ARM® コンパイラ

バージョン 5.04

fromelf ユーザガイド

ARM

ARM DUI0477JJ

ARM® コンパイラ

fromelf ユーザガイド

リリース情報

ドキュメント履歴

発行	日付	機密保持ステータス	変更点
A	28 5 月 2010	非機密扱い	ARM コンパイラ v4.1 リリース
B	30 9 月 2010	非機密扱い	ARM コンパイラ v4.1 のアップデート 1
C	28 1 月 2011	非機密扱い	ARM コンパイラ v4.1 パッチ 3 のアップデート 2
D	30 4 月 2011	非機密扱い	ARM コンパイラ v5.0 リリース
E	29 7 月 2011	非機密扱い	ARM コンパイラ v5.0 のアップデート 1
F	30 9 月 2011	非機密扱い	ARM コンパイラ v5.01 リリース
G	29 2 月 2012	非機密扱い	ARM コンパイラ v5.01 リリースマニュアルの更新 1
H	27 7 月 2012	非機密扱い	ARM コンパイラ v5.02 リリース
I	31 1 月 2013	非機密扱い	ARM コンパイラ v5.03 リリース
J	16 12 月 2013	非機密扱い	ARM コンパイラ v5.04 リリース

著作権

機密保持ステータス

本書は非機密扱いであり、本書を使用、複製、および開示する権利は、ARM および ARM が本書を提供した当事者との間で締結した契約の条項に基づいたライセンスの制限により異なります。

無制限アクセスは、ARM 社内による分類です。

製品ステータス

本書の情報は最終版であり、開発済み製品に対応しています。

Web アドレス

www.arm.com

目次
ARM® コンパイラ fromelf ユーザガイド

表の一覧
ARM® コンパイラ fromelf ユーザガイド

序章

この前書きでは、次について紹介します。*ARM® コンパイラ fromelf ユーザガイド*.

このドキュメントは、次で構成されています。

本書について

ARM コンパイラ fromelf ユーザガイド。このマニュアルでは、fromelf ユーティリティの使用方法について説明します。PDF で提供されています。

本書の構成

本書は以下の章から構成されています。

第 章 1 fromelf イメージ変換ユーティリティの概要

付属の **fromelf** イメージ変換ユーティリティの概要について説明します。ARM® コンパイラ.

第 章 2 fromelf ユーティリティの使用

付属の **fromelf** イメージ変換ユーティリティの使用方法について説明します。ARM コンパイラ.

第 章 3 fromelf コマンドラインオプション

付属の **fromelf** イメージ変換ユーティリティのコマンドラインオプションについて説明します。ARM コンパイラ

第 章 4 via ファイルの構文

fromelf でサポートされている via ファイルの構文について説明します。

付録 A fromelf ドキュメントの改訂

『fromelf イメージ変換ユーティリティユーザガイド』に対して加えられた技術的変更について 説明します。

表記規則

italic
: 重要用語、相互参照、引用箇所を示します。

bold
: メニュー名などのユーザインタフェース要素を太字で記載しています。また、必要に応じて記述リスト内の重要箇所、ARM プロセッサの信号名、重要用語、および専門用語にも太字を使用しています。

`monospace`
: コマンド、ファイル名、プログラム名、ソースコードなど、キーボードから入力可能なテキストを示しています。

monospace
: コマンドまたはオプションに使用可能な略語を示しています。コマンド名またはオプション名をすべて入力する代わりに、下線部分の文字だけを入力することができます。

`monospace italic`
: 引数が特定の値で置き換えられる場合のモノスペーステキストの引数を示しています。

`monospace bold`
: サンプルコード以外に使用される言語キーワードを示しています。

<and>
: コードまたはコードの一部のアセンブラ構文で置換可能な項が使用されている場合に、その項を囲みます。例えば、

```
MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>
```

スモールキャピタル
: 「*ARM 用語集*」で定義されている専門的な意味を持つ用語について、本文中で使用されます。例えば、IMPLEMENTATION DEFINED、IMPLEMENTATION SPECIFIC、UNKNOWN、UNPREDICTABLE などです。

ご意見、ご感想

本製品に関するフィードバック

本製品についてのご意見やご提案がございましたら、以下の情報を添えて購入元までお寄せ下さい。

- 製品名
- 製品のリビジョンまたはバージョン
- 説明にはできるだけ多くの情報を含めて下さい。適宜、症状と診断手順も含めて下さい。

内容に関するフィードバック

内容に関するご意見につきましては、電子メールを *errata@arm.com* まで送信して下さい。その際には、以下の内容を記載して下さい。

- タイトル
- 文書番号（ARM DUI0477JJ）
- 問題のあるページ番号
- 問題点の簡潔な説明

また、補足すべき点や改善すべき点についての全般的なご提案もお待ちしております。

第 1 章
fromelf イメージ変換ユーティリティの概要

付属の **fromelf** イメージ変換ユーティリティの概要について説明します。ARM® コンパイラ.

このドキュメントは、次で構成されています。

1.1 fromelf イメージ変換ユーティリティについて

fromelf イメージ変換ユーティリティを使用することにより、ELF イメージとオブジェクトファイルを編集したり、それらのファイルの情報を表示したりすることができます。

fromelf では、次のことが行えます。

- コンパイラ、アセンブラ、およびリンカにより作成された ARM ELF オブジェクトおよびイメージファイルを処理します。
- **armar** によって作成されたアーカイブ内のすべての ELF ファイルを処理し、処理したファイルを必要に応じて別のアーカイブに出力します。
- ELF イメージを、ROM ツールで使用したり直接メモリにロードしたりできる他の形式に変換します。使用できる形式は以下のとおりです。
  - プレーンバイナリ形式。
  - Motorola 32 ビット S レコード形式。
  - Intel Hex 32 ビット形式。
  - バイト指向（Verilog メモリモデル）16 進形式。
  - ELF。例えば、ELF イメージからデバッグ情報を削除するために、ELF 形式で保存し直すことができます。
- サードパーティに配信されるイメージとオブジェクトの*知的所有権*（IP）を保護します。
- 逆アセンブルの出力やシンボルリストなどの入力ファイルに関する情報を標準出力（**stdout**）やテキストファイルに出力します。

注

デバッグ情報を持たないイメージを生成した場合、**fromelf** ユーティリティには、以下の制限があります。

- イメージを別の形式のファイルに変換できません。
- 分かりやすい逆アセンブルリストを生成できません。

関連概念

2.3 fromelf を使用してイメージファイルに含まれるコードを保護するオプション (2-19 ページ).
2.4 fromelf を使用してオブジェクトファイルに含まれるコードを保護するオプション (2-20 ページ).

関連参照

1.2 fromelf の実行モード (1-13 ページ).
1.4 fromelf のコマンドライン構文 (1-15 ページ).
3 fromelf コマンドラインオプション (3-24 ページ).

1.2 fromelf の実行モード

fromelf は、さまざまな実行モードで実行できます。

fromelf には、以下の実行モードがあります。

- ファイルを ELF 形式で保存し直すための ELF モード(`--elf`)。
- オブジェクトまたはイメージファイルに関する情報を出力するためのテキストモード(`--text` など)。
- 形式変換モード(`--bin`、`--m32`、`--i32`、`--vhx`)。

関連参照

3.2 --bin (3-29 ページ).
3.24 --elf (3-56 ページ).
3.35 --i32 (3-70 ページ).
3.46 --m32 (3-82 ページ).
3.62 --text (3-99 ページ).
3.64 --vhx (3-103 ページ).

1.3 fromelf コマンドのヘルプの取得

主なコマンドラインオプションの一覧を表示するには、--help オプションを使用します。

これは、オプションやファイルを指定しない場合のデフォルトの動作です。

例

ヘルプ情報を表示するには、以下のように入力します。

```
fromelf --help
```

関連参照

1.4 fromelf のコマンドライン構文 (1-15 ページ).
3.32 --help (3-67 ページ).

1.4 fromelf のコマンドライン構文

fromelf コマンドラインでは ELF ファイルまたは ELF ファイルのライブラリを指定できます。

構文

```
fromelf options input_file
```

options
: **fromelf** コマンドラインのオプション。

input_file
: 処理対象の ELF ファイルまたはライブラリファイル。いくつかのオプションを使用すると、複数の入力ファイルを指定できます。

関連参照

3 fromelf コマンドラインオプション (3-24 ページ).
3.41 input_file (3-76 ページ).

第 2 章
fromelf ユーティリティの使用

付属の **fromelf** イメージ変換ユーティリティの使用方法について説明します。ARM コンパイラ.

このドキュメントは、次で構成されています。

2.1 fromelf 使用時の一般的な注意事項

一部の変更は、**fromelf** を使用してイメージに対して行うことができません。

fromelf を使用する場合、以下の制限があります。

- `--base` オプションを使用して Motorola S レコード形式または Intel Hex 形式の出力のベースアドレスを変更する場合を除き、イメージの構造やアドレスを 変更できません。
- 分散ロードされる ELF イメージを、別の形式の非分散ロードのイメージに変換することは できません。構造やアドレスに関する情報は、すべてリンク時にリンカに 渡す必要があります。

関連参照

3.1 --base [[object_file::]load_region_ID=]num (3-27 ページ).
3.41 input_file (3-76 ページ).

2.2 アーカイブ内の ELF ファイルを処理する例

アーカイブ内のすべての ELF ファイルまたはこれらのファイルのサブセットを処理する方法の例。処理されたファイルは、処理されていないファイルと共に別のアーカイブに出力されます。

サンプル

以降の例に、アーカイブ内の ELF ファイルを処理する方法を示します。このアーカイブ `test.a` には、以下のファイルが含まれています。

```
bmw.o bmw1.o call_c_code.o newtst.o shapes.o strmtst.o
```

アーカイブ内のすべてのファイルを処理する例

この例では、アーカイブ内のすべてのファイルから、すべてのデバッグ、コメント、メモ、およびシンボルが削除されます。

```
fromelf --elf --strip=all test.a -o strip_all/
```

この例を実行すると、`test.a` という名前の出力アーカイブが、サブディレクトリ `strip_all` に作成されます。

アーカイブ内のファイルのサブセットを処理する例

アーカイブ内の `shapes.o` および `strmtst.o` ファイルのみから、すべてのデバッグ、コメント、メモ、およびシンボルを削除するには、以下のように入力します。

```
fromelf --elf --strip=all test.a(s*.o) -o subset/
```

この例を実行すると、`test.a` という名前の出力アーカイブが、サブディレクトリ `subset` に作成されます。このアーカイブには、処理されたファイルが処理されていない残りのファイルと共に格納されます。

アーカイブ内の `bmw.o`、`bmw1.o`、および `newtst.o` ファイルを処理するには、以下のように入力します。

```
fromelf --elf --strip=all test.a(??w*) -o subset/
```

アーカイブ内のファイルの逆アセンブルされたバージョンを表示する例

アーカイブ内の `call_c_code.o` の逆アセンブルされたバージョンを表示するには、以下のように入力します。

```
fromelf --disassemble test.a(c*)
```

関連参照

3.22 --disassemble (3-54 ページ).
3.24 --elf (3-56 ページ).
3.41 input_file (3-76 ページ).
3.49 --output=destination (3-85 ページ).
3.60 --strip=option[,option,...] (3-96 ページ).

2.3 fromelf を使用してイメージファイルに含まれるコードを保護するオプション

イメージをサードパーティに配布するとき、そこに含まれるコードを保護した方がよい場合があります。

fromelf には、コードを保護するための `--strip` オプションと `--privacy` オプションが用意されています。これらのオプションを使用すると、イメージ内のシンボル名を削除したりあいまいにしたりできます。どちらのオプションを選択するかは、削除する情報の量次第です。これらのオプションの効果は、イメージファイルによって異なります。

制約条件

これらのオプションは `--elf` と組み合わせて使用する必要があります。`--elf` を使用する必要があるため、`--output` も使用する必要があります。

イメージファイルのコードを保護するオプションの効果

イメージファイルの場合:

表 2-1 イメージファイルに対する fromelf の --privacy オプションと--strip オプションの効果

オプション	ローカルシンボル	セクション名	マッピングシンボル	ビルド属性
`fromelf --elf --privacy`	シンボルテーブル全体が削除されます。 セクション名 `.comment` を削除します。 `fromelf --text` の出力結果には、これが `[Anonymous Section]` としてマークされます。 セクション名にデフォルト値が付けられます。例えば、コードセクション名は `'.text'` に変わります。			
`fromelf --elf --strip=symbols`	シンボルテーブル全体が削除されます。 セクション名は変更されません。			
`fromelf --elf --strip=localsymbols`	削除されます。	そのまま残ります。	削除されます。	そのまま残ります。

例

シンボルテーブル全体が削除され、各セクション名が変更された新しい ELF 実行可能イメージを生成するには、以下のように入力します。

```
fromelf --elf --privacy --output=outfile.axf infile.axf
```

関連概念

2.4 fromelf を使用してオブジェクトファイルに含まれるコードを保護するオプション (2-20 ページ).

関連参照

1.4 fromelf のコマンドライン構文 (1-15 ページ).
3.24 --elf (3-56 ページ).
3.49 --output=destination (3-85 ページ).
3.50 --privacy (3-86 ページ).
3.60 --strip=option[,option,...] (3-96 ページ).

2.4 fromelf を使用してオブジェクトファイルに含まれるコードを保護するオプション

オブジェクトをサードパーティに配布するとき、そこに含まれるコードを保護した方がよい場合があります。

fromelf には、コードを保護するための `--strip` オプションと `--privacy` オプションが用意されています。これらのオプションを使用すると、オブジェクト内のシンボル名を削除したりあいまいにしたりできます。どちらのオプションを選択するかは、削除する情報の量次第です。これらのオプションの効果は、オブジェクトファイルによって異なります。

制約条件

これらのオプションは `--elf` と組み合わせて使用する必要があります。`--elf` を使用する必要があるため、`--output` も使用する必要があります。

オブジェクトファイルのコードを保護するオプションの効果

オブジェクトファイルの場合:

表 2-2 オブジェクトファイルに対する fromelf の --privacy オプションと--strip オプションの効果

オプション	ローカルシンボル	セクション名	マッピングシンボル	ビルド属性
`fromelf --elf --privacy`	機能を損なうことなく削除できるローカルシンボルは削除されます。 再配置のターゲットなど、削除できないシンボルは維持されます。このようなシンボルについては、名前が削除されます。 `fromelf --text` の出力結果には、これらが `[Anonymous Symbol]` としてマークされます。	セクション名にデフォルト値が付けられます。例えば、コードセクション名は `'.text'` に変わります。	そのまま残ります。	そのまま残ります。
`fromelf --elf --strip=symbols`	機能を損なうことなく削除できるローカルシンボルは削除されます。 再配置のターゲットなど、削除できないシンボルは維持されます。このようなシンボルについては、名前が削除されます。 `fromelf --text` の出力結果には、これらが `[Anonymous Symbol]` としてマークされます。	セクション名は変更されません。	そのまま残ります。	そのまま残ります。
`fromelf --elf --strip=localsymbols`	機能を損なうことなく削除できるローカルシンボルは削除されます。 再配置のターゲットなど、削除できないシンボルは維持されます。このようなシンボルについては、名前が削除されます。 `fromelf --text` の出力結果には、これらが `[Anonymous Symbol]` としてマークされます。	セクション名は変更されません。	そのまま残ります。	そのまま残ります。

例

シンボルテーブル全体が削除され、各セクション名が変更された新しい ELF オブジェクトを生成するには、以下のように入力します。

```
fromelf --elf --privacy --output=outfile.o infile.o
```

関連概念

2.3 fromelf を使用してイメージファイルに含まれるコードを保護するオプション (2-19 ページ).

関連参照

1.4 fromelf のコマンドライン構文 (1-15 ページ).
3.24 --elf (3-56 ページ).
3.49 --output=destination (3-85 ページ).
3.50 --privacy (3-86 ページ).
3.60 --strip=option[,option,...] (3-96 ページ).

2.5 ELF ファイルに固有の詳細を出力するオプション

`--emit` オプションを使用すると、テキスト出力に含める ELF オブジェクトの要素を指定できます。

出力には、ELF ヘッダおよびセクション情報が含まれます。各要素はコンマで区切って指定できます。

注

`--emit` オプションの一部は、`--text` オプションを使用して指定できます。

サンプル

ELF ファイル `infile.axf` のデータセクションの内容を出力するには、以下のように入力します。

```
fromelf --emit=data infile.axf
```

ELF ファイル `infile2.axf` の再配置情報およびダイナミックセクションの内容を出力するには、以下のように入力します。

```
fromelf --emit=relocation_tables,dynamic_segment infile2.axf
```

関連参照

1.4 fromelf のコマンドライン構文 (1-15 ページ).
3.25 --emit=option[,option,...] (3-57 ページ).
3.62 --text (3-99 ページ).

2.6 実行可能な ELF イメージ内のシンボルの場所を fromelf を使用して調べる方法

実行可能な ELF イメージ内のシンボルの場所を調べることができます。

ELF イメージファイル内のシンボルの場所を調べるには、`--text -s -v` オプションを使用して、各セグメントとセクションヘッダのシンボルテーブルおよび詳細情報を表示します。以下に例を示します。

シンボルテーブルを見ると、シンボルが配置されているセクションを確認できます。

例

以下の手順に従います。

1. 以下のソースコードを含む `s.c` という名前のファイルを作成します。

```
long long altstack[10] __attribute__ ((section ("STACK"), zero_init)); int main()
{     return sizeof(altstack); }
```

2. ソースをコンパイルします。

   `-c s.c -o s.o`

3. オブジェクト `s.o` をリンクし、STACK シンボルは残します。

   `armlink --keep=s.o(STACK) s.o --output=s.axf`

4. fromelf コマンドを実行して、各セグメントおよびセクションヘッダのシンボルテーブルと詳細情報を表示します。

   `fromelf --text -s -v s.o`

5. STACK シンボルと `altstack` シンボルを `fromelf` の出力結果で探します。以下に例を示します。

```
...** Section #9     Name        :.symtab     Type        :SHT_SYMTAB
(0x00000002)     Flags       :None (0x00000000)     Addr        :0x00000000
File Offset :2792 (0xae8)     Size        :2896 bytes (0xb50)
Link        :Section 10 (.strtab)     Info        :Last local symbol no = 115
Alignment   :4     Entry Size  :16    Symbol table .symtab (180 symbols, 115
local)       #  Symbol Name               Value      Bind  Sec  Type  Vis  Size
============================================================================ ...16
STACK                     0x00008228   Lc    2  Sect  De   0x50 ...179
altstack                  0x00008228   Gb    2  Data  Hi   0x50 ...
```

   スタックが配置されているセクションが、`Sec` 列に表示されています。この例では、セクション 2 です。

6. 確認したシンボルのセクションを `fromelf` の出力結果から探します。以下に例を示します。

```
...=================================== ** Section #2     Name        :ER_ZI
Type        :SHT_NOBITS (0x00000008)     Flags       :SHF_ALLOC + SHF_WRITE
(0x00000003)     Addr        :0x000081c8     File Offset :508 (0x1fc)
Size        :176 bytes (0xb0)     Link        :SHN_UNDEF     Info        :0
Alignment   :8     Entry Size  :0 ================================== ...
```

   以上の結果から、シンボルは ZI 実行領域に存在することがわかります。

関連参照

3.62 --text (3-99 ページ).

第 3 章
fromelf コマンドラインオプション

付属の **fromelf** イメージ変換ユーティリティのコマンドラインオプションについて説明します。
ARM コンパイラ

このドキュメントは、次で構成されています。

3.1 --base [[object_file::]load_region_ID=]num

Motorola S レコード形式または Intel Hex ファイル形式の 1 つ以上のロード領域に指定されたベースアドレスを変更できます。

構文

`--base [[` *object_file* `::]` *load_region_ID* `=]` *num*

各項目には以下の意味があります。

object_file
オプションの ELF 入力ファイル。

load_region_ID
オプションのロード領域。これには、ロード領域に属する実行領域のシンボル名か、最初の領域を示す場合には `#0` などのゼロベースのロード領域番号のいずれかを指定できます。

num
10 進数値または 16 進数値。

以下のことができます。

- ワイルドカード文字 `?` および `*` を *object_file* 引数および *load_region_ID* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

All addresses encoded in the output file start at the base address *num* . If you do not specify a `--base` option, the base address is taken from the load region address.

制約条件

このオプションは、`--i32`、`--i32combined`、`--m32`、または `--m32combined` のいずれかの出力形式と組み合わせて使用する必要があります。したがって、オブジェクトファイルに対してはこのオプションを使用できません。

例

以下の表に例を示します。

表 3-1 --base の使用例

`--base 0`	10 進数値
`--base 0x8000`	16 進数値
`--base #0=0`	最初のロード領域のベースアドレス
`--base foo.o::*=0`	`foo.o` のすべてのロード領域の ベースアドレス
`--base #0=0,#1=0x8000`	最初および 2 番目のロード領域のベースアドレス

関連概念

2.1 fromelf 使用時の一般的な注意事項 (2-17 ページ).

関連参照

3.35 --i32 (3-70 ページ).
3.36 --i32combined (3-71 ページ).

3.46 --m32 (3-82 ページ).
3.47 --m32combined (3-83 ページ).

3.2 --bin

各ロード領域に対して 1 つのプレーンバイナリ形式の出力ファイルを作成します。`--widthxbanks` オプションを使用して、このオプションによって生成される出力を 複数ファイルに分割できます。

制約条件

以下の使用制限があります。

- オブジェクトファイルに対してはこのオプションを使用できません。
- このオプションは `--output` と組み合わせて使用する必要があります。

Considerations when using --bin

複数のロード領域を含む ELF イメージをバイナリ形式に変換すると、**fromelf** によって *destination* という名前の出力ディレクトリが作成され、入力イメージ内のロード領域ごとに 1 つのバイナリ出力ファイルが生成されます。出力ファイルは **fromelf** によって *destination* ディレクトリに配置されます。

——— 注 ———

複数のロード領域の場合、対応するロード領域内の最初の空でない実行領域の名前がファイル名に使用されます。

ファイルは、ELF ファイルに含まれているコードまたはデータをロード領域が記述している場合にのみ作成されます。例えば、ZI データのある実行領域のみを含んでいるロード領域から出力ファイルが生成されることはありません。

———————————

例

ELF ファイルをプレーンバイナリファイル（`outfile.bin` など）に変換するには、以下のように入力します。

```
fromelf --bin --output=outfile.bin infile.axf
```

関連参照

3.49 --output=destination (3-85 ページ).
3.68 --widthxbanks (3-107 ページ).

3.3 --bincombined

プレーンバイナリ形式の出力を作成します。複数のロード領域を含むイメージ用に 1 つの出力ファイルが生成されます。

使用法

デフォルトでは、メモリ内の最初のロード領域の開始アドレスがベースアドレスとして使用されます。fromelf ユーティリティにより、お互いに対して正しい相対オフセットになるように、必要に応じてロード領域間にパディングが挿入されます。この方法でロード領域を分けることにより、メモリに出力ファイルをロードし、ベースアドレスから始まるように正しく整列することができます。

ベースアドレスとパディングのデフォルト値を変更するには、このオプションを --bincombined_base および --bincombined_padding と組み合わせて使用します。

制約条件

以下の使用制限があります。

- オブジェクトファイルに対してはこのオプションを使用できません。
- このオプションは --output と組み合わせて使用する必要があります。

Considerations when using --bincombined

ベースアドレスのデフォルト値を変更するには、このオプションを --bincombined_base と組み合わせて使用します。

パディングのデフォルト値は 0xFF です。パディングのデフォルト値を変更するには、このオプションを --bincombined_padding と組み合わせて使用します。

大きなアドレス空間を挟んで存在する 2 つのロード領域を定義するスキャッタファイルを使用した場合、パディングが大部分を占めることになって、最終的なバイナリが非常に大きくなる場合があります。例えば、 0x00000000 というアドレスにサイズ 0x100 バイトのロード領域があり、0x30000000 というアドレスに別のロード領域がある場合、パディングのサイズは 0x2FFFFF00 バイトとなります。

ロード領域の間のスペースが大きい場合は、--bin などの他の方法を使用し、複数の出力ファイルを正しいアドレスにロードできるように処理することを推奨します。

サンプル

開始アドレス 0x1000 にロードできるバイナリファイルを生成するには、以下のように入力します。

```
fromelf --bincombined --bincombined_base=0x1000 --output=out.bin in.axf
```

プレーンバイナリ形式の出力を生成し、32 ビットのワード 0x12345678 のコピーでロード領域の間のスペースを埋めるには、以下のように入力します。

```
fromelf --bincombined --bincombined_padding=4,0x12345678 --output=out.bin in.axf
```

関連参照

3.4 --bincombined_base=address (3-32 ページ).
3.5 --bincombined_padding=size,num (3-33 ページ).
3.49 --output=destination (3-85 ページ).
3.68 --widthxbanks (3-107 ページ).

関連情報

入力セクション、出力セクション、領域、およびプログラムセグメント.

3.4 --bincombined_base=address

--bincombined 出力モードで使用するベースアドレスを下げることができます。生成される出力ファイルは、指定されたアドレスから始まるメモリにロードできます。

構文

--bincombined_base= *address*

address はイメージをロードする開始アドレスです。

- 指定されたアドレスが最初のロード領域の始めより下位にある場合は、**fromelf** ユーティリティにより、出力ファイルの始めにパディングが追加されます。
- 指定されたアドレスが最初のロード領域の始めより上位にある場合は、**fromelf** ユーティリティによりエラーが出力されます。

デフォルト

デフォルトでは、メモリ内の最初のロード領域の開始アドレスがベースアドレスとして使用されます。

制約条件

このオプションは --bincombined と組み合わせて使用する必要があります。If you omit --bincombined を省略した場合は、警告メッセージが表示されます。

例

```
--bincombined --bincombined_base=0x1000
```

関連参照

3.3 --bincombined (3-30 ページ).

3.5 --bincombined_padding=size,num (3-33 ページ).

関連情報

入力セクション、出力セクション、領域、およびプログラムセグメント.

3.5 --bincombined_padding=size,num

--bincombined 出力モードで使用されるパディング値に、デフォルトとは異なる値を指定できます。

構文

--bincombined_padding= *size,num*

各項目には以下の意味があります。

size
バイト、ハーフワード、またはワードを指定するための 1 バイト、2 バイト、または 4 バイト。

num
パディングに使用する値。指定されたサイズに収まりきらない大きな値を指定すると、警告メッセージが表示されます。

——— 注 ———

fromelf ユーティリティは、入力ファイルの適切なエンディアンに 2 バイトおよび 4 バイトのパディング値が指定されていると想定します。例えば、ビッグエンディアンの ELF ファイルをバイナリに変換する場合は、指定されたパディング値がビッグエンディアンのワードまたはハーフワードとして扱われます。

———————

デフォルト

デフォルトは --bincombined_padding=1,0xFF です。

制約条件

このオプションは --bincombined と組み合わせて使用する必要があります。If you omit --bincombined を省略した場合は、警告メッセージが表示されます。

例

以下に、--bincombined_padding の使用例を示します。

--bincombined --bincombined_padding=4,0x12345678
この例では、プレーンバイナリ形式の出力が作成され、ロード領域の間のスペースが 32 ビットのワード 0x12345678 のコピーで埋められます。

--bincombined --bincombined_padding=2,0x1234
この例では、プレーンバイナリ形式の出力が作成され、ロード領域の間のスペースが 16 ビットのハーフワード 0x1234 のコピーで埋められます。

--bincombined --bincombined_padding=2,0x01
ビッグエンディアンメモリに対してこの例を指定すると、ロード領域の間のスペースが 0x0100 で埋められます。

関連参照

3.3 --bincombined (3-30 ページ).
3.4 --bincombined_base=address (3-32 ページ).

3.6 --cad

バイナリ出力を含む C 配列定義または C++ 配列定義を作成します。

使用法

別のアプリケーションのソースコードには、それぞれの配列定義を使用できます。例えば、組み込みオペレーティングシステムなど、別のアプリケーションのアドレス空間にはイメージを組み込むことができます。

イメージ内のロード領域が 1 つの場合、デフォルトで出力が標準出力（stdout）に送られます。ファイルに出力を保存するには、--output オプションをファイル名と共に使用します。

イメージに複数のロード領域がある場合は、--output オプションをディレクトリ名と共に使用する必要があります。完全パス名を指定しない限り、パスは現在のディレクトリに対する相対パスになります。指定したディレクトリに、各ロード領域用のファイルが作成されます。各ファイルの名前は、対応する実行領域の名前です。

イメージ内のロード領域ごとに 1 つの出力ファイルを生成するには、このオプションを --output と組み合わせて使用します。

制約条件

オブジェクトファイルに対してはこのオプションを使用できません。

Considerations when using --cad

ファイルは、ELF ファイルに含まれているコードまたはデータをロード領域が記述している場合にのみ作成されます。例えば、ZI データのある実行領域のみを含んでいるロード領域から出力ファイルが生成されることはありません。

例

以下に、--cad の使用例を示します。

- 1 つのロード領域を持つイメージの配列定義を作成するには、以下のように入力します。

```
fromelf --cad myimage.axf unsigned char LR0[] = {     0x00,0x00,0x00,0xEB,
0x28,0x00,0x00,0xEB,0x2C,0x00,0x8F,0xE2,0x00,0x0C,0x90,0xE8,     0x00,0xA0,0x8A,
0xE0,0x00,0xB0,0x8B,0xE0,0x01,0x70,0x4A,0xE2,0x0B,0x00,0x5A,0xE1,
0x00,0x00,0x00,0x1A,0x20,0x00,0x00,0xEB,0x0F,0x00,0xBA,0xE8,0x18,0xE0,0x4F,
0xE2,     0x01,0x00,0x13,0xE3,0x03,0xF0,0x47,0x10,0x03,0xF0,0xA0,0xE1,0xAC,
0x18,0x00,0x00,     0xBC,
0x18,0x00,0x00,0x00,0x30,0xB0,0xE3,0x00,0x40,0xB0,0xE3,0x00,0x50,0xB0,0xE3,
0x00,0x60,0xB0,0xE3,0x10,0x20,0x52,0xE2,0x78,0x00,0xA1,0x28,0xFC,0xFF,0xFF,
0x8A,     0x82,0x2E,0xB0,0xE1,0x30,0x00,0xA1,0x28,0x00,0x30,0x81,0x45,0x0E,
0xF0,0xA0,0xE1,     0x70,0x00,0x51,0xE3,0x66,0x00,0x00,0x0A,
0x64,0x00,0x51,0xE3,0x38,0x00,0x00,0x0A,     0x00,0x00,0xB0,0xE3,0x0E,
0xF0,0xA0,0xE1,0x1F,0x40,0x2D,0xE9,0x00,0x00,0xA0,0xE1, ...0x3A,
0x74,0x74,0x00,0x43,0x6F,0x6E,0x73,0x74,0x72,0x75,0x63,0x74,0x65,0x64,0x20,
0x41,0x20,0x23,0x25,0x64,0x20,0x61,0x74,0x20,0x25,0x70,0x0A,
0x00,0x00,0x00,0x00,     0x44,0x65,0x73,0x74,0x72,0x6F,
0x79,0x65,0x64,0x20,0x41,0x20,0x23,0x25,0x64,0x20,
0x61,0x74,0x20,0x25,0x70,0x0A,0x00,0x00,0x0C,0x99,0x00,0x00,0x0C,
0x99,0x00,0x00,
0x50,0x01,0x00,0x00,0x44,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
  0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 };
```

- 複数のロード領域を持つイメージには、以下のコマンドを使用すると、ディレクトリ *root*\myprojects\multiload\load_regions にロード領域ごとのファイルが作成されます。

```
cd root\myprojects\multiloadfromelf --cad image_multiload.axf --output
load_regions
```

image_multiload.axf に実行領域 EXEC_ROM および RAM が含まれている場合、EXEC_ROM ファイルおよび RAM ファイルが load_regions サブディレクトリに作成されます。

関連参照

3.7 --cadcombined (3-36 ページ).
3.49 --output=destination (3-85 ページ).

関連情報

入力セクション、出力セクション、領域、およびプログラムセグメント.

3.7 --cadcombined

バイナリ出力を含む C 配列定義または C++ 配列定義を作成します。

使用法

別のアプリケーションのソースコードには、それぞれの配列定義を使用できます。例えば、組み込みオペレーティングシステムなど、別のアプリケーションのアドレス空間にはイメージを組み込むことができます。

デフォルトでは、出力が標準出力（`stdout`）に送られます。ファイルに出力を保存するには、`--output` オプションをファイル名と共に使用します。

制約条件

オブジェクトファイルに対してはこのオプションを使用できません。

例

The following commands create the file `load_regions.c` in the directory *`root`*`\myprojects\multiload`:

```
cd root\myprojects\multiloadfromelf --cadcombined image_multiload.axf --output
load_regions.c
```

関連参照

3.6 --cad (3-34 ページ).
3.49 --output=destination (3-85 ページ).

3.8 --compare=option[,option,…]

2 つの入力ファイルを比較し、その違いをテキスト形式のリストに出力します。

使用法

入力ファイルは、2 つの ELF ファイルまたは 2 つのライブラリファイルの、同じ種類とする必要があります。ライブラリファイルはメンバ単位で比較され、その違いが出力で連結して示されます。

2 つの入力ファイルのすべての違いは、`--relax_section` オプションを使用して警告に格下げされている場合を除いて、エラーとして報告されます。

構文

`--compare=` *option* [`,` *option* `,…`]

option には以下のいずれかを指定できます。

section_sizes
ライブラリファイルの各 ELF ファイルまたは各 ELF メンバのすべてのセクションのサイズを比較します。

`section_sizes::`*object_name*
名前が *object_name* と一致する ELF オブジェクト内のすべてのセクションのサイズを比較します。

`section_sizes::`*section_name*
名前が *section_name* と一致するすべてのセクションのサイズを比較します。

セクション
ライブラリファイルの各 ELF ファイルまたは各 ELF メンバのすべてのセクションのサイズおよび内容を比較します。

`sections::`*object_name*
名前が *object_name* と一致する ELF オブジェクト内のすべてのセクションのサイズおよび内容を比較します。

`sections::`*section_name*
名前が *section_name* と一致するすべてのセクションのサイズおよび内容を比較します。

function_sizes
ライブラリファイルの各 ELF ファイルまたは各 ELF メンバのすべての関数のサイズを比較します。

`function_sizes::`*object_name*
名前が *object_name* と一致する ELF オブジェクト内のすべての関数のサイズを比較します。

`function_size::`*function_name*
名前が *function_name* と一致するすべての関数のサイズを比較します。

global_function_sizes
ライブラリファイルの各 ELF ファイルまたは各 ELF メンバのすべてのグローバル関数のサイズを比較します。

`global_function_sizes::`*function_name*
名前が *function_name* と一致する ELF オブジェクト内のすべてのグローバル関数のサイズを比較します。

以下のことができます。

- ワイルドカード文字 **?** および ***** を *section_name* 、 *function_name* 、および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

関連参照

3.37 --ignore_section=option[,option,...] (3-72 ページ).
3.38 --ignore_symbol=option[,option,...] (3-73 ページ).
3.52 --relax_section=option[,option,...] (3-88 ページ).
3.53 --relax_symbol=option[,option,...] (3-89 ページ).

3.9 --continue_on_error

すべてのエラーを報告した上で実行を続行します。

使用法

このオプションの代わりに `--diag_warning=error` を使用して下さい。

関連参照

3.21 --diag_warning=tag[,tag,...] (3-53 ページ).

3.10 --cpu=list

`--cpu=name` オプションでサポートされている アーキテクチャとプロセッサ名を一覧表示します。

構文

```
--cpu=list
```

関連参照

3.11 --cpu=name (3-41 ページ).

3.11 --cpu=name

指定されたプロセッサまたはアーキテクチャによる解釈と同じように逆アセンブルされるように、-c、--disassemble などのオプションでマシンコードを逆アセンブルする方法を変更します。

構文

--cpu= *name*

name はプロセッサまたはアーキテクチャの名前です。

- *name* がプロセッサ名の場合は、ARM7TDMI、ARM1176JZ-S、MPCore など、ARM データシートに記載されているとおりに名前を入力します。
- *name* がアーキテクチャ名の場合は、名前が以下の表に示すアーキテクチャのリストに含まれている必要があります。

プロセッサ名とアーキテクチャ名では、大文字と小文字は区別されません。

ワイルドカード文字は使用できません。

表 3-2 サポートされている ARM アーキテクチャ

アーキテクチャ	説明	プロセッサの例
4	Thumb をサポートしていない ARMv4	SA-1100
4T	Thumb をサポートしている ARMv4	ARM7TDMI、ARM9TDMI、ARM720T、ARM740T、ARM920T、ARM922T、ARM940T、SC100
5T	ARMv5 with Thumb and interworking	-
5TE	ARMv5 with Thumb, interworking, DSP multiply, and double-word instructions	ARM9E, ARM946E-S, ARM966E-S
5TEJ	ARMv5 with Thumb, interworking, DSP multiply, double-word instructions, and Jazelle® extensions ——— 注 ——— **fromelf** cannot generate Java bytecodes. ———	ARM926EJ-S, ARM1026EJ-S, SC200
6	Thumb、インターワーク、DSP 乗算、ダブルワード命令、非境界整列、混合エンディアン、Jazelle 拡張、およびメディア拡張をサポートしている ARMv6	ARM1136J-S、ARM1136JF-S
6-M	Thumb のみ、およびプロセッサステート命令をサポートしている ARMv6 マイクロコントローラプロファイル	Cortex-M1（OS 拡張なし）、Cortex-M0、SC000、Cortex-M0plus
6S-M	Thumb のみと、プロセッサステート命令および OS 拡張をサポートしている ARMv6 マイクロコントローラプロファイル	Cortex-M1（OS 拡張あり）
6K	SMP 拡張をサポートしている ARMv6	MPCore
6T2	Thumb をサポートしている ARMv6（Thumb-2 テクノロジ）	ARM1156T2-S、ARM1156T2F-S

表 3-2 サポートされている ARM アーキテクチャ（続き）

アーキテクチャ	説明	プロセッサの例
6Z	Security Extensions をサポートしている ARMv6	ARM1176JZF-S、ARM1176JZ-S
7	Thumb（Thumb-2 テクノロジ）のみをサポートし、ハードウェア除算をサポートしていない ARMv7	-
7-A	仮想 MMU ベースメモリシステムをサポートする、ARM、Thumb（Thumb-2 テクノロジ）および ThumbEE をサポートし、DSP サポート、および 32 ビット SIMD サポートが指定された ARMv7 アプリケーションプロファイル	Cortex-A5、Cortex-A7、Cortex-A8、Cortex-A9、Cortex-A15
7-A.security	v7-A アーキテクチャ向けにアセンブルする場合に SMC 命令（以前の SMI）の使用を許可する	Cortex-A5、Cortex-A7、Cortex-A8、Cortex-A9、Cortex-A15
7-R	ARM、Thumb（Thumb-2 テクノロジ）、DSP サポート、および 32 ビット SIMD サポートが指定された ARMv7 リアルタイムプロファイル	Cortex-R4、Cortex-R4F、Cortex-R7
7-M	Thumb（Thumb-2 テクノロジ）のみとハードウェア除算をサポートしている ARMv7 マイクロコントローラプロファイル	Cortex-M3、SC300
7E-M	DSP（サチュレートおよび 32 ビット SIMD）命令による ARMv7-M 拡張	Cortex-M4

——— 注 ———

- ARMv7 は実際の ARM アーキテクチャではありません。--cpu=7 は、ARMv7-A、ARMv7-R、および ARMv7-M のアーキテクチャに共通の機能を示します。つまり、--cpu=7 とともに使用される機能が ARMv7-A、ARMv7-R、および ARMv7-M のアーキテクチャ上に存在します。
- 7-A.security は実際の ARM アーキテクチャではなく、7-A にセキュリティ拡張機能を追加したものを表します。

———————————

Usage

以下に、プロセッサとアーキテクチャに関するオプションの一般的な特徴を示します。

プロセッサ

- プロセッサを選択すると、適切なアーキテクチャ、*浮動小数点ユニット*（FPU）、およびメモリ構成が選択されます。
- --cpu の値には、現時点でのすべての ARM 製品名またはアーキテクチャバージョンを指定できます。

  Marvell Feroceon や Marvell XScale など、他の ARM アーキテクチャベースのプロセッサもサポートされています。

Architectures

- --cpu オプションでアーキテクチャ名を指定すると、そのアーキテクチャに対し、-c や --disassemble などのオプションによってマシンコードが逆アセンブルされます。--disassemble を指定すると、そのアーキテクチャをサポートしているすべてのプロセッサに対して逆アセンブリがアセンブルされます。例えば、--cpu=5TE --disassemble と指定すると、ARM926EJ-S® プロセッサに対してアセンブルされる逆アセンブリが生成されます。

FPU

- `--cpu` を指定すると、`--fpu` が暗黙的に選択されることがあります。

  ——— 注 ———
  コマンドラインで `--fpu` によって明示的に指定された FPU は、オーバーライドされることに注意して下さい。
  ———————

- `--fpu` オプションも `--cpu` オプションも指定されていない場合には、`--fpu=softvfp` が使用されます。

デフォルト

`--cpu` オプションを指定しない場合、**fromelf** はマシン命令をアーキテクチャに依存しない方法で逆アセンブルします。つまり、**fromelf** は、アーキテクチャの命令として認識されるものをすべて逆アセンブルします。

すべてのアーキテクチャおよびプロセッサが記載された一覧を表示するには、`cpu=list` オプションを使用します。

制約条件

同じコマンドラインでプロセッサとアーキテクチャの両方を指定することはできません。

例

To select the disassembly for the Cortex™-A8 processor, use:

```
--cpu=Cortex-A8
```

関連参照

3.10 --cpu=list (3-40 ページ).
3.15 --device=list (3-47 ページ).
3.16 --device=name (3-48 ページ).
3.22 --disassemble (3-54 ページ).
3.40 --info=topic[,topic,...] (3-75 ページ).
3.62 --text (3-99 ページ).

3.12 --datasymbols

シンボル定義をインターリーブするようにデータセクションの出力情報を変更します。

使用法

このオプションは、`--text -d` と共に指定する必要があります。

関連参照

3.62 --text (3-99 ページ).

3.13 --debugonly

コードセクションまたはデータセクションの内容を削除します。

使用法

このオプションにより、出力ファイルにデバッグに必要な情報（デバッグセクション、シンボルテーブル、ストリングテーブルなど）のみが含まれるようになります。セクションヘッダは、シンボルのターゲットとして機能する必要があるため保持されます。

制約条件

このオプションは `--elf` と組み合わせて使用する必要があります。

例

ELF ファイル `debugout.axf` を、デバッグ情報のみを含む ELF ファイル `infile.axf` から作成するには、以下のように入力します。

```
fromelf --elf --debugonly --output=debugout.axf infile.axf
```

関連参照

3.24 --elf (3-56 ページ).

3.14 --decode_build_attributes

標準ビルド属性の場合は人間が読める形式でビルド属性セクションの内容を出力し、非標準ビルド属性の場合は未加工の 16 進形式で出力します。

——— 注 ———

標準ビルド属性については、*『Application Binary Interface for the ARM Architecture』*を参照して下さい。

制約条件

このオプションは、テキストモードでのみ使用できます。

例

以下に、`--decode_build_attributes` の出力例を示します。

```
** Section #12 '.ARM.attributes' (SHT_ARM_ATTRIBUTES)     Size   :69 bytes     'aeabi'
file build attributes:0x000000:05 41 52 4d 37 54 44 4d 49 00 06 02 08 01 11
01    .ARM7TDMI.......0x000010:12 02 14 02 17 01 18 01 19 01 1a 01 1e 03 20
02    ..............0x000020:41 52 4d 00
ARM.Tag_CPU_name = "ARM7TDMI"         Tag_CPU_arch = ARM v4T (=2)
Tag_ARM_ISA_use = ARM instructions were permitted to be used (=1)
Tag_ABI_PCS_GOT_use = Data are imported directly (=1)         Tag_ABI_PCS_wchar_t =
Size of wchar_t is 2 (=2)         Tag_ABI_FP_denormal = This code was permitted to
require that the sign of a flushed-to-zero number be preserved in the sign of 0
(=2)         Tag_ABI_FP_number_model = This code was permitted to use only IEEE 754
format FP numbers (=1)         Tag_ABI_align8_needed = Code was permitted to depend on
the 8-byte alignment of 8-byte data items (=1)         Tag_ABI_align8_preserved = Code
was required to preserve 8-byte alignment of 8-byte data objects (=1)
Tag_ABI_enum_size = Enum values occupy the smallest container big enough to hold all
values (=1)         Tag_ABI_optimization_goals = Optimized for small size, but speed
and debugging illusion preserved (=3)         Tag_compatibility = 2, "ARM"     'ARM'
file build attributes:0x000000:04 01 12 01                                        ....
```

関連参照

3.23 --dump_build_attributes (3-55 ページ).
3.25 --emit=option[,option,...] (3-57 ページ).
3.27 --extract_build_attributes (3-60 ページ).

関連情報

『Application Binary Interface for the ARM Architecture』.

3.15 --device=list

`--device=name` オプションと一緒に使用可能なサポートされているデバイス名を一覧表示します。

——— 注 ———————
このオプションの使用は廃止される予定です。
———————————

関連参照

3.16 --device=name (3-48 ページ).

3.16 --device=name

特定のマイクロコントローラまたは SoC（System-on-Chip）デバイスを選択します。

構文

`--device=` *name*

name は特定のデバイス名です。

使用法

このオプションは、**fromelf** が入力ファイルで見つけた命令を解釈する方法に影響します。形式は、コンパイラでサポートされている形式と同じです。

各デバイスには、CPU および*浮動小数点ユニット*（FPU）のデフォルト値があります。しかし、`--device` オプションの後に `--fpu` オプションを指定することにより、コマンドラインから FPU をオーバーライドできます。

CPU および FPU の実装の詳細については、デバイスのマニュアルを参照して下さい。

使用可能なデバイスの完全なリストを表示するには、`--device=list` オプションを使用します。

——— **注** ———————
このオプションの使用は廃止される予定です。
———————————

関連参照

3.10 --cpu=list (3-40 ページ).
3.11 --cpu=name (3-41 ページ).
3.15 --device=list (3-47 ページ).
3.29 --fpu=list (3-63 ページ).
3.30 --fpu=name (3-64 ページ).

3.17 --diag_error=tag[,tag,…]

特定のタグがある診断メッセージにエラーの重大度を設定します。

構文

```
--diag_error= tag[,tag,…]
```

tag には以下のいずれかを指定できます。

- エラーの重大度を設定する診断メッセージ番号
- `warning`（すべての警告をエラーとして扱う場合）

関連参照

3.18 --diag_remark=tag[,tag,...] (3-50 ページ).
3.19 --diag_style={arm|ide|gnu} (3-51 ページ).
3.20 --diag_suppress=tag[,tag,...] (3-52 ページ).
3.21 --diag_warning=tag[,tag,...] (3-53 ページ).

3.18 --diag_remark=tag[,tag,…]

特定のタグがある診断メッセージに注釈の重要度を設定します。

構文

```
--diag_remark= tag[,tag,…]
```

tag は診断メッセージ番号のコンマ区切りのリストです。

関連参照

3.17 --diag_error=tag[,tag,...] (3-49 ページ).
3.19 --diag_style={arm|ide|gnu} (3-51 ページ).
3.20 --diag_suppress=tag[,tag,...] (3-52 ページ).
3.21 --diag_warning=tag[,tag,...] (3-53 ページ).

3.19 --diag_style={arm|ide|gnu}

診断メッセージの表示スタイルを指定します。

構文

`--diag_style=` *`string`*

`string` には以下のいずれかを指定できます。

`arm`
ARM コンパイラの形式を使用してメッセージを表示します。

`ide`
エラーのある行の行番号と文字数を表示します。これらの値は括弧に囲まれて表示されます。

`gnu`
`gcc` で使用される形式でメッセージを表示します。

使用法

`--diag_style=gnu` は、GNU コンパイラが報告する形式 `gcc` と一致します。

`--diag_style=ide` は、Microsoft Visual Studio が報告する形式と一致します。

デフォルト

デフォルトは `--diag_style=arm` です。

関連参照

3.17 --diag_error=tag[,tag,...] (3-49 ページ).
3.18 --diag_remark=tag[,tag,...] (3-50 ページ).
3.20 --diag_suppress=tag[,tag,...] (3-52 ページ).
3.21 --diag_warning=tag[,tag,...] (3-53 ページ).

3.20 --diag_suppress=tag[,tag,…]

特定のタグがある診断メッセージを非表示にします。

構文

```
--diag_suppress= tag[,tag,…]
```

tag には以下のいずれかを指定できます。

- 非表示にする診断メッセージ番号
- `error`（降格できるすべてのエラーを非表示にする場合）
- `warning`（すべての警告を非表示にする場合）

関連参照

3.17 --diag_error=tag[,tag,...] (3-49 ページ).
3.18 --diag_remark=tag[,tag,...] (3-50 ページ).
3.19 --diag_style={arm|ide|gnu} (3-51 ページ).
3.21 --diag_warning=tag[,tag,...] (3-53 ページ).

3.21 --diag_warning=tag[,tag,…]

特定のタグがある診断メッセージに警告の重大度を設定します。

構文

```
--diag_warning= tag[,tag,…]
```

tag には以下のいずれかを指定できます。

- 警告の重大度を設定する診断メッセージ番号
- `error`（警告に降格できるすべてのエラーを設定する場合）

関連参照

3.17 --diag_error=tag[,tag,...] (3-49 ページ).
3.18 --diag_remark=tag[,tag,...] (3-50 ページ).
3.19 --diag_style={arm|ide|gnu} (3-51 ページ).
3.21 --diag_warning=tag[,tag,...] (3-53 ページ).

3.22 --disassemble

逆アセンブルされたイメージを標準出力（stdout）に表示します。

使用法

このオプションを --output *destination* と組み合わせて使用した場合、armasm により出力ファイルを再アセンブルできます。

このオプションを使用すると、ELF イメージファイルまたは ELF オブジェクトファイルを逆アセンブルできます。

注

この出力は、--emit=code および --text -c の出力とは異なります。

例

ARM1176JZF-S™ プロセッサ用の ELF ファイル infile.axf を逆アセンブルしてソースファイル outfile.asm を作成するには、以下のように入力します。

```
fromelf --cpu=ARM1176JZF-S --disassemble --output=outfile.asm infile.axf
```

関連参照

3.11 --cpu=name (3-41 ページ).
3.25 --emit=option[,option,...] (3-57 ページ).
3.42 --interleave=option (3-78 ページ).
3.49 --output=destination (3-85 ページ).
3.62 --text (3-99 ページ).

3.23 --dump_build_attributes

ビルド属性セクションの内容を未加工の 16 進形式で出力します。

制約条件

このオプションは、テキストモードでのみ使用できます。

例

`--dump_build_attributes` の出力例を以下に示します。

```
...** Section #12 '.ARM.attributes' (SHT_ARM_ATTRIBUTES)     Size   :69 bytes
0x000000:41 33 00 00 00 61 65 61 62 69 00 01 29 00 00 00    A3...aeabi..)...
0x000010:05 41 52 4d 37 54 44 4d 49 00 06 02 08 01 11 01    .ARM7TDMI.......
0x000020:12 02 14 02 17 01 18 01 19 01 1a 01 1e 03 20 02    ..............0x000030:41
52 4d 00 11 00 00 00 41 52 4d 00 01 09 00 00    ARM.....ARM.....0x000040:00 04 01 12
01                                          .....
```

関連参照

3.14 --decode_build_attributes (3-46 ページ).
3.25 --emit=option[,option,...] (3-57 ページ).
3.27 --extract_build_attributes (3-60 ページ).
3.62 --text (3-99 ページ).

3.24 --elf

ELF 出力モードを選択します。

使用法

ELF イメージからデバッグ情報を削除するには、`--strip=debug,symbols` と組み合わせて使用します。

制約条件

このオプションは `--output` と組み合わせて使用する必要があります。

関連参照

3.39 --in_place (3-74 ページ).
3.49 --output=destination (3-85 ページ).
3.60 --strip=option[,option,...] (3-96 ページ).

3.25 --emit=option[,option,…]

テキスト出力に表示する ELF オブジェクトの要素を指定できます。出力には、ELF ヘッダおよびセクション情報が含まれます。

制約条件

このオプションは、テキストモードでのみ使用できます。

構文

`--emit=` *option* `[,` *option* `,…]`

option には以下のいずれかを指定できます。

addresses
グローバルデータアドレスと静的データアドレス(構造体と共用体の内容のアドレスも含む)を出力します。これは、`--text -a` と同じ効果があります。

このオプションは、デバッグ情報を含むファイルに対してのみ使用できます。デバッグ情報が含まれない場合、警告メッセージが生成されます。

データアドレスのサブセットを出力する場合は、`--select` オプションを使用します。

構造体内外で展開された配列のデータアドレスを参照するには、このテキストカテゴリと共に `--expandarrays` オプションを使用します。

build_attributes
標準ビルド属性の場合は人間が読める形式でビルド属性セクションの内容を出力し、非標準ビルド属性の場合は未加工の 16 進形式で出力します。生成される出力は、`--decode_build_attributes` オプションの出力と同じです。

code
逆アセンブル対象の元のバイナリデータのダンプおよび命令のアドレスと共に、コードを逆アセンブルします。これは、`--text -c` と同じ効果があります。

——— 注 ———
`--disassemble` とは異なり、逆アセンブリをアセンブラに入力することはできません。
———————

データ
データセクションの内容を出力します。これは、`--text -d` と同じ効果があります。

data_symbols
シンボル定義をインターリーブするようにデータセクションの出力情報を変更します。

debug_info
デバッグ情報を出力します。これは、`--text -g` と同じ効果があります。

dynamic_segment
ダイナミックセグメントの内容を出力します。これは、`--text -y` と同じ効果があります。

exception_tables
オブジェクトの例外テーブル情報をデコードします。これは、`--text -e` と同じ効果があります。

frame_directives
オブジェクトモジュールに組み込まれたデバッグ情報に指定されているように、逆アセンブルされたコードに `FRAME` ディレクティブの内容を出力します。

このオプションは `--disassemble` と組み合わせて使用します。

got
グローバルオフセットテーブル(GOT)オブジェクトの内容を出力します。

heading_comments
: .comment セクションのツール情報およびコマンドライン情報を含む、逆アセンブリの冒頭にある見出しコメントを出力します。

  このオプションは --disassemble と組み合わせて使用します。

raw_build_attributes
: 未加工の 16 進形式、つまりデータと同じ形式でビルド属性セクションの内容を出力します。

relocation_tables
: 再配置情報を出力します。これは、--text -r と同じ効果があります。

string_tables
: ストリングテーブルを出力します。これは、--text -t と同じ効果があります。

summary
: ファイルのセグメントおよびセクションの概要を出力します。fromelf --text のデフォルト出力です。ただし、概要は一部の --info オプションにより非表示になります。必要に応じて --emit summary を使用して明示的に概要を再度有効にします。

symbol_annotations
: それぞれのプロパティ情報を含むコメントの注釈を付けて、逆アセンブルされるコードおよびデータのシンボルを出力します。

  このオプションは --disassemble と組み合わせて使用します。

symbol_tables
: シンボルテーブルとバージョン管理テーブルを出力します。これは、--text -s と同じ効果があります。

vfe
: 使用されていない仮想関数の情報を出力します。

whole_segments
: リンクビューがある場合でも、逆アセンブルされた実行可能ファイルまたは共有ライブラリをセグメントごとに出力します。

  このオプションは --disassemble と組み合わせて使用します。

1 つの *オプション* にコンマ区切りの引数リストを続けることで、オプションを複数指定できます。

関連参照

3.22 --disassemble (3-54 ページ).
3.14 --decode_build_attributes (3-46 ページ).
3.26 --expandarrays (3-59 ページ).
3.62 --text (3-99 ページ).

3.26 --expandarrays

構造体内外で展開された配列を含むデータアドレスを出力します。

制約条件

このオプションは、`--text -a` と共に指定する必要があります。

関連参照

3.62 --text (3-99 ページ).

3.27 --extract_build_attributes

ビルド属性のみを属性の型に依存した形式で出力します。

使用法

ビルド属性を次の形式で出力します。

- 標準ビルド属性の場合は人間が読める形式。
- 非標準ビルド属性の場合は未加工の 16 進形式。

制約条件

このオプションは、テキストモードでのみ使用できます。

例

`--extract_build_attributes` の出力例を以下に示します。

```
============================================================================ ** Object/
Image Build Attributes     'aeabi' file build attributes:0x000000:05 41 52 4d 37 54 44
4d 49 00 06 02 08 01 11 01    .ARM7TDMI.......0x000010:12 02 14 02 17 01 18 01 19 01
1a 01 1e 03 20 02    ..............0x000020:41 52 4d
00                                      ARM.Tag_CPU_name = "ARM7TDMI"
Tag_CPU_arch = ARM v4T (=2)         Tag_ARM_ISA_use = ARM instructions were permitted
to be used (=1)         Tag_ABI_PCS_GOT_use = Data are imported directly (=1)
Tag_ABI_PCS_wchar_t = Size of wchar_t is 2 (=2)         Tag_ABI_FP_denormal = This
code was permitted to require that the sign of a flushed-to-zero number be preserved
in the sign of 0 (=2)         Tag_ABI_FP_number_model = This code was permitted to use
only IEEE 754 format FP numbers (=1)         Tag_ABI_align8_needed = Code was
permitted to depend on the 8-byte alignment of 8-byte data items (=1)
Tag_ABI_align8_preserved = Code was required to preserve 8-byte alignment of 8-byte
data objects (=1)         Tag_ABI_enum_size = Enum values occupy the smallest
container big enough to hold all values (=1)         Tag_ABI_optimization_goals =
Optimized for small size, but speed and debugging illusion preserved (=3)
Tag_compatibility = 2, "ARM"     'ARM' file build attributes:0x000000:04 01 12
01                                      ....
```

関連参照

3.14 --decode_build_attributes (3-46 ページ).
3.23 --dump_build_attributes (3-55 ページ).
3.25 --emit=option[,option,...] (3-57 ページ).
3.62 --text (3-99 ページ).

3.28 --fieldoffsets

アセンブリ言語の EQU ディレクティブのリストを出力します。このディレクティブによって、C++ クラスや C 構造体のフィールド名は、そのクラスまたは構造体のベースからのオフセットを表すようになります。

使用法

入力 ELF ファイルは、再配置可能なオブジェクトまたはイメージです。

--output を使用すると、出力がファイルに転送されます。**armasm** で INCLUDE コマンドを使用すると、生成されたファイルをロードし、C++ クラスおよび C 構造体メンバにアセンブリ言語から名前でアクセスできます。

このオプションを指定すると、構造体に関するすべての情報が出力されます。構造体のサブセットを出力するには、--select *select_options* を使用します。

armasm で入力ファイルとして指定できるファイルを生成する必要がない場合は、--text -a オプションを使用して、表示されるアドレスを読みやすい形式にすることができます。-a オプションを指定すると、アドレスは再配置可能オブジェクト内には存在しないため、イメージ内の構造体と静的データのアドレス情報のみが出力されます。

制約条件

このオプションを使用すると、以下のようになります。

- ソースファイルにデバッグ情報がない場合は使用できません。
- テキストモードでのみ使用できます。

例

以下に、--fieldoffsets の使用例を示します。

- inputfile.o ファイル内にあるすべての構造体のすべてのフィールドオフセットが含まれた一覧を stdout に出力するには、以下のように入力します。

```
fromelf --fieldoffsets inputfile.o
```

- inputfile.o ファイル内で、名前が p で始まる構造体のすべてのフィールドオフセットが含まれた一覧を outputfile.s に出力するには、以下のように入力します。

```
fromelf --fieldoffsets --select=p* --output=outputfile.s inputfile.o
```

- inputfile.o ファイル内で、tools または moretools という名前の構造体のすべてのフィールドオフセットが含まれた一覧を outputfile.s に出力するには、以下のように入力します。

```
fromelf --fieldoffsets --select=tools.*,moretools.* --output=outputfile.s
inputfile.o
```

- inputfile.o ファイル内の構造体 tools の構造体フィールド top の中にあり、名前が number で始まる構造体フィールドのすべてのフィールドオフセットが含まれた一覧を outputfile.s に出力するには、以下のように入力します。

```
fromelf --fieldoffsets --select=tools.top.number* --output=outputfile.s
inputfile.o
```

以下に、出力の例を示します。

```
; Structure, Table , Size 0x104 bytes, from inputfile.cpp |
Table.TableSize|                          EQU    0        ;  int |
Table.Data|                               EQU    0x4      ;  array[64] of
MyClassHandle ; End of Structure Table ; Structure, Box2 , Size 0x8 bytes, from
inputfile.cpp |Box2.|                                    EQU    0        ;  anonymous |
Box2..|                                   EQU    0        ;  anonymous |
Box2...Min|                               EQU    0        ;  Point2 |
Box2...Min.x|                             EQU    0        ;  short |
Box2...Min.y|                             EQU    0x2      ;  short |
Box2...Max|                               EQU    0x4      ;  Point2 |
Box2...Max.x|                             EQU    0x4      ;  short |
Box2...Max.y|                             EQU    0x6      ;  short ; Warning:duplicate
name (Box2..) present in (inputfile.cpp) and in (inputfile.cpp) ; please use the --
qualify option |Box2..|                                  EQU    0        ;  anonymous |
Box2...Left|                              EQU    0        ;  unsigned short |
Box2...Top|                               EQU    0x2      ;  unsigned short |
Box2...Right|                             EQU    0x4      ;  unsigned short |
Box2...Bottom|                            EQU    0x6      ;  unsigned short ; End of
Structure Box2 ; Structure, MyClassHandle , Size 0x4 bytes, from inputfile.cpp |
MyClassHandle.Handle|                     EQU    0        ;  pointer to MyClass ; End of
Structure MyClassHandle ; Structure, Point2 , Size 0x4 bytes, from defects.cpp |
Point2.x|                                 EQU    0        ;  short |
Point2.y|                                 EQU    0x2      ;  short ; End of Structure
Point2 ; Structure, __fpos_t_struct , Size 0x10 bytes, from C:\Program Files\DS-5\bin
\..\include\stdio.h |__fpos_t_struct.__pos|                 EQU    0        ;
unsigned long long |__fpos_t_struct.__mbstate|             EQU    0x8      ;
anonymous |__fpos_t_struct.__mbstate.__state1|     EQU    0x8      ;  unsigned int |
__fpos_t_struct.__mbstate.__state2|     EQU    0xc      ;  unsigned int ; End of
Structure __fpos_t_struct     END
```

関連参照

3.51 --qualify (3-87 ページ).

3.55 --select=select_options (3-91 ページ).

3.62 --text (3-99 ページ).

関連情報

EQU.

GET、INCLUDE.

3.29 --fpu=list

--fpu=name オプションでサポートされている FPU アーキテクチャを一覧表示します。

非推奨オプションは表示されません。

関連参照

3.30 --fpu=name (3-64 ページ).

3.30 --fpu=name

ターゲットの FPU アーキテクチャを指定します。

FPU アーキテクチャがすべて記載された一覧を表示するには、`--fpu=list` オプションを使用します。

構文

`--fpu=` *name*

name には以下のいずれかを指定できます。

なし
浮動小数点オプションが使用されないことを示します。このオプションを指定すると、浮動小数点コードは使用できません。

vfpv2
アーキテクチャ VFPv2 に適合する、ハードウェアの浮動小数点ユニットを選択します。

vfpv3
アーキテクチャ VFPv3 に適合する、ハードウェアのベクタ浮動小数点ユニットを選択します。VFPv3 は、浮動小数点の例外をトラップできないことを除いては、VFPv2 と下位互換性があります。

vfpv3_fp16
半精度拡張機能も備えたアーキテクチャ VFPv3 に適合する、ハードウェアのベクタ浮動小数点ユニットを選択します。

vfpv3_d16
アーキテクチャ VFPv3-D16 に適合する、ハードウェアのベクタ浮動小数点ユニットを選択します。

vfpv3_d16_fp16
半精度拡張機能も備えたアーキテクチャ VFPv3-D16 に適合する、ハードウェアのベクタ浮動小数点ユニットを選択します。

vfpv4
VFPv4 アーキテクチャに適合するハードウェア浮動小数点ユニットを選択します。

vfpv4_d16
アーキテクチャ VFPv4-D16 に適合するハードウェア浮動小数点ユニットを選択します。

fpv4-sp
アーキテクチャ FPv4 の単精度バリアントに適合するハードウェア浮動小数点ユニットを選択します。

softvfp
浮動小数点演算が浮動小数点ライブラリ `fplib` によって実行されるソフトウェア浮動小数点サポートを選択します。`--fpu` オプションが指定されていない場合、または FPU を備えていない CPU を選択した場合は、これがデフォルトになります。

softvfp+vfpv2
VFPv2 に適合するハードウェア浮動小数点ユニットとソフトウェア浮動小数点リンケージを選択します。VFP ユニットを実装するシステムで ARM コードを Thumb コードとインターワークさせる場合は、このオプションを選択します。

softvfp+vfpv3
VFPv3 に適合するハードウェアのベクタ浮動小数点ユニットとソフトウェア浮動小数点リンケージを選択します。VFPv3 ユニットを実装するシステムで ARM コードを Thumb コードとインターワークさせる場合は、このオプションを選択します。

softvfp+vfpv3_fp16
VFPv3-fp16 に適合するハードウェアのベクタ浮動小数点ユニットとソフトウェア浮動小数点リンケージを選択します。

softvfp+vfpv3_d16
VFPv3-D16 に適合するハードウェアのベクタ浮動小数点ユニットとソフトウェア浮動小数点リンケージを選択します。

softvfp+vfpv3_d16_fp16
VFPv3-D16-fp16 に適合するハードウェアのベクタ浮動小数点ユニットとソフトウェア浮動小数点リンケージを選択します。

softvfp+vfpv4
FPv4 に適合するハードウェア浮動小数点ユニットとソフトウェア浮動小数点リンケージを選択します。

softvfp+vfpv4_d16
VFPv4-D16 に適合するハードウェア浮動小数点ユニットとソフトウェア浮動小数点リンケージを選択します。

softvfp+fpv4-sp
FPv4-SP に適合するハードウェア浮動小数点ユニットとソフトウェア浮動小数点リンケージを選択します。

使用法

このオプションは、特定の FPU アーキテクチャの逆アセンブリを選択します。これにより、fromelf が入力ファイルで見つかった命令を処理する方法に影響が出ます。

このオプションを指定した場合、コマンドラインの暗黙的な FPU オプション（`--cpu` オプションなどを指定した場合など）がオーバーライドされます。

`--fpu` オプションを使用して明示的に選択された FPU は、`--cpu` オプションを使用して暗黙的に選択された FPU を常にオーバーライドします。例えば、オプション `--disassemble --cpu=ARM1136JF-S --fpu=softvfp` は、CPU の選択によってアーキテクチャ VFPv2 の使用が暗黙的に示されている場合でも、ソフトウェア浮動小数点ライブラリ `fplib` を使用するコードを逆アセンブルします。

制約条件

`softvfp` では NEON サポートは無効です。

デフォルト

デフォルトのターゲット FPU アーキテクチャは、使用された `--cpu` オプションに基づいて決定されます。

`--cpu` で指定した CPU に VFP コプロセッサがある場合、デフォルトのターゲット FPU アーキテクチャは、その CPU の VFP アーキテクチャになります。例えば、オプション `--cpu ARM1136JF-S` を選択すると、オプション `--fpu vfpv2` が暗黙的に選択されます。VFP コプロセッサが存在する場合は、VFP 命令が生成されます。

関連参照

3.15 --device=list (3-47 ページ).
3.16 --device=name (3-48 ページ).
3.22 --disassemble (3-54 ページ).
3.29 --fpu=list (3-63 ページ).
3.40 --info=topic[,topic,...] (3-75 ページ).
3.62 --text (3-99 ページ).

3.31 --globalize=option[,option,…]

選択されたシンボルをグローバルシンボルに変換します。

構文

`--globalize=` *option* `[,` *option* `,…]`

option には以下のいずれかを指定できます。

object_name`::`
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルが、グローバルシンボルに変換されます。

object_name`::`*symbol_name*
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルと、シンボル名が *symbol_name* と一致するすべてのシンボルは、グローバルシンボルに変換されます。

symbol_name
: シンボル名が *symbol_name* と一致するすべてのシンボルは、グローバルシンボルに変換されます。

以下のことができます。

- ワイルドカード文字 `?` および `*` を *symbol_name* および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

Restrictions

このオプションは `--elf` と組み合わせて使用する必要があります。

関連参照

3.24 --elf (3-56 ページ).
3.33 --hide=option[,option,...] (3-68 ページ).

3.32 --help

主なコマンドラインオプションの一覧を表示します。

デフォルト

これは、オプションやソースファイルなしで **fromelf** を指定する場合のデフォルトです。

関連参照

3.58 --show_cmdline (3-94 ページ).
3.63 --version_number (3-102 ページ).
3.66 --vsn (3-105 ページ).

3.33 --hide=option[,option,…]

シンボルの可視性プロパティを変更して、選択したシンボルを非表示としてマークします。

構文

```
--hide= option [, option ,…]
```

option には以下のいずれかを指定できます。

object_name::
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボル。

object_name::*symbol_name*
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルと、シンボル名が *symbol_name* と一致する ELF オブジェクト内のすべてのシンボル。

symbol_name
: シンボル名が *symbol_name* と一致するすべてのシンボル。

以下のことができます。

- ワイルドカード文字 `?` および `*` を *symbol_name* および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

Restrictions

このオプションは `--elf` と組み合わせて使用する必要があります。

関連参照

3.24 --elf (3-56 ページ).
3.56 --show=option[,option,...] (3-92 ページ).

3.34 --hide_and_localize=option[,option,…]

シンボルの可視性プロパティを変更して、選択したシンボルを非表示としてマークし、選択したシンボルをローカルシンボルに変換します。

構文

`--hide_and_localize=` *option* `[,` *option* `,…]`

option には以下のいずれかを指定できます。

object_name`::`
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルが非表示としてマークされ、ローカルシンボルに変換されます。

object_name`::`*symbol_name*
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルと、シンボル名が *symbol_name* と一致する ELF オブジェクト内のすべてのシンボルが非表示としてマークされ、ローカルシンボルに変換されます。

symbol_name
: シンボル名が *symbol_name* と一致する ELF オブジェクト内のすべてのシンボルが非表示としてマークされ、ローカルシンボルに変換されます。

以下のことができます。

- ワイルドカード文字 `?` および `*` を *symbol_name* および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

Restrictions

このオプションは `--elf` と組み合わせて使用する必要があります。

関連参照

3.24 --elf (3-56 ページ).

3.35 --i32

Intel Hex32 ビット形式の出力を作成します。これによりイメージ内のロード領域ごとに 1 つの出力ファイルを生成できます。

この出力のベースアドレスは、`--base` オプションを使用して指定できます。

制約条件

以下の使用制限があります。

- オブジェクトファイルに対してはこのオプションを使用できません。
- このオプションは `--output` と組み合わせて使用する必要があります。

Considerations when using --i32

複数のロード領域を含む ELF イメージをバイナリ形式に変換すると、**fromelf** によって *destination* という名前の出力ディレクトリが作成され、入力イメージ内のロード領域ごとに 1 つのバイナリ出力ファイルが生成されます。出力ファイルは **fromelf** によって *destination* ディレクトリに配置されます。

——— 注 ———

複数のロード領域の場合、対応するロード領域内の最初の空でない実行領域の名前がファイル名に使用されます。

ファイルは、ELF ファイルに含まれているコードまたはデータをロード領域が記述している場合にのみ作成されます。例えば、ZI データのある実行領域のみを含んでいるロード領域から出力ファイルが生成されることはありません。

———————————

例

ELF ファイル `infile.axf` を Intel Hex-32 形式のファイル（`outfile.bin` など）に変換するには、以下のように入力します。

```
fromelf --i32 --output=outfile.bin infile.axf
```

関連参照

3.1 --base [[object_file::]load_region_ID=]num (3-27 ページ).
3.36 --i32combined (3-71 ページ).
3.49 --output=destination (3-85 ページ).

3.36 --i32combined

Intel Hex32 ビット形式の出力を作成します。複数のロード領域を含むイメージ用に 1 つの出力ファイルが生成されます。

この出力のベースアドレスは、`--base` オプションを使用して指定できます。

Restrictions

以下の使用制限があります。

- オブジェクトファイルに対してはこのオプションを使用できません。
- このオプションは `--output` と組み合わせて使用する必要があります。

Considerations when using --i32combined

If you convert an ELF image containing multiple load regions to a binary format, **`fromelf`** creates an output directory named ***`destination`*** and generates one binary output file for all load regions in the input image. **`fromelf`** places the output file in the ***`destination`*** directory.

ELF イメージは、複数のロード領域を定義しているスキャッタファイルを使用してビルドされた場合などに、複数のロード領域を保持します。

例

1 つの出力ファイル `outfile2.bin` を、2 つのロード領域があり、開始アドレスが `0x1000` のイメージファイル `infile2.axf` から作成するには、以下のコマンドを入力します。

```
fromelf --i32combined --base=0x1000 --output=outfile2.bin infile2.axf
```

関連参照

3.1 --base [[object_file::]load_region_ID=]num (3-27 ページ).
3.35 --i32 (3-70 ページ).
3.49 --output=destination (3-85 ページ).

3.37 --ignore_section=option[,option,…]

比較時に無視するセクションを指定します。これらのセクションに比較する入力ファイル間の違いが含まれる場合はそれが無視されます。

構文

`--ignore_section=` *option* `[,` *option* `,…]`

option には以下のいずれかを指定できます。

object_name`::`
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのセクション。

object_name`::`*section_name*
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのセクションと、セクション名が *section_name* と一致するすべてのセクション。

section_name
: 名前が *section_name* と一致するすべてのセクション。

以下のことができます。

- ワイルドカード文字 `?` および `*` を *symbol_name* および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

Restrictions

このオプションは `--compare` と組み合わせて使用する必要があります。

関連参照

3.8 --compare=option[,option,...] (3-37 ページ).
3.38 --ignore_symbol=option[,option,...] (3-73 ページ).
3.52 --relax_section=option[,option,...] (3-88 ページ).

3.38 --ignore_symbol=option[,option,…]

比較中に無視するシンボルを指定します。比較する入力ファイル間でのこれらのシンボルに関連する違いが無視されます。

構文

`--ignore_symbol=` *option* `[,` *option* `,…]`

option には以下のいずれかを指定できます。

object_name`::`
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボル。

object_name`::`*symbol_name*
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルと、シンボル名が *symbol_name* と一致するすべてのシンボル。

symbol_name
: 名前が *symbol_name* と一致するすべてのシンボル。

以下のことができます。

- ワイルドカード文字 `?` および `*` を *symbol_name* および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

Restrictions

このオプションは `--compare` と組み合わせて使用する必要があります。

関連参照

3.8 --compare=option[,option,...] (3-37 ページ).
3.37 --ignore_section=option[,option,...] (3-72 ページ).
3.53 --relax_symbol=option[,option,...] (3-89 ページ).

3.39 --in_place

入力ファイルの ELF メンバを変換して以前の内容を上書きできます。

制約条件

このオプションは `--elf` と組み合わせて使用する必要があります。

例

ライブラリファイル `test.a` のメンバからデバッグ情報を削除するには、以下のように入力します。

```
fromelf --elf --in_place --strip=debug test.a
```

関連参照

3.24 --elf (3-56 ページ).
3.60 --strip=option[,option,...] (3-96 ページ).

3.40 --info=topic[,topic,…]

特定のトピックに関する情報を出力します。

構文

`--info=` *topic* `[,` *topic* `,…]`

topic は、以下のトピックキーワードからのコンマ区切りのリストです。

instruction_usage
各入力ファイルのコードセクションで定義された ARM 命令と Thumb 命令を分類して一覧表示します。

function_sizes
1 つ以上の入力ファイルで定義されたグローバル関数の名前と、そのサイズ（バイト単位）および分類（ARM 関数か Thumb 関数か）を一覧表示します。

function_sizes_all
1 つ以上の入力ファイルで定義されたローカル関数およびグローバル関数の名前と、そのサイズ（バイト単位）および分類（ARM 関数か Thumb 関数か）を一覧表示します。

sizes
イメージ内の入力オブジェクトおよびライブラリのメンバごとに、Code、RO Data、RW Data、ZI Data、および Debug のサイズが一覧表示されます。このオプションを使用すると、`--info=sizes,totals` を指定したことになります。

totals
入力オブジェクトとライブラリの Code、RO Data、RW Data、ZI Data、および Debug サイズの合計が一覧表示されます。

——— **注** ———
コード関連のサイズには、実行専用コードのサイズも含まれます。
———————————

`--info=sizes,totals` の出力には、常に入力オブジェクトとライブラリの合計にパディング値が含まれます。

——— **注** ———
リスト内のトピックキーワードの間にはスペースを挿入しないで下さい。例えば、「`--info=sizes,totals`」と入力することはできますが、「`--info=sizes, totals`」と入力することはできません。
———————————

制約条件

このオプションは、テキストモードでのみ使用できます。

関連参照

3.62 --text (3-99 ページ).

3.41 input_file

処理する ELF ファイルまたは ELF ファイルを含むアーカイブを指定します。

使用法

以下の場合は、複数の入力ファイルを指定できます。

- `--text` 形式を出力する。
- `--compare` オプションを使用する。
- `--elf` を `--in_place` と組み合わせて使用する。
- `--output` を使用して出力ディレクトリを指定する。

input_file が複数のロード領域を含む分散ロードイメージであり、その出力形式に `--bin`、`--cad`、`--m32`、`--i32`、または `--vhx` のいずれかが指定されている場合、`fromelf` によって各ロード領域用に個別のファイルが生成されます。

input_file が複数のロード領域を含む分散ロードイメージであり、その出力形式に `--cadcombined`、`--m32combined`、または `--i32combined` が指定されている場合、`fromelf` によって、すべてのロード領域を含んだ 1 つのファイルが生成されます。

input_file がアーカイブの場合は、アーカイブ内のすべてのファイルまたはファイルのサブセットを処理できます。アーカイブ内のファイルのサブセットを処理するには、以下に示すようにアーカイブ名に続けてフィルタを指定します。

```
archive.a(filter_pattern)
```

filter_pattern には、メンバファイルを指定します。ファイルのサブセットを指定する場合、以下のワイルドカード文字を使用できます。

*****
: 0 文字以上の文字と一致する。

?
: 任意の 1 文字と一致する。

——— 注 ———————

Unix システムの一般的なシェルでは、かっこを使用し、バックスラッシュでこれらの文字をエスケープする必要があります。あるいは、以下に示すように、アーカイブ名とフィルタを単一引用符で囲みます。

```
'archive.a(??str*)'
```

———————————

アーカイブ内の処理されていないファイルは、処理されたファイルと共に出力アーカイブに格納されます。

例

アーカイブ内の s で始まるすべてのファイルからデバッグ情報を削除した後、新しいアーカイブ `my_archive.a` を作成して、処理されたファイルと処理されていないファイルを格納するには、以下のように入力します。

```
fromelf --elf --strip=debug archive.a(s*.o) --output=my_archive.a
```

関連概念

2.2 アーカイブ内の ELF ファイルを処理する例 (2-18 ページ).

関連参照

3.2 --bin (3-29 ページ).
3.6 --cad (3-34 ページ).
3.7 --cadcombined (3-36 ページ).
3.8 --compare=option[,option,...] (3-37 ページ).
3.24 --elf (3-56 ページ).
3.35 --i32 (3-70 ページ).
3.36 --i32combined (3-71 ページ).
3.39 --in_place (3-74 ページ).
3.46 --m32 (3-82 ページ).
3.47 --m32combined (3-83 ページ).
3.49 --output=destination (3-85 ページ).
3.62 --text (3-99 ページ).
3.64 --vhx (3-103 ページ).

3.42 --interleave=option

デバッグ情報が存在する場合に、元のソースコードを逆アセンブル結果にコメントとして挿入します。

構文

`--interleave=` *option*

option には、以下のいずれかの値を設定できます。

line_directives
逆アセンブルされた命令のファイル名および行番号を含む `#line` ディレクティブをインターリーブします。

line_numbers
逆アセンブルされた命令のファイル名および行番号を含むコメントをインターリーブします。

なし
インターリーブを無効にします。これは、作成されたメイクファイルで、fromelf コマンドに `--interleave` に加えて複数のオプションがある場合に役立ちます。この場合は、`--interleave=none` を最後のオプションとして指定することで、fromelf コマンド全体を再度指定しなくてもインターリーブを無効にすることができます。

source
ソースコードを含むコメントをインターリーブします。ソースコードが使用できなくなった場合は、`fromelf` ユーティリティは `line_numbers` と同じようにインターリーブします。

source_only
ソースコードを含むコメントをインターリーブします。ソースコードが使用できなくなった場合は、`fromelf` ユーティリティはそのコードをインターリーブしません。

使用法

このオプションは `--emit=code`、`--text -c`、または `--disassemble` と組み合わせて使用します。

ソースコードの検索パスを追加するには、このオプションを `--source_directory` と組み合わせて使用します。

デフォルト

デフォルトは `--interleave=none` です。

関連参照

3.22 --disassemble (3-54 ページ).
3.25 --emit=option[,option,...] (3-57 ページ).
3.59 --source_directory=path (3-95 ページ).
3.62 --text (3-99 ページ).

3.43 --licretry

フローティングライセンスを使用している場合、**fromelf** を起動したときに、10 回までライセンスの取得を試みます。

使用法

このオプションは、ライセンスサーバからライセンスを取得できなかった場合に、ネットワークまたはライセンスサーバの設定に関する他の問題がないことを確認した後でのみ使用して下さい。

このオプションは、ARMCC5_FROMELFOPT 環境変数で指定することをお勧めします。これにより、ビルドファイルを修正する必要がなくなります。

関連情報

ツールチェーンの環境変数.
ARM DS-5 ライセンス管理ガイド.

3.44 --linkview、--no_linkview

ELF イメージのセクションレベルのビューを制御します。

使用法

`--no_linkview` を指定すると、セクションレベルのビューが破棄され、セグメントレベルのビュー (ロード時のビュー) のみが保持されます。

セクションレベルのビューを破棄することによって、以下が削除されます。

- セクションのヘッダテーブル。
- セクションのヘッダストリングテーブル。
- ストリングテーブル。
- シンボルテーブル。
- すべてのデバッグセクション。

出力に含まれるのは、プログラムのヘッダテーブルとプログラムのセグメントのみです。『*System V Application Binary Interface*』仕様によれば、プログラムローダが ELF ファイルに存在すると想定してよいのはこれらのみです。

——— **注** ———

このオプションの使用は廃止される予定です。

———————————

制約条件

以下の使用制限があります。

- `--elf` は、`--linkview` および `--no_linkview` と組み合わせて使用する必要があります。
- SysV イメージには `--no_linkview` オプションを使用しないで下さい。

例

`image.axf` の ELF 形式の出力を生成するには、以下のように入力します。

```
fromelf --no_linkview --elf image.axf --output=image_nlk.axf
```

関連参照

3.24 --elf (3-56 ページ).
3.50 --privacy (3-86 ページ).
3.60 --strip=option[,option,...] (3-96 ページ).

関連情報

--privacy リンカオプション.

3.45 --localize=option[,option,…]

選択されたシンボルをローカルシンボルに変換します。

構文

`--localize=` *option* `[,` *option* `,…]`

option には以下のいずれかを指定できます。

object_name`::`
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルは、ローカルシンボルに変換されます。

object_name`::`*symbol_name*
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルと、シンボル名が *symbol_name* と一致するシンボルは、ローカルシンボルに変換されます。

symbol_name
: シンボル名が *symbol_name* と一致するすべてのシンボルは、ローカルシンボルに変換されます。

以下のことができます。

- ワイルドカード文字 `?` および `*` を *symbol_name* および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

Restrictions

このオプションは `--elf` と組み合わせて使用する必要があります。

関連参照

3.24 --elf (3-56 ページ).
3.33 --hide=option[,option,...] (3-68 ページ).

3.46 --m32

Motorola 32 ビット形式（32 ビット S レコード形式）の出力を作成します。これによりイメージ内のロード領域ごとに 1 つの出力ファイルが生成されます。

この出力のベースアドレスは、`--base` オプションを使用して指定できます。

制約条件

以下の使用制限があります。

- オブジェクトファイルに対してはこのオプションを使用できません。
- このオプションは `--output` と組み合わせて使用する必要があります。

Considerations when using --m32

複数のロード領域を含む ELF イメージをバイナリ形式に変換すると、**fromelf** によって *destination* という名前の出力ディレクトリが作成され、入力イメージ内のロード領域ごとに 1 つのバイナリ出力ファイルが生成されます。出力ファイルは **fromelf** によって *destination* ディレクトリに配置されます。

——— 注 ———

複数のロード領域の場合、対応するロード領域内の最初の空でない実行領域の名前がファイル名に使用されます。

ファイルは、ELF ファイルに含まれているコードまたはデータをロード領域が記述している場合にのみ作成されます。例えば、ZI データのある実行領域のみを含んでいるロード領域から出力ファイルが生成されることはありません。

———————————

例

ELF ファイル `infile.axf` を Motorola 32 ビット形式のファイル（`outfile.bin` など）に変換するには、以下のように入力します。

```
fromelf --m32 --output=outfile.bin infile.axf
```

関連参照

3.1 --base [[object_file::]load_region_ID=]num (3-27 ページ).
3.47 --m32combined (3-83 ページ).
3.49 --output=destination (3-85 ページ).

3.47 --m32combined

Motorola 32 ビット形式（32 ビット S レコード形式）の出力を作成します。複数のロード領域を含むイメージ用に 1 つの出力ファイルが生成されます。

この出力のベースアドレスは、`--base` オプションを使用して指定できます。

Restrictions

以下の使用制限があります。

- オブジェクトファイルに対してはこのオプションを使用できません。
- このオプションは `--output` と組み合わせて使用する必要があります。

Considerations when using --m32combined

If you convert an ELF image containing multiple load regions to a binary format, **`fromelf`** creates an output directory named ***`destination`*** and generates one binary output file for all load regions in the input image. **`fromelf`** places the output file in the ***`destination`*** directory.

ELF イメージは、複数のロード領域を定義しているスキャッタファイルを使用してビルドされた場合などに、複数のロード領域を保持します。

例

2 つのロード領域を持ち、開始アドレスが `0x1000` のイメージファイル（`infile2.axf`）から、Motorola 32 ビット形式の単一の出力ファイル（`outfile2.bin`）を作成するには、以下のように入力します。

```
fromelf --m32combined --base=0x1000 --output=outfile2.bin infile2.axf
```

関連参照

3.1 --base [[object_file::]load_region_ID=]num (3-27 ページ).
3.46 --m32 (3-82 ページ).
3.49 --output=destination (3-85 ページ).

3.48 --only=section_name

指定されたセクションのみを出力結果として表示します。

構文

`--only=` *`section_name`*

`section_name` は、表示するセクションの名前です。

以下のことができます。

- セクションの名前には、ワイルドカード文字 `?` および `*` を使用できます。
- 複数の `--only` オプションを使用することによって、表示するセクションを追加指定できます。

例

以下に `--only` の使用例を示します。

- シンボルテーブル(`.symtab`)のみを表示するには、以下のように入力します。

  ```
  fromelf --only=.symtab --text -s test.axf
  ```

- すべての `ERn` セクションを表示するには、以下のように入力します。

  ```
  fromelf --only=ER? test.axf
  ```

- `HEAP` セクションと、すべてのシンボルテーブルセクションおよびすべてのストリングテーブルセクションを表示するには、以下のように入力します。

  ```
  fromelf --only=HEAP --only=.*tab --text -s -t test.axf
  ```

関連参照

3.62 --text (3-99 ページ).

3.49 --output=destination

出力ファイルの名前、または複数の出力ファイルが作成される場合は出力ディレクトリの名前を指定します。

構文

`--output=` *destination*

`--o` *destination*

destination にはファイルまたはディレクトリを指定できます。以下に例を示します。

`--output=foo`
出力ファイルの名前。

`--output=foo/`
出力ディレクトリの名前。

使用法

`--bin` または `--elf` との使用法を以下に示します。

- 1 つの入力ファイルと 1 つの出力ファイル名を指定できます。
- 多くの入力ファイルを指定して `--elf` を使用する場合、`--in_place` を使用して、入力ファイルに各ファイル処理の出力を上書きできます。
- 多くの入力ファイル名と 1 つの出力ディレクトリを指定した場合、各ファイル処理の出力が出力ディレクトリに書き込まれます。各出力ファイル名は、対応する入力ファイルから付けられます。したがって、`fromelf` を 1 回実行することで多くの ELF ファイルをバイナリ形式または 16 進形式に変換するには、この方法で出力ディレクトリを指定することが唯一の手段になります。
- アーカイブファイルを入力として指定した場合は、出力ファイルもアーカイブになります。例えば、以下のコマンドは、`output.o` という名前のアーカイブファイルを作成します。

```
fromelf --elf --strip=debug mylib.a --output=output.o
```

- アーカイブ内のオブジェクトのサブセットを選択するパターンを括弧で囲んで指定した場合、そのサブセットのみが fromelf によって変換されます。その他のすべてのオブジェクトは、変更されずにそのまま出力アーカイブに渡されます。

関連参照

3.2 --bin (3-29 ページ).
3.24 --elf (3-56 ページ).
3.62 --text (3-99 ページ).

3.50 --privacy

サードパーティに配布されるイメージとオブジェクトに含まれるコードを保護するには、出力ファイルを編集します。

使用法

これらのオプションの効果は、イメージとオブジェクトファイルとで異なります。

イメージの場合:

- セクション名をデフォルト値に変更します。例えば、コードセクションの名前は `.text`
- `--strip symbols` の場合と同様に、シンボルテーブル全体を削除します。
- `.comment` セクション名が削除され、**`fromelf --text`** の出力結果では `[Anonymous Section]` とマークされます。

オブジェクトファイルの場合:

- セクション名をデフォルト値に変更します。例えば、コードセクションの名前は `.text` に変更されます。
- マッピングシンボルおよびビルド属性はシンボルテーブルに維持されます。
- 機能を損なうことなく削除できるローカルシンボルは削除されます。

  再配置のターゲットなど、削除できないシンボルは維持されます。このようなシンボルについては、名前が削除されます。 **`fromelf --text`** の出力結果には、これらが `[Anonymous Symbol]` としてマークされます。

関連参照

3.60 --strip=option[,option,...] (3-96 ページ).

関連情報

--locals、--no_locals リンカオプション.
--privacy リンカオプション.

3.51 --qualify

各出力シンボル名に、関連する構造体を含むソースファイルが示されるように、--fieldoffsets オプションの効果を変更します。

使用法

これにより、2 つのソースファイルが同じ名前の異なる構造体を定義している場合でも、--fieldoffsets オプションで機能的な出力が作成されます。

ソースファイルが現在の場所とは異なる場所にある場合は、ソースファイルパスもインクルードされます。

例

foo という構造体は、例えば、one.h および two.h という 2 つのヘッダで定義されます。

fromelf --fieldoffsets を使用して、リンカで以下のようなシンボルを定義できます。

- foo.a、foo.b、および foo.c
- foo.x、foo.y、および foo.z

fromelf --qualify --fieldoffsets を使用して、リンカで以下のシンボルを定義します。

- oneh_foo.a、oneh_foo.b、および oneh_foo.c
- twoh_foo.x、twoh_foo.y、および twoh_foo.z

関連参照

3.28 --fieldoffsets (3-61 ページ).

3.52 --relax_section=option[,option,…]

指定されたセクションの比較レポートの重大度をエラーから警告に変更します。

制約条件

このオプションは --compare と組み合わせて使用する必要があります。

構文

--relax_section= *option* [, *option* ,…]

option には以下のいずれかを指定できます。

object_name::
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのセクション。

object_name::*section_name*
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのセクションと、セクション名が *section_name* と一致するすべてのセクション。

section_name
: 名前が *section_name* と一致するすべてのセクション。

以下のことができます。

- ワイルドカード文字 ? および * を *symbol_name* および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

関連参照

3.8 --compare=option[,option,...] (3-37 ページ).
3.37 --ignore_section=option[,option,...] (3-72 ページ).
3.53 --relax_symbol=option[,option,...] (3-89 ページ).

3.53 --relax_symbol=option[,option,…]

指定されたシンボルの比較レポートの重大度をエラーから警告に変更します。

制約条件

このオプションは --compare と組み合わせて使用する必要があります。

構文

--relax_symbol= *option* [, *option* ,…]

option には以下のいずれかを指定できます。

object_name::
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボル。

object_name::*section_name*
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルと、シンボル名が *symbol_name* と一致するすべてのシンボル。

symbol_name
: 名前が *symbol_name* と一致するすべてのシンボル。

以下のことができます。

- ワイルドカード文字 ? および * を *symbol_name* および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

関連参照

3.8 --compare=option[,option,...] (3-37 ページ).
3.38 --ignore_symbol=option[,option,...] (3-73 ページ).
3.52 --relax_section=option[,option,...] (3-88 ページ).

3.54 --rename=option[,option,…]

出力 ELF オブジェクトに指定されたシンボルの名前を変更します。

制約条件

このオプションは `--elf` および `--output` と組み合わせて使用する必要があります。

構文

`--rename=` *option* `[,` *option* `,…]`

option には以下のいずれかを指定できます。

object_name`::`*old_symbol_name*`=`*new_symbol_name*
: シンボル名が *old_symbol_name* と一致する ELF オブジェクト *object_name* 内のすべてのシンボルを 置き換えます。

old_symbol_name`=`*new_symbol_name*
: シンボル名が *old_symbol_name* と一致するすべてのシンボルを置き換えます。

以下のことができます。

- ワイルドカード文字 `?` および `*` を *old_symbol_name* 、*new_symbol_name* 、および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

例

This example renames the `clock` symbol in the `timer.axf` image to `myclock`, and creates a new file called `mytimer.axf`:

```
fromelf --elf --rename=clock=myclock --output=mytimer.axf timer.axf
```

関連参照

3.24 --elf (3-56 ページ).
3.49 --output=destination (3-85 ページ).

3.55 --select=select_options

--fieldoffsets または --text -a オプションと共に使用したとき、指定されたパターンリストに一致するフィールドのみを選択します。

構文

--select= *select_options*

select_options は、一致させるパターンリストです。複数のフィールドを選択する場合は、以下のように特殊文字を使用します。

- 複数のフィールドを指定するには、コンマ区切りのリストを使用します。例えば、以下のように入力します。

  a*,b*,c*
- 任意の名前と一致させるには、ワイルドカード文字 * を使用します。
- 任意の 1 文字と一致させるには、ワイルドカード文字 ? を使用します。
- インクルードするフィールドを指定するには、 *select_options* 文字列の前に + を付けます。これはデフォルトの動作です。
- インクルードするフィールドを指定するには、 *select_options* 文字列の前に ~ を付けます。

UNIX プラットフォームで特殊文字を使用する場合は、シェルによって文字列展開がされないように、これらのオプションを引用符で囲む必要があります。

使用法

このオプションは、--fieldoffsets または --text -a と組み合わせて使用します。

関連参照

3.28 --fieldoffsets (3-61 ページ).
3.62 --text (3-99 ページ).

3.56 --show=option[,option,…]

選択されたシンボルの可視性プロパティを変更して、デフォルトの可視性でマークします。

構文

`--show=` *option* `[,` *option* `,…]`

option には以下のいずれかを指定できます。

object_name`::`
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルにデフォルトの可視性がマークされます。

object_name`::`*symbol_name*
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルと、シンボル名が *symbol_name* と一致するすべてのシンボルにデフォルトの可視性がマークされます。

symbol_name
: シンボル名が *symbol_name* と一致するすべてのシンボルにデフォルトの可視性がマークされます。

以下のことができます。

- ワイルドカード文字 `?` および `*` を *symbol_name* および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

Restrictions

このオプションは `--elf` と組み合わせて使用する必要があります。

関連参照

3.24 --elf (3-56 ページ).
3.33 --hide=option[,option,...] (3-68 ページ).

3.57 --show_and_globalize=option[,option,…]

選択されたシンボルの可視性プロパティを変更して、デフォルトの可視性でマークし、選択されたシンボルをグローバルシンボルに変換します。

構文

`--show_and_globalize=` *option* `[,` *option* `,…]`

option には以下のいずれかを指定できます。

object_name`::`
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボル。

object_name`::`*symbol_name*
: 名前が *object_name* と一致する ELF オブジェクト内のすべてのシンボルと、シンボル名が *symbol_name* と一致するすべてのシンボル。

symbol_name
: シンボル名が *symbol_name* と一致するすべてのシンボル。

以下のことができます。

- ワイルドカード文字 `?` および `*` を *symbol_name* および *object_name* 引数のシンボル名に使用できます。
- 1 つの *オプション* にコンマ区切りの引数リストを続けることで、複数の値を指定できます。

Restrictions

このオプションは `--elf` と組み合わせて使用する必要があります。

関連参照

3.24 --elf (3-56 ページ).

3.58 --show_cmdline

ELF ファイル変換ツール によって使用されたコマンドラインを出力します。

使用法

ELF ファイル変換ツール によって処理された後のコマンドラインを表示することによって、以下の点を確認できます。

- ビルドシステムによって使用されているコマンドライン
- 指定されたコマンドラインが ELF ファイル変換ツール によってどのように解釈されているか（コマンドラインオプションの順序など）

コマンドは正規化されて表示されます。また、via ファイルの内容は展開されます。

出力結果は標準エラーストリーム（`stderr`）に送られます。

関連参照

3.65 --via=file (3-104 ページ).

3.59 --source_directory=path

ソースコードのディレクトリを明示的に指定します。

構文

```
--source_directory= path
```

使用法

デフォルトでは、ELF 入力ファイルの相対ディレクトリにソースコードが配置されているものと想定されます。このオプションを複数回使用して、複数のディレクトリの検索パスを指定できます。

このオプションは `--interleave` と組み合わせて使用します。

関連参照

3.42 --interleave=option (3-78 ページ).

3.60 --strip=option[,option,…]

サードパーティに配布されるイメージとオブジェクトに含まれるコードを保護できます。さらに、出力イメージのサイズを減らすためにも使用できます。

構文

`--strip=` *option* [`,` *option* `,…`]

option には以下のいずれかを指定できます。

all
オブジェクトモジュールの場合、このオプションによって ELF ファイルからすべてのデバッグ、コメント、メモ、およびシンボルが削除されます。実行可能ファイルの場合、このオプションは `--no_linkview` と同じように機能します。

——— 注 ———
SysV イメージには `--strip=all` オプションを使用しないで下さい。
———————

debug
ELF ファイルから、すべてのデバッグセクションを削除します。

comment
ELF ファイルから、 **`.comment`** セクションを削除します。

filesymbols
`STT_FILE` シンボルが ELF ファイルから削除されます。

localsymbols
これらのオプションの効果は、イメージとオブジェクトファイルとで異なります。

イメージの場合、マッピングシンボルを含むすべてのローカルシンボルが出力シンボルテーブルから削除されます。

オブジェクトファイルの場合:

- マッピングシンボルおよびビルド属性はシンボルテーブルに維持されます。
- 機能を損なうことなく削除できるローカルシンボルは削除されます。

  再配置のターゲットなど、削除できないシンボルは維持されます。このようなシンボルについては、名前が削除されます。 **`fromelf --text`** の出力結果には、これらが `[Anonymous Symbol]` としてマークされます。

notes
ELF ファイルから、 **`.notes`** セクションを削除します。

pathnames
タイプが `STT_FILE` であるすべてのシンボルからパス情報を削除します。例えば、`STT_FILE` シンボルの名前が `C:\work\myobject.o` の場合、`myobject.o` という名前に変更されます。

——— 注 ———
デバッグ情報に含まれるパス名は、このオプションでは排除されません。
———————

symbols

これらのオプションの効果は、イメージとオブジェクトファイルとで異なります。

イメージの場合、シンボルテーブル全体とすべてのスタティックシンボルが削除されます。そのようなスタティックシンボルがスタティックな再配置ターゲットとして使用されている場合、再配置情報も削除されます。どの場合でも、STT_FILE シンボルは削除されます。

オブジェクトファイルの場合:

- マッピングシンボルおよびビルド属性はシンボルテーブルに維持されます。
- 機能を損なうことなく削除できるローカルシンボルは削除されます。

  再配置のターゲットなど、削除できないシンボルは維持されます。このようなシンボルについては、名前が削除されます。 **fromelf --text** の出力結果には、これらが [Anonymous Symbol] としてマークされます。

——— 注 ———

シンボル、パス名、ファイルシンボルを外すと、ファイルのデバッグがしにくくなる可能性があります。

———————————

制約条件

このオプションは --elf および --output と組み合わせて使用する必要があります。

例

デバッグ情報を含めて生成した ELF ファイル infile.axf からデバッグ情報を含まない output.axf ファイルを生成するには、、以下のように入力します。

```
fromelf --strip=debug,symbols --elf --output=outfile.axf infile.axf
```

関連参照

3.24 --elf (3-56 ページ).
3.44 --linkview、--no_linkview (3-80 ページ).
3.50 --privacy (3-86 ページ).

関連情報

マッピングシンボルについて.
--locals、--no_locals リンカオプション.
--privacy リンカオプション.

3.61 --symbolversions、--no_symbolversions

このオプションを指定すると、シンボルバージョン管理テーブルのデコードが無効になります。

制約条件

このオプションと共に `--elf` を使用する場合は、`--output` も使用する必要があります。

関連情報

シンボルバージョン管理について.
Base Platform ABI for the ARM Architecture .

3.62 --text

このオプションを指定すると、イメージ情報をテキスト形式で出力できます。このオプションを使用すると、ELF イメージファイルまたは ELF オブジェクトファイルをデコードできます。

構文

`--text [ options ]`

options は表示する内容です。以下のいずれかを指定できます。

-a

グローバルデータアドレスとスタティックデータアドレス(構造体とユニオンの内容のアドレスも含む)を出力します。

このオプションは、デバッグ情報を含むファイルに対してのみ使用できます。デバッグ情報が含まれない場合、警告が表示されます。

データアドレスのサブセットを出力する場合は、`--select` オプションを使用します。

構造体内外で展開された配列のデータアドレスを参照するには、このテキストカテゴリと共に `--expandarrays` オプションを使用します。

-c

このオプションは、逆アセンブル対象の元のバイナリデータのダンプおよび命令のアドレスと共に、コードを逆アセンブルします。

——— 注 ———

`--disassemble` とは異なり、逆アセンブリをアセンブラに入力することはできません。

———————

-d

データセクションの内容を出力します。

-e

オブジェクトの例外テーブル情報をデコードします。イメージを逆アセンブルするときに、`-c` と組み合わせて使用します。

-g

デバッグ情報を出力します。

-r

再配置情報を出力します。

-s

シンボルテーブルとバージョン管理テーブルを出力します。

-t

ストリングテーブルを出力します。

-v

イメージの各セグメントヘッダとセクションヘッダに関する詳細情報を出力します。

-w

行を折り返しません。

-y

ダイナミックセグメントの内容を出力します。

-z

コードサイズとデータサイズを出力します。

これらのオプションはテキストモードでのみ認識されます。

使用法

コードの出力形式を指定しない場合は、--text が想定されます。つまり、--text を指定しなくてもオプション(複数可)を指定することができます。例えば、fromelf -a は fromelf --text -a と同じ意味です。

コードの出力形式(--bin など)を指定した場合、--text オプションはすべて無視されます。

destination が --output オプションと組み合わせて指定されていない場合、または --output が指定されていない場合、情報が標準出力(stdout)に出力されます。

例

以下に --text の使用例を示します。

- 逆アセンブルされた ELF イメージとシンボルテーブルを含むプレーンテキスト出力ファイルを生成するには、以下のように入力します。

```
fromelf --text -c -s --output=outfile.lst infile.axf
```

- すべてのグローバルデータ変数とスタティックデータ変数、すべての構造体フィールドのアドレス一覧を stdout に出力するには、以下のように入力します。

```
fromelf -a --select=* infile.axf
```

- inputfile.axf に含まれるすべての構造体のアドレスを保持し、グローバルデータ変数またはスタティックデータ変数に関する情報は保持しないテキストファイルを生成するには、以下のように入力します。

```
fromelf --text -a --select=*.* --output=structaddress.txt infile.axf
```

- ネストされた構造体のアドレスのみを含むテキストファイルを生成するには、以下のように入力します。

```
fromelf --text -a --select=*.*.* --output=structaddress.txt infile.axf
```

- inputfile.axf に含まれるすべてのグローバル変数またはスタティックデータ変数の情報を保持し、構造体のアドレスは保持しないテキストファイルを生成するには、以下のように入力します。

```
fromelf --text -a --select=*,~*.* --output=structaddress.txt infile.axf
```

関連タスク

2.6 実行可能な ELF イメージ内のシンボルの場所を fromelf を使用して調べる方法 (2-23 ページ).

関連参照

3.11 --cpu=name (3-41 ページ).
3.22 --disassemble (3-54 ページ).
3.25 --emit=option[,option,...] (3-57 ページ).
3.26 --expandarrays (3-59 ページ).
3.40 --info=topic[,topic,...] (3-75 ページ).
3.42 --interleave=option (3-78 ページ).
3.48 --only=section_name (3-84 ページ).
3.49 --output=destination (3-85 ページ).
3.55 --select=select_options (3-91 ページ).
3.67 -w (3-106 ページ).

関連情報

イメージに関する情報を取得するためのリンカオプション.

3.63 --version_number

使用している **fromelf** のバージョンを表示します。

使用法

ELF ファイル変換ツールは、nnnbbb 形式のバージョン番号を表示します。各項目には以下の意味があります。

- nnn はバージョン番号です。
- bbbb はビルド番号を示します。

例

バージョン 5.01 ビルド 0019 は `5010019` と表示されます。

関連参照

3.32 --help (3-67 ページ).
3.66 --vsn (3-105 ページ).

3.64 --vhx

バイト指向（Verilog メモリモデル）16 進形式の出力を作成します。

使用法

この形式は、*ハードウェア記述言語*（HDL）シミュレータのメモリモデルへのロードに適しています。*--widthxbanks* オプションを使用して、このオプションによって生成される出力を複数ファイルに分割できます。

制約条件

以下の使用制限があります。

- オブジェクトファイルに対してはこのオプションを使用できません。
- このオプションは `--output` と組み合わせて使用する必要があります。

Considerations when using --vhx

複数のロード領域を含む ELF イメージをバイナリ形式に変換すると、**fromelf** によって *destination* という名前の出力ディレクトリが作成され、入力イメージ内のロード領域ごとに 1 つのバイナリ出力ファイルが生成されます。出力ファイルは **fromelf** によって *destination* ディレクトリに配置されます。

——— 注 ———

複数のロード領域の場合、対応するロード領域内の最初の空でない実行領域の名前がファイル名に使用されます。

ファイルは、ELF ファイルに含まれているコードまたはデータをロード領域が記述している場合にのみ作成されます。例えば、ZI データのある実行領域のみを含んでいるロード領域から出力ファイルが生成されることはありません。

———————————

サンプル

ELF ファイル `infile.axf` をバイト指向 16 進形式のファイル（`outfile.bin` など）に変換するには、以下のように入力します。

```
fromelf --vhx --output=outfile.bin infile.axf
```

8 ビットのメモリバンクを 2 つ持つイメージファイル multiload.axf から、`regions` ディレクトリに複数の出力ファイルを作成するには、以下のように入力します。

```
fromelf --vhx --8x2 multiload.axf --output=regions
```

関連参照

3.49 --output=destination (3-85 ページ).
3.68 --widthxbanks (3-107 ページ).

3.65 --via=file

入力ファイル名と ELF ファイル変換ツール オプションの追加リストを *filename* から読み取ります。

構文

`--via=` *filename*

filename は、コマンドラインでインクルードされるオプションを含む via ファイルの名前です。

使用法

ELF ファイル変換ツール コマンドラインでは複数の `--via` オプションを入力できます。オプション、`--via` は、via ファイル内に含めることもできます。

3.66 --vsn

バージョン情報とライセンス情報が表示されます。

例

出力例:

```
&gt; fromelf --vsn
製品:ARM Compiler N.nn
コンポーネント:ARM Compiler N.nn
ツール:fromelf [build_number]
license_type ソフトウェアの提供元:ARM Limited
```

関連参照

3.32 --help (3-67 ページ).
3.63 --version_number (3-102 ページ).

3.67 -w

通常なら複数行で表示されるテキスト出力情報が 1 行で表示されます。

使用法

Perl などのテキスト処理ユーティリティで解析する際に、出力結果を処理しやすい形式にすることができます。

以下に例を示します。

```
&gt; fromelf --text -w -c test.axf
============================================================================ ** ELF Header
Information ...============================================================================
 ** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]    Size   :36 bytes
(alignment 4)    Address:0x00000000    $a      .text ...** Section #7 '.rel.text'
(SHT_REL)    Size   :8 bytes (alignment 4)    Symbol table #6 '.symtab'    1
relocations applied to section #1 '.text' ** Section #2 '.ARM.exidx' (SHT_ARM_EXIDX)
[SHF_ALLOC + SHF_LINK_ORDER]    Size   :8 bytes (alignment 4)    Address:0x
00000000    Link to section #1 '.text' ** Section #8 '.rel.ARM.exidx' (SHT_REL)
Size   :8 bytes (alignment 4)    Symbol table #6 '.symtab'    1 relocations applied to
section #2 '.ARM.exidx' ** Section #3 '.arm_vfe_header' (SHT_PROGBITS)    Size   :4
bytes (alignment 4) ** Section #4 '.comment' (SHT_PROGBITS)    Size   :74 bytes **
Section #5 '.debug_frame' (SHT_PROGBITS)    Size   :140 bytes ** Section #9
'.rel.debug_frame' (SHT_REL)    Size   :32 bytes (alignment 4)    Symbol table #6
'.symtab'    4 relocations applied to section #5 '.debug_frame' ** Section #6
'.symtab' (SHT_SYMTAB)    Size   :176 bytes (alignment 4)    String table #11
'.strtab'    Last local symbol no. 5 ** Section #10 '.shstrtab' (SHT_STRTAB)
Size   :110 bytes ** Section #11 '.strtab' (SHT_STRTAB)    Size   :223 bytes **
Section #12 '.ARM.attributes' (SHT_ARM_ATTRIBUTES)    Size   :69 bytes
```

関連参照

3.62 --text (3-99 ページ).

3.68 --widthxbanks

このオプションを指定すると、複数のメモリバンク用に複数のファイルが出力されます。

構文

--widthxbanks

各項目には以下の意味があります。

banks
ターゲットメモリシステム内のメモリバンクの数を指定します。これにより、各ロード領域に生成される出力ファイルの数が決まります。

width
ターゲットメモリシステムにおけるメモリの幅を指定します（8 ビット、16 ビット、32 ビット、または 64 ビット）。

有効な設定は以下のとおりです。

```
--8x1 --8x2 --8x4 --16x1 --16x2 --32x1 --32x2 --64x1
```

使用法

複数の設定が指定されている場合、**fromelf** は最後に指定された設定を使用します。

イメージに 1 つのロード領域がある場合は、fromelf によって、 *banks* で指定された数のファイルが生成されます。ファイル名は、以下の命名規則に基づいて --output=*destination* 引数から付けられます。

- メモリバンクが 1 つしかない場合（ *banks* =1）、出力ファイルの名前は *destination* になります。
- 複数のメモリバンクがある場合（ *banks* >1）、fromelf は、*destinationN* に指定された *banks* 数の ファイルを生成します。*N* は、0 から *banks* - 1 までの範囲です。出力ファイル名のファイル拡張子を指定すると、ファイル拡張子の前に番号 *N* が付けられます。例えば、

  ```
  fromelf --vhx --8x2 test.axf --output=test.txt
  ```

  これにより、test0.txt および test1.txt という名前の 2 つのファイルが生成されます。

イメージに複数のロード領域がある場合は、fromelf は、*destination* という名前のディレクトリを作成し、そのディレクトリに各ロード領域用の *banks* ファイルを生成します。各ロード領域用のファイルには *load_regionN* という名前が付けられます。 *load_region* はロード領域の名前で、*N* は、0 から *banks* - 1 までの範囲です。以下に例を示します。

```
fromelf --vhx --8x2 multiload.axf --output=regions/
```

これにより regions ディレクトリに以下のようなファイルが生成されます。

```
EXEC_ROM0 EXEC_ROM1 RAM0 RAM1
```

width により指定されたメモリの幅によって、各出力ファイルの 1 行に保存されるメモリの量が決まります。各出力ファイルのサイズは、読み取るメモリのサイズを、作成されるファイル数で分割したものです。以下に例を示します。

- **fromelf --vhx --8x4 test.axf --output=file** により 4 つのファイル（**file0**、**file1**、**file2**、**file3**）が生成されます。各ファイルには、以下のような 1 バイトの行が含まれます。

```
00 00 2D 00 2C 8F
…
```

- **fromelf --vhx --16x2 test.axf --output=file** により 2 つのファイル（**file0** および **file1**）が生成されます。各ファイルには、以下のような 2 バイトの行が含まれます。

```
0000 002D 002C
…
```

制約条件

このオプションは **--output** と組み合わせて使用する必要があります。

関連参照

3.2 --bin (3-29 ページ).
3.49 --output=destination (3-85 ページ).
3.64 --vhx (3-103 ページ).

第 4 章
via ファイルの構文

fromelf でサポートされている via ファイルの構文について説明します。

このドキュメントは、次で構成されています。

4.1 via ファイルの概要

via ファイルは、ELF ファイル変換ツールコマンドライン引数とオプションを指定できるプレーンテキストファイルです。

通常、コマンドラインの長さの制限を解決するために via ファイルを使用します。ただし、以下のような複数の via ファイルを作成します。

- 同じような引数とオプションをグループ化するファイル。
- 異なるシナリオで使用する異なる引数とオプションのセットを含んでいるファイル。

注

一般的には、via ファイルを使用して、ツールに対して任意のコマンドラインオプション（`--via` を含む）を指定できます。つまり、ネストされた複数の via ファイルを via ファイル内から呼び出すことができます。

via ファイルの評価

ELF ファイル変換ツール が呼び出されると、以下の処理が行われます。

1. 指定されている最初の `--via` *`via_file`* 引数を、via ファイルから抽出された引数ワードのシーケンスに置き換えます。この中には、再帰処理を行う、via ファイル内でネストされた `--via` コマンドも含まれます。
2. それ以降の `--via` *`via_file`* 引数についても、出現した順番で同じように処理します。

つまり、via ファイルは指定された順番で処理され、ネストされた via ファイルを含めて各 via ファイルが完全に処理されてから次の via ファイルが処理されます。

関連参照

4.2 via ファイルの構文規則 (4-111 ページ).
3.65 --via=file (3-104 ページ).

4.2 via ファイルの構文規則

via ファイルは構文規則に準拠している必要があります。

- via ファイルは、一連のワードで構成されるテキストファイルです。テキストファイル内の各ワードは、引数文字列に変換されてからツールに渡されます。
- 区切られた文字列内にある場合を除き、ワードはホワイトスペースまたは行の終わりで区切られます。以下に例を示します。

  `--debugonly --privacy`（2 ワード）

  `--debugonly --privacy`（1 ワード）
- 行の終わりはホワイトスペースとして処理されます。以下に例を示します。

  ```
  --debugonly--privacy
  ```

  これは以下のコードと同等です。

  `--debugonly --privacy`
- 二重引用符（"）またはアポストロフィ（'）で囲まれた文字列は、1 ワードとして処理されます。二重引用符で囲まれたワード内で使用されているアポストロフィは通常の文字として処理されます。アポストロフィで区切られたワード内では、二重引用符は通常の文字として処理されます。

  二重引用符を使用して、スペースを含むファイル名またはパス名を 1 つのワードとしてまとめます。以下に例を示します。

  `--output C:\My Project\output.txt`（3 ワード） `--output "C:\My Project \output.txt"`（2 ワード）

  また、アポストロフィを使用して、二重引用符を含むワードを 1 つのワードとしてまとめます。以下に例を示します。

  `-DNAME='"ARM コンパイラ"'`（1 ワード）
- 括弧で囲まれた文字は、1 ワードとして処理されます。以下に例を示します。

  `--option(x, y, z)`（1 ワード）

  `--option (x, y, z)`（2 ワード）
- 二重引用符またはアポストロフィで囲まれた文字列内では、バックスラッシュ（\）文字を使用して、二重引用符、アポストロフィ、およびバックスラッシュ文字をエスケープできます。
- 1 つのワードとしてまとめられたワードのすぐ隣にあるワードは、1 ワードとして処理されます。以下に例を示します。

  `--output"C:\Project\output.txt"`

  これは、以下の 1 ワードとして処理されます。

  `--output C:\Project\output.txt`
- 先頭にあるホワイトスペース文字を除いて、セミコロン（;）またはハッシュ（#）文字で始まる行は、コメント行として解釈されます。行頭以外の場所にあるセミコロンまたはハッシュ文字は、コメントの開始を表す文字としては解釈されません。以下に例を示します。

  ```
  -o objectname.axf      ;これはコメントではありません
  ```

  コメントの終わりは、行の終わりまたはファイルの終わりとなります。複数行にわたるコメントはなく、行の一部だけがコメントになることもありません。

関連概念

4.1 via ファイルの概要 (4-110 ページ).

関連参照

3.65 --via=file (3-104 ページ).

付録 A
fromelf ドキュメントの改訂

『fromelf イメージ変換ユーティリティユーザガイド』に対して加えられた技術的変更について 説明します。

このドキュメントは、次で構成されています。

- *A.1 『fromelf イメージ変換ユーティリティユーザガイド』に対する改訂 (付録-A-114 ページ).*

A.1 『fromelf イメージ変換ユーティリティユーザガイド』に対する改訂

『fromelf イメージ変換ユーティリティユーザガイド』に対して、以下の技術的変更が加えられました。

表 A-1 発行 H と発行 J の相違点

変更点	関連するトピック
fromelf は空のロード領域用にはバイナリ出力ファイルを生成しない旨のメモを追加しました。	• • *3.2 --bin (3-29 ページ)* • *3.6 --cad (3-34 ページ)* • *3.35 --i32 (3-70 ページ)* • *3.46 --m32 (3-82 ページ)*
サンプルを対応するコマンドラインオプションに移動して、トピックを削除しました。	• ELF イメージから Intel Hex 32 ビット形式への変換。 • ELF イメージから Motorola 32 ビット形式への変換。 • ELF イメージからプレーンバイナリ形式への変換。 • ELF イメージからバイト指向 (Verilog メモリモデル) 16 進形式への変換。 • 出力ファイル内にデバッグ情報を含めるかどうかの制御。 • ELF 形式のファイルの逆アセンブル。
`--cpu` オプションと `--fpu` オプションについて詳しく記述しました。	• *3.11 --cpu=name (3-41 ページ)* • *3.30 --fpu=name (3-64 ページ)*
via ファイル構文に関する章を追加しました。	• *4 via ファイルの構文 (4-109 ページ)*

表 A-2 発行 G と発行 H の相違点

変更点	関連するトピック
`--base` の説明を改善しました。	*3.1 --base [[object_file::]load_region_ID=]num (3-27 ページ)*
`--fieldoffsets` の例を修正し、出力例を追加しました。	*3.28 --fieldoffsets (3-61 ページ)*
`--qualify` の説明を改善しました。	*3.51 --qualify (3-87 ページ)*

表 A-3 発行 F と発行 G の相違点

変更点	関連するトピック
トピックのタイトルにある誤ったアンダースコアを削除し、代替の構文を追加しました。	*3.49 --output=destination (3-85 ページ)*

表 A-4 発行 D と発行 F の相違点

変更点	関連するトピック
--device オプションが廃止された旨のメモを追加しました。	• *3.15 --device=list (3-47 ページ)* • *3.16 --device=name (3-48 ページ)*
`--version_number` および `--vsn` で報告されたバージョン番号を変更しました。	• *3.63 --version_number (3-102 ページ)* • *3.66 --vsn (3-105 ページ)*.

表 A-5 発行 C と発行 D の相違点

変更点	関連するトピック
`--project=filename` 、`--no_project` のトピックタイトルを修正しました。 `--project`、`--reinitialize_workdir`、および `--workdir` オプションの説明についてのメモを追加しました。	• *--project=filename、--no_project* • *--reinitialize_workdir* • *--workdir=directory*

表 A-6 発行 A と発行 B の相違点

変更点	関連するトピック
fromelf でアーカイブ内のすべてのファイルを処理できるようになった旨のリスト項目を追加しました。	*1.1 fromelf イメージ変換ユーティリティについて (1-12 ページ)*
アーカイブ内の ELF ファイルを処理する方法を説明する新しいトピックを追加しました。	*2.2 アーカイブ内の ELF ファイルを処理する例 (2-18 ページ)*
fromelf を使用してイメージおよびオブジェクトに含まれるコードを保護する方法および `--privacy` と `--strip` のコマンドラインオプションの使用に関する説明を明確にしました。	• *2.3 fromelf を使用してイメージファイルに含まれるコードを保護するオプション (2-19 ページ)* • *2.4 fromelf を使用してオブジェクトファイルに含まれるコードを保護するオプション (2-20 ページ)*
`--decode_build_attributes` コマンドラインオプションの例を追加しました。	*3.14 --decode_build_attributes (3-46 ページ)*
`--dump_build_attributes` コマンドラインオプションの例を追加しました。	*3.23 --dump_build_attributes (3-55 ページ)*
`--emit` コマンドラインオプションの `build_attributes` オプションの説明を変更しました。	*3.25 --emit=option[,option,...] (3-57 ページ)*
`--extract_build_attributes` コマンドラインオプションの例を追加しました。	*3.27 --extract_build_attributes (3-60 ページ)*
`input_file` の説明を変更して、アーカイブ内の ELF ファイルの処理に関する説明を追加しました。	*3.41 input_file (3-76 ページ)*
`--[no_]linkview` コマンドラインオプションが廃止された旨を明記しました。	*3.44 --linkview、--no_linkview (3-80 ページ)*
`--output` コマンドラインオプションで入力アーカイブファイルを使用する方法に関する情報を 追加しました。	*3.49 --output=destination (3-85 ページ)*

表 A-6 発行 A と発行 B の相違点（続き）

変更点	関連するトピック
`--privacy` コマンドラインオプションの説明を明確にしました。	*3.50 --privacy (3-86 ページ)*
`--strip` コマンドラインオプションの `localsymbols` オプションと `symbols` オプションの説明を明確にしました。	*3.60 --strip=option[,option,...] (3-96 ページ)*
オブジェクトファイルで使用できない以下のコマンドラインオプションに関する制限を追加しました。`--bin`、`--bincombined`、`--cad`、`--cadcombined`、`--i32`、`--i32combined`、`--m32`, `--m32combined`、および `--vhx`	• *3.2 --bin (3-29 ページ)* • *3.3 --bincombined (3-30 ページ)* • *3.6 --cad (3-34 ページ)* • *3.7 --cadcombined (3-36 ページ)* • *3.35 --i32 (3-70 ページ)* • *3.36 --i32combined (3-71 ページ)* • *3.46 --m32 (3-82 ページ)* • *3.47 --m32combined (3-83 ページ)* • *3.64 --vhx (3-103 ページ)*.