[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3. リンカスクリプト

リンクは *リンカスクリプト*によって制御される.このスクリプトは リンカコマンド言語で書かれる.

リンカスクリプトの主目的は,入力ファイルのセクションを出力ファイルに どのようにマッピングするか, そして,出力ファイルのメモリレイアウトを 調整することである.ほとんどのリンカスクリプトはそれ以上のことは しない.ただし,必要であれば,リンカスクリプトを使って, その他のたくさんの操作を行なわせることができる.以下で説明する コマンドが使える.

リンカは常にリンカスクリプトを使う.読者自身が提供しなくても, リンカは,リンカの実行形式にコンパイルされて組み込まれたデフォルトの スクリプトを使う.コマンド行オプション `--verbose' を使うと, デフォルトのリンカスクリプトを表示することができる.あるコマンド行 オプションは,`-r' や `-N' のように,デフォルトのリンカ スクリプトに影響する.

コマンド行オプション `-T' を使うと,読者自身のリンカスクリプトを 与えることができる.これを行なうと,読者のリンカスクリプトは, デフォルトのリンカスクリプトを置き換える.

また,リンカへの入力ファイルとして指定することで,リンクされるファイルの ように見えても,暗黙のリンカスクリプトとして使うことができる. See section 3.11 暗黙のリンカスクリプト.

3.1 基本的なリンカスクリプトの考え方
3.2 リンカスクリプトの形式
3.3 簡単なリンカスクリプトの例
3.4 簡単なリンカスクリプトのコマンド
3.5 シンボルへの値の代入
3.6 SECTIONS コマンド
3.7 MEMORY コマンド
3.8 PHDRS コマンド
3.9 VERSION コマンド
3.10 リンカスクリプトの式
3.11 暗黙のリンカスクリプト

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.1 基本的なリンカスクリプトの考え方

我々は,リンカスクリプト言語を説明するのに, いくつかの基本的な概念と語彙を定義しておく必要がある.

リンカは複数の入力ファイルを組み合わせて一つの出力ファイルとする. 出力ファイルと各入力ファイルは, *オブジェクトファイル形式*として 知られる特別なデータ形式になっている.各ファイルは, *オブジェクトファイル*と呼ばれる. 出力ファイルは, *executable* と呼ばれることが多いが, 我々の目的に合わせて,これもオブジェクトファイルと呼ぶことにする. 各オブジェクトファイルには, とりわけ, *セクション*のリストが 入っている. 入力ファイル中のセクションをよく *入力セクション*と 呼ぶ. 同様に, 出力

ファイル中のセクションは *出力セクション*である.

オブジェクトファイルの中のセクションは,それぞれ名前と大きさがある. ほとんどのセクションには,関連するデータのブロックがあり, *セクションの内容*として知られている. セクションには *ロード可能*という印をつけることができる. これは,そのセクションの内容が,出力ファイルを実行した時に メモリにロードされるはずであるということを意味する. 内容のないセクションを *割り当て可能*とすることができる. これは,メモリのある領域が別途設定されるが,何も特別なものは ここにはロードされないということを意味する(場合によっては, このメモリ領域はゼロで初期化しなければならない). ロード可能でも割り当て可能でもないセクションは,多くの場合, 一種のデバッグ情報を保持している.

ロード可能か割り当て可能なあらゆる出力セクションには二つのアドレスが ある.一つは *VMA*,仮想メモリアドレスである.これは,出力 ファイルを実行する時のセクションのアドレスである.二つ目は, *LMA*,ロードメモリアドレスである.これは,セクションがロードされる アドレスである.ほとんどの場合,この二つのアドレスは同じになる. 違うアドレスになる例は,データセクションが ROM にロードされて, その後, プログラムの起動時に RAM にコピーされる場合である (この方法は,ROM ベースのシステムでグローバル変数を初期化するのに 良く使われる). この場合,ROM のアドレスが LMA であり,RAM のアドレスが VMA になる.

オブジェクトファイル中のセクションは,`objdump` に `-h' オプションを指定することでみることができる.

各オブジェクトファイルは,また,*シンボル*のリストも持っている. これは,*シンボル表* として知られている.シンボルは定義済みで あっても未定義であっても良い.シンボルにはそれぞれ名前があり, 定義済みのシンボルにはそれぞれ,とりわけアドレスがある. C または C++ のプログラムをオブジェクトファイルにコンパイルすると, 定義済み関数やグローバル変数,静的変数のそれぞれについて定義済み シンボルを得ることになる.入力ファイルで参照されている 未定義関数やグローバル変数はそれぞれ,未定義シンボルとなる.

オブジェクトファイル中のシンボルは,`nm` か,`objdump` に `-t' オプションを指定することで見ることができる.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.2 リンカスクリプトの形式

リンカスクリプトはテキストファイルである。

リンカスクリプトは,コマンドの列として書く.各コマンドは, 後ろに引数が続くこともあるキーワードか, シンボルへの代入である. コマンドを区切るのにセミコロンを使うことができる.空白は一般に 無視される.

ファイル名や形式名などの文字列は,通常そのまま書くことができる. ファイル名にカンマなどの文字が含まれている場合は,それが ファイル名の区切りとなるのを避けるために,そのファイル名を ダブルクォートで囲むことができる.

リンカスクリプトには,ちょうど C のように,`/*' と `*/' で区切ることでコメントを入れることができる.Cの場合と同様, コメントは文法的には空白に同じである.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.3 簡単なリンカスクリプトの例

多くのリンカスクリプトは極めて単純である.

可能な限りもっとも単純なリンカスクリプトは,ただ一つのコマンド `SECTIONS' があるだけである.`SECTIONS' コマンドを使って, 出力ファイルのメモリレイアウトを記述するのである.

`SECTIONS' コマンドは強力なコマンドである.以下にその簡単な 使い方を示す.読者のプログラムが, コードと初期化済みデータ,未初期化データ だけからなっているとしよう.これらが,それぞれ,`.text', `.data',`.bss' セクションに入っているとする. さらに,入力ファイルに現れるセクションはこれだけとする.

例として,コードをアドレス 0x10000 に置くこと,データがアドレス 0x8000000 から始まるとする.これを行なうリンカスクリプトは以下の通り.

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

`SECTIONS' コマンドは,キーワード `SECTIONS' をまず書き, 続けて,シンボルの代入と,中括弧に囲まれた出力セクション記述の 列を置く.

上の例で,`SECIONS' コマンド内の最初の行は,特別なシンボル `.',位置カウンタの値を設定している. 出力セクションのアドレスを 何か他の方法(これについては後述)を使って指定しない時は,アドレスは 位置カウンタの現在が設定される.位置カウンタは,その後,出力セクション の大きさだけ加算される.`SECTIONS' コマンドの開始時点では, 位置カウンタの値は `0' である.

二行目では,出力セクション `.text' を定義している. コロンは文法的に必要だが,現在では無視される.出力セクション名の後の 中括弧の中に,この出力セクションに置くべき,入力セクション名を 列挙する.`*' は,任意のファイル名に一致するワイルドカードである. `*(.text)' という式は,全入力ファイルの全入力 `.text' セクションを 表す.

出力セクション `.text' が定義された時の位置カウンタは `0x10000' なので,出力ファイルの `.text' セクションのアドレスは `0x10000' に設定される.

その他の行は,出力ファイルの `.data' と `.bss' セクションを 定義している.リンカは,出力セクション `.data' をアドレス `0x8000000' に置く.リンカが出力セクション `.data' を 置いた後,位置カウンタの値は `0x8000000' に,出力セクション `.data' の大きさを足したものになる.その効果は,出力 セクション `.bss' をメモリ中で出力セクション `.data' の直後に 置くことである.

リンカは,必要なら位置カウンタの大きさを増やして,各出力セクションが 必要とするアラインメントになることを保証する.この例では, `.text' と `.data' セクションに指定されたアドレスは, おそらく,どんなアラインメントの制約であっても満たしているだろう. ただし,`.data' と `.bss' の間に小さな隙間を作る可能性がある.

これでお終い! 単純だが,完全なリンカスクリプトである.

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

3.4 簡単なリンカスクリプトのコマンド

本節では,単純なリンカスクリプトコマンドについて説明する.

3.4.1 入口点の設定
3.4.2 ファイルを扱うコマンド
3.4.3 オブジェクトファイル形式を扱うコマンド

3.4.4 その他のリンカスクリプトコマンド

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

3.4.1 入口点の設定

プログラムで最初に実行される命令を *エントリポイント*と呼ぶ. リンカスクリプトコマンド `ENTRY` を使って,エントリポイントを 設定することができる. 引数はシンボル名である.

```
ENTRY(symbol)
```

エントリポイントを設定するにはいくつか方法がある.リンカは,エントリポイント を設定するのに,以下の方法を順にそれぞれ試し,どれか一つが成功したら そこで停止する.

- コマンド行オプションの `-e' *entry*
- リンカスクリプトの `ENTRY(symbol)` コマンド
- 定義されていれば, シンボル `start` の値
- 存在するなら, `.text` セクションの先頭のバイトのアドレス
- アドレス `0`

[<][>] [<<][Up][>>] [Top] [Contents] [Index] [?]

3.4.2 ファイルを扱うコマンド

ファイルを取り扱うリンカスクリプトコマンドを説明する.

`INCLUDE filename`
: リンカスクリプト *filename* を現在の地点で取り込む. このスクリプトファイルは, カレントディレクトリ, それに `-L` オプションで指定されたディレクトリから検索される. `INCLUDE` は 10 段まで入れ子にできる.

`INPUT(file, file, ...)`
`INPUT(file file ...)`
: `INPUT` コマンドは, リンカに対し, 指定されたファイルを コマンド行で指定したかのようにリンクに含めることを指示する.

例えば,リンクを行なう時に必ず `subr.o' を含めたいが, 毎回コマンド行で指定するのは面倒だという場合は,リンカスクリプトに `INPUT (subr.o)' と入れれば良い.

実際,望むなら,全ての入力ファイルをリンカスクリプトに列挙することが できるのであり,そうすれば `-T' オプション以外は何も指定せずに リンカを起動することができる.

リンカはまず,カレントディレクトリにあるファイルをオープンしようとする. 見つからなかった場合には,アーカイブライブラリの検索パスから探す. Command Line Options の `-L' についての 説明を参照のこと.

`INPUT(-l*file*)' とすると, ld は, コマンド行引数 `-l' の場合と同じように, 名前を lib*file*.a に変換する.

INPUT コマンドを暗黙のリンカスクリプトで使うと, 指定されたファイルは,リンカスクリプトファイルが指定された位置で, リンクに取り込まれる.これは,アーカイブの検索に影響する.

GROUP(*file*, *file*, ...)
GROUP(*file file* ...)

GROUP コマンドは INPUT に似ているが, ただし, 指定するファイルは 全てアーカイブでなければならず, 未定義参照がなくなるまで 繰り返し検索されるという点が異なる. Command Line Options の `-(' の説明を参照のこと.

OUTPUT(*filename*)

OUTPUT コマンドは出力ファイルを指定する.リンカスクリプトで OUTPUT(*filename*) を使うのは,コマンド行で `-o *filename*' を使うのに全く同じである(see section Command Line Options). 両方が指定された場合は,コマンド行オプションが優先する.

OUTPUT コマンドを使って,出力ファイルのデフォルト名を, 通常のデフォルトの `a.out' 以外のものに定義することができる.

SEARCH_DIR(*path*)

SEARCH_DIR コマンドは, ld がアーカイブライブラリを 検索するパスのリストに *path* を追加する. リンカスクリプトで SEARCH_DIR(*path*) を使うのは,コマンド行で `-L *path*' を使うのに全く同じである(see section Command Line Options). 両方が指定された場合は,両方のパスを検索する. コマンド行オプションで指定されたパスが先に検索される.

STARTUP(*filename*)

STARTUP コマンドは,ちょうど INPUT コマンドのようであるが, *filename* が,あたかもコマンド行で先頭に指定されたかのように, リンクされる最初の入力ファイルになるという点が異なる. これは,エントリポイントが常に最初のファイルの先頭になるようなシステムを 使っている時に役に立つ.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.4.3 オブジェクトファイル形式を扱うコマンド

オブジェクトファイル形式を取り扱うリンカスクリプトコマンドが 二つある.

OUTPUT_FORMAT(*bfdname*)
OUTPUT_FORMAT(*default*, *big*, *little*)

OUTPUT_FORMAT コマンドは,出力ファイルに使う BFD 形式を 指定する(see section 5. BFD). OUTPUT_FORMAT(*bfdname*) を 使うのは,コマンド行で `-oformat *bfdname*' を使うのと 全く同じである (see section Command Line Options). 両方指定された場合は,コマンド行オプションが優先する.

引数が三つの OUTPUT_FORMAT を使って,コマンド行オプション `-EB' と `-EL' に応じて異なる形式を使うことができる. これにより,リンカスクリプトが,希望のエンディアンに応じた出力形式を設定できるようになる.

`-EB' も `-EL' も指定されない場合は,出力形式は, 最初の引数 *default* になる.`-EB' を指定した場合は, 出力形式は二番目の引数 *big* になる.`-EL' を指定した 場合は,出力形式は三番目の引数 *little* になる.

例えば,MIPS ELF ターゲット用のデフォルトリンカスクリプトは以下の コマンドを使っている.

```
OUTPUT_FORMAT(elf32-bigmips, elf32-bigmips, elf32-littlemips)
```

これは,出力ファイルのデフォルトの形式は `elf32-bigmips' であり, コマンド行オプション `-EL' を指定した場合は,出力ファイルは `elf32-littlemips' の形式で作られるということを述べている.

TARGET(*bfdname*)
: TARGET コマンドは,入力ファイルを読む時に使う BFD 形式を 指定する. これは,後に現れる INPUT コマンドと GROUP コマンドに影響する.このコマンドは,コマンド行で `-b *bfdname*' オプションを使うのに同じである(see section Command Line Options). TARGET コマンドは使われているが OUTPUT_FORMAT はない という場合は,最後の TARGET コマンドが出力ファイルの形式を 設定するのにも使われる.See section 5. BFD.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.4.4 その他のリンカスクリプトコマンド

他にもいくつか,リンカスクリプトコマンドがある.

ASSERT(*exp*, *message*)
: *exp* がゼロでないことを保証する.もしゼロの場合は, あるエラーコードでリンカを終了し,*message* を表示する.

EXTERN(*symbol symbol ...*)
: *symbol* を強制的に未定義シンボルとして出力ファイルに入れる. こうすると, 例えば, 標準ライブラリからさらにモジュールをリンクする 引きがねとなる. それぞれの EXTERN に複数の *symbol* を 指定することができ,また,EXERN を複数回指定できる. このコマンドは,コマンド行オプション `-u' と同じ効果を持つ.

FORCE_COMMON_ALLOCATION
: このコマンドは, コマンド行オプションの `-d' と同じ効果を持つ. 再配置可能な出力が指定された(`-r')場合でもコモンシンボルに スペースを割り当てる.

NOCROSSREFS(*section section ...*)
: このコマンドを指定すると, 特定の出力セクション間で参照があった場合に エラーを発行する.

ある種のプログラム, とりわけ組み込みシステムでオーバーレイを使っている 場合, あるセクションはメモリにロードされるが, もう一つのセクションは ロードされないということがある. この二つのセクション間で直接的な参照が あるとエラーになる. 例えば, 一つのセクションの中のコードが, 他のセクションで定義されている 関数を呼び出すとエラーになる.

NOCROSSREFS コマンドには出力セクション名のリストを指定する. ここで指定したセクション間に相互参照を見つけると, エラーを出し, ゼロでない終了コードを返す. NOCROSSREFS コマンドは, 入力セクション名でなく,出力セクション名を使うことに注意.

OUTPUT_ARCH(*bfdarch*)
: 特定の出力機種アーキテクチャを指定する.引数は,BFD のバックエンド ルーチンで使われる名前の一つとする(see section 5. BFD). あるオブジェクトファイルのアーキテクチャは,objdump プログラムに `-f' オプションを指定することで見ることができる.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.5 シンボルへの値の代入

リンカスクリプトでシンボルに値を代入することができる. これは,シンボルをグローバルシンボルとして定義する.

3.5.1 単純な代入
3.5.2 PROVIDE

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.5.1 単純な代入

シンボルに値を代入するのに, 以下の C の代入演算子が使える.

```
symbol = expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
symbol <<= expression ;
symbol >>= expression ;
symbol &= expression ;
symbol |= expression ;
```

一番目の場合は,*symbol* を定義し,その値を *expression* にしている.それ以外は,*symbol* は定義済みでなければならず, 値は適切に調整される.

特別なシンボル名 `.' は,位置カウンタを表す. これは,SECTIONS コマンドの中でだけ使うことができる.

expression の後のセミコロンは必須である.

式は以下のように定義する.3.10 リンカスクリプトの式 参照.

シンボル代入はそれ自身独立したコマンドとして書くこともできるし, SECTIONS コマンド内の文としても,あるいは SECTIONS コマンド内の出力セクション記述の一部としても 書くことができる.

シンボルのセクションは,その式のセクションになる. 詳細については,3.10.6 式のセクション 参照.

以下の例では,三つの異なる場所でシンボルの代入を行なうことができることを 示している.

```
floating_point = 0;
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
    }
  _bdata = (. + 3) & ~ 4;
  .data : { *(.data) }
}
```

この例では,シンボル `floating_point' がゼロとして定義される. シンボル `_etext' は,最後の `.text' 入力セクションの直後の アドレスとして定義される.シンボル `_bdata' は, `.text' 出力セクションの直後のアドレスを,4バイト境界まで切り上げた ものとして定義される.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.5.2 PROVIDE

場合によっては, リンカスクリプトでシンボルを定義するのは, そのシンボルが 参照された時だけ, かつ, リンク対象となるどのオブジェクトファイルでも 定義されていないときだけにするのが望ましい場合がある. 例えば, 従来のリンカは `etext' というシンボルを定義していた. しかし, ANSI C はユーザが `etext' を関数名として使用しても エラーにならないことを要求している. PROVIDE というキーワードを 使うと, `etext' のようなシンボルを, 参照はされているが 定義されていない場合にだけ定義することができる. 書き方は PROVIDE(*symbol = expression*) となる.

以下の例は,PROVIDE を使って `etext' を定義している.

```
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
      PROVIDE(etext = .);
    }
... }
}
```

この例では,プログラムが `_etext' (頭にアンダースコア付き) を定義したなら,リンカは多重定義のエラーを出す. 一方,プログラムが `etext' (頭にアンダースコアなし)を定義すると, リンカは黙ってプログラムの定義を使う. プログラムが `etext' を参照はしているが定義していない場合, リンカはリンカスクリプトの定義を使う.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6 SECTIONS コマンド

SECTIONS コマンドは、リンカに、入力セクションから出力セクションへの 対応方法、それに出力セクションのメモリ配置方法について指示する。

SECTIONS コマンドの形式は以下の通り。

```
SECTIONS
{
  sections-command
  sections-command
  ...
}
```

各 *sections-command* は、以下のうちのどれか一つである。

- ENTRY コマンド (see section Entry command)
- シンボル代入 (see section 3.5 シンボルへの値の代入)
- 出力セクション記述
- オーバーレイ記述

ENTRY コマンドとシンボル代入は、位置カウンタを使えるように SECTIONS コマンド内で使用することができる。これにより、リンカスクリプトも理解しやすくなる。出力ファイルのレイアウトの意味のある 点でこれらのコマンドを使うことができるからである。

出力セクション記述とオーバーレイ記述を以下に示す。

リンカスクリプトで SECTIONS コマンドを使わないと、リンカは 各入力セクションを同じ名前の出力セクションに、セクションが入力ファイルに 現れた順に置いていく。例えば、全ての入力セクションが先頭のファイルに存在した 場合は、出力ファイルのセクション順は先頭の入力ファイルでの順序に一致する。先頭のセクションはアドレス 0 に置かれる。

3.6.1 出力セクション記述
3.6.2 出力セクション名
3.6.3 出力セクションアドレス
3.6.4 入力セクション記述
3.6.5 出力セクションデータ
3.6.6 出力セクションキーワード
3.6.7 出力セクション削除
3.6.8 出力セクション属性
3.6.9 オーバーレイ記述

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.1 出力セクション記述

出力セクションの完全な記述は以下のようになる。

```
section [address] [(type)] : [AT(lma)]
```

```
{
  output-section-command
  output-section-command
  ...
} [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
```

ほとんどの出力セクションは、ほとんどのオプションのセクション属性を使用しない。

section の両側の空白は、セクション名が曖昧にならないように、必須に なっている。コロンと中括弧も必須である。改行や他の空白はあってもなくても良い。

各 *output-section-command* は以下のどれか一つである。

- シンボル代入 (see section 3.5 シンボルへの値の代入)
- 入力セクション記述 (see section 3.6.4 入力セクション記述)
- 直接取り込むデータ値 (see section 3.6.5 出力セクションデータ)
- 特別な出力セクションキーワード (see section 3.6.6 出力セクションキーワード)

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.2 出力セクション名

出力セクションの名前は *section* になる。*section* は、出力形式の制約を満たしていなければならない。`a.out` のように、限られた数のセクションしかサポートしていない形式では、セクション名は その形式でサポートされている名前の一つでなければならない(例えば、`a.out` の場合、`.text'、`.data', `.bss' しか 使えない)。出力形式が任意個数のセクションに対応していていも、アルファベット ではなく数字しか使えない場合(例えば Oasys の場合)、その名前は、数字文字列を引用符で囲って与えなければならない。セクション名は任意の文字の 並びで構成可能だが、カンマのような通常ではない文字を含む名前は引用符で 囲まなければならない。

出力セクション名 `/DISCARD/' は特別である。3.6.7 出力セクション削除.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.3 出力セクションアドレス

address は、出力セクションの VMA(仮想メモリアドレス)を表す式である。*address* を指定しない場合、*region* が存在すればそれに基づいて 設定し、存在しない場合は、位置カウンタの現在値に基づいて設定する。

address を指定した場合、出力セクションのアドレスは正確にその値に 設定される。*address* も *region* も指定しない場合、出力セクションの アドレスは、位置カウンタの現在値を出力セクションに必要なアラインメントに 揃えた値に設定される。出力セクションのアラインメント要求は、出力セクション に含まれる入力セクションのうち、最も厳しいアラインメントになる。

例えば

```
.text . : { *(.text) }
```

と

```
.text : { *(.text) }
```

は、微妙に異なる。最初のは、出力セクション `.text' のアドレスを 位置カウンタの現在値に設定する。二番目は、入力セクション `.text' の 最も厳しいアラインメントに揃えた、位置カウンタの現在値に設定される。

address は任意の式で良い。3.10 リンカスクリプトの式. 例えば、セクションを 0x10 バイト境界に揃えて、セクションのアドレスの下位4ビットがゼロに なるようにしたいのであれば、以下のようにすれば良い。

```
.text ALIGN(0x10) : { *(.text) }
```

これが動作するのは、`ALIGN` が、現在の位置カウンタを指定された 値にまで揃えるからである。

セクションに対して *address* を指定すると、位置カウンタの値が 変わる。

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.4 入力セクション記述

最も良く使う出力セクションコマンドは、入力セクション記述である。

入力セクション記述は、最も基本的なリンカスクリプト操作である。出力セクションを使って、プログラムのメモリレイアウト方法を指定する。入力セクション記述を使って、入力ファイルをどのようにメモリレイアウトに 対応させるかを指示するのである。

3.6.4.1 入力セクションの基本
3.6.4.2 入力セクションのワイルドカードパターン指定
3.6.4.3 コモンシンボル用の入力セクション
3.6.4.4 入力セクションとゴミ集め
3.6.4.5 入力セクションの例

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.4.1 入力セクションの基本

入力セクション記述は、ファイル名の後ろにオプションで、セクション名の リストを括弧で囲んだものを置いたものから構成される。

ファイル名とセクション名にはワイルドカードを指定することもできる。これについては後で説明する (see section 3.6.4.2 入力セクションのワイルドカードパターン指定).

最も良く使われる入力セクション記述は、全ての入力セクションを、出力セクションの特定の名前で取り込むことである。例えば、全ての入力 `.text' セクションを取り込むには、以下のように書く。

```
*(.text)
```

ここで、`*' は、任意のファイル名に一致するワイルドカードである。ファイル名ワイルドカードに一致するものからファイルのリストを除外するには、EXCLUDE_FILE を使うと、EXCLUDE_FILE で指定したリストにあるものを除いた全てのファイルに一致する。例えば、

```
(*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors))
```

と書くと、`crtend.o' と `otherfile.o' を除く全てのファイルの .ctors セクションを取り込む。

複数のセクションを取り込むには二つの方法がある。

```
*(.text .rdata)
*(.text) *(.rdata)
```

この二つの違いは、入力セクション `.text' と `.rdata' が、出力セクションに現れる順番にある。最初の例では、`.text' と `.rdata' が混ざり合う。二番目の例では、全ての `.text' 入力セクションがまず現れ、次に全ての `.rdata' 入力セクションが 続く。

ファイル名を指定することで、特定のファイルからセクションを取り込むことが できる。こういうことをしたくなるのは、メモリの特定の位置に配置される 必要がある特別なデータを含むファイルがある時である。例えば、以下のように する。

```
data.o(.data)
```

ファイル名にセクションのリストをつけないと、入力ファイル中の全セクション が出力セクションに取り込まれる。

```
data.o
```

ワイルドカード文字を全く含まないファイル名を使った場合、リンカはまず、指定されたファイルがコマンド行や INPUT コマンドでも指定されていない かどうかを見る。指定されていなければ、そのファイルを入力ファイルとして、コマンド行で指定されたかのように、オープンしようとする。これは INPUT コマンドとは違っていることに注意。アーカイブ検索パスは 検索しないのである。

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.4.2 入力セクションのワイルドカードパターン指定

入力セクション記述において、ファイル名またはセクション名のどちらか、あるいは両方ともワイルドカードのパターンとすることができる。

多くの例に見える `*' というファイル名は、簡単なファイル名用 ワイルドカードパターンである。

ワイルドカードパターは、Unix のシェルで使われているものに似ている。

`*'
: 任意個数の文字に一致する

`?'
: 任意の一文字に一致する

`[*chars*]'
: *chars* のうちのどれか任意の一文字に一致する。`-' 文字を使って、文字の範囲を指定するこ

とができる。`[a-z]' は、任意の小文字に 一致する

`¥'
直後の文字を引用する

ファイル名がワイルドカードで照合された場合, ワイルドカード文字は 文字 `/' には一致しない. (`/' は Unix でディレクトリ名を 区切るのに使われる.) ただし, 一個の `*' 文字からなるパターンは 例外で, それが `/' を含んでいようがいまいが,常にどんなファイル名 にも一致する. セクション名の場合は, ワイルドカード文字は `/' に一致する.

ファイル名ワイルドカードは, コマンド行か INPUT コマンドで 明示的に指定されたファイルにのみ一致する. リンカは, ワイルドカードを展開するためにディレクトリを検索することはしない.

一個のファイル名が複数のワイルドカードのパターンに一致するか, あるいは,ファイル名が明示的に現れており,ワイルドカードにも一致する 場合には,リンカスクリプトで最初に一致したものを使う.例えば, 以下の入力セクション記述の列はおそらく誤りである.`data.o' の 規則は使われないからである.

```
.data : { *(.data) }
.data1 : { data.o(.data) }
```

通常,リンカはリンク中に見えた順番で,ワイルドカードに一致した ファイルとセクションを置いていく. これを変えるには,SORT キーワードを,括弧内のワイルドカードパターンの前に置けば良い (例えば,SORT(.text*)).SORT キーワードを使うと, リンカは,出力ファイルに置く前に,ファイルやセクションを名前の 昇順にソートする.

入力セクションがどこに行くのか分からなくなったら,`-M' オプションを 指定して,マップファイルを作ると良い.マップファイルは,入力セクション が出力セクションにどのようにマップされるのかを正確に示す.

以下の例では,ワイルドカードのパターンを使って,どのようにファイルを 区分けするかを示している.リンカスクリプトは,リンカに対し, 全 `.text' セクションを `.text' に,前 `.bss' セクションを `.bss' に置くよう指示を出す.大文字で始まる全ファイルの `.data' セクションを `.DATA' に置く.それ以外の全てのファイルについては,`.data' セクションを `.data' に置く.

```
SECTIONS {
  .text : { *(.text) }
  .DATA : { [A-Z]*(.data) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.4.3 コモンシンボル用の入力セクション

コモンシンボルを表すには特別な記法が必要である.多くのオブジェクトファイル 形式では,コモンシンボルには特定の入力セクションがないからである. リンカはコモンシンボルを,`COMMON' という入力セクションにあったかの ように取り扱う.

`COMMON' セクションに対するファイル名は,他の入力セクションの 場合と同様に使うことができる.これ

を使って,ある特定の入力ファイルの コモンシンボルを一つのセクションに置き,他の入力ファイルのコモンシンボル を別のセクションに置くということもできる.

ほとんどの場合,入力ファイルのコモンシンボルは,出力ファイルの `.bss' セクションに置かれる.例えば,以下のようになる.

```
.bss { *(.bss) *(COMMON) }
```

オブジェクトファイル形式によっては,コモンシンボルに複数の種類が ある.例えば,MIPS ELF オブジェクトファイル形式は,標準のコモンシンボル と小さなコモンシンボルを区別する.この場合,リンカは,標準以外の種類の コモンシンボルに対し,異なった特別なセクション名を使う. MIPS ELF の場合,リンカは標準のコモンシンボルには `COMMON' を, 小さなコモンシンボルには `.scommon' を使う. こうすることで,異なる種類のコモンシンボルをメモリの異なる位置に マッピングすることができる.

古いリンカスクリプトで `[COMMON]' というのを目にすることが あるかもしれない.この記法は,現在では廃れていると考えられている. これは,`*(COMMON)' に同じである.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.4.4 入力セクションとゴミ集め

リンク時のゴミ集めが使用されている(`--gc-sections')時, 削除すべきでないセクションに印を付ける必要もあるだろう. これは,入力セクションのワイルドカードエントリを, KEEP(*(.init)) や KEEP(SORT(*)(.ctors))のように, KEEP()で囲めば良い.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.4.5 入力セクションの例

次の例は, 完全なリンカスクリプトである.ファイル all.o の 全セクションを読み込んで, 先頭位置が 0x10000 である セクション outputa の先頭に置いている. ファイル foo.o のセクション .input1 全体が 同じ出力セクション内のその直後に続く. foo.o のセクション .input2 が出力セクション outputb に置かれ, その後に foo1.o のセクション .input1 が続く. .input1 セクションと .input2 セクションが残っていれば 全て, 出力セクション outputc に置かれる.

```
SECTIONS {
  outputa 0x10000 :
    {
    all.o
    foo.o (.input1)
    }
  outputb :
    {
    foo.o (.input2)
    foo1.o (.input1)
    }
  outputc :
    {
    *(.input1)
    *(.input2)
    }
```

```
}
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.5 出力セクションデータ

データを明示的なバイト列で出力セクションに取り込むには, 出力用セクションコマンドとして, BYTE, SHORT, LONG, QUAD, SQUAD を使えば良い.各キーワードに 続けて,格納する値を表す式を括弧で囲んで置く (see section 3.10 リンカスクリプトの式). この式の値は,位置カウンタの現在値のところに格納される.

BYTE, SHORT, LONG, QUAD コマンドはそれぞれ, 1, 2, 4, 8 バイトを格納する.格納後,位置カウンタは格納された バイト数だけ加算される.

例えば,以下の例では,1 という値のバイトに続けて,`addr' という シンボルの 4 バイトの値を格納する.

```
BYTE(1)
LONG(addr)
```

64 ビットホストやターゲットを使う場合には,QUAD と SQUAD は 同じになる.どちらも,8 バイトあるいは 64 ビットの値を格納する. ホストとターゲットが両方 32 ビットである場合には, 式は 32 ビットとして 計算される.この場合,QUAD は 32 ビット値を 64 ビットに ゼロ拡張して格納し, SQUAD は 32 ビット値を 64 ビットに符合拡張 して格納する.

出力ファイルのオブジェクトファイル形式のエンディアンがはっきり決まっている なら,値はそのエンディアンで格納される.普通はそうなっている. オブジェクトファイル形式のエンデァインが決まっていない場合は,例えば, S-record がそうであるが,値は最初の入力オブジェクトファイルのエンディアンで 格納される.

これらのコマンドはセクション記述の内側でしか使えないことに注意. セクション記述とセクション記述の間には置くことができない. このため,

```
SECTIONS { .text : { *(.text) } LONG(1) .data : { *(.data) } }
```

とするとリンカがエラーを出すが,

```
SECTIONS { .text : { *(.text) ; LONG(1) } .data : { *(.data) } }
```

だと問題ない.

FILL コマンドを使うと,現在のセクションの「埋め潰しパターン」を 指定することができる. このコマンドの後には,括弧で囲んだ式を続ける. このセクション内の他に指定がないメモリ領域(例えば, 入力セクションで 必要とされるアラインメトのためにできた隙間)は, この式の 最下位二バイトを,必要なだけ繰り返すことで埋め潰される. FILL 文は, セクション定義内のその文が現れた後のメモリ位置をカバーする. FILL 文を複数回使うと, 出力セクションの異なる部分に 別々の埋め潰しパターンを指定することができる.

以下の例は,メモリの未指定領域を `0x9090' という値で埋める方法を 示している.

```
FILL(0x9090)
```

FILL コマンドは,出力セクション属性の `=fillexp' (see section 3.6.8.5 出力セクション埋め潰し)に似ているが,セクション全体ではなく, FILL コマンド以降のセクションの一部にのみ影響する. 両方指定した場合,FILL コマンドが優先する.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.6 出力セクションキーワード

出力セクションコマンドとして現れることが可能なキーワードが2,3ある.

CREATE_OBJECT_SYMBOLS
: このコマンドは,入力ファイル毎にシンボルを作ることをリンカに指示する. 各シンボルの名前は,対応する入力ファイル名になる.各シンボルの セクションは,CREATE_OBJECT_SYMBOLS コマンドが現れた 出力セクションになる.

  これは,a.out オブジェクトファイル形式の約束である. 通常,他のどのオブジェクトファイル形式でも使われない.

CONSTRUCTORS
: リンカは普通のとは異なる構文を使って, C++ のグローバルなコンストラクタ とデストラクタに対応する. ECOFF や XCOFF のように, 自由に使えるセクションに対応していないオブジェクトファイル形式を リンクする時は, リンカは名前によって C++ のグローバルなコンストラクタと デストラクタを自動的に認識する. これらのオブジェクトファイル形式の場合, CONSTRUCTORS コマンドはリンカに,コンストラクタ情報を出力セクション の,CONSTRUCTORS コマンドが現れた場所に置くよう指示する. その他のオブジェクトファイル形式では, CONSTRUCTORS コマンドは 無視される.

  シンボル __CTOR_LIST__ がグローバルコンストラクタ群の 先頭を, __DTOR_LIST__ が最後を表す. このリストの先頭のワードにはエントリ数が入る. その後に, 各コンストラクタ やデストラクタのアドレスが続き, 最後のワードにゼロが入る. コンパイラは, これらのコードを実際に実行するように調整を 行なわなければならない. これらのオブジェクトファイル形式に ついては, GNU C++ は, コンストラクタを通常サブルーチン __main から呼び出す. __main の呼びだしは main のスタートアップ コードに自動的に挿入される. デストラクタは, 通常,atexit を使ってか, あるいは関数 exit から直接実行される.

  COFF や ELF のように任意のセクション名に対応している オブジェクトファイル形式の場合は, GNU C++ は通常, グローバルコンストラクタとデストラクタのアドレスを セクション .ctors と .dtors に置く. リンカスクリプトに以下のようなコードを置くと, GNU C++ の 実行時コードが想定している形の表を構築する.

```
__CTOR_LIST__ = .;
LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
*(.ctors)
LONG(0)
__CTOR_END__ = .;
__DTOR_LIST__ = .;
LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
*(.dtors)
LONG(0)
```

```
__DTOR_END__ = .;
```

GNU C++ の初期化優先順サポートを使うとき,これは, グローバルコンストラクタの実行順序を制御するものであるが, リンク時にコンスタラクタをソートして,正しい順序で実行されることを 保証しなければならない.これには,CONSTRUCTORS コマンドを 使うところで,代わりに `SORT(CONSTRUCTORS)' を使うのである. また,.ctors セクションと .dtors セクションを使うところで, 単に `*(.ctors)' と `*(.dtors)' とするのでなく, `*(SORT(.ctors))' と `*(SORT(.dtors))' を使うのである.

普通は GCC と GNU ld がこれらの問題を自動的に取り扱うので, 読者自身は 考える必要はない. ただし, C++ を使っている場合で, 読者自身で リンカスクリプトを書く場合には, このことを考慮に入れる必要がある.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.7 出力セクション削除

リンカは内容が全くない出力セクションは作らない. これは, 入力ファイルに存在するかどうかわからない入力セクションを参照する時に 役に立つ. 例えば,

```
.foo { *(.foo) }
```

とすると, 出力ファイルに `.foo' というセクションが作られるのは, 少なくとも一つの入力ファイルに `.foo' というセクションが ある場合に限る.

出力セクションコマンドとして入力セクション記述以外の,シンボル代入 などを使うと,対応する入力セクションがなくても,出力セクションが いつでも作られる.

特別なセクション名 `/DISCARD/' を使うと, 入力セクションを捨て去ることができる. `/DISCARD/' という名前の出力セクションに割り当てられたセクションはどれも 最終的なリンクの出力に入らないのである.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.8 出力セクション属性

上で,以下のような出力セクションの完全な記述を示した.

```
section [address] [(type)] : [AT(lma)]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
```

既に *section*, *address*, *output-section-command* は 説明済みである.このセクションでは残りのセクション属性を説明する.

3.6.8.1 出力セクション型

3.6.8.2 出力セクション LMA
3.6.8.3 出力セクション領域
3.6.8.4 出力セクション phdr
3.6.8.5 出力セクション埋め潰し

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.8.1 出力セクション型

出力セクションはそれぞれ型を持つ.型は,キーワードを括弧で括って 示す.以下の型が定義されている.

NOLOAD
: セクションはロード可能でないという印が付けられる.このため, プログラムの実行時にメモリにロードされない.

DSECT
COPY
INFO
OVERLAY
: これらの型名は,後方互換性のためにサポートされており,滅多に 使われない.これらは全部同じ効果を持つ.セクションは, 割り当て可能ではないという印が付けられるので,プログラムの実行時に このセクションにはメモリが割り当てられない.

リンカは,出力セクションの属性を,それに対応する入力セクションを元に して設定する.セクション型を使うとそれを変更することができる. 例えば, 以下のスクリプト例 では, セクション ROM はメモリ位置 `0' に置かれ, プログラムの 起動時にロードされる必要がない. セクション ROM の内容は 通常通りリンカの出力ファイルに現れる.

```
SECTIONS {
  ROM 0 (NOLOAD) : { ... }
  ...
}
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.8.2 出力セクション LMA

各セクションには,仮想アドレス(VMA)とロードアドレス(LMA)がある. 3.1 基本的なリンカスクリプトの考え方参照.出力セクション記述に現れる アドレス式は VMA を設定する(see section 3.6.3 出力セクションアドレス).

リンカは普通は LMA を VMA に等しく設定する.AT キーワードを 使うとそれを変更できる.AT キーワードに続く式 *lma* は, セクションのロードアドレスを指定する.あるいは, `AT>*lma_region*' 式を使うと, セクションのロードアドレス用の メモリ領域を指定できる.See section 3.7 MEMORY コマンド.

この機能は, ROM イメージを簡単に作れるように設計されている. 例えば, 次のリンカスクリプトは, 三つの出力セクションを作る. 一つは `.text' セクションで, 0x1000 から始まる. もう一つは `.mdata' セクションで, その再配置アドレスが 0x2000 であっても, `.text' セクションの末尾にロードされる. 最後

の一つは,`.bss' セクションで,非初期化データを 0x3000 のアドレスに保持する. シンボル _data は 0x2000 という値に定義される. これは,LMA 値ではなく,VMA 値を保持する位置カウンタを保持していることを 表している.

```
SECTIONS
  {
  .text 0x1000 : { *(.text) _etext = . ; }
  .mdata 0x2000 :
    AT ( ADDR (.text) + SIZEOF (.text) )
    { _data = . ; *(.data); _edata = . ;  }
  .bss 0x3000 :
    { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

このようにして生成された ROM イメージと共に使うことを想定した, 実行時初期化コードでは, 以下のように, 初期化データを ROM イメージから その実行時アドレスにコピーするコードをなにかしら含めておかなければならない. このコードが,リンカスクリプトで定義されたシンボルの利点をどのように 生かしているかに注意して欲しい.

```
extern char _etext, _data, _edata, _bstart, _bend;
char *src = &_etext;
char *dst = &_data;

/* ROM has data at end of text; copy it. */
while (dst < &_edata) {
  *dst++ = *src++;
}

/* Zero bss */
for (dst = &_bstart; dst< &_bend; dst++)
  *dst = 0;
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.8.3 出力セクション領域

`>*region*' を使って,定義済みメモリ領域にセクションを 割り当てることができる.See section 3.7 MEMORY コマンド.

以下に簡単な例を示す。

```
MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM : { *(.text) } >rom }
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.8.4 出力セクション phdr

セクションに,`:*phdr*' を使って,定義済みプログラムセグメントを 割り当てることができる.See section 3.8 PHDRS コマンド. あるセクションが一つ以上のセグメントに割り当てられた場合, 後続の割当

済みセクションは全て, 明示的に :*phdr* 修飾子を 指定していない限り, 同じくこれらのセグメントに割り当てられる. :NONE を使って,セクションをどのセグメントにも全く置かないように 指示することができる.

以下に簡単な例を示す.

```
PHDRS { text PT_LOAD ; }
SECTIONS { .text : { *(.text) } :text }
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.8.5 出力セクション埋め潰し

セクション全体の埋め潰しパターンは,`=*fillexp*' を使って設定 できる.*fillexp* は一個の式である(see section 3.10 リンカスクリプトの式). 出力セクション内の, 他に指定がないメモリ領域(例えば, 入力セクションで 必要とされるアラインメントのためにできた隙間)は, この値の 最下位二バイトを,必要なだけ繰り返すことで埋め潰される.

埋め潰し値を変更するには,出力セクションコマンド内で FILL コマンドを使っても良い.3.6.5 出力セクションデータ 参照.

以下に簡単な例を示す.

```
SECTIONS { .text : { *(.text) } =0x9090 }
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.6.9 オーバーレイ記述

オーバーレイ記述は, 一個のメモリイメージの一部にロードされるが, 同じメモリアドレスで実行すべきセクションを簡単に記述する方法を 提供する. 実行時に, 一種のオーバーレイマネジャが, 必要に応じて オーバーレイセクションを実行時メモリアドレスにコピーしたり, そのアドレスから コピーすることになるだろう. これは, おそらく単にアドレッシングのビットを 操作することで行なわれる. この方法は, 例えば, ある特定のメモリ領域が 他の部分よりも高速であるというような場合に有効である.

オーバーレイは,OVERLAY コマンドを使って記述する. OVERLAY コマンドは,出力セクション記述同様, SECTIONS コマンドの中で使う. OVERLAY コマンドの完全な形式は以下の通り.

```
OVERLAY [start] : [NOCROSSREFS] [AT ( ldaddr )]
  {
    secname1
      {
        output-section-command
        output-section-command
        ...
      } [:phdr...] [=fill]
    secname2
      {
        output-section-command
        output-section-command
        ...
```

```
    } [:phdr...] [=fill]
  ...
} [>region] [:phdr...] [=fill]
```

キーワード OVERLAY 以外は全て省略可能であり, 各セクションには 名前がなくてはならない(上だと secname1 と secname2). OVERLAY 構文の中のセクション定義は, 普通の SECTIONS 構文 (see section 3.6 SECTIONS コマンド)の中でのセクション定義と同じであるが, ただし, OVERLAY 内のセクションにはアドレスやメモリ領域を定義できない.

セクションは, 全て同じ開始アドレスで定義される. セクションのロードアドレスは, OVERLAY 全体に使われる ロードアドレスから始まるメモリ領域に連続に置かれるように調整される. (普通のセクション定義と同じように, ロードアドレスは省略可能であり, デフォルトは開始アドレスになる. 開始アドレスもまた省略可能であり, デフォルトは位置カウンタの現在値になる).

キーワード NOCROSSREFS を使うと, セクション間で参照がある場合, リンカはエラーを報告する. セクションは全て同じアドレスに置かれるので, 普通は一つのセクションから他のセクションを直接参照することは意味を なさない. See section NOCROSSREFS.

OVERLAY 内のセクション毎に, リンカは自動的にシンボルを二つ定義する. シンボル __load_start_*secname* はセクションの開始ロードアドレス として定義される. シンボル __load_stop_*secname* は セクションの終端ロードアドレスとして定義される. *secname* の 中の文字で, C の識別子として正しくないものは削除される. C(あるいはアセンブラ)のコードで, これらのシンボルを使って, 必要に応じてオーバーレイされたセクションを移動することができる.

オーバーレイの最後で, 位置カウンタの値はオーバーレイの開始アドレスに 最大のセクションの大きさを加えた値に設定される.

以下に例を示す. ただし, これは SECTIONS 構文の中に 現れることに注意.

```
OVERLAY 0x1000 : AT (0x4000)
 {
   .text0 { o1/*.o(.text) }
   .text1 { o2/*.o(.text) }
 }
```

これは, .text0 と .text1 がどちらも開始アドレスが 0x1000 になるように定義する. .text0 はアドレス 0x4000 に ロードされ, .text1 は .text0 の直後にロードされる. 次のシンボルが定義される. すなわち, __load_start_text0, __load_stop_text0, __load_start_text1, __load_stop_text1 である.

オーバーレイ .text1 をオーバーレイ領域にコピーする C のコードは 以下のようになる.

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x1000, &__load_start_text1,
        &__load_stop_text1 - &__load_start_text1);
```

OVERLAY コマンドは単に省略表記に過ぎないことに注意. このコマンドで行なえることは全て, 他のもっと基本的なコマンドを 使って行なえるからである. 上の例は以下のようにも書ける.

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
__load_start_text0 = LOADADDR (.text0);
__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0);
.text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
```

```
__load_start_text1 = LOADADDR (.text1);
__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1);
. = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.7 MEMORY コマンド

リンカのデフォルトのコンフィギュレーションでは, 利用可能なメモリを 全て割り当て可能になっている. MEMORY コマンドを使うとこれを 変更することができる.

MEMORY コマンドはターゲットのメモリブロックの位置と大きさを記述する. これを使えば, リンカがどのメモリ領域を使って良いか, どの領域を使っては いけないかを記述することができる. その後,セクションを特定のメモリ領域に割り当てることができる. リンカは,メモリ領域に基づいてセクションアドレスを設定し, 一杯になった領域について警告を出す. リンカは利用できる領域に詰めるためにセクションを切り混ぜることはしない.

一個のリンカスクリプトには多くとも 1 回 MEMORY コマンドを 入れることができる. だが, メモリブロックは必要な数だけ定義することが できる.

```
MEMORY
  {
    name [(attr)] : ORIGIN = origin, LENGTH = len
    ...
  }
```

name は,リンカスクリプトでメモリ領域を参照するのに使われる名前である. 領域名は,リンカスクリプト以外では何の意味も持たない. 領域名は独立した名前空間に置かれるので, シンボル名やファイル名, セクション名と衝突することはない. 各メモリ領域は異なる名前でなければならない.

文字列 *attr*は, 省略可能な属性のリストであり, リンカスクリプトで 明示的に対応付されていないセクション用に特定のメモリを使うかどうかを指定する. 3.6 SECTIONS コマンド で説明したように,どれか入力セクションに対し 出力セクションを指定しないと,リンカは,入力セクションと同じ名前の 出力セクションを作る.領域属性を定義すると,リンカはそれを使って, 作成する出力セクションのメモリ領域を選択する.

文字列 *attr* は以下の文字だけから成っていなければならない.

`R'
: 読み出し専用セクション

`W'
: 読み書き両用セクション

`X'
: 実行可能セクション

`A'
: 割り当て可能セクション

`I'
: 初期化済みセクション

`L'

`I` に同じ.

`!'
後に続く属性の意味を否定する.

対応付されていないセクションが,`!' 以外の列挙された属性のどれかに 一致すれば,そのメモリ領域に置かれる. `!' 属性はこのテストの意味を逆に するので,対応付されていないセクションは,列挙された属性のどれにも 一致しない時に,メモリ領域に置かれる.

*origin*は,メモリ領域の開始アドレスを表す式である. この式は, メモリ割当が行なわれる前に定数に評価されるようになっている式で なければならない. これは,シンボルに相対のセクションは使えないという ことである.キーワード `ORIGIN` は `org` または `o` と 略記できる. ただし, 例えば, `ORG' とはできない.

len は,メモリ領域の大きさをバイト数で表す式である. *origin* 式同様,この式は, メモリ割当が行なわれる前に定数に 評価されるようになっている式でなければならない. キーワード `LENGTH` は `len` または `l` と略記できる.

以下の例では, 割り当てに使用できるメモリ領域が二つあることを指定している. 領域の一つは開始アドレスが 0 で大きさが 256 KB, もう一つは開始アドレスが `0x40000000` で大きさが 4MB である. リンカは,明示的にメモリ領域に対応付けられておらず,読み出し専用か 実行可能コードである全てのセクションをメモリ領域 `rom' に置く. 明示的にメモリ領域に対応付けられていないその他のセクションは,メモリ領域 `ram' に置かれる.

```
MEMORY
  {
    rom (rx)  : ORIGIN = 0, LENGTH = 256K
    ram (!rx) : org = 0x40000000, l = 4M
  }
```

一度メモリ領域を定義しておけば, 出力セクション属性 `>*region*' を使って,特定の出力セクションをそのメモリ領域に置くことができる. 例えば,`mem' という名前のメモリ領域があったとすると, 出力セクション定義で `>mem' を使うことができる. 出力セクションにアドレスが指定されていないと,リンカは,アドレスを そのメモリ領域内で次に使用可能なアドレスに設定する. ある領域に組み合わされた出力セクションを入れようとした時に, 出力セクションがその領域には 大き過ぎるときは, リンカがエラーメッセージを出す.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.8 PHDRS コマンド

ELF オブジェクトファイル形式は, *セグメント*とも呼ばれる, *プログラム・ヘッダ*を使っている. プログラムヘッダは,プログラムをどのようにメモリにロードすべきかを記述する. `objdump` プログラムに `-p' オプションを指定することで プログラムヘッダを表示できる.

ELF のプログラムをネイティブの ELF システムで実行する時は, システムのローダがプログラムヘッダを読んで,プログラムのロードの 方法を理解する.このためには,プログラムヘッダが正しく設定されて いなければならない. 本マニュアルでは, システムのローダが プログラム・ヘッダをどのように解釈するかの詳細については説明しない. 詳細については, ELF の ABI ドキュメントを見て欲しい.

リンカは, デフォルトで適切なプログラム・ヘッダを作成する. ただし, 場合によってはプログラムヘッダをさらに詳細に指定する必要が 出てくる.そのためには, PHDRS コマンドを使うと良い. PHDRS コマンドを使うと, リンカは, 自分ではプログラム・ヘッダを一切生成しない.

リンカは,ELF 出力ファイルを生成する時にしか,PHDRS コマンドに 注意を向けない.その他の場合は,リンカは単に PHDRS を無視する.

PHDRS コマンドの記法は以下の通りである. PHDRS, FILEHDR, AT, FLAGS という語はキーワードである.

```
PHDRS
{
  name type [ FILEHDR ] [ PHDRS ] [ AT ( address ) ]
        [ FLAGS ( flags ) ] ;
}
```

name はリンカ・スクリプトの SECTIONS コマンドの中で, 参照のみに使われる. 出力ファイルに置かれることはない. プログラムヘッダ名は独立した名前空間に置かれるので,シンボル名や ファイル名,セクション名と衝突することはない.各プログラムヘッダは 異なる名前でなければならない.

ある特定のプログラム・ヘッダ型はシステムローダがファイルから ロードするメモリセグメントについて記述する. リンカ・スクリプトでは, これらのセグメントの内容は, 割り当て可能な出力セクションをそのセグメントに置くことで指定する. あるセクションを特定のセグメントに 置くには,`:*phdr*' 出力セクション属性を使う. See section 3.6.8.4 出力セクション phdr.

特定のセクションを複数のセグメントに置くのは普通のことである. これは単にメモリの一セグメントが別のセグメントを含んでいるということに すぎない. `:*name*' を繰り返し指定して,セクションを含むべきセグメント毎に一回使う.

あるセクションを `:*phdr*' を使って一つ以上のセグメントに 置いたなら, リンカは, `:*phdr*' を指定していない, 後続の割り当て可能なセクションを,全て同じセグメントに置く. これは利便性のためのものである. 一般に隣接しているセクションの組は 全体として一個のセグメントに置かれるからである. :NONE を使って,デフォルトのセグメントを変更し,セクションを どのセグメントにも置かないようにすることができる.

キーワード FILEHDR と PHDRS をプログラムヘッダ型の 後に置くことで,さらにセグメントの内容について記述することができる. キーワード FILEHDR はセグメントが ELF ファイルヘッダを含んでいる べきであることを意味する.キーワード PHDRS は,セグメントが ELF プログラムヘッダ自身を含んでいるべきであることを意味する.

type は以下のどれかの一つである. 数字はキーワードの値を示す.

PT_NULL (0)
: 使用されないプログラム・ヘッダであることを示す.

PT_LOAD (1)
: このプログラム・ヘッダがファイルからロードされるセグメントを記述する ものであることを示す.

PT_DYNAMIC (2)
: 動的リンク情報のあるセグメントであることを示す.

PT_INTERP (3)
: プログラムのインタプリタ名のあるセグメントであることを示す.

PT_NOTE (4)
: ノート情報を持つセグメントであることを示す.

PT_SHLIB (5)
: 予約プログラムヘッダ型であり, ELF ABI により定義はされているが 指定は行なわれていない.

PT_PHDR (6)
: プログラム・ヘッダの置かれているセグメントであることを示す.

expression
: プログラムヘッダ型の数値を与える式である. これは上で定義されていない 型に対して使うことができる.

AT 式を使うと,あるセグメントをメモリの特定のアドレスにロード されるように指定することが可能である. これは, 出力セクション属性として使われる AT コマンドに 等価である(see section 3.6.8.2 出力セクション LMA). AT コマンドをプログラムヘッダに対し使用すると, 出力セクション属性を上書きする.

リンカは,普通, セグメントのフラグを,そのセグメントを構成する セクションに基づいて設定する. キーワード FLAGS を使って明示的にセグメントのフラグを 指定することができる. *flags* の値は整数でなければならない. プログラムヘッダの p_flags フィールドを 設定するのに使われる.

以下に PHDRS の例を示す. ネイティブの ELF システムでの プログラムヘッダの典型的な使い方である.

```
PHDRS
{
  headers PT_PHDR PHDRS ;
  interp PT_INTERP ;
  text PT_LOAD FILEHDR PHDRS ;
  data PT_LOAD ;
  dynamic PT_DYNAMIC ;
}

SECTIONS
{
  . = SIZEOF_HEADERS;
  .interp : { *(.interp) } :text :interp
  .text : { *(.text) } :text
  .rodata : { *(.rodata) } /* defaults to :text */
  ...
  . = . + 0x1000; /* move to a new page in memory */
  .data : { *(.data) } :data
  .dynamic : { *(.dynamic) } :data :dynamic
  ...
}
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.9 VERSION コマンド

リンカは ELF を使用する場合,シンボルバージョンをサポートする. シンボルバージョンは共有ライブラリを使う場合にのみ意味がある. 動的リンカはシンボルバージョンを使って,古いバージョンの共有ライブラリに リンクされた可能性のあるプログラムを実行する時に,関数の特定のバージョンを 選択することができる.

バージョンスクリプトは使用しているリンカスクリプトに直接 組み込むこともできるし, 暗黙のリンカスクリプトとして与えることもできる. また,リンカの `--version-script' オプションを使うこともできる.

VERSION コマンドの形式は単純に以下のようになる.

```
VERSION { version-script-commands }
```

バージョンスクリプトコマンドの形式は Solaris 2.5 の Sun のリンカで 使われているものと同じである. バージョンスクリプトはバージョンノードの 木構造を定義する.バージョンスクリプト内でノード名と相互依存関係を指定する. どのシンボルがどのバージョンノードに束縛されるかを指定することができる. また, 指定された組のシンボルを ローカルのスコープに移すことができるので, 共有ライブラリの外側からは グローバルには見えないようになる.

バージョンスクリプト言語を説明するのに一番良いのは例をいくつか お見せすることである.

```
VERS_1.1 {
         global:
                 foo1;
         local:
                 old*;
                 original*;
                 new*;
};

VERS_1.2 {
                 foo2;
} VERS_1.1;

VERS_2.0 {
                 bar1; bar2;
} VERS_1.2;
```

この例のバージョンスクリプトは, 三つのバージョンノードを定義している. 最初に定義されているバージョンノードは,`VERS_1.1' はであり, 他に依存関係はない. このスクリプトは,シンボル `foo1' を, `VERS_1.1' に束縛している. 多数のシンボルをローカルのスコープに移すので, 共有ライブラリの外側からは見えなくなる.

次に, バージョンスクリプトはノード `VERS_1.2' を定義している. このノードは, `VERS_1.1' に依存する. シンボル `foo2' を バージョンノード `VERS_1.2' に束縛する.

最後に, このバージョンスクリプトはノード `VERS_2.0' を定義している. このノードは `VERS_1.2' に依存する.スクリプトは バージョンノード `VERS_2.0' に束縛されるシンボル `bar1' と `bar2' を束縛する.

リンカが,ライブラリ中で定義されているシンボルで, 特にバージョンノードに束縛されていないものを見つけると, 事実上,そのライブラリの未指定のベースバージョンに束縛する. 他の全ての未指定の

シンボルを,指定したバージョンノードに 束縛するには `global: *' をバージョンスクリプトのどこかで使えば良い.

バージョンノードの名前は読む人に示唆を与える以外には 特別な意味を持たない. バージョン `2.0' が, `1.1' と `1.2' の間に現れても良いのである. だが, そんなことをするのは, バージョンスクリプトを書くのを混乱させるだけであろう.

あるアプリケーションをバージョン付きシンボルのある共有ライブラリと リンクする時, アプリケーション自体は各シンボルのどのバージョンを必要と するかを知っており, リンクしようとしている各共有ライブラリが必要とする バージョンノードも知っている. このため, リンクしたライブラリが, アプリケーションが全ての動的シンボルを 解決するのに必要なバージョンノードを全て実際に提供できることを確認する素早い検査を動的ローダが実行時に行なうことができる. こうして, 動的ローダが確実に, 必要な全ての外部シンボルを 各シンボル参照を探すことなく, 解決できることを知ることが可能になるのである.

シンボルのバージョン管理は, 事実上, SunOS が行なっているマイナー バージョンの検査よりももっと洗練された方法である. ここで解決されている基本的な問題は, 普通は外部関数への参照は 必要に応じて束縛され, アプリケーションの起動時には必ずしも全てが 束縛されているわけでないという点である. 共有ライブラリが古いままだと, 必要なインターフェースがないということがある. アプリケーションが そのインターフェースを使おうとすると, 突然予期しない形で失敗に終る. シンボルのバージョン管理を行なっていると, アプリケーションと共に 使うライブラリがあまりにも古いとそのプログラムの起動時に警告を出す.

Sun のバージョン管理方法に対し GNU の拡張がいくつかある. その一つ目は, バージョンスクリプトではなく, シンボルが定義されている ソースファイルのバージョンノードにシンボルを結び付けることができる 点である. これは主にライブラリの保守担当者の負荷を少なくするために行なわれた. これを行なうには, C のソースファイルに

```
__asm__(".symver original_foo,foo@VERS_1.1");
```

のようなコードを置く. こうすると, 関数 `original_foo' を バージョンノード `VERS_1.1' に束縛されている `foo' の 別名に変える. `local:' を付けると, シンボル `original_foo' が輸出されるのを防ぐ.

第二の GNU の拡張は, 同じ関数の複数のバージョンが,ある指定した 共有ライブラリに現れることを可能にすることである. こうすると, インターフェースに互換性のない変更を加えても, 共有ライブラリの メジャーバージョン番号を増やさなくても済み, 一方で, アプリケーションを 古いインターフェースにリンクしても機能し続けるのである.

これを行なうには, ソースファイルで `.symver' 疑似命令を複数回 指定しなければならない. 以下に例を示す.

```
__asm__(".symver original_foo,foo@");
__asm__(".symver old_foo,foo@VERS_1.1");
__asm__(".symver old_foo1,foo@VERS_1.2");
__asm__(".symver new_foo,foo@@VERS_2.0");
```

この例では, `foo@' はシンボルの未指定ベースバージョンに束縛された シンボル `foo' を表す. この例を含むソースファイルでは, C の関数を 4 つ定義する. `original_foo', `old_foo', `old_foo1', `new_foo' である.

指定したシンボルに複数の定義を与える時は, このシンボルへの外部からの 参照に束縛されるデ

フォルトのバージョンを何らかの方法で指定する 必要がある. これを行なうには, `foo@@VERS_2.0' という 形式の `.symver' 疑似命令を使う. この方法では一個のシンボルについては一つのバージョンしか「デフォルト」と宣言できない. そうしないと, 実質的に同じシンボルに複数の定義を 与えることになる.

ある参照を, 共有ライブラリ内のシンボルの特定のバージョンに束縛 したいときは, 便宜的に別名 (つまり `old_foo')を使うか, `.symver' 疑似命令を使って, 問題の関数の外部バージョンに 特別に束縛することができる.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.10 リンカスクリプトの式

リンカスクリプト言語の式の構文は, C の式に同じである. 全ての式は評価すると整数になる.全ての式は同じサイズに評価される. ホストとターゲット双方が 32 ビットであれば,32 ビットになり, さもなければ 64 ビットになる.

式の中でシンボル値を使ったり,設定することができる.

リンカは,式で使える,様々な特殊目的の組み込み関数を定義している.

3.10.1 定数
3.10.2 シンボル名
3.10.3 位置カウンタ
3.10.4 演算子
3.10.5 評価
3.10.6 式のセクション
3.10.7 組み込み関数

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.10.1 定数

全ての定数は整数である.

C と同様,リンカは `0' で始まる整数は 8 進数,`0x' か `0X' で始まる整数は 16 進数とみなす.それ以外の整数は 10 進数とみなす.

さらに, K や M というサフィックスを使って定数をそれぞれ 1024 倍または 1024*1024 倍 することができる. 例えば, 以下の例では全て同じ量を表している.

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.10.2 シンボル名

引用符で囲まない場合は, シンボル名は英文字か下線, ピリオドで始まり, 英文字, 下線, 数字, ピリオド, ハイフンが続くものである. 引用符なしのシンボル名はどのキーワードとも一致してはいけない. 上記以外の文字を含んだり, キーワードと同じ名前のシンボルを 指定するには, そのシンボル名をダブルクォートで囲めば良い.

```
"SECTION" = 9;
"with a space" = "also with a space" + 10;
```

シンボルは非アルファベット文字をたくさん含めることができるので, シンボルは空白で区切るのが最善である. 例えば, `A-B' は 一個のシンボルだが, `A - B' は引き算を含む式である.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.10.3 位置カウンタ

特別なリンカ変数であるドット `.' は常に現在の出力位置のカウンタを 保持している. . は常にある出力セクション内の位置を指しているので, SECTIONS コマンド内の式の中にしか現れることができない. . というシンボルは, 普通のシンボルが式の中で許される位置なら どこに現れても良い.

シンボル . へ値を代入すると位置カウンタが移動する. これを使うと, 出力セクションに隙間を作ることができる. 位置カウンタを戻すことはできない.

```
SECTIONS
{
  output :
    {
      file1(.text)
      . = . + 1000;
      file2(.text)
      . += 1000;
      file3(.text)
    } = 0x1234;
}
```

この例では file1 の `.text' セクションが出力セクション `output' の先頭に置かれる. 次に1000バイトの隙間がおかれる. その後に file2 が現れ, また 1000バイトの隙間が続き, 次に file3 がロードされる. `= 0x1234' という書き方で, 隙間に書き込むべきデータを指定している (see section 3.6.8.5 出力セクション埋め潰し).

注意.. は実際には,現在のオブジェクトの先頭からのバイト単位の オフセットを指す.通常これは SECTIONS 文であり,その開始アドレス は 0 であるので,. を絶対アドレスとして使うことができる. ただし,. をセクション記述の中で使うと,そのセクションの先頭から のバイトオフセットになるので,絶対アドレスにはならない.以下に例を示す.

```
SECTIONS
{
    . = 0x100
    .text: {
      *(.text)
      . = 0x200
```

```
    }
    . = 0x500
    .data: {
      *(.data)
      . += 0x600
    }
}
```

`.text' セクションは,開始アドレスに 0x100 が割り当てられ, 入力 `.text' セクションにそれだけの領域を埋める程のデータが なくても,大きさは 0x200 バイトちょうどになる. (データが多過ぎる時は,これは . を後戻りさせることになるので エラーになる.) `.data' セクションは 0x500 から始まり,入力 `.data' セクション からの値の末尾の後と,出力 `.data' セクション自身の末尾の間に 0x600 バイトの余分の空間が置かれる.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.10.4 演算子

GNU リンカは標準 C の算術演算子を認識する. 結合規則と優先順位も 以下の標準のものである.

優先順位 (高)	結合規則	演算子	注意
1	left	! - ~	(1)
2	left	* / %	
3	left	+ -	
4	left	>> <<	
5	left	== != > < <= >=	
6	left	&	
7	left	|	
8	left	&&	
9	left	||	
10	right	? :	
11	right	&= += -= *= /=	(2)
(低)			

注: (1) 前置演算子 (2) See section 3.5 シンボルへの値の代入.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.10.5 評価

本リンカは式を遅延評価する. 本当に必要になるまで式の値を計算しないのである.

GNU ld はどんなリンクを行なうにしても結局, 先頭のセクションの開始アドレス値, メモリ領域の先頭と長さのような情報が必要である. これらの値はリンカがリンカスクリプトを読み込むと, できるだけすぐに計算される.

だが, それ以外の値(シンボル値など)は格納領域割り当てが行なわれるまで 分からないし, 必要としない. このような値は後で評価される. それは, 他の情報(出力セクションの大きさなど)がシンボルの代入式で 使えるようになるときである.

セクションの大きさは,割り当てるまでわからないので,大きさに依存する 代入は割当が行なわれるまで実行されない.

式の中には,位置カウンタ `.' に依存するもののように,セクション 割り当ての最中にも評価しなければならないものがある.

式の結果が必要だが,その値は利用可能でないという時には,エラーになる. 例えば,

```
SECTIONS
  {
    .text 9+this_isnt_constant :
      { *(.text) }
  }
```

というスクリプトは,`non constant expression for initial address' というエラーメッセージを出す.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.10.6 式のセクション

リンカが式を評価したとき, 結果はセクションに対し絶対か相対かの どちらかになる.相対式はセクションの基底からの固定のオフセットとして 表される.

リンカスクリプト内での式の位置により,それが絶対か相対かが決まる. 出力セクション定義の中に現れる式は,出力セクションの基底に相対である. それ以外の場所に現れる式は絶対である.

相対式に設定されたシンボルは,`-r' オプションを使って 再配置可能な出力を要求した場合は,再配置可能になる.これは,以後の リンク操作で,そのシンボルの値が変わり得るということである. そのシンボルのセクションは,相対式のセクションになる.

絶対式に設定されたシンボルは,以後どんなリンクを行なっても同じ値の ままである.シンボルは絶対になり,特定の関連セクションを持たない.

組み込み関数 ABSOLUTE を使って,本来相対になる式を強制的に 絶対式にすることができる.例えば, ある絶対シンボルを作って, 出力セクション `.data' の末尾のアドレスを設定するには 以下のようにする.

```
SECTIONS
  {
    .data : { *(.data) _edata = ABSOLUTE(.); }
  }
```

ここで `ABSOLUTE' を使わないと, `_edata' は `.data' セクションに相対になる.

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.10.7 組み込み関数

リンカスクリプト言語には,リンカスクリプト式で使える多数の 組み込み関数がある.

ABSOLUTE(*exp*)
: 式 *exp* の絶対値(負にならないという意味ではなく, 再配置不可という 意味である)を返す. 主に, セクション定義の中でシンボルに絶対値を 割り当てるのに使う. セクション定義内では, シンボル値は普通は セクションに相対である. See section 3.10.6 式のセクション.

ADDR(*section*)
: 指定されたセクション *section* の絶対アドレス(VMA)を返す. 事前に,スクリプト中でそのセクションの位置を定義しておかなければ ならない.以下の例では,symbol_1 と symbol_2 は 同じ値が割り当てられる.

```
SECTIONS { ...
  .output1 :
    {
    start_of_output_1 = ABSOLUTE(.);
    ...
    }
  .output :
    {
    symbol_1 = ADDR(.output1);
    symbol_2 = start_of_output_1;
    }
... }
```

ALIGN(*exp*)
: 次の *exp* の境界まで整列させた位置カウンタ(.)を 返す. *exp* は値が 2 の冪乗となる式でなければならない. これは以下に等価である.

```
(. + exp - 1) & ~(exp - 1)
```

ALIGN は位置カウンタの値を変えない. 単にその値を使って 計算を行なうだけである. 以下の例では,出力 .data セクションを 先行するセクションの次の 0x2000 バイト境界に整列させ, .data セクション内の変数 variable を, 入力セクションの後の 次の 0x8000 バイト境界の値に設定している.

```
SECTIONS { ...
  .data ALIGN(0x2000): {
    *(.data)
    variable = ALIGN(0x8000);
  }
... }
```

この例の最初の ALIGN の使い方は, セクションの位置を指定する. セクション定義の, 省略可能な *adress* 属性として使っているからで ある(see section 3.6.3 出力セクションアドレス). ALIGN の二番目の使い方は単にシンボルの値を定義している.

ALIGN に近いコマンドとして, 組み込みの NEXT がある.

BLOCK(*exp*)
: これは ALIGN の別名であり,古いリンカスクリプトとの互換性の ためにある.出力セクションのアドレスを設定する時に一番良く使われる.

DEFINED(*symbol*)
: *symbol* がリンカのグローバルシンボル表に存在し, 定義されていれば 1 を返す. そうでなければ 0 を返す. この関数を使うとシンボルのデフォルト値 を提供することができる. 例えば, 以下のスクリプトの断片は, グローバルシンボル begin に .text セクションの先頭位置を 設定する方法を示している. この場合, begin というシンボルが既に 存在するなら, その値は保存される.

```
SECTIONS { ...
```

```
.text : {
  begin = DEFINED(begin) ? begin : . ;
  ...
}
...
}
```

LOADADDR(*section*)
: *section* というセクションの絶対 LMA を返す. これは普通は ADDR と同じだが, AT 属性を 出力セクション定義で使っている場合は違ってくる可能性がある (see section 3.6.8.2 出力セクション LMA).

MAX(*exp1*, *exp2*)
: *exp1* と *exp2* の最大値を返す.

MIN(*exp1*, *exp2*)
: *exp1* と *exp2* の最小値を返す.

NEXT(*exp*)
: 次の *exp* の倍数となる, 未割当のアドレスを返す. この関数は ALIGN(*exp*) に良く似ている. MEMORY コマンドを使って, 出力ファイルに飛び飛びのメモリ領域を 定義するのでない限り, この二つの関数は同じである.

SIZEOF(*section*)
: *section* で指定されるセクションが割当済みであれば, そのセクションの 大きさをバイト数で返す. これが評価される時にセクションが割り当てられて いなければ,リンカはエラーを報告する. 次の例では, symbol_1 と symbol_2 に割り当てられる値は同じになる.

```
SECTIONS{ ...
  .output {
    .start = . ;
    ...
    .end = . ;
    }
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
... }
```

SIZEOF_HEADERS
sizeof_headers
: 出力ファイルのヘッダの大きさをバイト数で返す. これは出力ファイルの先頭に現れる情報である. 読者が選択するなら, この数を先頭のセクションの開始アドレスとして使うなら, ページングを利用することができる.

  ELF 出力ファイルを生成する時,リンカスクリプトが組み込み関数 SIZEOF_HEADERS を使っていると,リンカは,全セクションのアドレスと 大きさが決まる前にプログラムヘッダの数を計算しなければならない. リンカが後でプログラムヘッダを追加する必要が生じた時には, `not enough room for program headers' というエラーを出すことになる. このエラーを避けるには,関数 SIZEOF_HEADERS を使わないようにするか, リンカスクリプトを書き直して,追加のプログラムヘッダを使うように強制するのをやめるか,自分で PHDRS コマンドを使ってプログラム ヘッダを定義する必要がある (see section 3.8 PHDRS コマンド).

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

3.11 暗黙のリンカスクリプト

リンカがオブジェクトファイルやアーカイブファイルとして 認識できない入力ファイルを与えると,そのファイルをリンカスクリプトとして 読もうとする.そのファイルをリンカスクリプトとしてパースできないときは, リンカはエラーを報告する.

この暗黙のリンカスクリプトは,デフォルトのリンカスクリプトは 置き換えない.

通常,暗黙のリンカスクリプトは,シンボル代入や,`INPUT`, `GROUP`,`VERSOIN` コマンドしか含まない.

暗黙のリンカスクリプトのために読み込まれる入力ファイルは, コマンド行の,暗黙のリンカスクリプトが読まれた位置で読み込まれる. このため,アーカイブの検索に影響する.

[<<] [>>] [Top] [Contents] [Index] [?]

This document was generated by *YABUKI Youichi* on *March, 15 2002* using *texi2html*