めば良い。

```

1 .global ymzwritel
2 .type ymzwritel, %function
3 ymzwritel:
4 @ Function supports interworking.
```

```

5 @ args = 0, pretend = 0, frame = 0
6 @ frame_needed = 0, uses_anonymous_args = 0
7 @ link register save eliminated.
8 mvn      r3, #-1073741824@0x-40000000
9 sub      r3, r3, #12288@r3=0x-40003000
10 mov     r2, #0
11 str      r2, [r3, #-3947]@ 0 -400030F6B (FIO4PIN=0 × 00 に対応
12 str      r1, [r3, #-3947]@FIO4PIN=adr; 1clkでアクセス
13 add     r1, r1, #512
14 str      r1, [r3, #-3947]@2clk分でtCSW<30ns<2clk分=28nsあれ?
15 add     r2, r0, #256
16 str      r2, [r3, #-3947]@余談だけどr2つかわずに済むよね

```

またもや 2clk で FIO4PIN=(0 × 100+value); これは t_{WOFF} を満たしていないはずだ。

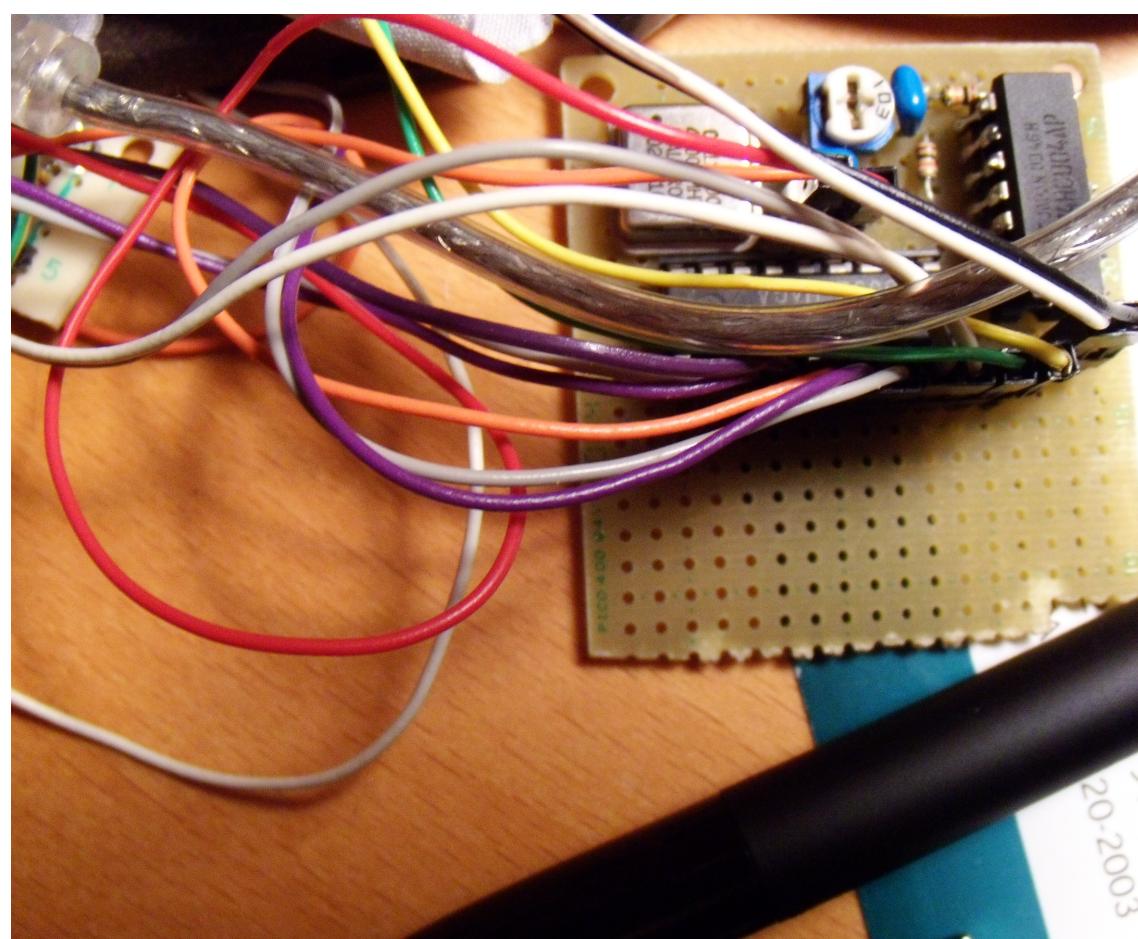
```

1 add     r0, r0, #768
2 str      r0, [r3, #-3947]
3 bx      lr
4 .size   ymzwrite1, ..ymzwrite1

```

データシートの定格中の最速より 1.4 倍も高速にうごかせるのか。
 理由として考えられるのは、古い IC であるため、そのままの仕様で生産するのが難しく、製造プロセスの微細化などがされた、高速に動作可能な IC の製造が可能な近代的な半導体工場に、設備の集約などに伴って、製造が移管されたため、公称性能よりも速度が上がったのかもしれない。

図 8.4 中央が音源 LSI, 右上から中央上がアンプ



第9章

ソースコードを読む技術

9.1 関数値渡しの謎

ARMでの関数の値渡しに疑問を持ち研究を行った。
 FreeRTOS上であるプロセスのサブルーチンが孫ルーチンを呼ぶ部分を
 まず見てみることにする。
 子ルーチン:(思ったより長くなつたので抜粋)

リスト 9.1 子ルーチン

```

1 .LVL64:
2 .loc 1 264 0
3 mov ip, #0
4 .loc 1 265 0
5 mov r0, ip
6 mov r1, #66
7 mov r2, #8
8 .loc 1 264 0
9 str ip, [r4, #0]
10 .loc 1 265 0
11 bl i2csender

```

リスト 9.2 孫ルーチン

```

1 i2csender:
2 .LFB5:
3 .loc 1 134 0
4 .cfi_startproc
5 @ Function supports interworking.
6 @ args = 0, pretend = 0, frame = 0
7 @ frame_needed = 0, uses_anonymous_args = 0
8 .LVL22:
9 .loc 1 136 0
10 cmp r0, #0
11 .loc 1 134 0
12 stmfd sp!, {r3, r4, r5, r6, r7, lr}

```

```

13 .LCFI1:
14 .cfi_def_cfa_offset 24
15 .loc 1 134 0
16 mov r4, r1
17 .cfi_offset 14, -4
18 .cfi_offset 7, -8
19 .cfi_offset 6, -12
20 .cfi_offset 5, -16
21 .cfi_offset 4, -20
22 .cfi_offset 3, -24
23 mov r6, r2
24 .loc 1 136 0
25 beq .L58
26 .loc 1 148 0 discriminator 1
27 cmp r2, #0
28 .loc 1 149 0 discriminator 1
29 ldrne r5, .L71
30 .loc 1 150 0 discriminator 1
31 movne r7, #40
32 .loc 1 148 0 discriminator 1
33 beq .L57
34 .LVL23:
35 .L68:
36 .loc 1 149 0
37 and r1, r4, #255
38 str r1, [r5, #8]
39 .loc 1 150 0
40 str r7, [r5, #24]
41 .L63:
42 .loc 1 151 0 discriminator 1
43 ldr r3, [r5, #4]
44 cmp r3, #40
45 bne .L63
46 .loc 1 152 0
47 ldr r0, .L71+4
48 mov r1, r4
49 bl printf

```

ここで、孫ルーチンの C 言語での記述を見てみよう。

リスト 9.3 孫ルーチン:C 言語

```

1 void i2csender(int Continue, unsigned int Data, int place){
2 if(Continue==0){
3     I22DAT = Data;
4     I22CONSET |= 0x04;
5     I22CONCLR = 0x08;
6     printf(" requesting\x% x", Data);
7     while(I22STAT!=0x18 && I22STAT!=0x20) printf(".");
8     if(I22STAT==0x20){
9         printf(" No such device\x%4 x\n", Data);

```

```

10 i2cErr = 0x22;
11 return;
12 }
13 }else {
14 for(;place != 0;place-=8){
15 I22DAT = (Data & 0xFF);
16 I22CONCLR = 0x28;
17 while(I22STAT!=0x28);
18 printf(" Data \%4x\n",Data);
19 Data = Data >> 8;
20 }
21 }
22 }
```

呼び出し部:引数が 3 つある。 i2csender(0,0 × 42,8); 0,0 × 42(0d66),8 を投げている。もう一度子ルーチンを見る。

リスト 9.4 子ルーチンさらに抜粋

```

1 .LVL64:
2 .loc 1 264 0
3 mov ip, #0
4 .loc 1 265 0
5 mov r0, ip @引数1つめ、何故mov ip, #0をへるのか不明、mov r0, #0ではいけないのか
6 mov r1, #66 @引数2つめ
7 mov r2, #8 @引数3つめ
8 .loc 1 264 0
9 str ip, [r4, #0]
10 .loc 1 265 0
11 bl i2csender
```

ということでどうやら ARM では 3 つの引数を渡すときに r0 ~ r2 に置くということが分かった。本当だろうか。最初にある stmfd sp!, r3, r4, r5, r6, r7, lr と最後にある ldmfd sp!, r3, r4, r5, r6, r7, lr をみる。つまり、スタックに r3 ~ r7 と lr を積んで、作業スペースを確保しているのだろう。

では次に、printf() のような文字列引数の場合、どうやって渡してるんだろうという疑問が沸く。レジスタ 1 こ文字列が収まりきるとは限らないからだ。

リスト 9.5 printf 呼び出し:アセンブリ

```

1 .LCFI1:
2 .cfi_def_cfa_offset 24
3 .loc 1 134 0
4 mov r4, r1
5 .cfi_offset 14, -4
6 .cfi_offset 7, -8
7 .cfi_offset 6, -12
8 .cfi_offset 5, -16
```

```

9   .cfi_offset 4, -20
10  .cfi_offset 3, -24
11  mov r6, r2 @第3引数をr6に退避(r2が0だと0フラグが立つ)
12  .loc 1 136 0
13  beq .L58 @0フラグ立ってたら分岐
14  .loc 1 148 0 discriminator 1
15  cmp r2, #0
16  .loc 1 149 0 discriminator 1
17  ldrne r5, .L71 @I22DATのアドレス
18  .loc 1 150 0 discriminator 1
19  movne r7, #40 @0を第3引数にしてないので、r7に40を入れる。
20  .loc 1 148 0 discriminator 1
21  beq .L57
22  .LVL23:
23  .L68:
24  .loc 1 149 0
25  and r1, r4, #255
26  str r1, [r5, #8] @I22DATにいれます
27  .loc 1 150 0
28  str r7, [r5, #24] @I22CONCLRに0x28をいれる
29  .L63:
30  .loc 1 151 0 discriminator 1
31  ldr r3, [r5, #4] @I22STATをロード
32  cmp r3, #40 @それって0x28?
33  bne .L63 @違う間ループ，3命令かな？(実は.locがようわかっていない)
34  .loc 1 152 0
35  ldr r0, .L71+4 ここが問題のprintf()にたいする引数渡し
36  mov r1, r4
37  bl printf

```

さて、L71+4 が何をさすのだろうか。

リスト 9.6 ラベル L71

```

1  .L71:
2  .word -536346624
3  .word .LC3
4  .word .LC1
5  .word .LC2
6  .word i2cErr
7  .cfi_endproc

```

ふむ。+4 ということは、きっと.LC3 だろう。(4byte = 32bit, まえのが.word だから 32bit でアラインされている).LC3 にはなにがはいっているのであろうか。

リスト 9.7 ラベル LC3

```

1  .LC3:
2  .ascii "Data\\%4x$\\backslash$012$\\backslash$000
3  .space 3

```

これを見るとどうやら .ascii というアライン方法があるらしいが、そんなものは見たことが無い。

ということで、データシートを開く。

https://dl.dropbox.com/u/15570814/051020DDI0100HJ_v61.pdf しかし載ってない。

あれ？とにかく、.ascii っていうアライン方法で、“ ASCIIencoded string ”ってやってやれば、(多分アセンブラーが) 対応するバイナリに変換して LC3 に置いているのだろう。

ところで、\012\000 ってなんだろう。とりあえず、\n を \t に変えてつかいコンパイルする。

.ascii “ Data%4x\011\000 ” どうやら \n が \012 で、\t では \011 っぽい。
しかし、なんのことかわからない。

では、PC ではどうなるんだろう。

.string “ Hello,World! ” おや？

movl \$.LC0, %edi

call puts

そんな最適化いらないぞ！ということ…

\t にしてみる。

.string “ Hello,world!\t ”

\ とでた。 \012\000 にあたりそうな ascii コードも \n になさそうだし、
PC じゃふつうに \t って渡すため謎。

とにかく、これでは 8byte なため、r0 にわたして printf にリンク付き分岐するようだ。

printf(“ Data%4x::::\n ” ,Data);

と言う風に変えてみた。

ところが、

リスト 9.8 ラベル LC3

```

1 .LC3:
2 .ascii  " Data%\4x::::1200
3 .space 3
4 .L71:
5 .word -536346624
6 .word .LC3
7 .word .LC1
8 .word .LC2
9 .word i2cErr
10 .cfi_endproc

```

命令も

リスト 9.9 printf()呼び出し部

```

1 ldr r0, .L71+4
2 mov r1, r4
3 bl printf

```

変わらない。

もしや、.L71+4には.LC3のアドレスがはいってるので？

arm-none-eabi-objdumpする

リスト 9.10 objdumpで見たprintf()呼び出し部

```

1 3c: e59f009c ldr r0, [pc, #156] ; e0 <i2csender+0xe0>
2 40: e1a01004 mov r1, r4
3 44: ebfffffe bl 0 <printf>
4 e0: 0000003c .word 0x0000003c

```

あれ？3cってだれ？0x3cをr0にロードしているのにもかかわらず、0x3cをなにがはいるかは未知であるはず。

この問題はProcedure Call Standard for the ARM Architectureで答え合わせが出来る様だ。

9.2 返り値の研究

この疑問は一度置いておくこととして、こんどは返り値について考えてみよう。

ついでに1つの未知の命令の働きを推測する方法も紹介する。

```

1 int ymzwriteln(int value,int adr){
2     FIO4PIN=0x00;
3     FIO4PIN=adr;
4     FIO4PIN=(0x200+adr);
5     FIO4PIN=(0x100+value);
6     FIO4PIN=(0x300+value);
7     return adr*16+value;
8 }

```

この関数をコンパイルして出たアセンブリファイルを追っていこう。未知のlsl命令での演算を仮に#とおく。

```

1 ymzwriteln:
2 .LFB15:
3     .loc 1 378 0
4     .cfi_startproc
5     push {r4, lr}
6     .LCFI19:
7         .cfi_def_cfa_offset 8

```

```

8   .cfi_offset 4, -8
9   .cfi_offset 14, -4
10  .LVL117:
11    .loc 1 379 0
12    ldr r3, .L171 @左辺を準備
13    mov r4, #0 @まずFI04PINの右辺を準備
14    .loc 1 381 0
15    mov r2, #128 @代入処理を後回しにして次の仕込み
16    @いきなりr2に512を入れないのには、Thumb命令だからという制約があるはず
17    .loc 1 379 0
18    str r4, [r3] @r4は0が入っていた。代入処理
19    .loc 1 381 0
20    lsl r4, r2, #2 @r4には128#2を代入
21    add r2, r1, r4 @r2=128#2+r1(addr)
22    .loc 1 382 0
23    add r4, r0, #1 @r4=1+val
24    .loc 1 380 0
25    str r1, [r3] @Adr代入
26    .loc 1 382 0
27    add r4, r4, #255 @r4=r4+255=val+256
28    .loc 1 381 0
29    str r2, [r3] @r2は128#2+r1(addr)である。3度目の代入である
30    @FI04PIN=(0x200+addr);が行われているはずだ。
31    .loc 1 383 0
32    mov r2, #192 @r2=192
33    .loc 1 382 0
34    str r4, [r3] @val+256をr3に代入
35    .loc 1 383 0
36    lsl r4, r2, #2
37    add r2, r0, r4 @r2=val+r2(192)#2
38    .loc 1 385 0
39    lsl r1, r1, #4 @r1=r1#4
40  .LVL118:
41    add r0, r1, r0 @r0=val+adr#4
42  .LVL119:
43    .loc 1 383 0
44    str r2, [r3] @
45    .loc 1 386 0
46    @ sp needed for prologue
47    pop {r4}
48    pop {r1}
49    bx r1

```

以上から、

$$0x200 + A = 0x80\#2 + A$$

$$0x300 + B = 0xC0\#2 + B$$

が分かる。

どうやら、頭の少なくとも 2bit が対応しているという可能性がある。する

と、#は

- 加算
- シフト
- 乗算
- その他ビット演算

の可能性がある。

桁数から察するに、加算はおかしい。

乗算だった場合、2が即値であつたらありえない。

ここで、

リスト 9.11 戻り値？

```

1    lsl r1, r1, #4 @r1=r1#4
2 .LVL118:
3    add r0, r1, r0 @r0=val+adr#4

```

に注目する。すると、

$adr \times 16 + value;$

が [戻り値?] の様に表現されている可能性があるとわかる。さらに戻り値の可能性をあつっていくと、最初に push した lr を取り出すために、わざと pop {r1} を pop {r4} の後に行っていることに気づく。

これは戻り先番地である。なぜなら bx r1 があり、どうやら呼び出し元に帰っているという予測ができるからだ。

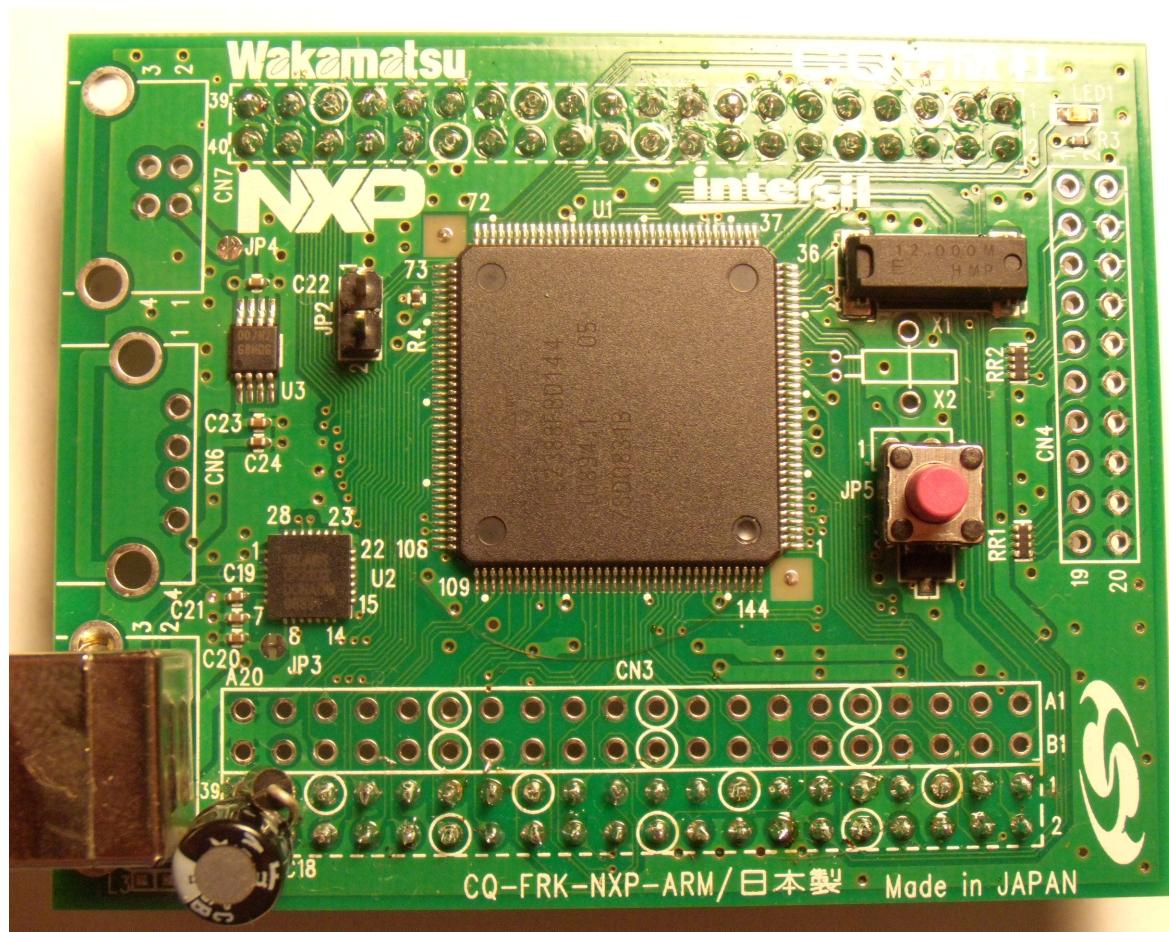
するとこの直前に置かれている r0 がもっとも戻り値として最適であるはずだ。

すると、 $adr \times 16 = adr \# 4$ であることがわかる。

これらの状況は、明らかに # と lsl がシフト演算である可能性が高い。

その場合、戻り値は r0 に記述される。この問題は ARM アーキテクチャマニュアルで答え合わせが出来る。LSL 命令は確かに左シフト演算であった。

図 9.1 ターゲットのマイコンボード



第三部

社会

第 10 章

この論文に使われている技術

10.1 ヴァージョン管理

ヴァージョン管理システムの長所は既にあげたとおりだ。

これは何もソフトウェアにのみに適用されるものではない。以前の状態にもどしたり、もどしたことを取り消したり出来たら便利なのは当然である。2通りに分けて進めたり、後でそれぞれの成果をまとめたりするのも簡単にできる。

ヴァージョン管理システムはさまざまなものがあるが、今回は git を使用した。理由は

- 分散型であること データの損失の可能性が非常に低くなる
- 簡単に扱えること
- 話題性があること

分散型というのは、手元など、一ヶ所のリポジトリに対して変更を書きつづける方法と違い、まず新しいヴァージョンを手元のリポジトリにコミットし、ある程度コミットがたまつたり、1度重要なコミットをしたりなど、好きな時に中央のサーバにプッシュする形式をとるものである。

複数の正式なリポジトリが存在するために、中央や開発者のマシンが壊れても両方同時に壊れることはほとんどないため、データが失われる可能性は極めてひくくなる。この様な形式をとるシステムは他に mercurial などがある。

このシステムの歴史は特異で、現在は日本人の濱野純氏によって開発が続けられているが、もともとは Linux の開発者である Linus Torvalds によって開発された。

これは、Linux Kernel の開発には、Bitkeeper というプロプライエタリなソフトウェア（バイナリのみ無料）を使用していたが、ライセンスや価格改定などの問題があって、乗り換えざるを得なくなったという背景に基づいて創られている。

これは、常にプロプライエタリな製品を使いつづける危険に対して、OSSで立ち向かって解決したという構図にもとることが出来る。

10.2 pLATEX

T_EX というのはドナルドクヌースが開発した、非常にバグが少なく見た目のよい組版ソフトウェアであり、pLATEX はその拡張版である。

また、バグが少ないのは、オープンソースであるとともに、バグが見つかった場合、前のバグ発見者に出した倍の謝礼が支払われるなどの遊びがあるためである。

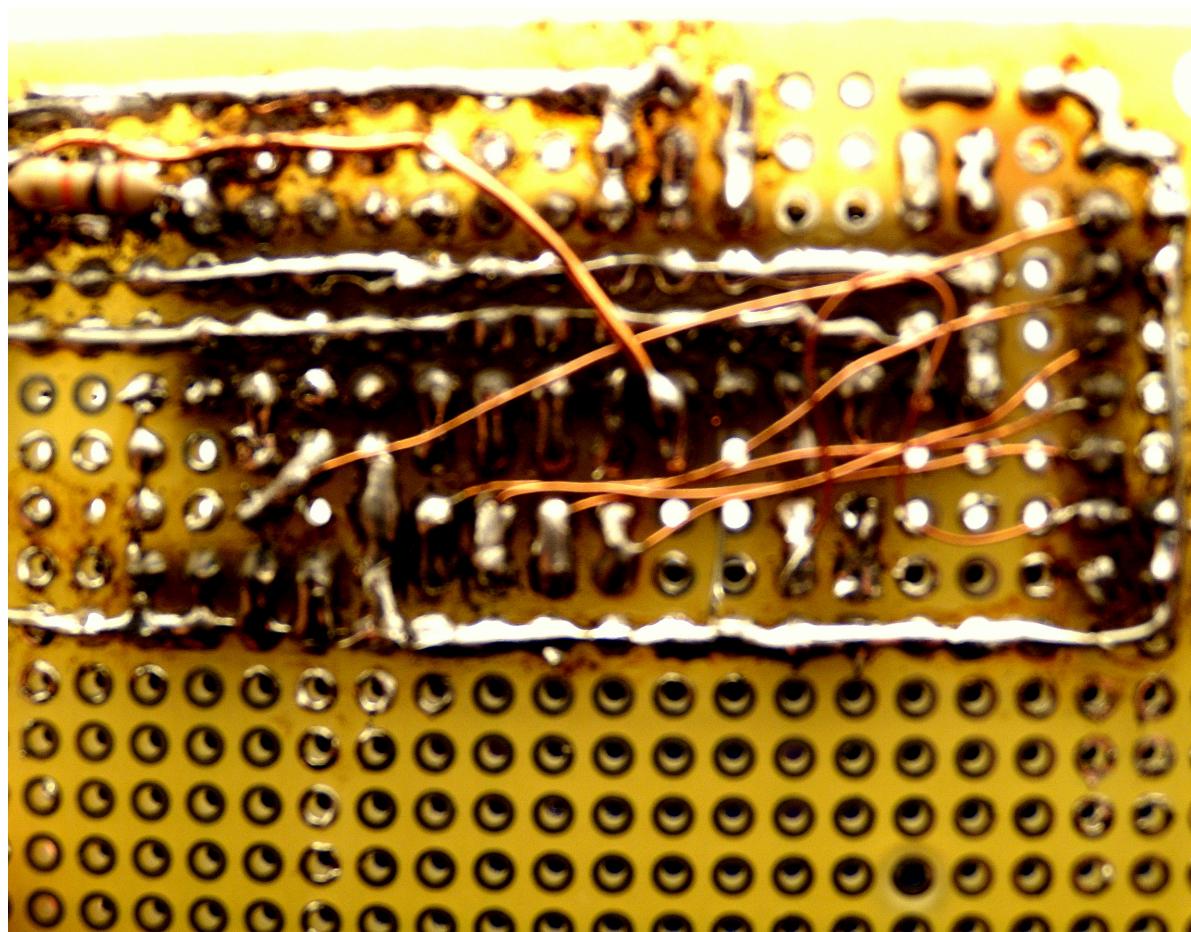
今回、私がこれを採用したのは、この分量を OpenOffice などで書くのは大変であること、編集はテキストエディタで行うため非常に安定していること、見た目が美しいことである。

GUI でかつ WYSWYG な編集ソフトも大変結構だが、あれでは執筆に集中することが出来ないだろう。

だいたい、word 形式など、ソフトウェアどころかヴァージョンや OS で見た目や果てはページ数まで変わって来てしまう。

印刷が自前ならともかく、発注する場合は体裁が自分の手元から離れたときと紙になって帰ってきたときで変わってしまうかもしれないわけで、出版などでは致命的な問題に思える。

図 10.1 このような交差した配線などは高周波を流すのにはまったく向いていない



第 11 章

ノウハウの共有

人間、誰しも最初は初心者である。であるから、初心者を育てることなしに、その分野の発展はむりである。それは、アマチュアの中でもかわらない。

であるから、自分より初心者の人がリファレンスして分かるようなノウハウを残すことは発展上重要である。

11.1 ブログ

ブログでの問題点は、

- 整理された情報が少なく、どうしてもワンポイントノウハウになりがちである
- 整合性を気にして丁寧に書かれた立派なサイトと比べて間違いが非常に多く分布する
- 一度かかれた記事はメンテナンスされにくいという点があげられる。

しかし僕は、ブログの発展はオープンソースの推進に良い影響を与えていくと信じている。それはこの様な状況になったからだ。

- オープンソースを利用する人の裾野がひろがり、数が増えたために、立派な体裁のサイトがすべてに存在するには厳しい状況となった
- マニュアルをよみながら操作を習得する方法は非常に有意であるが、実際はインストール後には殆ど常に読まれることは少ない
- その代わり、問題にぶつかった場合に How-to-solve 式に Google 検索される風潮になった
- 人数が増えたのにもかかわらず、1次ソースのマニュアルのみが重宝されるというのは非効率的であって、利用者がそのまま指導者になる形が好ましい
- 大きなマニュアルと違い、ブログは、その日つまずいたことと直し方なり、その日の作業の要約なりで有用なノウハウが貯蓄されるにもか

かわらず、目的が不在で、書きつづけ整備することにほとんど心の準備が必要ない

利用者が裾野が広がる形で増えたために、そこまで手間な方法でなくともノウハウの共有が必要とされた。ライトユーザをオープンソースが”ただの”利用者から”利用メインだが貢献もする”利用者に変えるという重要なオープンソースの発展の鍵にブログは最適にマッチした。これで、記事の間違いや死角は、人数によってカバーされるようになった。

11.2 SNS

近年の SocialNetworkService の発達は目覚ましい。
このようなメディアを利用すると、どのようなことが技術者になるために役に立つのであろうか。

11.2.1 背中を見る視点

もっとも素晴らしいことは非常にすぐれた技術者との出会いがあるということだ。もし自分のまわりの知り合いだけが、知っている技術者であれば、関心のあるトピックが偏り、また、同じようなレベルの人とのつながりでしかない。このつながりは普段技術を身につける上ではもっと大切で、得意な分野が違うために教え合うということが一番のモチベーションにもなり、またスキルアップにもつながる。

しかし、それだけでは上への広がりが見えにくい。いま自分の持っている、または習得しようとしている技術がどのような展開をしていくか、ということを、その場でまさに使ったり、開発したりしている人がいるというのは意欲をかきたてられるし、背伸びして技術を学ぶようになる。

初心者に限らず、技術者に限らず、すべてのスキルアップは背伸びなしにはありえない。

11.2.2 人に教えるということ

逆に、自分がある程度技術を持つようになると、今度は困った人を見る機会が多くなる。そのような場合に、いろいろなアドバイスをすることを楽しむというのは、非常に大きな自分の訓練になる。

教わる、高度な Guru 的技術者と話す、技術者同士がやりとりをしている現場を見る、そして教えるということ。これらのコミュニケーションは、プロプライエタリな、企業などの環境の外において、技術者として育つために必須の環境であることは間違いない。これが最もリアルタイムにできる場所という意味で、SNS は非常に有意である。

SNS にはさまざまな種類のサービスがある。たとえば、github にも SNS の機能がある。これは注目しているソフトウェアをウォッチするということで実現する。

ウォッチしているソフトウェアにあたらしいリリースがなされれば、それが自分のダッシュボードでわかるというものだ。

またプログラミングコンテストの SNS もある。Codeforces などでは頻繁にやりとりを行う人々もいるし、TopCoder などでもどうやらそのようだ。

また、専門性のない一般的な SNS、たとえば Twitter などでも、そのようなメリットは享受できる。ハッカー（この場合は What's a Hack?での意味をとる）の掛け合い、文句、興味の最先端が垣間見える。

いままで、技術者は大学や会社などでしか育成出来なかった。それは人間が直接かかわらないと人間は育たないからだ。しかし、いま、誰でも好奇心さえあれば教えてもらえる立場に立つことが出来る。

11.3 フォーラム

もっと専門性の高い疑問や問題にぶつかって、サラッと答えられないような問題では、SNS では解決がなされない。また、SNS では過去のやりとりが流れてしまって、ほとんど参照できない。少なくとも参照性がわるい。また、どうしても日本の中の、SNS でつながっている人間だけだと教えられる人が限られてくる。

そこで海外のフォーラムを活用すべきだ。

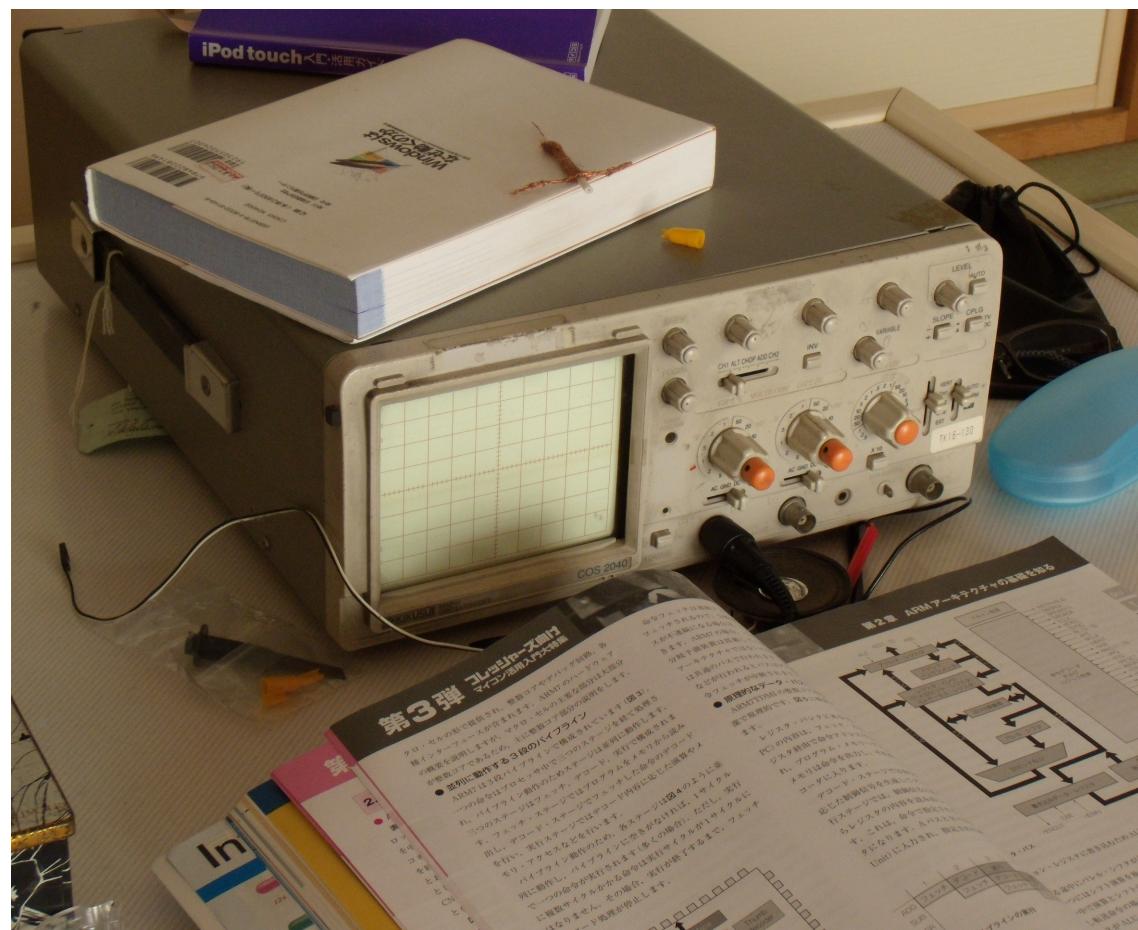
たとえば、AlteraForum は頻繁に利用した。ここでは、教えたことの量でランクづけがなされる。これはもしかしたら教える側にモチベーションの一つとなっているのだろうが、やはり一番はその質問者の”Thx!”だろう。

11.4 man

どのように xxx はうごいているのか、とか、xxx とは一体何か、といった漠然とした疑問が初心者から出された場合、よくあるのは、「ここを参考にしろ」といってソースコードとオンラインマニュアルが示されることだ。この様な風潮だけでは初心者を遠ざけてしまう。

しかし、man というコマンドが UNIX 系の PC 用 OS には必ずといつていほどあり、これが何十年もの間非常に重宝され、また丁寧にメンテナンスされてきたというのは事実であり、素晴らしい伝統である。

図 11.1 オシロスコープ



第 12 章

社会に於けるオープンソースの立場

12.1 ビジネスマodel

オープンソース製品を企業が開発することは、決して儲からないというわけではない。

むしろ、そのような企業は年々増えている。これは、サポートの販売がメインとなっている。特に Linux や仮想化などの、エンタープライズの分野ではよく耳にする。

たとえばオープンソース ソリューションの検索結果と windows サーバ ソリューションの検索結果ではほとんど数が変わらない。

また、2011 年はモバイルデバイスの販売数がパソコンを上回るとの予測 japan.internet という予測速報が先日発表されたが、このようなモバイル端末の OS は Android にかぎらず、Linux や TRON の様なオープンソース OS が搭載されていることが多い。

さらに、OS や仮想化よりも上のレイヤにおける、HTTP サーバやメールサーバ、FTP サーバなど主要なサーバサービスや javascript などもそれぞれ Apache、postfix、ncftpd、jQuery など、OSS が 1 位のシェアを誇っている。

また、ブラウザも、firefox や chrome など、著名なブラウザがオープンソースである。

12.2 ホスティングサービス

Google や github.com、sourceforge などが、OSS ホスティングを無料で行うサービスを近年発表したために、OSS をつくったが寄付も広告収入もなく、サーバ運営の実費は嵩んでいく…というようなリスクが回避され、巨大な OSS プロジェクトが成熟してきた今日において、今度は小さなプロジェクトがたくさんできるだろう。

それらのうちの一部が、GIMP や Inkscape のように有名になるのだろう。それによる派生利益（ダウンロード時に人々が見る広告の収入やサポート費）は、既に集約されていたサーバシステムの追加管理費より当然大きく

なる。

結果として、そうはならなかったプロジェクトも恩恵を受けることが出来る。

12.3 Arduino

また、2008年ごろから注目が集まっている有名プロジェクトとしてArduinoがある。これは、手軽に初心者でも電子工作を樂のしめるように設計されたオープンハードウェア端末である。

それまでにオープンソースソフトウェアが発達してきたのに対し、オープンソースハードウェアというものはあまり存在し無かったのだが、これをきっかけに Armadillo や Gainer など様々な電子工作を容易化するハードウェアが公開された。

この他に、オープンソースハードウェアなモバイルPCやCPUなども数多く存在する。

12.4 幅広い層の参画

また、オープンソースはすでに技術的なプロダクトにのみ適用されるものではない。

Richard Stallman、P2Pを語るで述べられているとおり、ソフトウェアだからといって product(製品) というだけの見方は正しくない。企業が営利のためだけに開発したものと違って、そこには哲学も流れているし、ソースコードが知らない人に読まれるということはソースコードやデータベースをいかに美しく築くかという努力も行われる。

それはすでに Artwork の領域だ。

また、逆に、本当の芸術活動としてのアートワークもオープンソース化されてきている。

wikipediaの記事や、この論文にも適応されているようなクリエイティブコモンズも既に有名であるし、GPLやMITライセンスが適応された芸術作品は、PCのデスクトップピクチャやアイコン、アピアランスがまだおおいものの、それ以外の芸術作品も徐々に増えている。

駆け出しやいくら頑張ってもダメなどの様に売れないよりは、多くの人に楽しんでもらおう、という人もいれば、高クオリティ版や便利に視聴するためには料金が発生するが低クオリティ版や不便な形のものは再配布可能、という人もいる。

もっとも好ましいのはオープンソースの発展に夢を感じている人が、それらのソフトウェアで利用出来るようにするためなどの目的で公開していることだろうが。

たとえば以下のサイトがそうである。

- gnome-art
- deviantART
- KDE-Look.org
- SOUNDCLICK

第 13 章

90 億総ソースコード・リテラシイ の時代へ

13.1 PC 価格の低下

ここ数十年のコンピュータの価格は下落を続けている。それは性能あたりもそうだが、一般的な家庭用 PC の中央値も下がり続けている。

特に去年流行したネットブックなど、回線契約費を含めて 1 円だったこともある。どうせん回線維持費はかかる訳だが、その端末でなければ接続できないわけでもない。

これにともなって、子供のような、プログラミングを習得しやすい世代のうちに、コンピュータリテラシイを身につけるようになってきている。

小さいころからプログラミングをした方が、よりすぐれたコンピュータ技術者になりやすいことは自明である。ネイティブスピーカーの方が、第 2 言語として学んだ人より堪能であるのは一般に知られた事実だ。

事実、技術系の部活に所属していると、新しい世代の方が、よりすごい人が入ってくることが多いという実感がある。

よりすぐれたコンピュータ技術者がより多い世代が活躍するようになれば、それだけ技術者の密度が上がるため交流が盛んにもなるはずだ。これはコンピュータ科学やオープンソーステクノロジに革新をもたらす可能性を秘めている。

特に、発展途上国でも個人が PC を持つようになってきている。プログラミングは、ほとんどインターネットさえあれば出来る、お金のかからない趣味となりうる。

13.2 ツールの成熟、コミュニティの発達

また、これまで見てきた通り、コミュニティも開発環境も、十分とは言えないまでも、急速に成熟してきていて、今後、よりよい環境が整ってい

くことが期待される。

よりよい開発環境や言語の登場はソフトウェア開発を加速させるであろう。

もし、これから、学校の授業でコンピュータ科学などをおしえるまでになって、ほとんどの一般人がプログラミングできるようになり、読書やスポーツのように、ソースコードを読んだり書いたりする人がでてくれればオープンソースコミュニティに本格的に参画する人は否応なしに増えるであろう。

この究極的な形は、世界中の人口の大半がオープンソースソフトウェアを利用してオープンソースコミュニティの一員となることである。2050年には世界人口が90億に達するらしいので、おかげさにこの値を設定した。しかし、21世紀内に達成されないと強く主張できる要素はどこにもない。

そして、これこそが眞の技術的特異点かもしれないのだ。

図13.1 飲み干されたRedBullはRedNullである

