C28x 組込みアプリケーション バイナリ インターフェイス

Application Note

Literature Number: JADA099C
FEBRUARY 2019 – REVISED MARCH 2026

目次

1 はじめに.....9
1.1 C28x 向けの ABI.....10
1.2 範囲.....11
1.3 ABI バリアント.....12
1.4 ツールチェーンと相互運用性.....12
1.5 ライブラリ.....12
1.6 オブジェクト ファイルの形式.....13
1.7 セグメント.....13
1.8 C28x アーキテクチャの概要.....13
1.9 C28x のメモリモデル.....13
1.10 参考資料.....13
1.11 コード フラグメント表記.....14
2 データ表現.....15
2.1 基本型.....16
2.2 レジスタ内のデータ.....17
2.3 メモリ内のデータ.....17
2.4 ポインタ タイプ.....17
2.5 複素数型.....18
2.6 構造と共用体.....18
2.7 配列.....18
2.8 ビット フィールド.....19
2.8.1 volatile ビット フィールド.....20
2.9 列挙型.....20
3 呼び出し規約.....21
3.1 呼び出しと復帰.....22
3.1.1 呼び出し命令.....22
3.1.2 復帰命令.....22
3.1.3 パイプライン規約.....22
3.1.4 弱関数.....22
3.2 レジスタ規約.....22
3.2.1 引数レジスタ.....24
3.2.2 呼び出し先保存レジスタ.....25
3.3 引数の受け渡し.....25
3.3.1 16 ビット引数の受け渡し.....25
3.3.2 長い引数を渡す.....26
3.3.3 C++ における引数の受け渡し.....27
3.3.4 構造と共用体を渡す.....27
3.3.5 レジスタに渡されない引数のスタック レイアウト.....27
3.3.6 フレーム ポインタ.....27
3.4 戻り値.....27
3.5 参照によって渡される構造と共用体と、返される構造と共用体.....28
3.6 コンパイラ ヘルパー関数の規則.....28
3.7 プロローグおよびエピローグのヘルパー関数.....29
3.8 既知の関数のスクラッチ レジスタ.....29
3.9 割り込み関数.....29
4 データのアロケーションとアドレッシング.....31

Copyright © 2026 Texas Instruments Incorporated

4.1 データ セクションとセグメント....32
4.2 データ ブロッキング....33
4.3 アドレッシング モード....33
4.4 静的データの割り当てとアドレッシング....34
4.4.1 静的データのアドレッシング方法....34
4.4.2 静的データの配置規則....34
4.4.3 静的データの初期化....34
4.5 自動変数....34
4.6 フレーム レイアウト....35
4.6.1 スタックの整列....36
4.6.2 レジスタ保存順序....36
4.7 ヒープ割り当て済みオブジェクト....37
5 コードのアロケーションとアドレッシング....38
5.1 コード ラベルのアドレスの計算....39
5.2 呼び出し....39
5.2.1 直接 呼び出し....39
5.2.2 ファー呼び出しトランポリン....39
5.2.3 間接呼び出し....39
6 ヘルパー関数 API....40
6.1 浮動小数点の動作....41
6.2 C ヘルパー関数 API....41
6.3 C99 用の浮動小数点ヘルパー関数....43
7 標準 C ライブラリ API....44
7.1 標準 C ライブラリについて....45
7.2 予約済みシンボル....45
7.3 <assert.h> の実装....45
7.4 <complex.h> の実装....45
7.5 <ctype.h> の実装....46
7.6 <errno.h> の実装....46
7.7 <float.h> の実装....46
7.8 <inttypes.h> の実装....46
7.9 <iso646.h> の実装....46
7.10 <limits.h> の実装....47
7.11 <locale.h> の実装....47
7.12 <math.h> の実装....47
7.13 <setjmp.h> の実装....48
7.14 <signal.h> の実装....48
7.15 <stdarg.h> の実装....48
7.16 <stdbool.h> の実装....48
7.17 <stddef.h> の実装....48
7.18 <stdint.h> の実装....49
7.19 <stdio.h> の実装....49
7.20 <stdlib.h> の実装....49
7.21 <string.h> の実装....50
7.22 <tgmath.h> の実装....50
7.23 <time.h> の実装....50
7.24 <wchar.h> の実装....50
7.25 <wctype.h> の実装....50
8 C++ ABI....51
8.1 制限 (GC++ABI 1.2)....52
8.2 エクスポート テンプレート (GC++ABI 1.4.2)....52
8.3 データ レイアウト (GC++ABI 第 2 章)....52
8.4 初期化ガード変数 (GC++ABI 2.8)....52
8.5 コンストラクタ戻り値 (GC++ABI 3.1.5)....52
8.6 ワンタイム構築 API (GC++ABI 3.3.2)....52
8.7 オブジェクト構成順序を制御する (GC++ ABI 3.3.4)....52

Copyright © 2026 Texas Instruments Incorporated

8.8 デマングラ API (GC++ABI 3.4)....53
8.9 静的データ (GC++ ABI 5.2.2)....53
8.10 仮想テーブルと key function (GC++ABI 5.2.3)....53
8.11 アンワインド テーブルの位置 (GC++ABI 5.3)....53
9 例外処理....54
9.1 概要....55
9.2 PREL31 エンコーディング....55
9.3 例外インデックス テーブル (EXIDX)....56
9.3.1 アウトオブライン EXTAB エントリへのポインタ....56
9.3.2 EXIDX_CANTUNWIND....56
9.3.3 インライン化された EXTAB エントリ....56
9.4 例外処理命令テーブル (EXTAB)....57
9.4.1 EXTAB 一般モデル....57
9.4.2 EXTAB コンパクト モデル....57
9.4.3 パーソナリティ ルーチン....58
9.5 アンワインド命令....58
9.5.1 共通シーケンス....58
9.5.2 バイト エンコードされたアンワインド命令....59
9.6 ディスクリプタ....61
9.6.1 型識別子のエンコード....61
9.6.2 範囲....61
9.6.3 クリーンアップ記述子....62
9.6.4 キャッチ ディスクリプタ....62
9.6.5 関数例外仕様 (FESPEC) ディスクリプタ....63
9.7 特別なセクション....63
9.8 C++ 以外のコードとの相互作用....63
9.8.1 EXIDX エントリの自動生成....63
9.8.2 ハンドコードで記述されたアセンブリ関数....63
9.9 システム機能との相互作用....64
9.9.1 共有ライブラリ....64
9.9.2 オーバーレイ....64
9.9.3 割り込み....64
9.10 TI ツールチェーンにおけるアセンブリ言語オペレータ....64
10 DWARF....65
10.1 DWARF レジスタの名前....66
10.2 呼び出しフレーム情報....67
10.3 ベンダー名....68
10.4 ベンダー拡張....68
11 ELF オブジェクト ファイル (プロセッサ補足)....70
11.1 登録済みベンダ名....71
11.2 ELF ヘッダー....71
11.3 セクション....72
11.3.1 セクション インデックス....72
11.3.2 セクション タイプ....72
11.3.3 拡張セクション ヘッダー属性....73
11.3.4 サブセクション....73
11.3.5 特別なセクション....73
11.3.6 セクションの整列....75
11.4 シンボル テーブル....76
11.4.1 シンボル タイプ....76
11.4.2 共通ブロック シンボル....76
11.4.3 シンボル名....76
11.4.4 予約済みシンボル名....76
11.4.5 マッピング シンボル....76
11.5 再配置....76
11.5.1 再配置タイプ....77

English Document: SPRAC71
Copyright © 2026 Texas Instruments Incorporated

11.5.2 再配置操作......79
11.5.3 未解決の弱参照の再配置......79
12 ELF プログラムのロードとリンク (プロセッサ補足)......80
12.1 プログラム ヘッダー......81
12.1.1 ベース アドレス......81
12.1.2 セグメントの内容......81
12.1.3 スレッドローカル ストレージ......81
12.2 プログラムのロード......82
13 ビルド属性......83
13.1 ビルド属性について......84
13.2 C28x ABI ビルド属性サブセクション......84
13.3 ビルド属性タグ......85
14 コピー テーブルと変数の初期化......88
14.1 テーブルのコピーについて......89
14.2 コピー テーブルのフォーマット......91
14.3 圧縮データ フォーマット......92
14.3.1 RLE......92
14.3.2 LZSS フォーマット......93
14.4 変数の初期化......93
15 改訂履歴......96

図の一覧

図 1-1. ABI 仕様の一部......11
図 3-1. C28x のレジスタ......24
図 4-1. データ セクションとセグメント (標準)......32
図 4-2. ローカル フレーム レイアウト......36
図 9-1. 短いフォームの範囲......61
図 9-2. 長いフォームの範囲......61
図 14-1. テーブ のコピーの概要......90
図 14-2. ハンドラ テーブルのフォーマット......91
図 14-3. 圧縮ソース データ フォーマット......92
図 14-4. cinit を介した ROM ベースの変数の初期化......94
図 14-5. .cinit セクション......94

表の一覧

表 2-1. 標準型のデータ サイズ......16
表 2-2. ポインタのデータ サイズ......17
表 3-1. TMS320C28x レジスタ規約......23
表 4-1. TMS320C28x アドレッシング モード......33
表 6-1. TMS320C28x EABI 関数......41
表 6-2. 予約済み浮動小数点分類ヘルパー関数......43
表 9-1. C28x TDEH パーソナリティ ルーチン......58
表 9-2. スタック アンワインド命令......59
表 10-1. C28x の DWARF3 レジスタ番号......66
表 10-2. C28x における FPU の DWARF3 レジスタ番号......67
表 10-3. TI ベンダー固有のタグ......68
表 10-4. TI ベンダー固有の属性......68
表 11-1. 登録済みベンダ......71
表 11-2. ELF 識別フィールド......71
表 11-3. ELF セクション タイプと TI セクション タイプ......72
表 11-4. C28x 特別なセクション......74
表 11-5. C28x 再配置タイプ......77
表 11-6. C28x 再配置操作......79
表 12-1. ELF 実行可能ファイルからプロセス イメージを作成する手順......82
表 12-2. 実行環境を初期化する手順......82
表 12-3. 終了手順......82

Copyright © 2026 Texas Instruments Incorporated

表 13-1. C28x ABI ビルド属性タグ....87
表 15-1. 改訂履歴....96

Copyright © 2026 Texas Instruments Incorporated

This page intentionally left blank.

Copyright © 2026 Texas Instruments Incorporated

Chapter 1
はじめに

このドキュメントは、テキサス インスツルメンツのプロセッサの C28x ファミリ向けの ELF ベースの組込みアプリケーション バイナリ インターフェイス (EABI) に関する仕様です。EABI は、プログラム、プログラム構成要素、実行環境 (存在する場合はオペレーティング システムを含む) との間の低レベル インターフェイスを定義する幅広い標準です。EABI の構成要素には、呼び出し規則、データ レイアウトおよびアドレッシング規則、オブジェクト ファイル フォーマットなどがあります。

この仕様の目的は、ツール プロバイダ、ソフトウェア プロバイダ、C28x のユーザーが、相互に互換性を持つツールやプログラムを構築できるようにすることです。

1.1 C28x 向けの **ABI**......10
1.2 範囲......11
1.3 ABI バリアント......12
1.4 ツールチェーンと相互運用性......12
1.5 ライブラリ......12
1.6 オブジェクト ファイルの形式......13
1.7 セグメント......13
1.8 C28x アーキテクチャの概要......13
1.9 C28x のメモリモデル......13
1.10 参考資料......13
1.11 コード フラグメント表記......14

Copyright © 2026 Texas Instruments Incorporated

1.1 C28x 向けの ABI

TI の C28x コンパイラ ツール リリース 18.12.0.LTS 以前に存在した唯一の C28x 向け ABI は、元々の COFF ベースの ABI でした。完全にベアメタル ABI であり、実行レベルのコンポーネントは存在しませんでした。

TI コンパイラ ツールのリリース 18.12.0.LTS では、C28x EABI と呼ばれる新しい ABI が導入されました。ELF オブジェクト ファイル フォーマットに基づいています。*IA-64 C++ ABI* や *ELF および動的リンクのための System V ABI* などの、業界標準モデルに由来します。データ レイアウトや呼び出し規則など、ABI のプロセッサ固有の側面は、いくつかの違いはあるものの、大部分は COFF ABI から変更されていません。言うまでもなく、COFF ABI と EABI は互換性がありません。すなわち、あるシステム内のすべてのコードは同じ ABI に従わなければなりません。TI のコンパイラ ツールは、新しい EABI と古い COFF ABI の両方をサポートしますが、将来的に COFF ABI のサポートが中止される可能性があるため、新しい ABI への移行を推奨します。

*プラットフォーム*とは、プログラムが実行されるソフトウェア環境のことです。ABI にはプラットフォーム固有の側面があり、特にプログラム セグメントの数と使用法、アドレッシング規則、可視性規則、プリエンプション、、プログラムのロードおよび初期化といった実行環境に関連する規制の分野に存在します。現在ベアメタルが唯一サポートされているプラットフォームです。*ベアメタル*という用語は、特定の環境が存在しないことを意味します。これは OS が存在できないという意味ではなく、単に OS 固有の ABI 仕様が存在しないということです。言い換えると、プログラムがどのようにロードされ実行されるか、システムの他の部分とどのように相互作用するかは、ベアメタル ABI の対象には含まれません。

ベアメタル ABI は多くの特定の側面において大きな可変性を許容します。たとえば、ある実装は位置の独立性 (PIC) を提供するかもしれませんが、特定のシステムで位置の独立性が必要でなければ、これらの規則は適用されません。この可変性のため、プログラムは ABI に準拠していても互換性がない場合があります。たとえば、あるプログラムが PIC を使用し、もう一方が使用しない場合、それらは相互運用できません。ツールチェーンはそのような非互換性への対応に取り組まなければなりません。

Copyright © 2026 Texas Instruments Incorporated

1.2 範囲

図 1-1 には ABI のコンポーネントとその関係が記載されています。図の下から上に向かってコンポーネントの概要が説明されています。また、この ABI 仕様の該当章への参照が記載されています。

下部領域のコンポーネントは、オブジェクト レベルの相互運用性に関連するものです。

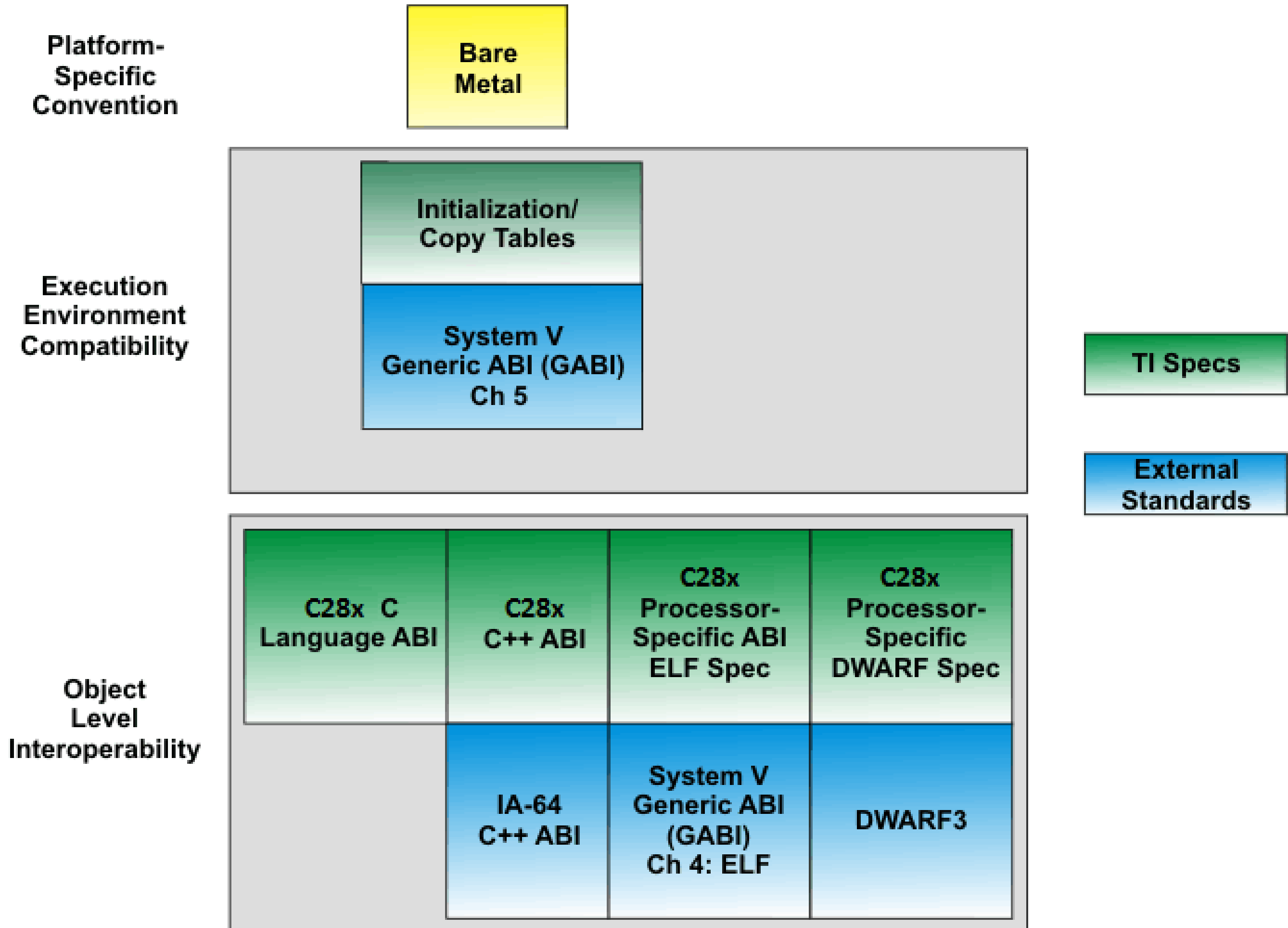

図 1-1. ABI 仕様の一部

C 言語 ABI (Chapter 2、Chapter 3、Chapter 4、Chapter 5、Chapter 6、Chapter 7) は、関数呼び出し規則、データ型の表現、アドレッシング規則、C ランタイム ライブラリへのインターフェイスを規定します。

C++ ABI (Chapter 8) は、C++ 言語がどのように実装されるかを規定します。これには、仮想関数テーブル、名前マングリング、コンストラクタがどのように呼び出されるか、例外処理のしくみ (Chapter 9) に関する詳細が含まれます。C28x C++ ABI は、一般的な IA-64 (Itanium) C++ ABI に基づいています。

DWARF コンポーネント (Chapter 10) は、オブジェクト レベルのデバッグ情報の表現を規定します。基本標準は DWARF3 標準です。この仕様は、プロセッサ固有の拡張について詳細を規定します。

ELF コンポーネント (Chapter 11) は、オブジェクト ファイルの表現を規定します。この仕様は、System V ABI 仕様をプロセッサ固有の情報によって拡張したものです。

ビルド属性 (Chapter 13) は、ターゲット デバイスの前提、メモリ モデル、ABI のバリアントなど、オブジェクト間の互換性に影響するさまざまなパラメータをオブジェクト ファイルにエンコードする手段を指します。ツールチェーンは、ビルド属性を使用することによって、互換性のないオブジェクト ファイルが結合またはロードされるのを防ぐことができます。

Copyright © 2026 Texas Instruments Incorporated

図の中央部分にあるコンポーネントは、実行時の相互運用性に関連します。

図 1-1 の上部にあるコンポーネントは、プログラム セグメントの数と使用方法、アドレッシング規則、可視性規則、プリエンプション、プログラムのロード、初期化など、実行環境と互換性を持つ実行可能ファイルの要件を定義するプラットフォーム固有の規則によって ABI を拡張します。ベアメタルとは、特定の環境が存在しないことを指します。

最後に、ABI の正式な一部ではありませんが、参考のため、また他のツールチェーンが任意で実装できるように、ここに文書化されている一連の仕様があります。

初期化 (Chapter 14) は、初期化済み変数がその初期値を取得するしくみを指します。通常、これらの変数は .data セクションに存在し、.data セクションがロードされるときに直接初期化されるため、ツールからの追加の関与を必要としません。ただし TI ツールチェーンは、.data セクションをオブジェクト ファイル内に圧縮形式でエンコードし、起動時に解凍するしくみをサポートしています。これは、圧縮されたコードやデータをオフライン ストレージ (ROM など) から実行アドレスにプログラム的にコピーする一般的なしくみを特別な用途に応用したものです。この機能を *コピー テーブル* と呼びます。初期化およびコピー テーブルのしくみは ABI の一部ではありませんが、他のツールチェーンが必要に応じてサポートできるように、ここに文書化されています。

1.3 ABI バリアント

前述したように、ABI はすべてのインスタンスで特定の動作を定義するものではなく、プラットフォームやシステム固有の変動を許容する一連の原則を示すものです。ABI には、使用可能される場合と使用されない場合があるモデルのバリアントがあります。ABI は、そのようなバリアントが使用される場合の実装方法を標準化します。バリアントの中には互換性のないものもあります。あるオブジェクトが特定のモデルを使用する場合、すべてのオブジェクトがそのモデルを使用する必要があります。このような場合、ツールチェーンはビルド属性を使用して、互換性のないオブジェクトが結合されないようにする必要があります。

- **ベア メタル - スタンドアロン**。このモデルは、単一で自己完結型の静的にリンクされた実行可能ファイルを指します。相互運用性の観点から最もシンプルな形式です。ABI の関連部分は、図 1-1 の下部にあるオブジェクトレベルのコンポーネントです。実行可能ファイルは静的にリンクされ、バインド (再配置) されるため、、位置独立性を必要としません。

1.4 ツールチェーンと相互運用性

この ABI は、特定のベンダのツールチェーンに固有ではありません。実際、その目的は、代替ツールチェーンが存在し、相互運用できるようにすることです。ABI は、メカニズムの実装方法を記述します。ツールチェーンがそれらをユーザー レベルでどのようにサポートするかについては記述しません。TI のツールが参照されている場合もありますが、説明のみを目的としています。ただし、TI の C28x コンパイラ ツールは、その性質上、シリコン ベンダに由来し、ABI 仕様と共同で開発されており、一部ではその基盤となっているため、特別なステータスを持っています。

TI ツールの動作がこの ABI と矛盾する場合、ツール内の欠陥とみなされます。そのような事例を見つけた場合は、欠陥レポートを support@tools.ti.com に送信してください。ただし、この仕様が不完全または不明確な場合は、TI ツールの動作を最終的な基準とみなすものとします。ABI 規格の主な目標は、TI ツールとの相互運用性です。ツールチェーン ベンダは、規格自体に欠陥やあいまいさがあっても、この目的を達成するように努力する必要があります。その場合はぜひご連絡ください。当社は仕様の明確化に努めております。

1.5 ライブラリ

一般に、ツールチェーンにはリンカに加えて、ツールチェーンが提供する言語サポートの一部を実装する標準ランタイムライブラリが含まれます。

C28x で使用されるライブラリ フォーマットは、一般的な GNU/SVR4 ar フォーマットです。

しばしば、リンカとライブラリには ABI の範囲外にある相互依存関係があります。たとえば、多くのリンカはさまざまなライブラリ コンポーネントの包含または除外を制御するために特別なシンボルを使用します。逆に、一部のライブラリはリンカによって定義される特別なシンボルを参照します。このため、リンカとライブラリは同じツールチェーンから提供されることが期待されます。異なるツールチェーンのリンカとライブラリを混在させて使用することは、この ABI ではサポートされません。これは、ツールチェーンに含まれる組込みライブラリにのみ適用されます。異なるツールチェーンでビルドされたアプリケーションライブラリはリンク可能です。

Copyright © 2026 Texas Instruments Incorporated

1.6 オブジェクト ファイルの形式

ELF 形式では、オブジェクト ファイルは以下のクラスに分類されます。

- **再配置可能**ファイルは、他のオブジェクト ファイルと静的リンクして実行可能ファイルを作成するのに適したコードとデータを保持します。
- **実行可能**ファイルは、実行に適したプログラムを保持します。

この仕様では、実行可能ファイル を指すために、*静的リンク ユニット*および *ロード モジュール*という用語を同義として使用します。

1.7 セグメント

ELF ロードモジュール (実行可能ファイル) は、*セグメント*の形でプログラムのメモリ イメージを表現します。この文脈において、セグメントとは共通のプロパティを持つ連続した不可分のメモリ範囲です。セグメントは、そのアドレスが決定されたときにバインドされます。これは、リンク時に静的に行われます。

1.8 C28x アーキテクチャの概要

C28x デバイスには 16 ビットおよび 32 ビットの CPU レジスタが備わっています。

C28x デバイスには 32 ビットのアドレス レジスタが備わっていますが、ほとんどの C28x デバイスは 22 ビットのアドレス空間しか持ちません。

C28x はリトル エンディアン モードでのみコンパイルされます。

C28x はワード単位でアドレス指定でき、ワードは 16 ビットです。

C28x デバイスには 8 ビットのオブジェクトはありません。このことは、C28x デバイスにおける ELF オブジェクト ファイル フォーマットの実装に特有の課題をもたらします。ELF ファイルの詳細については、「Chapter 11」を参照してください。

1.9 C28x のメモリモデル

C28x EABI がサポートするメモリ モデルは 1 種類のみで、統合型の大規模メモリ モデルです。このモデルでは、ポインタと ptrdiff_t 型はいずれも 32 ビットになります。

ポインタに関する詳細は、セクション 2.4 を参照してください。

1.10 参考資料

ドキュメント タイトル	リンクまたは URL
TMS320C28x 最適化 C/C++ コンパイラ ユーザー ガイド	SPRU514
TMS320C28x アセンブリ言語ツール ユーザー ガイド	SPRU513
TMS320C28x DSP CPU および命令セット リファレンス ガイド	SPRU430
ELF 仕様 - GABI 第 4 章および第 5 章	http://www.caldera.com/developers/gabi/2003-12-17/contents.html
IA64 (Itanium) C++ ABI	http://refspecs.linux-foundation.org/cxxabi-1.83.html
IA64 (Itanium) 例外処理 ABI	http://www.codesourcery.com/public/cxx-abi/abi-eh.html
ARM アーキテクチャのアプリケーション バイナリ インターフェイス	http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.abi/index.html
ARM アーキテクチャの C ライブラリ ABI	http://infocenter.arm.com/help/topic/com.arm.doc.ihi0039b/IHI0039B_clibabi.pdf
DWARF デバッグ フォーマット バージョン 3	http://dwarfstd.org/Dwarf3.pdf
C 言語標準	http://www.open-std.org/jtc1/sc22/wg14, ISO/IEC 9899:1990
C99 言語標準	http://www.open-std.org/jtc1/sc22/wg14, ISO/IEC 9899
C++ 言語標準	http://www.open-std.org/jtc1/sc22/wg21, ISO/IEC 14882:1998

Copyright © 2026 Texas Instruments Incorporated

1.11 コード フラグメント表記

このドキュメントでは、全体を通してアドレッシングや呼び出しシーケンスなどを説明するためにコード フラグメントを使用しています。フラグメントでは次の表記規則がよく使用されます。

sym	参照されているシンボル
label	コード アドレスを参照するシンボル
func	関数を参照するシンボル
tmp	一時レジスタ (tmp1、tmp2 なども)
reg, **reg1**, **reg2**	任意のレジスタ
dest	最終的な値またはアドレスの宛先レジスタ

アセンブラ組み込み演算子がいくつか導入されています。これらはさまざまなアドレッシング構造のための適切な再配置を生成するのに役立ち、一般的に自明です。

Copyright © 2026 Texas Instruments Incorporated

Chapter 2

データ表現

このセクションでは、標準的な C データ型のメモリおよびレジスタにおける表現について記述します。他の言語もサポートされる場合があり、その言語で使用される型は、それぞれがこれらの表現への対応を独自に定義します。

このセクションにおける記述と図において、ビット 0 は常に最下位ビットを指します。

2.1 基本型.......16
2.2 レジスタ内のデータ.......17
2.3 メモリ内のデータ.......17
2.4 ポインタ タイプ.......17
2.5 複素数型.......18
2.6 構造と共用体.......18
2.7 配列.......18
2.8 ビット フィールド.......19
2.9 列挙型.......20

Copyright © 2026 Texas Instruments Incorporated

2.1 基本型

整数値は 2 の補数表現を使用します。浮動小数点値は IEEE 754.1 表現を使用して表されます。浮動小数点演算は、ハードウェアがサポートする範囲で IEEE 754.1 に従います。

表 2-1 は C データ型のサイズと整列です (ビット単位)。

表 2-1. 標準型のデータ サイズ

タイプ	汎用的な名前	サイズ	整列
符号付き文字	schar	16	16
符号なし文字	uchar	16	16
char	修飾のない文字	16	16
ブール (C99)	uchar	16	16
_Bool (C99)	uchar	16	16
ブール (C++)	uchar	16	16
短整数、符号付き短整数	int16	16	16
符号なし短整数	uint16	16	16
整数、符号付き整数	int16	16	16
符号なし整数	uint16	16	16
長整数 、符号付き長整数	int32	32	32
符号なし長整数	uint32	32	32
長々整数、符号付き長々整数	int64	64	32
符号なし長々整数	uint64	64	32
列挙	--	変化 (「セクション 2.9」を参照)	32
フローティング	float32	32	32
精度浮動小数点	float64	64	32
拡張精度浮動小数点	float64	64	32
ポインタ	--	32	16

表中の汎用的な名前は、この仕様において、言語に依存しない方法で型を識別するために使用されます。

文字 型はデフォルトで符号なしです。符号の動作を指定する「符号付き文字」型および「符号なし文字」型とは対照的です。

整数型には補完的な符号なしのバリアントがあります。汎用的な名前には接頭辞「u」が付与されます (例:uint32)。

ブール 型は、0 の値を false、1 の値を true として使用します。その他の値は未定義です。

C、C99、C++ の追加の型は、標準型の同義語として定義されています。

```
typedef unsigned long    wchar_t;
          typedef unsigned long    wint_t;
          typedef char *           va_list;
```


Copyright © 2026 Texas Instruments Incorporated

2.2 レジスタ内のデータ

一般に、実装ではレジスタを自由に使用できます。このセクションで規定される標準的なレジスタ表現は、関数に渡される値または関数から返される値にのみ適用されます。

一部の構造オブジェクトもレジスタに収めることができます。詳細については「セクション 2.6」を参照。

レジスタ内の数値は常に右詰めされます。すなわち、レジスタのビット 0 が値の最下位ビットを保持します。16 ビット より小さい符号付き整数値は、レジスタの上位ビットに符号拡張されます。16 ビット より小さい符号なし値は、ゼロ拡張されます。

C28x にはサイズが異なるレジスタがあります。最も一般的に使用される CPU レジスタは 16 ビットまたは 32 ビットであり、使用するレジスタはデータのサイズによって異なります。

ACC レジスタ、P レジスタ、XT レジスタは 32 ビット データを保持できます。

- ACC は AH:AL のレジスタ ペアです。2 つの 16 ビット サブレジスタは独立してアクセスできます。
- P は PH:PL のレジスタ ペアです。2 つの 16 ビット サブレジスタは独立してアクセスできます。
- XT は T:TL です。T は独立してアクセス可能ですが、TL にはアクセスできません。

次のレジスタ ペアは 64 ビット データまたはポインタを保持できます：ACC:P、XAR1:XAR0、XAR3:XAR2、AR5:XAR4、XAR7:XAR6。これらのペアにおいて、最下位ビットはペアの 2 番目のレジスタに含まれます。

XAR0～XAR7 のレジスタは 32 ビットのポインタを保持できます。

AH、AL、T、PH、PL、および AR0～AR7 のレジスタは 16 ビットのデータを保持できます。

FPU をサポートするデバイスでは、R0～R7 のレジスタが 32 ビットの浮動小数点値を保持できます。

C28x レジスタについての詳細は、『*TMS320C28x CPU および命令セット リファレンス ガイド*』(SPRU430) を参照してください。

2.3 メモリ内のデータ

C28x はリトル エンディアン モードのみを使用します。エンディアンネスとは、マルチバイト値のメモリ レイアウトを指します。リトル エンディアン モードでは、値の最下位バイトが最も小さいアドレスに格納されます。エンディアンネスはオブジェクトのメモリ表現のみに影響し、レジスタ内のスカラー値はエンディアンにかかわらず常に同じ表現を持ちます。エンディアンネスは構造およびビット フィールドのレイアウトに影響し、それがレジスタ表現にも及びます。

スカラー変数は、その型に適したネイティブ命令を用いてロードおよびストアできるように整列されます：ワード用の MOV、ダブルワード用の MOVL。64 ビット型をロードまたはストアするネイティブ命令は存在しません。これらの命令は、メモリとの間でデータを移動するときにエンディアンネスを正しく処理します。

2.4 ポインタ タイプ

ポインタのデータ サイズは次のとおりです。

表 2-2. ポインタのデータ サイズ

タイプ	サイズ	保存	整列
関数ポインタ	32	32	32
データ ポインタ	32	32	32
size_t	32	32	32
ptrdiff_t	32	32	32

ポインタは 32 ビットとして格納されますが、コンパイラは、グローバル変数および関数のアドレスが 22 ビットの範囲内にあると仮定する必要があります。

English Document: SPRAC71
Copyright © 2026 Texas Instruments Incorporated

2.5 複素数型

C99 標準で定義された _Complex 型がサポートされています。内部表現は以下のとおりです。

```
struct _Complex
    { float_type real;
      float_type imag; };
```

2.6 構造と共用体

構造メンバーには、0 から順にオフセットが割り当てられます。各メンバーには、その整列制約を満たす中で最も小さい使用可能なオフセットが割り当てられます。この整列制約を満たすために、メンバー間にパディングが必要になる場合があります。

共用体のすべてのメンバーには、オフセット 0 が割り当てられます。

C++ のクラスの基本表現は構造です。このドキュメントの他の部分でも、*構造* という用語はクラスにも適用されます。

構造または共用体の整列要件は、そのメンバー (次のセクションで説明するビット フィールド コンテナを含む) の中で最も厳しい整列要件に等しくなります。メモリ内での構造または共用体のサイズは、その整列の倍数になるよう、最後のメンバーの後にパディングを挿入して切り上げられます。構造および共用体をスタック上で値渡しする場合には、特別な整列規則が適用されます (セクション 3.3 を参照)。

一般に、サイズが 32 ビット以下の構造は、関数に渡す場合や関数から返される場合に、レジスタまたはレジスタ ペアに格納されることがあります。このような構造は、R0H～R3H レジスタに値渡しされ、その後スタック上でも値渡しされます。1 つのフィールドの構造は、その基礎となるスカラー型に対応する値渡しで、渡されたり返されたりします。より大きな構造および共用体の参照による受け渡しについては、セクション 3.5 を参照してください。

FPU32 または FPU64 をサポートするデバイスでは、サイズが 128 ビット未満の浮動小数点型で統一された構造は値渡しされます。また、FPU64 をサポートするデバイスでは、64 ビットの倍精度浮動小数点型 (R0～R3) も値渡しされます。

リトル エンディアン モードでは、レジスタ内の構造は常に右詰めになります。すなわち、先頭バイトがレジスタの LSB (ペアの場合は偶数番レジスタ) に格納され、構造の以降のバイトがより上位のバイト位置に順に格納されます。C28x はリトル エンディアン モードのみを使用します。

2.7 配列

*配列*型のオブジェクトの最小整列は、その要素の型によって指定されます。

C62x、C67x	4 バイト
その他すべて	8 バイト

Copyright © 2026 Texas Instruments Incorporated

2.8 ビット フィールド

C28x EABI は、そのビット フィールドのレイアウトについて、IA64 C++ ABI のレイアウトを採用しています。以下の説明は、その標準に一致しています。ただし、明示的に示されている場合を除きます。

ビット フィールドの**宣言された型**は、ソース コードに記述されている型です。C および C++ の標準では、ビット フィールドの値を保持する際、その値を格納できる十分な大きさの *アドレッシング可能なストレージ ユニット* を割り当てることができます。その単位は、宣言された型と関係している必要はありません。アドレッシング可能なストレージ ユニットは、通常 *コンテナ型* と呼ばれており、本書でもその呼び方を用います。コンテナ型は、ビット フィールドがどのようにパックされ、整列されるかを決定する主要な要因です。

効率化のため、コンパイラは宣言された型やコンテナ型とは異なる型を使ってビット フィールドにアクセスすることがあります。宣言された型とコンテナ型は、ビット フィールドのパックおよび整列を決定するために厳密に使用されます。ビットフィールドを実際にロードするために使用される型は、アクセス型です。アクセス型は、ビット フィールドのサイズやオフセットに基づいて計算された、より狭い型になることがあります。たとえば、次の例では、コンテナ型は 32 ビットですが、ビット フィールドは 16 ビット アクセスを使用してロードされます。

```
struct S
    {
        long :16;
        long bf:16;
    };
```

C89、C99、C++ 言語の標準では、宣言された型の要件が異なります。

C89	整数、符号なし整数、符号付き数
C99	整数、符号なし整数、符号付き整数、_Bool または「実装定義によるその他の型」
C++	任意の整数型または列挙型 (ブールを含む)

厳密な C++ には *長々整数* 型は存在しませんが、C99 にはあるため、C++ コンパイラは通常これを拡張としてサポートしています。C99 標準は、ビット フィールドに対して長整数または長々整数の宣言された型をサポートすることを実装に要求していません。しかし、C++ ではそれが許可されているため、C コンパイラがそれらをサポートすることも珍しくありません。

ビット フィールドの値は、パディング ビットを除いて、そのコンテナ内に完全に格納されます。コンテナはそのタイプに合わせて適切に整列されます。そのフィールドを含む構造の整列は、コンテナの整列によって、その型のメンバー オブジェクトと同じように影響を受けます。これは無名フィールドにも適用されますが、ここが IA64 C++ ABI と異なる点です。コンテナは他のフィールドやオブジェクトを含む場合があり、また他のコンテナと重なる場合もありますが、1 つのフィールドに割り当てられたビット (サイズ超過フィールドに対するパディングを含む) が、他のフィールドのビットと重なることは決してありません。

C28x EABI において、ビット フィールドのコンテナ型は、その宣言された型ですが、1 つの例外があります。C++ では、いわゆるサイズ超過ビット フィールド、すなわち宣言された型よりも大きなサイズで宣言されたビット フィールドが許可されます。この場合、コンテナは、フィールドの宣言されたサイズを超えない最大の整数型となります。

レイアウト アルゴリズムは、ビット フィールドを割り当てるための開始点となる、*次に利用可能なビット* を保持します。レイアウト アルゴリズムにおける手順は次のとおりです。

1. 前述のとおり、コンテナ型 T を決定します。
2. 次に利用可能なビットを含む、型 T の適切に整列されたコンテナを C とします。C は以前に割り当てられたコンテナと重なる場合があります。
3. そのフィールドが C 内で、次に利用可能なビットから割り当てられる場合は、そのようにします。
4. 割り当てられない場合は、新しいコンテナを適切に整列された次のアドレスに割り当て、そこにフィールドを割り当てます。
5. サイズ超過フィールドに対するパディングを含めてビット フィールドのサイズを追加し、次に利用可能なビットを決定します。

Copyright © 2026 Texas Instruments Incorporated

リトル エンディアン モードでは、コンテナは LSB から MSB へと埋められます。C28x はリトル エンディアン モードのみを使用します。

ゼロ長のビット フィールドは、構造の次のメンバーを、その宣言された型に対応する次の整列境界に強制的に整列させ、構造の整列に影響を及ぼします。

修飾のない *整数* の宣言された型は、C28x EABI によって *符号付き整数* として扱われます。

2.8.1 volatile ビット フィールド

volatile ビット フィールドとは、C の *volatile* キーワードを使用して宣言されるビット フィールドです。volatile ビット フィールドを読み取る際には、そのコンテナ全体を適切なアクセス方式を用いて正確に 1 回だけ読み取る必要があります。

コンテナよりも小さいサイズの volatile ビット フィールドに書き込む場合も、そのコンテナは適切なアクセスを用いて、正確に 1 回読み取り、正確に 1 回書き込む必要があります。読み取りと書き込みは、互いに対してアトミックである必要はありません。

コンパイラは volatile ビット フィールドに対して、より狭い型を使用することはありません。代わりに、正確に宣言された型を使用します。

volatile ビット フィールドのサイズがコンテナ サイズと完全に等しい場合に書き込みを行ったとき、読み取りが発生するかどうかは未定義です。そのような読み取りは未定義であるため、両者がコンテナとまったく同じ幅を持つ volatile ビット フィールドに書き込む場合、異なる実装でコンパイルされたオブジェクト ファイルを相互リンクするのは安全ではありません。この理由から、外部インターフェイスで volatile ビット フィールドを使用することは避けるべきです。

同じ volatile ビット フィールドや、同一コンテナ内の追加の volatile ビット フィールドに対する複数のアクセスは、マージされない可能性があります。たとえば、volatile ビット フィールドのインクリメントは、常に 2 回の読み取りと 1 回の書き込みとして実装する必要があります。これらのルールは、たとえビット フィールドの幅や整列によって、より狭い型を使った効率的なアクセスが可能であっても、常に適用されます。書き込み操作では、たとえコンテナ全体の内容が完全に置き換わる場合でも、読み取りを行う必要があります。2 つの volatile ビット フィールドのコンテナが重複している場合、一方のビット フィールドへのアクセスは、他方へのアクセスも引き起こします。

「a」へのアクセスは「b」へのアクセスも引き起こしますが、その逆は発生しません。非 volatile ビット フィールドのコンテナが volatile ビット フィールドと重複している場合、非 volatile フィールドへのアクセスが volatile フィールドへのアクセスを引き起こすかどうかは未定義です。

2.9 列挙型

列挙型 (C 型 *enum*) は、ベースとなる積分型を使用して表現されます。通常、ベースとなる型は int か unsigned int ですが、どちらもすべての列挙子を表すことができない場合は除きます。この場合、long または unsigned long がすべての列挙子を表すことができる場合、その型が使用されます。それ以外の場合、ベースとなる型は long long または unsigned long long です。符号付きバージョンと符号なしバージョンの両方がすべての値を表すことができる場合、ABI では実装に代わる 2 つの選択肢の中から選択肢を残します(異なるツールチェーン間で一貫性を必要とするアプリケーションは、負の列挙子を宣言することで、署名された代替の選択肢を確実にすることができます)。

C 標準では、列挙定数が「signed int」型に収まる必要があります。そのため、enum 型は厳密 ANSI モードでは int または unsigned int のみになります。C++ ではより広い列挙型が可能です。TI コンパイラでは、緩和モードと GCC モードでより広い enum 型も使用できます。

Copyright © 2026 Texas Instruments Incorporated

Chapter 3

呼び出し規約

この章では、戻り値、レジスタ、引数の受け渡しの動作など、関数呼び出しの規則について説明します。

3.1 呼び出しと復帰.. 22
3.2 レジスタ規約.. 22
3.3 引数の受け渡し..25
3.4 戻り値.. 27
3.5 参照によって渡される構造と共用体と、返される構造と共用体..28
3.6 コンパイラ ヘルパー関数の規則..28
3.7 プロローグおよびエピローグのヘルパー関数..29
3.8 既知の関数のスクラッチ レジスタ..29
3.9 割り込み関数.. 29

Copyright © 2026 Texas Instruments Incorporated

3.1 呼び出しと復帰

関数呼び出しは専用の LCR 命令を使用して行われます。この命令は復帰アドレスを関数呼び出しスタックに積み上げた上で呼び出し先関数に分岐します。呼び出し先関数は専用の LRETR 命令を実行し、スタックから復帰アドレスを取り出して、そのアドレスに分岐します。

3.1.1 呼び出し命令

3.1.1.1 間接呼び出し

呼び出す関数がコンパイル時に不明な場合、すべてのアーキテクチャにおいて、その関数のアドレスは CPU レジスタ XAR7 に格納されます。この命令はアドレス空間全体に到達します。次に例を示します。

```
        LCR  *XAR7
```

3.1.1.2 直接呼び出し

呼び出される関数がコンパイル時に既知である場合、すべてのアーキテクチャは直接呼び出し命令を使用します。この命令ではイミディエイト、絶対、またはシンボリックのアドレッシング モードを使用できます。ここに示す例はイミディエイト アドレッシング モードのみを扱います。

C28x では LCR 命令を使用します。このアドレッシング モードは、有効なコード メモリ全域をカバーします。

```
        LCR #func    ; immediate mode, call func
```

3.1.2 復帰命令

呼び出し先関数は専用の LRET 命令を実行し、スタックから復帰アドレスを取り出して、そのアドレスに分岐します。

関数が割り込みハンドラ関数である場合は、代わりに IRET 命令が使用されます。

3.1.3 パイプライン規約

C28x パイプラインは保護されています。パイプラインのレイテンシや命令の完了について考慮する必要はありません (ただし、コードの性能向上には役立つ場合があります)。

3.1.4 弱関数

弱関数とは、シンボルがバインディング STB_WEAK を持つ関数です。プログラムは弱関数の定義が存在しなくても正常にリンクでき、その参照は未解決のまま残されます。

ABI はインポートされた弱関数の呼び出しをサポートします。つまり、異なる静的リンク ユニットで定義される可能性のある関数です。弱関数への参照がリンク時に未解決のまま残る場合、リンカはそのアドレスをゼロに置き換えます。ユーザーは弱関数を呼び出す前に、そのアドレスがゼロまたは NULL でないことを確認するチェックを追加する必要があります。

3.2 レジスタ規約

実装では、専用の特殊用途以外の目的で特殊用途レジスタを使用してはなりません。残りのレジスタは汎用レジスタです。

SP はコール スタック ポインタです。スタック ポインタは、ハンドコーディングされたアセンブリ関数内であっても、常に適切に整列されている必要があります (セクション 4.6.1 を参照)。TMS320C28x は 16 ビット ワードに合わせた整列が必要です。スタック管理とローカル フレーム構造については、セクション 4.6 で説明します。

ABI では、次のものを呼び出し先保存レジスタとして指定しています。つまり、呼び出された関数は、それらが関数から戻った時点で呼び出し時と同じ値を保持するように保存しておくことが求められます。

- XAR1-XAR3
- R4H ～ R7H (FPU32)
- R4L ～ R7L (FPU64)

Copyright © 2026 Texas Instruments Incorporated

その他すべてのレジスタは呼び出し元保存レジスタです。つまり、それらは呼び出し間で保持されないため、呼び出し後に値が必要な場合は、呼び出し元がそのコンテンツを保存して復元する責任を負います。

表 3-1. TMS320C28x レジスタ規約

登録	サイズ	説明
ACC	32 ビット	アキュームレータ
AH	16 ビット	ACC の上半分
AL	16 ビット	ACC の下半分
XAR0	16 ビット	補助レジスタ 0
XAR1	32 ビット	補助レジスタ 1
XAR2	32 ビット	補助レジスタ 2
XAR3	32 ビット	補助レジスタ 3
XAR4	32 ビット	補助レジスタ 4
XAR5	32 ビット	補助レジスタ 5
XAR6	32 ビット	補助レジスタ 6
XAR7	32 ビット	補助レジスタ 7
AR0	16 ビット	XAR0 の下半分
AR1	16 ビット	XAR1 の下半分
AR2	16 ビット	XAR2 の下半分
AR3	16 ビット	XAR3 の下半分
AR4	16 ビット	XAR4 の下半分
AR5	16 ビット	XAR5 の下半分
AR6	16 ビット	XAR6 の下半分
AR7	16 ビット	XAR7 の下半分
DP	16 ビット	データ-ページ ポインタ
IFR	16 ビット	割り込みフラグ レジスタ
IER	16 ビット	割り込みイネーブル レジスタ
DBGIER	16 ビット	デバッグ割り込みイネーブル レジスタ
P	32 ビット	積レジスタ
PH	16 ビット	P の上半分
PL	16 ビット	P の下半分
PC	22 ビット	プログラム カウンタ
RPC	22 ビット	復帰プログラム カウンタ
SP	16 ビット	スタック ポインタ
ST0	16 ビット	ステータス レジスタ 0
ST1	16 ビット	ステータス レジスタ 1
XT	32 ビット	被乗数レジスタ
T	16 ビット	XT の上半分
TL	16 ビット	XT の下半分

さらに、一部のデバイスには FPU 命令セットがあり、これによりレジスタ R0 ～ R7 が追加されます。

Copyright © 2026 Texas Instruments Incorporated

C28x レジスタを図 3-1 に示します。レジスタの詳細については、『*TMS320C28x DSP CPU および 命令セット リファレンス ガイド*』(SPRU430) を参照してください。

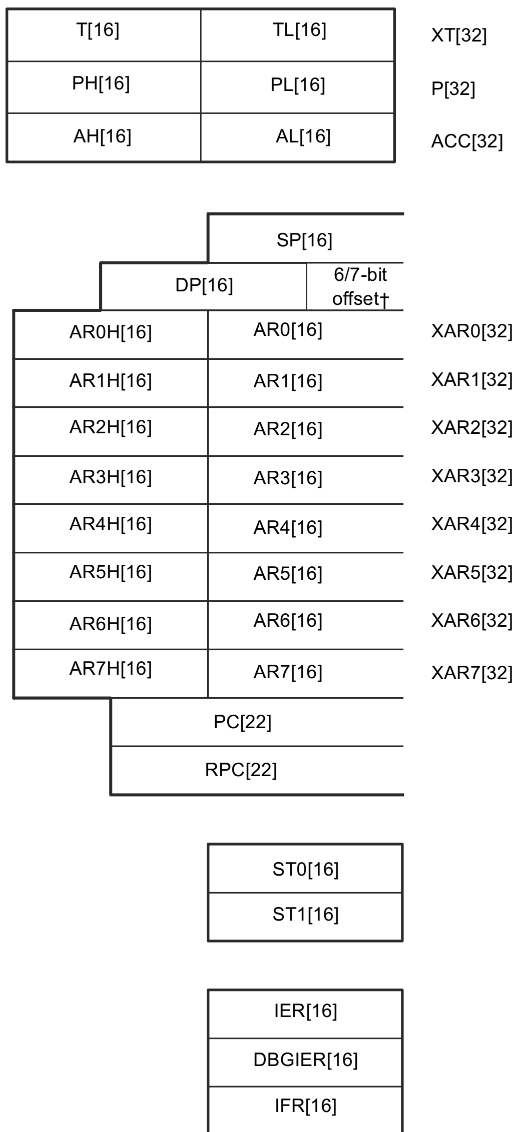

図 3-1. C28x のレジスタ

3.2.1 引数レジスタ

引数を渡すために使用できるレジスタには、R0～3 (FPU をサポートするデバイスにおける 32 ビット浮動小数点引数)、ACC:P (64 ビット引数)、ACC (32 ビット引数)、XAR4 および XAR5 (ポインタ引数)、AL および AH (16 ビット引数) があります。各型の最初の引数はその型用のレジスタに配置され、その型の残りの引数はスタックに配置されます。

FPU64 をサポートするデバイスでは、倍精度浮動小数点数 (64 ビット) はレジスタで渡されます。FPU64 をサポートしないデバイスでは、精度浮動小数点型は参照によって渡されます。

『*TMS320C28x C/C++ コンパイラの最適化ユーザー ガイド*』(SPRU514) の「関数で呼び出しを行う方法」を参照してください。

English Document: SPRAC71
Copyright © 2026 Texas Instruments Incorporated

3.2.2 呼び出し先保存レジスタ

呼び出された関数は、呼び出し先保存レジスタを必ず保持しなければなりません。これにより、関数からの復帰時に、それらのレジスタの値が呼び出し時と同じ状態に保たれます。

XAR1、XAR2、XAR3 の各レジスタは呼び出し先保存レジスタです。ターゲットが FPU をサポートしている場合には、R4H、R5H、R6H、R7H も呼び出し先保存レジスタとなります。

それ以外の汎用レジスタはすべて呼び出し元保存であり、呼び出しをまたいで値が保持されることはありません。そのため、呼び出し後に値が必要な場合は、呼び出し元がその内容を保存・復元する責任を持ちます。

3.3 引数の受け渡し

レジスタに渡される引数の数は、各引数のサイズと型によって異なります。引数は、宣言された順に、適切なサイズの利用可能なレジスタに割り当てられます。追加の引数はスタックに渡されます。『*TMS320C28x C/C++ コンパイラの最適化ユーザー ガイド*』(SPRU514) の「関数で呼び出しを行う方法」を参照してください。

3.3.1 16 ビット引数の受け渡し

1 つの CPU レジスタに収まる型の引数は、1 つの CPU レジスタに渡されます。つまり、16 ビットまでの型が 1 つのレジスタで渡されます。ポインタ型もサイズに関係なく 1 つのレジスタで渡されます。

ポインタ型は 32 ビット値として格納されますが、アドレス空間に対しては 22 ビットの制限を持つものとして扱う必要があります。

事例 1:

C のソース コード:

```
void func1(int a0, int a1, int a2, int a3);
int a0, a1, a2, a3;
void func2(void)
{
    func1(a0, a1, a2, a3);
}
```

コンパイル済みのアセンブリ コード:

```
MOVW      DP,#a0
MOV       AL,@a0
MOV       AH,@a1
MOVZ      AR4,@a2
MOVZ      AR5,@a3
; call instruction here
```


Copyright © 2026 Texas Instruments Incorporated

事例 2:

C のソース コード:

```
void func1(int *a0, int *a1, int *a2, int *a3);
int a0, a1, a2, a3;
void func2(void)
{
    func1(&a0, &a1, &a2, &a3);
}
```

コンパイル済みのアセンブリ コード:

```
    MOVL      XAR4,#a2
    MOVL      XAR5,#a1
    MOVL      *-SP[2],XAR4
    MOVL      XAR4,#a3
    MOVL      *-SP[4],XAR4
    MOVL      XAR4,#a0
    ; call instruction here
```

3.3.2 長い引数を渡す

32 ビットの ACC レジスタ (AH:AL) は long 型の引数を渡すために使用されます。

例:

C のソース コード:

```
void func1(int a0, long a1, int a2);
int a0, a2;
long a1;
func2(void)
{
    func1(a0, a1, a2);
}
```

コンパイル済みのアセンブリ コード:

```
    MOVW      DP,#a0
    MOVZ      AR4,@a0
    MOVL      ACC,@a1
    MOVZ      AR5,@a2
```

64 ビットの ACC:P レジスタペアは long long 型の引数を渡すために使用されます。

例:

C のソース コード:

```
void func1(long long a0);
long long a0;
func2(void)
{
    func1(a0);
}
```

コンパイル済みのアセンブリ コード:

```
    MOVW      DP,#a0
    MOVL      P,@a0
    MOVL      ACC,@a0+2
```


Copyright © 2026 Texas Instruments Incorporated

3.3.3 C++ における引数の受け渡し

C++ では、「this」ポインタは非静的メンバー関数の暗黙の最初の引数として XAR4 に渡されます。(非静的メンバー関数が参照によって構造を返す場合は、引数の順序が「&struct」、「this」となります)。

3.3.4 構造と共用体を渡す

32 以下の構造と共用体は値渡しされます。32 ビットを超える構造および共用体は、一般的に、参照渡しされます (セクション 3.5 を参照)。ただし、構造および共用体が、FPU32 または FPU64 を使用して値渡しされる追加のケースについては、セクション 2.6 を参照してください。

3.3.5 レジスタに渡されない引数のスタック レイアウト

レジスタに渡されない引数は、逆順でスタックに配置されます。各引数は、その型に対して正しく整列された次に利用可能なアドレスに配置され、以下の追加の考慮事項に従います。

- スカラーのスタック整列は、その宣言された型の整列と同じです。
- メンバーによって要求される整列に関係なく、値によって渡される構造体のスタック整列は、そのサイズ以上の最小の 2 のべき乗となります(値渡しされる構造に許容される最大サイズである 2 バイトを超えることはできません)。これは、型が十分に自然な整列になっていなくても、整列済みロードで引数を読み込めるようにするためです。たとえば、文字列の配列を含むサイズ 32 の構造の場合がそれにあたります。
- 各引数は、そのサイズをスタック整列の次の倍数に切り上げた分に等しいスタック領域を予約します。

可変長 C 関数 (すなわち、省略記号によって宣言され、可変個の引数で呼び出される関数) の場合、最後に明示的に宣言された引数と残りのすべての引数はスタックに渡され、そのスタック アドレスは宣言されていない引数にアクセスするための参照として機能します。

可変長関数に渡される整数より小さい未宣言のスカラー引数は、C 言語に従い、整数に昇格されて整数として渡されます。

スタックに渡された引数の間に整列の「ホール」が生じることがありますが、「バックフィル」は行われません。

3.3.6 フレーム ポインタ

C28x はフレーム ポインタを使用しません。これは、1 つのコール フレームを 0xfffe バイトに実質的に制限します。これは、任意の命令でサポートされる最小の SP オフセットです。

3.4 戻り値

関数の戻り値は、その型とサイズに基づいて、通常の第 1 引数レジスタと同じレジスタに格納されます。

- 16 ビットの結果は AL レジスタに返されます。
- 32 ビットの結果は ACC レジスタに返されます。
- 64 ビットの結果は ACC:P レジスタ ペアで返されます。
- 参照によって返される構造体は *XAR6 で返されます。
- FPU32 と FPU64 については、32 ビットの浮動小数点の結果が R0H で返されます。
- FPU64 では 64 ビットの double 結果が R0 で返されます。FPU64 がサポートされていない場合は、参照によって double 結果が返されます。

32 ビットを超える集成体は、参照によって返されます。

『*TMS320C28x Optimizing C/C++ コンパイラ ユーザー ガイド*』(SPRU514) の「呼び出し先関数の応答方法」を参照してください。

Copyright © 2026 Texas Instruments Incorporated

3.5 参照によって渡される構造と共用体と、返される構造と共用体

構造 (クラスを含む) や共用体が 32 ビットを超えるサイズである場合、それらは参照によって渡されたり、返されたりします。FPU32 または FPU64 を使用するときに、構造や共用体が値渡しされる追加のケースについては、セクション 2.6 を参照してください。

参照によって構造または共用体を渡す場合、呼び出し元は引数リスト内の位置に応じて、構造や共用体のアドレスをレジスタまたはスタックに格納します。C および C++ において必要とされる値渡しのセマンティクスを維持するため、呼び出し先は参照先オブジェクトの自分用のコピーを作成する必要はありません。ただし、呼び出し先がリーフ関数であり、参照先のオブジェクトを変更しない場合には、コピーを作成する必要はありません。

呼び出された関数が、64 ビットより大きい構造または共用体を返す場合、呼び出し元は、戻り値の格納先アドレスを格納した追加の引数を渡す必要があります。戻り値が使用されない場合には、NULL を渡します。

この追加の引数は、暗黙的な最初の引数として、最初の引数レジスタに渡されます。呼び出し先は、指定されたアドレスにオブジェクトをコピーして返します。必要に応じて、呼び出し元はメモリの割り当てを行います。通常はスタックに領域が予約されますが、場合によっては既存のオブジェクトのアドレスを渡すことができ、割り当ては不要となります。たとえば、f が構造を返す場合、s = f() の代入は、最初の引数レジスタ に &s を渡すことでコンパイルできます。

例

C のソース コード:

```
struct S { char big[100]; } g;
struct S accepts_and_returns_struct(struct S s)
{
    s.big[0] = 1;
    return s;
}
void caller(void)
{
    struct S w;
    w.big[0] = 0;
    g = accepts_and_returns_struct(w);
}
```

「下位レベル化された」C コード:(上位レベルの C コードがより下位レベルの C コードに変換されたもの)

```
struct S { char big[100]; } g;
void accepts_and_returns_struct(struct S *dst, struct S *sptr)
{
    struct S s;
    s = *sptr;
    s.big[0] = 1;
    if (dst) *dst = s;
}
void caller(void)
{
    struct S w;
    w.big[0] = 0;
    accepts_and_returns_struct(&g, &w);
}
```

3.6 コンパイラ ヘルパー関数の規則

ABI は、コンパイラが言語機能を実装するために使用する *ヘルパー関数* を規定します。一般に、これらの関数は標準的な呼び出し規則に従います。ヘルパー関数の一覧については、セクション 6.2 を参照してください。

Copyright © 2026 Texas Instruments Incorporated

3.7 プロローグおよびエピローグのヘルパー関数

以下のプロローグ関数とエピローグ関数は、コード サイズを削減するためのヘルパー関数として使用されます。各関数は、標準的な POP-and-RET 関数エピローグ シーケンスを実行します。コード サイズは、標準的な POP-and-RET エピローグ シーケンスをこれらの関数のいずれかへの分岐に置き換えることによって削減されます。各関数は、それが復元する連続レジスタの数にちなんで命名されています。

```
_prolog_c28x_1
_prolog_c28x_2
_prolog_c28x_3
_epilog_c28x_1
_epilog_c28x_2
```

--opt_for_space オプションは、プロローグやエピローグ コードのような共通のコード ブロックを、ランタイム ライブラリ内で定義された関数への呼び出しに置き換えることによって、手続きの抽象化を実行します。このため、--opt_for_space オプションを使用する場合は、提供されたランタイム ライブラリとリンクする必要があります。詳細については、『*TMS320C28x 最適化 C/C++ コンパイラ ユーザー ガイド*』(SPRU514) の「コード サイズの最適化を拡大する」を参照してください。なお、この手続きの抽象化は FPU レジスタをサポートしていません。

3.8 既知の関数のスクラッチ レジスタ

呼び出し元保存レジスタが呼び出しをまたいで生存しているが、呼び出し先がそのレジスタを変更しないことが分かっている場合、コンパイラは呼び出しの前後における保存および復元を省略することによって、呼び出し元関数のコードを最適化できます。これは、定義が既知の場合に生じます。

3.9 割り込み関数

割り込み関数 (つまり、ISR を実行するアセンブリ関数) では多数のレジスタを保存する必要があります。これらのレジスタには、AR1H、AR0H、XT、XAR4-XAR7 が含まれます。次のアセンブリ コードでは、FPU32 を*サポートしていない*デバイスに対して、適切なコンテキスト保存と C 環境固定アクションを実行します。

```
        ASP       ; [CPU_]
        PUSH      AR1H:AR0H             ; [CPU_]
        SPM       0                     ; [CPU_]
        MOVL      *SP++,XT              ; [CPU_]
        MOVL      *SP++,XAR4            ; [CPU_]
        MOVL      *SP++,XAR5            ; [CPU_]
        MOVL      *SP++,XAR6            ; [CPU_]
        MOVL      *SP++,XAR7            ; [CPU_]
        CLRC      PAGE0,OVM             ; [CPU_]
        CLRC      AMODE                 ; [CPU_]
        LCR       #||call||             ; [CPU_] |3|
        ; call occurs [#||call||] ; [] |3|
        MOVL      XAR7,*--SP            ; [CPU_]
        MOVL      XAR6,*--SP            ; [CPU_]
        MOVL      XAR5,*--SP            ; [CPU_]
        MOVL      XAR4,*--SP            ; [CPU_]
        MOVL      XT,*--SP              ; [CPU_]
        POP       AR1H:AR0H             ; [CPU_]
        NASP      ; [CPU_]
        IRET      ; [CPU_]
```


Copyright © 2026 Texas Instruments Incorporated

次のアセンブリ コードでは、FPU32 をサポートしているデバイスに対して、適切なコンテキスト保存と C 環境固定アクションを実行します。

```
ASP       ; [CPU_]
PUSH      RB                    ; [CPU_] x
PUSH      AR1H:AR0H             ; [CPU_]
MOVL      *SP++,XT              ; [CPU_]
MOVL      *SP++,XAR4            ; [CPU_]
MOVL      *SP++,XAR5            ; [CPU_]
MOVL      *SP++,XAR6            ; [CPU_]
MOVL      *SP++,XAR7            ; [CPU_]
MOV32     *SP++,STF             ; [CPU_] x
MOV32     *SP++,R0H             ; [CPU_]
MOV32     *SP++,R1H             ; [CPU_]
MOV32     *SP++,R2H             ; [CPU_]
MOV32     *SP++,R3H             ; [CPU_]
SETFLG    RNDF32=1, RNDF64=1    ; [CPU_]
SPM       0                     ; [CPU_]
CLRC      PAGE0,OVM             ; [CPU_]
CLRC      AMODE                 ; [CPU_]
LCR       #||call||             ; [CPU_] |3|
; call occurs [#||call||] ; [] |3|
MOV32     R3H,*--SP             ; [CPU_]
MOV32     R2H,*--SP             ; [CPU_]
MOV32     R1H,*--SP             ; [CPU_]
MOV32     R0H,*--SP             ; [CPU_]
MOV32     STF,*--SP             ; [CPU_]
MOVL      XAR7,*--SP            ; [CPU_]
MOVL      XAR6,*--SP            ; [CPU_]
MOVL      XAR5,*--SP            ; [CPU_]
MOVL      XAR4,*--SP            ; [CPU_]
MOVL      XT,*--SP              ; [CPU_]
POP       AR1H:AR0H             ; [CPU_]
POP       RB                    ; [CPU_]
NASP      ; [CPU_]
IRET      ; [CPU_]
```

割り込みでは、SR および PC レジスタをスタックにプッシュし、割り込みハンドラに分岐します。割り込み関数から復帰するには、この関数が特別命令 IRET を実行する必要があります。この命令は SR レジスタを復元し、割り込みが発生した PC に分岐します。

Copyright © 2026 Texas Instruments Incorporated

Chapter 4

データのアロケーションとアドレッシング

この章では、データの格納に関する規則について説明します。ABI によって定義されるデータ セクションは、図 4-1 に示されています。

4.1 データ セクションとセグメント.......... 32
4.2 データ ブロッキング.......... 33
4.3 アドレッシング モード.......... 33
4.4 静的データの割り当てとアドレッシング.......... 34
4.5 自動変数.......... 34
4.6 フレーム レイアウト.......... 35
4.7 ヒープ割り当て済みオブジェクト.......... 37

Copyright © 2026 Texas Instruments Incorporated

4.1 データ セクションとセグメント

コンパイラまたはアセンブラによって出力された再配置可能オブジェクト ファイルにおいて、変数はデフォルトの規則およびコンパイラ ディレクティブを用いてセクションに割り当てられます。セクションは、再配置可能ファイルにおけるアロケーションの不可分単位です。セクションには、しばしば類似の特性を持つオブジェクトが含まれます。セクションが初期化されているかどうか、書き込み可能か読み取り専用か、どのようにアドレス指定されるか、およびどの種類のデータを含むかに応じて、さまざまなセクションがデータ用に指定されます。

静的変数をセクションに配置する方法や、それらがどのようにアドレス指定されるかの規約については、セクション 4.4.2 に記載されています。

リンカはオブジェクト ファイルからセクションを組み合わせ、ELF ロード モジュール (実行可能ファイル) 内にセグメントを形成します。セグメントとは、ロード モジュールに割り当てられた連続したメモリ領域であり、プログラムの実行イメージの一部を表します。

ロード モジュールは、1 つ以上のデータ セグメントを含むことがあり、その中にリンカがスタック、ヒープ、静的変数を割り当てます。項目は 1 つのセグメントまたは複数のセグメントにまとめることができます。ただし、以下の制限が適用されます。

- セグメント内では、初期化済みデータは未初期化データよりも前に置かれなければなりません。これは ELF の構造上の制約です。
- プラットフォーム固有の規約によって課される追加の制約。

ランタイム環境は、スタックやヒープなどの項目に領域を割り当てるために、未初期化データ セグメントを動的に割り当てたりリサイズしたりできます。

ABI によって定義されたデータ セクションと、セクションをセグメントに抽象的にマッピングしたものを 図 4-1 に示します。このマッピングはあくまで代表的なものであり、具体的な構成はプラットフォームやシステムによって異なる場合があります。初期化済みセクションは青色で表示され、未初期化セクションは灰色で表示されます。

Data Sections		
.data	Initialized read-write data	RAM
.bss	Uninitialized read-write data	
.stack	Program stack	
.sysmem	Dynamic data (heap)	

Read-Only Sections		
.const	Far const (read-only) data	ROM or RAM

図 4-1. データ セクションとセグメント (標準)

.const セクションは読み取り専用の定数が含まれています。.const セクションは読み取り専用メモリに配置される場合があり、その場合は絶対アドレッシングで参照されることがあります。

.data セクションには、初期化済みの読み書き可能データが含まれます。

.bss セクションには、未初期化の読み取り / 書き込み可能データが含まれます。

リンカ コマンド ファイルによって配置される追加の特殊セクションは、セクション 11.3.5 に一覧として示されています。

Copyright © 2026 Texas Instruments Incorporated

4.2 データ ブロッキング

ブロッキングにより、オブジェクトがページ内に完全に収まるようにしたり、ページ境界上で開始したりすることができます。データ ブロッキングによって、コンパイラで不要な DP ロードの数を減らすことができます。

C28x EABI のデフォルトのブロッキング ルールは次のとおりです。

- 配列とそのセクションはブロック*されません*。
- スカラーとそのセクションはブロックされます。
- 外部リンケージ (C では外部) がある構造体はブロックされます。
- 内部リンケージ (C ではスタティック) がある構造体はブロック*されません*が、それらのセクションはブロックされます。
- 初期化されていないデータ、初期化されているデータ、const データはブロックされます。

ただし、データ ページ ブロッキングでは、データがページ境界に合わせて整列するために、メモリに整列ホールが生じることがあります。そのため、コード サイズと速度の最適化に関するアプリケーションのニーズとデータ サイズの最適化に関するニーズの間でトレードオフが発生します。**blocked** および **noblocked** データ属性を使用して、特定の変数におけるブロッキングを制御できます。『*TMS320C28x C/C++ コンパイラ最適化ユーザ ガイド*』(SPRU514) の「データページ (DP) ポインタ ロード最適化」と「変数属性」を参照してください。

4.3 アドレッシング モード

C28x デバイスは、さまざまなアセンブリ コードのアドレッシング モードを使用します。ここではこれらのモードを簡単に挙げておきますが、詳細は『*TMS320C28x DSP CPU と命令セット参照ガイド*』(SPRU430) の「アドレッシング モード」の章で説明されています。

C28x EABI 呼び出し規則では、ステータス レジスタ (ST1) のアドレス モード ビット (AMODE) を 0 に設定することが要求されます。これはデフォルト値です。AMODE=0 を設定すると、許可されるアドレッシング モードの集合は制限されますが、SP 相対アドレッシングが可能となります。

アセンブリ コードでは AMODE=1 が設定される場合があるため、リンカは 7 ビット直接アドレッシングを処理できるように準備しておく必要があります。

.

表 4-1. TMS320C28x アドレッシング モード

モード名	アセンブリの例	再配置タイプ	注
直接アドレッシング	MOV AL, @var	DP 相対再配置	
スタック アドレッシング	MOV AL,*-SP[4]	再配置なし	
間接アドレッシング	MOV AL,*XAR3	再配置なし	
レジスタ アドレッシング	MOV ACC, @T	再配置なし	メモリ マップト レジスタ移動
データ イミディエイト アドレッシング	MOV AL, *(0:var)	絶対再配置	
プログラム イミディエイト アドレッシング	MAC P, loc16, 0:pma	絶対再配置	loc16 は完全に独立したメモリ オペランド
I/O イミディエイト アドレッシング	OUT *(addr), loc16	絶対再配置	loc16 は完全に独立したメモリ オペランド

Copyright © 2026 Texas Instruments Incorporated

4.4 静的データの割り当てとアドレッシング

自動変数または動的変数でないすべての変数は静的データと見なされます。すなわち、C のストレージ クラス *extern* または *static* を持ち、そのアドレスが (静的) リンク時に確立される変数です。これらの変数は特性に応じてさまざまなセクションに割り当てられ、その後 1 つ以上の静的データ セグメントに結合されます。

静的変数を含む追加のデータ セグメントは *絶対データ セグメント*と呼ばれ、絶対アドレッシングのアドレッシングのアドレッシングを用いて参照されます。それらの数、大きさ、配置には制限はありません。

4.4.1 静的データのアドレッシング方法

この ABI は静的データに対しては絶対形式のアドレッシングのみをサポートします。

4.4.2 静的データの配置規則

ツールチェーン間の互換性を確保するには、一方が生成したアドレッシングが、もう一方が生成した配置と一貫性を持つ必要があります。これは特に アドレッシングに関して重要です。

このため、ABI はいくつかの規則を定める必要があります。これらの規則の一部は、ツールチェーン固有の動作 (対応しているコード生成モデルなど) や、ユーザーの行動 (コマンド ライン オプションの選択や言語拡張の適用など) に依存します。そのため、ABI は 2 つの柱からなるアプローチを採用しています。

- 互換性を確保するため、ABI は配置とアドレッシングに関してアブストラクト規定を定義し、それをツールチェーン固有の方法で対応付けています。これらの規則により、異なるツールチェーンで互換性のあるオブジェクトファイルを構築できるようになりますが、具体的な構築方法を正確に規定することはできません。
- 一貫性を保つために、ABI は、アドレッシングの制約を満たす形でプログラムをリンクするか、リンクを拒否するようにリンカに要求します。

アドレッシングを生成するツールチェーンは、変数の定義ではなく宣言しか参照できない場合があります。したがって、規則はその両方で利用可能な情報にのみ基づいている必要があります。たとえば、配列の次元はその対象外となります。

4.4.2.1 アドレッシングに関するアブストラクト規定

すべての変数は絶対アドレッシング (位置依存) の範囲内にあります。位置独立アドレッシングはサポートされていません。

4.4.3 静的データの初期化

初期値がゼロでない静的変数は、初期化されたデータ セクションに割り当てる必要があります。セクションの内容は、セクション内のすべての変数の初期値に対応するメモリの内容のイメージである必要があります。したがって、セクションがメモリにロードされると、変数はその初期値を直接取得します。これは、ほとんどの ELF ベースのツールチェーンで使用される、いわゆる*直接初期化モデル*です。

ゼロに初期化されることが想定される変数は未初期化セクションに割り当てることができます。ローダは、データ セグメントの末尾にある未初期化領域をゼロに設定します。

初期化された変数を直接エンコードするためにコンパイラが必要ですが、リンカは必要ではありません。リンカでは、オブジェクトファイル内の直接エンコードされた初期化済みセクションを実行可能ファイル用のエンコードされたフォーマットに変換し、ライブラリ関数に依存して情報をデコードして、プログラムの起動時に初期化を実行できます(リンカはライブラリが同じツールチェーンにあると仮定できる点を思い出してください)。初期化データをエンコードすることで、実行可能ファイルのスペースを節約できます。また、ローダに依存しない自己起動 ROM ベース システム向けの初期化メカニズムも提供します。Chapter 14 で説明しているように、TI ツールチェーンがこのようなメカニズムを実装します。他のツールチェーンは、互換性のあるメカニズムを採用することも、異なるメカニズムを採用することもあれば、まったく採用しないこともあります。

4.5 自動変数

プロシージャのローカル変数、すなわち C のストレージ クラス *auto* を持つ変数は、コンパイラの裁量によりスタックまたはレジスタに割り当てられます。スタック上の変数は、スタック ポインタ (SP) のを介してアドレッシングされます。

English Document: SPRAC71
Copyright © 2026 Texas Instruments Incorporated

スタックは .stack セクションから割り当てられ、プログラムのデータ セグメントの一部になります。

スタックは下位アドレスから上位アドレスに向かって拡張されます。スタック ポインタは常に 2 ワード (32 ビット) の境界に整列する必要があります。SP は次に未使用のメモリ位置を指します。

セクション 4.6 には、スタック規則およびローカル フレーム構造についての詳細が記載されています。

4.6 フレーム レイアウト

ローカル フレーム用の標準化されたレイアウトと、呼び出し先退避レジスタの順序が必要なケースは少なくとも 2 つあります。それが例外処理とデバッグです。

このセクションでは、スタックの管理規則、フレームの一般的なレイアウト、および呼び出し先退避領域のレイアウトについて説明します。

スタックはゼロから上位のアドレスに向かって増加します。SP はを指します。つまり、*(SP+4) の単語が割り当てられますが、*SP は次の未使用メモリの場所ではありません。

フレーム内のオブジェクトには、正のオフセットがある SP 相対アドレッシングを使用してアクセスします。

コンパイラは、フレームにアクセスするために 1 つ以上の「フレーム ポインタ」レジスタを自由に割り当てることができます。TI コンパイラでは ではフレーム ポインタを使用しないため、1 つの呼び出しフレームは 0xffff バイトに制限されます。

フレーム ポインタが関数間のリンケージの一部ではない限り、フレーム ポインタを使用するかどうか、どのレジスタを使用するか、どこを指すかの選択はツールチェーンの裁量に委ねられます。ただし、スタック巻き戻し命令を処理する例外処理では、フレーム ポインタが利用できないことを前提としています。

関数のスタック フレームには次の領域が含まれます。

- スタック上で渡される**着信引数**は呼び出し元のフレームの一部です。
- **呼び出し先退避領域**には、保持する必要がある関数によって変更されたレジスタが格納されます。例外またはデバッグが有効になっている場合は、特定のレイアウトに従う必要があります。そうでない場合、コンパイラはレジスタを保存するための代替スキームを自由に使用できます。
- **ローカルおよびスピル テンプ** エリアは、関数によって使用される一時ストレージで構成されています。
- **発信引数**セクションは、セクション 3.3 で説明されているように、レジスタ以外の引数を呼び出した関数に渡すためのものです。セクションのサイズは 1 つの呼び出しに必要な最大値です。

Copyright © 2026 Texas Instruments Incorporated

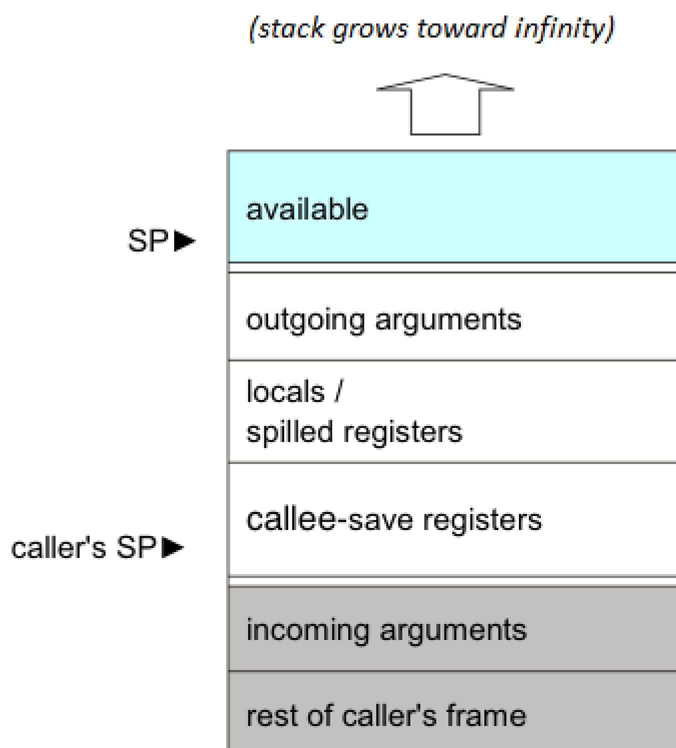

図 4-2. ローカル フレーム レイアウト

フレームが割り当てられる前に、SP は復帰アドレス (割り込み関数の SR) を指します。

4.6.1 スタックの整列

スタック ポインタ (SP) は 2 ワード (32 ビット) に整列されています。スタックは成長するにつれてアドレスが増加します。

C++ 例外を伝播する可能性のある C/C++ 呼び出し可能関数においては、スタックは常に 32 ビットにアラインされていなければなりません。アセンブリ関数は、任意の C 呼び出し可能な関数を呼び出す前に、(ASP 命令を使用して) SP を整列させる必要があります。

4.6.2 レジスタ保存順序

セクション 3.2 において議論されたように、関数は *呼び出し先保存* と指定されたレジスタの内容を保持する責任があります。これは通常、関数を開始する際に、変更されたレジスタをローカル フレームに保存し、関数を終了する前に復元することで実現されます。通常、呼び出し先保存レジスタのスタック上での順序や位置は重要ではなく、保存された場所と同じ場所から復元されれば問題はありません。ほとんどの場合、コンパイラは任意の順序でレジスタを保存します。ただし、次のように既知の順序が必要となる機能もあります。

- **安全なデバッグの順序**。レジスタは、次の順序で保存されます。XAR1-XAR3 の詳細を示します。次に、ターゲットが FPU をサポートしている場合、レジスタ R4L、R4H、..R7L、R7H が保存されます。
- **例外処理**。例外処理のためのスタック アンワインド処理では、関数エピローグをシミュレートできるように、各レジスタが正確にどこにあるかを知る必要があります。この情報をビット ベクトルを使用して効率的にエンコードするために、固定順序が定義されました。例外処理では、ビット ベクトルのエンコードに *呼び出し先保存レジスタの安全なデバッグ順序* が再利用されるため、順序は同じです。

コンパイラは常にフレームの底 (最上位アドレス) から順序どおりにレジスタを保存します。レジスタが保存されない場合、スタック内にホールが生じないようにレジスタはパックされますが、相対的な順序は同じままです。

English Document: SPRAC71
Copyright © 2026 Texas Instruments Incorporated

4.7 ヒープ割り当て済みオブジェクト

C の malloc() または C++ の演算子「new」によって動的に割り当てられるオブジェクトは、ランタイム ライブラリによって割り当てられます。実行環境は、言語標準によって指定された API に準拠している限り、これらの関数の独自の実装を提供できます。この ABI では動的割り当てメカニズムに関して追加の要件は規定されていません。

Copyright © 2026 Texas Instruments Incorporated

Chapter 5

コードのアロケーションとアドレッシング

コンパイラやアセンブラは、1 つ以上のセクションにコードを生成します。デフォルトのコード セクションは .text と呼ばれますが、プログラマはコードを追加の名前付きセクションに配置することもできます。リンカはこれらのコード セクションを 1 つ以上のセグメントにまとめます。ベース ABI では、コード セクションの数・サイズ・配置に制約は設けていませんが、プラットフォーム固有の制約が課される場合があります。命令は 16 ビットから 64 ビットまでの可変長であり、16 の正確な倍数となります。

5.1 コード ラベルのアドレスの計算 39
5.2 呼び出し 39

Copyright © 2026 Texas Instruments Incorporated

5.1 コード ラベルのアドレスの計算

アセンブリ コード セクションでは、以下の目的でコード アドレスを計算する必要があります。

- 呼び出しまたは分岐を実行する
- 関数ポインタを作成する
- スイッチ テーブルに入力する

アドレスを指定するためのモードの概要は セクション 4.3 にまとめられており、『*TMS320C28x DSP CPU および命令セット リファレンス ガイド*』(SPRU430) の「アドレッシング モード」セクションで詳しく説明しています。

5.2 呼び出し

関数呼び出しは専用の LCR 命令を使用して行われます。この命令は復帰アドレスを関数呼び出しスタックに積み上げた上で呼び出し先関数に分岐します。呼び出された関数は、専用の LRETR 命令を実行して復帰します。RETLRETR 命令はスタックから復帰アドレスを取り出し、そのアドレスに分岐します。

```
        LCR   funcname
```

5.2.1 直接 呼び出し

直接呼び出しのターゲット関数が、直接 CALL 命令のオフセットでは到達できない位置に配置されている場合、静的リンカは CALL 命令を書き換え、代わりに *トランポリン* と呼ばれるヘルパー スタブ関数を呼び出すようにします。トランポリンは単にターゲット関数を呼び出します。リンカは CALL 命令の到達範囲内にトランポリンを配置する責任を負います。

5.2.2 ファー呼び出しトランポリン

C28x のアドレス空間全体は直接呼び出しによって到達可能です。したがって、トランポリンは使用されません。

5.2.3 間接呼び出し

関数ポインタを介した間接呼び出しでは、レジスタ オペランドがある分岐が生成されます。次に例を示します。

```
        LCR   *XAR7                 ; indirect call
```


Copyright © 2026 Texas Instruments Incorporated

Chapter 6
ヘルパー関数 API

1 つのツールチェーンで構築されたオブジェクト ファイルを別のツールチェーンのランタイム サポート (RTS) ライブラリとリンクできるようにするためには、両者の間の API を規定する必要があります。インターフェイスには 2 つの部分があります。1 つ目は、命令セットが直接サポートしていない言語機能を実現するために、コンパイラが利用する関数の定義です。これらは *ヘルパー関数* と呼ばれ、このセクションで取り上げられています。2 つ目は、C、C99、C++ 標準ライブラリといったソース言語ライブラリ標準のコンパイル時の側面を標準化するものであり、これは別のセクションで扱われます。

6.1 浮動小数点の動作....................41
6.2 C ヘルパー関数 **API**....................41
6.3 C99 用の浮動小数点ヘルパー関数....................43

Copyright © 2026 Texas Instruments Incorporated

6.1 浮動小数点の動作

浮動小数点の動作はデバイスやツールチェーンによって異なるため、標準化が困難です。ABI の目標は、C、C99、C++ の各標準への準拠の基礎を提供することです。これらのうち C99 は、浮動小数点に関して最適に規定されています。C99 標準の付録 F では、C 言語の動作の浮動小数点の動作を IEEE 浮動小数点標準 (ISO IEC 60559:1989、以前は ANSI/IEEE 754−1985 として指定) に基づいて定義しています。

C28x ABI では、浮動小数点値で動作するこのセクションのヘルパー関数が、C99 標準の付録 F で指定されている動作に準拠している必要があると規定しています。

C99 では、<fenv.h> ヘッダ ファイルを使用して、浮動小数点動作環境のカスタマイズとアクセスが可能です。ヘルパー関数の動作を標準化する目的で、ABI では、次のプロパティを使用して、基本的なデフォルト環境に従って動作するよう規定しています。

- 丸めモードは最も近い値に丸められます。動的丸め精度モードはサポートされていません。
- 浮動小数点例外はサポートされていません。
- シグナリング NaN を表す入力はクワイエット NaN のように動作します。
- ヘルパー関数では、FENV_ACCESS *off* 状態での動作のみをサポートしています。つまり、プログラムはノンストップモードで実行され、浮動小数点環境にはアクセスしないことを前提としています。

ツールチェーンは独自のライブラリを用いて、より包括的な浮動小数点サポートを自由に実装できます。ツールチェーン固有の浮動小数点サポートを呼び出すユーザは、そのツールチェーンのライブラリを使用したリンクが必要になる場合があります (ABI 準拠のヘルパー関数ライブラリに加えて)。

6.2 C ヘルパー関数 API

コンパイラは、専用のハードウェアを持たないデバイスでの浮動小数点演算など、コンパイラでサポートする必要があるもののアーキテクチャでは直接サポートされていない演算を実行するために、ヘルパー関数の呼び出しを生成します。これらのヘルパー関数は、ABI に準拠するツールチェーンの RTS ライブラリに実装する必要があります。

ヘルパー関数には _ _C28x_ という接頭辞が付けられます。この接頭辞を持つ識別子は ABI 用に予約されています。

ヘルパー関数は、 に示されている場合を除き、標準の呼び出し規約に準拠します。

次の表では、C 表記法と構文を使用してヘルパー関数を指定しています。表の型は、セクション 2.1 で指定されている汎用データ型に対応しています。

表 6-1 の関数は、さまざまな数学演算、論理演算、比較演算を行います。

表 6-1. TMS320C28x EABI 関数

シグネチャ	説明
__c28xabi_absll	long long int の絶対値を返します。
__c28xabi_addd	2 つの倍精度浮動小数点数を加算します。
__c28xabi_addf	2 つの単精度浮動小数点数を加算します。
__c28xabi_andll	2 つの long long 整数値にビット単位の AND 演算を行います。
__c28xabi_cmpd	2 つの倍精度浮動小数点数を比較します。
__c28xabi_cmpf	2 つの単精度浮動小数点数を比較します。
__c28xabi_cmpll	2 つの符号付き long long int 値を比較します。
__c28xabi_cmpull	2 つの符号なし long long int 値を比較します。
__c28xabi_divd	2 つの倍精度浮動小数点数を除算します。
__c28xabi_divf	2 つの単精度浮動小数点数を除算します。
__c28xabi_divi	2 つの符号付き 16 ビット整数を除算します。
__c28xabi_divl	2 つの符号付き 32 ビット整数を除算します。
__c28xabi_divll	2 つの符号付き 64 ビット long long 整数を除算します。
__c28xabi_divu	2 つの符号なし 16 ビット整数を除算します。

Copyright © 2026 Texas Instruments Incorporated

表 6-1. TMS320C28x EABI 関数 (続き)

シグネチャ	説明
__c28xabi_divul	2 つの符号なし 32 ビット整数を除算します。
__c28xabi_divull	2 つの符号なし 64 ビット long long 整数を除算します。
__c28xabi_dtof	倍精度浮動小数点数を単精度浮動小数点数に変換します。
__c28xabi_dtoi	倍精度浮動小数点数を符号付き 16 ビット整数に変換します。
__c28xabi_dtol	倍精度浮動小数点数を符号付き 32 ビット整数に変換します。
__c28xabi_dtoll	倍精度浮動小数点数を符号付き 64 ビット long long 整数に変換します。
__c28xabi_dtou	倍精度浮動小数点数を符号なし 16 ビット整数に変換します。
__c28xabi_dtoul	倍精度浮動小数点数を符号なし 32 ビット整数に変換します。
__c28xabi_dtoull	倍精度浮動小数点数を符号なし 64 ビット long long 整数に変換します。
__c28xabi_ftod	単精度浮動小数点数を倍精度浮動小数点数に変換します。
__c28xabi_ftoi	単精度浮動小数点数を符号付き 16 ビット整数に変換します。
__c28xabi_ftol	単精度浮動小数点数を符号付き 32 ビット整数に変換します。
__c28xabi_ftoll	単精度浮動小数点数を符号付き 64 ビット long long 整数に変換します。
__c28xabi_ftou	単精度浮動小数点数を符号なし 16 ビット整数に変換します。
__c28xabi_ftoul	単精度浮動小数点数を符号なし 32 ビット整数に変換します。
__c28xabi_ftoull	単精度浮動小数点数を符号なし 64 ビット long long 整数に変換します。
__c28xabi_itod	符号付き 16 ビット整数を倍精度浮動小数点数に変換します。
__c28xabi_itof	符号付き 16 ビット整数を単精度浮動小数点数に変換します。
__c28xabi_lltod	符号付き 64 ビット整数を倍精度浮動小数点数に変換します。
__c28xabi_lltof	符号付き 64 ビット整数を単精度浮動小数点数に変換します。
__c28xabi_ltod	符号付き 32 ビット整数を倍精度浮動小数点数に変換します。
__c28xabi_ltof	符号付き 32 ビット整数を単精度浮動小数点数に変換します。
__c28xabi_modi	符号付き 16 ビット整数の除算の余りを計算します。
__c28xabi_modl	符号付き 32 ビット整数の除算の余りを計算します。
__c28xabi_modll	符号付き 64 ビット long long 整数の除算の余りを計算します。
__c28xabi_modu	符号なし 16 ビット整数の除算の余りを計算します。
__c28xabi_modul	符号なし 32 ビット整数の除算の余りを計算します。
__c28xabi_modull	符号なし 64 ビット long long 整数の除算の余りを計算します。
__c28xabi_mpyd	2 つの倍精度浮動小数点数を乗算します。
__c28xabi_mpyf	2 つの単精度浮動小数点数を乗算します。
__c28xabi_mpyll	2 つの符号付き 64 ビット long long 整数を乗算します。
__c28xabi_negd	倍精度浮動小数点数の符号を反転します。
__c28xabi_negf	単精度浮動小数点数の符号を反転します。
__c28xabi_orll	2 つの long long 整数値にビット単位の OR 演算を行います。
__c28xabi_subd	ある倍精度浮動小数点数から別の倍精度浮動小数点数を減算します。
__c28xabi_subf	ある単精度浮動小数点数から別の単精度浮動小数点数を減算します。
__c28xabi_ulltod	符号なし 64 ビット long long 整数を倍精度浮動小数点数に変換します。
__c28xabi_ulltof	符号なし 64 ビット long long 整数を単精度浮動小数点数に変換します。
__c28xabi_ultod	符号なし 32 ビット整数を倍精度浮動小数点数に変換します。
__c28xabi_ultof	符号なし 32 ビット整数を単精度浮動小数点数に変換します。
__c28xabi_utod	符号なし 16 ビット整数を倍精度浮動小数点数に変換します。
__c28xabi_utof	符号なし 16 ビット整数を単精度浮動小数点数に変換します。
__c28xabi_xorll	2 つの long long 整数値にビット単位の XOR 演算を行います。

Copyright © 2026 Texas Instruments Incorporated

6.3 C99 用の浮動小数点ヘルパー関数

これらの関数は未実装ですが、その名前は C99 コンパイラが使用できるように予約されています。TI ライブラリは現在、これらの関数を実装していません。C99 に関連する API は変更される可能性があります。

表 6-2. 予約済み浮動小数点分類ヘルパー関数

シグネチャ	説明
int32 _ _**C28x_isfinite**(float64 x);	x が表現可能な値である場合に限り真となります
int32 _ _**C28x_isfinitef**(float32 x);	x が表現可能な値である場合に限り真となります
int32 _ _**C28x_isinf**(float64 x);	x が「無限大」を表す場合に限り真となります
int32 _ _**C28x_isinff**(float32 x);	x が「無限大」を表す場合に限り真となります
int32 _ _**C28x_isnan**(float64 x);	x が「非数」を表す場合に限り真となります
int32 _ _**C28x_isnanf**(float32 x);	x が「非数」を表す場合に限り真となります
int32 _ _**C28x_isnormal**(float64 x);	x が非正規化されていない場合に限り真となります
int32 _ _**C28x_isnormalf**(float32 x);	x が非正規化されていない場合に限り真となります
int32 _ _**C28x_fpclassify**(float64 x);	浮動小数点値を分類します
int32 _ _**C28x_fpclassifyf**(float32 x);	浮動小数点値を分類します

Copyright © 2026 Texas Instruments Incorporated

Chapter 7

標準 C ライブラリ API

次のセクションでは、C 標準ヘッダ ファイルに適用されるすべての規約について説明します。これらの事項は、ANSI C 規格では規定されていないものの、ツールチェーンが C28x ABI をサポートするために遵守する必要があるすべての要件を扱っています。

7.1 標準 **C** ライブラリについて....................45
7.2 予約済みシンボル....................45
7.3 <assert.h> の実装....................45
7.4 <complex.h> の実装....................45
7.5 <ctype.h> の実装....................46
7.6 <errno.h> の実装....................46
7.7 <float.h> の実装....................46
7.8 <inttypes.h> の実装....................46
7.9 <iso646.h> の実装....................46
7.10 <limits.h> の実装....................47
7.11 <locale.h> の実装....................47
7.12 <math.h> の実装....................47
7.13 <setjmp.h> の実装....................48
7.14 <signal.h> の実装....................48
7.15 <stdarg.h> の実装....................48
7.16 <stdbool.h> の実装....................48
7.17 <stddef.h> の実装....................48
7.18 <stdint.h> の実装....................49
7.19 <stdio.h> の実装....................49
7.20 <stdlib.h> の実装....................49
7.21 <string.h> の実装....................50
7.22 <tgmath.h> の実装....................50
7.23 <time.h> の実装....................50
7.24 <wchar.h> の実装....................50
7.25 <wctype.h> の実装....................50

Copyright © 2026 Texas Instruments Incorporated

7.1 標準 C ライブラリについて

ツール チェーンには通常、サポートする言語の標準ライブラリ (C、C99、C++ など) が含まれています。これらのライブラリには、コンパイル時コンポーネント (ヘッダー ファイル) とランタイム コンポーネント (変数と関数) があります。このセクションでは、ヘッダー ファイルとライブラリの互換性について説明します。

この ABI に準拠した実装は C 標準に準拠し、他の実装で生成されたものと互換性のあるオブジェクト ファイルを生成する必要があります。

コンパイル時に、コンパイラとライブラリ ヘッダー ファイルは同じ実装からのものである必要があります。リンク時には、リンカとライブラリは同一の実装に由来している必要があり、その実装はコンパイラの実装とは異なる場合があります。C28x EABI では、ある実装のヘッダー ファイルを使用してコンパイルされたモジュールが、別の実装のライブラリと互換性がなければなりません。これを「ヘッダー ファイルの互換性」と呼びます。この要件により、ライブラリ ヘッダー ファイルには C 標準で指定されている制限を超える追加の制限が課されます。

C28x は ARM EABI に基づいて設計されています。EABI 向けに標準 C ライブラリを実装する方法に関する背景情報や解説については、ARM Infocenter の Web サイト に掲載されているドキュメント『ARM アーキテクチャ向け C ライブラリ ABI』を参照してください。ARM に適用される詳細が、必ずしも C28x に適用されるとは限りません。このドキュメントの「セクションごとの C ライブラリ セクション」の章を参照してください。

7.2 予約済みシンボル

ABI で説明されているように、いくつかのシンボルは RTS ライブラリで使用するために予約されています。これには以下が含まれます。

- _ftable
- _ctypes_

さらに、セクション 11.4.4 に示すシンボル、またはセクション 11.1 に示す接頭辞のあるシンボルも予約されています。

7.3 <assert.h> の実装

ライブラリは、assert をマクロとして実装する必要があります。その式の引数が false である場合、最終的にヘルパー関数を呼び出して失敗メッセージを出力する必要があります。ヘルパー関数が実際に何かを出力するかどうかは実装定義です。C 標準で規定されているように、このヘルパー関数は abort を呼び出して終了する必要があります。セクション 6.2 を参照してください。

7.4 <complex.h> の実装

C99 標準は、複素数を、対応する実数型の 2 要素からなる配列を含む構造として表現することを要求しています。要素 0 が実数成分であり、要素 1 が虚数成分です。たとえば、_Complex double は次のとおりです。

```
    { double _val[2]; } /* where 0=real 1=imag */
```

TI の C28x ツールセットは C99 複素数をサポートし、このヘッダー ファイルを提供します。

Copyright © 2026 Texas Instruments Incorporated

7.5 <ctype.h> の実装

ctypes.h の関数はロケールに依存しているため、インライン化されない場合があります。これらの関数には、次のものが含まれます。

- isalnum
- isalpha
- isblank (C99 関数。TI ツールセットではまだ提供されていません)
- iscntrl
- isdigit
- isgraph
- islower
- isprint
- ispunct
- isspace
- isupper
- isxdigit
- isascii (廃止された関数。標準の C99 関数ではありません)
- toupper (現在 TI コンパイラによりインライン化されますが、変更される可能性があります)
- tolower (現在 TI コンパイラによりインライン化されますが、変更される可能性があります)
- toascii (廃止された関数。標準の C99 関数ではありません)

7.6 <errno.h> の実装

以下は、errno とともに使用されるために定義されている定数の一部です。詳細なリストについては、errno.h ファイルを参照してください。

```
#define EDOM 0x21
#define ERANGE 0x22
#define EILSEQ 0x58
#define ENOENT 0x2
#define EFPOS 0x98
```

7.7 <float.h> の実装

このファイルのマクロは標準的な方法で定義されています。浮動小数点は IEEE-32、精度浮動小数点および拡張精度浮動小数点は IEEE-64 です。

7.8 <inttypes.h> の実装

このファイル内のマクロ、関数、typedef は、アーキテクチャの整数型に従って自然な方法で定義されています。セクション 2.1 を参照してください。

7.9 <iso646.h> の実装

このファイル内のマクロは C 標準によって完全に規定されており、自然な方法で定義されています。

Copyright © 2026 Texas Instruments Incorporated

7.10 <limits.h> の実装

MB_LEN_MAX を除き、このファイル内のマクロは、アーキテクチャの整数型に従って自然な方法で定義されています。セクション 2.1 を参照してください。

MB_LEN_MAX は次のように定義されます。

```
#define MB_LEN_MAX 1
```

7.11 <locale.h> の実装

TI のツールセットでは「C」ロケールのみが提供されます。LC_* マクロは次のように定義されます。

```
#define LC_ALL        0
#define LC_COLLATE    1
#define LC_CTYPE      2
#define LC_MONETARY   3
#define LC_NUMERIC    4
#define LC_TIME       5
```

lconv 構造のフィールドの順序は、次のとおりです:

(これらは C89 のフィールドです。C99 で追加されたフィールドは含まれていません)

```
char *decimal_point;
char *grouping;
char *thousands_sep;
char *mon_decimal_point;
char *mon_grouping;
char *mon_thousands_sep;
char *negative_sign;
char *positive_sign;
char *currency_symbol;
char  frac_digits;
char  n_cs_precedes;
char  n_sep_by_space;
char  n_sign_posn;
char  p_cs_precedes;
char  p_sep_by_space;
char  p_sign_posn;
char *int_curr_symbol;
char  int_frac_digits;
```

7.12 <math.h> の実装

このライブラリで定義されるマクロは、(ライブラリ変数ではなく) 浮動小数点定数である必要があります。

- HUGE_VALF は浮動小数点の無限大である必要があります。
- HUGE_VAL は倍精度浮動小数点の無限大である必要があります。
- HUGE_VALL は拡張精度浮動小数点の無限大である必要があります。
- INFINITY は浮動小数点の無限大である必要があります。
- NAN は quiet NaN である必要があります。
- MATH_ERRNO は現在規定されていません。
- MATH_ERREXCEPT は現在規定されていません。

次の FP_* マクロが定義されています。

```
#define FP_INFINITE  1
#define FP_NAN       2
#define FP_NORMAL    (-1)
#define FP_SUBNORMAL (-2)
#define FP_ZERO      0
```

その他の FP_* マクロは現在規定されていません。

Copyright © 2026 Texas Instruments Incorporated

7.13 <setjmp.h> の実装

jmp_buf の型とサイズは setjmp.h で定義されています

FPU 以外のターゲットでは、jmp_buf は通常 5 ワードの長さであり、長整数に整列されます。FPU ターゲットでは、jmp_buf は 9 ワードの長さであり、長整数に整列されます。

setjmp 関数と longjmp 関数は jmp_buf が不透明であるため、インライン化してはなりません。つまり、構造のフィールドは標準によって定義されていないため、その構造の内部には setjmp() および longjmp() を除いてアクセスできず、これらは同じライブラリからのアウトオブライン呼び出しである必要があります。これらの関数はマクロとして実装することはできません。

7.14 <signal.h> の実装

TI のツールセットは、シグナル ライブラリ関数を実装していません。

TI のツールセットは「整数」に対して次の typedef を作成します。

```
typedef int sig_atomic_t;
```

TI のツールセットは、次の定数を定義します。

```
#define SIG_DFL ((void (*)(int)) 0)
#define SIG_ERR ((void (*)(int)) -1)
#define SIG_IGN ((void (*)(int)) 1)
#define SIGABRT 6
#define SIGFPE  8
#define SIGILL  4
#define SIGINT  2
#define SIGSEGV 11
#define SIGTERM 15
```

7.15 <stdarg.h> の実装

インターフェイスに公開されるのは va_list 型のみです。マクロは va_start、va_arg、va_end を実装するために使用されます。va_list 内の引数の形式については、Chapter 3 を参照してください。

省略記号 (...) で宣言された可変長 C 関数の呼び出し時には、最後に宣言された引数と追加の引数は、セクション 3.3 に記載されているとおりにスタックに渡され、<stdarg.h> 内のマクロを使用してアクセスされます。これらのマクロは、va_start の呼び出しによって初期化され、va_arg の呼び出しによって進められる永続的な引数ポインタを使用します。これらのマクロの実装には、以下の規則が適用されます。

- va_list の型は文字列 * です。
- マクロ va_start(ap, parm) を呼び出すと、ap は parm に割り当てられた最後のアドレスの直後 (1 バイト後) を指すように設定されます。
- va_arg(ap, type) を続けて呼び出すと、ap は指定された type の引数に割り当てられた領域の最後のアドレスの 1 バイト後を指す設定のままとなります。

7.16 <stdbool.h> の実装

C++ では、型「ブール」は組み込み型です。

C99 では、型「_Bool」は組み込み型です。C99 では、ヘッダー ファイル stdbool.h においてマクロ「bool」が定義され、これは _Bool に展開されます。

これらの型はいずれも 8 ビットの符号なし型として表現されます。

7.17 <stddef.h> の実装

stddef.h に定義されている各型のサイズと整列は、セクション 2.4 に規定されています。

Copyright © 2026 Texas Instruments Incorporated

7.18 <stdint.h> の実装

このヘッダ ファイル内のマクロと typedefs は、そのアーキテクチャの整数型に応じて自然な方法で定義されます。セクション 2.1 を参照してください。

7.19 <stdio.h> の実装

TI ツールセットは、stdio.h ライブラリで使用するために以下の定数を定義します。

```
#define _IOFBF 1
#define _IOLBF 2
#define _IONBF 4
#define BUFSIZ 256
#define EOF (-1)
#define FOPEN_MAX
#define FILENAME_MAX
#define TMP_MAX
#define L_tmpnam
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
#define stdin &_ftable[0]
#define stdout &_ftable[1]
#define stderr &_ftable[2]
```

FOPEN_MAX、FILENAME_MAX、TMP_MAX、および L_tmpnam の値は、実際には最大値の下限です。ライブラリは、より大きな値をサポートすることはできますが、少なくとも指定された値を提供する必要があります。

TI ツールセットでは stdout と stderr をそれぞれ &_ftable[1] および &_ftable[2] として定義しているため、FILE のサイズは実装で認識されている必要があります。

TI ヘッダ ファイルでは、stdin、stdout、stderr は配列 _ftable への参照を展開します。このようなファイルと正しく相互リンクするには、他の実装でも FILE 配列をその名前で正確に実装する必要があります。C28x EABI には (ARM EABI のモードのような)「互換モード」がありません。このモードでは、stdin、stdout、stderr がマクロではなくリンク時シンボルです。互換モードが存在しないため、stdin を直接参照するモジュールと相互リンクする必要があるリンカは、support_ftable サポートする必要があります。

プログラムが stdin、stdout、または stderr マクロ (またはこれらのマクロのいずれかを参照するマクロとして実装された関数) を使用しない場合、FILE 配列を使用しても問題はありません。

C の I/O 関数で一般的にマクロとして実装される getc、putc、getchar、putchar はインライン化してはいけません。

fpos_t 型は long として定義されます。

7.20 <stdlib.h> の実装

TI ツールセットでは、stdlib.h の構造を次のように定義しています。

```
typedef struct { int quot; int rem; } div_t;
typedef struct { long int quot; long int rem; } ldiv_t;
typedef struct { long long int quot; long long int rem; } lldiv_t;
```

stdlib.h ライブラリで利用される定数についても、TI ツールセットでは次のように定義されています。

```
#define EXIT_SUCCESS  0
#define EXIT_FAILURE  1
#define MB_CUR_MAX 1
```

rand 関数の返す結果は ABI 仕様では定義されていません。

この ABI 仕様では、ライブラリに getenv 関数や system 関数を実装する必要はありません。TI ツールセットには getenv 関数が用意されていますが、デバッガのサポートが必要です。TI ツールセットには system 関数は用意されていません。

Copyright © 2026 Texas Instruments Incorporated

7.21 <string.h> の実装

strtok 関数は静的な状態を持つため、インライン化してはなりません。strcoll 関数と strxfrm 関数も実行環境のロケールに依存するため、インライン化してはなりません。

7.22 <tgmath.h> の実装

C99 規格では、このヘッダ ファイルが完全に規定されています。TI ツールセットは、このヘッダ ファイルを提供していません。

7.23 <time.h> の実装

このライブラリに定義されている一部の typedefs や定数は、実行環境に依存しています。コードを移植可能にするため、コードで time_t や clock_t の型や範囲について仮定してはなりません。

CLOCKS_PER_SEC の型は clock_t です。

7.24 <wchar.h> の実装

TI ツールセットは、このライブラリで使用するために次の型と定数を定義しています。

```
typedef int wint_t;
#define WEOF ((wint_t)-1)
```

型 mbstate_t は

7.25 <wctype.h> の実装

TI ツールセットは、このライブラリで使用するために次の型を定義しています。

```
typedef void * wctype_t;
typedef void * wctrans_t;
```


Copyright © 2026 Texas Instruments Incorporated

Chapter 8
C++ ABI

C++ ABI は、異なるツールチェーンで生成されたコード同士が相互運用できるように、標準化が必要となる C++ 言語の実装上の要素を定めています。この C28x C++ ABI は、当初 IA-64 用に開発され、現在では GCC を含む多くの C++ ツールチェーンに広く採用された Generic C++ ABI を基盤としています。この基本標準は「GC++ABI」と呼ばれ、http://refspecs.linux-foundation.org/cxxabi-1.83.html で公開されています。

このセクションでは、その基本仕様に対する追加点や相違点について説明します。

8.1 制限 **(GC++ABI 1.2)**........ 52
8.2 エクスポート テンプレート **(GC++ABI 1.4.2)**........ 52
8.3 データ レイアウト **(GC++ABI** 第 **2** 章**)**........ 52
8.4 初期化ガード変数 **(GC++ABI 2.8)**........ 52
8.5 コンストラクタ戻り値 **(GC++ABI 3.1.5)**........ 52
8.6 ワンタイム構築 **API (GC++ABI 3.3.2)**........ 52
8.7 オブジェクト構成順序を制御する **(GC++ ABI 3.3.4)**........ 52
8.8 デマングラ **API (GC++ABI 3.4)**........ 53
8.9 静的データ **(GC++ ABI 5.2.2)**........ 53
8.10 仮想テーブルと **key function (GC++ABI 5.2.3)**........ 53
8.11 アンワインド テーブルの位置 **(GC++ABI 5.3)**........ 53

Copyright © 2026 Texas Instruments Incorporated

8.1 制限 (GC++ABI 1.2)

GC++ABI は、RTTI 実装のため、完全なオブジェクト内に含まれる非仮想基本サブオブジェクトのオフセットを、56 ビット符号付き整数で表現できるように制限します。C28x ファミリの場合、この制限は 24 ビットに縮小されます。これは、ベースクラスのサイズに対して 2^{23} -1 (すなわち 0x7fffff) バイトの実用上の制限を意味します。

8.2 エクスポート テンプレート (GC++ABI 1.4.2)

エクスポート テンプレートは、現時点では ABI によって規定されていません。

8.3 データ レイアウト (GC++ABI 第 2 章)

POD (Plain Old Data) のレイアウトは、本書の Chapter 2 において規定されています。POD 以外のデータのレイアウトは、基本文書で規定されています。ビット フィールドについては、小さな例外があり、これはセクション 2.8 で取り上げています。

8.4 初期化ガード変数 (GC++ABI 2.8)

ガード変数は、16 ビット コンテナの最初のバイトに格納される 1 バイトのフィールドです。ガード変数が 0 以外の値であれば、初期化が完了していることを示します。これは IA-64 の方式に従いますが、コンテナは 64 ビットではなく 16 ビットです。

以下は、ヘルパー関数 __cxa_guard_acquire のリファレンス実装であり、ガード変数を読み取り、初期化がまだ完了していなければ 1 を返し、完了している場合は 0 を返します:

```
int __cxa_guard_acquire(unsigned int *guard)
{
   char *first_byte = (char *)guard;
   return (*first_byte == 0) ? 1 : 0;
}
```

以下は、ヘルパー関数 __cxa_guard_release のリファレンス実装であり、ガード オブジェクトを変更して初期化が完了したことを示します:

```
void __cxa_guard_release(unsigned int *guard)
{
   char *first_byte = (char *)guard;
   *first_byte = 1;
}
```

8.5 コンストラクタ戻り値 (GC++ABI 3.1.5)

C28x は ARM EABI に従っており、その下では C1 および C2 コンストラクタは *this* ポインタを返します。このようにすることで、これらの関数に対する呼び出しにおいて末尾呼び出し最適化を可能にします。

同様に、非仮想呼び出しにおける D1 デストラクタおよび D2 デストラクタも this を返します。仮想デストラクタへの呼び出しは thunk 関数を使用し、これらのデスクトラクタは this を返しません。

GC++ABI のセクション 3.3 では、配列 new および delete 用にいくつかのライブラリ ヘルパー関数が規定されており、これらはコンストラクタまたはデストラクタへのポインタをパラメータとして受け取ります。GC++ABI では、これらのパラメータは戻り値が void の関数へのポインタとして宣言されていますが、C28x ABI では、「this」に対応する void * を返す関数へのポインタとして宣言されています。

8.6 ワンタイム構築 API (GC++ABI 3.3.2)

ガード変数は、16 ビット コンテナの最初のバイトに格納される 8 ビットのフィールドです。セクション 8.4 を参照してください。

8.7 オブジェクト構成順序を制御する (GC++ ABI 3.3.4)

C28x ABI は、オブジェクト構築を制御するメカニズムを規定していません。

Copyright © 2026 Texas Instruments Incorporated

8.8 デマングラ API (GC++ABI 3.4)

C28x ABI では、デマングラにランタイム インターフェイスを提供する関数 _cxa_demangle を提供する、実装に関する要件が一時停止されています。

8.9 静的データ (GC++ ABI 5.2.2)

GC++ ABI では、インライン関数から参照される静的オブジェクトは COMDAT グループ内に定義される必要があります。そのようなオブジェクトに関連するガード変数がある場合、そのガード変数も COMDAT グループに定義する必要があります。GC++ ABI では、静的変数とガード変数を別々のグループに置くこと自体は許されていますが、その方法は推奨されていません。C28x ABI はこれを完全に禁止しており、静的変数とそのガード変数は 1 つの COMDAT グループに定義され、そのシグネチャは静的変数の名前でなければなりません。

8.10 仮想テーブルと key function (GC++ABI 5.2.3)

GC++ABI では、クラスの key function が定義されています。この関数の定義は、そのクラスに対する仮想テーブルの生成をトリガします。key function とは、*クラス定義の時点*でインラインでない最初の非純粋仮想関数です。C28x ABI ではこれを修正し、*遷移ユニットの末尾*でインラインでない最初の非純粋仮想関数としています。言い換えると、クラス定義の後に初めてインラインとして宣言されたメンバーは key function ではありません。

8.11 アンワインド テーブルの位置 (GC++ABI 5.3)

例外処理については、このドキュメントの Chapter 9 に記載されています。

Copyright © 2026 Texas Instruments Incorporated

Chapter 9
例外処理

C28x EABI は、テーブルに基づく例外処理 (TDEH) を採用しています。TDEH は、C++ のような例外をサポートする言語に対して、例外処理を実装します。

TDEH は例外処理に必要な情報をエンコードするためのテーブルを使用します。これらのテーブルは、プログラムの読み取り専用データの一部です。例外がスローされると、ランタイム サポート ライブラリ内の例外処理コードが、スタックを *アンワインド*して、例外をキャッチするキャッチ句を持つ関数のスタック フレームまで例外を伝播させます。スタックをアンワインドする過程では、ローカルに定義されたオブジェクトを順に (デストラクタ呼び出しによって) 破棄しなければなりません。テーブルには、スタックのアンワインド方法、どのオブジェクトをいつ破棄するか、例外が最終的にキャッチされたときにどこに制御を移すか、といった情報がエンコードされています。

TDEH テーブルはリンカによって実行可能ファイルに生成されます。その際には、コンパイラが再配置可能ファイルに出力した情報が使用されます。このセクションでは、これらのテーブルのフォーマットとエンコード方式、およびその情報が例外伝播にどのように使用されるかを定義します。ABI に準拠したツールチェーンは、ここで規定された形式でテーブルを生成する必要があります。

9.1 概要....55
9.2 PREL31 エンコーディング....55
9.3 例外インデックス テーブル **(EXIDX)**....56
9.4 例外処理命令テーブル **(EXTAB)**....57
9.5 アンワインド命令....58
9.6 ディスクリプタ....61
9.7 特別なセクション....63
9.8 C++ 以外のコードとの相互作用....63
9.9 システム機能との相互作用....64
9.10 TI ツールチェーンにおけるアセンブリ言語オペレータ....64

Copyright © 2026 Texas Instruments Incorporated

9.1 概要

C28x の例外処理テーブルのフォーマットとメカニズムは、ARM プロセッサ ファミリのものに基づいており、それ自体は IA-64 例外処理 ABI (http://www.codesourcery.com/public/cxx-abi/abi-eh.html) に基づいています。このセクションでは、C28x 固有の部分に焦点を当てます。

TDEH データは、EXIDX、EXTAB、キャッチ / クリーンアップ ブロックの 3 つの主要コンポーネントで構成されます。

例外インデックス テーブル (EXIDX) は、プログラム アドレスを例外アクション テーブル (EXTAB) のエントリにマッピングします。プログラム内のすべてのアドレスは EXIDX によってカバーされます。

EXTAB は、スタック フレームをアンワインドする方法 (レジスタの復元とスタック ポインタの調整による) や、例外が伝播したときに呼び出すべきキャッチ ブロックとクリーンアップ ブロックを記述する命令をエンコードします。

キャッチ ブロックとクリーンアップ ブロック (総称して *ランディング パッド*) は、例外処理を行うコード片です。クリーンアップ ブロックにはデストラクタ関数の呼び出しが含まれます。キャッチ ブロックはユーザー コード内にキャッチ句を実装します。これらのブロックは、例外が実際にスローされた場合にのみ実行されます。これらのブロックは、関数の残りの部分が生成される際に同時に生成され、関数と同じスタック フレーム内で実行されますが、異なるセクションに配置することもできます。

9.2 PREL31 エンコーディング

EXIDX テーブルおよび EXTAB テーブルの一部のフィールドは、プログラム メモリのアドレスや、テーブル内の他の位置へのポインタを記録する必要があります。これらはいずれも通常、コード セグメントまたは読み取り専用セグメントに存在します。位置の独立性を実現するために、これらは R_C28x_PREL31 (ここでは略して PREL31) と呼ばれる、特殊な用途向けの PC 相対再配置を用いて処理されます。PREL31 フィールドは、32 ビット ワードの最下位 31 ビットに格納される、スケーリングされた符号付き 31 ビットオフセットとしてエンコードされます。残りの (最上位の) 1 ビットは、コンテキストによって異なる目的に使用されます。当該フィールドが参照する再配置されたアドレスは、エンコードされたオフセットを 1 ビット左シフトし、それをフィールドのアドレスに加えることで求められます。

Copyright © 2026 Texas Instruments Incorporated

9.3 例外インデックス テーブル (EXIDX)

ソース コード内に throw 文が現れると、コンパイラは _cxa_throw という名前のランタイム サポート ライブラリ関数への呼び出しを生成します。throw が実行されると、_ _cxa_throw 呼び出しサイトの復帰アドレスが、どの関数が例外を投げているかを識別するために使用されます。ライブラリは、その復帰アドレスを EXIDX テーブル内で検索します。

テーブル内の各エントリは、プログラム アドレス範囲の例外処理動作を表しており、それは 1 つ以上の関数で、まったく同じ例外処理動作を共有している可能性があります。各エントリは、プログラム アドレス範囲の開始をエンコードし、次のエントリにエンコードされたアドレスまでのすべてのプログラム アドレスをカバーするとみなされます。リンカは同一の動作を持つ隣接する関数を 1 つのエントリにまとめる場合があります。

各エントリは 2 つの 32 ビット ワードで構成されます。各エントリの最初のワードは PREL31 フィールドであり、関数または関数群の開始プログラム アドレスを表します。最初のワードのビット 31 は 0 である必要があります。2 番目のワードは、2 番目のワードのビット 31 に応じて 3 つの形式のいずれかになります。ビット 31 が 0 の場合、2 番目のワードは、メモリの別の場所にある EXTAB エントリへの PREL31 ポインタ、または特別な値 EXIDX_CANTUNWIND のいずれかです。ビット 31 が 1 の場合、2 番目のワードはインライン化された EXTAB エントリです。これら 3 つの形式については、以降のサブセクションで詳しく説明します。

9.3.1 アウトオブライン EXTAB エントリへのポインタ

この形式では、EXIDX テーブル エントリの 2 番目のワードには、最上位ビットに 0 が格納され、それ以外のビットには、このアドレス範囲に対応する EXTAB エントリの PREL-31 エンコードされたアドレスが格納されます。

31	30-0
0	PREL31 Representation of function address
0	PREL31 Representation of EXTAB entry

9.3.2 EXIDX_CANTUNWIND

特殊なケースとして、EXIDX の 2 番目のワードの値が 0x1 の場合、その EXIDX は EXIDX_CANTUNWIND を表し、関数をまったくアンワインドできないことを示します。例外がこのような関数を通過しようとすると、アンワインダは、言語に応じて abort または std::terminate を呼び出します。

31	30-0
0	PREL31 Representation of function address
0x00000001 (EXIDX_CANTUNWIND)	

9.3.3 インライン化された EXTAB エントリ

この関数用の EXTAB エントリ全体が十分小さい場合、それは 2 番目の EXIDX ワードに格納され、上位ビットが 1 に設定されます。2 番目の ワードは、セクション 9.4 に記載されている EXTAB コンパクト モデルと同じエンコーディングを使用しますが、ディスクリプタも終端 NULL もありません。これにより、アウトオブラインの EXTAB エントリへのポインタに必要であった 4 バイトと、終端 NULL に必要であった 4 バイトを節約できます。

31	30-28	27-24	23-0
0	PREL31 Representation of function address		
1	**000**	PR Index	Data for personality routine specified by 'index'

English Document: SPRAC71
Copyright © 2026 Texas Instruments Incorporated

9.4 例外処理命令テーブル (EXTAB)

各 EXTAB エントリは、キャッチおよびクリーンアップ処理を行うためのフレーム巻き戻し命令および記述子をエンコードした、1 つ以上の 32 ビットワードで構成されます。最初のワードはそのエントリの*パーソナリティ*を表します。これは、そのエントリの形式と解釈です。

例外がスローされると、EXTAB エントリはランタイム サポート ライブラリで提供される「パーソナリティ ルーチン」によってデコードされます。ABI で指定されているパーソナリティ ルーチンを表 9-1 に示します。

9.4.1 EXTAB 一般モデル

一般的な EXTAB エントリは、最初のワードのビット 31 を 0 に設定することで示されます。最初のワードには、パーソナリティ ルーチンのアドレスを表す PREL31 エントリが含まれます。EXTAB エントリの残りのワードは、パーソナリティ ルーチンに渡されるデータです。

31	30-0
0	PREL31 Representation of personality routine address
Optional data for the personality routine	

オプション データの形式はパーソナリティ ルーチンの裁量に任されますが、長さは 32 ビット ワード単位の整数倍でなければなりません。アンワインダは、オプション データの先頭ワードへのポインタを渡してパーソナリティ ルーチンを呼び出します。

9.4.2 EXTAB コンパクト モデル

コンパクトな EXTAB エントリは、最初のワードの 31 ビット目に 1 が設定されていることで示されます。(EXTAB エントリが EXIDX エントリの 2 番目のワードにエンコードされる場合は、常にこのコンパクト フォームが使用されます)。コンパクト フォームでは、パーソナリティ ルーチンは、エントリの最初のバイトにある 4 ビットの PR インデックスによってエンコードされます。残りの 3 バイトには、パーソナリティ ルーチンによって定義されたアンワインディング命令が含まれます。非インラインの EXTAB エントリでは、追加のデータが、連続する 32 ビット ワードとして提供されます。ここでは、追加のアンワインディング命令の後に、および必要に応じてアクション ディスクリプタが続き、最後に NULL ワードで終了します。

31	30-28	27-24	23-0
1	**000**	PR Index	Encoded unwinding instructions
Zero or more additional 32-bit words of unwinding instructions (out-of-line EXTAB only)			
Zero or more catch, cleanup, or FESPEC descriptors (out-of-line EXTAB only)			
32-bit NULL terminator (out-of-line EXTAB only)			

Copyright © 2026 Texas Instruments Incorporated

9.4.3 パーソナリティ ルーチン

C28x には、ABI により定義された以下のパーソナリティ ルーチンが含まれています。これらは ARM EABI と同じフォーマットです。以下の表は、パーソナリティ ルーチンとその PR インデックスを示しています。

表 9-1. C28x TDEH パーソナリティ ルーチン

PR インデックス (ビット 27～24)	パーソナリティ	ルーチン名	アンワインド命令	範囲フィールドの幅	注
0000	PR0 (Su16)	_ _C28x_unwind_cpp_pr0	最大 3 つの 1 バイト命令	16	
0001	PR1 (Lu16)	_ _C28x_unwind_cpp_pr1	無制限の 1 バイト命令	16	
0010	PR2 (Lu32)	_ _C28x_unwind_cpp_pr2	無制限の 1 バイト命令	32	16 ビットの範囲フィールドが到達しない場合は、使用する必要がある

コンパクト モデルの EXTAB エントリを使用する場合、再配置可能ファイルは、R_C28x_NONE 再配置の形で、EXTAB セクションから対応するパーソナリティ ルーチン シンボルへの参照を含めることによって、利用するルーチンを明示的に示す必要があります。

9.5 アンワインド命令

フレームのアンワインドは、関数のエピローグをシミュレーションすることによって実行されます。スタック アンワインダがその情報をデコードしてエピローグをシミュレーションできるようにするため、関数のエピローグで実行される可能性のあるあらゆる操作は、EXTAB エントリにエンコードされる必要があります。

アンワインド命令は スタック レイアウトを前提に設計されています。特に、*呼び出し先保存レジスタの安全なデバッグの順序* が常に仮定されています。

9.5.1 共通シーケンス

抽象的に見ると、すべてのアンワインドのシーケンスは次のような手順を踏みます。

1. SP を復元する (SP += 定数)
2. (任意) 呼び出し先保存レジスタを復元する (reg1 := SP[0]; reg2 := SP[-1]、など)
3. リターンする

ステップ **1:SP** を復元する

実際のエピローグでは、呼び出し先保存レジスタを復元されるまでは SP は復元されませんが、スタック アンワインドは仮想的な処理であるため、TDEH によるシミュレーション アンワインドでは先に SP の復元を行うことがあります。この仕組みにより、他の呼び出し先保存レジスタの復元が容易になります。

SP は、定数でインクリメントすることで復元されます。明示的なインクリメントに加えて、呼び出し先保存領域のサイズ分だけ SP が暗黙的にインクリメントされます。

ステップ **2:**レジスタを復元する

抽象的に見ると、呼び出し先保存レジスタは *安全なレジスタ デバッグ* 順 (セクション 4.6.2) に従って復元されます。復元は (古い) SP が指す位置から始まり、アドレスを下げながら進みます。

ステップ **3:**戻り値

すべてのアンワインド シーケンスは、暗黙的または明示的な「RET 」で終了し、これにより現在のフレームに対するアンワインドが完了したことが示されます。

English Document: SPRAC71
Copyright © 2026 Texas Instruments Incorporated

9.5.2 バイト エンコードされたアンワインド命令

パーソナリティ ルーチン PR0、PR1、PR2 では、フレームをアンワインドする方法を記述するために、バイト エンコードされた一連の命令が使用されます。最初のいくつかの命令は、EXTAB の最初のワードの残り 3 バイトに格納され、追加の命令は後続のワードにパックされます。最後のワードの未使用バイトは、「RET 」命令で埋められます。

命令はバイト エンコードされますが、それらは常に MSB から始まる 32 ビットワードにパックされます。その結果、最初のアンワインド命令は、リトル エンディアン モードでは最下位アドレスのバイトには存在しません。

パーソナリティ ルーチン PR0 では、最大 3 つのアンワインド命令が可能であり、これらはすべて最初の EXTAB ワードに格納されます。アンワインド命令が 4 つ以上ある場合は、他のいずれかのパーソナリティ ルーチンを使用する必要があります。

31	30-28	27-24	23-16	15-8	7-0
1	**000**	0000 (PR0)	First unwind instruction	Second unwind instruction	Third unwind instruction
Optional descriptors					
NULL					

PR1 と PR2 においては、ビット 23～16 が追加の 32 ビット ワードのアンワインド命令の数をエンコードしており、これは 0 である場合もあります。

31	30-28	27-24	23-16	15-8	7-0
1	**000**	PR Index	Number of additional unwinding words	First unwind instruction	Second unwind instruction
Third unwind instruction			Fourth unwind instruction	...	...
Optional descriptors					
NULL					

表 9-2 は、一連のアンワインド命令の概要です。各命令の詳細は、表の後に記載されています。

表 9-2. スタック アンワインド命令

エンコード	手順	説明
0000 0xxx	POP (XAR1-XAR3) + RET	(x=0 の場合は RET のみ)
0000 1000 0xxx xxxx	POP (XAR1-XAR3、R4-R7) + RET	(x=0 の場合は RET のみ)
0000 1yyy yxxx xxxx		予約済み (yyyy は 0 ではない)
0001 0000		アンワインドできません (関数は例外を捕捉するかもしれませんが、伝播はしません)
0001 0001 xxxx xxxx	SP -= (ULEB128 << 1) + 512;	範囲は [514, max]
0001 001x		予約済み
0001 01xx		予約済み
0001 1xxx		予約済み
001x xxxx		予約済み
01xx xxxx		予約済み
1xxx xxxx	SP -= (xxxxxxx << 1) + 2;	範囲は [2, 256]

次の制限事項が適用されます。

- C++ 例外を伝播する可能性のある C/C++ 呼び出し可能関数においては、スタックは常に 32 ビットにアラインされていなければなりません。

Copyright © 2026 Texas Instruments Incorporated

- プログラムでは 0x80000000 を超える領域にコードやデータを配置することはできません。配置すると、再配置エラーが発生します。

その他のすべてのビット パターンは予約済みです。

以降の段落では、アンワインド命令の解釈について詳しく説明します。

POP + RET

POP+RET 命令は、この関数のプロローグによって保存されたレジスタを表すビットマスクを指定します。これらのレジスタは、XAR1 から XAR3 まで順に POP される必要があります。FPU が有効化されている場合、R4 から R7 が順番に POP されます。完了すると、アンワインド命令はなくなります。ビットマスクのどのビットもセットされていない場合、単なる RET 命令となります。

小さなインクリメント

7	6	5	4	3	2	1	0
1	1	k	k	k	k	k	k

k の値はエンコーディングの下位 6 ビットから抽出されます。この命令では、SP を 0x8～0x200 の範囲の値だけインクリメントできます。0x208～0x400 の範囲のインクリメントは、これらの命令のうち 2 つを使用して行う必要があります。

大きなインクリメント

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1
k	k	k	k	k	k	k	k
...							

値 ULEB128 は、8 ビットのオペコードに続くバイト列に ULEB128 エンコードされます。この命令では、SP を 0x408 以上の値だけインクリメントできます。0x408 未満のインクリメントは、1 つまたは 2 つの小さなインクリメント命令で行うべきです。

CANTUNWIND

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

この命令は、関数がアンワインドできないことを示します。通常、それは割り込み関数であるためです。しかし、割り込み関数は依然として try/catch コードを持つことができるため、EXIDX_CANTUNWIND は適切ではありません。

Copyright © 2026 Texas Instruments Incorporated

9.6 ディスクリプタ

ローカル オブジェクトの破棄が必要な場合や、この関数で例外が捕捉される場合、EXTAB には処理内容と対象となる例外の種類を記述した *ディスクリプタ* が含まれます。

ディスクリプタがある場合、アンワインド命令の後に配置されます。ディスクリプタの形式は、複数のディスクリプタ エントリが並び、その後に 32 ビットのゼロ (NULL) ワードが続く形式になります。各ディスクリプタは *範囲* から始まり、ディスクリプタの種類と、そのディスクリプタが適用されるプログラム アドレス範囲を示します。範囲の後に追加のディスクリプタ固有のワードが続きます。

ディスクリプタは、すべての該当するディスクリプタを一度に処理できるように、深さ優先順で列挙されなければなりません。

ディスクリプタを伴う EXTAB エントリの一般形式は次のとおりです。

31	30-28	27-24	23-0
1	**000**	PR Index	Unwinding instructions
Zero or more additional 32-bit words of unwinding instructions			
Zero or more catch, cleanup, or FESPEC descriptors			
32-bit NULL terminator			

9.6.1 型識別子のエンコード

キャッチ記述子および FESPEC 記述子 (セクション 9.6.5) は、スローされたオブジェクトの型をキャッチ句および例外仕様と照合するために使用される型識別子をエンコードします。これらのフィールドは、指定された型に対応する type_info オブジェクトを参照するようにエンコードされます。

9.6.2 範囲

範囲はディスクリプタ タイプを識別し、アクションが実行されるプログラム アドレスの範囲を指定します。その範囲は、例外をスローする可能性のある呼び出しサイトに対応します。アンワインダは、呼び出しサイトを含む範囲を持つディスクリプタを探してディスクリプタ リストを調べ、一致するものが見つかるとそのディスクリプタが有効化されます。

範囲は、関数の開始アドレスからのオフセットと長さ (いずれもバイト単位) を指定して、プログラム アドレスの範囲をエンコードします。長さとオフセットがそれぞれ 15 ビットの符号なしフィールドに収まる場合、範囲は短いフォームのエンコーディングを使用し、EXTAB エントリの残りは PR0 または PR1 のいずれかにエンコードできます。長さまたはオフセットのいずれかが 15 ビットを超える場合、範囲は長いフォームのエンコーディングを使用します。その際には PR2 を使用する必要があります。

31-17	16	15-1	0
Length	**X**	Offset	**Y**
Data for descriptor			

図 9-1. 短いフォームの範囲

短いフォームの範囲は PR2 (Lu32) では使用できません。

31-1	0
Length	**X**
Offset	**Y**
Data for descriptor	

図 9-2. 長いフォームの範囲

長さまたはオフセットが長いフォームの範囲を必要とする場合、パーソナリティ ルーチン PR2 (Lu32) を使用する必要があります。

Copyright © 2026 Texas Instruments Incorporated

範囲のエンコーディングにおけるビット X と Y は、スコープの後に続くディスクリプタの種類を示します。

X	Y	Descriptor
0	0	Cleanup descriptor
1	0	Catch descriptor
0	1	Function exception specification (FESPEC) descriptor

9.6.3 クリーンアップ記述子

クリーンアップ記述子は、完全に構築され、範囲を抜ける直前のローカル オブジェクトを確実に破棄するために利用されます。

<table>
<tr><td colspan="2" align="center">31-0</td></tr>
<tr><td colspan="2" align="center">Scope (long or short form)</td></tr>
<tr><td>0</td><td align="center">PREL31 program address of landing pad</td></tr>
</table>

クリーンアップ記述子は、1 つ以上のデストラクタ関数を呼び出すクリーンアップ コード ブロックへのポインタのみを保持します。

9.6.4 キャッチ ディスクリプタ

キャッチ ディスクリプタでは、どの例外がいつ捕捉されるかを制御します。関数にはいくつかの catch 句がある場合があり、それぞれが潜在的にスローされる関数呼び出しの異なるサブセットに適用されます。1 つのコール サイトには、それぞれ異なるタイプの複数のキャッチ ディスクリプタを含めることができます。

キャッチ ディスクリプタの型がスローされた型と一致する場合、コントロールは*ランディング パッド*に転送されます。ランディング パッドはキャッチ ブロックを表すコード フラグメントにすぎません。キャッチ ブロックはユーザのコードに *catch* 句を実装します。これらのブロックは、例外が実際にスローされたときにのみ実行されます。これらのブロックは、関数の残りの部分が生成されたときに関数用に生成され、関数と同じスタック フレーム内で実行されますが、別のセクションに配置することもできます。

<table>
<tr><td colspan="3" align="center">31-0</td></tr>
<tr><td colspan="2" align="center">Scope (long or short form)</td><td>0</td></tr>
<tr><td>R</td><td colspan="2" align="center">PREL31 program address of landing pad</td></tr>
<tr><td colspan="3" align="center">Type</td></tr>
</table>

ビット R が 1 の場合、catch 句の型は TYPE で表される参照型です。ビット R が 0 の場合、型は参照型ではありません。

type フィールドは、type_info オブジェクトへの参照 と次の 2 つの特別な値のいずれか 1 つのどちらかになります。

- 特別な値 0xFFFFFFFF (-1) は*任意の*型 ["catch(...)"] を意味します。
- 特別な値 0xFFFFFFFE (-2) は*任意の*型 ["catch(...)"] を意味し、また、パーソナリティ ルーチンが直ちに _URC_FAILURE を返す必要があることを示します。この場合、ランディング パッドのアドレスは 0 に設定する必要があります。このイディオムは、そのスコープでカバーされているコードから例外が伝播するのを防ぐために使用できます。

Copyright © 2026 Texas Instruments Incorporated

9.6.5 関数例外仕様 (FESPEC) ディスクリプタ

FESPEC ディスクリプタは、ユーザー コード内で throw() 宣言を強制します。throw 宣言が使用される場合、この関数に対して FESPEC ディスクリプタが作成され、リストされた型のみがスローされるようになります。リストされていない型がスローされた場合、アンワインダは通常 std::unexpected を呼び出します (ただし例外があります)。

31-0	
Scope (long or short form)	
D	Number of type info pointers
Reference to type_info object	
Reference to type_info object	
. . .	
0	(if D == 1) PREL31 program address of landing pad

ディスクリプタの最初のワードは 31 ビットの符号なし整数で構成されており、その後に続く type_info フィールドの数を指定します。

ビット D が 1 の場合、type_info リストの後には、スローされた型がリスト内のいずれにも一致しない場合に呼び出されるコード フラグメントの PREL31 プログラム アドレスを含む 32 ビットワードが続きます。このワードのビット 31 は 0 に設定されます。

ビット D が 0 であり、リスト内のどの型もスローされた型と一致しない場合、アンワインディング コードは _cxa_call_unexpected を呼び出す必要があります。この形式に一致するディスクリプタがある場合、EXTAB セクションには _ _cxa_call_unexpected への R_C28x_NONE 再配置が含まれている必要があります。

9.7 特別なセクション

すべての例外処理テーブルは、2 つのセクションに格納されます。EXIDX テーブルは、SHT_C28x_UNWIND タイプの C28x.exidx というセクションに格納されます。リンカは、すべての入力 C28x.exidx セクションを 1 つの連続した C28x.exidx 出力セクションに結合し、それらが参照するコード セクションと同じ相対順序を維持する必要があります。つまり、EXIDX テーブル内のエントリはアドレス順にソートされます。再配置可能ファイル内の各 EXIDX セクションは、この要件を示すために SHF_LINK_ORDER フラグを設定しておく必要があります。

EXTAB は、SHT_PROGBITS タイプの C28x.extab というセクションに格納されます。EXTAB は連続している必要はなく、並び順の要件もありません。

例外テーブルはメモリ内の任意の場所にリンクできます。

9.8 C++ 以外のコードとの相互作用

9.8.1 EXIDX エントリの自動生成

EXIDX エントリを持たない関数には、リンカによって自動的にエントリが作成されます。そのため、例外処理を有効にせずにコンパイルされたライブラリ (C のみのライブラリなど) の関数も、TDEH を使用するアプリケーションで使用できます。自動生成されるエントリは EXIDX_CANTUNWIND となります。したがって、例外処理サポートを有効にせずにコンパイルされた関数が、例外を伝播する関数を呼び出した場合、std::terminate が呼び出され、アプリケーションは停止します。

9.8.2 ハンドコードで記述されたアセンブリ関数

ハンドコードで記述された関数は、例外を処理または伝播できるように組込むことができます。これは、その関数が例外を伝播する可能性のある関数を呼び出し、その例外をアセンブリ関数の外に伝播させる必要がある場合にのみ必要です。ユーザーは、少なくともアンワインド命令を含む適切な EXIDX エントリおよび EXTAB を作成する必要があります。

Copyright © 2026 Texas Instruments Incorporated

9.9 システム機能との相互作用

9.9.1 共有ライブラリ

例外処理テーブルは、実行可能ファイル内で例外を伝播できます。異なるロード モジュール間の呼び出しをまたいで例外を伝播させるには、OS の支援が必要です。

9.9.2 オーバーレイ

例外を伝播する可能性のある C++ 関数は、オーバーレイの一部であってはなりません。EXIDX 参照テーブルはオーバーレイ関数を処理せず、特定の場所において可能な複数の関数を区別できませんでした。

9.9.3 割り込み

割り込み、ハードウェア例外、OS シグナルは、例外によって直接処理できません。

割り込み関数はどこで発生するか分からないため、割り込み関数からの例外伝播をサポートできません。すべての割り込み関数は EXIDX_CANTUNWIND になります。ただし、割り込み関数は自ら例外をスローする可能性のある関数を呼び出すことができます。したがって、割り込み関数は EXIDX テーブルに含まれ、ディスクリプタを持つことはありますが、アンワインディング命令を持つことは決してありません。

割り込みを例外で表現するアプリケーションでは、割り込みが割り込み関数でキャッチされるようにし、そこでグローバルな volatile オブジェクトを設定して割り込みが発生したことを示し、その後割り込み関数が戻った後に、その変数の値を使用して例外をスローする必要があります。

OS がシグナルを提供する場合、シグナルを表す例外も同様に処理する必要があります。

9.10 TI ツールチェーンにおけるアセンブリ言語オペレータ

これらの実装に関する詳細は TI ツールチェーンに関するものであり、ABI の一部ではありません。

TI コンパイラは特殊な組み込みアセンブラ関数を使用して、例外処理テーブル内の特定の式が特別な処理を受けるべきことをアセンブラに示します。

$EXIDX_FUNC

引数は PREL31 表現を使用してエンコードされる関数アドレスです。

$EXIDX_EXTAB

引数は PREL31 表現を使用してエンコードされる EXTAB ラベルです。

$EXTAB_LP

引数は PREL31 表現を使用してエンコードされるランディング パッド ラベルです。

$EXTAB_RTTI

引数は型を表すユニークな type_info オブジェクトのラベルです。(これらのオブジェクトは、ランタイム タイプの識別用に生成されます)。

$EXTAB_SCOPE

引数は関数へのオフセットです。この式はスコープ記述子で使用され、関数のどの部分で適用するかを示します。

Copyright © 2026 Texas Instruments Incorporated

Chapter 10
DWARF

C28x は、DWARF デバッグ情報フォーマット バージョン 3 (通称 DWARF3) を使用して、オブジェクト ファイル内にシンボリック デバッガ用の情報を表現します。DWARF3 の仕様は http://www.dwarfstd.org/doc/Dwarf3.pdf に公開されています。このセクションでは、その標準に加えて、C28x に固有の表現を定義しています。

10.1 DWARF レジスタの名前....................66
10.2 呼び出しフレーム情報....................67
10.3 ベンダー名....................68
10.4 ベンダー拡張....................68

Copyright © 2026 Texas Instruments Incorporated

10.1 DWARF レジスタの名前

DWARF3 レジスタではレジスタ名演算子を使用します (DWARF3 標準のセクション 2.6.1 を参照)。レジスタ名演算子のオペランドは、アーキテクチャ レジスタを表すレジスタ番号です。「表 3-1」に C28x レジスタの一覧が記載されています。表 10-1 では DWARF3 レジスタ番号/名前から C28x レジスタへのマッピングを定義します。

表 10-1. C28x の DWARF3 レジスタ番号

DOWARF レジスタ番号	C28x ISA レジスタ	サイズ	説明
0	AL	16 ビット	ACC アキュムレータ下半分
1	AH	16 ビット	ACC アキュムレータ上半分
2	PL	16 ビット	P の下半分
3	PH	16 ビット	P の上半分
4	AR0	16 ビット	XAR0 の下半分
5	XAR0	32 ビット	補助レジスタ 0
6	AR1	16 ビット	XAR1 の下半分
7	XAR1	32 ビット	補助レジスタ 1
8	AR2	16 ビット	XAR2 の下半分
9	XAR2	32 ビット	補助レジスタ 2
10	AR3	16 ビット	XAR3 の下半分
11	XAR3	32 ビット	補助レジスタ 3
12	AR4	16 ビット	XAR4 の下半分
13	XAR4	32 ビット	補助レジスタ 4
14	AR5	16 ビット	XAR5 の下半分
15	XAR5	32 ビット	補助レジスタ 5
16	AR6	16 ビット	XAR6 の下半分
17	XAR6	32 ビット	補助レジスタ 6
18	AR7	16 ビット	XAR7 の下半分
19	XAR7	32 ビット	補助レジスタ 7
20	SP	16 ビット	スタック ポインタ
21	TL	16 ビット	XT の下半分
22	T	16 ビット	XT の上半分
23	ST0	16 ビット	ステータス レジスタ 0
24	ST1	16 ビット	ステータス レジスタ 1
25	PC	22 ビット	プログラム カウンタ 0x3F FFC0
26	RPC	22 ビット	復帰プログラム カウンタ
27	--		内部使用のため予約済み
28	FP		XAR2 フレーム ポインタ
29	DP	16 ビット	データ-ページ ポインタ
30	SXM		ステータス レジスタ ビット
31	PM		ステータス レジスタ ビット
32	OVM		ステータス レジスタ ビット
33-35、38			内部使用のため予約済み
36	IFR	16 ビット	割り込みフラグ レジスタ
37	IER	16 ビット	割り込みイネーブル レジスタ
38	EALLOW		内部使用のため予約済み

Copyright © 2026 Texas Instruments Incorporated

FPU32 レジスタは FPU64 レジスタのサブセットです。たとえば、FPU32 では レジスタ 41 は 32 ビット レジスタ R0 を表し、FPU64 では R0 64 ビット レジスタの下位 32 ビットを表します。同様に、FPU32 では レジスタ 43 は 32 ビット レジスタ R0H を表し、FPU64 では R0 64 ビット レジスタの上位 32 ビットを表します。

表 10-2. C28x における FPU の DWARF3 レジスタ番号

DOWARF レジスタ番号	FPU32 レジスタ (すべて 32 ビット)	FPU64 レジスタ (特に記述のない限り 64 ビット)	説明
39-40	STF	STF (32 ビット)	フローティング ポインタ ステータス レジスタ
41	R0	R0H:R0L	
43	R0H		
45	R1	R1H:R1L	
47	R1H		
49	R2	R2H:R2L	
51	R2H		
53	R3	R3H:R3L	
55	R3H		
57	R4	R4H:R4L	
59	R4H		
61	R5	R5H:R5L	
63	R5H		
65	R6	R6H:R6L	
67	R6H		
69	R7	R7H:R7L	
71	R7H		
73-74	RB	RB (32 ビット)	リピート ブロック レジスタ
75-76	PSEUDO	PSEUDO (32 ビット)	内部使用のため予約済み
その他すべて			内部使用のため予約済み

10.2 呼び出しフレーム情報

デバッガは、関数の実行中にローカル変数を確認したり変更したりできる必要があります。

DWARF3 では、コンパイラが関数がデータを保存している場所 (レジスタかスタックか) を追跡することでこれを実現します。コンパイラは、この情報を DWARF3 標準のセクション 6.4 で定められたバイト コード形式で符号化します。これにより、デバッガはバイト コード化された言語を解釈することによって、過去の状態を段階的に再現できます。それぞれの関数のアクティベーションは、Canonical Frame Address (CFA) と呼ばれるベース アドレスと、そのときのマシン レジスタの内容に対応する値の集合によって表されます。アクティベーションの実行がどこまで進んでいるかがわかれば、デバッガは関数内のデータの位置を特定し、スタックをさかのぼって以前の状態 (過去の関数のアクティベーションを含む) を再現することができます。

DWARF3 標準では、大規模なスタック アンワインド テーブルを提案しています。このテーブルには、コード アドレスごとに 1 行、レジスタ (CFA を含む、仮想かどうかに関係なく、各レジスタ) ごとに 1 列が設けられています。各セルには、その時点 (コード アドレス) でのレジスタをアンワインドするための命令が格納されています。

CFA の定義と、状態を構成するレジスタの集合はいずれもアーキテクチャ固有です。

このレジスタ集合には、表 10-1 に記載されているすべてのレジスタが含まれており、DWARF レジスタ番号 (表の最初の列) でインデックス付けされます。

CFA については、この C28x ABI は DWARF3 標準で推奨されている規則に従い、呼び出しサイトの前のフレーム (呼び出し側の手続き) での呼び出し時点の SP (R1) の値として定義しています。

Copyright © 2026 Texas Instruments Incorporated

アンワインド テーブルには、C28x ISA によっては存在しないレジスタが含まれることがあります。そのため、プログラムを実行する ISA に呼び出しフレーム情報に記載されていないレジスタがある場合もあり得ます。この場合、インタープリタは次のように動作する必要があります。

- 呼び出し先保存レジスタは same-value ルールで初期化する。
- その他のレジスタは undefined ルールで初期化する。

10.3 ベンダー名

DW_AT_producer 属性は、オブジェクト ファイルを生成したツールチェーンを識別するために使用されます。この属性のオペランドは、ベンダーの接頭辞で始まる文字列です。以下の接頭辞は、特定のベンダー用に予約されています。

TI	テキサス インスツルメンツの C28x コード ジェネレータ ツール
GNU	GNU コンパイラ コレクション (GCC)

10.4 ベンダー拡張

DWARF 標準では、ツールチェーン ベンダーが、特定のアーキテクチャやツールチェーンに固有の情報を表現するための追加のタグおよび属性を定義することが許されています。TI もそれぞれの定義を行っています。このセクションでは、C28x アーキテクチャ全般に適用されるそれらの拡張について説明します。

残念ながら、利用可能な値の集合はすべてのベンダーで共有されているため、ABI がベンダー間で共通の標準値を規定することはできません。TI にできる最善の対応は、情報生成側に対して、同じ意味を持つ (可能であれば同じ値を使った) ベンダー固有のタグおよび属性を定義するよう要請し、情報消費側に対しては、ツールチェーンごとに異なるベンダー固有の値を解釈するために DW_AT_producer 属性を使用するよう求めることです。

C28x 向けの TI ベンダー固有の DIE タグは、表 10-3 に定義されています。TI ベンダー固有の属性は、表 10-3 に定義されています。

表 10-3. TI ベンダー固有のタグ

名称	値	説明
DW_TAG_TI_branch	0x4088	呼び出しと復帰を識別します

DW_TAG_TI_branch

このタグは、呼び出しおよび復帰として使用される分岐命令を識別します。このタグは、DW_TAG_subprogram DIE の子要素として生成されます。このタグには、分岐命令の位置を対応する DW_AT_lowpc 属性があります。

この分岐が関数呼び出しである場合、DW_AT_TI_call 属性がゼロ以外の値を持ちます。また、呼び出し先関数の名前を示す DW_AT_name 属性や、ポインタを介した呼び出しなどで呼び出し先が不明な場合には DW_AT_TI_indirect 属性を持つこともあります。

分岐が復帰である場合は、DW_AT_TI_return 属性がゼロ以外の値を持ちます。

表 10-4. TI ベンダー固有の属性

名称	値	クラス	説明
DW_AT_TI_symbol_name	0x2001	文字列	オブジェクト ファイル名 (マングル済み)
DW_AT_TI_return	0x2009	フラグ	分岐は復帰
DW_AT_TI_call	0x200A	フラグ	分岐は呼び出し
DW_AT_TI_asm	0x200C	フラグ	関数はアセンブリ言語
DW_AT_TI_indirect	0x200D	フラグ	分岐は間接呼び出し
DW_AT_TI_max_frame_size	0x2014	定数	アクティベーション レコード サイズ

DW_AT_TI_call、**DW_AT_TI_return**、**DW_AT_TI_indirect**:これらの属性は、前述のように、DW_TAG_TI_branch DIE に適用されます。

Copyright © 2026 Texas Instruments Incorporated

DW_AT_TI_symbol_name:この属性は、DW_symbol_name を持つ任意の DIE に出現することがあります。この属性は、変数または関数に関連付けられたオブジェクト ファイル レベルの名前、すなわち、ソースレベルの名前に対してツールチェーンによるマングリングやその他の変更が適用された後の名前を提供します。

DW_AT_TI_max_frame_size:この属性は、DW_TAG_subprogram DIE に出現することがあります。この属性は、関数のアクティベーションに必要なスタック領域の量 (バイト単位) を示します。この属性は、静的スタック深度解析を実行するダウンストリームのツールでの使用が想定されています。

Copyright © 2026 Texas Instruments Incorporated

Chapter 11

ELF オブジェクト ファイル (プロセッサ補足)

C28x ABI は ELF オブジェクト ファイル フォーマットを基盤としています。ELF の基本仕様は、包括的な System V ABI 仕様の第 4 章と第 5 章 (http://www.sco.com/developers/gabi/2003-12-17/contents.html) に記載されています。

以下の小節では、この仕様の第 4 章 (オブジェクト ファイル) に対する C28x プロセッサ固有の補足を示します。第 5 章 (プログラムのロードと動的リンク) に関するプロセッサ固有の補足は、本書の Chapter 12 に記載されています。

なお、C28x の ELF フォーマットでは、ターゲット アドレスを表すフィールドはネイティブ形式 (ワード単位) で表現されますが、ターゲット サイズを表すフィールドはバイト単位で表現されます。

11.1 登録済みベンダ名....71
11.2 ELF ヘッダー....71
11.3 セクション....72
11.4 シンボル テーブル....76
11.5 再配置....76

Copyright © 2026 Texas Instruments Incorporated

11.1 登録済みベンダ名

コンパイラ ツール セットは、ベンダ固有のシンボルを作成して使用します。潜在的な競合を回避するために、TI はベンダ固有の名前空間を定義して使用することを推奨しています。現在登録されているベンダと、その推奨される略称は、表 11-1 に記載されています。

表 11-1. 登録済みベンダ

名称	ベンダ
cxa、_ _cxa	C++ ABI 名前空間。C++ ABI で指定されるすべてのシンボルに適用されます。
c28xabi、_ _c28xabi	C28x EABI で指定されるシンボルの共通名前空間。
C28X	C28x で指定されたシンボルの共通名前空間。
TI、_ _TI	TI ツールチェーン固有のシンボル用に予約されています。これは、すべての TI プロセッサ ABI の複合名前空間も表します。
gnu、_ _gnu	GCC ツールチェーン固有のシンボル用に予約されています。

注

TI または__TI 仕様は、プロセッサ固有のセクション タイプや特殊セクションなどの名前を定義します。異なる TI プロセッサ間で共通性がある場合、これらのエンティティにはプロセッサごとに異なる名前を定義するのではなく、TI を使用して命名します。たとえば、例外テーブル インデックス テーブルのセクション タイプは、C28x では SHT_C28x_EXIDX、C2000 では SHT_C2000_EXIDX などにするのではなく、すべての TI プロセッサで SHT_TI_EXIDX となります。

11.2 ELF ヘッダー

ELF ヘッダーには、ファイルを解釈する際の手がかりとなる複数のフィールドが用意されています。その多くは System V ELF 仕様で定義されています。このセクションでは、それらに加えて、C28x に関する詳細を補足しています。

e_indent

16 バイトの ELF 識別フィールドは、対象のファイルがオブジェクト ファイルであることを識別し、ファイルの内容をデコード・解釈するための機種非依存データを提供します。「表 11-2」では、C28x オブジェクト ファイルに使用する値が規定されています。

表 11-2. ELF 識別フィールド

インデックス	シンボリック値	数値	備考
EI_MAG0		0x7f	System V ABI 準拠
EI_MAG1		E	System V ABI 準拠
EI_MAG2		L	System V ABI 準拠
EI_MAG3		F	System V ABI 準拠
EI_CLASS	ELFCLASS32	1	32 ビット ELF
EI_DATA	ELFDATA2LSB	1	リトル エンディアン
EI_VERSION	EV_CURRENT	1	
EI_ABIVERSION		0	

EI_OSABI フィールドは、特定のプラットフォーム規則によって無効にされない限り、ELFOSABI_NONE に設定されます。C28x ファミリのプラットフォームでは、EI_OSABI フィールドのデフォルト値は変更されず、常に ELFOSABI_NONE です。

e_type

現時点では、C28x に固有のオブジェクト ファイル タイプは存在しません。ET_LOPROC から ET_HIPROC までのすべての値は、この仕様の将来の改訂のために予約されています。

Copyright © 2026 Texas Instruments Incorporated

e_machine

本仕様に準拠するオブジェクト ファイルは、EM_TI_C2000 (141, 0x8D) の値を持つ必要があります。

e_entry

ELF の基本仕様では、アプリケーションにエントリ ポイントがない場合、このフィールドはゼロに設定することとされています。ただし、一部のアプリケーションではエントリ ポイントがゼロであることを必要とする場合があります (例:リセット ベクタを利用する場合)。

また、プラットフォーム標準によっては、実行可能ファイルは必ずエントリ ポイントを持つと規定されることがあり、その場合 e_entry にはゼロであってもエントリ ポイントが設定されます。

e_flags

このメンバーは、そのファイルに関連するプロセッサ固有のフラグを保持します。C28x 向けには e_flags に固有のフラグは定義されていません。

11.3 セクション

プロセッサ固有の特別なセクション インデックスは定義されていません。すべてのプロセッサ固有の値は、この仕様の将来の改訂に予約されています。

プログラムでは 0x80000000 を超える領域にコードやデータを配置することはできません。配置すると、再配置エラーが発生します。

11.3.1 セクション インデックス

C28x ABI では、特別なセクション インデックスは定義されません。

11.3.2 セクション タイプ

ELF 仕様では、セクション タイプ 0x70000000 以上がプロセッサ固有の値として予約されています。TI はこの空間を 2 つの部分に分割しました。すなわち、0x70000000 から 0x7EFFFFFF まではプロセッサ固有の値、0x7F000000 から 0xFFFFFFFF までは複数の TI アーキテクチャに共通の TI 固有セクション用の値です。組み合わされたものの一覧は、表 11-3 のとおりです。

これらすべてのセクション タイプが C28x ABI で使用されるわけではありません。一部は TI ツールチェーン固有ですが ABI の範囲外であり、また一部は TI ツールチェーンが C28x 以外のアーキテクチャ用に使用しています。それらは完全性のため、またタグ値を予約するためにここに記載されています。

表 11-3. ELF セクション タイプと TI セクション タイプ

名称	値	コメント
SHT_C28x_UNWIND	0x70000001	スタック アンワインディング用のアンワインド関数テーブル
SHT_C28x_PREEMPTMAP	0x70000002	DLL 動的リンク プリエンプション マップ (C28x では非対応)
SHT_C28x_ATTRIBUTES	0x70000003	オブジェクト ファイル互換性属性
SHT_TI_ICODE	0x7F000000	リンク時最適化用の中間コード
SHT_TI_XREF	0x7F000001	シンボル クロスリファレンス情報
SHT_TI_HANDLER	0x7F000002	予約済み
SHT_TI_INITINFO	0x7F000003	C 変数を初期化するための圧縮データ
SHT_TI_SH_FLAGS	0x7F000005	拡張セクション ヘッダー属性
SHT_TI_SYMALIAS	0x7F000006	シンボル エイリアス テーブル
SHT_TI_SH_PAGE	0x7F000007	セクションごとのメモリ空間テーブル

SHT_C28x_UNWIND はスタック アンワインディング用のアンワインド関数テーブルを含むセクションを識別します。詳しくは、Chapter 9 を参照してください。

Copyright © 2026 Texas Instruments Incorporated

SHT_C28x_ATTRIBUTES は、オブジェクト互換性属性を含むセクションを識別します。Chapter 13 を参照してください。

SHT_TI_ICODE は、リンク時の再コンパイルおよび最適化に使用される、ソース コードの TI 固有の中間表現を含むセクションを識別します。

SHT_TI_XREF は、シンボリック クロスリファレンス情報を含むセクションを識別します。

SHT_TI_HANDLER は現在は使用されていません。

SHT_TI_INITINFO は、C 変数の初期化用の圧縮データを含むセクションを識別します。このセクションには、ソース アドレスと宛先アドレスを示すレコードの表と、通常は圧縮形式で格納されたデータ自体が含まれます。Chapter 14 を参照してください。

SHT_TI_SH_FLAGS は、TI 固有のセクション ヘッダー フラグのテーブルを含むセクションを識別します。

SHT_TI_SYMALIAS は、シンボルを他のシンボル (外部で定義されている可能性もある) と同等であると定義するテーブルを含むセクションを識別します。TI リンカはこのテーブルを使用して、単に他の関数に転送するだけの単純な関数を除去します。

SHT_TI_SH_PAGE は、分離した (重複している可能性のある) アドレス空間 (ページ) を持つターゲットでのみ使用されます。このセクションには、他のセクションをページ番号と関連付けるテーブルがあります。このセクション タイプは C28x では使用されません。

11.3.3 拡張セクション ヘッダー属性

C28x では、TI のツールチェーンでは次のプロセッサ固有属性フラグを使用できます。

TI_SHF_NOINIT は、初期化されていない変数を含むセクションを示します。NOINIT 属性は、.TI.noinit セクションと .TI.persistent セクションにのみ適用できます。次に例を示します。

```
    ".TI.noinit"      SHT_NOBITS      TI_SHF_NOINIT
    ".TI.persistent" SHT_PROGBITS    TI_SHF_NOINIT
```

リンカは、これらのセクションに対して .cinit レコードを作成すべきではありません。

11.3.4 サブセクション

C28x オブジェクト ファイルでは、リンク時にセクションを結合する際の既定ルールの利便性を維持しつつ、より細かい制御が可能となるセクション命名規則が使用されます。名前にコロンを含むセクションは、*サブセクション*と呼ばれます。サブセクションは、すべての点において通常のセクションと同様に動作しますが、リンカがセクションを出力ファイルにまとめる際に、その名前が指針となります。サブセクションのルート名とは、コロンより前までの部分の名前を指します。接尾辞には、コロンの後に続くすべての文字列が含まれます。デフォルトでは、リンカはルートが一致するすべてのセクションを、同じ名前の 1 つのセクションに結合します。たとえば、.text、.text:func1、.text:func2 は、.text という 1 つのセクションに結合されます。このデフォルトの動作は、ツールチェーン固有の方法でユーザーが無効にできる可能性があります。

コロンが複数ある場合、セクションの結合は、最も右側のコロンから順に再帰的に行われます。たとえば、ユーザーが別途指定しない限り、デフォルトのルールでは .bss:func1:var1 と .bss:func1:var2 が結合され、それらがさらに .bss に結合されます。

ルート名が特別なセクション名と一致するサブセクションは、セクション 11.3.5 に定義されているとおり、その対応するセクションと同じ ABI 定義の性質を持ちます。たとえば、.text:func1 は .text セクションのインスタンスです。

11.3.5 特別なセクション

System V ABI は、他の基本ドキュメントやこの ABI のセクションと合わせて、特定用途のセクションをいくつか定義しています。表 11-4 には、C28x によって使用される専用セクションがまとめられており、機能ごとに分類されています。

セクション名は ABI によって規定されていません。特別なセクションは、名前ではなくタイプによって識別する必要があります。ただし、これらの命名規則に従うことでツールチェーン間の相互運用性が高まります。たとえば、これらの名前を使用

Copyright © 2026 Texas Instruments Incorporated

すれば、異なるコンパイラで生成された再配置可能ファイルをリンクする際に、独自のリンカ コマンドを書かなくても済む可能性が高くなります。

ABI は、名前が表のエントリと一致するセクションについて、その指定された目的に使用されなければならないことを義務付けています。たとえば、コンパイラがコードを .text というセクションに生成する義務はありませんが、.text という名前を持ちながらコード以外を格納するセクションを生成することはできません。

以下の表に示されるセクション名はすべて接頭辞です。その型や属性は、それらの文字列で始まる名前を持つすべてのセクションに適用されます。

表 11-4. C28x 特別なセクション

接頭辞	タイプ	属性
	コード セクション	
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
	データ セクション	
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.TI.noinit	SHT_NOBITS	TI_SHF_NOINIT
.TI.persistent	SHT_PROGBITS	TI_SHF_NOINIT
.const	SHT_PROGBITS	SHF_ALLOC
	例外処理データ セクション	
.C28x.exidx	SHT_C28x_UNWIND	SHF_ALLOC + SHF_LINK_ORDER
.C28x.extab	SHT_PROGBITS	SHF_ALLOC
	初期化セクションと終了セクション	
.init_array	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE
	ELF 構造	
.rel	SHT_REL	なし
.rela	SHT_RELA	なし
.symtab	SHT_SYMTAB	なし
.symtab_shndx	SHT_SYMTAB_SHNDX	なし
.strtab	SHT_STRTAB	SHF_STRINGS
.shstrtab	SHT_STRTAB	SHF_STRINGS
.note	SHT_NOTE	なし
	ビルド属性	
.C28x.attributes	SHT_C28x_ATTRIBUTES	なし
	シンボリック デバッグ セクション	
.debug (1)	SHT_PROGBITS	なし
	TI ツールチェーン固有のセクション	
.stack	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.sysmem	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.switch	SHT_PROGBITS	SHF_ALLOC
.binit	SHT_PROGBITS	SHF_ALLOC
.cinit	SHT_TI_INITINFO	SHF_ALLOC
.const:handler_table	SHT_PROGBITS	SHF_ALLOC
.ovly	SHT_PROGBITS	SHF_ALLOC
.ppdata	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.ppinfo	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.TI.crctab	SHT_PROGBITS	SHF_ALLOC
.TI.icode	SHT_TI_ICODE	なし
.TI.xref	SHT_TI_XREF	なし

Copyright © 2026 Texas Instruments Incorporated

表 11-4. C28x 特別なセクション (続き)

接頭辞	タイプ	属性
.TI.section.flags	SHT_TI_SH_FLAGS	なし
.TI.symbol.alias	SHT_TI_SYMALIAS	なし
.TI.section.page	SHT_TI_SH_PAGE	なし
	System V ABI のセクションであるが、**C28x EABI** によって使用されていないもの	
.comment		
.data1		
.dsbt		
.dynamic		
.dynstr		
.dynsym		
.far		
.fardata		
.fardata:const		
.fini		
.fini_array		
.gnu.version		
.gnu.version_d		
.gnu.version_r		
.got		
.hash		
.init		
.interp		
.line		
.neardata		
.plt		
.preinit_array		
.rodata		
.rodata1		
.tbss		
.tdata		
.tdata1		
.TI.tls_init		

(1) .debug_info や .debug_line といった名前の追加セクションも使用されます。.debug はセクション名の接頭辞であり、他のセクション名も同様に接頭辞として扱われます。その型や属性は、.debug で始まるすべてのセクションに適用されます。

前の表にある「**TI ツールチェーン固有のセクション**」は、TI ツールチェーンによってツールチェーン固有の用途に利用されます。ABI はこれらのセクションの利用を義務付けてはいませんが (相互運用性の観点からは利用が推奨されます)、名前自体は予約されています。

また、前の表にある「**System V ABI のセクションであるが、C28x EABI によって使用されていないもの**」は、System V ABI で規定されているものの、C28x ABI では利用も定義もされていません。その他のセクションは TI が他のデバイス向けに利用しており、これらの名前も予約されています。

11.3.6 セクションの整列

C28x のコードを含むセクションは、16 ビット (ワード) 境界に整列され、16 ビット (ワード) 境界までパディングされる必要があります。

Copyright © 2026 Texas Instruments Incorporated

プラットフォームの標準によっては、保証できる最大の整列に制限を設ける場合があります (通常は仮想メモリのページ サイズ)。

11.4 シンボル テーブル

プロセッサ固有のシンボル タイプやシンボル バインディングは存在しません。すべてのプロセッサ固有の値は、この仕様の将来の改訂に予約されています。

C28x ABI は、グローバルおよび弱シンボルの定義、ならびにシンボル値の意味に関して、ELF 仕様に従います。

11.4.1 シンボル タイプ

この仕様は、シンボル タイプに関して、ARM ELF 仕様に準拠します。具体的には、以下のとおりです。

- オブジェクト ファイルからエクスポートされるすべてのコード シンボル (バインディング STB_GLOBAL を持つシンボル) は、型 STT_FUNC を持たなければなりません。
- すべての extern データ オブジェクトは、型 STT_OBJECT を持たなければなりません。STB_GLOBAL データ シンボルが型 STT_FUNC を持つことはありません。
- 未定義シンボルの型は、STT_NOTYPE またはその期待される定義の型でなければなりません。
- 実行可能ファイルのセクションで定義されるその他のシンボルの型は、STT_NOTYPE にすることができます。

11.4.2 共通ブロック シンボル

ELF 仕様によれば、型が STT_COMMON のシンボルはリンカによって割り当てられます。

他のアドレッシング方式で参照される共通ブロック シンボルは、ELF の基本仕様で規定されているとおり、セクション インデックス SHN_COMMON を持ちます。

11.4.3 シンボル名

C またはアセンブリ言語のエンティティを命名するシンボルは、そのエンティティの名前を持つ必要があります。たとえば、*func* という C 関数は、*func* というシンボルを生成します。(以前の COFF ABI の場合のように、先頭にアンダースコアは付きません)。シンボル名では大文字と小文字が区別され、リンカによって完全一致で照合されます。

C28x コンパイラは、一時シンボルに対して次の命名規則に従います。

- パーサーによって生成されたシンボルには先頭に P が付きます
- オプティマイザによって生成されたシンボルには先頭に O が付きます
- Codegen によって生成されたシンボルには先頭に C が付きます

11.4.4 予約済みシンボル名

次のシンボルは、この仕様書および将来の改訂で予約されています。

- ローカル シンボル (STB_LOCAL) で、$ で始まるもの
- グローバル シンボル (STB_GLOBAL、STB_WEAK) で、表 11-1 に記載されたベンダー名のいずれかで始まるもの
- グローバル シンボル (STB_GLOBAL、STB_WEAK) で、$$Base または $$Limit のいずれかで終わるもの
- パターン ${Tramp}${I|L|S}[$PI]$$symbol に一致するシンボル
- コンパイラ生成の一時シンボルで、P、O、C のいずれかで始まるもの (セクション 4.7 に記載)

11.4.5 マッピング シンボル

マッピング シンボルは、プログラム データを分類するのに役立つローカル シンボルです。現在、ABI ではマッピング シンボルを使用する動作は指定されていません。ただし、次の 2 つの名前は将来の使用に備えて予約されています。$code および $data。

11.5 再配置

C28x の ELF 再配置は、再配置を実行するために必要なすべての情報が、再配置エントリ、オブジェクトフィールド、関連するシンボルに含まれるように定義されています。リンカは、再配置を実行する際に、オブジェクト フィールドをアンパッ

Copyright © 2026 Texas Instruments Incorporated

クする以外には命令をデコードする必要はありません。その結果、古い C28x COFF ABI よりもやや多くの再配置タイプが存在します。再配置タイプは COFF と ELF の間で互換性がありません。

再配置は、再配置可能なフィールドに対して操作を行うものとして指定されています。大まかに言えば、再配置可能フィールドとは、再配置によって影響を受けるプログラム イメージのビットです。このフィールドは、アドレス指定可能なコンテナの観点から定義され、そのコンテナのアドレスは再配置エントリの r_offset フィールドによって与えられます。フィールドのサイズとコンテナ内での位置、ならびに再配置後の値の計算は、再配置タイプによって規定されます。再配置操作は、再配置可能フィールドの抽出、操作の実行、結果の値のフィールドへの再挿入、という手順で構成されます。

ELF 再配置は Elf32_Rela 型または Elf32_Rel 型です。Rela エントリには、再配置計算に使用される明示的な加数が含まれています。Rel 型のエントリは、再配置可能フィールド自体を加数として使用します。特定の再配置は Rela のみとして識別されます。ほとんどの場合、これらは 32 ビット アドレスの上位 16 ビットに対応し、結果の値は、そのフィールド内で利用できない下位ビットからのキャリー伝播に依存します。Rela が指定されている場合、実装はこの要件を満たす必要があります。その他の再配置については、実装は Rel または Rela 型のいずれかを選択して使用できます。

アドレッシング モードが再配置に与える影響については、セクション 4.3 に簡単に記載されています。

11.5.1 再配置タイプ

再配置タイプは 2 つの表に記載されています。表 11-5 は、再配置型の数値値と、再配置後の値の計算の概要を示しています。表の後には、再配置型の説明とその使用例が記載されています。表 11-6 には、それぞれの型について、再配置フィールドの抽出および挿入、オーバーフロー検査、スケーリングやその他の調整を含む正確な計算が記載されています。

表 11-5 では以下の表記が使用されます。

S	再配置に関連付けられたシンボルの値。これは、再配置エントリの r_info フィールドに含まれるシンボル テーブル インデックスによって指定されます。
A	再配置可能フィールドの値を計算するために使用される加数。Elf32_rel 再配置の場合、A は 表 11-6 に従って再配置可能フィールドにエンコードされます。Elf32_Rela 再配置の場合、A は再配置エントリの r_addend フィールドによって与えられます。
PC	フィールドを含むコンテナのアドレス。これは再配置を含む命令のアドレスと同じでない場合があります。

表 11-5. C28x 再配置タイプ

名称	値	動作	制約
R_C28X_NONE	0		
R_C28X_ABS8	1	S + A	
R_C28X_ABS16	2	S + A	
R_C28X_ABS32	3	S + A	
R_C28X_ABSLO6	4	S+A	
R_C28X_ABSLO6_BLKD	4	S+A	重複しているが、アクセスがブロックされている
R_C28X_ABS22	5	S+A	
R_C28X_ABS22_BR	5	S+A	重複しているが、関数呼び出しでのみ使用される
R_C28X_HI6	6	S+A	Rela のみ
R_C28X_DP_HI10	7	S+A-PC	Rela のみ
R_C28X_DP_HI16	8	S+A	
R_C28X_PCREL16	9	S+A-PC	
R_C28X_PCREL8	10	S+A-PC	
R_C28X_HI16	11	S+A	Rela のみ
R_C28X_NEGWORD	12	特殊	
R_C28X_NEGBYTE	13	特殊	
R_C28X_ABS8_HI	14		
R_C28X_ABS13_SE16	15	特殊	
R_CLA_ABS16	16	S+A	

Copyright © 2026 Texas Instruments Incorporated

表 11-5. C28x 再配置タイプ (続き)

名称	値	動作	制約
R_C28X_ABSLO7	17	S+A	
R_C28X_PREL31	18	S+A-PC	

11.5.1.1 絶対再配置

絶対再配置は、シンボルの再配置アドレスを直接エンコードします。C28x の「ダイレクト」、「データ即値」、「プログラム即値」、「I/O 即値」アドレッシング モードは、すべて絶対再配置を必要とします。名前に「ABS」を含む再配置タイプは、すべて絶対再配置タイプです。

11.5.1.2 PC 相対再配置

PC 相対再配置では、符号付き PC 相対オフセットとしてアドレスをエンコードします。名称に「PCR」を含む再配置タイプは、PC 相対再配置タイプです。

アセンブラおよびリンカでは、ディスプレイスメントは、命令の開始アドレスではなく、表 11-6 で定義されている再配置のコンテナのアドレスを基準として計算されます。命令の個々のワードを読みながら PC が進むため、アドレッシング モードを実行する際にハードウェアが使用する有効な PC 値は、再配置コンテナのアドレスとは異なる可能性があります。これを補正するには、差分によって、アセンブラで再配置加数を調整する必要があります。

11.5.1.3 データ セクションにおける再配置

R_C28X_ABS16/32 再配置タイプは、シンボルの再配置後のアドレスを 16 ビットまたは 32 ビットのフィールドに直接エンコードします。これらの再配置は、初期化されたデータ セクション内のアドレスを再配置するために使用されます。R_C28X_ABS16/32 におけるフィールドの符号属性は未指定です。すなわち、これらの再配置タイプは符号付き値にも符号なし値にも使用されます。また、これらはここに示すように一部の命令再配置にも使用されます。

```
        .field X,32         ; R_C28X_ABS32
        .field X,16         ; R_C28X_ABS16
```

11.5.1.4 C28x 命令の再配置

以下のステートメントは、さまざまなタイプの再配置を実行します。各ステートメントの再配置タイプはコメントに示されています。

次のステートメントは直接アドレッシングの再配置を実行します。

```
    MOV DP, #var ; R_C28X_DP_HI10
    MOV AL, @var ; R_C28X_ABSLO6
    MOV AL, @@var ; R_C28X_ABSLO7
    MOVW DP, #var ; R_C28X_DP_HI16
```

次のステートメントは分岐の再配置を実行します。

```
    LCR function ; R_C28X_ABS22
    SB label ; R_C28X_PCREL8
    B label ; R_C28X_PCREL16
```

次のステートメントは特殊な再配置を実行します。

```
    MOV AH, #HI(var) ; R_C28X_HI6
    SUB loc16, #lab ; R_C28X_NEGWORD (on lab)
    SUBB AH, #lab ; R_C28X_NEGBYTE
    MOVB loc16, #lab, EQ ; R_C28X_ABS_HI (on lab)
```

次のステートメントは C2xLP 互換アドレッシングを使用して特殊な再配置を実行します。

```
    MPY #lab ; R_C28X_ABS13_SE16 ; sign extend 13->16 bits
```


Copyright © 2026 Texas Instruments Incorporated

次のステートメントは CLA 再配置を使用して特殊な再配置を実行します。

```
MI16TOF32 MR0, @lab ; R_CLA_ABS16
```

11.5.1.5 その他の再配置タイプ

R_C28X_NONE 再配置タイプは、何の処理も行いません。これは、参照元のセクションがリンクされている場合に参照先のセクションもリンクされるように、あるセクションから別のセクションへの参照を作成するために使用されます。

R_C28X_PREL31 は、例外処理テーブル内でコード アドレスをエンコードするために使用されます。セクション 9.2 を参照してください。

11.5.2 再配置操作

表 11-6 には、それぞれの再配置のエンコードおよび実行の方法についての詳細が記載されています。表では以下の表記を使用します:

F	再配置可能フィールド。フィールドはタプル **[CS, O, FS]** を使用して指定されます。ここで CS はコンテナ サイズ、O はコンテナの LSB からフィールドの LSB までの開始オフセット、FS はフィールドのサイズです。すべての値はビット単位です。表記 [x,y]+[z,w] は、再配置が非連続のビット範囲を占有しており、それらを連結してフィールドを形成することを示します。「F」が加数列に使用されている場合、フィールドがすでにアドレス空間の正確なサイズであることを示します。
R	再配置操作の算術結果
EV	再配置フィールドに書き戻されるエンコードされた値
SE(x)	x の符号拡張値。符号拡張は概念的にアドレス空間の幅まで行われます。
ZE(x)	x のゼロ拡張値。ゼロ拡張は概念的にアドレス空間の幅まで行われます。
r_addend	加数は RELA フィールドに格納する必要があり、再配置コンテナに格納できません。

オーバーフロー検査が有効になっている再配置型については、コンコードされた値 (符号がある場合は符号も含めて) が再配置可能フィールドにエンコードできない場合、オーバーフローが発生します。つまり、

- 符号付き再配置は、エンコードされた値が半開区間 [$-2^{FS-1}...2^{FS-1}$). の範囲外にある場合にオーバーフローします。
- 符号なし再配置は、エンコードされた値が半開区間 [$0 ... 2^{FS}$) の範囲外にある場合にオーバーフローします。
- 符号属性が *いずれでもよい* とされる再配置は、エンコードされた値が半開区間 [$-2^{FS-1}... 2^{FS}$) の範囲外にある場合にオーバーフローします。

表 11-6. C28x 再配置操作

再配置名	符号属性	コンテナ サイズ (CS)	Field [O, FS] (F)	加数 (A)	結果 (R)	オーバーフロー検査	エンコードされた値 (EV)
R_C28X_NONE	なし	32	[0.32]	なし	なし	なし	なし
R_C28X_ABS32	いずれでもよい	32	[0.32]	F	S + A	なし	R
R_C28X_ABS16	いずれでもよい	16	[0.16]	SE(F)	S + A	なし	R
R_C28X_ABS8	いずれでもよい	8	[0.8]	SE(F)	S + A	あり	R
R_C28X_PCR16	符号付き	16	[0.16]	SE(F)	S + A - P	なし	R
R_C28X_ABS_HI16	なし	16	[0.16]	r_addend	S + A	なし	R >> 16
R_C28X_PREL31	符号付き	32	[0.31]	SE(F)	S + A - P	なし	R >> 1

11.5.3 未解決の弱参照の再配置

未定義の弱シンボルを参照する再配置は、以下のように処理されます。

- 弱関数への参照は、イミディエイト アドレッシング モードを使用して実装します。
- 絶対再配置型 (R_C28x_ABS*) で使用される場合、参照は 0 に解決されます。

その他すべてのケースは、ABI に準拠しません。

Copyright © 2026 Texas Instruments Incorporated

Chapter 12

ELF プログラムのロードとリンク (プロセッサ補足)

プログラムのロードとは、一般に ELF ファイルとして表現されたプログラムを取り込み、実行を開始するまでの手順を指します。この処理は性質上、プラットフォームやシステムに依存します。

システムによっては、要件に応じてこれらの仕組みの一部だけを利用する場合もあります。

この ABI の該当部分は、System V ABI 標準の第 5 章 (http://www.sco.com/developers/gabi/2003-12-17/contents.html) に基づいています。ここで説明されているのは、実行可能なプログラムを生成する際のオブジェクト ファイル情報やシステムの動作です。このセクションでは、その標準に対する補足として、C28x ベースのシステムに共通する要素に関するプロセッサ固有の情報を示しています。

12.1 プログラム ヘッダー.......... 81
12.2 プログラムのロード.......... 82

Copyright © 2026 Texas Instruments Incorporated

12.1 プログラム ヘッダー

プログラム ヘッダーには、以下のフィールドがあります。

p_type

C28x は、プログラム ヘッダーの p_type フィールドに対してプロセッサ固有のセグメント型を定義しません。

p_vaddr、p_paddr

C28x は、現在のところ仮想アドレスを使用していません。p_vaddr フィールドと p_paddr フィールドは共に、セグメントの実行アドレスを示します。一方のアドレスでロードされ、別のアドレスにコピーされて実行されるセグメントは、オブジェクト ファイル内では 2 つの異なるセグメントとして表現されます。1 つは、セグメントのコードまたはデータを含み、そのアドレス フィールドがロード アドレスを示すロード イメージ セグメントで、もう 1 つは、初期化されておらず、アドレス フィールドが実行アドレスを参照する実行イメージ セグメントです。アプリケーションは、適切なタイミングでロード イメージの内容を実行アドレスにコピーする責任を負います。

p_flags

C28x に定義されるプロセッサ固有のセグメント フラグはありません。

p_align

System V ABI に記載されているように、ロード可能セグメントはファイル内で整列され、p_vaddr (メモリ上のアドレス) と p_offset (ファイル内のオフセット) が、p_align を法として合同になるように配置されます。仮想メモリを使用するシステムでは、p_align は通常ページ サイズを指定します。特定のプラットフォームについて明示されていない限り、C28x における p_align の意味および設定は未定義です。

12.1.1 ベース アドレス

C28x は、System V ABI 標準の第 5 章「ベース アドレス」セクションに記載されている位置独立コードをサポートしていません。

位置独立でないセグメントは、指定されたアドレスにロードされるか、ロード時に再配置する必要があります。

12.1.2 セグメントの内容

ベース ABI (このセクション) は、どのセグメントが存在しなければならないか、またはその内容が何であるかについて、いかなる要件も定義しません。たとえば、C28x プログラムは任意の数のコードおよびデータ セグメントを含むことができ、これには複数のコード セグメント、複数の絶対データ セグメントが含まれます (Chapter 4 と Chapter 5 を参照)。特定のプラットフォームには独自の要件がある場合があります。たとえば、一部の高水準オペレーティング システムでは、プログラムが持てるものを、コード セグメント 1 つとデータ セグメント 1 つのみ、あるいは両方を兼ねた 1 つのセグメントのみに制限される場合があります。

12.1.3 スレッドローカル ストレージ

ABI は現在、スレッドローカル ストレージの標準的なしくみを規定していません。

Copyright © 2026 Texas Instruments Incorporated

12.2 プログラムのロード

プログラムのロードと実行の開始には、多くのシステム固有の側面があります。このセクションでは、ほとんどのシステムに共通するプロセスを概観しつつ、特に C28x に固有の要素に焦点を当てて説明します。

これらの処理は、ホスト ベースのローダなどのオフライン エージェント、オペレーティングシステムなどのターゲット システム上のランタイム コンポーネント、あるいは自己起動コードのようにプログラム自体にリンクされたライブラリ コンポーネントなどの組み合わせによって実行できます。

一般に、プログラムのロードは、プロセス イメージの作成、実行環境の初期化、プログラムの実行、終了処理の実行の 4 つの処理系列で構成されます。

プロセス イメージの作成は、プログラムおよびそのサブコンポーネントをメモリにコピーし、必要に応じて再配置を実施する作業です。これらの手順は、ホスト ベースのローダやオペレーティング システムなどの外部エージェントによって実行される必要があります。

実行環境の初期化とは、プログラムの実行が開始される前 (すなわち、main が呼び出される前) に行われる処理を指します。これらの手順は、外部エージェントまたはプログラム自身のいずれかによって実行できます。同様に、終了処理は main がリターンする (あるいは exit を呼び出す) 際に行われ、これも外部またはプログラム自身のいずれかで実行できます。

表 12-1、表 12-2、表 12-3 には、プログラムを生成、初期化、終了するための手順が列挙されています。手順の順序は絶対的なものではありませんが、順守しなければならない依存関係があります。

表 12-1. ELF 実行可能ファイルからプロセス イメージを作成する手順

ステップ	
1.	各ロード可能セグメントのアドレスを決定します。ベアメタルまたは非動的システムでは、これは通常、セグメントのプログラム ヘッダーの p_vaddr フィールドにあるアドレスです。その他の考慮事項については、セクション 12.1 に記載されています。
2.	メモリ システムを初期化し、メモリを割り当てます。
3.	各セグメントの内容をメモリにコピーします。セグメントに未使用領域がある場合 (すなわち、ファイル サイズがメモリ サイズより小さい場合)、未使用領域を 0 に初期化します。
.	コマンドライン引数と環境変数をマーシャリングします。この手順はプラットフォーム固有です。

表 12-2. 実行環境を初期化する手順

ステップ	
.	SP を設定します。SP (R1) は、シンボル _ _TI_STACK_END の値に設定され、8 バイト境界に正しく整列させる必要があります。
.	変数を初期化します。自己起動型の ROM ベース システムでは、RAM ベース (読み取り / 書き込み) の変数を初期値で初期化するためのメカニズムが必要です。このメカニズムはツールチェーンとプラットフォームに固有です。TI ツールに実装されているそのようなメカニズムの 1 つが、Chapter 14 に記載されています。
.	初期化呼び出しを実行します。一般的に、これはモジュール内で定義されたグローバル オブジェクトのコンストラクタへの呼び出しです。初期化関数へのポインタはテーブルに格納されます。一ブルは一対のグローバル シンボル。_ _TI_INITARRAY_Base と _ _TI_INITARRAY_Limit によって区切られます。
.	エントリ ポイントに分岐します。エントリ ポイントは ELF ヘッダーの e_entry フィールドで指定されます。OS などの基盤となるソフトウェア ファブリックがあるシステムでは、エントリ ポイントは通常 main 関数です。ベアメタル システムでは、この表に示された初期化手順の大部分は、main の前に実行されるライブラリ コードを通じてプログラム自身が実行することがあります。その場合、ELF のエントリ ポイントはそのコードのアドレスです。たとえば TI ツールは _c_int00 と呼ばれるエントリ ルーチンを提供しており、プロセス イメージが作成されるとステップ 10 (SP の設定) から一連の処理を開始します。

表 12-3. 終了手順

ステップ	
.	atexit 呼び出しを実行します。atexit によって登録された関数は、登録の逆順で呼び出されます。

Copyright © 2026 Texas Instruments Incorporated

Chapter 13

ビルド属性

ARM ABIv2 仕様の ABI 仕様は、ビルド時のオプションを記録し、リンカが再配置可能ファイルの互換性を強制できるようにするために、ビルド属性のメカニズムを定義しています。ELF 仕様は、ARM ABIv2 ビルド属性仕様に記載されたものと同じ構造を使用してビルド属性をエンコードします。これらは『*ABI for the ARM Architecture*』の「ARM Addenda」および「Errata」に文書化されており、文書番号 ARM IHI0045A として 2007 年 11 月 13 日に公開されています。

13.1 ビルド属性について..84
13.2 C28x ABI ビルド属性サブセクション..84
13.3 ビルド属性タグ..85

Copyright © 2026 Texas Instruments Incorporated

13.1 ビルド属性について

ビルド属性は、ベンダー固有または ABI 固有に分類されます。このセクションでは、ABI 固有のビルド属性について説明します。ベンダーは、ツール チェーン固有の追加属性を自由に実装できます。

すべての ABI 準拠の再配置可能ファイルには、SHT_C28x_ATTRIBUTES タイプ (0x70000003)のビルド属性セクションが含まれている必要があります。これには、従来的に C28x.attributes という名前が付けられています。実行可能ファイルには、オプションでビルド属性セクションを含めることができます。準拠ツールは、セクション タイプのみを使用して、ビルド属性セクションを認識する必要があります。

ビルド属性セクションは 'A' (0x41) の値を持つ 1 バイトのバージョン指定子で構成され、その後にベンダー サブセクションのシーケンスが続きます。

'A'	vendor subsection	vendor subsection	…

各サブセクションの形式は次のとおりです。

length	vendor name	0	vendor data
uint32	char[ ]	uint8	

長さフィールドは、サブセクション全体の長さをバイト単位で指定します。ベンダー名「C28x」は、ABI 指定の属性用に予約されています。他のサブセクションでは、ベンダー データの形式と解釈はベンダー固有です。

13.2 C28x ABI ビルド属性サブセクション

この ABI で指定された属性は、ベンダ文字列 C28x を持つサブセクションに記録されます。ツールチェーンは、これらの属性のみを使用して再配置可能ファイル間の互換性を判断する必要があります。ベンダ固有の情報は、この目的のために提供される Tag_Compatibility 属性で許可されている場合以外は使用しないでください。

C28x サブセクションのベンダ データには、任意の数の属性ベクトルが含まれています。属性ベクトルは、ファイル全体に適用されるか、リストされているセクションまたはシンボルにのみ適用されるかを指定する範囲タグで始まります。属性ベクトルのフォーマットは次の 3 つのうちのいずれかになります。

1	length	(omitted)		attributes	Apply to file
2	length	section numbers	0	attributes	Apply to specified sections
3	length	section numbers	0	attributes	Apply to specified sections
ULEB128	uint32	ULEB128[ ]	ULEB128[ ]	See below	

長さフィールドは、他のフィールドを含む属性ベクトル全体の長さをバイト単位で指定します。シンボル フィールドとセクション番号フィールドは、セクション インデックスまたはシンボル インデックスのシーケンスであり、0 で終了します。

属性ベクトル内の属性は、タグと値のペアのシーケンスとして表されます。タグは ULEB128 定数で表されます。値は ULEB128 定数または NULL で終了する文字列のいずれかです。

ファイル範囲内のタグを省略する効果は、パラメータ タイプに応じて、値 0 または "" を含む効果と同じです。

認識されないタグをコンシューマがスキップできるように、パラメータ タイプは偶数番号のタグでは ULEB128、奇数番号のタグでは NULL で終了する文字列として標準化されています。タグ 1、2、3 (範囲タグ) および 32 (Tag_ABI_Compatibility) はこの規則の例外です。

ABI が進化するにつれて、新しい属性が追加されることがあります。古いツールチェーンが、理解できない属性を含むファイルを確実に処理できるようにするために、ABI は次の規則を採用しています。

Copyright © 2026 Texas Instruments Incorporated

- タグ 0～63 は、コンシューミング ツールで理解しておく必要がある。この範囲内の不明なタグが検出された場合、コンシューミング ツールはエラーを生成する場合があります。
- タグ 64～127 ではコンシューマが安全に無視できる情報を伝える。
- N が 128 以上の場合、タグ N のプロパティはタグ N を 128 で割ったときの余りと同じになる。

13.3 ビルド属性タグ

OFBA_C28XABI_Tag_C28x (=4), ULEB128

このタグは、ファイルにエンコードされた命令を実行できる C28x ISA を指定します。次の値が定義されています。

0	C28x コードなし
1	C28x コードあり

リンクするには、ビルド内のすべてのオブジェクト ファイルが同じ ISA タグを持つ必要があります。

OFBA_C28XABI_Tag_Code_FPU, (=6), ULEB128

0	FPU コードなし
1	FPU32 コードあり
2	FPU64 コードあり

このタグは、対応する FPU のバージョンを指定します (存在する場合)。

リンクするには、ビルド内のすべてのオブジェクト ファイルを同じ FPU サポートでコンパイルする必要があります。

OFBA_C28XABI_Tag_CLA, (=8), ULEB128

0	CLA なし。
1	CLA 0 対応
2	CLA 1 対応
3	CLA 2 対応

このタグは、対応する CLA のバージョンを指定します (存在する場合)。

リンクするには、ビルド内のすべてのオブジェクト ファイルを同じ CLA サポートでコンパイルする必要があります。

OFBA_C28XABI_Tag_TMU, (=10), ULEB128

0	TMU なし
1	TMU 0 対応

このタグは、対応する TMU のバージョンを指定します (存在する場合)。

リンクするには、ビルド内のすべてのオブジェクト ファイルを同じ TMU サポートでコンパイルする必要があります。

OFBA_C28XABI_Tag_VCU, (=12), ULEB128

0	VCU なし。
1	VCU 0 対応
2	VCU 2 対応
3	VCU 2.1 対応

このタグは、対応する VCU のバージョンを指定します (存在する場合)。

Copyright © 2026 Texas Instruments Incorporated

リンクするには、ビルド内のすべてのオブジェクト ファイルを同じ VCU サポートでコンパイルする必要があります。

Copyright © 2026 Texas Instruments Incorporated

OFBA_C28XABI_Tag_float_args, (=14), ULEB128

0	浮動小数点引数なし
1	浮動小数点引数あり

このタグは、単精度浮動小数点引数を使用するかどうかを指定します。

OFBA_C28XABI_Tag_double_args, (=16), ULEB128

0	倍精度浮動小数点引数なし
1	倍精度浮動小数点引数あり

このタグは倍精度浮動小数点引数を使用するかどうかを指定します。

表 13-1 は、ABI によって定義されるビルド属性タグの概要です。

表 13-1. C28x ABI ビルド属性タグ

タグ	タグ値	パラメータ タイプ	互換性ルール
Tag_File	1	uint32	
Tag_Section	2	uint32	
Tag_Symbol	3	uint32	
OFBA_C28XABI_Tag_C28x	4	ULEB128	オブジェクト ファイル間で混在させることはできません。
OFBA_C28XABI_Tag_FPU	6	ULEB128	オブジェクト ファイル間で混在させることはできません。
OFBA_C28XABI_Tag_CLA	8	ULEB128	オブジェクト ファイル間で混在させることはできません。
OFBA_C28XABI_Tag_TMU	10	ULEB128	オブジェクト ファイル間で混在させることはできません。
OFBA_C28XABI_Tag_VCU	12	ULEB128	オブジェクト ファイル間で混在させることはできません。
OFBA_C28XABI_Tag_float_args	14	ULEB128	なし (複数のオブジェクト ファイル間で混在可能)。
OFBA_C28XABI_Tag_double_args	16	ULEB128	なし (複数のオブジェクト ファイル間で混在可能)。

Copyright © 2026 Texas Instruments Incorporated

Chapter 14
コピー テーブルと変数の初期化

このセクションでは、まずコピー テーブルのメカニズムの概要を説明し、続いてその際に使用されるデータ構造の仕様を示します。最後に、TI ツールチェーンにおける変数初期化の実装が、この基本的なコピー テーブル機能をどのように利用しているかを解説します。

14.1 テーブルのコピーについて 89
14.2 コピー テーブルのフォーマット 91
14.3 圧縮データ フォーマット 92
14.4 変数の初期化 93

Copyright © 2026 Texas Instruments Incorporated

14.1 テーブルのコピーについて

コピー テーブルとは、TI ツール チェーンの一般的な機能を指し、オフライン ストレージからオンライン ストレージにデータを移行しやすくします。オフライン ストレージとは、通常、プログラムがロードされる場所を指します。ROM や低速メモリなどがあります。オンライン ストレージとは、通常、プログラムの実行時にデータが存在する場所を指します。コピーされるデータは、コードまたは変数のいずれかです。コピー テーブルという用語は、コピーするオブジェクトが登録される送信元アドレスと宛先アドレスのテーブルを指します。ライブラリ関数の形式のランタイム コンポーネントもあり、プログラムの呼び出しに応じてテーブルを読み取り、コピーを実行します。

コピー テーブルには多数のアプリケーションがありますが、最も一般的なアプリケーションは次の 2 つです。

- 初期化 — ROM ベースのベアメタル システムでは、初期化された読み取り/書き込み変数は、プログラムの起動時に ROM から RAM にコピーする必要があります。
- オーバーレイ — プログラムが実行されると、異なるコード コンポーネントとデータ コンポーネントがメモリ領域との間でスワップイン/スワップアウトされます。

コピー テーブル メカニズムは ABI の一部ではありません。初期化された変数が初期値を取得するための手段は、リンカとランタイム ライブラリの間のコントラクトにより、同じツール チェーンからのものである必要があります。しかし、他のツール チェーンが TI のメカニズムに従うことにも利点がある場合があります。または、このフォーマットを認識するためのダウンストリーム ツールが必要になる場合もあります。そのため、ここにそれが記載されています。

下図に、一般的なメカニズムを示します。オブジェクト ファイルには、初期化されたセクション .mydata が含まれています。リンク時に、ユーザーは .mydata に個別のロード アドレスと実行アドレスを指定し、コピー テーブル エントリを作成するように指定します。リンカは .mysect からデータを削除し、それを未初期化セクションにし、そのアドレスを実行位置として割り当てます。.mydata.load1 という名前の新しい初期化セクションが作成されます。このセクションには、.mydata のデータがエンコードされた形式で格納され、ロード場所に配置されます。実行時にデータをデコードしコピーするためにランタイム ライブラリから copy_in と呼ばれる関数にリンクし、追加のフォーマット固有のヘルパー関数もリンクします。最後に、コピー テーブルを含むセクション (例では .ovly1) を作成します。コピー テーブルは、ソース データと宛先アドレスを指すコピー レコードのシーケンスであり、コピー関数が正しいデコード ヘルパー関数を選択するために使用するハンドラ テーブル (図には示されていません) を作成します。

実行時に、アプリケーションは copy_in を呼び出してデータを解凍し、コピーします。copy_in の引数は、セクションに関連付けられたコピー テーブルのアドレスです。この関数はテーブルを解析し、指定されたコピー操作を実行します。

複数のオブジェクトをエンコードし、copy-in に登録することができます。各オブジェクトは、.ovly [1] セクションに独自のコピー テーブルを生成します。

[1] コピー テーブル セクションおよび圧縮ソース データのセクション名は、リンカによって任意に決定されます。

Copyright © 2026 Texas Instruments Incorporated

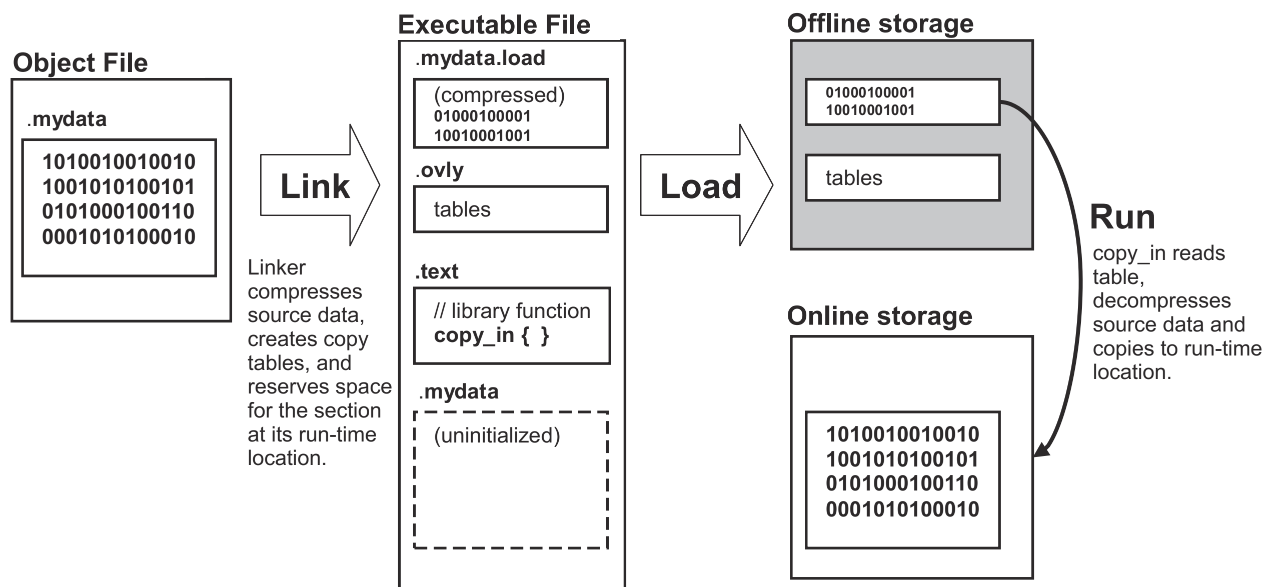

図 14-1. テーブ のコピーの概要

次のようないくつかのバリエーションがあります。

- **複数のオブジェクト**。複数のセクションを 1 つのコピー テーブルに登録できます。これは、オーバーレイに関連付けられたすべてのコードとデータを 1 回の呼び出しでコピーできるようにするためです。オーバーレイを構成する個別のコンポーネントの数をアプリケーションが認識する必要はありません。1 つのコピー テーブルに複数のコピー レコードを含めることができます。各コピー レコードは、連続したコードまたはデータのチャンクの copy-in を制御します。
- **圧縮なし**。圧縮はオプションです。圧縮が有効になっていない場合、セクションの個別のロード バージョンは必要ありません。リンカは単純に初期化されたセクションに別のロード アドレスと実行アドレスを割り当てます。
- **初期化**。変数の初期化は、一般的なメカニズムの特別な場合です。初期化のためのコピー レコードの形式は少し異なり、.cinit という別のセクションに格納され、copy-in と同様にゼロ初期化をサポートします。セクション 14.4 で、これらの詳細について説明します。
- **ブート時の Copy-In**。.binit という特別なセクションには、アプリケーションの起動時に自動的に呼び出されるコピー テーブルが含まれています。これは初期化の場合と似ていますが、初期化は言語実装の一部であり、ツール チェーンに組み込まれていますが、ブート時の copy-in は厳密にアプリケーション レベルの操作です。

Copyright © 2026 Texas Instruments Incorporated

14.2 コピー テーブルのフォーマット

コピー テーブルには次のフォーマットがあります。

```
typedef struct
{
   uint16      rec_size;
   uint16      num_recs;
   COPY_RECORD recs[num_recs];
} COPY_TABLE;
```

rec_size は、テーブル内の各コピー レコードのサイズを 16 ビット単位で指定する 16 ビット符号なし整数です。

num_recs は、テーブル内のコピー レコードの数を指定する 16 ビット符号なし整数です。

テーブルの残りの部分はコピー レコードのベクトルで構成されます。各レコードのフォーマットは次のとおりです。

```
typedef struct
{
   uint32   load_addr; /* 32-bit storage for data or code pointer */
   uint32   run_addr;  /* 32-bit storage for data or code pointer */
   uint32   size;
} COPY_RECORD;
```

load_addr フィールドはオフライン ストレージ内のソース データのアドレスです。

run_addr フィールドはデータがコピーされる宛先アドレスです。

size フィールドはオーバーロードされます。

- サイズがゼロの場合、ロード データは圧縮されます。ソース データには、そのサイズを示すフォーマット固有のエンコーディングがあります。この場合、ソース データの最初の 16 ビットが圧縮フォーマットをエンコードします。このフォーマットは、使用する各フォーマットのハンドラ ルーチンへのポインタのテーブルであるハンドラ テーブルへのインデックスとしてエンコードされます。
- サイズがゼロでない場合、ソース データはコピーするデータの正確なイメージです。つまり、圧縮されません。コピーイン操作では、*サイズ* 16 ビット単位のデータをロード アドレスから実行アドレスにコピーするだけです。

ソース データの残りの部分はフォーマット固有です。コピーイン ルーチンでは、ソース データの最初の 16 ビットを読み取ってフォーマット/インデックスを判断し、その値を使用してハンドラ テーブルにインデックスを作成し、ハンドラを呼び出してデータの解凍とコピーを完了します。

ハンドラ テーブルには次のフォーマットがあります。

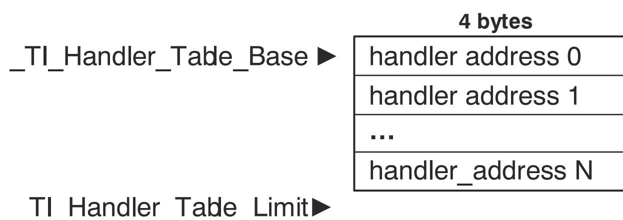

図 14-2. ハンドラ テーブルのフォーマット

コピーイン ルーチンでは、説明のとおり特別なリンカ定義シンボルを使用してテーブルを参照します。ハンドラ インデックスの割り当ては固定されていません。リンカは、そのアプリケーションに必要な解凍ルーチンに応じて、各アプリケーションのインデックスを再割り当てします。ハンドラ テーブルは、実行可能ファイルの .cinit セクションに生成されます。

TI ツールチェーンのランタイム サポート ライブラリには、サポートされているすべての圧縮フォーマットに対応するハンドラ関数が含まれています。ハンドラ関数への最初の引数は 16 ビット インデックスの後の 16 ビットを指すアドレスです。2 番目の引数は宛先アドレスです。

Copyright © 2026 Texas Instruments Incorporated

「copy_in 関数のリファレンス実装」で copy_in 関数のリファレンス実装を紹介しています。

copy_in 関数のリファレンス実装

```
typedef void (*handler_fptr)(const unsigned char *src, unsigned char *dst);
extern int __TI_Handler_Table_Base;
void copy_in(COPY_TABLE *tp)
{
   unsigned short i;
   for (i = 0; i < tp->num_recs; i++)
   {
      COPY_RECORD crp = tp->recs[i];
      const unsigned char *ld_addr = (const unsigned char *)crp.load_addr;
      unsigned char       *rn_addr = (unsigned char *)crp.run_addr;
      if (crp.size)       // not compressed, just copy the data.
         memcpy(rn_addr, ld_addr, crp.size);
      else                // invoke decompression routine
      {
         unsigned char index = *ld_addr++;
         handler_fptr  hndl  = ((handler_fptr *)(__TI_Handler_Table_Base))[index];
        (*hndl)(ld_addr, rn_addr);
      }
   }
}
```

14.3 圧縮データ フォーマット

抽象的に、圧縮ソース データのフォーマットは次のとおりです。

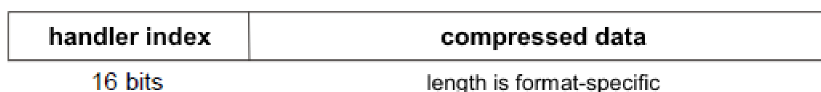

図 14-3. 圧縮ソース データ フォーマット

ハンドラ インデックスでは、データの残りを解釈するデコード関数を指定します。現在、コピー テーブルでは Run-length encoding (RLE) と Lempel-Ziv Storer and Szymanski compression (LZSS) の 2 つの圧縮フォーマットがサポートされています。

14.3.1 RLE

16 ビットのインデックスに続くデータは、ラン レングス エンコーディング (RLE) 形式で圧縮されます。C28x は単純なラン レングス エンコーディングを使用し、次のアルゴリズムで解凍できます。

1. 最初の 16 ビットを読み取り、それを区切り文字 (D) として割り当てます。
2. 次の 16 ビット (B) を読み取ります。
3. B != D の場合、B を出力バッファにコピーし、手順 2 に戻ります。
4. 次の 16 ビット (L) を読み取ります。
5. L > 0 かつ L < 4 の場合、D を出力バッファに L 回コピーします。手順 2 に戻ります。
6. L = 4 の場合、次の 16 ビット (B') を読み取ります。B' を出力バッファに L 回コピーします。手順 2 に戻ります。
7. L == 0 の場合、次の 16 ビット (L) を読み取ります。
   - L == 0 の場合、データの終端に到達したため、手順 10 に進みます。
   - L が 32 ビットのラン レングスの最上位 16 ビットである場合は、これを L.hi として保存し、次の 16 ビットを読み取ります。これが 32 ビット ランレングスの最下位 16 ビットである L.lo です。L.hi と L.lo を連結して、32 ビットのラン レングス L を形成します。
8. 次の 16 ビットの反復文字 (C) を読み取ります。
9. C を出力バッファに L 回コピーします。手順 2 に戻ります。
10. 処理の終了。

TI ツールチェーンにおける RLE ハンドラ関数は __TI_decompress_rle と呼ばれます。

Copyright © 2026 Texas Instruments Incorporated

14.3.2 LZSS フォーマット

16 ビット インデックスに続くデータは、LZSS 圧縮を使用して圧縮されます。TI ツールチェーン内の LZSS ハンドラ関数は _ _TI_decompress_lzss と呼ばれます。フォーマットの詳細については、RTS ソース コード内のこの関数の実装を参照してください。

LZSS の解凍アルゴリズムは次のとおりです。

1. 次の LZSS コンコード済みパケットの開始を示すエンコーディング フラグ (F) として 16 ビットを読み取ります。
2. F 内の各ビット (B) について、最下位ビットから最上位ビットまで順に次を実行します。
   a. (B & 0x1) ならば、次の 16 ビットを読み取り、それを出力バッファに書き込みます。その後、F 内の次のビット (B) に進み、この手順を繰り返します。
   b. それ以外の場合は、次の 16 ビットを temp (T) に読み取り、長さ (L) = (T & 0xf) + 2、オフセット (O) = (T >> 4) とします。
      i. L == 17 ならば、次の 16 ビット (L') を読み取り、L += L' とします。
      ii. O == LZSS end of data (LZSS_EOD) ならば、データの終端に到達したことになり、アルゴリズムが終了します。
      iii. 位置 (P) = 出力バッファ − オフセット (O) − 1 において、位置 P から L バイトを読み取り、それを出力バッファに書き込みます。
      iv. ステップ 2a に戻ります。

14.4 変数の初期化

セクション 4.1 に記載されているように、初期化された読み取り / 書き込み変数はオブジェクト ファイルの専用セクション (.data など) にまとめられます。このセクションには、プログラムが起動した直後の初期状態を示すイメージが含まれています。

TI ツールチェーンは、このようなセクションをロードするための 2 つのモデルをサポートしています。いわゆる *RAM モデル* では、ローダなどの未特定の外部エージェントが、実行可能ファイルから読み取り / 書き込みメモリへのデータ転送を担当します。これは、OS ベースのシステムや一部のブートロードされるシステムで使用される、一般的な直接初期化モデルです。

もう 1 つの *ROM モデル* は、OS や他のローダのサポートなしにコールド スタートが可能でなければならないベアメタル組込みシステム向けに設計されています。プログラムの初期化に必要なデータはすべて、永続的なオフライン ストレージ (ROM) に格納され、起動時に RAM の所定の位置にコピーされます。TI ツールチェーンは、コピー テーブル機能 (Chapter 14 を参照) を活用することによってこれを実現しています。初期化のしくみは概念的にはコピー テーブルと似ていますが、細かい点で若干異なります。

図 14-4 は、ROM モデルにおける変数初期化の概念的な処理を示しています。このモデルでは、リンカが初期化された変数を含むセクションからデータを *削除* します。これらのセクションは、未初期化セクションとなり、ランタイム アドレスで RAM に割り当てられます (たとえば .bss と同様です)。リンカは、初期化データを .cinit (C 初期化用) と呼ばれる特殊なセクションにエンコードし、ランタイム ライブラリの起動時コードがそれをデコードして実行アドレスへコピーします。

Copyright © 2026 Texas Instruments Incorporated

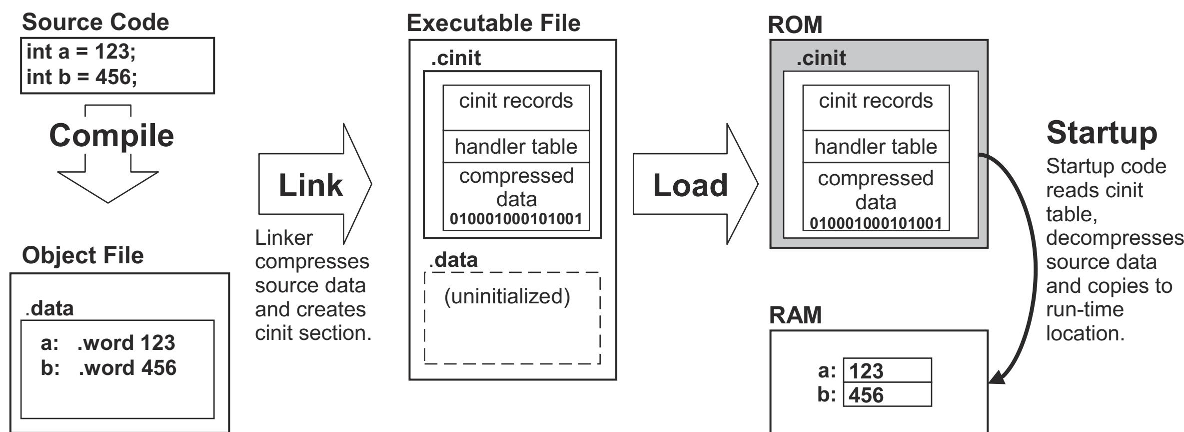

図 14-4. cinit を介した ROM ベースの変数の初期化

コピー テーブルと同様に、.cinit テーブル内のソース データは圧縮されている場合も、されていない場合もあります。圧縮されている場合、そのエンコード方式およびデコード方式はコピー テーブルと同一であり、これによりハンドラ テーブルと解凍ハンドラが共有可能となります。

.cinit セクションには、以下の項目の一部またはすべてが含まれます。

- コピー レコードに類似した cinit レコードから構成される **cinit テーブル**。
- 解凍ルーチンへのポインタで構成される**ハンドラ テーブル** (セクション 14.2 を参照)。ハンドラ テーブルとハンドラは、初期化テーブルとコピー テーブルの両方で共有されます。
- 編集の初期化に使用される、圧縮または解凍されたデータで構成される**ソース データ**。

これらの項目は、任意の順序で配置される場合があります。

図 14-5 は、.cinit セクションの概略図です。

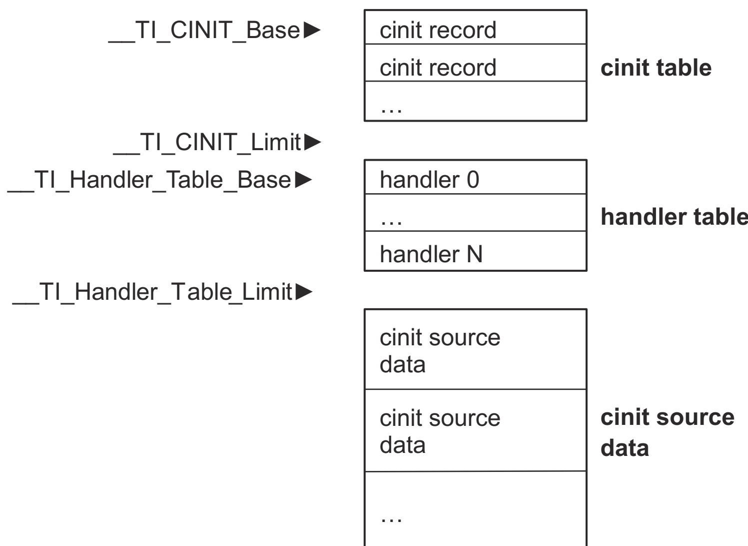

図 14-5. .cinit セクション

.cinit セクションは、その形式を示すセクション タイプ SHT_TI_INITINFO を持ちます。ツールは .cinit という名前ではなく、このセクション タイプに基づいて処理を行う必要があります。

Copyright © 2026 Texas Instruments Incorporated

cinit テーブルを区切るために、次の 2 つの特別なシンボルが定義されています。__TI_CINIT_Base は cinit テーブルを指し、__TI_CINIT_Limit はテーブルの末尾から 16 ビット先を指します。起動時コードはこれらのシンボルを使ってテーブルを参照します。

cinit テーブル内のレコードは次の形式を取ります。

```
typedef struct
{
   uint32  source_data;
   uint32  dest;
} CINIT_RECORD;
```

- **source_data** フィールドは .cinit セクション内のソース データを指します。
- **dest** フィールドは宛先アドレスを指します。コピー テーブルのレコードとは異なり、cinit レコードにはサイズ フィールドがありません。サイズは常にソース データ内にエンコードされます。

ソース データは、圧縮されたコピー テーブルのソース データと同じ形式です (セクション 14.2 を参照)。ハンドラも同様のインターフェイスを持ちます。RLE 形式と LZSS 形式に加えて、cinit レコードには非圧縮形式とゼロ初期化形式の 2 つの追加形式が定義されています。

- **非圧縮**形式が明示的に定義されているのは、コピー テーブルと違って、cinit レコードにはサイズ情報を兼ねるフィールドが存在しないためです。圧縮が使用されていない場合でも、サイズ フィールドは常にソース データ内にエンコードされます。エンコード形式は以下のとおりです。

handler index	padding	size	data
16 bits	16 bits	32 bits	size number of 16-bit data chunks

  エンコードされたデータにはサイズ フィールドが含まれており、これはハンドラ インデックスの次に来る 32 ビット境界に整列されます。サイズ フィールドは、データ ペイロードに含まれる 16 ビット単位のデータ数を指定します。このデータ ペイロードは、サイズ フィールドの直後から始まります。初期化処理では、データ フィールドから指定された *サイズ* の 16 ビット単位のデータ数が、宛先アドレスにコピーされます。TI のランタイム ライブラリには、非圧縮形式に対応した _ _TI_decompress_none というハンドラがあります。
- **ゼロ初期化**形式は、初期値がゼロである変数の一般的なケースに対応するための、コンパクトな形式です。エンコード形式は以下のとおりです。

handler index	padding	size
16 bits	16 bits	32 bits

  サイズ フィールドは、ハンドラ インデックスの次に来る 32 ビット境界に整列されます。初期化処理は、宛先アドレスに連続した *サイズ* 分の 16 ビット単位のデータをゼロで埋めます。TI のランタイム ライブラリには、この形式に対応した _ _TI_zero_init というハンドラがあります。

  最適化として、リンカは隣接する複数のオブジェクトを、同じ形式で効率的にエンコードできる場合には、1 つの cinit レコードにまとめて処理できます。この最適化は、通常、ゼロ初期化されたオブジェクトに対して効果的です。

English Document: SPRAC71
Copyright © 2026 Texas Instruments Incorporated

改訂履歴

Changes from OCTOBER 31, 2023 to MARCH 27, 2026 (from Revision B (October 2023) to Revision C (March 2026)) **Page**

- FPU32 および FPU64 の float 戻り値の位置を修正。 ... 27

Changes from JUNE 3, 2022 to OCTOBER 31, 2023 (from Revision A (June 2022) to Revision B (October 2023)) **Page**

- ELF ヘッダーの e_machine フィールドに関する情報を修正しました。 ... 71

表 15-1 は、このドキュメントの以前のバージョンが公開されてから行われた変更を一覧にしています。

表 15-1. 改訂履歴

	ロケーション	追加/変更/削除
SPRAC71A	セクション 11.5.1	R_C28X_ABS22_BR と R_C28X_ABSLO6_BLKD の再配置タイプを追加しました。

English Document: SPRAC71

Copyright © 2026 Texas Instruments Incorporated

重要なお知らせと免責事項

TI は、技術データと信頼性データ (データシートを含みます)、設計リソース (リファレンス デザインを含みます)、アプリケーションや設計に関する各種アドバイス、Web ツール、安全性情報、その他のリソースを、欠陥が存在する可能性のある「現状のまま」提供しており、商品性および特定目的に対する適合性の黙示保証、第三者の知的財産権の非侵害保証を含むいかなる保証も、明示的または黙示的にかかわらず拒否します。

これらのリソースは、TI 製品を使用する設計の経験を積んだ開発者への提供を意図したものです。(1) お客様のアプリケーションに適した TI 製品の選定、(2) お客様のアプリケーションの設計、検証、試験、(3) お客様のアプリケーションに該当する各種規格や、その他のあらゆる安全性、セキュリティ、規制、または他の要件への確実な適合に関する責任を、お客様のみが単独で負うものとします。

上記の各種リソースは、予告なく変更される可能性があります。これらのリソースは、リソースで説明されている TI 製品を使用するアプリケーションの開発の目的でのみ、TI はその使用をお客様に許諾します。これらのリソースに関して、他の目的で複製することや掲載することは禁止されています。TI や第三者の知的財産権のライセンスが付与されている訳ではありません。お客様は、これらのリソースを自身で使用した結果発生するあらゆる申し立て、損害、費用、損失、責任について、TI およびその代理人を完全に補償するものとし、TI は一切の責任を拒否します。

TI の製品は、 TI の販売条件 、 TI の総合的な品質ガイドライン 、 ti.com または TI 製品などに関連して提供される他の適用条件に従い提供されます。TI がこれらのリソースを提供することは、適用される TI の保証または他の保証の放棄の拡大や変更を意味するものではありません。 TI がカスタム、またはカスタマー仕様として明示的に指定していない限り、TI の製品は標準的なカタログに掲載される汎用機器です。

お客様がいかなる追加条項または代替条項を提案する場合も、TI はそれらに異議を唱え、拒否します。

Copyright © 2026, Texas Instruments Incorporated

最終更新日：2025 年 10 月