ブートローダのつくりかた

今井邦彦

2007 年 3 月 3 日版

概 要

組み込み Linux 環境でのブートローダのつくりかたを解説したかったんだけど，チェインローダの解説になってしまいました．ターゲットは OpenBlockS266.

目 次

法的宣言等

責任の放棄

- 本文書は無保証です.
- 本文書の内容の正当性について筆者は一切保証しません.
- 本文書の運用によって生じたいかなる損害についても筆者は補償しません.

著作権

再配布

- 不特定多数に対する再配布は禁じます.
- 私的再配布は制限しません.
- 本文書の配布元 URL の伝達（リンク）に関しては制限しません.

第1章　はじめに

ブートローダを作る過程を書いていきたいと思います.
読者として想定しているのは

C 言語 C で一通りのプログラムを書くことができ，ポインタを使いこなすことができれば ok.

アセンブリ言語 何らかの CPU でアセンブリ言語でプログラムを書いたことがあれば ok．x86 がわからなくても大丈夫（実際，筆者もよくわかりません）．Z80 もくしは 6809 程度の経験があれば問題ありません.

論理回路 できればこのスキルは欲しいところですが，論理回路の知識が無くても理解できるよう，記述には努力します．が，現場のハードウェア担当者と意志の疎通が取れる程度のレベルなので，本格的に勉強する場合は別の参考書を当たってください.

というところです.

第2章　ターゲットボード

本文書では **OpenBlockS266** をターゲットにブートローダの作り方を書いていこうと思います．

OpenBlockS266 はぷらっとホーム[1]から発売されている小型のサーバで，PowerPC コアの SoC を採用しています．現在は 128MB 版のみの販売となっているようですが筆者が所有しているモデルは 64MB 版なので，これの上で開発を行っていきます．これが問題となることはおそらく無いでしょう．

実は，このターゲットの選定には大きな問題かあります．OpenBlockS266 では IBM の **PPC405GPr** という SoC を使用しています．が，この SoC の所有権は IBM から AMCC[2]に売却されており，IBM の web サイトからユーザーズマニュアルが入手ができなくなっています．

AMCC 版のマニュアルは，AMCC の web ページでユーザ登録を行えば入手は可能です．しかし，このマニュアルは「機密情報」であり，評価目的以外では使用してはいけないようです．というわけで，本文書では IBM 版のマニュアルを元に議論します．

表 2.1 に OpenBlockS266 のハードウェア仕様を示します．

図 2.1: OpenBlockS266

[1] http://www.plathome.co.jp/
[2] http://www.amcc.com/

表 2.1: OpenBlockS266 ハードウェア仕様（抜粋）

CPU	IBM PowerPC 405GPr 266MHz
Memory	64MB (PC133,SDRAM)
Flash ROM	8MB
LAN (RJ45)	・10/100Base-TX 2 ポート （LAN 側：CPU 内蔵 / WAN 側：DEC CHIP 互換）
シリアルポート	・RS232C (RJ45 , PPP/コンソール) ・コンソール（別途オプションの変換アダプタが必要）
その他 I/F	・JTAG
内蔵ストレージ	・コンパクトフラッシュ（PIO4 まで対応) ※ 1 ・HDD(2.5”)（IDE（内蔵） UDMA100) ※ 1
拡張スロット/バス	・IDE（内蔵）※ 1 ・PCI（外付け、独自仕様）
電源	DC5V/3.0A （専用 AC アダプタ付属）
スイッチ	INIT
表示・警告	・ステータス LED x3 ・LAN アクセス LED ・CF/IDE アクセス LED
オプション	・PC Card(PCMCIA) スロット拡張 BOX ・専用簡易 UPS ユニット※ 2 ・コンソールシリアル変換アダプタ (AUX ポート専用)

※ 1　CF と HDD とは排他仕様

「ぷらっとホーム – OpenBlockS シリーズ – OpenBlockS266：仕様[3]」web ページより，メモリ容量のみ 64MB に変更

第3章　資料を集めよう

ブートローダのような低レイヤのプログラムのでは，どんな名人級のプログラマでもターゲットボードの知識無しでプログラムを組むのは不可能です．名人ならば資料無しでもリバースエンジニアリングによって仕様を調べることも不可能ではないのかもしれませんが,「仕事」でプログラムを組むのならまずは正道を進むべきでしょう.

3.1　ユーザーズマニュアル

ユーザーズマニュアル (user's manual) とは,「〜のデバイスを使うにはどのポートをどう操作する」という説明のある資料のことです.

似たような資料に**データシート** (data sheets) というものもあります．こちらはパッケージのピン配置や電気的特性が記されたものです．と書くとブートローダの作成には関係なさそうですが,共用ピンの設定パラメータがなぜかデータシート側に記述されている SoC に筆者は遭遇したことがあります．これも入手して一応目を通しておきましょう.

あと，忘れてはならないのが**エラッタ** (errata) です．マニュアルの誤記[1]や SoC のバグが記されています．また，エラッタはその性格上，随時更新されるものです．更新が無いかこまめにチェックしましょう.

3.2　CPU の資料

ブートローダを記述するにはアセンブリ言語を使用する必要があります．ということは，CPU についてもある程度の知識が必要になることになります.

SoC では，ARM Ltd. や MIPS Technologies Inc. のような別の CPU コア IP ベンダから CPU コアの設計を買ってきて CPU コアを実装することが多くなっています．このため，ユーザーズマニュアルにおいても「CPU コアについては別資料を参照すること」で済ませてあるものが多くなっています.

というわけで，CPU コア IP ベンダ（と言うのかな）の URL を挙げておきます．MIPS のほうはユーザ登録画面がありますが，必要事項を入力すれば CPU 資料を入手することができます.

- ARM Ltd.[2]
- MIPS Technologies Inc.[3]

これらの資料は英語で書かれています.「いきなり英語はしんどい」という場合は，以下の資料で予習してからチャレンジすると楽かもしれません．そのものズバリの解説ではありませんが，CPU の理解には役立つはずです.

[1] こっちが本来の意味でのエラッタ
[2] http://www.arm.com/
[3] http://www.mips.com/

- 「改訂 ARM プロセッサ–32 ビット RISC のシステム・アーキテクチャ」(CQ 出版社)[4]
- 「mips RISC アーキテクチャ R2000 R3000」(共立出版)[5]

3.3　ターゲットボードの仕様書

SoC 単体でターゲットボードが構成されているわけではありません．最低でも DRAM・フラッシュROM・LED などのデバイスが接続されてボードを構成しています．ブートローダのプログラミングにはこの部分の仕様が必要です．

具体的には

- フラッシュROM の型番と SoC への接続
- DRAM の型番と SoC への接続
- LED の接続
- PLD がある場合はその仕様
- その他デバイスの SoC への接続

というあたりです．

あと，できればターゲットボードの回路図（と部品表）も入手したいところです．逆にこちらのほうが仕様書よりも重要かもしれません．というのは，そもそもブートローダを書くのは，システム開発の初期に行われる作業です．この段階で漏れの無く誤りの無い仕様書があるかというと…難しいでしょう（本来は荘あるべきなのでしょうが）．一方，回路図のほうは CAD を使用しているので，ターゲットボードとの対応に誤りは無いことが期待できます．

え，回路図の見かたがわからない？…精進してください．

3.4　バスやプロトコルなどの資料

例えば UART インターフェースの説明では，スタートビット・ストップビット・ブレーク信号などの UART の信号については詳細には説明されていません．これらのことは既知であることが読者に求められているわけです．

必要な知識が無い場合，資料が求めている知識を仕入れることが必要です．web で検索するなり参考となる書籍を購入するなりしましょう．書籍の場合は CQ 出版社の雑誌 Interface[6] やその別冊が参考になるでしょう．

3.5　実際に集めてみよう

IBM の web サイト

PPC405GPr Embedded Processor User's Manual(入手不可)

AMCC 版のユーザーズマニュアルについては 2 章を参照のこと．

[4] http://www.amazon.co.jp/gp/product/4789833577
[5] http://www.amazon.co.jp/gp/product/4320025989
[6] http://www.cqpub.co.jp/interface/default.asp

OpenBlockS266 付属 CDROM ぷらっとホームの web に最新版あり.

- PCC2000 ハードウェア機能仕様書
- OpenBlockS-266 ユーザーズガイド
- OpenBlockS-266 シリアルセットアップツールガイド
- OpenBlockS-266 WEB セットアップガイド

Linux カーネル ドキュメントに記述されていない事項を知るには，実際に動作しているプログラムのソースコードを読むのもひとつの手です．ぷらっとホームの ftp サーバよりダウンロード.

第4章　ライセンス

4.1　NDA

NDA(Non Disclosure Agreement; 非開示契約) というのは，例えば SoC ベンダとの間で「ユーザーズマニュアルを提供するが，他者には開示しないこと」という条件のもとで契約を結ぶことです．NDA の下に開示されたマニュアルを元にプログラムを組んだ場合，通常はそのソースコードも NDA の下にと置かれます．つまり，他者に対しソースコードを開示してはなりません．

ただし，最近は「文書を元に作成したプログラムのソースコードは他者に提供してもよい」という条件の付いた NDA も結ばれるようになってきています．

4.2　GPL

プログラムの場合，**GPL**(GNU General Public Licence) というものがあります．

- GPL で公開されたプログラムを他者に提供する場合，その他者にソースコードを提供しなければならない（と考えておいたほうが良い）
- GPL のプログラムを引用もしくはリンクしたプログラムも GPL としなければならない

という特徴があります．要するに,「GPL のプログラムを利用したプログラムも GPL としなければならない」ということです．

ただし，GPL の場合はあくまでも著作権に基づいた，プログラムに対するライセンスです．従って，GPL のプログラムから設定情報や操作方法を読み取りそれを元に新たにプログラムを書く場合は，GPL は伝播しません．

4.3　GPL と NDA の板ばさみ

仮に，NDA の下で開示された情報を元にプログラムを書き，それには GPL が適用されるコードが含まれる場合を考えてみましょう．このプログラムは，NDA と GPL の両方のライセンスに従わなければなりません．

で，このプログラムの実行形式を他者に提供したとします．GPL のプログラムを受け取った人は，提供者にそのソースコードを請求することができます．しかし，このソースコードは NDA に抵触する情報が含まれているため，GPL によるソースコード開示には従うことができません．**GPL と NDA の板ばさみ**になってしまうわけです．

最近は，GPL に対する理解もある程度広がっているため，先に述べたように「NDA 情報を利用したソースコードは開示可能」という付帯事項のある契約を結べるベンダも出てきています．が，まだまだ無頓着なベンダも多いので，この板ばさみには気を付ける必要があります．

表 4.1: ライセンスとプログラムの取扱い

プログラム＼文書 or プログラム	NDA	NDA(ソース公開可)	公開	GPL
バイナリ供給のみ	○	○	○	×
ソース公開	×	○	○	○

ところで，この板ばさみが発生しても問題とならない状態があります．開発したシステムを自社内のみで使用する場合です．プログラムを他者に提供していないので，他者による GPL 下でのソースコードの請求は発生しないことになります．

と，システムの利用形態がプログラム開発に影響するようになっています．「これは SE の仕事だからプログラマには関係ない」というわけにはいかなくなっています．

第5章　道具を揃えよう

機材の値段については記憶に頼って書いているので，現在は状況は違うかもしれません．が，これより安くはなってはいるでしょう．

5.1　評価ボード

大抵の SoC の場合，SoC ベンダ（やサードパーティ）が**評価ボード** (evaluation board) というものを用意しています．

5.1.1　特徴

評価ボードの特徴は以下の通りです．

SoC のほとんどの機能を利用可能　実用のボードの場合，SoC 内で利用しないデバイスの信号は引き出してありません．が，評価ボードの場合は文字通り「評価」が目的ですから，一通りのデバイスが利用できるようにしてあります．また，バスの信号をモニタできるように出力インターフェースとコネクタを装備しているボードもあります．最近は組み込みデバイスと言えどもバスは高速なので，プローブを直接当てて信号をモニタするのは困難なためでしょう．

基板サイズ　一般に評価ボードは基板サイズが巨大です．PC のマザーボードぐらいのサイズがあります．SoC の機能をフルに評価できるようにしてあることもありますが，作業性を良くするために部品実装をわざとスカスカにしてあるものもあります．

価格　ターゲットボードのでは単価が数千円オーダであっても，その SoC の評価ボードは数十万円のオーダです．仕事で開発をする場合は評価ボードを貸し出してくれる場合もあります．

ソフトウェア　パフォーマンスの評価も行えるよう，ブートローダ・OS などの一通りのソフトウェアが用意されています．

5.1.2　ターゲットボードとの関係

ターゲットボードを設計する場合，評価ボードを元に不要な機能を切り落として不足するデバイスを付け足す場合がほとんどです．ということは，ターゲットボードでのプログラムをある程度評価ボード上で素組みし，その後ターゲットボードに移植する，という開発も可能でしょう．

5.2　母艦 PC

入手性とコストパフォーマンスを考えると，x86 の PC を使うというのが普通の考えです．が，ターゲットが PowerPC の場合は PowerPC Mac を使うというのも悪くない考えです．

OS は Linux などの PC UNIX をインストールしておきましょう．Windows 上で cygwin をインストールすれば開発は不可能ではないですが，使い勝手は PC UNIX より数段劣ります（とわたしは感じました）．VMware 等の PC エミュレータをインストールしこの上で PC UNIX を動作させるという手もあるでしょう．

で，この母艦 PC の役割ですが

- シリアルコンソール
- クロスコンパイル環境
- bootp サーバ・tftp サーバ

というところです．

5.2.1　シリアルコンソール

組み込みターゲットボードは小型化されているため，ビデオコンソールデバイスが無い場合が多々あります．また，ビデオコンソールがある場合でも画面に文字を出すだけでもかなりの量のプログラムを書く必要があるでしょう．

一方，**シリアルコンソール**の場合，UART デバイスを叩くだけで文字を出力することができます．また，大抵の SoC には UART デバイスが内蔵されており，「シリアル」デバイスであるためピン数も少なく，デバッグ用に引き出せる場合が多いでしょう．

ただし，シリアルインターフェース (EIA-232C/RS-232C) では信号の電圧レベルが特殊なため，SoC の外にレベルコンバータがが必要なものがほとんどです．運用時にシリアルを使用しない場合，シリアルのピンが引き出されていてもレベルコンバータを省略してある場合が多いでしょう．このような場合，外付けでレベルコンバータが必要になります．

5.2.2　クロスコンパイル環境

クロスコンパイラとは，ホスト環境とは別のアーキテクチャのバイナリを生成するコンパイラのことです．例えば，cygwin 上で PowerPC Linux 用のバイナリを生成するコンパイラがこれに当たります．

ブートローダを作成する場合は，ヘッダ情報は全て削除し裸のバイナリとしてターゲットボードに書き込むことになります．このため，クロスコンパイラはターゲットボードの CPU アーキテクチャと一致していれば，クロスコンパイラのターゲット OS は不問です．

5.2.3　BOOTP サーバ・TFTP サーバ

既存のブートローダでは，「BOOTP で IP アドレス等を取得し，TFTP サーバから OS カーネルをロードし，OS を起動する」という機能を搭載しているものがあります．既存のブートローダが

既に存在している場合，BOOTP サーバと TFTP サーバを立ち上げておけば OS の代わりに開発中のブートローダを転送することもできます．

BOOTP とは，IP アドレスなどをクライアントに割り当てるプロトコルです．似たようなプロトコルに DHCP がありますが，実は DHCP は BOOTP プロトコルを拡張したものです．BOOTP と DHCP の最大の違いは,「BOOTP でのアドレス割り当ては無期限である」という点です．

TFTP(Trivial File Transfre Protocol) とは，ファイル転送のためのプロトコルです．ファイル転送というと FTP というプロトコルがありますが,「ファイル転送」という目的以外はあまり類似点は無いようです．TFTP を FTP と比べると

- FTP は TCP 接続だが TFTP は UDP 接続
- ユーザ-パスワード認証は無い
- FTP よりも実装は容易

という特徴があります．

5.3 テスタ

テスタとは，電圧計・電流計・抵抗計をコンパクトな筐体にまとめたものです．

テスタには**アナログテスタ**と**ディジタルテスタ**の 2 種類があります．ボード開発にはディジタルテスタのほうがオススメです．というのも，ディジタルテスタはアナログテスタに比べて入力インピーダンスが高く，被測定回路にあまり影響を与えずに測定を行うことができます．電圧測定モードの場合，アナログテスタの入力インピーダンスは数 10 kΩ ですが，ディジタルテスタでは 10 MΩ もあります．

テスタの場合，反応速度とその表示形態のため，観測できる信号の変化は数秒周期ぐらいが限界です．が,「ピンの出力が H か L か」というような確認には十分利用することかできます．また,「電源は来ているか」「断線はしていないか」というチェックもできますね．

価格ですが，ディジタルテスタは，安いものでは 1000 円ぐらいからあります．機能としては 1000 円テスタでも十分で，問題はありません．しかし，さすがに作りは安っぽいので，選ぶポイントはそのあたりになるでしょう．

5.4 オシロスコープ

テスタでは観測できないような変化周期のある信号を観測する場合は，**オシロスコープ** (oscilloscope) を使用します．オシロスコープは略して**オシロ**と呼ぶことが多いです．年配の方は**シンクロスコープ**もしくは略して**シンクロ**と呼ぶこともあります．が，これは岩崎通信機[1]の登録商標なので一般名称としては「オシロスコープ」が正解です．

オシロスコープにも**アナログオシロスコープ**と**ディジタルオシロスコープ**の 2 種類があります．またアナログとディジタルの二者択一が出てきましたが，これもディジタルオシロのほうがオススメです．

というのも，ボード開発では繰り返し波形ではなく単発波形を観測する場合が多いのですが，アナログオシロで単発波形を観測するのは困難なためです．アナログオシロで単発波形を見ようとし

[1] http://www.iwatsu.co.jp/

たら消え行く残像を必死に追いかけることになります．一方，ディジタルオシロの場合は入力された波形をいったんメモリに蓄えてから表示するため，単発波形でも表示が薄くなって消えていく，ということはありません．このため，ディジタルオシロスコープは**ストレージオシロスコープ**と呼ばれることもあります．

なお，ディジタルオシロもアナログオシロも，被測定回路に接続するプローブの設計は同一です．このため，被測定回路に与える影響はアナログオシロでもディジタルオシロでも違いはありません．この点はテスタとは違うところですね．

ディジタルオシロの値段ですが，安いものでは新品のブランド物 (ソニーテクトロニクス) でも 10 万円台であります．選ぶ際の重要なポイントとしては

チャネル数 入力チャネルが多いほうが複数の波形を同時に測定できるので便利でしょう．

周波数帯域 100 MHz あれば十分です．これ以上の周波数の波形をまともに観測しようとしても普通のパッシブプローブでは無理です．基板に出力用のインターフェースとコネクタを付けて同軸ケーブルで接続するか，アクティブプローブを使用する必要があります．

といったところでしょう．

5.5　ロジックアナライザ

ロジックアナライザ (logic analyzer) もオシロスコープ同様に比較的変化の速い信号を観測する道具です．これもしばしば**ロジアナ**と略されることがあります．

ロジックアナライザはオシロスコープに比べて次のような違いがあります．

ディジタル回路に特化 オシロスコープはアナログ信号を観測できるように作られていますが，ロジックアナライザはディジタル信号の観測に特化されています．

信号レベル オシロスコープでは連続した電圧値を観測・記録できますが，ロジックアナライザではハイ・ローに量子化されたレベルでしか電圧を観測できません．

入力チャネル数 バスの信号を観測できるよう，少なくとも 10 数本の入力チャネルがあります．

表示 複数チャネルの信号値から 16 進数で表示する，などの機能があります．

トリガ条件 「複数のチャネルがある値になったら」というような，複雑なトリガ条件を設定することができます．

しかし，基板の高密度化やバスの高速化により，基板に端子を立てて信号を観測するのは難しくなっています．このため，次に述べる JTAG デバッガにその役目を譲りつつあるのが現状かもしれません．

値段ですが，プローブを PC に接続し，Windows 上で操作するタイプのものが 20 万円越え程度であったような記憶があります．

5.6　JTAG デバッガ

JTAG というのは，もともとはバウンダリスキャンのためのインターフェース標準のことを指し，IEEE で定義されています．これにより，LSI の各ピンの状態 (H もしくは L レベル) を得ることができます．つまり，ロジックアナサイザのような機能がまずあります．

また，LSI の任意の出力ピンに任意の値を出力するように設定することができます．JTAG の機能は CPU や FPGA などの機能と独立しているので，CPU が動作していない状態でも使用することができます．つまり,「ピンの値を読みだす」「ピンに値を出力させる」という機能を組み合わせると，SoC に直結したフラッシュROM をボードに乗ったまま書き換えることができます．

あと，JTAG を拡張して，CPU レジスタなどを読み取り・書きだしできるような機能のある SoC も多いようです．このような SoC では，JTAG インターフェースから SoC 上で動くプログラムのデバッグを行うことができます．

で JTAG デバッガのお値段ですが…数 10 万円〜100 万円といったところでしょうか．個人的には MITOUJTAG[2]あたりに興味があります．デバッガとは方向性が違うようですが値段も手頃 (10 万程度) のようです．

5.7 ROM ライタ

ROM ライタ (ROM writer) は，文字通り ROM にバイナリ列を書き込む道具です．ブートデバイスが EPROM の場合，ROM ライタでブートローダを EPROM に書き込み，ターゲットボードのソケットに装着して動作を確認する，という開発形態になると思います．また，EPROM の場合，その消去には **EPROM イレーサ** (EPROM eraser) が必要になります．これは紫外線灯を内蔵した箱で，EPROM の窓からチップに紫外線を当てることで書かれている内容を消去する道具です．

ブートデバイスがフラッシュROM の場合でも，ROM ライタによるブートローダの開発は可能です．フラッシュROM が搭載される位置にソケットをはんだ付けし，ROM ライタでブートローダを書き込んだフラッシュROM を装填します．フラッシュROM の場合は，ROM ライタで端子に消去シーケンスを送れば内容を消去できるので，EPROM イレーサは不要です．というか，フラッシュROM には窓が無いので，EPROM のように紫外線の照射で内容は消去できません．

なお，ROM ライタでプログラミングする場合，書き込み対象となる ROM がその ROM ライタでサポートされているか，事前に確かめておきましょう．また，同じ ROM でもパッケージ (DIP, PLCC, SOP など) が複数用意されいる場合があります．パッケージにあったソケットが ROM ライタに付属しているかも確認してください（私はこれでエライ目に逢いかけたことがあるので…）．

また，EPROM には**ワンタイム PROM**(One Time PROM) タイプのものもあります．中に入っているシリコンチップや足の並びや窓有りの EPROM と一緒ですが，窓の無いパッケージにして部品単価を抑えたものです．窓が無いので，一度書き込んだデータは紫外線を当てても消去することはできません．その名のごとく書き込みは１回ポッキリです．こんなものを開発に使ってたら使い捨てになってしまうので，精神衛生上よくありません．この場合は量産用の部品とは別に，開発用に別途窓有りの EPROM を手配しておきましょう．

5.8 レンタル

これらの開発機材は購入しても良いのですが，使用頻度によってはレンタルで済ませてもよいでしょう．レンタルの場合は

- オリックスレンテック[3]

[2] http://www.nahitech.com/jtag/
[3] http://www.orixrentec.jp/

- 横河レンタ・リース[4]

がよく使用されるようです.

5.9　実際に揃えてみると

母艦 PC とテスタしか揃いませんでした．この文書は個人的に書いているものなので，まぁこんなもんです．なお，この PC には debian[5]をインストールしてあるのでこれをそのまま使います.

[4] http://www.yrl.com/
[5] http://www.debian.org/

第6章　開発手順を考えよう

6.1　ブートローダの書き込み方法

まずは，ブートローダの書き込み方法について考えてみます．

ブートローダを書き込むということは，まずは，その手段自身が実行可能であることが必要です．たとえば,「念力でブートローダを書き込む」というのは却下，となるわけです．

次に，起動しないブートローダを書き込んでしまった場合に復旧できるか，というのも重要なポイントです．確かに「ブートローダ自身でブートローダを書き変える」というのも，書き込みという点では可能です．が，書き込んだブートローダが動作しない場合，そこで途方に暮れてしまいます．それこそ「念力でフラッシュROM を書き換えたい」という衝動に駆られることでしょう．

ただし，綱渡りの綱から落ちたときのセーフネットを用意した上で「うまくいくことを前提とした書き換え方法」で開発するのは問題ありません．例えば，普段は Linux 上でフラッシュROM のブートローダ領域を書き換えるけどブートローダが起動しなくなってしまった場合は JTAG 経由でフラッシュROM にブートローダを書き込んでやる，というのも有効な書き込み手順です．

6.1.1　OS 上

例えば Linux では MTD デバイスとして，フラッシュROM を読み書きする機能があります．これを使ってブートローダを書き換えることは可能です．

ただし，起動しないブートローダを書き込んでしまった場合は OS も起動しないので，別途セーフネット的な手段が必要です．

6.1.2　ブートローダ自身

ブートローダ自身でフラッシュROM 書き換えを行う方法です．この場合もうまく動作しないブートローダを書き込んでしまうと何もできなくなってしまうので，別途セーフネット的な手段が必要です．

6.1.3　JTAG デバッガ

5.6 節で述べたように，JTAG デバッガでボード上のフラッシュROM を書き換えることができます．この場合は動作しないブートローダを書き込んでしまってもブートローダの書き換えは可能です．

6.1.4　ROM ソケット

ボード上のフラッシュROM や EPROM のところにソケットを実装し，書き換えのときは ROM ライタを用いる方法です．ブートデバイスが EPROM の場合は，これが唯一の書き換え方法になります．

この場合も動作しないブートローダを書き込んでしまってもブートローダの書き換えは可能です．

6.1.5　ROM 切替え

- ２個以上の ROM 領域を切替える
- 片方の ROM にセーフネット用のブートローダ
- もう片方の ROM は開発版のブートローダを保存
- ブートしない側の ROM も別の領域にマップされている必要がある

例えば L-Card+[1]の場合，ジャンパピンで*CS 信号を切替えることにより外付けの ROM ボードからのブートが可能です．また，評価ボードの中にもブートするフラッシュROM を切替えることができるものもあります．

この場合，セーフネット用のブートローダ自身でフラッシュROM の書き換えを行ってもよいですが，OS を起動してその上でフラッシュROM を書き換えるという方法でも問題ありません．

6.1.6　まとめ

以上の議論をまとめると，表 6.1 のようになります．

表 6.1: ブートローダの書き換え方法

方法	セーフネット
OS 上	×
ブートローダ自身	×
JTAG デバッガ	○
ROM ライタ	○
ROM 切替え	○

6.2　考えてみる

まず，5.9 節で述べたように，機材は PC とテスタしかないので，ブートローダ自身の書き換えは無謀です．というわけで，ブートローダから起動されるチェインローダで妥協することにします．

[1] http://www.laser5.co.jp/product/embe/lc/16m.html

6.3 テスト

実際に転送が行えるかテストしてみます.

まず,「PCC2000 ハードウェア機能仕様書」と「OpenBlockS266 ユーザーズガイド」をを見るとボード上の DIP スイッチの SW3 の設定でブートモードを変更できるようです. BOOTP と TFTP でターゲット OS を転送するということなので，母艦 PC に dhcpd(bootp) と bootp の設定をしてみます.

6.3.1 開発用ネットワーク

会社で開発をする場合，ネットワークには他人の PC も接続されています. このネットワークで自分用の DHCP サーバなんか動かしたら，罵倒の嵐となるのは目に見えています. だからと言って外と完全にネットワークを切り離してしまうと，それはそれで面倒です.「ぐぐる」ことさえままなりません.

というわけで，図 6.1 のように自分用のネットワークを設定することにします.

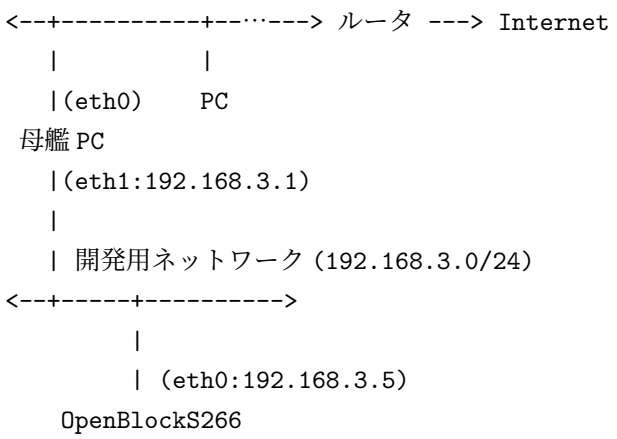

図 6.1: 開発用ネットワーク

6.3.2 tftpd の設定

tftpd のパッケージをインストールし，/etc/inetd.conf に以下の行を追加します.

```
tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /tftpboot
```

編集が終わったら，inetd に設定を読み込ませます.

```
# killall -HUP inetd ↵
```

転送用のディレクトリを作成し，ディレクトリにファイルを置きます.

```
# mkdir /tftpboot ↵
# chmod 777 /tftpboot ↵
# cp somewhere/zImage.treeboot /tftpboot ↵
```

試しに別のホストから tftp でファイルを転送してみます．192.168.1.20 は母艦 PC の eth0 のアドレスです．

```
$ tftp 192.168.1.20 ↵
tftp> get /tftpboot/zImage.treeboot ↵
Received 912772 bytes in 2.0 seconds
tftp> quit ↵
$
```

うまく動作していますね．

6.3.3　dhcpd の設定

/etc/dhcpd.conf を以下のように書き換えます．

```
subnet 192.168.3.0 netmask 255.255.255.0 {
        range 192.168.3.10 192.168.3.200;
        allow bootp;
        option broadcast-address 192.168.3.255;
        option routers 192.168.3.1;
}

host jr0bak-obs {
        hardware ethernet 00:0a:85:01:84:2e;
        allow booting;
        fixed-address 192.168.3.5;
        server-name "192.168.3.1";
        filename "/tftpboot/zImage.treeboot";
}
```

6.3.4　いざ転送

準備の整ったところで OpenBlockS266 の電源を入れてみます．

```
405GPr 1.2 ROM Monitor (5/25/02)
…(略)…
 -- FLASH BOOT Update --
Sending bootp request ...
Loading file "/tftpboot/zImage.treeboot" ...
Sending tftp boot request ...
TFTP boot failed.
Booting from [FLASH] System FLASH ...
```

…失敗してます．が，/tftpboot/zImage.treeboot というファイル名は取得できています．ということは，tftp のほうに問題があることになります．が，6.3.2 節ではちゃんとテストして，他のホストからのファイル取得にも成功しています．

こうなったら tcpdump でパケットキャプチャしてみましょう．

```
# tcpdump -i eth1 ↵
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode li
stening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
21:22:17.505812 IP 0.0.0.0.bootpc > 255.255.255.255.bootps: BOOTP/DHCP, Reque
st from 00:0a:85:01:84:2e (oui Unknown), length 300
21:22:17.507031 IP 169.254.100.163.bootps > 192.168.3.5.bootpc: BOOTP/DHCP, R
eply, length 300
21:22:17.593668 arp who-has 192.168.3.1 tell 0.0.0.0
21:22:17.593820 arp reply 192.168.3.1 is-at 00:20:78:90:32:b7 (oui Unknown)
21:22:17.596639 IP 192.168.3.5.3909 > 192.168.3.1.tftp:  34 RRQ "/tftpboot/zI
mage.treeboot" octet
21:22:18.117818 arp who-has 192.168.3.5 tell 192.168.3.1
21:22:19.117650 arp who-has 192.168.3.5 tell 192.168.3.1
21:22:20.117508 arp who-has 192.168.3.5 tell 192.168.3.1
21:22:23.119063 arp who-has 192.168.3.5 tell 192.168.3.1
21:22:24.118918 arp who-has 192.168.3.5 tell 192.168.3.1
21:22:25.118773 arp who-has 192.168.3.5 tell 192.168.3.1
```

OpenBlockS266 側が arp に応答できていないのが問題のようです．

それならば，

```
# arp -s 192.168.3.5 00:0A:85:01:84:2E ↵
```

と，母艦 PC に手動で arp テーブルを追加してやります．

そして，再度チャレンジ．

```
405GPr 1.2 ROM Monitor (5/25/02)

…（略）…
 -- FLASH BOOT Update --
Sending bootp request ...
Loading file "/tftpboot/zImage.treeboot" ...
Sending tftp boot request ...
Transfer Complete ...
Loaded successfully ...
FLASH UPDATE Success
```

今度はイメージの転送に成功しました．が，問答無用でフラッシュROM の書き込みに行ってしまいました．何度も転送を繰り返すとフラッシュROM が消耗してしまいます．

6.3.5　TEST モード

というわけで,「PCC2000 ハードウェア機能仕様書」にある「TEST モード」を調べてみます．
まず，(#1, #2) = (on, on) で起動してみます．

```
405GPr 1.2 ROM Monitor (5/25/02)

…（略）…
 -- Build in TEST MODE --
Input current time. Format is YY/MM/DD/HH/MM/SS
0/0/0/0/0/ Correct? [y|n]
Checking installed RAM - press any key to exit test
Checking memory address: 00C00000
```

時計の設定をやった後，メモリチェックに入ってしまいました．工場出荷試験のようです．
(#1, #2) = (on, off) で起動してみます．

```
405GPr 1.2 ROM Monitor (5/25/02)

…（略）…
 ----------------------------
 Debugger: Disabled
 ----------------------------
  1 - Enable/disable tests
  2 - Enable/disable boot devices
  3 - Change IP addresses
  4 - Ping test
  5 - Toggle ROM monitor debugger
  6 - Toggle automatic menu
  7 - Display configuration
  8 - Save changes to configuration
  9 - Set baud rate for s1 boot
  A - Enable/disable I cache (Enabled )
  B - Enable/disable D cache (Enabled )
  F - FLASH image update
  M - Memory Test
  0 - Exit menu and continue
->
```

何やらメニューが出てきました．これです，欲しかったのは．

一通りメニューをいじってみると

```
  2 - Enable/disable boot devices
```

でフラッシュROM からのブートを無効にして

```
  0 - Exit menu and continue
```

でメニューを抜けると，bootp と tftp を使ってターゲットイメージを読み込んで，フラッシュROM には書かずにそのまま起動してきました．

以上の結果を表 6.2 にまとめます．

表 6.2: OpenBlockS266 SW3 設定

#1	#2	説明	備考
off	off	運用	フラッシュROM ブート
off	on	フラッシュROM 書き換え	bootp と tftp
on	off	対話モード	ドキュメントに記載無し
on	on	工場出荷試験	ドキュメントに記載無し

6.4　転送するイメージの形式

さて，このプライマリブートローダがロードするバイナリの形式はどうなっているのでしょうか．

6.4.1　適当なファイル

ためしに適当なバイナリを/tftpboot/zImage.treeboot に指定してみましょう．

```
->0
Booting from [ENET] Ethernet     ...
Sending bootp request ...
Loading file "/tftpboot/zImage.treeboot" ...
Sending tftp boot request ...
Not a valid boot image file
Booting from [S1] Serial Port 1...

PLEASE NOTE:  You must now...


       a. Exit from terminal emulation mode

       b. Modify the baud rate of your host session

       c. Transmit a file to the target in binary mode

       d. Reset the host baud rate to 9600

       e. Reenter terminal emulation mode

       f. Hit enter to execute the downloaded program
```

Not a valid boot image file なんていうメッセージが出ています．ということは，なんらかの形式があるようです．そもそも，この手順ではロードアドレスを入力する箇所がありません．この「形式」では，ロードアドレスも埋め込まれているのでしょう．

「この形式」では説明上不便なので，zImage.treeboot の形式をこの文書では **treeboot 形式**と呼ぶことにします．一般的な名称でない（と思う）ので注意してください．

6.4.2　linux カーネルを見る

というわけで，zImage.treeboot をビルドした linux のソースファイルを追ってみます．といっても Makefile だけですが．

linux-2.4.20/arch/ppc/boot/simple/Makefile に以下のような記述があります．

```
MKTREE                          := ../utils/mktree
…(略)…
zvmlinux: $(obj-y) $(LIBS) ../ld.script ../images/vmlinux.gz
../common/dummy.o
        $(OBJCOPY) $(OBJCOPY_ARGS) \
                --add-section=.image=../images/vmlinux.gz \
                --set-section-flags=.image=contents,alloc,load,readonly,data \
                ../common/dummy.o image.o
        $(LD) $(LD_ARGS) -o $@ $(obj-y) image.o $(LIBS)
        $(OBJCOPY) $(OBJCOPY_ARGS) $@ $@ -R .comment -R .stab -R
.stabstr \
                -R .ramdisk -R .sysmap
…(略)…
zImage: $(ZIMAGE)
        mv zvmlinux ../images/zImage.elf
…(略)…
zImage-TREE: zvmlinux
        $(MKTREE) zvmlinux ../images/zImage.treeboot
```

ここから

- zImage.treeboot ファイルは，zvmlinux ファイルから mktree コマンドによって生成される
- zvmlinux ファイルは elf 形式のファイルで，linux/arch/ppc/boot/images/zImage.elf にリネームされている

ということがわかります.

6.4.3 ブートイメージの捏造

それでは，適当な elf 形式ファイルから treeboot 形式のファイルを生成し，プライマリブートローダが受け入れるかどうか試してみます.

まず，母艦 PC 上で以下のようなダミーのプログラムを作成します.

```
main ()
{
        int a = 0;
}
```

そして，コンパイルし，elf 実行形式を生成します.

```
$ gcc -O0 -static hello.c
```

先ほどの mktree コマンドですが，使いかたは

```
$ mktree --help
usage: mktree <zImage-file> <boot-image> [entry-point]
```

ということなので，

```
$ mktree a.out zImage.treeboot 0x500000
```

と，treeboot 形式を生成します．これを/tftpboot ディレクトリにコピーし，プライマリブートローダから読み込ませてみます．

```
->0
Booting from [ENET] Ethernet     ...
Sending bootp request ...
Loading file "/tftpboot/zImage.treeboot" ...
Sending tftp boot request ...
Transfer Complete ...
Loaded successfully ...
Entry point at 0x500000 ...
```

元となったファイルが x86 の実行形式なので，もちろんマトモに実行できるはずはありません．が，プログラムをロードして開始アドレスにジャンプするところまでは動いているようです．今のところはこれでよし，とします．

第7章　橋頭堡

Memory space, the final frontier.

現在の状態では，まだ何もプログラムを動かしていません．まずは橋頭堡を確保し，ターゲットボードのメモリ領域を開拓していきましょう．

7.1　hello,world の意味

「プログラミング言語 C」に始まり，大抵のプログラミング言語の最初のステップは “hello,world” であることが多いようです．つまり，最初にプログラムからの出力を学ぶわけです．なぜでしょう？

仮に入出力が全く無いコンピュータを考えてみましょう．入出力が全く無いので，内部でどんな計算をやっているのか人間が知ることは不可能です．もちろん計算結果を得ることもできません．たとえ宇宙の誕生の神秘を解くような素晴らしいシミュレーションを実行していたとしても，熱を発生するだけの電熱器と一緒です．

それでは，入力と出力はどっちが大事なのでしょうか．Jargon には write only memory という話があります．書き込み専用のメモリです．もちろんジョークなのですが，なぜジョークかというとそれは簡単.「役に立たない」からです．write only memony は書き込み専用のメモリ，入力のみで出力の無いデバイスです．入力のみで出力が無い情報処理装置というのは，モノとして意味を為さないのです．

逆に入力が無く出力のみの装置というのはどうでしょう？例えば電源を入れると大砲の弾の軌道を計算し，結果を出力する装置を考えてみましょう．これは意味がありますね．

つまり，入力と出力では出力のほうが大事なのです．意味のあるプログラムを記述するには，最低限，出力処理ができなくてはいけません．プログラミング言語入門の最初が “hello,world” なのはこのためなのではないか，と私は思います．

7.2　LED が hello,world

5.2.1 節で述べたように，一般的な組み込みボードの場合，パソコンのようなディスプレイやキーボードが接続できるものは稀です．また，接続できる場合でも，ディスプレイに文字を出力するにはかなりの量のプログラムが必要です．

というわけで，組み込みボードでの “hello,world” は何かと考えると,「LED の点滅」がそれに相当するのではないかと思います．ほとんどのボードには状態表示用の LED が付いていますし，付いていない場合でもデバッグ用に LED を接続できるピンが少なくともスルーホール（ビア）として基板上に用意されいるでしょう[1].

[1] それすら無い場合は，ボードの基板設計者やプロジェクトマネージャや，企画に参加した己を呪ってください :)

もちろん，hello,world に比べればコードの量は多いです．が,「ブートローダを書こう」などと思い立つ人が C の初心者であるはずもない（というか，本書の対象外）ので，それ自身は問題は無いでしょう．

で，LED の点滅を制御できるとコンピュータから「プログラムが動いているぞ」という，最低限のシグナルを外界に発信することできます．シリアルコンソールが使えるようになるまでのデバッグ手段としても有効ですし，シリアルコンソールが使えるようになっても動作チェックの役に立つでしょう．

7.3　OpenBlockS266 では

PCC2000 ハードウェア機能仕様書によると，LED は

- GPIO 12, 13, 14 に接続
- 0 で点灯，1 で消灯

とあります．

7.4　GPIO

GPIO(General Purpose I/O) とは，一般的に次のような機能を持つ端子です．GPIO は SoC だけでなく，IDE コントローラなどの周辺チップにも実装されていることもあります．

7.4.1　入力と出力

レジスタの設定により，入力と出力を切り替えることができます．

出力に設定した場合は，レジスタの設定により特定のピンの出力値を H レベルまたは L レベルに固定することができます．入力に設定した場合は，特定の GPIO 端子にかかっている電圧が H レベルか L レベルかを，レジスタの値を読むことにより調べることができます．

図 7.1 と図 7.2 に GPIO の入出力の等価回路を示します．プログラミングにおいてはこのレベルの理解で十分でしょう．

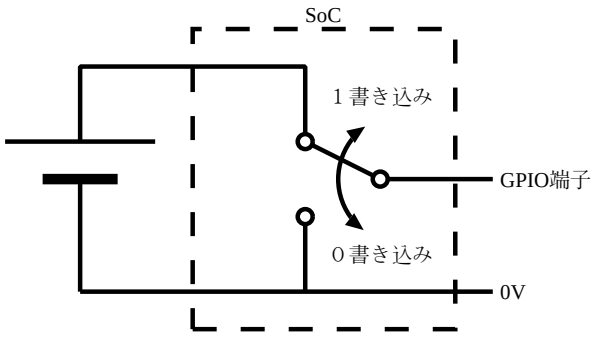

図 7.1: GPIO 出力

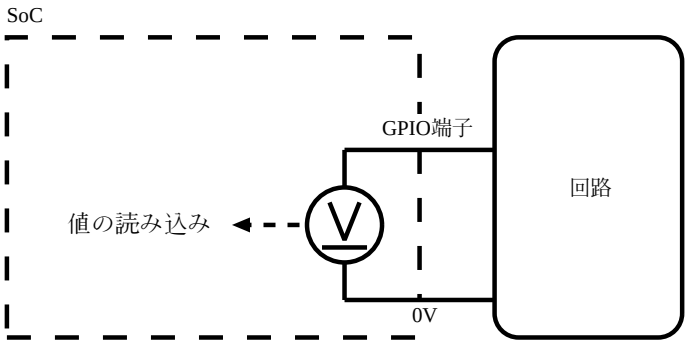

図 7.2: GPIO 入力

7.4.2 割り込み

入力に設定した場合，レベルの変化（もしくはあるレベルになったとき）に割り込みを発生させることができるものもあります.

7.4.3 PWM

出力に設定した場合，設定した周期で変化する方形波を出力できるものもあります．さらに，出力の H レベルと L レベルの比率 (**デューティ比**) も設定できるものもあります．この場合はモータなどの **PWM**(Pulse Width Modulation) 制御に利用することができます．LED を接続していれば，CPU 負荷無しにソフトウェアで輝度調整をすることもできるでしょう.

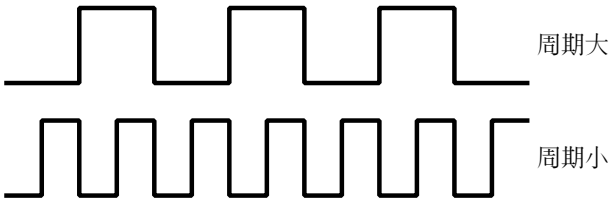

図 7.3: PWM パルス周期

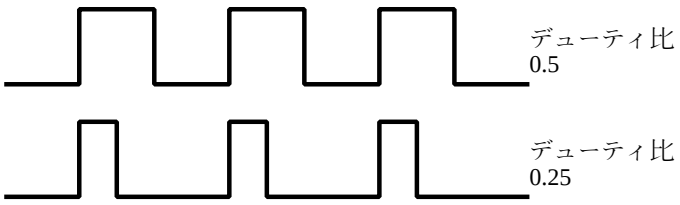

図 7.4: PWM デューティ比

7.4.4　正論理と負論理

端子の極性を**正論理**だけでなく，**負論理**に設定できるものも見たことがあります．

図 7.1 に正論理と負論理での電圧値と論理値の対応を示します．あくまでも H レベル/L レベルは電圧値であり，それを論理値の 1/0 にどう割り当てるかは任意です．回路によっては，正論理よりも負論理で設計したほうが簡単になる場合があります．また，40 シリーズや，74 シリーズのような標準ロジック IC を使用して外付け回路を組む場合，空きゲートを有効利用するために正論理と負論理を適宜使い分けるというテクニックもあります．

表 7.1: 正論理と負論理

論理	論理値	端子電圧
正論理	1	H
	0	L
負論理	1	L
	0	H

7.5　GPIO の叩きかた

PPC405GPr Embedded Processor User's Manual, 23.5 GPIO Register Overview より，GPIO レジスタのアドレスを表 7.2 に示します．

表 7.2: GPIO Register Summary

Mnemonic	Address	Access	Description
GPIO0_OR	0xEF600700	R/W	GPIO Output
GPIO0_TCR	0xEF600704	R/W	GPIO Three-State Control
GPIO0_ODR	0xEF600718	R/W	GPIO Open Drain
GPIO0_IR	0xEF60071C	R	GPIO Input
Note: GPIO registers are memory-mapped and accessed using load/store instructions at the register address.			

で，各レジスタですが，大雑把に説明すると

GPIO0_OR GPIO に出力したい値を書き込む (正論理)

GPIO0_TCR GPIO の入出力を指定する．1 ならば出力，0 ならば入力

GPIO0_ODR GPIO 出力のオープンドレイン設定．1 で有効

GPIO0_IR GPIO 入力値を読み出す

となります．

これらの動作を表 7.3 にまとめます．

ということは，GPIO レジスタのビット操作は

表 7.3: GPIO0_ODR Control Logic(一部編集)

GPIO0_ODR	GPIO0_OR	GPIO0_TCR	出力	LED
0	x	0	Z	–
0	0	1	L	点灯
0	1	1	H	消灯
1	0	x	L	–
1	1	x	Z	–

- GPIO0_ODR は 0 に，GPIO0_TCR は 1 に固定
- GPIO0_OR を 1 で消灯・0 で点灯

ということになります.

あと，マニュアルの 23.5.2 Detailed Register Description を見ると，各レジスタのビットの割り振りは MSB から数えて n ビット目が GPIOn に対するビットになるそうです．つまり，GPIO 12 に対するビットは `1<<(31-12)` です．`1<<12` ではないので注意しましょう.

7.6 仕様を考える

プログラムの流れはこんな感じで行ってみましょう.

1. GPIO0_ODR は 0 に，GPIO0_TCR は 1 に設定
2. 以下を繰り返す
   (a) GPIO0_OR を 1 に設定
   (b) 一定時間待つ
   (c) GPIO0_OR を 0 に設定
   (d) 一定時間待つ

ん，待てよ.

1. GPIO0_ODR は 0 に，GPIO0_TCR は 1 に設定
2. 以下を繰り返す
   (a) GPIO0_OR を XOR 演算で反転
   (b) 一定時間待つ

のほうがすっきりしますね．これでいきましょう.

7.7　アセンブリ言語

電源投入やリセット直後の，ブート直後の SoC は，一般的には

- プログラムカウンタはリセット用のアドレスに設定
- リセット用のアドレスには ROM がマップされている
- RAM は使用不可

という状態になります．RAM は使用できないので，スタックを使用することができません．ということは，C 言語で書いたプログラムは動きません．

ということは，ブート直後に動かすルーチンは**アセンブリ言語**で記述する必要があります．もちろん，RAM を使用できない，という縛りは C と同様です．

7.7.1　CPU レジスタへの即値のロード

新しい CPU をいじるときにまずつまずくのは，ここ．気合いを入れてマニュアルを読めばよいのですが，ものぐさなワザを 1 つ．

> gcc に訊け

これです．

では，やってみましょう．まず，こんな C プログラムを書き，ask_gcc.c という名前で保存します．

```
void loadlong ( void )
{
  unsigned a;

  a = 0x12345678;
}
```

コンパイルします．-O0(オー・ゼロ) オプションで，最適化を抑止するのがポイント．目的の部分はプログラムとしては無意味な処理なので，ここが最適化で消えてしまうのを防止します．

```
$ powerpc-linux-gcc -s -O0 ask_gcc.c ↵
```

ask_gcc.s というアセンブリ言語ファイルができました．

```
        .file   "ask_gcc.c"
        .section        ".text"
        .align 2
        .globl loadlong
        .type   loadlong, @function
loadlong:
```

```
        stwu 1,-32(1)
        stw 31,28(1)
        mr 31,1
        lis 0,0x8765
        ori 0,0,17185
        stw 0,8(31)
        lwz 11,0(1)
        lwz 31,-4(11)
        mr 1,11
        blr
        .size   loadlong, .-loadlong
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.4.4 20041218 (prerelease) (Debian 3.4.3-1)"
```

この出力は as へ入力可能のものです．が，レジスタの指定もただの数字なので読みづらいですね．そこで別の方法をもう 1 つ．

```
$ powerpc-linux-gcc -c -O0 ask_gcc.c ↵
```

で，ask_gcc.o ファイルを作ります．そして，

```
$ powerpc-linux-objdump -d ask_gcc.o ↵
```

で逆アセンブルします．

```
ask_gcc.o:     ファイル形式 elf32-powerpc

セクション .text の逆アセンブル:

00000000 <loadlong>:
   0:   94 21 ff e0     stwu    r1,-32(r1)
   4:   93 e1 00 1c     stw     r31,28(r1)
   8:   7c 3f 0b 78     mr      r31,r1
   c:   3c 00 87 65     lis     r0,-30875
  10:   60 00 43 21     ori     r0,r0,17185
  14:   90 1f 00 08     stw     r0,8(r31)
  18:   81 61 00 00     lwz     r11,0(r1)
  1c:   83 eb ff fc     lwz     r31,-4(r11)
  20:   7d 61 5b 78     mr      r1,r11
  24:   4e 80 00 20     blr
```

こちらのほうが読みやすいですね．ただし，こちらの形式はそのままではアセンブラは理解できないので注意．

で，CPU レジスタへの即値のロードは

```
lis     rX, 即値の上位 16bit
ori     rX,rX, 即値の下位 16bit
```

で良いことがわかりました．

32bit 即値のロードは 1 命令ではできず，上位・下位に分けて 2 命令必要になっています．これは PowerPC アーキテクチャだけでなく，ほとんどの RISC 系 CPU に共通です．というのも，RISC 系 CPU では，1 命令長は固定です．大抵はオペコードとオペランドを合わせて 32bit（か，それ以下）となっています．ここに 32bit の即値データを格納すると…オペコードが入る場所がありませんね．

あと，おまけとして，ディスプレースメント付きのレジスタ間接アドレッシング．

```
lwz     rX,N(rY)
```

で，アドレス (rY + N) からレジスタ rX に値をロードできることがわかります．

7.7.2 info

gcc や as などのオンラインマニュアル（man コマンド）を見ると

> 正式なドキュメントである Info ファイルのほうを参照して下さい。

とあります．この **info** について，as コマンドを例に見ていきましょう．

`as` についての正式なマニュアルは通常，info 形式のファイルでインストールされます（別パッケージで提供されている場合もあります）．そして，info 形式のマニュアルは `info` コマンドで閲覧することができます．実際に見てみましょう．

```
$ info -f as ↵
```

フルスクリーンで info の画面が現れました．図 7.5 に info の起動画面を示します．

カーソル操作は emacs と同様に Ctrl + P, Ctrl + N, Ctrl + F, Ctrl + B や Ctrl + V, Esc V で行うことができます．端末によってはカーソルキーを使用することもできます．

`* Overview::` のような表示はハイパーリンクです．info は，複数のページから構成されています．この上にカーソルがあるときに Enter キーを押すことにより，リンク先のページへとジャンプします．web と同じ要領ですね．また，Tab で次のハイパーリンクへカーソルが移動します．

ところでこの文書は「章節の順番」「章節の階層」という構造を持っています．同様に info でも異なり「ページの順番」「ページの階層」という概念があります．Space キーにより，ページを順番に閲覧することができます．また，U, P, N でそれぞれ上の階層，前のページ，次のページへジャンプします．

あと，マニュアルを参照するときに欠かさないのが単語の検索です．info では Ctrl + S でインクリメンタルサーチを行うことができます．

以上のキー操作を表 7.4 に示します．

```
File: as.info,  Node: Top,  Next: Overview,  Up: (dir)

Using as
********

This file is a user guide to the GNU assembler ‘as’ version 2.17.

   This document is distributed under the terms of the GNU Free
Documentation License.  A copy of the license is included in the
section entitled "GNU Free Documentation License".

* Menu:

* Overview::                    Overview
* Invoking::                    Command-Line Options
* Syntax::                      Syntax
--zz-Info: (as.info.gz)Top, 26 行 --Top-----------------------------------------
Info バージョン 4.8 にようこそ。? で使い方、m でメニュー項目を呼び出せます。
```

図 7.5: info 画面

表 7.4: info 主なキー操作

キー	説明
Ctrl + p	カーソル上
Ctrl + n	カーソル下
Ctrl + f	カーソル右
Ctrl + b	カーソル左
Ctrl + v	スクロール
Esc v	バックスクロール
Space	スクロール・ページ末で次ページ
Enter	リンク先へジャンプ
u	上の階層のページへジャンプ
p	前のページへジャンプ
n	次のページへジャンプ
Ctrl + s	インクリメンタルサーチ

7.7.3　疑似命令

疑似命令とは，アセンブリ言語において出力には直接表れないアセンブラへの指令のことです．ask_gcc.s のほうをもう少し見てみましょう．まず .section 命令．

```
        .section        ".text"
```

アセンブラの出力は ELF ファイルです．ELF ファイルの中には，**セクション**といって，いろいろなオブジェクトを分類して格納することができます．

セクション名には任意の名前を付けることができますが，一部の名前のセクションは OS のローダ（やブートローダ）で特別な意味を持っています．そのようなセクションを表 7.5 にまとめます．

表 7.5: 主なセクション名と内容

セクション名	C コード例	説明
text	`a = b;`	プログラムコード
bss	`static char a[1024];`	データ領域．プログラム開始前にローダ側で 0 クリアする
data	`static char b[] = "abcdefg";`	書き換え可能のデータ領域

次は .globl 疑似命令．この疑似命令によって，定義されたシンボルがオブジェクトファイルの外部から参照可能となります．C では static 宣言の無い変数・関数に当たります．

7.8　プログラミング

下調べも一通りできたので，プログラムを組んでいきましょう．

7.8.1　ファイル名

まず，プログラムファイルは .S の付いた名前で作成します．7.7.1 節での .s ファイルの場合は，直接 as が解釈可能な形である必要があります．が，.S ファイルは，cpp プリプロセッサでマクロ展開された後に as へと渡されます．つまり，.S ファイルでは C プログラミングでおなじみの #define や#include などのプリプロセッサ指令を使うことができます．

というわけで，ファイル名を led.S とします．

7.8.2　マクロ

7.7.1 節のように，as が直接受け付ける形式では，CPU レジスタは 0 のように数字のみです．プリプロセッサで定義を加え，これを r0 のように記述できるようにしましょう．

```
#define r0      0
#define r1      1
…（以下同文）…
#define r31     31
```

おちゃのこさいさいですね.
それでは次に GPIO レジスタのアドレスを定義しましょう.

```
#define GPIO0_base      0xEF600000      /* ベースアドレス */
#define GPIO0_OR        0xEF600700      /* R/W: GPIO Output */
#define GPIO0_TCR       0xEF600704      /* R/W: GPIO Three-State Control */
#define GPIO0_ODR       0xEF600718      /* R/W: GPIO Open Drain */
#define GPIO0_IR        0xEF60071C      /* R:   GPIO Input */
```

こんな感じ.
GPIO レジスタを操作するためのビット列も定義しておきましょう.

```
#define LED1_N  12
#define LED2_N  13
#define LED4_N  14

#define LED1_BIT        (1<<(31-LED1_N))
#define LED2_BIT        (1<<(31-LED2_N))
#define LED4_BIT        (1<<(31-LED4_N))
```

このようなビット列の定義の場合，直接 16 進数で定義してもプログラム上の動作は一緒なのですが，ビットシフト演算子を使用すれば人間にとって可読性の高い表現になります．また，GPIO レジスタ上で GPIOn に対応するビットは MSB から数えて n ビットめ，なのでこうなります.

7.8.3 コーディング

ここでおもむろにレジスタの割り振りを考えてみます．実は PowerPC の汎用レジスタは，r0 のみに対称性が無く，`lwz rX, Y(r0)` のような記述ができません．オフセット 0 の `lwz rX, Y(0)` と解釈されてしまいます．というわけで，r0 は未使用とします.
GPIO0_* レジスタ群の上位 16 bit のベースアドレスは r1 に設定することにします.

```
lis     r1,GPIO0_base@h
```

上位 16 bit のみなので，ori で下位 16 bit を設定するのは省略できます．あと GPIO0_base@h という表記ですが，定数の上位 16 bit のみを取り出す as の機能です．ここでの意味としては C での GPIO0_base>>16 と同様です．同様に C での GPIO0_base & 0x0000ffff に相当する GPIO0_base@l という表記もあります．

GPIO0_ODR で，LED1 に対応するビットを 0 に設定してみましょう．普通に考えれば，

1. GPIO レジスタの値を CPU レジスタにに読み出し
2. 即値のビットマスクと and を取り
3. GPIO レジスタに書き戻す

でいけそうです．

```
/* 誤り */
lwz     r2,GPIO0_ODR@l(r1)
andis.  r2,r2,(~LED1_BIT)@h
andi.   r2,r2,(~LED1_BIT)@l
stw     r2,GPIO0_ODR@l(r1)
```

が，andis. 命令は即値の下位ビットを 0 でパディングしてしまうので，これではうまくいきません．7.7.1 節の要領で gcc に訊いてみると rlwinm という命令を使うといいようです．が，人が読むにはいまひとつ理解しづらい命令なのでここでは使わないことにします．速度やメモリに厳しい制限があるわけでもないので．

というわけで，CPU レジスタ間の演算をする and 命令を使って書くことにします．

1. r2 にビットマスクを設定
2. r3 に GPIO レジスタの値を読み込む
3. r3 と r2 のビットマスクの and を取り
4. r3 を GPIO レジスタに書き戻す

という手順でいきましょう．

```
lis     r2,(~LED1_BIT)@h
ori     r2,r2,(~LED1_BIT)@l
lwz     r3,GPIO0_ODR@l(r1)
and     r3,r3,r2
stw     r3,GPIO0_ODR@l(r1)
```

できました.

次は GPIO0_TCR で，LED1 に対応するビットを 1 に設定してみましょう．今度は

1. r3 に GPIO レジスタを読み出して
2. r3 とビットパターンの or を取って
3. r3 を GPIO レジスタに書き戻す

という手順でいけます．PowerPC の命令セットでも問題はないようです.

```
lwz r3,GPIO0_TCR@l(r1)
oris r3,r3,LED1_BIT@h
ori r3,r3,LED1_BIT@l
stw r3,GPIO0_TCR@l(r1)
```

できました.

GPIO0_OR の LED1 のビットを反転する部分を書いていきましょう．xoris 命令もここで期待している通りに，下位ビットを 0 で埋めてくれるので，期待した通りにコーディングすることができます.

```
        lwz     r3,GPIO0_OR@l(r1)
        xoris   r3,r3,LED1_BIT@h
        xori    r3,r3,LED1_BIT@l
        stw     r3,GPIO0_OR@l(r1)
```

次は「時間つぶし」です．ここでは**ビジーループ**を使用します．ビジーループとは，空ループを実行することによって一定時間の遅延を作るという手法です．が，最近では邪悪な手法とされています.

というのも，まずマルチタスク OS においてはビジーループは CPU 資源の無駄使いである，ということが挙げられます．明示的にタスクスイッチを行って，他のプロセスに CPU を明け渡したほうがシステム全体のためになります.

また，環境によって得られる遅延の大きさが不安定，ということもあります．CPU をより処理速度の高いものにアップグレードしてしまうと，ビジーループによる遅延は小さくなります.

が，今回は

- マルチタスク環境ではない
- 環境は OpenBlockS266 決め打ちである

ということから，ビジーループを使用します．

```
        lis     r4,0x0200
        ori     r4,r4,0x0000
2:
        addic.  r4,r4,-1
        bne     2b
```

r4 に 0x02000000 を代入し，1 ずつカウントダウンしていきます．0 になったらループ脱出です．

ここで 2: というのはラベルです．数字のみのラベルの場合，有効範囲は関数内部のみで外部には出ません．このループは関数内部で閉じているのでここでは数字を使います．

addic．というのは，即値の加算命令です．−1 を足すことにより引き算を行っています．また，．（ピリオド）は，演算結果をフラグレジスタに反映させる，という意味です．これを付け忘れると後の命令で条件判定ができません．

bne 2b ですが，bne は branch not equal，つまり 0 以外ならば分岐せよ，という条件分岐命令です．で，2b ですが，これは先ほどの 2 のラベルを指しています．b は backward，つまりプログラムを後戻りする方向のラベルを探せ，という意味です．

時間つぶしのループを抜けた後は，ビット反転の前にジャンプしなくてはいけません．コードは

```
1:
        …（略）…
        b       1b
```

ですね．

7.9　アセンブル

プログラムができたので，アセンブルしてみます．

```
$ powerpc-linux-gcc -c led.S ↵
```

gcc コマンドでアセンブルも行うことができます．実は裏で as を呼び出しているのですが，普段は気にする必要は無いでしょう．

で，led.o ファイルができました．これを実行ファイルに変換します．

```
$ powerpc-linux-ld -o led.elf led.o -s -Ttext 0x500000 ↵
```

今度は gcc を使用することはできません．gcc でリンクすると，何も指定しなくても libc をリン

クしようとするためです．この環境に入っている libc は PowerPC linux 上のプログラムのためのもので，OS 無し環境がターゲットの場合は使うことができません．

で，libc 抜きでリンクしたいので，ld を直接呼び出します．-s は，余計なシンボルテーブルを削除するオプションです．-Ttext 0x500000 で，text セクションを 0x500000 に配置するよう指定します．

で，led.elf ができました．これを treeboot 形式に変換します．

```
$ mktree led.elf led.treeboot ↵
```

led.treeboot ファイルができました．

これらの手順を Makefile に記述しておいて，make コマンド一発で led.treeboot まで自動生成するようにします．

```
led.treeboot: led.elf
        mktree led.elf led.treeboot

led.elf: led.o
        powerpc-linux-ld -o led.elf led.o -s -Ttext 0x500000

led.o: led.S
        powerpc-linux-gcc -c led.S
```

コマンド行の先頭は空白ではなく tab です．ここを空白にすると正常に動作しないので注意しましょう．

7.10 実行

led.treeboot を /tftpboot/zImage.treeboot にコピーして実行します．

6.3 節の手順に従って，/tftpboot/zImage.treeboot を OpenBlockS266 に転送し，実行させます．

シリアルコンソールにメニューが出たら 2 1 ↵ 0 で bootp + tftp でブートを開始します．

```
Booting from [ENET] Ethernet     ...
Sending bootp request ...
Loading file "/tftpboot/zImage.treeboot" ...
Sending tftp boot request ...
Transfer Complete ...
Loaded successfully ...
Entry point at 0x500000 ...
```

LED 1 が点滅してるでしょうか？LED の点滅を十分堪能したら，AC アダプタを外して終了させましょう．無限ループなので，それ以外に終わらせる手段はありません．

と，当初の目的通り LED を点滅させることができたので，この章はこれでおしまい．

第8章　C の世界へ

前の章ではアセンブリ言語でのプログラミングを体験しました．が，ブートローダ全体をアセンブラで記述するのは大変です．というわけで，C でプログラムを書くことを考えてみます．

8.1　C プログラムに必要な環境

7.7.1 節の C プログラムの逆アセンブルコードをもう一度見てみましょう．

```
ask_gcc.o:     ファイル形式 elf32-powerpc

セクション .text の逆アセンブル:

00000000 <loadlong>:
   0:   94 21 ff e0     stwu    r1,-32(r1)
   4:   93 e1 00 1c     stw     r31,28(r1)
   8:   7c 3f 0b 78     mr      r31,r1
   c:   3c 00 87 65     lis     r0,-30875
  10:   60 00 43 21     ori     r0,r0,17185
  14:   90 1f 00 08     stw     r0,8(r31)
  18:   81 61 00 00     lwz     r11,0(r1)
  1c:   83 eb ff fc     lwz     r31,-4(r11)
  20:   7d 61 5b 78     mr      r1,r11
  24:   4e 80 00 20     blr
```

まず，CPU レジスタに注目してみましょう．r1, r31, r0, r11 が使われています．このうち，r1 だけが値のロード無しにいきなり参照されています．そして，他のレジスタは r1 の指すアドレスを基準にして値の退避や復元を行っています．そう，r1 は**スタックポインタ** (stack pointer) です．

で，スタックポインタが機能するためには，スタックポインタが指しているアドレスに RAM が存在していなくてはなりません．

つまり，C プログラムを動作させるには

- RAM を動作可能にする
- スタックポインタを設定する

ことが必要です．

8.2　プログラムの骨組み

それでは，プログラムの構造を考えてみましょう．

1. アセンブリ言語部
   (a) DRAM を使用可能にする
   (b) r1 にスタックポインタを設定し
   (c) C の関数 (bootloader_main) へジャンプ
2. C 言語部
   (a) bootloader_main 関数．無限ループ．

こんなところですね．

8.3　DRAM を使用可能に

本当のブートローダの場合，起動直後は DRAM は使用できない状態にあります．このため，DRAM コントローラのレジスタを設定して DRAM を使用できるようにしなくてはなりません．

が，今回はチェインローダであり，この処理はブートローダで行われています．また，チェインローダは DRAM 上にロードされているため，DRAM 関係のレジスタを操作すると，レジスタを操作しているプログラム自身がアドレス空間から一時的に見えなくなってしまうことも有り得ます．つまり，自分で自分の首を絞めるようなことになってしまいます．

というわけで，今回はこの処理は省略します．が，リセット直後に動作するホントのブートローダでは絶対に必要な処理です．お忘れなく．

8.4　スタックポインタの設定

r1 にスタックポインタを設定するには，7.7.1 節で調べたように

```
lis     r1,stack_pointer@h
ori     r1,r1,stack_pointer@l
```

でできますね．

stack_pointer のシンボルの値は，後で定義します．

8.5 bootloader_main へジャンプ

これも

```
        b       bootloader_main
```

で ok.

bootloader_main のシンボルの値は，リンク時に解決されます.

8.6 C プログラム

C 部分のプログラムを組んでいきましょう．何もせずに無限ループを回すのも，動いているのかどうか確認できないので，前回と同様，LED を点滅させることにします.

8.6.1 C のポインタの正体

というわけで，GPIO レジスタを定義しましょう.

```
#define GPIO0_OR  (*(volatile unsigned *)0xEF600700)
#define GPIO0_TCR (*(volatile unsigned *)0xEF600704)
#define GPIO0_ODR (*(volatile unsigned *)0xEF600718)
#define GPIO0_IR  (*(volatile unsigned *)0xEF60071C)
```

アドレスは一緒なのだけど，アセンブラの時と定義がちょっと違いますね.

ここでみなさまに重大なお知らせがあります．アセンブリ言語と C を使える人ならば薄々勘づいていると思いますが，gcc では，ポインタの正体は論理アドレスそのものです.

例えば,

```
        unsigned *p = 0x12345678;
        *p = 0xdeadbeef;
```

で，0x12345678 番地へ 0xdeadbeef のライトアクセスをすることができます.

同様に

```
unsigned *p = 0x12345678;
unsigned d = *p;
```

で，0x12345678 番地からデータを読み込み，変数 d へ格納します．
　また，少々トリッキーですが

```
unsigned d = *((unsigned *)0x12345678);
*((unsigned *)0x12345678) = 0xdeadbeef;
```

のように書けば，ポインタ変数を使用せずにメモリ空間の特定のアドレスにアクセスすることができます．冒頭の #define の定義はこれを利用しています．

8.6.2　型修飾子 volatile

　よく見ると，#define のレジスタ定義には **volatile** なんてキーワードが付いてますね．この volatile は,「変数操作の最適化を抑制せよ」と C コンパイラに指示する型修飾子です．
　まず，C コンパイラは，ポインタの指し示す先にあるものはメモリであると思い込んでいます．そして，その思い込みを元に最適化を行います．例えば

```
unsigned mem;
unsigned *p = &mem;

*p = 1;
*p = 2;
*p = 3;
```

というコードについて考えてみましょう．
　RAM 上の同じアドレスに何回値を書き込んでも，結果は最後に書き込んだ値なので，結局は

```
unsigned mem;
unsigned *p = &mem;

*p = 3;
```

と同じことです．従って C コンパイラは，検出できる部分においてこのような最適化を行います．

I/O レジスタへの操作では，同じアドレスから連続して読み出したり，同じアドレスに連続して書き出したりすることがあります．UART の送受信データレジスタはまさにこれです．また，読み出しや書き込み操作を行うこと自体に意味があり，その値についてはどうでもいい，ということもあります．ここで

```
unsigned *p = (unsigned *)0x12345678;

*p = 'h';
*p = 'e';
*p = 'l';
*p = 'l';
*p = 'o';
```

というコードについて考えてみましょう．0x12345678 番地には UART のデータ出力レジスタがあるとします．このプログラムを書いた人は，きっと ‘‘hello’’ と出力されることを期待してるのでしょう．

ところが，ここでも C コンパイラは最適化を行います．0x12345678 番地は RAM だと思い込んでいるので，実行結果は最後に書き込んだ値になるはずだから，と

```
unsigned *p = (unsigned *)0x12345678;

*p = 'o';
```

のように最適化してしまいます．これでは ‘‘o’’ しか出力されません．困りましたね．

ここで volatile の登場です．

```
volatile unsigned *p = (volatile unsigned *)0x12345678;

*p = 'h';
*p = 'e';
*p = 'l';
*p = 'l';
*p = 'o';
```

変数 *p に対して最適化を抑制するようになるので，o の前の hell もちゃんと出力されるようになります．

8.6.3　流儀

#define で I/O レジスタを定義する場合,

```
#define REG_1a  ((volatile unsigned *)0x12345678)
#define REG_1b  (*(volatile unsigned *)0x12345678)
```

のように，ポインタとして定義する流儀と，そのポインタをデリファレンスしたものを定義する流儀があります．これらのマクロで例えばレジスタへのライトアクセスを記述すると

```
        *REG_1a = 0xdeadbeef;
        REG_1b = 0xdeadbeef;
```

のようになります.

どちらのスタイルで定義してもプログラミングには大した影響はありません．もちろん，パフォーマンスにも影響ありません．単純に好みの問題です．が，プログラム全体でスタイルは統一しておいたほうが良いでしょう．両方のスタイルが混在していると，プログラムを後で読み返したときに人間様のほうが混乱してしまいます.

今回はポインタをデリファレンスしたスタイルで定義します．というわけで，冒頭の #define の定義になるわけです.

8.6.4　GPIO レジスタの初期化

定義したマクロを使って GPIO レジスタを初期化します．初期化の内容は 7.5 節と一緒です.

```
        GPIO0_ODR &= ~LED1_BIT;
        GPIO0_TCR |= LED1_BIT;
```

こんな感じ．C が使えると楽ですね.

8.6.5　LED の点滅

LED を点滅させる部分を書いていきましょう．アセンブリ言語の場合と同様，ビット反転させます.

```
        GPIO0_OR ^= LED1_BIT;
```

1 行で書けてしまいました.

8.6.6 時間つぶし

これもアセンブリ言語のときと同様，ビジーループでいきます.

```
        volatile int n;
        …
        for ( n = LOOP_COUNT; n > 0; --n )
                ;
```

C コンパイラの最適化ではしょられてしまわないよう，volatile 修飾した変数で空ループを回します.

8.7 コンパイルとリンク

8.7.1 コンパイル

アセンブリ言語部は

```
$ powerpc-linux-gcc -c head.S
```

C 言語部は

```
$ powerpc-linux-gcc -c led.c
```

でコンパイルします．ここらはアセンブリ言語の場合と一緒ですね.

8.7.2 リンカスクリプト

で，お次は head.o と led.o をリンクします．ここでは**リンカスクリプト** (linker script) を使ってみることにします.

まずは，リンカスクリプトの使いかたを調べましょう info -f ld で ld のマニュアルを読んでみます．すると，リンカスクリプトの例が上がっています．あとは，Linux カーネルのリンクでもリンカスクリプトは使われているので，これも参考になります.

まず，プログラム領域・データ領域の配置は

```
SECTIONS
{
        . = 0x500000;
        .text : { *(.text) }
        .data : { *(.data) }
        .bss : { *(.bss) }
}
```

指定できます.

. = 0x500000; で配置する開始アドレスを指定します. そのあとに .text （プログラム）領域, .data（データ）領域, .bss 領域を続けて配置します.

あと, プログラムの開始番地が _start のラベルのアドレスであることをリンカに伝えなくてはなりません. これは

```
ENTRY(_start);
```

で指定することができます.

それから 8.4 節で宿題になっていたスタックポインタの初期値も指定します.

```
stack_pointer = 0x1000000;
```

以上をまとめると

```
ENTRY(_start);
SECTIONS
{
        . = 0x500000;
        .text : { *(.text) }
        .data : { *(.data) }
        .bss : { *(.bss) }
}
stack_pointer = 0x1000000;
```

こんな感じになります. bootloader.lds という名前で保存しておきましょう.

8.7.3 リンク

作成したリンカスクリプトを使ってリンクします.

```
$ powerpc-linux-ld -T bootloader.lds -o led.elf head.o led.o
```

リンカスクリプトは -T オプションで指定します.

あとは，アセンブリ言語のときと同様，elf 形式のファイルを treeboot 形式に変換します.

```
$ mktree led.elf led.treeboot
```

8.8 実行

7.10 節と同様の手順で led.treeboot を OpenBlockS に転送し，実行してみます.

アセンブリ言語のときと比べて LED の点滅がゆっくりですが，動きましたね.

というわけで，本章はおしまい.

第9章　つづく

以降，つづく予定.

付 録A　ソースコード

A.1　橋頭堡

A.1.1　Makefile

```
led.treeboot: led.elf
        mktree led.elf led.treeboot

led.elf: led.o
        powerpc-linux-ld -o led.elf led.o -s -Ttext 0x500000

led.o: led.S
        powerpc-linux-gcc -Wa,-m405 -c led.S
```

A.1.2　led.S

```
/*
 * 橋頭堡: LED の点滅
 *
 * Copyright (C) 2006 by Kunihiko IMAI <bak @ d2.dion.ne.jp>
 *
 * This software is provided 'as-is', without any express or implied
 * warranty.  In no event will the authors be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented ;  you must not
 *    claim that you wrote the original software. If you use this software
 *    in a product, an acknowledgment in the product documentation would be
 *    appreciated but is not required.
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
 *
 */

/*
 * GPIO レジスタのアドレス
 *
```

```
 * PPC405GPr Embedded Processor User's Manual
 * 23.5 GPIO Register Overview より
 */
#define GPIO0_base      0xEF600000      /* ベースアドレスの上位 16bit */

/* GPIO0 からの差分 */
#define GPIO0_OR  0xEF600700    /* R/W: GPIO Output */
#define GPIO0_TCR 0xEF600704    /* R/W: GPIO Three-State Control */
#define GPIO0_ODR 0xEF600718    /* R/W: GPIO Open Drain */
#define GPIO0_IR  0xEF60071C    /* R:   GPIO Input */

#define r0      0
#define r1      1
#define r2      2
#define r3      3
#define r4      4
#define r5      5
#define r6      6
#define r7      7
#define r8      8
#define r9      9
#define r10     10
#define r11     11
#define r12     12
#define r13     13
#define r14     14
#define r15     15
#define r16     16
#define r17     17
#define r18     18
#define r19     19
#define r20     20
#define r21     21
#define r22     22
#define r23     23
#define r24     24
#define r25     25
#define r26     26
#define r27     27
#define r28     28
#define r29     29
#define r30     30
#define r31     31

/*
 * LED ポートの定義
 *
 * PCC2000 ハードウェア機能仕様書
 * 4. LED より
 *
 * 0 で点灯, 1で消灯
 */
#define LED1_N  12
#define LED2_N  13
#define LED4_N  14

#define LED1_BIT        (1<<(31-LED1_N))
#define LED2_BIT        (1<<(31-LED2_N))
```

```
#define LED4_BIT        (1<<(31-LED4_N))

#define LOOP_COUNT      0x02000000      /* 時間つぶしループの回数 */

/*
 * メイン
 *
 * レジスタの割り振りは
 *
 * r0: 未使用
 * r1: GPIO0_* ベースアドレス
 * r2: ビットマスク
 * r3: GPIO レジスタの読み書き
 * r4: ループカウンタ
 */

        .text           0
        .globl          _start
_start:
        lis     r1,GPIO0_base@h

        lis     r2,(~LED1_BIT)@h
        ori     r2,r2,(~LED1_BIT)@l
        lwz     r3,GPIO0_ODR@l(r1)
        and     r3,r3,r2
        stw     r3,GPIO0_ODR@l(r1)

        lwz     r3,GPIO0_TCR@l(r1)
        oris    r3,r3,LED1_BIT@h
        ori     r3,r3,LED1_BIT@l
        stw     r3,GPIO0_TCR@l(r1)

1:
        /* 反転 */
        lwz     r3,GPIO0_OR@l(r1)
        xoris   r3,r3,LED1_BIT@h
        xori    r3,r3,LED1_BIT@l
        stw     r3,GPIO0_OR@l(r1)

        /* 時間つぶし */
        lis     r4,LOOP_COUNT@h
        ori     r4,r4,LOOP_COUNT@l
2:
        addic.  r4,r4,-1
        bne     2b

        b       1b

        .end
```

A.2 C の世界へ

A.2.1 Makefile

```
led.treeboot: led.elf
        mktree led.elf led.treeboot

led.elf: head.o led.o
        powerpc-linux-ld -T bootloader.lds -o led.elf head.o led.o

head.o: head.S
        powerpc-linux-gcc -c head.S

led.o: led.c
        powerpc-linux-gcc -c led.c
```

A.2.2 head.S

```
/*
 * C の世界へ:
 *
 * Copyright (C) 2006 by Kunihiko IMAI <bak @ d2.dion.ne.jp>
 *
 * This software is provided 'as-is', without any express or implied
 * warranty.  In no event will the authors be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented ;  you must not
 *    claim that you wrote the original software. If you use this software
 *    in a product, an acknowledgment in the product documentation would be
 *    appreciated but is not required.
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
 *
 */

#define r0      0
#define r1      1
#define r2      2
#define r3      3
#define r4      4
#define r5      5
#define r6      6
#define r7      7
#define r8      8
#define r9      9
#define r10     10
#define r11     11
#define r12     12
#define r13     13
#define r14     14
#define r15     15
```

```
#define r16     16
#define r17     17
#define r18     18
#define r19     19
#define r20     20
#define r21     21
#define r22     22
#define r23     23
#define r24     24
#define r25     25
#define r26     26
#define r27     27
#define r28     28
#define r29     29
#define r30     30
#define r31     31

        .text   0
        .globl  _start

_start:
        lis     r1,stack_pointer@h
        ori     r1,r1,stack_pointer@l

        b       bootloader_main

        .end
```

A.2.3 led.c

```
/*
 * C の世界へ: led.c
 *
 * Copyright (C) 2007 by Kunihiko IMAI <bak @ d2.dion.ne.jp>
 *
 * This software is provided 'as-is', without any express or implied
 * warranty.  In no event will the authors be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented ;  you must not
 *    claim that you wrote the original software. If you use this software
 *    in a product, an acknowledgment in the product documentation would be
 *    appreciated but is not required.
```

```
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
 *
 */

/*
 * GPIO レジスタのアドレス
 *
 * PPC405GPr Embedded Processor User's Manual
 * 23.5 GPIO Register Overview より
 */

/* R/W: GPIO Output */
#define GPIO0_OR  (*(volatile unsigned *)0xEF600700)
/* R/W: GPIO Three-State Control */
#define GPIO0_TCR (*(volatile unsigned *)0xEF600704)
/* R/W: GPIO Open Drain */
#define GPIO0_ODR (*(volatile unsigned *)0xEF600718)
/* R: GPIO Input */
#define GPIO0_IR  (*(volatile unsigned *)0xEF60071C)

/*
 * LED ポートの定義
 *
 * PCC2000 ハードウェア機能仕様書
 * 4. LED より
 *
 * 0 で点灯, 1で消灯
 */
#define LED1_N  12
#define LED2_N  13
#define LED4_N  14

#define LED1_BIT        (1<<(31-LED1_N))
#define LED2_BIT        (1<<(31-LED2_N))
#define LED4_BIT        (1<<(31-LED4_N))

#define LOOP_COUNT      0x02000000      /* 時間つぶしループの回数 */

void bootloader_main ( void )
{
        volatile unsigned n;
```

```
        /* GPIO レジスタの初期化 */
        GPIO0_ODR &= ~LED1_BIT;
        GPIO0_TCR |= LED1_BIT;

        for (;;) {
                /* LED の状態反転 */
                GPIO0_OR ^= LED1_BIT;

                for ( n = LOOP_COUNT; n > 0; --n )
                        /* 時間つぶし */
                        ;
        }
}
```

索 引