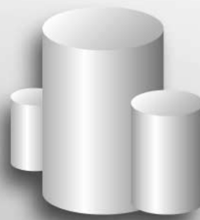
このPDFは，CQ出版社発売の「リンカ･ローダ実践開発テクニック」の一部分の見本です．
内容･購入方法などにつきましては以下のホームページをご覧下さい．
<http://shop.cqpub.co.jp/hanbai/books/38/38071.htm>

第1章

リンカとローダの役割



プログラムを書いてコンパイルした後，実行できるようになるまでの最終段階として「リンク」という過程があります．また，リンク後の実行形式をCPUが実行するためには，メモリにプログラムを読み込む「ロード」という作業が必要です．リンクやロードを行うアプリケーションを一般に「リンカ」，「ローダ」と呼びます．

コンパイル型のプログラミング言語では，ソース・コードをコンパイルした後に，複数のライブラリとリンクすることで，実行形式のプログラムを作成します．プログラムを実行できるようにするためには，リンクは必須の過程です．

また，実行形式のプログラムはファイル・システム上に通常は格納されていますが，CPUはメイン・メモリ上の実行コードしか実行できません．実行形式のプログラムを実行するためには，ファイル・システムからメイン・メモリへのロードが必須になります．しかし，汎用OSの上でアプリケーションを作成・利用しているだけならば，これらは暗黙のうちに行われるので，あまり意識しなくてもかまいません．

一般的なアプリケーション・プログラムならばそれでもよいのですが，組み込み系のプログラムやOSを作成/移植する際には，そうはいきません．これらの場合には，オブジェクトの配置を自由に行いたいために，リンカの知識やカスタマイズ，細かいチューニングが必須になってくるからです．

本書では，まず第1章で，リンカの一般的な動作について説明します．その次に，実験を通してさらに理解を深めます．単なる確認実験ではなく，リンカの機能を利用しないと実現できない(C言語の文法の範囲では実現できない)ようなテクニックを紹介します．最後に，実際にどのようなところで利用されているのか，どのようなことに利用できるのかを説明します．

なお，とくに断りのない限りは，FreeBSD-4.10を題材としています．コンパイラやリンカなどには，

見本

GNUプロジェクトのgccとbinutilsを利用しています．バージョンは，FreeBSD-4.10に付属しているgcc-2.95.4とbinutils-2.12.1です．また，一部ではNetBSDを参考にしていますが，これは原稿執筆時点(2004年10月3日)でのNetBSD-currentを利用しています．一部，Linuxカーネルについて言及している部分もありますが，これは本稿執筆時点での最新バージョンであるLinux-2.6.8.1を参照し，おもにLinux/PowerPCを題材にしています．

1.1　リンカとオブジェクト

組み込みシステムやOSを作るうえでは，リンカはメモリ配置と密接な関係がある部分になるので，リンカの知識が必須になります．しかし，リンカの動作や詳細について細かく説明してある資料は，書籍，雑誌記事，インターネットとも，あまりないように感じます．リンカの知識がないと，OSのカーネルがメモリ上にどのようにマッピングされるか，メモリがどのように利用されるかといったイメージをつかみにくくなります．このため，OSのカーネル・ソースを読んだり，OSの移植に挑戦する際に，リンクの知識の有無が，壁の一つとなることが多々あるようです．

リンクについては，書籍では，次のように説明されていることが多いように思います．

「複数のオブジェクトどうしを結合することである」

しかし，初心者のうちは単一ファイルのプログラム〔場合によっては`main()`しかないような〕しか作成したことがなかったりするので，このような説明では実感が湧かないことが多いようです(筆者自身がそうだった)．

そこで第1章では，リンカが何を行っているのか，プログラムがコンパイルされて実行形式となり，実際に動作するためには，何が必要なのかを説明します．

1.2　リンカの仕事

● リンカの必要性

何十万行もあるような，規模の大きなアプリケーションを作成することを想像してください．このようなとき，ソース・コードを単一のファイルにすべて押し込むようなことはまずありません．必ず複数のファイルに分割します．そうしないと，複数の人間が役割を分担して同時に開発することが，現実的に不可能になってしまうからです．

また，大規模なアプリケーションの場合には，すべてをコンパイルするまでに数時間かかってしまうことも珍しくありません．ソース・コードを複数のファイルに分割し，ファイル単位でコンパイルを行うことで，修正時には必要なファイルのみコンパイルを行えばよいようにすることができます．

● ファイルを分割した時

ソース・コードを複数のファイルに分割した際，通常のコンパイラは，ファイル単位でコンパイルをすることができます[注1.1]．このときには「オブジェクト・ファイル」と呼ばれる一次ファイルが作成されます．

オブジェクト・ファイルの拡張子は，通常は.oになります．オブジェクト・ファイルは，C言語のソース・コードを，ファイル単位で機械語に変換したものです．つまり，コンパイラは.cファイルに1対1に対応した.oファイルを作成することになります．

ファイルを分割すると，「あるファイルに実体が定義してある関数を，別のファイルから呼び出したい」という必要性が出てきます．つまり，ファイル間をまたいだ関数呼び出しが発生するのです．また，変数についても同様に，ファイルをまたいだ参照や書き込みが発生します．このような場合，呼び出し元のファイルを単体でコンパイルするときには，呼び出し先の関数の実体が見当たらないので，完全な機械語コードを作成することができません．このためオブジェクト・ファイルでは，関数呼び出しの部分に，「ここでは別ファイルの×××という関数を呼び出す」というマークが残されます．最終的に複数のオブジェクト・ファイルを結合したときに，マークの部分は実際の関数呼び出しに置き換えられます．この結合作業が「リンク」です．リンクすることで，最終的に実行可能なファイルができあがります．この実行可能なファイルのことを，「実行形式」と呼びます(**図1.1**)．なお，分割コンパイルの具体的な方法に関しては本書の範囲を越えてしまうので，稿末の参考文献(1)を参照してください．

● 単一のファイルでもリンカは必要

では，単一のファイルからなるプログラムだったとしたら，リンクは必要ないのでしょうか？　たとえばhello.c(**リスト1.1**)を見てください．これはいわゆる“Hello world”ですが，このようなmain()関数しかないプログラムの場合はどうでしょうか．

答は「それでもリンクは必要」です．hello.cでは，ライブラリ関数としてprintf()を利用してい

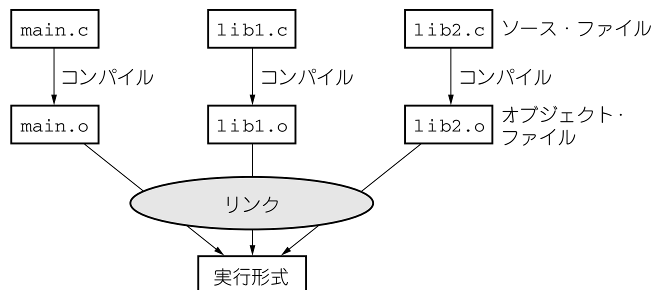


図1.1　コンパイルとリンク

リスト1.1　hello.c

```
#include <stdio.h>

int main()
{
  printf("Hello world!¥n");
  exit (0);
}
```

注1.1：gccでは-cオプションをつけることで，ファイル単位にコンパイルを行うことができる．この場合は，単体のファイル中には，main()関数は必要ない．

ます．また，printf()だけでなくexit()も実はライブラリ関数なのです．単一のファイルからなるプログラムで，main()関数しかないようなプログラムだったとしても，内部ではライブラリ関数を利用しているかもしれません．このような場合には，ライブラリのリンクが必要になります．

● ライブラリとのリンク

また，一般的なプログラムであればOSの機能を呼び出すためにシステム・コールを利用します．これらシステム・コールの呼び出しはアセンブラで書かれるのが普通です．アセンブリ言語で書かれたソース・ファイルを処理するのはアセンブラですが，C言語で書かれたソース・ファイルを処理するのはCコンパイラです．このため，アセンブリ言語で書かれたシステム・コール呼び出しをC言語側から利用するためには，アセンブリ言語のソースとC言語のソースを別々のオブジェクト・ファイルにして，最終的にリンクするような作業が必要です．このように，複数の種類の言語が混在する場合にも，リンクという作業が必要になります．

さらに，たとえライブラリ関数が利用されていないとしても，実行形式を作成する際には，スタートアップ・ルーチンという「初期化」を行うプログラムがリンクされます．

スタートアップ・ルーチンでは，レジスタの初期化やmain()への引き数（いわゆるargcとargv）の設定，そのほか各種の初期化が行われます．実はプログラムを実行したときに，いちばん最初に実行されるのはmain()ではありません．実際には，スタートアップ・ルーチンがいちばん最初に実行され，そこからmain()が呼び出されます．さらに言うならば，exit()をするとプログラムは即終了するわけではありません．exit()の後にいくつかの終了処理が行われた後，_exit()というシステム・コールによって，プロセスが終了します．

また，関数や変数の実際のアドレスへの配置も，リンクのときに行われます．このため，C言語（もしくはそれ以外のコンパイラ型言語）のプログラムを実行形式に変換するには，リンクという工程が絶対に必要なのです．

通常，コンパイラが行う作業は，C言語（もしくはそれ以外のコンパイラ型言語）のソース・コードを機械語に変換し，オブジェクト・ファイルを作成するだけです．リンカはこれらの機械語のコード（オブジェクト・ファイル）を結合し，実際にOSがロードして実行できるような実行形式に変換します．最終リンクの際には，ユーザが用意したオブジェクト・ファイルだけでなく，OSがシステムとして用意しているライブラリなどもリンクされます．つまりリンクとは，「コンパイラが出力した機械語のコードを，OSに依存する実行形式に変換する段階」であるということができます．

● リンカの動作例

このように，プログラムを実行形式に変換するためには，リンクという作業が必要です．では，先ほどのhello.cのようなプログラムをgccでコンパイルして実行形式を作成する際には，いつリンクが行われているのでしょうか．gccを-vオプション付きで実行すると，実行形式が作成されるまでの詳細が出力されます．**リスト1.2**は，hello.cをgcc -vでコンパイルしたときの出力結果です．

リスト1.2を見ると，18行目で，/usr/libexec/elf/ldというリンカが呼ばれていることがわか

リスト1.2　リンクまでの流れ――hello.cをgcc -vでコンパイルしたときの出力結果

```
001: % gcc hello.c -Wall -o hello -v
002: Using builtin specs.                                    (プリプロセッサ)
003: gcc version 2.95.4 20020320 [FreeBSD]
004:  /usr/libexec/cpp0 -lang-c -v -D__GNUC__=2 -D__GNUC_MINOR__=95 -Di386 -D__FreeBSD__
        =4 -D__FreeBSD_cc_version=460001 -Dunix -D__i386__ -D__FreeBSD__=4 -D__FreeBSD_cc_version
        =460001 -D__unix__ -D__i386 -D__unix -Acpu(i386) -Amachine(i386) -Asystem(unix)
        -Asystem(FreeBSD) -Wall -Acpu(i386) -Amachine(i386) -Di386 -D__i386 -D__i386__ -D__ELF__
         hello.c /tmp/ccqnUJxp.i
005: GNU CPP version 2.95.4 20020320 [FreeBSD] (i386 FreeBSD/ELF)
006: #include "..." search starts here:
007: #include <...> search starts here:
008:  /usr/include
009:  /usr/include
010: End of search list.
011: The following default directories have been omitted from the search path:
012:  /usr/include/g++                             (cc1が呼ばれている)
013: End of omitted list.
014:  /usr/libexec/cc1 /tmp/ccqnUJxp.i -quiet -dumpbase hello.c -Wall -version -o /tmp/ccbkxy21.s
015: GNU C version 2.95.4 20020320 [FreeBSD] (i386-unknown-freebsd) compiled by GNU C version
                               (アセンブラ)                                 2.95.4 20020320 [FreeBSD].
016:  /usr/libexec/elf/as -v -o /tmp/ccs4WTKL.o /tmp/ccbkxy21.s
017: GNU assembler version 2.12.1 [FreeBSD] 2002-07-20 (i386-obrien-freebsd5.0) using BFD version
                               (リンカ)                                     2.12.1 [FreeBSD] 2002-07-20
018:  /usr/libexec/elf/ld -V -dynamic-linker /usr/libexec/ld-elf.so.1 -o hello /usr/lib/crt1.o
                        /usr/lib/crti.o /usr/lib/crtbegin.o
                        -L/usr/lib /tmp/ccs4WTKL.o -lgcc -lc -lgcc /usr/lib/crtend.o /usr/lib/crtn.o
019: GNU ld version 2.12.1 [FreeBSD] 2002-07-20
020:   Supported emulations:
021:    elf_i386
022: %
```

ります．引き数には，crt1.o，crti.o，crtbegin.o，crtend.o，crtn.oの五つのオブジェクト・ファイルと，/tmp以下にある/tmp/ccs4WKTL.oというオブジェクト・ファイルが指定されています．前者の五つのファイルは，スタートアップ・ルーチンです(crtはC RunTime start upの略)．後者の/tmp/ccsWKTL.oは，hello.cから作成されたオブジェクト・ファイルです．また，4行目では/usr/libexec/cpp0(プリプロセッサ)が呼ばれ，16行目では/usr/libexec/elf/as(アセンブラ)が呼ばれています．

このように，リンクまでにはいくつかの作業が順番に行われます．/usr/libexec/elf/ldでのリンク時に/tmp/ccsWKTL.oという何やらよくわからない名前のオブジェクト・ファイルが指定されているのは，/tmpをテンポラリ・ディレクトリとして，前工程の出力(**リスト1.2**の場合には，16行目の/usr/libexec/elf/asの出力)を受け渡しているためです．

リスト1.2の14行目に注目してください．ここで/usr/libexec/cc1というコマンドが呼ばれています．gccは，指定されたファイルを読み込み，ファイルの形式に応じて，プリプロセス，コンパイル，アセンブル，リンクまでの作業を順番に行ってくれます．リンクも自動で行ってくれるため，プログラマはリンカを利用していることを意識せず，実行形式を作成することができます．つまりgccは，プリプロセスからリンクまでの，実行形式作成用のマネージャとでもいうべきものであり，正しい意味での「コンパイラ」はcc1である，ということになります．このため，gccは「コンパイラ・ドライバ」と呼ばれることもあります(**図1.2**)．

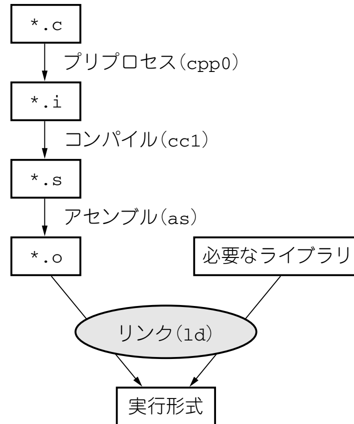


図1.2　gccが実際に行う作業

コンパイルというのは，正確には，ソース・コードを機械語に変換する作業のことなので，リンクとはまったく別の作業です．しかし通常は上記のgccの動作のように，リンクまでの一連の作業を総称して「コンパイル」と呼ぶことが多くあるようです．つまり，実行形式が作成されるまでを「コンパイル」と呼んでいるわけです．その意味では，cc1は狭義のコンパイラで，gccは広義のコンパイラだということができるでしょうか．

本稿では以後，これをはっきりと区別して，「コンパイル」は狭義の意味で使い，広義の「コンパイル」は「コンパイル＆リンク」と呼びます[注1.2]．

関数や変数の実体は，特定のアドレス上に置かれます．プログラム中の関数呼び出しや変数の参照は，実際には，特定アドレスへのジャンプや，特定アドレスのメモリの参照になります．しかし，オブジェクト・ファイルの段階では，すべてのオブジェクト・ファイルが出そろった段階でないとアドレスを決定できないため，これらのアドレスは決定されていません．リンクは，これらの関数や変数を実際に特定アドレスに割り当てて，それらを利用している部分に，割り当てられたアドレスを挿入する作業であるともいえます．

1.3　実行形式とセクション

● 実行形式のフォーマットについて

我々がアプリケーション・プログラムを作成し，gccによりコンパイル＆リンクを行うと，実行形式が作成されます．この「実行形式」は，CPUが実行する機械語コードを「ベタに」ファイルにしたもの，というわけではありません．先頭部分にヘッダ情報を持ち，ある特定のフォーマットになっています．

このフォーマットには，古くはa.out(Assembler OUTput)形式が利用されていましたが，現在ではELF(Executable and Linking Format)形式が多く利用されています．また一部では，COFF(Common

注1.2：「コンパイル＆リンク」のことを「ビルド」と呼ぶこともある．

Object File Format)という形式も利用されています．具体的なフォーマットに関しては，参考文献(2)を参照してください．また，これらのフォーマットを総称して，「オブジェクト・フォーマット」，「オブジェクト・ファイル・フォーマット」などと呼びます．呼び方は「オブジェクト・フォーマット」ですが，オブジェクト・ファイルに限らず，実行形式やダイナミック・リンク・ライブラリなども，このフォーマットで表されます．通常は*.oファイルのことを「オブジェクト・ファイル」と呼びますが，広義では*.oファイルだけに限らず，このように実行形式などのことも含めて「オブジェクト」と呼ぶ場合もあります．

● ファイル内の複数の領域

このように，実行形式にはヘッダ情報が添付されています．さらに，ファイルの内部はその目的ごとに，複数の領域に分けられています．これらの各領域のサイズは，sizeコマンドで確認することができます．たとえば，**リスト1.1**のhello.cをコンパイル＆リンクして実行形式helloを作成し，helloに対してsizeを実行すると，**リスト1.3**のようになります．

リスト1.3を見ると，helloという実行形式が，text，data，bssという三つの領域からできていることがわかります．これらはそれぞれ，テキスト領域，データ領域，BSS領域(BSS：Block Started by Symbol)と呼ばれます．各領域名の下に表示されている数値は，各領域のバイト・サイズです．なお，decは三つの領域の合計サイズ，hexは合計サイズを16進表記にしたものです．

テキスト領域には，機械語の実行コードが置かれます．また，変更されることのないデータ(const定義されている変数や，文字列リテラルの本体など)も，通常はここに置かれます．メモリ保護のあるOSの場合には，テキスト領域はread onlyに設定することで，コードやデータの不正な変更を防止します．

データ領域には，初期値のある変数の本体が置かれます．書き込みが可能な変数は，この領域に置かれます．ただしauto変数(ローカル変数で，スタック上に置かれるもの)は，対象外です．つまりデータ領域に置かれるのは，以下の変数のうち，初期値を定義しているものです．

- 外部に公開している変数(グローバル変数)
- 関数の外部で定義しているstatic変数(ファイル内で広域だが，ファイル外には公開しない変数)
- 関数の内部で定義しているstatic変数(関数にローカルだが，値が保存される変数)

BSS領域には，初期値が未定義の変数が配置されます．こちらもauto変数は，対象外になります．つまり，上記の変数のうち，初期値を定義していないものがBSS領域に置かれます．

初期値が未定義の場合には，実行形式中にデータとして値をもつ必要はありません．したがってBSS領域は，実行形式中では，サイズの情報だけで，実体はありません．プログラムの実行のためにOSが実

リスト1.3　size helloの結果

```
% gcc hello.c -o hello -Wall
% ./hello
Hello world!
% size hello
   text    data     bss     dec     hex filename
   1042     208      28    1278     4fe hello
%
          バイト・サイズ
```

行形式をロードしたときに，メモリ上に作成されます．

アプリケーション・プログラムの場合には，実行形式をメモリ上に展開して実行を開始するのは，OSの仕事です．プログラムの実行を行うには，`exec()`などのライブラリ関数〔実体は`execve()`システム・コール〕を利用します．このときOSは，機械語コードをメモリ上のどこに展開するか，どこから実行するか，といったことを知る必要があります．ヘッダにこれらの情報をもたせることで，OSはどのように展開・実行すればよいかという情報を得ることができます．

OSはプログラムの実行時には，実行形式のヘッダを参照し，そこに格納された情報に従って，それぞれの領域をメモリ上に展開し，実行を開始します[注1.3]．このように，実行形式をロードしてメモリ上に展開し，実際に実行を開始するためのプログラムを「ローダ」と呼びます．

a.out形式では，領域はテキスト，データ，BSSの３種類しかとることができませんでした．また，a.out形式では，領域にいろいろな属性を持たせることができません．これでは柔軟性に欠けるため，ELF形式では，任意数の「セクション」を確保して，さまざまな属性をもたせることができるようになっています．

`objdump`コマンドを利用すると，セクション情報が得られます．**リスト1.4**は，実行形式`hello`に対して`objdump`を実行して，セクション情報を表示させたときの結果です．

リスト1.4では，番号が0～19の20個の区画が表示されています．これらの区画のことを「セクション」と呼びます．また，これとは別に，「セグメント」ということばもあります．これらの「領域」，「セグメント」，「セクション」ということばの使い分けは，オブジェクト・フォーマットによって異なるので注意が必要です．上記の`objdump -h`で得られる区画情報は，実はELF形式でいうところの「セクション」に相当するので，本稿ではセクションと呼んでいます．また前述したテキスト，データ，BSSの三つに関しては，「領域」と呼ぶことにします．

● VMAとLMAの対応

リスト1.4では，VMAとLMAという２種類のアドレスが表示されています．VMAはVirtual Memory Addressの略で，セクションをリンクするときにベースとなるアドレスです．関数や変数のアドレスは，VMAを基準として配置されます．また，LMAはLoad Memory Addressの略で，セクションを展開する先のアドレスのことです．

通常のアプリケーションではVMA＝LMAとなりますが，例外としてVMA≠LMAとなることもあります．代表的なのはOSのカーネルです．OSのカーネルでは，最初は物理アドレスで動作して，後に自分自身を仮想アドレスにマップしなおして，途中から論理アドレスで動作するという場合があります．このような場合には，カーネルの展開先は物理アドレスになりますが，カーネル中の関数や変数は，実際にOSが動作する論理アドレスをベースにしてリンクすることになりますから，VMA≠LMAとなります[注1.4]．

注1.3：仮想メモリをもつOSでは，実際には仮想メモリ機構のデマンド・ローディングにより，要求された部分のみがそのつどメモリ上に展開されるのだが，ここでは本質ではないので深くは言及しない．

注1.4：このため，仮想アドレスのマップ前に関数呼び出しやグローバル変数の操作をする場合には，論理アドレスを物理アドレスに計算しなおしてアクセスするか，相対アドレスでアクセスする必要がある．

リスト1.4 objdump -h helloの結果

```
% objdump -h hello

hello:     file format elf32-i386

Sections:                       VMA        LMA
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000019  080480f4  080480f4  000000f4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000018  08048110  08048110  00000110  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         00000054  08048128  08048128  00000128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       00000100  0804817c  0804817c  0000017c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       0000009d  0804827c  0804827c  0000027c  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .rel.plt      00000018  0804831c  0804831c  0000031c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .init         0000000b  08048334  08048334  00000334  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  7 .plt          00000040  08048340  08048340  00000340  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  8 .text         00000178  08048380  08048380  00000380  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  9 .fini         00000006  080484f8  080484f8  000004f8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 10 .rodata       0000000f  080484fe  080484fe  000004fe  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 11 .data         0000000c  08049510  08049510  00000510  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 12 .eh_frame     00000004  0804951c  0804951c  0000051c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 13 .dynamic      00000098  08049520  08049520  00000520  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 14 .ctors        00000008  080495b8  080495b8  000005b8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 15 .dtors        00000008  080495c0  080495c0  000005c0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 16 .got          00000018  080495c8  080495c8  000005c8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 17 .bss          0000001c  080495e0  080495e0  000005e0  2**2
                  ALLOC
 18 .comment      000000a0  00000000  00000000  000005e0  2**0
                  CONTENTS, READONLY
 19 .note         00000050  00000000  00000000  00000680  2**0
                  CONTENTS, READONLY
%
```

● セクションの実例

リスト1.4を見ると，helloという実行形式は，実際には20個のセクションからできていることがわかります．しかし，**リスト1.3**で表示されたのはtext，data，bssの三つの領域だけでした．これらの結果の違いは何でしょうか．それは，sizeコマンドのソースを見るとわかります．

which sizeによると，sizeの本体は，/usr/bin/sizeとなっているので，セオリどおりにソースを追いかけるならば，そのソースは/usr/src/usr.bin以下にあることになります．そこで，/usr/src/usr.bin/sizeを見てみましょう．ここにはsize.cというファイルがあり，size.cを見ると，**リスト1.5**のようなことを行っている部分があります．どうやら，ヘッダ部分の解析を行ってい

リスト1.5　/usr/src/usr.bin/size/size.cより抜粋

```
001: int
002: show(count, name)
003:         int count;
004:         char *name;
005: {
006:         static int first = 1;
007:         struct exec head;
008:         u_long total;
009:         int fd;
010:
011:         if ((fd = open(name, O_RDONLY, 0)) < 0) {
012:                 warn("%s", name);
013:                 return (1);
014:         }
015:         if (read(fd, &head, sizeof(head)) !=sizeof(head) || N_BADMAG(head)) {
016:                 (void)close(fd);
017:                 warnx("%s: not in a.out format", name);
018:                 return (1);
019:         }
020:         (void)close(fd);
021:
022:         if (first) {
023:                 first = 0;
024:                 (void)printf("text¥tdata¥tbss¥tdec¥thex¥n");
025:         }
026:         total = head.a_text + head.a_data + head.a_bss;
027:         (void)printf("%lu¥t%lu¥t%lu¥t%lu¥t%lx", (u_long)head.a_text,
028:             (u_long)head.a_data, (u_long)head.a_bss, total, total);
029:         if (count > 1)
030:                 (void)printf("¥t%s", name);
031:         (void)printf("¥n");
032:         return (0);
033: }
```

るようです．

リスト1.5では，11行目でファイルをオープンして，15行目でファイルの先頭をstruct execという構造体に読み込んでいます．struct execは，/usr/include/sys/imgact_aout.hで**リスト1.6**のように定義されています．

つまり，ファイルの先頭16バイトには，マジック・ナンバ，テキスト領域のサイズ，データ領域のサイズ，BSS領域のサイズが4バイトずつ利用して格納されており，sizeコマンドはヘッダに格納されている情報を読み，表示しているだけだということになります．

本当にそうなのでしょうか．実行形式をダンプして，実際に見てみましょう．**リスト1.7**は，実行形式helloの先頭部分の16進ダンプです．

ここでおかしなことに気がつきます．**リスト1.7**を見て，先頭16バイトをstruct execに当てはめて解釈すると，テキスト領域のサイズは0x09010101バイト（リトル・エンディアンであることに注意）というとんでもなく大きな値になり，データ領域とBSS領域のサイズは，0バイトになってしまっています．これはあきらかに変なので，実行形式の先頭16バイト部分には，先に説明したような値は入っていないように見えます．さらに，**リスト1.7**のテキスト表示の部分（右端）には，何やら“ELF”という文字列が現れています．

リスト1.6 **struct execの定義**(imgact_aout.hより抜粋)

```
/*
 * Header prepended to each a.out file.
 * only manipulate the a_midmag field via the
 * N_SETMAGIC/N_GET{MAGIC,MID,FLAG} macros in a.out.h
 */

struct exec {
     unsigned long a_midmag; /* flags<<26 | mid<<16 | magic */
     unsigned long a_text;   /* text segment size */
     unsigned long a_data;   /* initialized data size */
     unsigned long a_bss;    /* uninitialized data size */
     unsigned long a_syms;   /* symbol table size */
     unsigned long a_entry;  /* entry point */
     unsigned long a_trsize; /* text relocation size */
     unsigned long a_drsize; /* data relocation size */
};
/* XXX Hack to work with current kern_execve.c */
#define a_magic a_midmag
```

リスト1.7 **helloの先頭部分の16進ダンプ**

```
% hexdump -C hello | head -n 3
00000000  7f 45 4c 46 01 01 01 09  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  80 83 04 08 34 00 00 00  |............4...|
00000020  7c 07 00 00 00 00 00 00  34 00 20 00 06 00 28 00  ||.......4. ...(.|
%
```

では，このsizeのソースは何者なのでしょうか．ソースがどのように利用されるのかを知りたいときには，Makefileを見るのが一番です．/usr/src/usr.bin/size/Makefileを見てみると，

```
BINDIR= /usr/libexec/aout
```

となっています．これから，このsizeコマンドは，/usr/libexec/aoutにインストールされるということがわかります．実はソース・コードをきちんと読むとわかるのですが，/usr/src/usr.bin/sizeは，a.out形式用の古いsizeコマンドのソース・コードなのです．struct execは，a.out形式のヘッダの構造体です．つまり，/usr/src/usr.bin/sizeにあるsizeコマンドは，もともとa.out形式の実行ファイルのテキスト，データ，BSS領域のサイズを出力するためのものだったことがわかります．また，struct execを見ればわかるように，a.out形式では，セクションはテキスト，データ，BSSの三つしかないという前提で，決めうちになっています．これが，a.out形式では，3種類の領域しかとることができない理由です．

FreeBSDの実行形式のフォーマットは，以前はa.out形式でしたが，現在ではアプリケーション・プログラム，カーネルともに，デフォルトでELF形式です．このため，**リスト1.3**でsizeコマンドを利用した際には，ELF形式用のsizeコマンドが実行されたはずです．では，ELF形式用のsizeコマンドはどこにあるのでしょうか．

FreeBSDでは，sizeコマンドは現在ではGNU binutilsの一部として配布されているものを利用しているため，ソース・コードは/usr/src/gnu/usr.bin/binutils/size，/usr/src/contrib/binutils/binutils/size.cにあります．また，その本体は，/usr/libexec/elfにあります．

リスト1.8は，/usr/src/contrib/binutils/binutils/size.cからの抜粋です．ELFの実行形式の，それぞれのセクションのサイズをカウントしている部分です．

リスト1.8のberkeley_sum()では，次のようにしてカウントしています．

1）ALLOCフラグが立っていないセクションは無視する
2）ALLOCフラグが立っていて，CODEフラグかREADONLYフラグが立っているものは，テキスト領域としてカウントする
3）テキスト領域ではないが，CONTENTSフラグが立っているものは，データ領域としてカウントする
4）それ以外はBSS領域としてカウントする

ここでもう一度，**リスト1.4**を見てください．20個のセクションには，それぞれにCONTENTS，ALLOC，LOADなどのフラグが設定されています．0～10番のセクションは，CODEフラグかREADONLYフラグが立っているため，テキスト領域としてカウントされます．11～16番のセクションは，CODEフラグもREADONLYフラグも立っていないが，CONTENTSフラグは立っているため，データ領域としてカウントされます．17番目のセクションは，ALLOCフラグだけのため，BSS領域となります．18，19番目のセクションは，ALLOCフラグが立っていないため，無視されます．これらのカウントの合計が，sizeコマンドの出力となります（**リスト1.3**のsizeコマンドの出力では10進表示だが，**リスト1.4**のsizeの項は16進表示になっていることに注意）．

つまり，ELF用のsizeコマンドの場合には，領域のサイズは，複数のセクションの「集計値」になり

リスト1.8　セクションのサイズをカウントしている部分
（/usr/src/contrib/binutils/binutils/size.cより抜粋）

```
static bfd_size_type bsssize;
static bfd_size_type datasize;
static bfd_size_type textsize;

static void
berkeley_sum (abfd, sec, ignore)
     bfd *abfd ATTRIBUTE_UNUSED;
     sec_ptr sec;
     PTR ignore ATTRIBUTE_UNUSED;
{
  flagword flags;
  bfd_size_type size;

  flags = bfd_get_section_flags (abfd, sec);
  if ((flags & SEC_ALLOC) == 0)
    return;

  size = bfd_get_section_size_before_reloc (sec);
  if ((flags & SEC_CODE) != 0 || (flags & SEC_READONLY) != 0)
    textsize += size;
  else if ((flags & SEC_HAS_CONTENTS) != 0)
    datasize += size;
  else
    bsssize += size;
}
```

ます．FreeBSDが扱うオブジェクト・フォーマットは，以前はa.out形式であったため，sizeコマンドはもともとはa.out形式を期待して，テキスト，データ，BSSの三つの領域を表示していました．しかしその後，オブジェクト・フォーマットはELF形式に変更されました．ELF形式では任意の数のセクションを持たせることができるため，フラグ情報を見てテキスト，データ，BSSのいずれかに分類するようなしくみになっていると思われます．

なお，ELF形式では，複数の「セクション」をまとめたものを「セグメント」として定義することができます(2)．セグメントは，ELF形式のファイル中に「プログラム・ヘッダ」として記述されています．実行形式のロード時には，プログラム・ヘッダを見て，セグメント単位で展開することで，似たような属性のセクションをまとめて展開することができます．

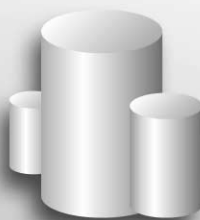
プログラム・ヘッダの情報は，objdump -pで知ることができます(**リスト1.9**)．またobjdumpは，--all-headersですべてのヘッダ情報を表示します．

なおobjdumpコマンドは，実行形式，オブジェクト・ファイル，ダイナミック・リンク・ライブラリ，コア・ダンプ，ライブラリ・アーカイブに対して実行可能です．

リスト1.9　objdump -p helloの結果

```
% objdump -p hello

hello:     file format elf32-i386

Program Header:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x000000c0 memsz 0x000000c0 flags r-x
  INTERP off    0x000000f4 vaddr 0x080480f4 paddr 0x080480f4 align 2**0
         filesz 0x00000019 memsz 0x00000019 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x0000050d memsz 0x0000050d flags r-x
    LOAD off    0x00000510 vaddr 0x08049510 paddr 0x08049510 align 2**12
         filesz 0x000000d0 memsz 0x000000ec flags rw-
 DYNAMIC off    0x00000520 vaddr 0x08049520 paddr 0x08049520 align 2**2
         filesz 0x00000098 memsz 0x00000098 flags rw-
    NOTE off    0x00000110 vaddr 0x08048110 paddr 0x08048110 align 2**2
         filesz 0x00000018 memsz 0x00000018 flags r--

Dynamic Section:
  NEEDED      libc.so.4
  INIT        0x8048334
  FINI        0x80484f8
  HASH        0x8048128
  STRTAB      0x804827c
  SYMTAB      0x804817c
  STRSZ       0x9d
  SYMENT      0x10
  DEBUG       0x0
  PLTGOT      0x80495c8
  PLTRELSZ    0x18
  PLTREL      0x11
  JMPREL      0x804831c

%
```

第2章 ELF形式の解析



BSDやLinuxなどのPC-UNIXの世界では，オブジェクト・フォーマットにELF形式が多く採用されています．ELF形式はそれなりに複雑ですが，ダイナミック・リンク・ライブラリやC++への対応など，さまざまな要求に対応できる優れたフォーマットです．このため広く利用されているのですが，一般のプログラマにとって，ELF形式の内部構造というのは，それほどなじみのあるものではありません．

本章は，ELF形式の構造について説明します．

2.1 オブジェクト・フォーマット

● オブジェクト・フォーマットとリンカとローダの関係

ELF形式やCOFF形式などは「オブジェクト・フォーマット」と呼ばれます．リンカやローダの動作とオブジェクト・フォーマットの間には，密接な関係があります．

リンカは複数のオブジェクト・ファイルをリンクして，一つの実行形式を作成します．そのためには「オブジェクト・ファイルのフォーマットが読めて，実行形式のフォーマットが書ける」必要があります．このため，リンカはオブジェクト・フォーマットを読み書きできる必要があります．

また，プログラムの実行時は，ローダによるロードが不可欠ですが，そのためには実行形式のフォーマットが読み込める必要があるため，ローダもオブジェクト・フォーマットを読めなければなりません．

● さまざまなオブジェクト・フォーマットとELF形式の利点

オブジェクト・フォーマットには，古くはa.out形式があり，ELF形式やCOFF形式など，いくつかの

種類があります．組み込み分野ではCOFF形式が多く利用されていますが，現在のPC-UNIXの世界では，ELF形式が主流です．その理由の一つとして，ELF形式はダイナミック・リンク・ライブラリやC++への対応，クロス・プラットホームや64ビット対応など，さまざまなことが考慮されており，さらに任意のセクションが作成できるため，融通が効くという利点があります．つまりELF形式を採用しておけば，ほとんどのことが問題なく実現できるため，ほかのフォーマットを採用したり，新しいフォーマットを考案する必要性がないわけです．

そこで，ここではオブジェクト・フォーマットの代表としてELF形式について説明します．

2.2　ELF形式の構造

● ELF32/ELF64

ELF形式には32ビットのものと64ビットのものが定義されており，それぞれELF32，ELF64と呼ばれます．

ELF形式については，FreeBSDでは`man elf`で詳しい説明を読むことができます．マニュアルは日本語にも訳されているので，`jman`を利用すれば，日本語版を読むこともできます．

またFreeBSDでは，ヘッダ・ファイルとして`/usr/include/elf.h`が用意されています．`elf.h`の内部では，直接または間接的に**表2.1**のヘッダ・ファイルがインクルードされます．これらのヘッダ・ファイルでは，ELF形式で利用される構造体などが定義されているため，ELF形式のフォーマットを理解するうえで参考になります．

● ELF形式の構造――セクション・ヘッダとプログラム・ヘッダ

ELF形式のファイルは，**図2.1**のような構造になっています．ELF形式の最大の特徴は，セクション・ヘッダとプログラム・ヘッダによる，2重構造になっているということです．

ELF形式の先頭には，ELFヘッダが付加されています．ELFヘッダは，ELF32/ELF64の区別やエンディアン，アーキテクチャなどの情報をもちます．さらにELF形式の内部は，複数のセグメントに区切られており，セグメントの内部が，複数のセクションに区切られています[注2.1]．**図2.1**では，セクションを破線で，セグメントを細線で表しています．

セグメントはプログラム・ヘッダによって記述されます．ELF形式は，ELFヘッダの直後に，プログラム・ヘッダの配列としてプログラム・ヘッダ・テーブルを持っています．それぞれのプログラム・ヘッダが，各セグメントの情報を保持しています．また，セクションはセクション・ヘッダによって記述されます．ELF形式の末尾には，セクション・ヘッダの配列としてセクション・ヘッダ・テーブルがあり，セクション・ヘッダのそれぞれが，各セクションの情報を保持しています．**図2.1**では，プログラム・

注2.1：実際にはセグメントとセクションは独立して存在するが，ここでは便宜上，セグメントがセクションを含むものとして説明している．実は，セクションの途中でセグメントを分けたり，部分的にセグメントに含まれない箇所を作ったり，セグメントを重ねることも可能である．

表2.1 FreeBSDでのELF関連のヘッダ・ファイル

ヘッダ・ファイル	内　容
/usr/include/machine/elf.h	CPU依存の定義
/usr/include/sys/elf32.h	ELF32依存の定義
/usr/include/sys/elf64.h	ELF64依存の定義
/usr/include/sys/elf_common.h	ELF32，ELF64共通の定義
/usr/include/sys/elf_generic.h	ELF32，ELF64に非依存の定義(後述)

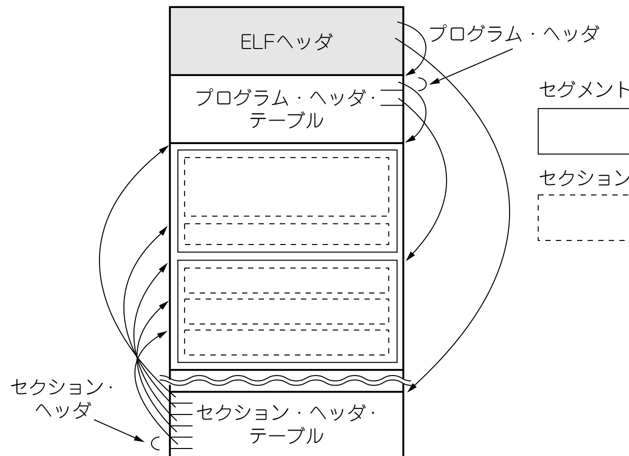


図2.1 ELF形式の構造

ヘッダとセグメント，セクション・ヘッダとセクションの対応関係を，矢印で表現しています．

なお，セグメントは，あってもなくてもかまいません．セグメントがない場合には，プログラム・ヘッダは付加されず，ELF形式の内部はセクションのみで構成されることになります．この場合，プログラム・ヘッダが付加されないため，「セグメントをまったく持たない」という構造になります．厳密にいうならば，「セクションはセグメントにかならずしも属する必要はない」ということになります．つまり，「あるセクションはあるセグメントに属しているが，別のセクションはどのセグメントにも属さない(全体を一つのセグメントとするのではない)」といった構造もとることができます．

リンクとロードは本来異なる作業です．このためELF形式では，リンクの単位としてセクションを持ち，ロードの単位としてセグメントを持っています．つまり，セクションはリンカのために存在するものであり，リンクはセクション単位で行われます．セグメントとロードに関しても，同様のことがいえます．

よってELF形式では，ロードが不可能なファイルは，プログラム・ヘッダを持ちません．

このため実行可能ファイルとダイナミック・リンク・ライブラリはプログラム・ヘッダを持ちますが，オブジェクト・ファイルはプログラム・ヘッダを持っていません．プログラム・ヘッダは，オブジェクト・ファイルのリンク時に，リンカによって付加されます．逆にセグメントは持つがセクションは持たないという構成も可能です．

2.3　使用するサンプル・プログラム

ここでは，ELF形式を説明するためのサンプル・プログラムとして，elfsamp.c(**リスト2.1**)を作成しました．また，elfsamp.cをコンパイル/リンクして実行形式を作成するためのmain()関数として，main.c(**リスト2.2**)を考えます．elfsamp.cでは，いくつかの変数と関数を定義しているだけです．

elfsamp.cとmain.cをコンパイルして，それぞれelfsamp.oとmain.oを作成します．さらにリンクして，実行形式elfsampを作成します．

```
% gcc -c elfsamp.c -Wall
% gcc -c main.c -Wall
% gcc elfsamp.o main.o -Wall -o elfsamp
```

elfsampを実行してみます．

```
% ./elfsamp
sample
%
```

無事に実行できています．

以後は，オブジェクト・ファイルのサンプルには，上で作成したelfsamp.oを利用します．また，実行形式のサンプルには，同じく上で作成したelfsampを利用します．

リスト2.1　ELFファイル解析用サンプル(elfsamp.c)

```
#include <stdio.h>

static int s1, s2 = 1;
int g1, g2 = 1;
const int g3 = 1;
extern int e3;
void efunc();

static void sfunc()
{
  efunc();
  s1 = s2 = 10;
  g1 = g2 = 10;
  e3 = 10;
}

void gfunc()
{
  sfunc();
  efunc();
  g1 = e3 = 20;
  fprintf(stdout, "sample¥n");
}
```

リスト2.2　ELFファイル解析用サンプル(main.c)

```
#include <stdio.h>

int e3;
void gfunc();
void efunc() {}

int main()
{
  gfunc();
  exit (0);
}
```

2.4 ELFヘッダ

● さまざまなELFヘッダ

ELF形式のファイルの先頭には，ELFヘッダが付加されています．ELFヘッダの内容は，FreeBSDではelf32.hで，構造体Elf32_Ehdrとして定義されています(**リスト2.3**)．

なお，ELFヘッダのフォーマットは，ELF32とELF64で異なります．elf32.hでElf32_Ehdrが定義されていることに対して，ELF64用には，elf64.hでElf64_Ehdrが定義されています．さらにi386用のFreeBSDの場合には，elf_genelic.hで，Elf32_EhdrがElf_Ehdrに再定義されています(64ビット・アーキテクチャ用のFreeBSDの場合には，Elf64_EhdrがElf_Ehdrに再定義される)．ELF形式を扱うプログラム中では，Elf32_EhdrやElf64_Ehdrといったビット依存な型ではなく，Elf_Ehdrを利用することで，異なるビット間での移植性のあるプログラムを書くことができます．elf_genelic.h中では，このような異なるアーキテクチャ間でのプログラムの移植性を確保するための再定義が行われています．

● ELFヘッダの中身

リスト2.3では，Elf32_Ehdrの先頭にe_ident[]という領域が確保されています．そのサイズはEI_NIDENTとなっていますが，これは/usr/include/sys/elf_common.hで，

```
#define EI_NIDENT 16    /* Size of e_ident array. */
```

のように定義されているので，ELF形式のファイルの先頭には，かならずこの16バイトの領域が存在することになります．ここには，マジック・ナンバやELF32/ELF64の区別，ファイルのエンディアン，対象となるOSの種類などの情報が格納されています．ここだけバイト単位になっている理由は，エンディアンやELF32/64などの情報がわからないことには，ファイルのほかの部分を読めないため，ここだけはエンディアンやビット数に依存することなく読み込める必要があるからです．

リスト2.3 ELF32ヘッダ(elf32.h)

```
/*
 * ELF header.
 */

typedef struct {
  unsigned char e_ident[EI_NIDENT]; /* File identification. */
  Elf32_Half  e_type;               /* File type. */
  Elf32_Half  e_machine;            /* Machine architecture. */
  Elf32_Word  e_version;            /* ELF format version. */
  Elf32_Addr  e_entry;              /* Entry point. */
  Elf32_Off   e_phoff;              /* Program header file offset. */
  Elf32_Off   e_shoff;              /* Section header file offset. */
  Elf32_Word  e_flags;              /* Architecture-specific flags. */
  Elf32_Half  e_ehsize;             /* Size of ELF header in bytes. */
  Elf32_Half  e_phentsize;          /* Size of program header entry. */
  Elf32_Half  e_phnum;              /* Number of program header entries. */
  Elf32_Half  e_shentsize;          /* Size of section header entry. */
  Elf32_Half  e_shnum;              /* Number of section header entries. */
  Elf32_Half  e_shstrndx;           /* Section name strings section. */
} Elf32_Ehdr;
```

● ELF32形式のリトル・エンディアンか確認してみる

e_ident[]は，**表2.2**のフィールドから構成されています．e_ident[]の各フィールドにアクセスする場合には，インデックス値，代入値ともに，sys/elf_common.hで定義されている定数を利用します．たとえば，当該のファイルがELF32形式のリトル・エンディアンであることを確認したい場合には，以下のように書きます．

```
Elf_Ehdr *ehdr;
if ((ehdr->e_ident[EI_CLASS] == ELFCLASS32)
  &&(ehdr->e_ident[EI_DATA]  == ELFDATA2LSB))
  …
```

さらに，FreeBSD/i386では，ELFCLASS32，ELFDATA2LSBはelf_generic.hでそれぞれELF_CLASS，ELF_DATAに再定義されているので[注2.2]，以下のように書くことができます．

```
Elf_Ehdr *ehdr;
if ((ehdr->e_ident[EI_CLASS] == ELF_CLASS)
  &&(ehdr->e_ident[EI_DATA]  == ELF_DATA))
  …
```

このようにしておけば，Elf_Ehdrと同様に，異なるアーキテクチャ間での移植性を確保することができます．

表2.2 e_ident[]の構成一覧

インデックス	内　容	取りうる値
0	マジック・ナンバ	0x7f
1	マジック・ナンバ	'E'
2	マジック・ナンバ	'L'
3	マジック・ナンバ	'F'
4	ELF32/ELF64の区別	ELFCLASSNONE ELFCLASS32 ELFCLASS64
5	エンディアン	ELFDATANONE ELFDATA2LSB ELFDATA2MSB
6	ELFフォーマットのバージョン	EV_NONE EV_CURRENT
7	OSのABI（後述）	ELFOSABI_NETBSD ELFOSABI_LINUX ELFOSABI_SOLARIS ELFOSABI_FREEBSD ...
8	OSのABIのバージョン	OSのABI依存
9～15	予約（パディング）	0

注2.2：このような気配りは，プログラマとしてぜひとも見習いたい．

● ELFヘッダの各フィールドの意味

リスト2.3では，e_ident[]の後にさまざまなフィールドが続いています．**表2.3**に各フィールドの意味と，取りうる値を説明します．なお，取りうる値に関しては，紙面のつごう上，すべてを載せていません．これらはelf_common.hで定義されているので，必要ならばman elf，もしくはelf_common.hを参照してください．

● readelfコマンドによる確認

ELFヘッダの内容は，readelfコマンドで確認することができます．たとえば**リスト2.4**は，elfsamp.oに対してreadelf -hを実行して，ELFヘッダを表示した結果です．

リスト2.4では，ELF Headerの先頭にMagicという16バイトの領域が表示されています．これが**リスト2.3**で説明した，e_ident[]の内容になります．その下にはClass，Data，Version，OS/ABI，ABI Versionといったフィールドの解析結果が続きますが，これはe_ident[]の内容を解析したものです．

リスト2.4では，ABI Versionの後には，Type，Machine，Version，…と続いています．これらはそれぞれ，**表2.3**で説明しているe_type，e_machine，e_version，…の解析結果です．**リスト2.4**と**表2.2**，**表2.3**を比較してみてください．**表2.2**と**表2.3**で説明しているメンバの内容が，**リスト2.4**で表示されている内容に一つ一つ対応することがわかります．

表2.3 ELFヘッダの内容

フィールド	説 明
e_type	ファイルのタイプを指定する．ELF形式には，実行可能ファイル(ET_EXEC)，オブジェクト・ファイル(ET_REL)，コア・ファイル(ET_CORE)，ダイナミック・リンク・ライブラリ(ET_DYN)の四つの種類がある(あまり知られていないことだが，コア・ダンプ時に生成されるコア・ファイルもELF形式)
e_machine	マシン・アーキテクチャ．Pentiumの場合には，EM_386になる
e_version	ファイルのバージョン．通常はEV_CURRENTになる
e_entry	実行可能ファイルの場合には，実行開始アドレス(エントリ・ポイント)が格納される．FreeBSDの場合には，第1章で説明した_start()のアドレスになる
e_phoff	ELFファイル中の，プログラム・ヘッダ・テーブルの位置．ファイルの先頭からのバイト・オフセットで表す
e_shoff	ELFファイル中の，セクション・ヘッダ・テーブルの位置．ファイルの先頭からのバイト・オフセットで表す
e_flags	現状では未使用
e_ehsize	ELFヘッダのサイズ．sizeof(Elf_Ehdr)と等しくなる
e_phentsize	プログラム・ヘッダのサイズ．sizeof(Elf_Phdr)と等しくなる
e_phnum	プログラム・ヘッダの個数．e_phentsize×e_phnumが，プログラム・ヘッダ・テーブルのサイズになる
e_shentsize	セクション・ヘッダのサイズ．sizeof(Elf_Shdr)と等しくなる
e_shnum	セクション・ヘッダの個数．e_shentsize×e_shnumが，セクション・ヘッダ・テーブルのサイズになる
e_shstrndx	セクション名格納用のセクションの，セクション番号を表す(各セクションの名前は，e_shstrndxが指すセクションに格納されている)