C6000 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: JAJA975B
SEPTEMBER 2011 – REVISED AUGUST 2025

目次

図の一覧

表の一覧

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 C6000 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、動的リンクの仕組みなどがあります。

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、C6000 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 C6000 向けの ABI

2009 年の TI の C6000 コンパイラ ツール リリース 7.0 以前に存在した唯一の C6000 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せず、さまざまなシステムが動的リンクの側面を実装していたものの、そのような仕組みに関する標準化やツールのサポートはありませんでした。

TI コンパイラ ツールのリリース 7.0 では、C6000 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おり、動的リンクおよび位置の独立性をサポートしています。*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、動的リンク、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。現在サポートされているプラットフォームは 2 つあります。ベアメタルと Linux 。*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。

Linux ABI は、その可変性を狭め、追加の要件を詳述することによってベアメタル ABI を拡張し、プログラムまたはサブプログラムが C6000 の Linux ベースの OS で実行できるようにします。

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。

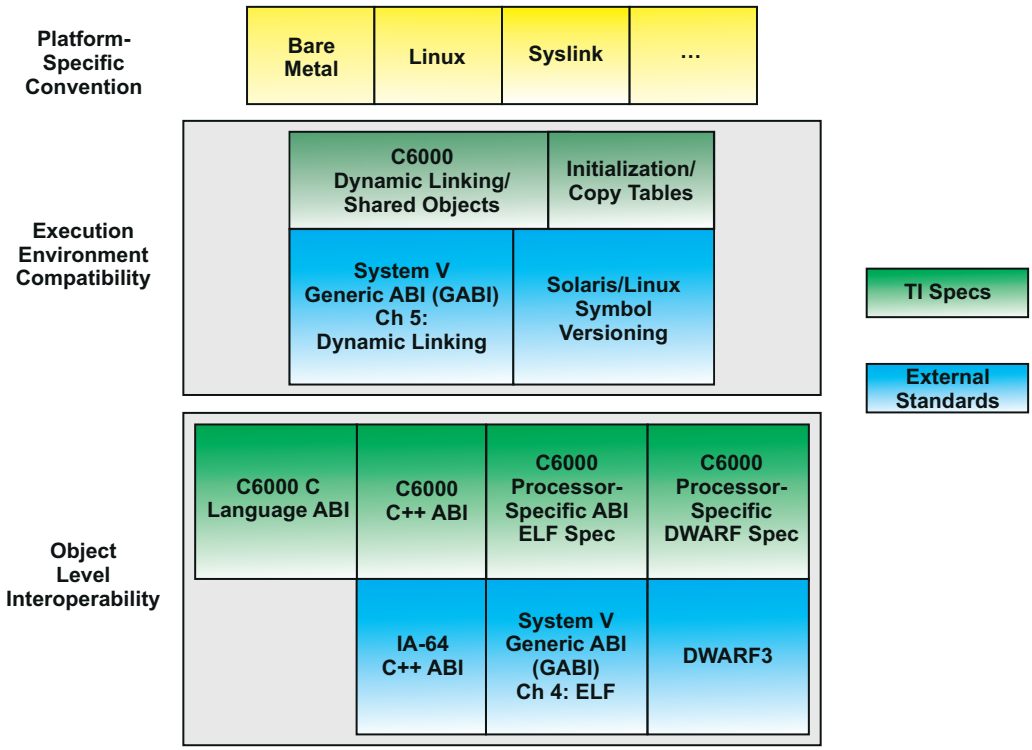


図 1-1. ABI 仕様の一部

C 言語 ABI (Chapter 2、Chapter 3、Chapter 4、Chapter 5、Chapter 8、Chapter 9) 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 10) 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 11) に関する詳細が含まれます。C6000 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 12) 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 13) 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 17) 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。

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。**動的リンク** コンポーネント (Chapter 6 と セクション 14.3) は、個別にリンクされたモジュールが相互運用できるしくみを規定します。これにはコードの共有も含まれます。動的リンクのしくみの一部はデータ アドレッシングの方法であり、これにより個別にリンクされたモジュールが再配置なしで互いのデータをアドレッシングできるようにします。

スレッドローカル ストレージ (Chapter 7) は、静的記憶域期間を持つスレッド固有の変数の作成を可能にします。スレッドローカル変数の仕様、表現、アクセスについてはこのドキュメントで解説されています。

シンボル バージョニング (Chapter 16) は、シンボリック参照が最小バージョンを含むようにし、そのバージョン以上を持つ定義と動的に解決されるようにすることによって、実行時の非互換性を防ぐしくみです。この ABI は、変更なしで標準の GCC/Linux モデルを採用しています。

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。ABI によって現在対象とされている他の唯一の環境は、Linux プラットフォーム (Chapter 15) です。

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 18) 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。

プログラム ヘッダー属性 (Chapter 19) は、基本 ELF 標準で規定されている内容を超えて、ELF セグメントのさまざまな追加プロパティを表すために、TI ツールチェーンによって実装された ELF の拡張です。TI ツールはこれらを使用して、メモリの接続性 / レイテンシ要件、保護、キャッシュ動作、その他のシステム固有のプロパティをエンコードします。柔軟性と拡張性を備えた設計になっています。ここでも、他のツールが必要に応じて相互運用できるよう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。たとえば、ABI はすべてのケースで PIC (位置独立コード) アドレッシングを使用することを規定していませんが、PIC が使用されるケースの実装方法を標準化しています。バリアントの中には互換性のないものもあります。たとえば、DBST PIC モデルを使用するオブジェクトがある場合、他のすべてのオブジェクトもこれ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。

このセクションでは、一般的なユース ケースと、それらが ABI とどのように関連しているかについて説明します。これらのケースは相互に排他的ではなく、すべての可能性を完全にカバーしているわけではありません。

- **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、通常、位置独立性を必要としません。自己完結型であるため、動的リンク情報、プロシージャ リンケージ テーブル (PLT) スタブ、グローバル オフセット テーブル (GOT) を含める必要はありません。
- **ベア メタル - 動的リンク**。このモデルは、実行可能ファイルが個別にリンクされたモジュールに動的にリンクできるシステムを指しますが、OS の制御された環境内ではありません。アドレッシングは、環境に応じて位置独立である場合とそうでない場合があります。環境によっては、アドレッシングや配置に追加の規則が課されることがあります。このモデルでは、図 1-1 の動的リンクのコンポーネントを使用します。ベアメタル動的リンクモデルの詳細については、セクション 14.4 を参照してください。
- **共有オブジェクト**。これは、静的にリンクされたモジュール (ライブラリ) を複数の個別にリンクされたクライアント (実行可能ファイルや他のライブラリ) 間で共有できる動的リンク モデルを指します。根本的な問題は、各クライアントがライブラリのデータの独自のコピーを持つ必要があることです。ABI は、位置独立アドレッシングとデータ セグメント ベース テーブル (DSBT) メカニズムという 2 つの関連構造によってこれを解決します。
- **位置独立性**。これはアドレス定数を使用しないアドレッシング方法を指し、コードやデータを再配置することなく任意のアドレスにロードして実行できるようにします。*PIC* という用語は、一般に位置独立コードを意味しますが、位置独立性はコード、データ、またはその両方を指す場合があります。共有ライブラリでは、複数のクライアントに個別のコピーを持たせるために、位置独立データが必要です。共有ライブラリのコンテキストでは、PIC という用語はこの狭義の定義を

指すことがあります。ROM 内のライブラリは、ROM 作成時にアドレスがバインドされていない場合、他のオブジェクトを参照するために位置独立アドレッシングが必要になることがあります。位置独立データは、データ ページ レジスタ (B14) に依存します。動的リンクのように複数のモジュールが関与している場合、*DSBT* (データ セグメント ベース テーブル) モデルは、あるモジュールから別のモジュールへの呼び出し時に DP をリセットするために使用できるメカニズムです。

- **Linux**。Linux 環境用に構築された実行可能ファイルと共有ライブラリは、特定の規則に従う必要があります。これらは動的リンク情報を持っています。また、DSBT モデルを使用した位置独立性が必要です。Linux 用に構築されたオブジェクトは、ELF ヘッダの EI_OSABI フィールドに ELFOSABI_C6000_LINUX フラグを持っています。Linux プラットフォーム向けの ABI の拡張については、Chapter 15 で詳しく説明します。
- **ROM 化**。ROM に常駐する個別にリンクされたモジュールをビルドすることが望ましい場合があります。一度リンクされると、そのアドレスは永久にバインドされます。その後、他のモジュールに対して静的または動的にリンクできます。この目的のために、ABI は、静的リンクビューと動的リンクビューの両方を示す特別な ELF ファイルのクラスと、アドレスが永続的にバインドされているセクションを示すいくつかのセクション フラグを定義します。ROM モジュールは通常、参照する他のモジュールの配置に依存しないように、PIC アドレッシングを使用します。

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 C6000 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。

C6000 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。

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または共有オブジェクト 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。動的リンク情報を含む場合と含まない場合があります。
- **共有オブジェクト ファイル**は、実行可能ファイルおよび他の共有オブジェクトとロード時に結合してプロセス イメージを形成できるプログラムの構成部分です。共有オブジェクトは常に動的リンク情報を含みます。再配置可能オブジェクト ファイルとの混同を避けるために、共有オブジェクトを *共有ライブラリ* と呼ぶこともあります。
- **再配置可能モジュール**。再配置可能モジュールは、静的リンク情報 (静的シンボル テーブル、セクション テーブル、静的再配置エントリ) も含む共有オブジェクト ファイルです。これは、静的にも動的にもリンク可能な ROM 可能ライブラリを意図しています。

この仕様では、実行可能ファイルおよび共有ライブラリ (再配置可能モジュールを含む) 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 (実行可能ファイルまたは共有オブジェクト) 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、またはロード時に動的に行われます。

1.8 C6000 アーキテクチャの概要

一般的に C6000 または C6x として知られている TMS320C6000 は、テキサス インスツルメンツの 32 ビット VLIW デジタル信号プロセッサのファミリです。このファミリには、固定小数点 (整数) と浮動小数点の両方のデバイスが含まれています。このアーキテクチャは、高度な並列処理を実現するために、1 サイクルあたり最大 8 つの 32 ビットの命令を発行できます。表 1-1 は、この ABI が対応する C6000 ファミリのメンバーの一覧です。

表 1-1. C6000 ISA

ISA	データ形式	説明
C62x	固定小数点	正規 ISA 製品
C64x	固定小数点	追加の命令とレジスタが加わった C62x
C64x+	固定小数点	追加の命令とコンパクトな命令エンコーディング
C67x	浮動小数点	正規の浮動小数点 ISA
C67x+	浮動小数点	C67x と追加の命令とレジスタ
C6740	固定/浮動小数点	C64x+ および C67x+ の共用体と追加の命令
C6600	固定/浮動小数点	128 ビットのデータ パスと追加の命令が加わった C6740

ほとんどのファミリ メンバーには後方互換性があります。つまり、新しい CPU は古いデバイス用に構築されたオブジェクト コードを正しく実行できます。特定のケースは セクション 17.2 の Tag_ISA ビルド属性で指定します。

C6000 デバイスはバイト単位でアドレス指定できます。メモリはビッグ エンディアンまたはリトル エンディアンとして構成できます。ほとんどのデバイスには一般的なメモリ管理ユニットが搭載されていないため、CPU アドレスは実際の物理メモリ位置を参照します (仮想メモリの参照なし)。

C6000 のパイプラインは保護されていません。つまり、まだパイプライン内にあり、まだ書き込まれていない、以前に発行された計算の送信先を CPU が読み取る場合、その読み取りは新しい計算を待つために停止するのではなく、古い値を取得します。これは、正しい結果を得るには、プログラマ (またはコンパイラ) がパイプライン遅延を管理し、演算をスケジュールしなければならないことを意味します。複数サイクルの遅延が発生する操作には、負荷 (4 サイクル)、分岐 (5 サイクル)、特定の乗算 (2 サイクル) が含まれます。

C6000 には、A0～A15 および B0～B15 に指定された、少なくとも 32 の汎用レジスタがあります。C64 ファミリのメンバーでは、これが 64 のレジスタに拡張されており、A0～A31 および B0～B31 に指定されます。これらのレジスタのうち 2 つは、ABI の下でアドレッシングおよびリンケージに使用するために、慣例により割り当てられています。B15 はスタック ポインタ (通常は **SP** と表記) として指定され、B14 はデータ ページ ポインタ (**DP** と表記) として指定されます。DP はデータ セグメントのベース アドレスとして使用され、位置の独立性と (ニア) データへの効率的なアクセスの両方を可能にする手段を提供します。

1.9 参考資料

ドキュメント タイトル	リンクまたは URL
TMS320C64x/C64x+ DSP CPU および命令セット リファレンス ガイド	SPRU732
TMS320C6000 最適化コンパイラ ユーザー ガイド	SPRU187
TMS320C6000 アセンブリ言語ツール	SPRU186
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

1.10 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。

単純化のため、コード シーケンスはスケジュールされていません。つまり、各命令は次の命令の実行を開始する前に完了していると見なされます。

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

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	char	8	8
符号なし文字	uchar	8	8
char	修飾のない文字	8	8
ブール (C99)	uchar	8	8
_Bool (C99)	uchar	8	8
ブール (C++)	uchar	8	8
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	int32	32	32
符号なし整数	uint32	32	32
長整数 (32 ビット)、符号付き長整数	int32	32	32
符号なし長整数	uint32	32	32
長整数 (40 ビット)	int40	40	64
長々整数、符号付き長々整数	int64	64	64
符号なし長々整数	uint64	64	64
列挙	--	変化 (「セクション 2.8」を参照)	32
フローティング	float32	32	32
精度浮動小数点	float64	64	64
拡張精度浮動小数点	float64	64	64
ポインタ	--	32	32

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。

記載されている整列値はデフォルト値です。型が「パックされた」構造内で使用される場合を除き、すべての場合に適用されます。「パックされた」構造内のすべての構造メンバー型は、8 ビットに整列されます。

文字 型は符号付きです。

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。

C6000 向けの旧 COFF ABI では、C の *長整数* 型は 40 ビット整数であり、元の 62x ハードウェアの最長のネイティブ整数型に対応していました。デフォルトでは、EABI は長整数を 32 ビットに変更し、広く共有されてきた暗黙の慣習に合わせています。しかし、ツールチェーンは、COFF ABI 下で開発されたレガシー コードとの互換性のために、長整数あるいはその他の名称で指定されたオプションの 40 ビット型のサポートが必要な場合があり、その表現がここで説明されています。この型は 40 ビットのサイズを持ちますが、整列は 64 ビットです。40 ビット *長整数* 型が「パックされた」構造のメンバーとして使用される場合、その型は 8 ビットの整列を持ちますが、64 ビットのコンテナ サイズを持ちます。

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 int     size_t;
typedef int              ptrdiff_t;
typedef unsigned int     wchar_t;
```

```
        typedef unsigned int     wint_t;
        typedef char *           va_list;
```

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。

サイズが 32 ビット以下のオブジェクトは 1 つのレジスタに収めることができます。

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。32 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。32 ビット より小さい符号なし値は、ゼロ拡張されます。

サイズが 32 ビットから 64 ビットのオブジェクトは、レジスタ ペアを使用します。レジスタ ペアは偶数番号のレジスタとその直後の奇数番号のレジスタで構成されます。偶数番号のレジスタは値の下位部分を保持し、奇数番号のレジスタは値の上位部分を保持します。レジスタ ペアは R_o:R_e として表記されます。ここで R_o は奇数レジスタ、R_e は偶数レジスタです (例 : A5:A4)。レジスタ ペアの数値は偶数レジスタに右詰めされます。すなわち、偶数レジスタのビット 0 が値の最下位ビットを保持し、奇数レジスタのビット 0 が値の 32 ビット目を保持します。符号付き整数値は奇数レジスタの上位ビットに符号拡張されます。符号なし値はゼロ拡張されます。

64 ビットを超えるオブジェクトには、指定されたレジスタ表現は存在しません。

2.3 メモリ内のデータ

C6000 はビッグ エンディアン モードまたはリトル エンディアン モードのいずれでも動作するように構成できます。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。ビッグ エンディアン モードでは、値の最上位バイトが最も小さいアドレスに格納されます。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 : バイト用の LDB/STB、ハーフワード用の LDH/STH、ワード用の LDW/STW、その他。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。

40 ビット整数は 5 バイトを持ち、0 (LSB) から 4 (MSB) まで番号が付けられます。メモリ内では、40 ビット値は 64 ビット (8 バイト) にパッドされます。メモリ内の値のアドレスが N である場合、その格納レイアウトは図 2-1 のようになります。

location	little-endian	big-endian
N	byte 0 (lsb)	padding
N+1	byte 1	
N+2	byte 2	
N+3	byte 3	byte 4 (msb)
N+4	byte 4 (msb)	byte 3
N+5	padding	byte 2
N+6		byte 1
N+7		byte 0 (lsb)

図 2-1. メモリ内の 40 ビット値の表現

2.4 複素数型

C99 標準は、その _Complex 型のレイアウトと整列を、対応する浮動小数点型の 2 要素配列と同等とし、最初の要素を実数部、2 番目の要素を虚数部とするよう規定しています。これにより、ABI には柔軟性がほとんどなくなります。したがって、複素数型に対する C6000 の表現は表 2-2 のようになります。

表 2-2. 複素数型

タイプ	汎用的な名前	サイズ	整列	外部整列
浮動小数点 _Complex	complex32	64	32	64
倍精度浮動小数点 _Complex	complex64	128	64	128
拡張倍精度浮動小数点 _Complex	complex64	128	64	128

複素数型、または外部可視性を持つ複素数配列型の変数は、その型で要求されるものよりも厳格な整列要件を持ちます。表の「外部整列」列は、そのような変数に対する最小整列を示します。

2.5 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)。

サイズが 64 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。

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。

ビッグ エンディアン モードでは、以下の規則に従って構造がレジスタ内に配置されます。

- 1 バイトの構造は、1 つのレジスタの LSB に格納されます。
- 2 バイトの構造では、1 バイト目がレジスタのバイト 1 に、2 バイト目がバイト 0 (LSB) に格納されます
- 3 バイトまたは 4 バイトの構造では、1 バイト目がレジスタのバイト 3 (MSB) に格納され、残りのバイトが LSB に向かって順に格納されます。3 バイト構造の場合、レジスタの LSB に 1 バイトのパディングが挿入されます。
- 5 ～ 8 バイトの構造では、1 バイト目が上位 (奇数) レジスタのバイト 3 (MSB) に格納され、残りのバイトが上位バイトから下位バイトに向かって順に格納されます。5～7 バイトの構造では、下位 (偶数) レジスタの LSB にパディングが含まれます。

図 2-2 は、1 バイトから 8 バイトまでのサイズの構造における、ビッグ エンディアン モードでのレジスタ内表現を示しています。

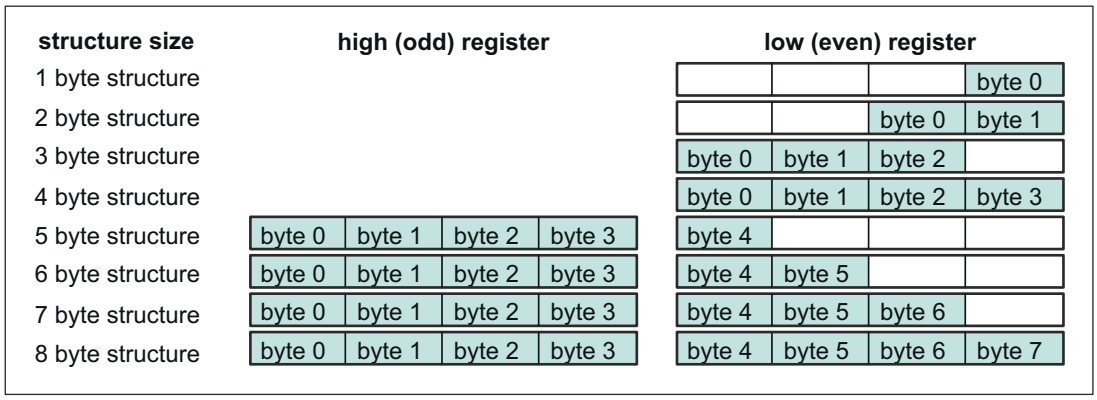


図 2-2. レジスタ内の構造または共用体のビッグ エンディアン レイアウト

このようなレイアウトが採用されている理由は、構造をレジスタとメモリの間で、可能な限り小さな適切なメモリ参照単位を使ってコピーできるようにするためです。たとえば、2 バイトの構造では 16 ビット参照、3 バイト構造では 32 ビット参照、5 バイト構造では 64 ビット参照が使われます。構造は、含まれるメモリ参照サイズ内に左詰めで配置されます。

2.6 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。

外部可視性を持つファイルスコープの配列変数には、より厳しい要件があります。ターゲットの ISA に応じて、このような変数の整列は、その要素の整列と次の値の最大値となります。

C62x、C67x	4 バイト
その他すべて	8 バイト

2.7 ビット フィールド

C6000 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。

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。

C6000 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。

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。ビッグ エンディアン モードでは、コンテナは MSB から LSB へと埋められます。

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、C6000 EABI によって *符号付き整数* として扱われます。

2.7.1 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。

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。

たとえば、次の構造を考えます。

```
struct S
{
   volatile int a:8;       // container is 32 bits at offset 0
   volatile char b:2       // container is 8 bits at offset 8
};
```

「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。

2.8 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。ただし、どちらもすべての列挙子を表すことができない場合、ベースとなる型は long long か 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。

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

C6000 では、ISA のバリアントや呼び出しの状況に応じて、関数呼び出しを実現するためのさまざまな命令が用意されています。共通して、関数呼び出しは復帰アドレスをレジスタ B3 に保存し、呼び出し先関数へ分岐することで実行されます。呼び出し先関数は、呼び出し時に B3 に格納されていたアドレスへの間接分岐を行うことで処理を戻します。

3.1.1 復帰アドレスの算出

64x ターゲットでは、呼び出しは 2 つの命令のシーケンスになります。PC 相対アドレッシングを使用して ADDKPC 命令が復帰アドレスを B3 に計算し、その後に B (分岐) 命令が続きます。

```
          ADDKPC return_label,B3     ; B3 := return_label
          B      func                ; goto func
    return_label:
```

64x 以外のターゲットでは、絶対アドレッシング、PC 相対アドレッシング、GOT ベース アドレッシングなど、他の方法を使用してアドレスを計算できます (セクション 5.1 に記載)。

3.1.2 呼び出し命令

呼び出し自体は、呼び出し先に制御を移す単純な PC 相対分岐として生成されます。

```
          B      func                ; goto func
```

このディスプレイスメントは 21 ビットの符号付きワード オフセットです。宛先が到達範囲外である場合、リンカはトランポリンと呼ばれるスタブ関数を生成します。これは、絶対、PC 相対、GOT 間接アドレッシングを使用して宛先関数のアドレスを指定します。トランポリンの詳細については、セクション 5.3 をご参照ください。

間接呼び出しの場合、宛先はレジスタです。

```
          B      reg                 ; goto address in reg
```

呼び出しを実装する分岐の場合、TI ツールチェーンは **CALL** 疑似命令を使用します。この命令は分岐としてエンコードされますが、デバッグ情報に注釈を付けて、プロファイラやデバッガなどの解析ツールが命令を関数呼び出しとして認識できるようにします (セクション 5.3 を参照)。したがって、先ほどの直接呼び出しは、実際のアセンブリ ソースでは次のように記述されます。

```
          CALL   func                ; encodes as B func
```

C64+ ISA には CALLP という複合命令があり、それ単独で次のように呼び出しを実装します。

- 次の実行パケットのアドレスを復帰アドレスとして B3 にロードする
- 分岐命令のディレイ 分岐のディレイスロットを埋めるために NOP 5 を行う

CALLP を使用した呼び出しは単純に以下のようになります。

```
          CALLP func,    B3
```

3.1.3 復帰命令

関数の復帰は、B3 に渡されたアドレスに分岐することによって実行されます。呼び出し先はアドレスを移動して他の場所に格納する場合がありますが、通常はネストされた呼び出しで必要になります。アドレスがまだ B3 にある場合、命令は次のようになります。

```
          B      B3                  ; return
```

関数が割り込みハンドラ関数である場合、これは代わりに IRP への分岐である必要があります。

```
        B      IRP                 ; return from interrupt handler
```

TI ツールチェーンは、関数の復帰を実装する分岐を指定するために **RET** 疑似命令を使用します。

3.1.4 パイプライン規約

C6000 では、分岐命令 (call や return を含む) のフェッチから、その命令が実行されるサイクルまでの間に 5 つのディレイ スロットがあります。これらのディレイ スロットは、次の条件に従って命令をスケジュールできます。呼び出し元は、呼び出し先に影響を与える可能性のあるすべての命令の効果が、呼び出し先の最初の命令の E1 フェーズの前に完了することを確認する責任があります。同様に、return 命令の場合は、呼び出し先は呼び出し元に影響を与える可能性のあるすべての命令が、復帰アドレスにある命令の E1 フェーズの前に完了することを確認する責任があります。

3.1.5 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。

リンカは未解決の弱関数呼び出しを、静的リンクが使用されているか動的リンクが使用されているかによって異なる方法で処理します。

静的リンクが使用されている場合、リンカは未定義の弱関数呼び出しを、実質的に NOP となるものに置き換えます。ただし、呼び出し先が呼び出し元サイトに戻らない最適化 (tail call elimination) を可能にするためには、置き換えでは呼び出し動作の一部を保持する必要があります。したがって、その置換は NOP ではなく、代理復帰命令となります。

```
        B.S2   B3                  ; replacement for unresolved weak call
```

この動作により、弱関数を呼び出す際には次の追加要件があります。

- S2 機能ユニットが利用可能である必要があります。元の呼び出しが S2 にエンコードされている場合は、この条件は自動的に満たされます。
- 呼び出し命令がパイプラインの E1 フェーズに到達する時点で、B3 に復帰アドレスが保持されている必要があります。言い換えると、コンパイラは呼び出しの遅延スロットに復帰アドレスの算出をスケジュールすることはできません。

  ABI はインポートされた弱関数 (すなわち、異なる静的リンク ユニットで定義される可能性がある関数) の呼び出しをサポートします。

動的リンクが使用される場合、グローバル オフセット テーブル (GOT) が使用され、未解決の弱関数を 0 の値のままにするか、または単に呼び出し元に戻る _ _c6xabi_weak_return() のアドレスにパッチするかが決定されます。0 の値が使用される場合 (Linux プラットフォーム向け)、ガードが必要です。_ _c6xabi_weak_return() 関数が使用される場合 (ベアメタル プラットフォーム向け)、ガードは不要です。

3.2 レジスタ規約

C6000 には、32 個以上の汎用 32 ビット レジスタがあります。レジスタには、整数、浮動小数点値、またはポインタを格納できます。汎用レジスタは、A と B で指定される 2 つのレジスタ ファイルに分割されています

B15 はスタック ポインタ (SP) と呼ばれます。スタック ポインタは 2 ワード (8 バイト) 境界で整列したままにする必要があります。SP は、現在割り当てられているスタックよりも下の、最初の整列アドレスを指します (セクション 4.3 を参照)。

B14 はデータ ページ ポインタ (DP) と呼ばれます。これは、現在アクティブなオブジェクトのデータ セグメントの先頭を指します。

ABI は専用フレーム ポインタ (FP) レジスタを指定しません。ただし、TI コンパイラは状況によっては A15 をフレーム ポインタとして使用します。

GCC は言語拡張として字句的にネストされた関数をサポートしています。この実装では、静的チェーン レジスタと呼ばれるレジスタを使用して、親関数のアクティベーション コンテキストを子に提供します。レジスタの選択は、呼び出しが何らか

の方法で (たとえばトランポリンによって) 仲介されない限り、大部分はツールチェーン固有です。このため、ABI は静的チェーン レジスタの推奨選択肢として A2 を指定しています。呼び出し規約では、この指定をサポートするために A2 を関数リンクに関係する 1 つのレジスタに含め、呼び出しサイトと呼び出し先のエントリ ポイントの間でその値が保持されることを要求しています。

ABI では、A10 ～ A15 および B10 ～ B15 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。このセットには SP (B15) と DP (B14) が含まれます。

さらに、ILC と RILC は呼び出し先保存レジスタです。これらは C64+ の SPLOOP 機構で使用される制御レジスタです。

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。

アドレス モード レジスタ (AMR) は、循環アドレッシングを有効にするユーザー書き込み可能な制御レジスタです。関数呼び出しの境界では、循環アドレッシングが無効になるように、AMR の境界ビット 0 ～ 15 を 0 にする必要があります。

表 3-1 に、ABI でのレジスタとその役割を示します。

表 3-1. C6000 のレジスタ規約

登録	別名	呼び出し先保存	呼び出し規約での役割
A0		なし	
A1		なし	
A2		なし	ネストされた関数の静的チェーン レジスタ
A3		なし	参照によって返される構造体のアドレス
A4		なし	第 1 引数、戻り値 (LSW)
A5		なし	第 1 引数、戻り値 (MSW)
A6		なし	第 3 引数 (LSW)
A7		なし	第 3 引数 (MSW)
A8		なし	第 5 引数 (LSW)
A9		なし	第 5 引数 (MSW)
A10		あり	第 7 引数 (LSW)
A11		あり	第 7 引数 (MSW)
A12		あり	第 9 引数 (LSW)
A13		あり	第 9 引数 (MSW)
A14		あり	
A15	FP	あり	フレーム ポインタ
A16-A31		なし	
B0		なし	遅延バインダへの動的再配置オフセット引数 (セクション 15.6 を参照)
B1		なし	遅延バインダへの動的再配置オフセット引数 (セクション 15.6 を参照)
B2		なし	
B3		なし	復帰アドレス
B4		なし	第 2 引数 (LSW)
B5		なし	第 2 引数 (MSW)
B6		なし	第 4 引数 (LSW)
B7		なし	第 4 引数 (MSW)
B8		なし	第 6 引数 (LSW)
B9		なし	第 6 引数 (MSW)
B10		あり	第 8 引数 (LSW)
B11		あり	第 8 引数 (MSW)

表 3-1. C6000 のレジスタ規約 (続き)

登録	別名	呼び出し先保存	呼び出し規約での役割
B12		あり	第 10 引数 (LSW)
B13		あり	第 10 引数 (MSW)
B14	DP	あり	データ ページ ポインタ
B15	SP	あり	スタック ポインタ
B16-B29		なし	
B30-B31		いいえ	トランポリン スクラッチ レジスタ (セクション 3.7 を参照)

3.3 引数の受け渡し

関数への最初の 10 個の引数は、レジスタで渡されます。引数は、宣言された順番で (ただしレジスタ クワッドで渡される引数を除いて)、以下の順序でレジスタに割り当てられます。

A4、B4、A6、B6、A8、B8、A10、B10、A12、B12

サイズが 32 ビットから 64 ビットの引数はレジスタ ペアで渡され、下位部分には前のリストの偶数レジスタが、上位部分には対応する奇数レジスタが使用されます。たとえば、次の例では「a」は A4 に渡され、「b」は B5:B4 に渡されます。

```
func1(int a, double b);
```

浮動複素数型の引数はレジスタ ペアで渡されます。順序はエンディアンによって異なります。リトル エンディアン モードでは、実数部は偶数レジスタに渡され、虚数部は奇数レジスタに渡されます。ビッグ エンディアン モードでは、この順序は逆になります。

倍精度複素数型の引数は、次のリストの最初に使用可能なクワッドを使用して、レジスタ クワッドに渡されます: A7:A6:A5:A4、B7:B6:B5:B4、A11:A10:A9:A8、B11:B10:B9:B8。リトル エンディアン モードでは、実数部は番号の小さいペア (A5:A4 など) に渡され、虚数部は番号の大きいペア (A7:A6) に渡されます。ビッグ エンディアンでは、この順序は逆になります。クアッド レジスタ引数のためにバイパスされたレジスタは、その後の引数で利用可能です。たとえば、次の関数では「w」は A4 に渡され、「x」は B4 に渡され、「y」は A11:A10:A9:A8 に渡され、「z」は A6 にバックフィルされます。

```
func2(int w, int x, double complex y, int z);
```

残りの引数は、SP+4 を開始点として、アドレスを増加させながらスタック上に配置されます。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ます。したがって、最初のスタック引数が 64 ビット整列を必要とする場合、そのアドレスは SP+8 になります。

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 A4 に渡されます。

サイズが 64 ビット以下の構造および共用体は値渡しされ、レジスタまたはスタック上で、次に示すリストに従って渡されます。64 ビットを超える構造および共用体は参照渡しされます (「セクション 3.5」を参照)。

レジスタに渡されない引数は、SP+4 を開始点として、アドレスを増加させながらスタック上に配置されます。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(値渡しされる構造体に許容される最大サイズである 8 バイトを超えることはできません)。
- 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。

SP+4 は 8 バイトに整列されません。したがって、最初の引数が 8 バイト整列を必要とする場合、それは SP+8 のメモリに格納されます。

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。

3.4 戻り値

サイズが 32 ビット以下のスカラや構造体は A4 で返されます。32 ビットから 64 ビットまでのスカラや構造体は A5:A4 で返されます。

float complex 型のオブジェクトは A5:A4 で返され、実数部は奇数レジスタに、虚数部は偶数レジスタに返されます。

double complex 型のオブジェクトは、実数部は A5:A4、虚数部は A7:A6 で返されます。

64 ビットを超える集成体は、参照によって返されます。

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 64 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。

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、呼び出し先は参照されているオブジェクトを変更できず、自身でコピーを作成する必要があります。

呼び出された関数が、32 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 A3 に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。

呼び出し先はオブジェクト A3 に格納されたアドレスにコピーして返します (ゼロでない場合)。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、A3 に &s を渡すことでコンパイルできます。

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が、性能を向上させるために、いくつかの関数については例外が設けられています。変更された規則を使用するヘルパー関数については、セクション 8.3 を参照してください。

3.7 セクション間呼び出しのスクラッチ 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、またはヘルパー関数を呼び出す際に特別な規則が存在する場合 (セクション 8.3 を参照) に生じます。

ただし、レジスタ B30 と B31 は、定義が既知の場合やヘルパー関数を呼び出す場合であっても、セクション境界をまたぐ呼び出しによって変更される可能性があるものとして指定されています。これは、その呼び出しがファー呼び出しトランポリン (セクション 5.3) を必要とする場合に、B30 および B31 がトランポリンにおけるスクラッチ レジスタとして使用できるようにするためです。

さらに、Linux の遅延バインディング機構では、遅延バインディングを実装するスタブ関数で呼び出し元保存レジスタを利用できるようにしておく必要があります。呼び出し先がインポートされ、従って遅延バインディングを介して呼び出される可能性がある場合、コンパイラは呼び出しサイトを最適化してはなりません。

同じセクション内の呼び出しでは、トランポリンは必要ありません。そのようなセクション内呼び出しでは、B30 および B31 は他の呼び出し元保存レジスタと何ら変わらずに扱われます。

3.8 DP の設定

共有オブジェクト用に DSBT モデルでコンパイルされたエクスポート関数は、エントリ時に DP を設定し、終了時にそれを復元する必要が生じることがあります。これについては、セクション 6.7.1 で説明さいます。

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

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。

ABI は静的データ セクションを *ニア* または *ファー* として指定します。ニア セクションは効率的なニア DP 相対アドレッシングを用いて参照できますが、そのサイズや配置には制約があります。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 4.2.2 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 (実行可能ファイルまたは共有ライブラリ) 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。

- すべてのニア DP 相対アドレッシングでアクセスされるセクションは、_ _C6000_DSBT_BASE によって定義される静的ベース アドレスからの符号なし 15 ビット アドレッシング範囲内に収まるように、グループ化されなければなりません。
- 指定されたセグメント内のすべてのデータは、同一のセグメント属性に従います (Chapter 19 を参照)
- 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。

セグメントは、DP 相対アドレッシングを使用してアクセスされる場合に DP 相対として指定されます。1 つの DP 相対セグメントは、前述の制約を満たす限り、ニアとファー アドレッシングを混在させることができます。

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。

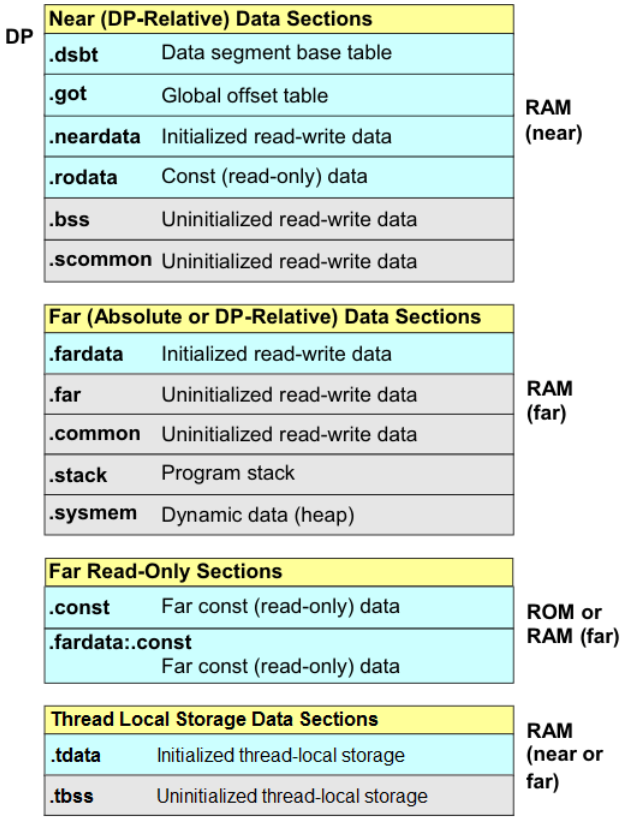


図 4-1. データ セクションとセグメント (標準)

.const セクションおよび .fardata:.const セクションには、読み取り専用の定数が含まれています。.const セクションには位置独立の定数が含まれます。プラットフォームによっては、.const セクションは読み取り専用メモリ内に配置され、絶対アドレッシングで参照される場合があります。また、位置独立モデルでは、コードに対して PC 相対アドレッシングを通じて参照される場合もあります。Linux のように読み取り専用セグメントを共有するプラットフォームでは、初期化子にアドレス定数を含む const オブジェクトは共有できません。したがって、それらは .fardata の一部とみなすことができるため、データ セグメント内で .fardata:.const と呼ばれる個別のセクションに配置されます。

.rodata セクションには、ニア DP 相対アドレッシングで参照可能な読み取り専用定数が含まれます。

.neardata セクションおよび .fardata セクションには、初期化済みの読み書き可能変数が含まれます。これらのセクションは、他のアーキテクチャで一般的に見られる .data セクションに対応します。

.bss セクションおよび .far セクションには、未初期化変数が含まれます。

.common セクションおよび .scommon セクションには、リンカによって割り当てられる共通ブロックシンボルが含まれます。これらはオブジェクト ファイル内の実際のセクションではありません。セクション名は、変数を配置するためにリンカ コマンド ファイルで用いられる慣習にすぎません。これらのセクションは、他の目的には使用しないでください。

.got セクションおよび .dsbt セクションには、動的リンクに関連するデータ構造が含まれます。Chapter 6 を参照してください。

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 13.3.5 に一覧として示されています。

4.2 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 (静的または動的) 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。

*DP 相対セグメント*として指定されたデータ セグメントは、DP 相対アドレッシングを使用して参照されます。ロード モジュール内の任意のコードに入る際、DP はロード モジュールの DP 相対セグメントのうち最も低いアドレスを持つプロセス専用コピーを指すように初期化されます。リンカはこのアドレスを指すシンボル _ _C6000_DSBT_BASE を定義します。

DP 相対アドレッシングには 2 つの形式があります。ニア DP 相対アドレッシングは、DP 相対オフセットが 15 ビットの符号なし定数として 1 つの命令に符号化できる場合に適用されます。ファー DP 相対アドレッシングは、それができない場合に適用され、追加の命令が必要になります。変数がニア形式を用いてアドレッシングされる場合、その配置は DP から 32KB 以内に制約されます。

DP 相対セグメントは、プログラム ヘッダー内の PF_C6000_DPREL フラグによって識別されます (「セクション 14.1」を参照)。

一部のプラットフォーム (特に Linux) では、ロード モジュールが 1 つを超える DP 相対セグメントを持たないように制約される場合があります。

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または GOT ベースのいずれか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。

プログラムが動的にリンクされ、共有ライブラリを持つ場合、各ロード モジュールのデータ セグメントは他のロード モジュールのデータ セグメントとは独立しています。特に、各ロード モジュールは DP 相対セグメントを含む独自のデータ セグメントを持ち、したがって独自の DP を持ちます。複数の実行可能ファイルが 1 つのライブラリを共有する場合、それぞれがそのライブラリのデータ セグメントの専用コピーを取得します。仮想アドレス変換が存在しない場合に複数のデータ セグメントを管理するモデルは DSBT モデルと呼ばれます。詳細は「セクション 6.7」を参照してください。

4.2.1 静的データのアドレッシング方法

ABI は静的データをアドレッシングするために、DP 相対、絶対、GOT 間接、PC 相対の基本的な方式をサポートしています。どの方式が使用されるかは、変数の宣言、実行プラットフォーム、モジュールが実行可能ファイルとして構築されるか共有ライブラリとして構築されるか、可視性規則など、さまざまな要因に依存します。コンパイラがアドレッシングを生成するため、そのコンテキストを認識していなければなりません。通常は、コマンドライン オプションまたはソース コード内の可視性ディレクティブを介して行われます。この ABI の他のセクションでは、各アドレッシング方式が *いつ* 適用されるかの詳細を提供しています。このセクションでは、アドレッシングが *どのように* 実行されるかを規定します。

4.2.1.1 ニア DP 相対アドレッシング

これは、ニア DP セグメントの静的変数に対するデフォルトのアドレッシングです。DP オフセットは 15 ビットの符号なしの値であり、この形式のアドレッシングは DP の 32KB 以内のオブジェクトに制限されます。

```
        LDW     *+DP(sym),dest              ;reloc R_C6000_SBR_U15_W (also _B and _H)
```

4.2.1.2 ファー DP 相対

これは *ファー データ*、すなわちニア DP セグメント以外のセグメント内のデータをアドレッシングする位置独立の方法です。32 ビットの DP 相対オフセットは、2 つの MVK 命令を使用してレジスタにロードされ、その後インデックス付きアドレッシングを使用して DP に追加されます。オフセットはアクセスのサイズに応じて適切にスケーリングする必要があります。TI ツールチェーンは、特殊なアセンブリ言語演算子を使用して、倍率を示します。

```
        MVKL    $DPR_word(sym),tmp          ;reloc R_C6000_SBR_L16_W
        MVKH    $DPR_word(sym),tmp          ;reloc R_C6000_SBR_H16_W
        LDW     *+DP(tmp),dest
        MVKL    $DPR_hword(sym),tmp         ;reloc R_C6000_SBR_L16_H
        MVKH    $DPR_hword(sym),tmp         ;reloc R_C6000_SBR_H16_H
        LDH     *+DP(tmp),dest
        MVKL    $DPR_byte(sym),tmp          ;reloc R_C6000_SBR_L16_B
        MVKH    $DPR_byte(sym),tmp          ;reloc R_C6000_SBR_H16_B
        LDB     *+DP(tmp),dest
```

4.2.1.3 絶対アドレッシング

次の命令は絶対アドレッシングを使用します。

```
        MVKL    sym,tmp                 ;reloc R_C6000_ABS_L16
        MVKH    sym,tmp                 ;reloc R_C6000_ABS_H16
        LDW     *tmp,dest
```

このアドレッシング モードはアドレスをエンコードするため、位置依存となります。これはファー データへのアクセスに使用できます。前に説明した ファー DP 相対スキームと比較して、実際のアクセス コストは同じですが、変数のアドレス (&sym) を計算する際に DP を加える必要がないため、1 命令分を節約できます。この場合、ロードされる定数がオフセットではなく実際のアドレスであるため、スケーリングは行われません。

4.2.1.4 GOT 間接アドレッシング

グローバル オフセット テーブル (GOT) は、静的リンク時に認識できないアドレスを動的に解決するために使用される位置非依存のメカニズムです。GOT 内のアドレスは動的ローダによって解決されます。GOT ベースのアドレッシングについては セクション 6.6 で説明しています。

4.2.1.5 PC 相対アドレッシング

これは、コード セグメント内の *ファー* データをアドレッシングする位置独立の方法です。データは、それにアクセスするコードからの (リンク時の) 一定のオフセット位置に配置されていると想定されます。例としては、switch 文に使用されるラベル テーブルや、コード セグメント (.const) に配置可能な読み取り専用定数変数などがあります。コードまたはデータのいずれにアクセスする場合でも、アドレッシングの仕組みは同一です (セクション 5.1 を参照)。

4.2.2 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 ニア DP 相対 アドレッシングに関して重要です。DP 相対アドレッシングでアドレッシングされる変数は、DP から 32KB 以内のセクションに割り当てられている必要があります。

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。

4.2.2.1 配置に関するアブストラクト規定

アブストラクト規定では、変数をニアまたはアーとして次のように指定します。

- ツールチェーン固有のキーワード、属性、プラグマでニアまたはファーとして指定された変数は、この指定に従います。
- ツールチェーン固有のキーワード、属性、プラグマで .bss、.rodata、.neardata 以外のセクションに属するように宣言された変数は、ファーとして指定されます。
- 残りの変数は、3 つのモデルのいずれかに従って指定され、通常はコマンド ライン オプションによって制御されます。
  - ニア モデル - 他の方法で指定されていないすべての変数は、ニアとして指定されます
  - ファー モデル - 他の方法で指定されていないすべての変数は、ファーとして指定されます
  - スカラー型の変数はニアとして指定されます。集成体 (すなわち、配列、クラス、構造、共用体) の変数はファーとして指定されます。これは、ツールチェーンのデフォルト モデルにすべきです。

ツールチェーンは他のモデルをサポートすることもありますが、少なくともこれら 3 つをサポートする必要があります。他のモデルが使用される場合、他のツールチェーンとの相互運用性は達成できる場合もあれば、できない場合もあります。

指定がコマンドライン オプションや言語拡張といったツールチェーン固有の要素に依存する場合、変数が宣言されるあらゆる場所でこれらの仕組みを一貫して使用する責任はプログラマにありますが、エラーを検出する責任はリンカにあります(「セクション 4.2.2.3」を参照)。

ABI はセクションへの変数の標準的な割り当てを定めます。変数がどのように割り当てられるかは、そのニア / ファーの指定と初期化カテゴリによって決まり、以下のリストで最初に一致する条件に従って決定されます。

- 変数は、初期化子を持たない場合、または起動時にコンストラクタ呼び出しによって初期化される場合、未初期化となります。
- 変数の型が const 修飾されている場合、変数は const です。
- 変数に初期化子がある場合は、初期化されます。

標準的なセクション割り当ては、表 4-1 に記載されています。

表 4-1. セクションへの変数の標準的な割り当て

	初期化カテゴリ		
指定	**初期化なし**	**初期化**	**定数**
ニア	.bss	.neardata	.rodata
ファー	.far	.fardata	.const

標準的な割り当ては、ツールチェーン固有の方法で上書きされる場合があります。たとえば、変数がユーザー定義のセクションに割り当てられることがあります。ただし、ツールチェーンは、ファーに指定された変数を 3 つのニア セクションに配置できないようにしなければなりません。

4.2.2.2 アドレッシングに関するアブストラクト規定

変数のアドレッシング方法は、その変数がニアかファーに指定されているか、その可視性、およびコード生成モデル (たとえば、位置独立か位置依存か) に依存します。

ニアと指定されたオブジェクトだけがニア DP 相対アドレッシングで参照できます。ニア オブジェクトも、絶対アドレッシング (位置依存) や GOT (位置独立) を通じた方法など、他の方法でもアドレッシングできる場合がありますが、これらの方法はいずれもニア配置と矛盾しません。

ファーと指定された変数は、ニア DP 相対アドレッシングではアドレッシングできません。

4.2.2.3 リンカの要件

リンカは、ニア DP 相対アドレッシングを使用してアドレッシングされる変数が、_ _C6000_DSBT_BASE symbol シンボルによって定められた、DP に必要とされる 15 ビット範囲内に配置されるようにする責任を負います。リンカは、そのようなアクセスが R_C6000_SBR_* 再配置エントリでマークされているものとして検出できます。リンカがこの制限を満たせない場合 (ユーザーからの指示が競合する場合など)、プログラムのリンクに失敗する必要があります。

4.2.3 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 18 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。

4.3 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 (B15) のいずれかを介してアドレッシングされますが、オフセットが大きすぎる場合には、アクティベーション フレームを指しより大きなオフセットを扱える一時的なフレーム ポインタ レジスタ (A15) を介してアドレッシングされます。

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。

スタックは上位アドレスから下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 (8 バイト) の境界に整列する必要があります。SP は現在割り当てられているスタックの直下 (より小さい) にある最初の整列されたアドレスを指します。

セクション 4.4 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。

4.4 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。

スタックはゼロに向かって増加します。SP は一番上の割り当てられた単語の上を指します。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 A15 をフレーム ポインタ (FP) として使用します。FP が割り当てられている場合、その値は、関数のフレームが作成される前の SP の値になります。つまり、FP は現在のフレームの一番下と呼び出し元の一番上を指します。フレーム内のオブジェクトには、負のオフセットがある FP を介してアクセスします。受信引数には FP を介してアクセスします。正のオフセット

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、スタックの巻き戻しに使用される仮想命令の一部では、A15 が前項のフレームを指していると仮定します。関数にフレーム ポインタがない場合、またはどのレジスタが使用されるか、どこを指すかについて異なる規則を使用する場合、これらの巻き戻し命令は使用できず、効率の低いシーケンスが必要になることがあります。

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。

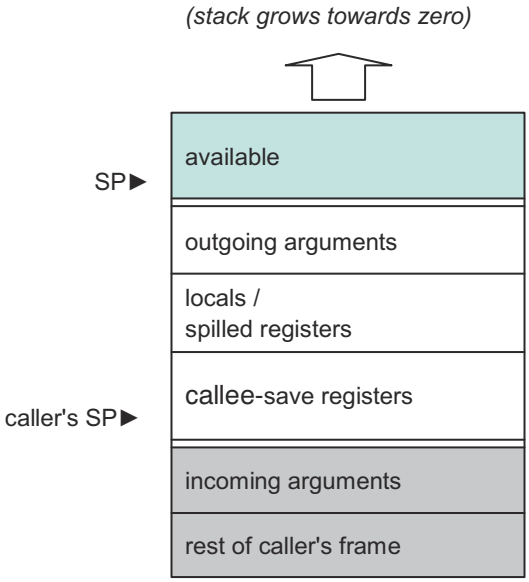


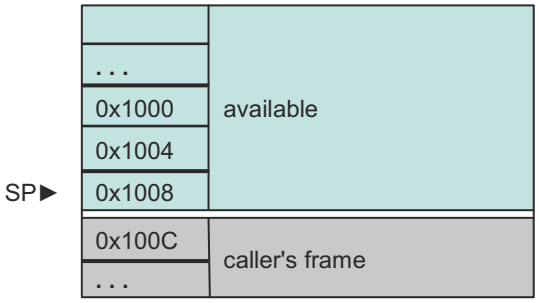
図 4-2. ローカル フレーム レイアウト

4.4.1 スタックの整列

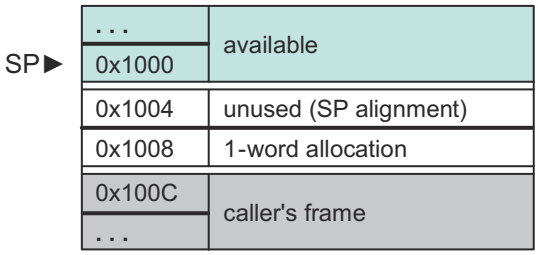
SP は 8 バイト境界に整列されており、フレームのアロケーションやアロケーション解除の間に割り込みが発生する場合に備えて、常に 8 バイト境界に整列されたままでなければなりません。したがって、SP を調整する操作は必ず 8 バイト単位で行う必要があります。

フレームの下端にあるダブル ワード (8 バイト) はフレーム境界をまたいで配置されます。つまり、最初のワードは呼び出し先のフレーム内にありますが、2 番目は呼び出し元のフレーム内にあるため、いずれもダブル ワードの格納には使用できません。これは、ダブル ワードのロードやストアを使用したレジスタの保存および復元の観点からは不都合ですが、もともとダブル ワードをサポートしていなかった過去のアーキテクチャから引き継がれた歴史的な仕様です。以下の図では、ダブル ワード境界は太線で示されています。

関数の最初の命令の前では、スタックは次のようになっています。



この関数がスタック上に 1 ワードを確保して何かを格納する必要がある場合、2 ワードのフレームを割り当てる必要があります (SP は常に 8 バイト境界に整列されていなければならないため)。アロケーションは SP を 8 ずつデクリメントすることによって行われます。その時点でスタックは次のようになります。



4.4.2 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**。安全なデバッグの規則は、シンボリック デバッグが有効になっている場合 (多くの場合 -g オプションによって指定) に適用されます。このモードでは、コンパイラはレジスタを固定された順序でスタックに保存および復元します。
- **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、一般的には順序は同じですが、以下に述べるいくつかの例外があります。

呼び出し先保存レジスタの安全なデバッグ順序は、A15、B15、B14、B13、B12、B11、B10、B3、A14、A13、A12、A11、A10 です。

安全なデバッグを使用し、かつ特別なスタック レイアウトが存在しない場合 (セクション 4.4.3 およびセクション 4.4.4 参照)、コンパイラは常にフレームの底 (最上位アドレス) からその相対的な順序で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。

4.4.2.1 ビッグ エンディアンにおけるペアのスワップ

64 ビットの LDDW 命令および STDW 命令のダブル ワードを持つターゲットでは、偶数・奇数のペアに属するレジスタをスタック上に並べ、そのペアを 1 回の LDDW で読み取れるようにする方が効率的です。リトル エンディアンにおける安全なデバッグ順序では、このようなレジスタ配置がしばしば行われますが、これは完全な偶然ではありません。ただし、ビッグ エンディアンでは各ペアの順序を逆にする必要があります。ビッグ エンディアン用にコンパイルする際、コンパイラはスタック上で同じように整列されたダブル ワードを占有するレジスタ ペアを探し、その順序をスワップします。これは、順序がリトル エンディアンと同じではなく、またビッグ エンディアンでは異なるレジスタを保存する関数ごとに順序が変わる場合があっても、依然として安全なデバッグ順序と見なされます。このスワップは、LDDW または STDW をサポートしない C6x ターゲットでも発生します。

レジスタをスタックに配置する際には、まず安全なデバッグ順序が参照されます。与えられた偶数・奇数ペアの順序は、そのオフセットが 8 で割り切れる場合にのみスワップされます。保存オフセットが整列していない場合、レジスタは元の順序で個別に保存されます。

4.4.2.2 例

C62x 用にコンパイルされた関数が、13 個すべての呼び出し先保存レジスタを保存する場合、保存領域は以下のようになります。ビッグ エンディアンの列で太字になっている項目は、ペアが入れ替えられていることを示します。

SP▶	*little-endian*	*big-endian*
. . .	rest of callee's frame	
0x1004		
0x1008	A10	**A11**
0x100C	A11	**A10**
0x1010	A12	**A13**
0x1014	A13	**A12**
0x1018	A14	A14
0x101C	B3	B3
0x1020	B10	**B11**
0x1024	B11	**B10**
0x1028	B12	**B13**
0x102C	B13	**B12**
0x1030	B14	**B15**
0x1034	B15	**B14**
0x1038	A15	A15
0x103C	caller's frame	
. . .		

図 4-3. 関数がすべての呼び出し先保存レジスタを保存する場合の C62x 保存領域

レジスタ B13、B12、A12、A11、A10 のみが保存される場合:

SP▶	*little-endian*	*big-endian*
. . .	rest of callee's frame	
0x1004		
0x1008	A10	**A11**
0x100C	A11	**A10**
0x1010	A12	A12
0x1014	B12	B12
0x1018	B13	B13
0x103C	caller's frame	
. . .		

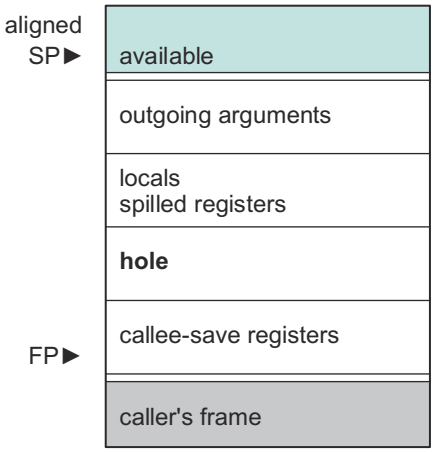
図 4-4. レジスタ B13、B12、A12、A11、A10 のみを保存する場合の C62x 保存領域

そのオフセットが正しく整列されていないため、B13:B12 が入れ替えられていないことが分かります。

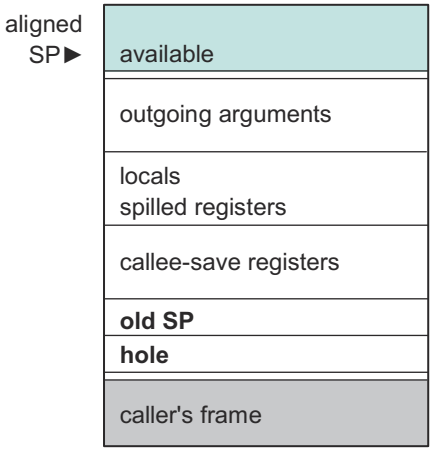
4.4.3 DATA_MEM_BANK

プラグマ DATA_MEM_BANK は、ローカル変数に特定の整列を保証するために、スタックにホールを作成します。これを実現するために、スタックに古い SP を保存し、SP の下位ビットをクリアします。

FP が割り当てられている場合、SP は単純に FP から復元されるため、古い SP をスタックに保存する必要はありません。



FP が割り当てられていない場合は、スタックに SP の古い値を保存する必要があります。



4.4.4 C64x+ 固有のスタック レイアウト

コード サイズを徹底的に削減するため、C64x+ および C674x の一部の関数では特殊なスタック レイアウトが用いられます。

4.4.4.1 _ _C6000_push_rts レイアウト

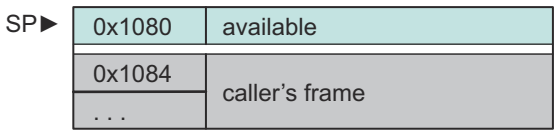
多くの関数は、呼び出し先保存レジスタをすべて保存して復元しますが、そのためのコードは相当に大きくなります。各関数のプロローグおよびエピローグにこの処理のコードを持つ代わりに、ランタイム ライブラリには代わりに呼び出すことのできる関数があります。これらの関数は、保存するレジスタを破壊しないようにするために、特別な呼び出し規則を使用します。すべての呼び出し先保存レジスタを保存する呼び出しは、次のようになります。

```
        CALLP __C6000_push_rts, A3      ; CALLP puts the return address in A3
```

復元するコードは次のとおりです。

```
        CALLP __C6000_pop_rts, A3       ; A3 is unused
                    ; Returns to the location saved from B3 by __C6000_push_rts
```

_ _C6000_push_rts が呼び出される前のスタックは次のようになります。



_ _C6000_push_rts には、すべての呼び出し先保存レジスタを格納し、その結果、次のようになります。

SP▶

	little-endian	*big-endian*
. . . 0x1048	available	
0x104C	unused (SP alignment)	
0x1050	X (pushed with B3)	B3
0x1054	B3	X (pushed with B3)
0x1058	A10	A11
0x105C	A11	A10
0x1060	B10	B11
0x1064	B11	B10
0x1068	A12	A13
0x106C	A13	A12
0x1070	B12	B13
0x1074	B13	B12
0x1078	A14	A15
0x107C	A15	A14
0x1080	B14	B14
0x1084 . . .	caller's frame	

4.4.4.2 コンパクト フレーム レイアウト

圧縮可能な命令の使用を促すため、C64x+ の一部の関数においては、レジスタの保存/復元コードが若干異なります。

通常、コンパイラは SP を 1 回デクリメントしてフレーム全体を割り当て、その後、SP 相対の書き込みで呼び出し先保存レジスタを保存します。

```
    STW   B14,     *SP--[12]
    STDW  A15:A14, *SP[5]
    STDW  A13:A12, *SP[4]
    STDW  A11:A10, *SP[3]
    STDW  B13:B12, *SP[2]
    STDW  B11:B10, *SP[1]
```

コンパクト フレーム モードでは、コンパイラは代わりに、各レジスタ ペアに対して一連の SP-auto-decrementing ストアを生成します。

```
    STW   B14,     *SP--[2]
    STDW  A15:A14, *SP--
    STDW  A13:A12, *SP--
    STDW  A11:A10, *SP--
    STDW  B13:B12, *SP--
    STDW  B11:B10, *SP--
```

この場合、スタック レイアウト自体は同じです。ただし、保存レジスタのサブセットが異なる場合には、スタック レイアウトが異なることがあります。たとえば、A10、A11、B10、B11、B3 を保存する必要があり、コード サイズを節約するためにコンパクト フレーム レイアウトを選んだとします。従来のレイアウトでは、スタック領域を最も効率的に利用できます。

```
STW   A11, *SP--[6]
STW   A10, *+SP[5]
STW   B3,  *+SP[4]
STW   B11, *+SP[3]
STW   B10, *+SP[2]
```

SP▶	0x1050	available
	0x1054	unused (SP alignment)
	0x1058	B10
	0x105C	B11
	0x1060	B3
	0x1064	A10
	0x1068	A11
	0x106C	caller's frame

ただし、コンパクト フレーム レイアウトは、より圧縮可能な SP デクリメント命令を使用するために、レジスタ保存領域に複数のホールを残します。すべてのプッシュは (割り込みの安全性のために) SP を 8 減算しなければならないため、コンパイラはレジスタ ペアのメンバーを一緒にプッシュしようとします。それができない場合、1 つのレジスタをダブル ワードにプッシュしなければならず、保存領域に穴が生じます。

```
STW   A11, *SP--[2]
STW   A10, *SP--[2]
STDW  B11:B10, *SP--
STW   B3, *SP--[2]
```

SP▶	. . . 0x1048	available
	0x104C	unused (SP alignment)
	0x1050	B3
	0x1054	unused
	0x1058	B11
	0x105C	B10
	0x1060	A10
	0x1064	unused
	0x1068	A11
	0x106C	caller's frame

4.5 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。

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。

C64x+ のコンパクト命令エンコーディング形式を除き、C6000 のすべての命令は 32 ビット幅です。関数アドレスを表すラベル、およびその他のほとんどのラベルは、常に 32 ビット境界に整列されます。コンパクト命令に関する考慮事項については、セクション 5.4 に記載されています。

コード オブジェクトを参照する方法は 3 つあります。そのアドレスを計算する方法、分岐先とする方法、または関数として呼び出す方法です。

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
- 呼び出しの復帰アドレスを形成する
- スイッチ テーブルに入力する
- リンカによって生成されたトランポリンまたは PLT エントリで使用する

絶対、PC 相対、および GOT ベースのアドレッシングという 3 つの方法でコード オブジェクトのアドレスを形成します。絶対アドレッシングは位置に依存し、PC 相対フォームと GOT フォームは位置に依存しません。

5.1.1 コードの絶対アドレッシング

基本的には、単純に宛先を絶対定数としてエンコードする方法をとります。

```
          MVKL label, B5       ; B5 := lower 16 bits of label
          MVKH label, B5       ; B5 := upper 16 bits
```

このような定数を直接エンコードする任意のコードは位置依存となり、たとえばロード時に再配置される場合には、パッチを必要とするという望ましくない性質を持つことになります。

5.1.2 PC 相対アドレッシング

これは、コード (またはコード セグメント内の定数データ) をアドレス指定するための、位置に依存しない方法です。アドレスは、現在のフェッチ パケットのアドレスと定数の合計として計算されます。

```
   base:  MVC   PCE1,tmp1                      ; address of current fetch packet
          MVK   $PCR_OFFSET(label,base),tmp2   ; label-base, reloc R_C6000_PCR_L16
          MVKH  $PCR_OFFSET(label,base),tmp2   ; label-base, reloc R_C6000_PCR_H16
          ADD   tmp1,tmp2,tmp2                 ; &label
```

$PCR_OFFSET アセンブリ演算子は、base (MVC) でラベル付きの命令を含むフェッチ パケットとターゲット シンボル間のオフセットを評価します。

5.1.3 同じセクション内での PC 相対アドレッシング

参照されるラベルが参照元と同じセクションにある場合、そのオフセットはアセンブリ時の定数になります。オフセットを 15 ビット以下でエンコードできる場合は、ADDK を使用してベース アドレスに直接追加できます。

```
   base:  MVC PCE1,tmp                      ; address of current fetch packet
          ADDK  label-(base & ~0x1F),tmp    ; no reloc; tmp == &label
          ...
   label: ; must be in same section
```

ここでは、再配置は必要ありません。アセンブラがオフセットを直接エンコードします。式 "(base&~0x1F)" は、base を含むフェッチ パケットのアドレスを表します。

(オフセットが大きすぎて ADDK でエンコードできない場合は、セクション 5.1.2 で説明されているように MVK/MVKH/ADD を使用する必要があります)。

5.1.4 短いオフセットの PC 相対アドレッシング (C64x)

C64x 以降の ISA には、近接するラベルに対して PC 相対アドレッシングを実装するための最適化された命令が備わっています。

```
          ADDKPC label, B5             ; B5 := label, position independent
```

この形式では、ラベルは ADDKPC 命令を含むフェッチ パケットから符号付き 7 ビット定数オフセット (± 64 ワード) 内に制約されます。

ADDKPC は復帰アドレスを計算するのに便利であり、これは多くの場合わずか数命令先に位置します。

5.1.5 コードに対する GOT ベースのアドレッシング

コード アドレスを計算するもう 1 つの位置独立の形式は、セクション 6.5 に記載されているように、グローバル オフセット テーブルからロードする形式です。

5.2 分岐

分岐は常に同一関数内にあると仮定されるため、常に PC 相対アドレッシングを使用でき、遅くとも静的リンク時までに解決されます。

エンコーディングでは 21 ビットの符号付きオフセットを 2 倍にスケーリングしたものを使用し、± 2^{22} バイト (4MB) の範囲となります。これにより事実上、任意の関数のサイズが 4MB に制限されます。

5.3 呼び出し

C6x には専用の呼び出し命令はありません。[1] 呼び出しは、復帰アドレスをレジスタ (B3) に生成し、分岐を実行することで生成されます。復帰アドレスの計算については、セクション 3.1 に説明があります。名前付き関数を直接呼び出す場合、PC 相対分岐が生成されるため、4MB の範囲制限を受けます。

5.3.1 直接 PC 相対呼び出し

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。

```
        CALL   sym                      ; reloc R_C6000_PCR_S21
```

注:CALL は擬似命令であり、この命令は B 命令としてエンコードされます。

5.3.2 ファー呼び出しトランポリン

呼び出しターゲットが同じ静的リンク ユニット内に定義されているが、21 ビットのワード オフセットでは到達できない場合、静的リンカはトランポリン (スタブ関数) を生成します。これは、別のアドレッシング方式を使用してターゲットに到達する関数です。次の例は、絶対アドレッシングを使用するトランポリンを示しています。[2]

```
   $Tramp$$sym2:
        MVKL sym,tmp                ;reloc R_C6000_ABS_L16
        MVKH sym,tmp                ;reloc R_C6000_ABS_H16
        B tmp
```

また、トランポリンは、他のアドレッシング方式を使用して、宛先関数のアドレスを計算することもできます (セクション 5.1 を参照)。

呼び出しターゲットが同じ静的リンク ユニットに定義されていない場合、静的リンカはトランポリンに類似した PLT エントリを生成します。このケースについてはセクション 6.5 で説明されています。

使用されるアドレッシング方式に応じて、トランポリンは通常 1 つまたは 2 つのスクラッチ レジスタ (前のシーケンスにおける tmp など) を必要とします。

C64x 以降のターゲットでは、B30 および B31 は任意のトランポリンで使用可能です。呼び出し元は、呼び出された関数がそれらを変更しないと分かっている場合であっても、呼び出しによって B30 および B31 が保持されると仮定することは許されません (セクション 3.7 を参照)。

旧式の C62 ターゲットおよび C67 ターゲットでは、トランポリンは使用したレジスタを保存および復元しなければなりません。

[1] C64x+ には CALLP 命令があり、これは ADDKPC と B に加えて NOP 命令を 5 回組み合わせて構成されています。
[2] トランポリン ラベルの名前は、規則により異なる場合があります。

5.3.3 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。

5.4 コンパクト命令をアドレッシングする

C64+ ISA といくつかの後継バリアントには、コンパクト命令として知られる機能があります。これは、16 ビット命令をペアにして、プログラム メモリの 32 ビット ワードにパックするエンコーディング形式です。ビッグ エンディアン モードでは、命令はソース プログラム中の記述順とは逆の順序でメモリに格納され、また実行順とも逆の順序になります。このセクションでは、この不一致に直面した場合のアドレス表記規則について説明します。

16 ビット命令は、次の 2 つのアドレスを持っていると見なされることがあります。

- **物理アドレス**は、その命令が実際に格納されているプログラム メモリ上のアドレスです。
- **論理アドレス**は、プログラムの制御フローの観点から見て、その命令が存在すると見なされるアドレスです。論理アドレスは、その命令に対応するプログラム カウンタ値と見なすことができます。プログラムは論理アドレスの順に命令を実行します。この順序は、プログラムの記述順に対応しています。分岐ターゲットとディスプレイスメントは、論理アドレスに従って計算されます。

リトル エンディアン構成では、論理アドレスは物理アドレスと同じです。ビッグ エンディアンでは、16 ビット命令のペアがプログラム メモリ内で入れ替えられ、アドレス A がそのペアを含む 32 ビット ワードの物理アドレスを表す場合、最初の論理命令は A+2 に、2 番目の論理命令は A に格納されます。

図 5-1 のコード断片は、論理アドレスと物理アドレスの違いを示しています。プログラムは、その記述順で示されています。1 列目はリトル エンディアンの物理アドレスを示しており、これはリトル エンディアンとビッグ エンディアンの両方において論理アドレスでもあります。2 列目はビッグ エンディアンの物理アドレスを示しています。破線はプログラム メモリにおける 32 ビット境界を表しています。

Physical Address		Opcode	Source Code		
Little-Endian	Big-Endian				
			code:		
0000	0002	40CE		MV.S1	A1,A2
0002	0000	41B0	||	ADD.L1	A2,A3,A3
0004	0006	58E7		NEG.L2	B1,B1
0006	0004	614F	||	MV.S2	B2,B3
0008	000A	814F		MV.S2	B2,B4
			local:		
000A	0008	A14F		MV.S2	B2,B5
000C	000E	C14F		MV.S2	B2,B6
000E	000A	40CE		MV.S1	A1,A2
0010	0012	41B0	||	ADD.L1	A2,A3,A3
0012	0010	58E7	||	NEG.L2	B1,B1

図 5-1. コンパクト命令をアドレッシングする

ABI では、オブジェクト ファイル内のすべてのプログラム アドレスが論理アドレスとして表現されることが規定されています。これには、分岐ディスプレイスメント、シンボル値、アンワインド テーブル内のアドレス、デバッグ情報中のアドレスが含まれます。

図 5-1 を参照すると、シンボル テーブルにおけるラベル code の値は 0x0000 であるにもかかわらず、それがラベル付けしている命令 (オペコード 0x40CE) は 0x0002 に格納されています。同様に、ラベル local の値は 0x000A であるにもかかわらず、それがラベル付けしている命令 (オペコード 0xA14F) は 0x0008 に格納されています。

論理アドレスと物理アドレスの違いは、ほとんどの場合、プログラマやツールチェーンにとっては透過的です。この透明性を維持するために、次の条件が課されています。

1. 分岐はラベル付き命令に対して行わなければなりません。ラベルのない命令への分岐を構築することは可能ですが、その結果は未定義です。
2. いくつかの例外を除き、ラベルは 32 ビット境界に整列していなければなりません。例外は以下に示すとおりです。
   a. 分岐先でないと判定できるラベル (DWARF ラベルなど)
   b. 同一セクション内のフェッチ パケット間の分岐のターゲットとなるラベル (BNOP のコンパクト形式はハーフワード オフセットをエンコードします)。
3. なお、再配置可能フィールドは 16 ビット命令内に配置できません。

条件 (1) と (2) を組み合わせることで、32 ビット境界に整列していないアドレスへの間接分岐は排除されます。なお、条件 (2b) のようなケースでは、オフセットがアセンブリ時の定数であるため、再配置は不要です。この仕組みにより、条件 (3) も成立し、再配置可能フィールドにアクセスする際に論理アドレスと物理アドレスを変換する必要がなくなります。

Chapter 6

動的リンクにおけるアドレッシング モデル

最も基本的なシナリオでは、ベアメタル環境向けにスタンドアロン プログラムを構築する場合、プログラムは静的にリンクされ、特定のアドレスで実行されるように固定されます。リンカが単純にすべての参照を最終的に解決されたアドレスでパッチすることで、プログラムを実行する準備が整います。このシナリオはシンプルで効率的です。

組み込みシステムでさえ、複数のコンポーネントがますます個別にリンクされるようになっています。これは当然、汎用システムで共通の動的リンク モデルにつながります。すなわち、Windows におけるダイナミックリンク ライブラリ (DLL) や、Linux を含む Unix 系プラットフォームにおけるダイナミック共有オブジェクト (DSO) です。このセクションでは、C6000 向けの基本レベルの動的リンクおよび共有オブジェクト機構に関する一連の規則について説明します。動的リンクに関連するオブジェクト ファイルの機構については、「セクション 14.3」で取り上げています。Linux のような特定の実行プラットフォームでは、追加の規則が指定される場合があります。「Chapter 15」を参照してください。

6.1 用語と概念

*静的リンク*とは、再配置可能なオブジェクト ファイルと静的ライブラリを結合して*静的リンク ユニット*を生成する従来のプロセスのことです。これは ELF *実行可能ファイル* (.exe) または ELF *共有オブジェクト* (.so) のいずれかになります。このドキュメントでは、*ロード モジュール* (または単にモジュール) という用語を静的リンク ユニットに対して使用し、*共有ライブラリ* (またはライブラリ) という用語を共有オブジェクトに対して使用します。

プログラム は、必ず 1 つの実行可能ファイルと、そのファイルが未定義参照を解決するために依存する共有ライブラリで構成されます。複数の実行可能ファイルが同じライブラリに依存する場合、それらはライブラリのコードの 1 つのコピーを共有でき (これが共有オブジェクトにおける「共有」の意味です)、これによりシステムのメモリ要件を大幅に削減できます。

複数オブジェクトから構成されるこのようなプログラムがロードされるとき、コンポーネント モジュール間の参照 (その一部はすでに他のアプリケーションの一部としてロードされている可能性があります) を解決する必要があります。このプロセスは*動的リンク*と呼ばれ、*動的リンカ*と呼ばれるランタイム コンポーネントによって処理されます。システムの状態は、任意の時点でどのオブジェクトがロードされているかによって異なるため、動的リンカはメモリ割り当てや配置を動的に制御することが求められる場合があります。プログラムの配置をロード時に割り当て、再配置できる機能は、*動的ロード*と呼ばれることもあります。動的リンクと動的ロードは、それぞれが単独でも有用であるという意味ではやや独立した機能ですが、それぞれを可能にするしくみは密接に関連しています。このドキュメントでは、動的リンクという用語をこの複合機能に対して使用し、動的リンカと動的ローダという用語を、これらの操作を実行するコンポーネントに対して区別なく使用します。

オブジェクト *自身の* 関数および変数 (総称して *シンボル*) は、そのオブジェクト内で定義されているものです。モジュール (実行可能ファイルまたはライブラリ) が、そのモジュール内では未定義であるが他のモジュールで定義されているシンボルを参照する場合、そのシンボルを *インポートする* と言います。その定義するモジュールは、そのシンボルを *エクスポートする* と言います。

一般に、動的にリンクされるモジュールのコードおよびデータのアドレスは、静的リンク時には不明です。さらに、インポートされたシンボルのアドレスも、動的リンカによって解決されるまでは不明です。したがって、動的リンカがモジュールをロードするとき、そのモジュールに割り当てられたアドレスや、インポートするシンボルのアドレスに応じて、そのコードまたはデータに対してパッチの適用が必要になる場合があります。動的リンク時に実行される再配置は、*動的再配置* と呼ばれます。ほとんどの動的リンクのしくみの設計は、動的再配置の数と複雑さを最小化することを目標としています。動的再配置および関連するシンボル情報は、ELF オブジェクト ファイル内の特別なセクションに格納されます。

共有ライブラリの基本的な課題は、そのライブラリを共有するすべての実行可能ファイルが、ライブラリのデータについては共有せず、***自分専用の*** コピーを持たなければならない点です。これは、共有コードがデータにアクセスするために絶対アドレッシングを使用できないことを意味します。*位置独立データ (PID)* という用語は、通常は相対アドレッシングまたは GOT ベースのアドレッシングによって、共有可能な方法でデータにアクセスするコードに適用されます。

より広い用語である *位置依存コード (PIC)* は、絶対アドレッシングを一切使用せず、自身の配置および他のロード モジュールの配置のいずれにも依存しないコードを指します。位置独立コードはコード セグメントにロード時のパッチを必要としないため、ロード時間を短縮したり、ROM に配置できるようにしたりできます。PIC の典型的な実現方法は、PC 相対アドレッシング、仮想メモリ、間接アドレッシング、あるいは C60 の DP (B14) のようなベース ポインタ レジスタからの相対アドレッシングに依存します。

ROM に配置されるモジュールには追加の考慮事項が適用されます。明らかに ROM 内のコードはロード時にパッチを適用できないため、位置の独立性に関して多くの類似した要件があります。

6.2 動的リンク機構の概要

ABI は、次のような関連メカニズムを通じてこれらの問題に対処します。

一般的な ***ELF 動的リンク*** メカニズムは、ロード時のシンボル解決と再配置をサポートするオブジェクト ファイル表現を定義します。これらの大部分はターゲット独立であり、GABI によって規定されています。ターゲット固有のものについては、セクション 14.3 に記載されています。

*プロシージャ リンケージ テーブル (PLT) エントリ*とは、インポートされた関数への呼び出しを解決するためにリンカによって生成されるスタブです。

グローバル オフセット テーブル (GOT) は、インポートされたオブジェクトを参照するためのアドレッシング手法であり、アドレス定数をコードにエンコードするのではなくデータセクション内のテーブルに配置することで、位置の独立性とプライベート化をサポートします。こうした利点を得る代わりに、GOT 参照では間接参照が 1 回余分に必要となり、さらにテーブル用に追加のデータ領域を確保する必要があります。

データ セグメント ベース テーブル (DSBT) モデル は、各コンポーネントが専用のデータ セグメントを持つことを可能にするソフトウェア規則であり、これにより他のコンポーネントを考慮することなく自分のデータ参照を静的に解決できます。DSBT メカニズムにより、仮想メモリを用いずに位置独立コードが可能になり、共有コードの 1 つのインスタンスが動的にバインドされたプライベート データの複数コピーをアドレッシングできるようになります。

6.3 DSO と DLL

システムによって、対応している動的リンクのモデルは異なります。Linux を含む UNIX システムでは、アプリケーション側からは動的リンクを意識しなくてもよいように設計されています。つまり、プログラムやライブラリは、未解決の参照が静的に解決されるか動的に解決されるかを気にせずに作成・コンパイルできます。たとえば、プログラムで「extern int f()」を宣言して f を呼び出した場合、コンパイラは f を静的にも動的にも解決できるコードを生成します。この方式の最大の利点は柔軟性です。つまり、プログラムはどのようにリンクされるかを気にせずに記述し、コンパイルできます。一方で欠点は効率が下がる場合があるという点です。コンパイラは、extern 参照が静的に解決されない可能性を常に考慮し、動的リンクに対応するアドレス指定コードを生成しなければならないためです。

Unix では、動的リンク ライブラリは動的共有オブジェクト *(DSO)* と呼ばれます。

Windows や Symbian、PalmOS などの組込みシステムでは、動的リンクはソース コードのシンボル宣言で明示的に指定され、通常は言語拡張の __declspec(import) が用いられます。この方式の利点は、コンパイラが特別なアドレス指定を生成すべき場面を明確に把握できることです。これらのシステムでは、リンク後の処理段階でシンボル参照による動的リンクをシンボル インデックス方式に置き換えるのが一般的です。このようなシステムでは共有ライブラリをダイナミック リンク ライブラリ *(DLL)* と呼びます。

6.4 プリエンプション

あるオブジェクトが別のオブジェクトで定義されているグローバル シンボルを参照するとき、そのシンボルをインポートすると言い、定義側のオブジェクトはそれをエクスポートすると言います。2 つの異なるオブジェクトが同じシンボルを定義・エクスポートしている場合、一方の定義が優先され、他方は *プリエンプト* されます。プリエンプションにより、動的リンクが静的リンクと同様の動作を実現できます。すなわち、実行可能ファイル側の定義がライブラリ側の定義をプリエンプトするため、ライブラリの定義はリンクされません。動的リンクの場合、ライブラリはすでにロードされている可能性があり、共有インスタンス内の定義が特定のクライアントでは必要で、他では不要ということもあります。

プリエンプションとは、静的リンク時にモジュール内で定義されているように見えるシンボルが、実際には動的リンク時に別の定義に置き換えられる可能性があることを意味します。これはコンパイラに影響を与え、コンパイラは当該シンボルがインポートされるものとしてコードを生成する必要があります。このような理由から、たとえ実際にプリエンプションが発生しない場合であっても、それはコストのかかる処理となります。パフォーマンスへの影響については、セクション 6.8 で説明されています。Linux では、実行可能ファイルのペナルティを軽減するために import-as-own と呼ばれる技法が使用されています (セクション 15.9 を参照)。

ELF シンボル テーブルにあるシンボル可視性フィールドは、シンボルがプリエンプト可能かどうかを示します。STV_HIDDEN または STV_INTERNAL にマークされたシンボルはエクスポートされません (したがってプリエンプトされません)。STV_PROTECTED にマークされたシンボルはエクスポートされますが、プリエンプトできません。STV_DEFAULT にマークされたシンボルはプリエンプトできます。

どのシンボルがプリエンプトできるか、またプログラマが可視性をどのように指定するかについては、プラットフォームやツールチェーンごとに異なる規則が適用されます。

6.5 PLT エントリ

通常、コンパイラが外部関数の呼び出しを検出すると、呼び出し先の関数がどこにあるかに関係なく、単純に CALL 命令を生成します。静的リンク時にその関数が別のソース ファイルや静的リンク ライブラリ内で定義されている場合、リンカは CALL 命令内の変位フィールドを単純に再配置して参照を解決します。

関数が共有ライブラリからインポートされると、そのアドレスは静的リンク時には不明であり、最終的に動的リンク時に解決されます。インポートされた関数をアドレス指定して呼び出すには、追加の命令が必要になる場合があります。その場合に備え、また動的リンク時に呼び出しをパッチしなくても済むように、静的リンカは代わりに、その関数を呼び出すための位置に依存しないスタブを生成し、元の呼び出しがこのスタブを経由するようにパッチを適用します。このスタブは PLT エントリと呼ばれます。PLT は *Procedure Linkage Table* (プロシージャ リンケージ テーブル) の略です。(PLT をテーブルと呼ぶのは、以前の実装に由来しています。そのエントリは独立して生成されたコード断片であり、ひとまとまりのエンティティとして収集されるわけではありません)。PLT エントリは、概念的には far コール トランポリンに似ています (セクション 3.1 を参照)。トランポリンの目的が、遠く離れた関数を呼び出すことであるのに対し、PLT エントはインポートされた関数を呼び出します。

6.5.1 インポートされた関数への直接呼び出し

PLT スタブは、呼び出しが行われるコード セグメント内に生成されます。PLT は、セクション 5.1 に記載された考慮事項に従って、宛先関数のアドレスをエンコードします。

6.5.2 絶対アドレスによる PLT エントリ

```
$sym$plt:
        MVKL   sym,tmp                  ;reloc R_C6000_ABS_L16
        MVKH   sym,tmp                  ;reloc R_C6000_ABS_H16
        B      tmp
```

次のセクションで説明される、tmp の選択に関するわずかな違いを除けば、このコード シーケンスはファー呼び出しトランポリンと同じです。

6.5.3 GOT による PLT エントリ

関数がプリエンプトされる可能性がある場合、たとえ位置独立の方法であっても、そのアドレスは PLT エントリ内にエンコードできません。アドレスは、GOT を通じて間接的にアドレッシングする必要があります。

```
$sym$plt:
        LDW    *+DP($GOT(sym)),tmp      ;reloc R_C6000_SBR_GOT_U15
        B      tmp
```

一部のコンパイラ ヘルパー関数は標準的でないレジスタ保存規則 (セクション 8.3) を使用しており、これが tmp 用に使用されるレジスタの選択に影響を及ぼします。さらに、遅延バインディング (セクション 15.6) は、PLT エントリで直接言及されているレジスタ以外にも影響を与える可能性があります。そのため、ABI では、標準以外の呼び出し規則を持つ関数はインポートできない、すなわち PLT エントリを介して呼び出すことはできないと規定しています。この規定により、リンカは、関数呼び出しインターフェイスに関与しない呼び出し元保存レジスタを PLT エントリ内で自由に変更できます。

コンパイラは、インポートされた関数であることが明確または疑わしい関数の呼び出しに対して、PLT エントリをインライン化することを選択できます。これにはコード サイズの増加と引き換えに、追加の分岐による待ち時間を減らすという利点があります。

動的ローダが、セクション 15.6 に記載されているように遅延バインディングを使用する場合、インライン化された PLT エントリはそこで定められた規則に従う必要があります。あるいは、インライン化された PLT において GOT 再配置を生成し、それを動的再配置テーブルの DT_JMPREL 部分から除外することもできます (System V ABI の第 5 章「*動的セクション*」参照)。これにより、それらは遅延バインディングの対象にはなりません。

6.6 グローバル オフセット テーブル

完全な位置の独立性とは、コードが自身の位置、自身のデータの位置、インポートされたコードやデータの位置に依存せず、ロード時の再配置パッチを必要としないことを意味します。この文脈において「自身の」とは、参照と同じ静的リンク ユニットの一部を意味します。以下に、それぞれのケースの意味を検討します。

- 独自のコードへの参照 (セクション 5.1):PC 相対アドレッシングまたは GOT ベースのアドレッシングを使用する必要があります。絶対アドレスは使用できません。このケースでは、トランポリン、スイッチ テーブル、復帰アドレス計算が影響を受けます。
- 独自のデータへの参照 (セクション 4.2):DP 相対、PC 相対、GOT ベースのアドレッシングを使用する必要があります。絶対アドレスは使用できません。一般に、この選択はコンパイル時に行う必要があります。このケースでは、ニア データおよびファー データへの参照が影響を受けます。
- インポートされたコードへの参照:絶対アドレッシングや PC 相対アドレッシングは使用できません。このケースは PLT エントリで生成される呼び出しに適用されます。
- インポートされたデータへの参照:絶対アドレスや DP 相対アドレッシングを使用できません。このケースはインポートされたデータへの任意の参照に適用されます。

位置依存の絶対アドレスをコード セグメントにエンコードしないように、それらは各静的リンク ユニットのデータ セグメントの一部であるグローバル オフセット テーブル (GOT) と呼ばれるテーブルに生成されます。オブジェクトに直接アクセスする代わりに、プログラムは GOT からシンボルのアドレスを読み取り、それを間接的にアドレッシングします。GOT はデータ セグメントの一部であり、常に静的リンク時に固定されたオフセットを使用して DP 相対でアドレッシングされます。これは、コンパイラによって出力される特別な GOT 生成再配置に応じてリンカによって生成されます。GOT 内のアドレスは、動的リンク時に、実際にそのシンボルが配置されたアドレスが判明した時点でパッチされます。

GOT ベースのアクセスは 2 回のメモリ参照を伴います。1 回は GOT からアドレスをロードするためのもので、もう 1 回は変数自体を参照するためのものです。GOT 自体にアクセスする最初の参照は、基本的に通常の DP 相対データ アクセスと同じです (セクション 4.2.1 を参照)。ほとんどの場合、GOT はニア DP セグメントに存在すると予想されるため、ニア DP 相対アドレッシングを使用してアクセスできます。

6.6.1 ニア DP 相対アドレッシングを用いた GOT ベースの参照

ニア DP 相対アドレッシング形式を使用した完全な GOT ベースの参照は次のようになります。

```
          LDW   *+DP($GOT(sym)),tmp     ;reloc R_C6000_SBR_GOT_U15
          LDW   *tmp,dest
```

ここで示されている再配置は、静的リンカに GOT エントリを割り当てさせ、その DP 相対オフセットに評価させます。テーブル エントリ自体は動的再配置でマークされ、そのシンボルのアドレスに評価されます。

6.6.2 ファー DP 相対アドレッシングを用いた GOT ベースの参照

完全性を保つために、ABI では GOT 自体が DP の 15 ビット オフセット範囲外、すなわちファーにある場合の GOT ベースのアドレッシングもサポートします。この場合、ファー DP 相対アドレッシングを使用して GOT にアクセスします。

```
          MVKL  $DPR_GOT(sym),tmp       ;reloc R_C6000_SBR_GOT_L16
          MVKH  $DPR_GOT(sym),tmp       ;reloc R_C6000_SBR_GOT_H16
          LDW   *+DP[tmp],tmp2
          LDW   *tmp2,dest
```

6.7 DSBT モデル

ライブラリのコードを共有する各実行可能ファイルは、ライブラリのデータの独自の自分専用のコピーを割り当てる必要があります。さらに、各静的リンク ユニット自身のデータ (GOT を含む) は、静的リンク時に固定されるオフセットを用いて、DP 相対アドレッシングによってアドレッシングされます。(MMU を備えたシステムでは、通常これは PC 相対アドレッシングを使用して位置独立の仮想オフセットを実現し、アドレス変換を使用して複数の物理コピーのデータ セグメントを同一の (仮想) アドレスにインスタンス化することによって達成されます)。C6000 のように MMU を持たないシステムでは、通常、何らかの形式の静的ベース ポインタ (DP) とオフセット アドレッシングに依存します。

与えられた静的リンク ユニットからのすべてのアドレッシングは、そのデータ セグメントに対して相対的であり、したがって他の静的リンク ユニットからは独立しています。その結果、実行可能ファイルおよび 1 つ以上の (おそらく共有される) ライブラリから構成されるプログラムが複数のデータ セグメントを持ち、それぞれが DP 相対オフセットの基準となる異なるアドレスを持つモデルとなります。制御があるモジュールから別のモジュールに移るとき、新しいモジュールのデータ セグメントのベース アドレスに DP を変更する必要があります。

このモデルにおける一般的な問題は、ほとんどの静的ベース アドレッシング方式に共通するものであり、それは次のような問題になります。

- DP を変更するのは誰か。呼び出し元か、それとも呼び出し先か
- 新しい DP 値はどのように決定されるのか
- 間接呼び出しはどのように処理されるのか

FDPIC、XFLAT、DSBT など、他のアーキテクチャ向けにはさまざまな解決策が採用されています。TI では、効率性、互換性、柔軟性の最良の妥協点として、DSBT モデルを採用することを選択しました。

インポートされた関数が呼び出されたとき、呼び出し先は、自身のデータ セグメント (より正確には、呼び出される関数を含むモジュールのデータ セグメントに対応する基礎となる実行可能ファイルの自分専用のコピー) を指すように DP を設定し、戻る際にそれを復元する責任を負います。

呼び出し先がこれをどのように実現するかを説明する前に、前提として理解しておくべき 2 つの事実があります。第一に、各モジュールにはそれ自身のデータ セグメントがあり、それぞれ独自のベース アドレスを持ち、複数の実行可能ファイルで共有される場合でも、それぞれが異なるベース アドレスを持つ自分専用のコピーを持ちます。さらに、これらのアドレスは動的に決定されます。したがって、GOT に格納されるアドレスと同様に、そのベース アドレスは絶対アドレスではないため、データ セグメント内に格納する必要があります。

第二に、*呼び出し先* が DP を変更する責任を持つものの、*呼び出し先* に入った時点では DP は依然として呼び出し元のデータ セグメントを指しています。つまり、呼び出し先が持っているのは呼び出し元のコンテキストだけであり、それを頼りに *自分の* DP を設定しなければなりません。

その解決策として、各データ セグメントの先頭部分に データ セグメント ベース テーブル (DSBT) と呼ばれるテーブルのコピーを配置します。このテーブルには、プログラムを構成する他のモジュールのすべてのデータ セグメントのベース アドレスが一覧として入っています。各共有ライブラリには、1 から始まる一意のインデックスが割り当てられます。インデックス 0 は実行可能ファイル用に予約されています。呼び出し先は、自身に割り当てられたインデックスを使用して呼び出し元のテーブルのコピーから自身のベース アドレスを検索し、その値を DP に割り当てます。与えられた実行可能ファイルとその共有ライブラリの自分専用のデータ セグメント内では、各 DSBT コピーは同一であるため、呼び出し先は任意の呼び出し元のテーブルを使用して自身のベース アドレスを見つけることができます。

この DSBT モデルの利点は、動的リンクのペナルティがエクスポートされた関数にのみ発生する点です。動的リンクを使用しないベアメタル プログラムや、エクスポートされた関数を持たない実行可能ファイルの ABI には影響がありません。ツールチェーン固有の宣言方法を適切に使用して、外部からアクセス可能な関数を明示的に指定すれば、プログラマはオーバーヘッドを最小化できます (セクション 6.7.2 を参照)。DP を調整する必要がある関数において、そのオーバーヘッドは通常わずか 3 つの命令になります。

DSBT モデルの欠点は、ライブラリ インデックスの割り当てを調整し、モジュールの最大数に関する合意を強制する必要があることです。これが、各データ セグメント内のテーブルのサイズを決定します。

DSBT は、静的リンカによって .dsbt セクションに割り当てられ、DP がそれを指すように、各モジュールの最初の DP 相対セグメントのベース アドレスに配置される必要があります。動的リンカは、モジュールがロードされるときにテーブル エントリを初期化します。

実行可能ファイルは常にインデックス 0 を使用してテーブルにアクセスします。ライブラリ インデックスは、特定のプラットフォームに指定されたように、1、2、またはその他のインデックスから始まります。ライブラリのインデックスは次の 2 つの方法のいずれかで割り当てられます。

- コマンド ライン オプションまたはその他のディレクティブを介して、静的リンク時に (または同等に、静的ポストリンク ツールによって) 静的に割り当てる。この方法は、ライブラリが ROM 常駐であり、動的ロード時に再配置できない場合に使用する必要があります。
- 動的リンカによって動的に割り当てる。この場合、ライブラリのコード セグメントをロード時に再配置 (パッチ適用) してインデックスを更新する必要があるため、動的に割り当てられたインデックスを持つライブラリは位置独立とはみなされません。

各オブジェクトの DSBT は、そのプログラムの一部として動的にロードされる任意のモジュールに割り当てられた最大インデックス以上の大きさでなければなりません。動的リンカには、すべてのモジュールが十分なサイズの DSBT を持っていることを確認する責任があります。もし不足していれば、動的リンカはプログラムのロードを失敗させなければなりません。DSBT のサイズは、コマンド ライン オプションまたは環境変数を介して、静的リンク時に (または静的ポストリンク ツールに対して) 指定されます。通常、組込みシステムでは必要な動的ライブラリの数は少ないため、一般的に DSBT のサイズは 5 以下で済みます。

モジュールの動的セクションには、その DSBT テーブルのサイズおよび割り当てられている場合はそのインデックスを指定する C6000 固有のタグが含まれます。詳細については、セクション 14.3.2 を参照してください。

6.7.1 エクスポートされた関数のエントリ/終了シーケンス

次のコード シーケンスは、エクスポートされた関数が、呼び出し元の DSBT をインデックス指定することによって、自身のデータ セグメントを指すように DP を変更する方法を示しています。DP を変更する関数は、リターン時に DP を復元する責任を負います (DP は呼び出し先保存です)。

DP を設定するためのエントリ シーケンス

```
func:
        MV      DP,somewhere                ;typically the stack
        LD      *+DP[$DSBT_index(func)],DP ;reloc R_C6000_DSBT_INDEX
        ; body of function
```

式 $DSBT_index(func) を評価すると、現在のオブジェクトの一意のライブラリ インデックスが得られ、これを示す特別な再配置が生成されます。このインデックスは、静的リンク時または動的にバインドされます。

終了シーケンス

```
        MV      somewhere, DP
        RET
```

終了シーケンスは単純に呼び出し元の DP を復元します。

エクスポートされた関数は、DP 相対アドレッシングを使用せず、かつ DP 相対アドレッシングを使用する関数を呼び出さない場合、DP を変更しないことを選択できます。

6.7.2 内部関数の DP 負荷を回避する

DP を調整する必要があるのは、他のリンク ユニットから呼び出される関数のみです。自分の静的リンク ユニット内からのみ呼び出される関数は、呼び出し元がすでに DP を調整しているため、DP を調整する必要はありません。関数が外部から呼び出し可能であるかどうかの設定は、その関数の *可視性* と呼ばれます。(注: 可視性はオブジェクトがプリエンプト可能であるかどうかにも適用されます。「セクション 6.4」を参照)

別のリンク ユニットからの外部呼び出しは直接呼び出しか、あるいは間接呼び出しかのいずれかです。直接呼び出しの場合は関数が名前で呼び出され、間接呼び出しの場合は関数のアドレスが取得されて外部呼び出し元に渡され、そのアドレスを介して呼び出されます。ELF は 4 段階の可視性を提供しており、それは他のモジュールからの直接呼び出しおよび間接呼び出しのさまざまなケースをカバーしています (概要については「表 6-1」を参照)。

表 6-1. ELF の可視性属性の解釈

名称	直接呼び出し可能	間接呼び出し可能	プリエンプト可能
STV_DEFAULT	あり	あり	あり
STV_PROTECTED	あり	あり	なし
STV_HIDDEN	なし	あり	なし
STV_INTERNAL	なし	なし	なし

関数の可視性は、その宣言とコンパイラおよびプラットフォーム固有の規則の組み合わせによって決まります。たとえば、Linux モデルでは、外部関数は宣言に _ _attribute_ _((visibility)) 修飾子で宣言を拡張することによって別途指定されない限り、STV_DEFAULT の可視性を持ちます。しかし、ベアメタル プラットフォームでは、デフォルトの可視性として STV_HIDDEN または STV_INTERNAL の方がより適切である場合があります。

6.7.3 関数ポインタ

一般的に、呼び出し先は自身の DP を設定する責任があるため、関数ポインタに特別な処理は必要ありません。エクスポートされた関数は、定義されているモジュールの内部または外部から間接的に安全に呼び出すことができます。(これは、他の MMU レスの手法と比較して、DSBT モデルの大きな利点です)。

ただし、関数ポインタを利用する際には注意すべきリスクがあります。内部可視性を持つ関数が、アドレスを取得され、それが別の静的リンク ユニットに渡され、さらに間接的に呼び出されると、DP が正しく設定されず、プログラムが失敗する可能性があります。

内部 関数のアドレスを取得して別のモジュールで利用可能にすることは、厳密に言えば、可視性宣言によって暗示される前提に違反するため、プログラミング エラーです。このような違反を検出するために、ツールチェーンは、非エクスポート関数のアドレスが取得された際にコンパイラが警告を出すように設定できる場合があります。正当な理由でこの操作を行っているユーザーは、その警告を無効化できます。

外部 関数のアドレスを取得して別のモジュールに渡すことは、常に正しい操作です。異なるモジュールで計算された関数ポインタの比較が期待どおりに動作するようにするために、ABI では、関数のアドレスを表す式がすべてのモジュール間で一意の値に評価されることが求められます。他のアーキテクチャ向けの一部の ABI は、実行可能ファイル内で関数のアドレスへの参照が PLT エントリに解決されることを許す規則を採用しており、それによりこれらの参照を静的に解決できるようにしています。(共有オブジェクトからの参照は、プリエンプションのため動的に解決される必要があります)。

C6000 ABI では、関数ポインタを介したモジュール間呼び出しに問題が発生するため、この規則を採用していません。そのようなポインタが PLT エントリに解決されると、間接呼び出しがその PLT エントリに到達したときに、DP 値が別の静的リンク ユニットのものとなり、PLT エントリが GOT にアクセスできなくなってしまう可能性があるためです。実際には、PLT エントリは内部関数であるため、モジュール外部から間接的に呼び出すことはできません。

したがって、C6000 ABI の規則としては、関数のアドレスへの参照は関数の実際のアドレスに解決される必要があります。つまり、インポートされたオブジェクトに対しては、そのような参照は静的に解決できず、ロード時に動的リンカによって解決されなければならない、ということになります。

6.7.4 割り込み

共有ライブラリを持たないスタンドアロン アプリケーションでは、DP は決して変更されません。この規則がシステム全体で維持されると仮定すると、割り込みサービス ルーチンは DP が唯一の RW セグメントを指していると確実に仮定できます。

動的リンクが存在する場合、割り込みルーチンは DP について何も仮定できません。任意のエクスポート関数と同様に、DP を自分自身で保存し、設定し、復元しなければなりません。

6.7.5 非 DSBT コードとの互換性

DSBT モデルは、位置の独立性および共有ライブラリをサポートするために ABI のバリアントとして提供されています。多くの組み込みシステムはこれらの機能を必要とせず、それゆえ追加の複雑さや性能上のオーバーヘッドを回避できます。DSBT モデルを使用するコードは、それを使用しないコードとはバイナリ互換性がありません。オブジェクト ファイル内のビルド属性は、それが DSBT モデルを使用して構築されたことを示します。リンカおよびローダは、DSBT コードが非 DSBT コードと混在することを防ぐ必要があります。

6.8 動的リンクによるパフォーマンスへの影響

動的リンクにはパフォーマンス上のペナルティがあります。PLT を介して呼び出されるインポートされた関数では、トランポリンのように追加の呼び出しオーバーヘッドが発生します。関数のアドレスに GOT を介してアクセスする場合も、アドレスを読み込むための間接参照によるオーバーヘッドが発生します。

DP を介したニア データのアドレッシングにはペナルティはありません。ファー データの場合、DP 相対アドレッシングには 3 つの命令が必要ですが、位置依存の絶対アドレッシングでは 2 つの命令で済みます。GOT を介してアドレッシングされるオブジェクトについては、アドレスをロードするために GOT を追加参照するオーバーヘッドが発生します。

シンボルのプリエンプションは、GOT によるペナルティを著しく悪化させます。プリエンプトされる可能性のあるすべてのシンボル、すなわち共有ライブラリ内で定義されたすべてのグローバル シンボルは、コンパイラおよび静的リンカによってインポートされたものとして扱う必要があります。ローカルに定義された関数であっても、PLT 経由で呼び出される必要があり、これによりインライン化や特殊化ができなくなります。ローカルに定義された変数は、GOT を介して間接アクセスする必要があります。これらの制限はコンパイラが生成するコードに適用されるため、最終的にそのシンボルがプリエンプトされなかったとしても、損失は通常回復できません。

プリエンプションによるペナルティは、共有ライブラリに対してのみ適用されます。実行可能ファイルで定義されたシンボル (つまりライブラリではないもの) はプリエンプトされることはありません。

システムでは、こうした影響を軽減するためにいくつかの手法が採用されています。Windows、Palm、Symbian などの DLL モデルを採用するシステムでは、シンボルが明示的にエクスポート宣言されない限り、定義されたシンボルはエクスポートされたものとはみなされません。

UNIX システム (Linux を含む) では、すべての外部シンボルが動的リンクされる可能性があるため、コンパイラはそれらすべてのシンボルに対して非効率な GOT 経由の間接参照を生成しなければなりません。この影響を軽減するために、UNIX モデルではセクション 15.9 で説明されている import-as-own モデルを採用しています。

ツールチェーンでは、extern シンボルのデフォルトの可視性を変更するオプションや宣言指定子など、プリエンプション ペナルティを軽減するベンダー固有の方法が追加で採用される場合があります。

DSBT モデルでは、エクスポートされる関数が DP を保存および復元する必要があるため、その分のオーバーヘッド (3 命令 + 2 メモリ アクセス) が発生します。また、テーブル自体にもデータ サイズのオーバーヘッドがあり、各実行可能ファイルとライブラリのデータ セグメントに N+1 ワードが追加されます (N はアプリケーションが使用するライブラリの最大インデックスです)。

Chapter 7

スレッドローカル ストレージの割り当てとアドレッシング

TEXAS INSTRUMENTS

C6000 プロセッサ ファミリを搭載する多くの組込みシステムでは、マルチスレッド プログラミングが一般的です。C6000 CPU ベースのマルチコア デバイスが増えているため、複数コアを有効活用する手段としてマルチスレッドの重要性は今後さらに高まります。また、OpenCL のようなマルチコア プログラミング モデルは、基盤となるマルチスレッド サポートに依存します。

7.1 マルチスレッドおよびスレッドローカル ストレージについて

スレッドが、静的ストレージ期間があり、かつそのスレッドに固有の変数を使用できる場合、複雑なマルチスレッド プログラムはより良い構造にでき、開発も容易になります。すなわち、他のスレッドからは、そのようなスレッド固有の静的ストレージ期間を持つ変数を参照したり、アクセスしたりすることはできません。次の C コードをご覧ください。

```
int global_x;
foo()  {
    int local_x;
    static int static_x = 0;
    ...
}
```

global_x 変数と static_x 変数は、プロセスごとに一度だけ割り当てられ、すべてのスレッドが同じインスタンスを共有されます。一方、local_x はスタックから割り当てられます。各スレッドはそれぞれ固有のスタックを持つため、変数 local_x はスレッド固有となり、static_x はスレッド固有ではありません。しかし、スレッド単位でグローバル / 静的変数を定義する簡単な方法は存在しません。POSIX のスレッド インターフェイスでは、pthread getspecific および pthread setspecific を使用して、スレッド固有の静的ストレージ変数を作成できます。ただし、このインターフェイスは使い勝手が良くありません。

この問題を解決するために、スレッドローカル ストレージ (TLS) は、静的ストレージ期間を持つスレッド固有変数を定義できるストレージのクラスとなっています。TLS 変数「thread-local」は、スレッドごとに 1 回だけインスタンス化されるグローバル / 静的変数です。

TLS に使用されるメモリは、プログラムの実行期間中ずっと静的に割り当てられます。各スレッドは、作成時点ロードされているすべての動的モジュールによって定義されたスレッドローカル変数 (たとえスレッド自身がそれを宣言したり使用したりしない場合でも) のインスタンスを すべて 個別に持ちます。スレッドが作成されると、その TLS ブロックは、基盤となる OS のスレッド サポート ライブラリによって割り当ておよび初期化されます。スレッドが完了し、同じプログラムの実行中に再び実行される場合、その TLS ブロックは再初期化されます。他のスレッドによってスレッドが中断またはブロックされ、ブロックの解除後に実行が再開した場合には、TLS 変数は再初期化されません。

TLS 変数へのアクセス方法は、OS または RTOS が各スレッドに対してスレッドローカル ストレージをどのように作成・管理するかによって異なります。Linux システムでは、複数の動的ライブラリおよび dlopen() を使用して実行時にロードされるライブラリに対する TLS の割り当てをサポートする必要があります。また、Linux システムでは、スレッドローカルがアクセスされたときに初めて TLS ストレージを割り当てる遅延が必要になる場合もあります。これには高度な TLS ストレージ管理が必要であり、その仕組みによってスレッドローカルへのアクセス方法も影響を受けます。一方、RTOS を含む静的実行可能ファイルでは、管理する TLS ブロックは 1 つだけで済み、アクセスは単純になります。

スレッドローカルの概念の概要の後、この文書では、スレッドローカルがソース コードでどのように指定され、ELF オブジェクト ファイル内でどのように表現されるかを説明します (セクション 7.4)。次に、C6x Linux、静的実行可能ファイル、ベアメタル動的リンクの TLS モデルにおいて、スレッドローカルがどのようにアクセスされるか (セクション 7.5)、さらにスレッドローカル変数への弱参照がどのように解決されるかを説明します。(セクション 7.6)。

C6000 TLS のメカニズムは、業界標準の規則に基づいており、たとえば Ulrich Drepper による論文『ELF Handling for Thread-Local Storage』で説明されているメカニズムを採用しています。

7.2 用語と概念

スレッドローカル変数はスレッド固有であり、静的ストレージ期間を持ちます。これらの変数は、初期化されている場合は .neardata セクションまたは .fardata セクションに割り当てられるグローバル変数や静的変数と同様に割り当てられ、初期化されていない場合は .bss に割り当てられる必要があります。グローバル変数と静的変数はプロセスごとに 1 つのコピーしか持ちませんが、スレッドローカル変数はスレッドごとに個別のインスタンスを必要とします。

スレッドは、プログラムがスレッドの生成を要求したときに、スレッド マネージャによって作成されます。たとえば、マルチスレッド アプリケーション内の並列領域では、OS スレッド ライブラリ呼び出しによりワーカ スレッドが生成され、これらのワーカ スレッドは並列領域の末尾でジョイン / マージされます。

スレッド生成中には、スレッドローカル変数用のストレージを割り当て、初期化する必要があります。これは、スレッドごとの TLS ストレージを初期化するために使用される初期化イメージが必要であることを意味します。スレッド ローカル ストレージが使用される場合、静的リンカの出力である静的リンク ユニットには、TLS 初期化イメージを含める必要があります。静的リンク ユニットは *モジュール* と呼ばれます。

1 つのモジュールに対する TLS 初期化イメージは *TLS イメージ*と呼ばれます。TLS はスレッド生成の一部としてスレッドごとに割り当てられ、TLS イメージからのデータによって初期化されます。1 つのモジュールからのスレッドローカル変数用にスレッドごとに割り当てられるメモリは、*TLS ブロック*と呼ばれます。

静的実行可能ファイル モデルでは、静的リンカは、開始アドレスからロードされ実行される実行可能ファイルを生成します。RTOS またはスレッド ライブラリは、実行可能ファイルの一部としてリンクされます。この場合、モジュールは 1 つのみであり、したがって TLS イメージと TLS ブロックもそれぞれ 1 つのみとなります。これにより、TLS へのアクセスが簡素化されます。メイン スレッドは通常、プログラム初期化時に作成されます。他のスレッドはスレッド ライブラリによって作成されます。メイン スレッドの TLS ブロックは、プログラム ローダによって割り当ておよび初期化される必要があります (セクション 14.2 を参照)。スレッド ライブラリは、それが作成するスレッドの TLS を割り当てて初期化する責任を負います。

C6x Linux システムでは、プログラム (プロセス) は、実行可能ファイルおよび 0 個以上の動的ライブラリという複数のモジュールをロードすることによって作成されます。各モジュールは TLS イメージを持つことができます。プログラムの TLS イメージは、すべてのモジュールの TLS イメージで構成されます。これは *TLS テンプレート*と呼ばれます。通常、実行可能ファイルとすべての依存モジュールは、プロセス起動時にロードされます。これらは、*初期ロード モジュール* と呼ばれます。Linux プログラムは、dlopen() システム関数を呼び出すことによって、起動後に動的ライブラリをロードすることもできます。起動後にロードされるモジュールは、*dlopen 済みモジュール* と呼ばれます。スレッド生成中には、TLS ブロックが TLS テンプレートに基づいて作成されます。すべてのモジュールの TLS ブロックからなるランタイム構造は、*TLS* と呼ばれます。

ベアメタル環境での動的リンクの場合、デフォルトでは初期ロード モジュールのみが存在し、それらを連続的に配置して TLS テンプレートを形成できます。

C6x プログラムのロードおよび動的リンクの詳細については、Chapter 14 を参照してください。

7.3 ユーザー インターフェイス

長年にわたり、C および C++ はスレッド ローカル変数を定義できるように拡張されてきました。

- Linux システム向けのコンパイラ (GCC、Sun、IBM、Intel) は、C/C++ 言語拡張として __thread ストレージ修飾子をサポートします。ただし、これは公式な言語拡張ではありません。
- Windows 向けのコンパイラ (MS VC++、Intel、Borland) は、__declspec(thread) ストレージ属性拡張をサポートしています。
- 最新の C++ 標準である C++11 (ISO/IEC 14882:2011) では、thread_local ストレージ クラス指定子が導入されています。
- 最新の C 標準である C11 では、_Thread_local ストレージ クラス指定子が導入されています。

スレッドローカル ストレージをサポートするために使用される言語拡張はツールチェーン固有であり、ABI の範囲外です。

スレッド ローカル変数は、初期化済みまたは未初期化として定義できます。未初期化のスレッド ローカル変数は、未初期化のグローバル変数や静的変数と同様にゼロに初期化されます。スレッド ローカル変数の割り当てと初期化は、静的であれ動的であれ、スレッドが作成されるときに行われます。

7.4 ELF オブジェクト ファイルの表現

ELF 仕様 (www.sco.com/developers/gabi/) には、スレッドローカル ストレージが ELF 再配置可能オブジェクト ファイルおよび ELF モジュールにおいてどのように表現されるかの詳細が規定されています。

ELF 仕様の関連部分の要点をまとめると、スレッドローカル変数は静的データとほぼ同じようにオブジェクト ファイルや ELF モジュールに格納されます。ただし、ELF では再配置可能ファイルの場合、これらのスレッドローカル変数を SHF_TLS フラグ付きのセクションに割り当てる必要があります。さらに、初期化済みのスレッドローカル ストレージに

は .tdata を、未初期化のものには .tbss をセクション名として用いることが規定されています。これらのセクションは読み取り/書き込みの権限を持ちます。

モジュールでは TLS セグメントに PT_TLS というセグメント タイプを持たせることが求められています。このセグメントは読み取り専用です。PT_TLS セグメントは TLS イメージです。

また、スレッドローカル シンボルは STT_TLS というシンボル タイプを持ちます。

7.5 TLS アクセス モデル

各スレッドは すべての スレッドローカル変数 (そのスレッドが宣言または使用しないものも含む) の自分自身のインスタンスを持ちます。スレッドローカル変数へのアクセスは、そのスレッドのインスタンスにアクセスする必要があります。これは、スレッドローカル アクセスが現在のスレッドの TLS を見つけ、その変数が定義されている TLS ブロック内のオフセットを使用して変数にアクセスする必要があることを意味します。

TLS データにアクセスする方法には 6 つのモデルがあります。これは、DLL/DSO がサポートされるか (別リンク)、dlopen() がサポートされるか、アクセスが自分自身のデータかインポートされたデータかといった要因に依存します。表 7-1 に、TLS アクセス モデルのさまざまな特性を示します。

表 7-1. スレッドローカル ストレージ アドレッシング モデル

モデル	一般動的	ローカル動的	初期 Exec	ローカル Exec	静的 Exec	ベアメタル動的
システムには DLL または DSO がある	あり	あり	あり	あり	なし	あり
別のモジュールの TLS データにアクセスできる	あり	なし	あり	なし	なし	あり
DLL または DSO からの TLS アクセス	あり	あり	あり	なし	なし	あり
dlopen されたモジュールからの TLS アクセス	あり	あり	なし	なし	なし	なし
TLS の初期化の実行	ローダ	ローダ	ローダ	ローダ	リンカ	ローダ
弱参照をサポートする	あり	あり	なし	なし	なし	なし
使用事例	別のモジュールの TLS データにアクセスする	DLL または DSO から独自の TLS データにアクセスする	ロード時にロードされたモジュールの TLS データにアクセスする	実行可能ファイルから独自の TLS データにアクセスする	DLL または DSO がない	ロード時にロードされたモジュールの TLS データにアクセスする
セクション	セクション 7.5.1.1	セクション 7.5.1.2	セクション 7.5.1.3	セクション 7.5.1.4	セクション 7.5.2	セクション 7.5.3

C6x Linux TLS アクセス モデルは、より多くの制約を満たす必要があり、複雑になる可能性もあります。これはすでに確立されている規則に準拠する必要があるためです。一方で、静的実行可能ファイルのアクセス モデルは単純です。TLS ブロックは 1 つのみであり、スレッドローカル変数にはスレッド ポインタ相対 (TPR) アドレッシングを使用してアクセスできます。説明の流れとしては、まず複雑な C6x Linux TLS モデル (セクション 7.5.1) を扱い、その後に単純な静的実行ファイル TLS モデル (セクション 7.5.2) を説明します。最後に、ベアメタル環境での動的リンクの場合 () について説明します。最後に、ベアメタル動的リンクのケース (セクション 7.5.3) を説明します。

文献上、広く利用されている TLS アクセス モデルは次の 4 種類とされています。[3] [4]

- 一般動的 TLS アクセス モデル (セクション 7.5.1.1)
- ローカル動的 TLS アクセス モデル (セクション 7.5.1.2)
- 初期 Exec TLS アクセス モデル (セクション 7.5.1.3)
- ローカル Exec TLS アクセス モデル (セクション 7.5.1.4)

[3] Ulrich Drepper, *ELF Handling for Thread-Local Storage*, http://www.uclibc.org/docs/tls.pdf, 2005, Version 0.20
[4] Alexandre Olivia and Glauber de Oliveira Costa, *Speeding Up Thread-Local Storage Access in Dynamic Libraries in the ARM Platform*, http://www.fsfla.org/~lxoliva/writeups/TLS/paper-lk2006.pdf, 2006.

TLS 用に使用される再配置の完全な一覧は表 13-5 および表 13-6 に記載されています。次のセクションでは、これらの再配置の使用方法を示します。

7.5.1 C6x Linux TLS モデル

Linux を含む一部の動的リンク モデルでは、dlopen () を使用して実行時にモジュールをロードできます。dlopen () でロードされたモジュールの TLS ブロックは動的に割り当てられるため、すべてのスレッドに対して TP からの固定オフセット位置に割り当てることはできません。そのため、スレッドローカル変数へのアクセスは、モジュール識別子と、そのモジュールの TLS ブロック内のスレッドローカル変数のオフセットを使用した参照によって行われます。

図 7-1 は、C6x Linux TLS のランタイム表現を示しています。各スレッドは、このランタイム TLS 構造体のインスタンスを持ちます。

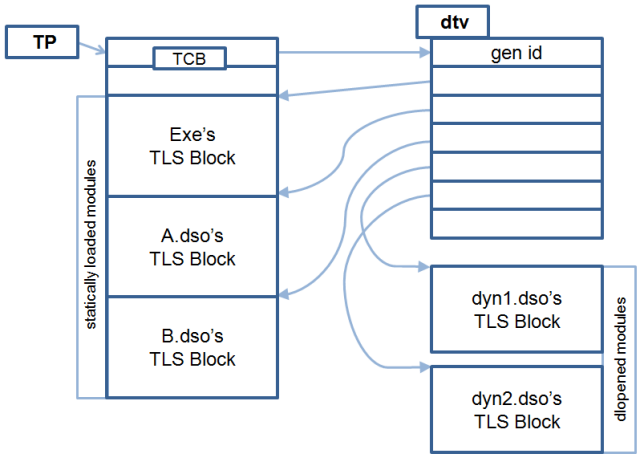


図 7-1. C6x Linux TLS ランタイム表現

スレッドごとに、スレッド ポインタ (TP) はスレッド コントロール ブロック (TCB) を指します。実行可能ファイルの TLS ブロックが存在する場合、アライメントを調整した上で TCB の後に配置されます。その他の非動的モジュールの TLS ブロックも、アライメント要件に従って順次配置されます。TCB と、それに続く静的モジュール用の TLS ブロックが、プログラムの静的 TLS を構成します。スレッドの静的 TLS は、スレッド作成に作成されます。

TCB は 64 ビット幅です。最初の 32 ビットは動的スレッド ベクタ (dtv) を指定します。残りの 32 ビットは予約されています。

TCB が指す dtv は 32 ビット要素のベクタです。dtv[0] 要素は世代 ID で、dlopen () でモジュールがロードされる際に、dtv の動的成長を管理するために使用されます。dtv[n] 要素 (n !=0) は、モジュール n の TLS ブロックへの 32 ビット ポインタです。TLS データを持つモジュールがロードされると、そのモジュールにモジュール ID が割り当てられます。このモジュール ID はプロセス固有です。複数のプロセスで共有される動的共有ライブラリでも、プロセスごとに異なるモジュール ID を持つことができます。モジュール ID 1 は常に実行可能ファイルに割り当てられます。

メイン スレッドは動的ローダによって作成され、後続のスレッドはスレッド ライブラリによって作成されます。メイン スレッドが作成されると、dtv 配列には最初にロードされたモジュールへのポインタのみを含める必要があります。

スレッドが新しいモジュールを開くと、そのモジュールの TLS ブロックがプロセス内のすべてのスレッドに割り当てられるはずです。これは、他のスレッドがこの新しいモジュールのスレッドローカル データにアクセスする場合に必要です。ただし、dlopen () でロードされたモジュールの TLS ブロックの割り当ては、最初にストレージにアクセスするまで延期できます。これは、適切な dtv[module-id] を TLS_DTV_UNALLOCATED に初期化することで実行できます。__tls_get_addr() 関数は、dtv[module_id] が TLS_DTV_UNALLOCATED であるかどうかをチェックし、そうである場合は、現在のスレッドの TLS ブロックを割り当てて初期化します。

7.5.1.1 一般動的 TLS アクセス モデル

これは最も汎用的な TLS アクセス モデルです。このアクセス モデルを使用するオブジェクトは、任意の Linux モジュール (実行可能ファイル、最初にロードされたモジュール、dlopen によってロードされたモジュール) を構築するために使用できます。このモデル用に生成されるコードは、静的リンク時に module-id またはオフセットが既知であるとは限りません。

このアクセス モデルを使用すると、動的モジュールを実行時にロードできます。これを実現するには、スレッド ライブラリのスレッド管理アーキテクチャが、動的モジュールのロードやアンロードに応じて TLS ブロックを追加・削除できる仕組みを備えている必要があります。

コンパイラは、スレッドローカル変数のアドレスを取得するために __tls_get_addr() を呼び出すコードを生成します。module-id とスレッドローカル変数のモジュール内 TLS ブロックにおけるオフセットがパラメータとして渡されます。コード

は、位置の独立性 (PIC) およびシンボルのプリエンプションを保証するために、グローバル オフセット テーブル (GOT) エントリからモジュール ID とオフセットを取得します。

__tls_get_addr() 関数が module-id とオフセットを渡す最も単純な方法は次のとおりです。

```
    void * __tls_get_addr(unsigned int module_id, ptrdiff_t offset);
```

どちらも 32 ビットの引数であり、GOT エントリも 32 ビットのエントリです。最適化として、ISA がこれをサポートしている場合、これら 2 つの GOT エントリを 64 ビット ダブル ワードとしてロードできます。2 つの GOT エントリは、連続して割り当てられ、64 ビット境界に整列される必要があります。この GOT エンティティは、次の構造とみなすことができます。

```
    struct TLS_descriptor
    {
        unsigned int module_id;
        ptrditt_t offset;
    } __attribute__ ((aligned (8)));
```

次に、__tls_get_addr() インターフェイスは以下のようになります。

```
    void * __tls_get_addr(struct TLS_descriptor);
```

この EABI では、サイズが 64 ビット以下の構造は値渡しされ、その結果、TLS ディスクリプタは A5:A4 レジスタ ペアで渡されます。リトル エンディアン モードでは、module-id が A4 に渡され、オフセットが A5 に渡されます。ビッグ エンディアン モードでは、C6x EABI 呼び出し規則に従ってレジスタが入れ替えられます。このセクションの例ではリトル エンディアン モードを使用します。

このインターフェイスを使用すると、スレッドローカル アクセスは次のようになります (C64 以上の場合)。

```
       LDDW   *+DP($GOT_TLS(X)), A5:A4  ;reloc R_C6000_SBR_GOT_U15_D_TLS
    || CALLP  __tls_get_addr,B3          ; A4 has the address of X at return
       LDW    *A4, A4                   ; A4 has the value of X
```

再配置 R_C6000_SBR_GOT_U15_D_TLS は、x の module-id とオフセットのための GOT エントリを次のようにリンカに作成させます。

```
64-bit aligned address:
        GOT[n]            ;reloc R_C6000_TLSMOD  (symbol X)
        GOT[n+1]          ;reloc R_C6000_TBR_U32 (symbol X)
```

リンカは、その後 R_C6000_SBR_GOT_U15_D_TLS 再配置を GOT エンティティの DP 相対オフセットで解決します。動的ローダは、R_C6000_TLSMOD を x が定義されているモジュールの module-id に解決します。また、R_C6000_TBR_U32 を、そのモジュールの TLS ブロックにおける x のオフセットに解決します。

C6x ISA には現在、64 ビット TLS ディスクリプタを直接ロードする命令はありません。しかし、将来の ISA がこのようなサポートを持つことを想定して、64 ビット ディスクリプタを使用する __tls_get_addr() インターフェイスが定義しされています。

```
    void * __tls_get_addr(struct TLS_descriptor);
```

R_C6000_SBR_GOT_U15_D_TLS 再配置が見つかった場合、リンカはスレッドローカル変数の module-id とオフセットの GOT エントリを連続して割り当て、最初のエントリを 64 ビット境界に整列させることが求められます。

DP 相対 64 ビット ロードをサポートしていないため、現行の ISA では次のシーケンスを使用できます。

```
       LDW    *+DP($GOT_TLSMOD(X)), A5   ;reloc R_C6000_SBR_GOT_U15_W_TLSMOD
       LDW    *+DP($GOT_TBR(X)), A4      ;reloc R_C6000_SBR_GOT_U15_W_TBR
    || CALLP  __tls_get_addr,B3          ; A4 has the address of X at return
       LDW    *A4, A4                    ; A4 has the value of X
```

再配置 R_C6000_SBR_GOT_U15_W_TLSMOD と再配置 R_C6000_SBR_GOT_U15_W_TBR は、それぞれ x の module-id とオフセットのための GOT エントリをリンカに作成させます。このアクセス モードでは、これらの GOT エントリが連続して 64 ビット境界に整列している必要はありません。リンカが同じシンボルに対して DW_TLS 再配置を同時に認識しない場合、module-id とオフセットの GOT エントリは 64 ビット整列を行わずに個別に定義できます。ただし、同じシンボルに対して TLSMOD/TBR 再配置に加えて DW_TLS が認識された場合、64 ビット整列された連続する GOT エントリを定義し、それを TLSMOD/TBR 再配置に再利用する必要があります。

GOT をファー DP アドレッシングで参照する必要がある場合、一般動的アドレッシングは次のようになります。

```
        MVKL $DPR_GOT_TLSMOD(X),  A5      ;reloc R_C6000_SBR_GOT_L16_W_TLSMOD
        MVKH $DPR_GOT_TLSMOD(X),  A5      ;reloc R_C6000_SBR_GOT_H16_W_TLSMOD
        ADD  DP, A5, A5
        LDW  *A5, A5
        MVKL $DPR_GOT_TPR(X),  A4         ;reloc R_C6000_SBR_GOT_L16_W_TBR
        MVKH $DPR_GOT_TPR(X),  A4         ;reloc R_C6000_SBR_GOT_H16_W_TBR
        ADD  DP, A4, A4
        LDW  *A4, A4
     || CALLP  __tls_get_addr,B3          ; A4 has the address of X at return
        LDW    *A4, A4                    ; A4 has the value of X
```

__tls_get_addr() はスレッドローカル アドレスを次のように計算できます。

```
    void * __tls_get_addr(struct TLS_descriptor desc)
    {
        void *TP  = __c6xabi_get_tp();
        int  *dtv = (int*)(((int*) TP)[0]);
        char *tls = (char *)dtv[desc.module_id];
        return tls + desc.offset;
    }
```

7.5.1.2 ローカル動的 TLS アクセス モデル

このアクセス モデルは、モジュール自身のデータにアクセスするための一般動的モデルの最適化です。コンパイラがモジュール自身のスレッドローカル ストレージにアクセスすることを認識している場合、このアクセス モデルを使用できます。スレッドローカル変数が、それがアクセスされるのと同じモジュール内で定義されている場合、TLS オフセットは静的リンク時に認識されます。ただし、module-id は静的リンク時には認識されません。

オフセット引数に 0 を指定して __tls_get_addr() を呼び出すと、そのモジュールの TLS ブロックのベース アドレスが返されます。このベース アドレスを使用すれば、そのモジュールに属するすべてのスレッドローカル データにアクセスできます。

コンパイル時には、スレッド自身のデータはシンボルのバインディングおよび可視性を使用して識別されます。静的範囲または非公開 / 保護の可視性を持つシンボルは、自身のデータです。このモデルでは、スレッドローカル変数 x は次のようにアクセスできます。

```
        LDW  *+DP($GOT_TLSMOD(x)), A4     ; reloc R_C6000_SBR_GOT_U15_W_TLSMOD
        MVK  $TBR_word(x), A5             ; reloc R_C6000_TBR_U15_W
     || CALLP    __tls_get_addr,B3   ; A4 has the address of x at return
```

前述のように、自身の TLS ベースは一度取得すれば、他の自身のスレッドローカル変数へのアクセスに再利用できます。

```
        LDW  *+DP($GOT_TLSMOD()), A4    ; reloc R_C6000_SBR_GOT_U15_W_TLSMOD w/ Symbol=0
        MVK  0x0, A5                    ;
     || CALLP    __tls_get_addr,B3 ; A4 has the module's own TLS base
        MVK  $TBR_byte(x), A5           ; reloc R_C6000_TBR_U15_B; Get x's scaled TLS offset
        LDB  *A4[A5], A6                ; A6 has the value of thread-local char x
        MVK  $TBR_hword(y), A5          ; reloc R_C6000_TBR_U15_H; Get y's scaled TLS offset
        LDH  *A4[A5], A6                ; A6 has the value of thread-local short y

        MVK  $TBR_word(z), A5           ; reloc R_C6000_TBR_U15_W; Get z's scaled TLS offset
        LDW  *A4[A5], A6                ; A6 has the value of thread-local int z
```

```
        MVK   $TBR_dword(l), A5          ; reloc R_C6000_TBR_U15_D; Get l's scaled TLS offset
        LDDW *A4[A5], A7:A6              ; A7:A6 has the value of thread-local long long l
```

再配置 R_C6000_SBR_GOT_U15_W_TLSMOD は、シンボルが 0 の場合、そのモジュール自身の module-id に解決されます。TBR_U15 再配置は、ニア TB (TLS ブロック ベース) アドレッシング用に、そのモジュールの TLS ベースからの 15 ビットの符号なしオフセットをエンコードします。これらはアクセス幅に応じてスケーリングされます。上述のアドレッシングは、32KB の TLS ブロックにアクセスできます。この仕様では、各モジュールの TLS ブロックのサイズが 32KB に制限されており、この制限はほとんどのユースケースで十分であると想定されています。そのため、ファー TB 相対アドレスは定義されていません。ファー TBR アドレッシングを定義することはできますが、それには新しい再配置を 8 個消費するため、ELF が許容する限られた数の再配置 (256) を無駄に消費しないことをお勧めします。

静的リンカは、すべての TBR 再配置を静的専用再配置を使用して解決します。つまり、これらの再配置は動的再配置テーブルに含めることはできません。

7.5.1.3 初期 Exec TLS アクセス モデル

最初にロードされたモジュールを構築するために使用されるオブジェクトでは、このアクセス モデルを使用できます。このアクセス モデルを使用するモジュールは dlopen できません。

モジュールは常に最初にロードされ、動的ローダは実行可能ファイルの TLS ブロックの後に連続して初期モジュールから TLS ブロックを割り当てることができるため、スレッド ポインタからのオフセットは動的リンク時に認識されます。スレッド ローカル変数には*(TP + offset) を使用してアクセスできます。この場合、PIC とシンボルのプリエンプションを確実にするために GOT からオフセットがロードされます。このタイプのアドレッシングで構築されたモジュールは dlopen できません。そのようなモジュールは動的フラグ DF_STATIC_TLS でマークされ、DF_STATIC_TLS とマークされた dlopen モジュールを動的ローダは拒否します。

7.5.1.3.1 スレッド ポインタ

初期 Exec モデルで使用されるアドレッシングには、現在のスレッドのスレッド ポインタを取得する方法が必要です。新しい c6xabi 関数 __c6xabi_get_tp() は、現在のスレッドのスレッド ポインタ値を返します。この関数では、復帰レジスタ A4 以外のレジスタは変更されません。この関数は PLT 経由で呼び出すことができるため、呼び出し元はこの関数の呼び出しによって B30 レジスタおよび B31 レジスタが変更されると想定する必要があります。この関数には以下のシグネチャがあります。

```
    void * __c6xabi_get_tp(void);
```

スレッド ライブラリは、この関数の定義を提供する責任を持ちます。

7.5.1.3.2 初期 Exec TLS アドレッシング

初期 Exec モデルでは、スレッドローカル変数は次のようにアクセスされます。

```
        callp  __c6xabi_get_tp()          ;Returns TP in A4; Can be CSEed
        LDW  *+DP($GOT_TPR_byte(x)), A5   ;reloc R_C6000_SBR_GOT_U15_W_TPR_B
        LDB    *A4[A5], B4                ;
        LDW  *+DP($GOT_TPR_hword(x)), A5  ;reloc R_C6000_SBR_GOT_U15_W_TPR_H
        LDH    *A4[A5], B4                ;
        LDW  *+DP($GOT_TPR_word(x)), A5   ;reloc R_C6000_SBR_GOT_U15_W_TPR_W
        LDW    *A4[A5], B4                ;
        LDW  *+DP($GOT_TPR_dword(x)), A5  ;reloc R_C6000_SBR_GOT_U15_W_TPR_D
        LDDW   *A4[A5], B4                ;
```

再配置 R_C6000_SBR_GOT_U15_W_TPR_[B|H|W] は、x の TPR オフセット用の GOT エントリをリンカに作成させます。

```
        GOT[m]          ;reloc R_C6000_TPR_U32_B  (symbol x)
        GOT[n]          ;reloc R_C6000_TPR_U32_H  (symbol y)
        GOT[o]          ;reloc R_C6000_TPR_U32_W  (symbol z)
        GOT[p]          ;reloc R_C6000_TPR_U32_D  (symbol z)
```

_TPR_U32_[B|H|W|DW] 再配置は、スレッドポインタからの x のオフセットを使用し、動的ローダによって解決されます。これらの再配置は、アクセス幅に従ってスケーリングされます。

GOT にファー DP アドレッシングを使用してアクセスする必要がある場合、シーケンスは次のようになります。

```
        callp  __c6xabi_get_tp()          ;Returns TP in A4; Can be CSEed
        MVKL   $DPR_GOT_TPR_byte(x), A5    ;reloc R_C6000_SBR_GOT_L16_W_TPR_B
        MVKH   $DPR_GOT_TPR_byte(x), A5    ;reloc R_C6000_SBR_GOT_H16_W_TPR_B
        ADD    DP, A5, A5
        LDW    *A5, A5
        LDB    *A4[A5], A6
        MVKL   $DPR_GOT_TPR_hword(x), A5   ;reloc R_C6000_SBR_GOT_L16_W_TPR_H
        MVKH   $DPR_GOT_TPR_hword(x), A5   ;reloc R_C6000_SBR_GOT_H16_W_TPR_H
        ADD    DP, A5, A5
        LDW    *A5, A5
        LDH    *A4[A5], A6
        MVKL   $DPR_GOT_TPR_word(x), A5    ;reloc R_C6000_SBR_GOT_L16_W_TPR_W
        MVKH   $DPR_GOT_TPR_word(x), A5    ;reloc R_C6000_SBR_GOT_H16_W_TPR_W
        ADD    DP, A5, A5
        LDW    *A5, A5
        LDW    *A4[A5], A6
        MVKL   $DPR_GOT_TPR_dword(x), A5    ;reloc R_C6000_SBR_GOT_L16_W_TPR_D
        MVKH   $DPR_GOT_TPR_dword(x), A5    ;reloc R_C6000_SBR_GOT_H16_W_TPR_D
        ADD    DP, A5, A5
        LDW    *A5, A5
        LDDW    *A4[A5], A6
```

7.5.1.4 ローカル Exec TLS アクセス モデル

これは初期 Exec モデルの最適化です。プログラムの初期 TLS イメージ (通常は静的 TLS イメージと呼ばれる) が作成されるとき、TLS ブロックは常にスレッド ポインタから既知のオフセットに配置されます。通常、これはスレッド制御ブロック (TCB) に TLS ブロック ベース オフセットを加えたものです。したがって、実行可能ファイル自身のスレッドローカル変数は、静的リンク時定数であるスレッド ポインタ相対オフセットを持ちます。この場合、スレッドローカル変数はインライン定数オフセットを使用してアクセスでき、GOT エントリは要ありません。このアクセス モデルを使用するオブジェクトは、動的ライブラリを構築するためには使用できません。

```
        CALLP  __c6xabi_get_tp()   ; Returns TP in A4. Can be CSEed.
        MVK    $TPR_byte(x), A5    ; reloc R_C6000_TPR_U15_B
        LDB    *A4[A5], A4         ; A4 contains the value of thread-local char x
        MVK    $TPR_hword(y), A5   ; reloc R_C6000_TPR_U15_H
        LDH    *A4[A5], A4         ; A4 contains the value of thread-local short y
        MVK    $TPR_word(z), A5    ; reloc R_C6000_TPR_U15_W
        LDW    *A4[A5], A4         ; A4 contains the value of thread-local int z
        MVK    $TPR_dword(l), A5   ; reloc R_C6000_TPR_U15_D
        LDDW   *A4[A5], A7:A6      ; A7:A6 contains the value of thread-local long long l
```

TPR_U15 再配置は、ニア TPR アドレッシングのために、15 ビット符号なし TPR オフセット (TP が指すアドレスからのオフセット) をエンコードします。これらはアクセス幅に応じてスケーリングされます。上述のアドレッシングは、32KB の TLS ブロックにアクセスできます。この仕様では、この制限はほとんどのユースケースで十分であると想定されているため、合計静的 TLS ブロックのサイズが 32KB に制限されています。そのため、ファー TPR アドレスは定義されていません。ファー TBR アドレッシングを定義することはできますが、定義するには新しい再配置を 8 個消費するため、ELF が許容する限られた数の再配置 (256) を無駄に消費しないことをお勧めします。

7.5.2 静的実行可能ファイル TLS モデル

静的実行可能ファイル TLS モデルは、C6x EABI に準拠したコンパイラで Quality of Implementation (QoI) の一部としてサポートされることがありますが、C6x EABI に準拠するための必須要件ではありません。

静的実行可能ファイルでは TLS ブロックは 1 つだけであり、各スレッドローカル変数の TLS オフセットは静的リンク時点に認識されます。スレッドローカル変数には *(TLS base + offset) という形でアクセスします。

図 7-2 には TLS の実行時レイアウトが示されています。TP は現在実行中のスレッドの TLS ブロックを指すスレッド ポインタです。x のオフセットは静的リンク時に認識されます。

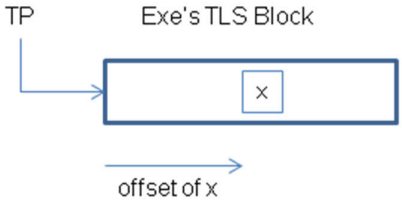


図 7-2. 静的実行可能ファイル TLS のランタイム表現

7.5.2.1 静的実行可能ファイル アドレッシング

静的実行可能ファイル TLS モデルの場合のスレッドローカル アクセス コードは、Linux Local Exec モデル (セクション 7.5.1.4) と同じです。静的実行可能ファイルの場合、スレッド コントロール ブロック (TCB) は存在しないため、TPR オフセットは TLS ブロック ベース相対アドレスと同じになります。

理想的には、この場合に TBR アドレッシングを生成できるのが理想的です。ただし、コンパイラ オプションを使用すればベアメタル動的リンク モデルを使用してビルドでき、その場合は TCB が必要になります。したがって、静的実行可能ファイル モデルでは TPR アドレッシングを次のように生成します。

```
        CALLP   __c6xabi_get_tp()   ; Returns TP in A4. Can be CSEed.
        MVK     $TPR_byte(x), B4    ; reloc R_C6000_TPR_U15_B
        LDB     *A4[B4], A4         ; A4 contains the value of thread-local char x
        MVK     $TPR_hword(y), B4   ; reloc R_C6000_TPR_U15_H
        LDH     *A4[B4], A4         ; A4 contains the value of thread-local short y
        MVK     $TPR_word(z), B4    ; reloc R_C6000_TPR_U15_W
        LDW     *A4[B4], A4         ; A4 contains the value of thread-local int z
        MVK     $TPR_dword(l), B4   ; reloc R_C6000_TPR_U15_D
        LDDW     *A4[B4], A7:A6     ; A4 contains the value of thread-local int l
```

TPR 再配置は、実行可能ファイルの TLS ブロック内における変数のオフセットを用いて、静的リンカによって解決されます。TLS ブロックが 32KB を超えると予想される場合には、ファー TPR 再配置を使用できます。

7.5.2.2 静的実行可能ファイルにおける TLS ランタイム アーキテクチャ

動的リンク システムでは、動的ローダがメイン スレッドを作成し、スレッド ライブラリが追加のスレッドを作成します。メイン スレッド作成の一環として、動的ローダはメイン スレッドの TLS を割り当て、初期化します。また、動的ローダは、セグメント タイプを使用して TLS 初期化イメージを簡単に見つけることができます。

静的実行可能ファイルの場合、これらの役割を実行する動的ローダは存在しません。静的リンク モデルは、次の要件をサポートしている必要があります。

- メイン スレッドの TLS のアロケーションと初期化は、main() や init_array からのユーザー コードが呼び出される前に行われる。
- メイン スレッドの実行中、__c6xabi_get_tp() はメイン スレッドの TLS へのポインタを返す必要がある。この関数はスレッド ライブラリがない場合でもサポートされている必要がある。
- スレッド ライブラリは、作成するスレッドの TLS ブロックを初期化できるよう、TLS 初期化イメージにアクセスする手段を持っている必要がある。

セクション 7.5.2.3.1 ～セクション 7.5.2.5 には、ツールチェーン固有の情報が含まれています。ツールチェーンが TLS モデルを実装する方法の例として、.TI.tls_init および .TI.tls セクション、__TI_tls_init_table コピー テーブル、__TI_TLS_MAIN_THREAD_BASE と __TI_TLS_BLOCK_SIZE シンボル、および __TI_tls_init() 関数に関する記述が含まれています。

7.5.2.3 静的実行可能ファイルにおける TLS のアロケーション

静的実行可能モデルで TLS をサポートするには、初期化イメージ、メイン スレッドの TLS ブロック、スレッド ライブラリが作成するスレッドに TLS ブロックを割り当てる TLS 領域の 3 つのメモリ領域を割り当てる必要があります。

7.5.2.3.1 TLS 初期化イメージの割り当て

TLS 初期化イメージは出力セクション .TI.tls_init に作成されます。このセクションは読み取り専用です。ユーザーはこの出力セクションの割り当てを次のように指定できます。

```
        .TI.tls_init > ROM
```

割り当てが指定されない場合、この出力セクションは .cinit の割り当てを使用して割り当てられます。.cinit に割り当てが指定されていない場合、デフォルトの割り当てが使用されます。ユーザーはこのセクションに対してセクション指定子を指定できません。

.TI.tls_init 出力セグメントは、リンカによって作成された以下のコンポーネントを結合することで構成されます。

- **.tdata.load** - 圧縮された TLS 初期化済みセクション
- **.tbss.load** - 未初期化セクションをゼロ初期化するためのゼロ初期化セクション
- **__TI_tls_init_table** - TLS ブロックを初期化するためのコピー テーブル。このコピー テーブルには 2 つのコピー レコードがあり、これらの初期化セクションごとに 1 つずつ存在します。

7.5.2.3.2 メイン スレッドの TLS 割り当て

ユーザーは次のコマンドを使用して、メイン スレッドの TLS ブロックの割り当てを指定できます。

```
        .TI.tls > RAM
```

この初期化されていない出力セクションは、ブート時に __TI_tls_init_table コピー テーブルを使用して初期化されます。ユーザーは、このセクションのセクション指定子を指定できません。

このセクションに割り当てが指定されていない場合は、.fardata 出力セクションの割り当てが使用されます。.fardata に割り当てが指定されていない場合は、.far の割り当てが使用されます。指定されている場合は、デフォルトの割り当てが使用されます。

リンカは、.TI.tls 出力セクションの先頭を指すための、シンボル __TI_TLS_MAIN_THREAD_BASE を定義します。

7.5.2.3.3 スレッド ライブラリの TLS 領域のアロケーション

スレッド ライブラリで使用される TLS 領域のアロケーションはライブラリに固有です。この仕様は、これを行う特定の方法を規定していません。TLS 領域を割り当てる 1 つの方法として、次の手順が考えられます。

```
        .tls_region { . += 0x2000; } START(TLS_REGION_START) > RAM
```

スレッド ライブラリは、シンボル TLS_REGION_START を使用して TLS 領域を特定できます。ユーザーが N 個のスレッドに TLS ブロックを割り当てる場合、TLS ブロックのサイズを把握しておくと便利です。ユーザーは次のように実行できます。

```
        .tls_region { . += MAX_THREADS * __TI_TLS_BLOCK_SIZE; } > RAM
```

静的リンカはシンボル __TI_TLS_BLOCK_SIZE を定義し、TLS ブロックのサイズに設定します。

7.5.2.4 静的実行可能ファイルにおける TLS の初期化

静的実行可能モデルで TLS をサポートするには、メイン スレッドの TLS ブロック、スレッズ ライブラリが作成するスレッドに TLS ブロックを割り当てる TLS 領域の 2 つのメモリ領域を割り当てる必要があります。

7.5.2.4.1 メイン スレッドの TLS 初期化

ブート時に、スタートアップ コードはランタイム サポート (RTS) 関数 __TI_tls_init(NULL) を呼び出して、メイン スレッドの TLS ブロックを初期化します。RTS 関数は、NULL 引数が渡された場合にメイン スレッドの TLS を初期化します。

7.5.2.4.2 TLS のスレッド ライブラリによる初期化

スレッド ライブラリは、特定のスレッド用に TLS ブロックを生成した後、それらを初期化しなければなりません。静的実行可能ファイル TLS モデルは、このために次の新しい RTS 関数を定義します。

```
__TI_tls_init(void * dest_addr);
```

スレッド ライブラリは、この関数に初期化される TLS ブロックのアドレスを渡す必要があります。

この RTS 関数はコピー テーブルを使用して初期化を実行します。ただし、この関数が TLS ブロックを初期化する方法は、静的リンカとこの RTS 関数とのインターフェイスに基づいており、今後変更される可能性があります。したがって、スレッド ライブラリは TLS ブロックを初期化するためのインターフェイスとして、この RTS 関数のみを使用する必要があります。

7.5.2.5 スレッド ポインタ

静的実行可能ファイル TLS モデルでは、現在のスレッドのスレッド ポインタ値を取得するために関数 __c6xabi_get_tp() が呼び出されます。スレッド ライブラリが使用される場合、そのライブラリがこの関数を提供する責任を持ちます。

スレッド ライブラリは、自身が作成するスレッドの TLS ブロックのアドレスを把握しています。ただし、メイン スレッドはスレッド ライブラリによって作成されないため、スレッド ライブラリはメイン スレッドの TLS ブロックのアドレスを見つける標準的な方法を必要とします。前述のように、この目的のために静的リンカはシンボル __TI_TLS MAIN_THREAD_BASE を定義します。

TI RTS は、__c6xabi_get_tp() 関数に対して次の定義を提供します。

```
extern __attribute__((weak)) far const void * __TI_TLS_MAIN_THREAD_Base;
__attribute__((weak)) void * __c6xabi_get_tp(void)
{
    return &__TI_TLS_MAIN_THREAD_Base;
}
```

この関数は「弱」として定義されており、スレッド ライブラリに強定義が存在する場合はそれが使用されます。

ユーザーがスレッドローカル変数を宣言したものの、スレッド ライブラリを含めないという珍しいケースについて考えてみましょう。当然ながら、新しいスレッドは作成できません。しかし、メイン スレッドは動作し、メイン スレッドのスレッドローカル変数にはアクセスできるはずです。このような場合、前述の RTS 関数がリンクされ、メイン スレッドの TLS へのアクセスを提供します。

7.5.3 ベアメタル動的リンク TLS モデル

ベアメタルの動的リンクには、最初にロードされたモジュールのみが含まれます。dlopen でロードされるモジュールは、現在ベアメタル動的リンクではサポートされていません。静的実行可能ファイル用にコンパイルされたオブジェクトは、ベアメタル動的実行可能ファイルまたはライブラリを作成するために使用できます。

7.5.3.1 ベアメタル動的リンクにおけるデフォルト TLS アドレッシング

TLS のデフォルトのコード生成は、静的実行可能ファイルとベアメタル動的リンクの両方で動作しなければなりません。静的実行可能ファイルの場合、TPR アドレッシングを使用して次のようなアドレッシングを生成します。

```
        CALLP   __c6xabi_get_tp()   ; Returns TP in A4. Can be CSEed.
        MVK     $TPR_byte(x), B4    ; reloc R_C6000_TPR_U15_B
        LDB     *A4[B4], A4         ; A4 contains the value of thread-local char x
```

ベアメタル動的リンクに対してデフォルトで生成されるコードは、すべてのモジュールが初期にロードされると想定できます。これは、スレッドローカル変数のオフセットが、図 7-3 に示すように動的リンク時定数であることを意味します。したがって、TPR アドレッシングを使用できます。唯一の違いは、ベアメタル動的リンクの場合、将来の dlopen() のサポートとの互換性を確保するために 64 ビットの TCB が必要であるという点です。静的実行可能ファイルの場合、TCB は存在しません。TPR アドレッシングは引き続き両方のモデルで使用できます。静的リンカは、静的実行可能ファイルに対しては TCB サイズを 0 とし、ベアメタル動的リンクに対しては 64 ビットの TCB サイズを使用します。

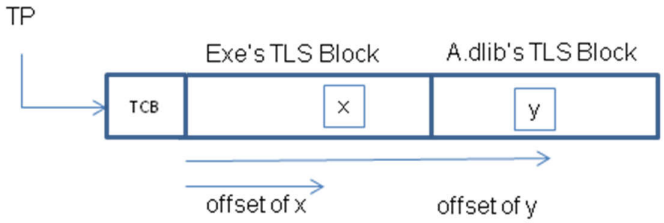


図 7-3. ベアメタルにおけるデフォルト TLS ランタイム表現

前述のように、最初にロードされたモジュールは連続して配置され、実行可能ファイルの TLS ブロックは TCB の後に配置されます。この場合、実行可能ファイル内の変数は、TP からの静的リンク時定数オフセットを使用してアクセスできます。動的ライブラリで定義された変数は、TP からの動的リンク時定数オフセットを用いてアクセスできます。

このアドレッシングが生成されると、モジュールは DF_STATIC_TLS としてマークされます。

動的実行可能ファイルを構築する際、静的リンカは、実行可能ファイル内で定義されたシンボル (独自のデータ) の TPR 再配置を TP オフセットに解決します。シンボルがインポートされている場合、その再配置は動的ローダによって解決されるように、動的再配置テーブルにコピーされます。動的ライブラリを構築する際には、TPR 再配置は動的再配置テーブルにコピーされます。

ベアメタルにおけるスレッドローカル アクセスにより、コード セグメント内で動的再配置が発生する可能性があります。これは、生成されたモジュールが真に PIC (位置独立コード) ではないことを意味します。TI コンパイラは --gen_pic オプションによりベアメタル PIC をサポートしています。このオプションが使用される場合、位置独立コードを生成するために、GOT エントリから TPR オフセットにアクセスする必要があります。

7.5.3.2 TLS ブロックの作成

ベアメタル動的リンク システムの場合、動的ローダがメイン スレッドの TLS ブロックの作成を担当します。動的ローダは ELF ファイルをロードする際に PT_TLS セグメントをロードし、スレッド ライブラリが PT_TLS 初期化イメージにアクセスできる方法を提供しなければなりません。これにより、スレッド ライブラリはその情報を利用して、新しく生成するスレッドの TLS ブロックを正しく初期化できます。静的リンカは、動的実行可能ファイル / ライブラリを構築する際に、ELF の要件に従って PT_TLS セグメントを生成します。

各動的モジュール (実行可能ファイル、共有オブジェクト、動的ライブラリ) は独自の TLS ブロックを取得します。PT_TLS セグメントには、指定されたモジュール内で定義された TLS オブジェクトの初期値が含まれます。

7.6 スレッドローカル シンボルの解決と弱参照

スレッドローカル参照はスレッドローカル定義によってのみ解決できます。リンカはこの要件を強制的に適用する必要があります。また、同じ名前を持つスレッドローカル定義と通常のグローバル定義が同時に存在するのはエラーとなります。

スレッドローカル変数は弱定義または宣言とすることができます。弱スレッドローカル定義は、利用可能な場合は強定義によって無効にされる場合があることを意味します。強定義が見つからない場合は、弱定義が使用されます。スレッドローカルの弱定義をサポートするために特別な配慮は必要ありません。

弱スレッドローカル シンボル参照は、定義が見つからない場合にはゼロ アドレスに解決されます。これには TLS アドレッシング モデルごとに特別な処理が必要です。

7.6.1 一般およびローカルの動的 TLS の弱参照アドレッシング

一般動的 TLS モデルとローカル動的 TLS モデルの両方で、スレッドローカル変数のアドレスを取得するために、関数 __tls_get_addr() が呼び出されます。module-id は、一般動的 TLS モデルとローカル動的 TLS モデルとも GOT から取得されます。オフセットは、一般動的モデルでは GOT から取得され、ローカル動的モデルでは静的リンク時定数として取得されます。弱未定義参照の場合、弱参照を解決するスレッドローカル定義が存在しません。定義が存在しないため、module-id と TBR オフセットは 0 に解決されます。

弱スレッドローカル参照については、参照をアクセスするために生成されるコードに変更はありません。R_C6000_TLSMOD 再配置およびすべての R_C6000_TBR 再配置は、スレッドローカル参照が弱で、かつ定義が存在しない場合に 0 に解決されます。

__tls_get_addr() 関数は、module-id とオフセットが 0 であるとき 0 を返します。これにより、未定義の弱参照アドレスが 0 に解決されます。

7.6.2 初期およびローカル実行可能 TLS の弱参照アドレッシング

スレッドポインタ相対アドレッシングは弱参照には使用できません。これは、シンボルが未定義の場合に 0 アドレスを生成する方法がないためです。したがって、初期実行可能アクセス モデルでは、弱参照に対して一般道的アドレッシングを使用する必要があります。同様に、ローカル実行可能アクセス モデルは、弱参照に対してローカル動的アドレッシングを使用する必要があります。

7.6.3 静的実行およびベア メタル動的 TLS モデルにおける弱参照

スレッド ポインタ相対アドレッシングは、シンボルが未定義の場合にゼロ アドレスを生成する方法がないため、弱参照には使用できません。したがって、静的実行可能形式およびベア メタル動的リンク アクセス モデルの弱参照には、ローカル動的フォームを使用する必要があります。

静的リンクおよびベア メタル動的リンクでは、弱参照に対して次のアドレッシングが生成されます。

```
        MVK   $TPR_S16(x), A5              ; reloc R_C6000_TPR_S16
          ||   CALLP    __c6xabi_get_addr,B3 ; A4 has the address of x at return
```

C6x eabi 関数 __c6xabi_get_addr() には次のシグネチャがあります。

```
    void * __c6xabi_get_addr(ptrdiff_t TPR_offst);
```

この関数は 32 ビットの TPR オフセットを受け取り、スレッド ローカル変数のアドレスを返します。TPR オフセットの特別の値 -1 は、弱い未定義参照を示すために使用されます。この場合、0 が返されます。

静的リンカと動的リンカは、弱い未定義参照に対して TPR_S16 再配置を -1 に解決します。

Chapter 8

ヘルパー関数 API

TEXAS INSTRUMENTS

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。

8.1 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。

C6000 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)。

8.2 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 __C6000_ 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。さらに、スレッドローカル ストレージへの動的リンク アクセスをサポートするには、__tls_get_addr() ヘルパー関数が必要です。

ヘルパー関数は、セクション 8.3 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 8-1 の関数は、C の変換規則とセクション 8.1 で指定された浮動小数点の動作に従って、浮動小数点値を整数値に変換します。

表 8-1. C6000 浮動小数点から整数への変換

シグネチャ	説明
int32 _ _**C6000_fixdi**(float64 x);	float64 を int32 に変換します
int40 _ _**C6000_fixdli**(float64 x);	float64 を int40 に変換します
int64 _ _**C6000_fixdlli**(float64 x);	float64 を int64 に変換します
uint32 _ _**C6000_fixdu**(float64 x);	float64 を uint32 に変換します
uint40 _ _**C6000_fixdul**(float64 x);	float64 を uint40 に変換します
uint64 _ _**C6000_fixdull**(float64 x);	float64 を uint64 に変換します
int32 _ _**C6000_fixfi**(float32 x);	float32 を int32 に変換します
int40 _ _**C6000_fixfli**(float32 x);	float32 を int40 に変換します
int64 _ _**C6000_fixflli**(float32 x);	float32 を int64 に変換します
uint32 _ _**C6000_fixfu**(float32 x);	float32 を uint32 に変換します
uint40 _ _**C6000_fixful**(float32 x);	単精度浮動小数点数を uint40 に変換します
uint64 _ _**C6000_fixfull**(float32 x);	単精度浮動小数点数を uint64 に変換します

表 8-2 の関数は、C の変換規則とセクション 8.1 で指定された浮動小数点の動作に従って、整数値を浮動小数点値に変換します。

表 8-2. C6000 整数から浮動小数点への変換

シグネチャ	説明
float64 _ _**C6000_fltid**(int32 x);	int32 を倍精度浮動小数点に変換します
float64 _ _**C6000_fltlid**(int40 x);	int40 を倍精度浮動小数点に変換します
float64 _ _**C6000_fltllid**(int64 x);	int64 を倍精度浮動小数点に変換します
float64 _ _**C6000_fltud**(uint32 x);	uint32 を倍精度浮動小数点に変換します
float64 _ _**C6000_fltuld**(uint40 x);	uint40 を倍精度浮動小数点に変換します
float64 _ _**C6000_fltulld**(uint64 x);	uint64 を倍精度浮動小数点に変換します
float32 _ _**C6000_fltif**(int32 x);	int32 を単精度浮動小数点に変換します
float32 _ _**C6000_fltlif**(int40 x);	int40 を単精度浮動小数点に変換します
float32 _ _**C6000_fltllif**(int64 x);	int64 を単精度浮動小数点に変換します
float32 _ _**C6000_fltuf**(uint32 x);	uint32 を単精度浮動小数点に変換します
float32 _ _**C6000_fltulf**(uint40 x);	uint40 を単精度浮動小数点に変換します
float32 _ _**C6000_fltullf**(uint64 x);	uint64 を単精度浮動小数点に変換します

表 8-3 の関数は、C の変換規則とセクション 8.1 で指定された浮動小数点の動作に従って、浮動小数点値を別のフォーマットに変換します。

表 8-3. C6000 浮動小数点フォーマット変換

シグネチャ	説明
float32 _ _**C6000_cvtdf**(float64 x);	倍精度浮動小数点を単精度浮動小数点に変換します
float64 _ _**C6000_cvtfd**(float32 x);	単精度浮動小数点を倍精度浮動小数点に変換します

表 8-4 の関数は、C 言語のセマンティクスと セクション 8.1 で指定された浮動小数点の動作に従って、浮動小数点演算を行います。

表 8-4. C6000 浮動小数点演算

シグネチャ	説明
float64 _ _**C6000_absd**(float64 x);	倍精度浮動小数点数の絶対値を返します
float32 _ _**C6000_absf**(float32 x);	単精度浮動小数点数の絶対値を返します
float64 _ _**C6000_addd**(float64 x, float64 y);	2 つの倍精度浮動小数点数を加算します (x+y)
float32 _ _**C6000_addf**(float32 x, float32 y);	2 つの単精度浮動小数点数を加算します (x+y)
float64 _ _**C6000_divd**(float64 x, float64 y);	2 つの倍精度浮動小数点数を除算します (x/y)
float32 _ _**C6000_divf**(float32 x, float32 y);	2 つの単精度浮動小数点数を除算します (x/y)
float64 _ _**C6000_mpyd**(float64 x, float64 y);	2 つの倍精度浮動小数点数を乗算します (x*y)
float32 _ _**C6000_mpyf**(float32 x, float32 y);	2 つの単精度浮動小数点数を乗算します (x*y)
float64 _ _**C6000_negd**(float64 x);	符号を反転した倍精度浮動小数点数 (-x) を返します
float32 _ _**C6000_negf**(float32 x);	符号を反転した単精度浮動小数点数 (-x) を返します
float64 _ _**C6000_subd**(float64 x, float64 y);	2 つの倍精度浮動小数点数の差 (x-y) を計算します
float32 _ _**C6000_subf**(float32 x, float32 y);	2 つの単精度浮動小数点数の差 (x-y) を計算します
int64 _ _**C6000_trunc**(float64 x);	倍精度浮動小数点数を 0 に向かって切り捨てます
int32 _ _**C6000_truncf**(float32 x);	単精度浮動小数点数を 0 に向かって切り捨てます

表 8-5 の関数は、C 言語のセマンティクスと セクション 8.1 で指定された浮動小数点の動作に従って、浮動小数点比較を行います。

_ _C6000_cmp* 関数は、x が y より小さい場合は 0 未満、値が等しい場合は 0、x が y より大きい場合は 0 より大きい整数を返します。いずれかのオペランドが NaN の場合、結果は未定義です。

明示的比較関数は、順序付けられていない (NaN) オペランドでも正しく動作します。つまり、オペランドの 1 つが NaN であっても比較が true であれば 0 以外を返し、それ以外の場合は 0 を返します。

表 8-5. 浮動小数点の比較

シグネチャ	説明
int32 _ _**C6000_cmpd**(float64 x, float64 y);	倍精度浮動小数点の比較
int32 _ _**C6000_cmpf**(float32 x, float32 y);	単精度浮動小数点の比較
int32 _ _**C6000_unordd**(float64 x, float64 y);	倍精度浮動小数点数における順序付けられていないオペランドのチェック
int32 _ _**C6000_unordf**(float32 x, float32 y);	単精度浮動小数点数における順序付けられていないオペランドのチェック
int32 _ _**C6000_eqd**(float64 x, float64 y);	倍精度浮動小数点の比較:x == y
int32 _ _**C6000_eqf**(float32 x, float32 y);	単精度浮動小数点数の比較:x == y
int32 _ _**C6000_neqd**(float64 x, float64 y);	倍精度浮動小数点の比較:x != y
int32 _ _**C6000_neqf**(float32 x, float32 y);	単精度浮動小数点数の比較:x != y
int32 _ _**C6000_ltd**(float64 x, float64 y);	倍精度浮動小数点の比較:x < y
int32 _ _**C6000_ltf**(float32 x, float32 y);	単精度浮動小数点の比較:x < y
int32 _ _**C6000_gtd**(float64 x, float64 y);	倍精度浮動小数点の比較:x > y
int32 _ _**C6000_gtf**(float32 x, float32 y);	単精度浮動小数点の比較:x > y
int32 _ _**C6000_led**(float64 x, float64 y);	倍精度浮動小数点の比較:x <= y
int32 _ _**C6000_lef**(float32 x, float32 y);	単精度浮動小数点の比較:x <= y
int32 _ _**C6000_ged**(float64 x, float64 y);	倍精度浮動小数点の比較:x >= y
int32 _ _**C6000_gef**(float32 x, float32 y);	単精度浮動小数点の比較:x >= y

表 8-6 の整数除算関数と整数剰余関数は、C 言語のセマンティクスに従って動作します。_ _C6000_divremi および _ _C6000_divremu 関数は、商 (x/y) と剰余 (x%y) の両方を計算し、商を A4 に、剰余を A5 に返します。_ _C6000_divremll および _ _C6000_divremull 関数は、64 ビット整数の商 (x/y) と剰余 (x%y) を計算し、商を A5:A4 に、剰余を B5:B4 に返します。

表 8-6. C6000 整数の除算と剰余

シグネチャ	説明
int32 _ _**C6000_divi**(int32 x, int32 y);	32 ビット符号付き整数の除算 (x/y)
int40 _ _**C6000_divli**(int40 x, int40 y);	40 ビット符号付き整数の除算 (x/y)
int64 _ _**C6000_divlli**(int64 x, int64 y);	64 ビット符号付き整数の除算 (x/y)
uint32 _ _**C6000_divu**(uint32 x, uint32 y);	32 ビット符号なし整数の除算 (x/y)
uint40 _ _**C6000_divlu**(uint40 x, uint40 y);	40 ビット符号なし整数の除算 (x/y)
uint64 _ _**C6000_divllu**(uint64 x, uint64 y);	64 ビット符号なし整数の除算 (x/y)
int32 _ _**C6000_remi**(int32 x, int32 y);	32 ビット符号付き整数の剰余 (x%y)
int40 _ _**C6000_remli**(int40 x, int40 y);	40 ビット符号付き整数の剰余 (x%y)
int64 _ _**C6000_remlli**(int64x. int64 y);	64 ビット符号付き整数の剰余 (x%y)
uint32 _ _**C6000_remu**(uint32 x, uint32 y);	32 ビット符号なし整数の剰余 (x%y)
uint40 _ _**C6000_remul**(uint40, uint40);	40 ビット符号なし整数の剰余 (x%y)
uint64 _ _**C6000_remull**(uint64, uint64);	64 ビット符号なし整数の剰余 (x%y)
_ _**C6000_divremi**(int32 x, int32 y);	32 ビット整数の除算と剰余の同時計算
_ _**C6000_divremu**(uint32 x, uint32 y);	32 ビット符号なし整数の除算と剰余の同時計算
_ _**C6000_divremull**(uint64 x, uint64 y);	64 ビット符号なし整数の除算と剰余の同時計算

表 8-7 の広域整数演算関数は、C 言語のセマンティクスに従って動作します。

表 8-7. C6000 広域整数演算

シグネチャ	説明
int64 _ _**C6000_negll**(int64 x);	64 ビット整数の符号を反転します
uint64 _ _**C6000_mpyll**(uint64 x, uint64 y);	64x64 ビット乗算
int64 _ _**C6000_mpyiill**(int32 x, int32 y);	32×32 ビット乗算
uint64 _ _**C6000_mpyuiill**(uint32 x, uint32 y);	32×32 ビット符号なし乗算
int64 _ _**C6000_llshr**(int64 x, uint32 y);	64 ビット符号付き右シフト (x>>y)
uint64 _ _**C6000_llshru**(uint64 x, uint32 y);	64 ビット符号なし右シフト (x>>y)
uint64 _ _**C6000_llshl**(uint64 x, uint32 y);	64 ビット左シフト (x<<y)

表 8-8 のその他のヘルパー関数については、以降のセクションで説明します。

表 8-8. C6000 その他のヘルパー関数

シグネチャ	説明
void _ _**C6000_strasgi**(int32 *dst, const int32 *src, uint32 cnt);	割り込みセーフブロックコピー。cnt >= 28
void _ _**C6000_strasgi_64plus**(int32*, const inst32*, uint32) ;	割り込みセーフブロックコピー。cnt >= 28
void _ _**C6000_abort_msg**(const char *string);	失敗したアサーションを報告
void _ _**C6000_push_rts**(void);	呼び出し先保存レジスタをすべてプッシュします
void _ _**C6000_pop_rts**(void);	呼び出し先保存レジスタをすべてポップします
void _ _**C6000_call_stub**(void);	呼び出し元保存レジスタを保存します。B31 を呼び出し
void _ _**C6000_weak_return**(void);	インポートされた弱い呼び出しの解決ターゲット
void _ _**C6000_get_addr**(ptrdiff_t TPR_offst);	スレッドポインタ レジスタ (TPR) のオフセットのアドレスを取得します。
void _ _**C6000_get_tp**(void);	現在のスレッドのスレッド ポインタ値を取得します。

表 8-8. C6000 その他のヘルパー関数 (続き)

シグネチャ	説明
void * _ _**tls_get_addr**(struct TLS_descriptor);	スレッドローカル変数のアドレスを取得します。

_ _C6000_strasgi

_ _C6000_strasgi 関数は、アウトオブラインの構造体や配列のコピー処理を効率的に行うためにコンパイラによって生成されます。cnt 引数はバイト単位のサイズで、28 (7 ワード) 以上の 4 の倍数である必要があります。次のような前提条件があります。

- src と dst のアドレスはワード境界に揃っていること。
- ソースと宛先のオブジェクトが重なっていないこと。

7 ワードの最小値は、C64x+ でソフトウェアパイプライン ループを使用できるようにするスレッショルドです。小さいオブジェクトの場合、コンパイラは通常、ロード/ストア命令のインライン シーケンスを生成します。_ _C6000_strasgi は割り込みを無効にしないため、安全に割り込み可能です。

_ _C6000_strasgi_64plus 関数は、C64x+ アーキテクチャ向け最適化された _ _C6000_strasgi のバージョンです。

_ _C6000_abort_msg

_ _C6000_abort_msg 関数は、実行時アサーション (例:C 言語の assert マクロ) が失敗したときに診断メッセージを出力するために生成されます。この関数は戻ってはいけません。つまり、abort を呼び出すか、他の方法でプログラムを終了させる必要があります。

_ _C6000_push_rts and _ _C6000_pop_rts

_ _c6x_push_rts 関数は、コード サイズを最適化する際に C64x+ アーキテクチャで使用されます。多くの関数は、呼び出し先保存レジスタのほとんどまたはすべてを保存および復元します。このような各関数のプロローグでの保存コードとエピローグでの復元コードの重複を回避するために、コンパイラは代わりにこのライブラリ関数を使用できます。この関数は、セクション 4.4.4 のプロトコルに従って、13 個すべての呼び出し先保存レジスタをスタックにプッシュし、SP を 56 バイト減算します。

_ _c6x_push_rts 関数は、次のように実装されています。

```
__c6xabi_push_rts:
        STW       B14, *B15--[2]
        STDW  A15:A14, *B15--
        STDW  B13:B12, *B15--
        STDW  A13:A12, *B15--
        STDW  B11:B10, *B15--
        STDW  A11:A10, *B15--
        STDW  B3:B2,   *B15--
        B     A3
```

(これは連続した、スケジュールされていない表現です。実際の実装については、TI のランタイム ライブラリのソース コードを参照してください)

_ _C6000_pop_rts 関数は、_ _C6000_push_rts によってプッシュされた呼び出し先保存レジスタを復元し、スタックを 56 バイト分インクリメント (ポップ) します。

_ _C6000_call_stub

_ _C6000_call_stub 関数も、c64x+ 関数のコードサイズを最適化するために使用されます。多くの呼び出し元にには、呼び出し全体にわたって有効な呼び出し先保存レジスタが複数あります。これらのレジスタは呼び出し先によって保存されないため、呼び出し元が保存して復元する必要があります。コンパイラは、呼び出しを _ _C6000_call_stub を経由してルーティングできます。この関数は次の手順を実行します。

- 選択された呼び出し元レジスタをスタックに保存する
- 関数を呼び出す

- 保存したレジスタを復元する
- 戻り値

このように、選択されたレジスタは呼び出し全体にわたって保存されるため、呼び出し元がレジスタを保存して復元する必要はありません。_ _C6000_call_stub によって保存されるレジスタは次のとおりです。A0、A1、A2、A6、A7、B0、B1、B2、B4、B5、B6、B7。

呼び出し元は、呼び出される関数のアドレスを B31 に設定し、その後 _ _C6000_call_stub に分岐することで、_ _C6000_call_stub を呼び出します。(復帰アドレスは通常どおり B3 に格納されます)。

_ _C6000_call_stub 関数は、次のように実装されています。

```
__c6xabi_call_stub:
        STW    A2, *B15--[2]
        STDW   A7:A6, *B15--
        STDW   A1:A0, *B15--
        STDW   B7:B6, *B15--
        STDW   B5:B4, *B15--
        STDW   B1:B0, *B15--
        STDW   B3:B2, *B15--
        ADDKPC __STUB_RET, B3, 0
        CALL   B31
__STUB_RET:
        LDDW   *++B15, B3:B2
        LDDW   *++B15, B1:B0
        LDDW   *++B15, B5:B4
        LDDW   *++B15, B7:B6
        LDDW   *++B15, A1:A0
        LDDW   *++B15, A7:A6
        LDW    *++B15[2], A2
        B      B3
```

(これは連続した、スケジュールされていない表現です。実際の実装については、TI のランタイム ライブラリのソース コードを参照してください)

_ _C6000_call_stub は非標準の規則を使用しているため、PLT エントリ経由で呼び出すことはできません。ライブラリ内の定義は、共有ライブラリからインポートできないように、STV_INTERNAL または STV_HIDDEN としてマークする必要があります。

_ _C6000_weak_return

_ _C6000_weak_return 関数は、単に return を行うだけの関数です。リンカは、インポートされた弱いシンボルへの未解決の呼び出しを含む動的実行可能ファイルまたは共有オブジェクトに、この関数を含める必要があります。動的リンカは、動的ロード時にこれらの呼び出しが未解決のままである場合、この関数を使用して解決できます。

_ _C6000_get_addr

_ _C6000_get_addr 関数は、32 ビットの TPR オフセットを受け取り、スレッドローカルのアドレスを返します。特別な値 -1 は、弱い未定義参照を示すために使用され、この場合は 0 が返されます。この関数は、静的実行可能ファイルおよびベア メタル動的 TLS アクセス モデルのコンパイル時に使用されます。スレッドローカル ストレージの詳細については、Chapter 7 を参照してください。

_ _C6000_get_tp

_ _C6000_get_tp 関数は、現在のスレッドのスレッド ポインタ値を返します。この関数は、戻り値レジスタ A4 以外のレジスタを変更しません。この関数は PLT 経由で呼び出すことができるため、呼び出し元は B30 と B31 がこの関数の呼び出しによって変更される可能性があると仮定する必要があります。スレッドローカル ストレージの詳細については、Chapter 7 および セクション 14.1.4 を参照してください。

_ _tls_get_addr

_ _tls_get_addr 関数は、スレッドローカル変数のアドレスを返します。この関数と、スレッドローカル変数のオフセットを指定するために渡される TLS_descriptor 構造体の詳細については、セクション 7.5.1.1 を参照してください。この関数は、

静的実行可能ファイルおよびベア メタル動的 TLS アクセス モデル以外のすべてのアクセス モデルのコンパイル時に使用されます。スレッドローカル ストレージの詳細については、Chapter 7 を参照してください。

8.3 ヘルパー関数における特別なレジスタ規則

ヘルパー関数は、前述で特に触れた例外を除き、通常の呼び出し規則に従います。ただし、典型的な実装では使用可能なレジスタの小さなサブセットしか必要としません。呼び出し元が、本来であれば呼び出しをまたいで保存されなければならないレジスタ (すなわち呼び出し元保存レジスタ) を使用していても、そのヘルパー関数がそれを使用しないことが分かっていれば、呼び出し元は保存を省略できます。この理由から、ABI は関数ごとにこれらのレジスタの指定を変更し、呼び出し元が不要なレジスタを保存しなくて済むようにしています。

コンパイラの立場から見れば、この情報の利用は任意であり、単に最適化の余地を与えるものです。ライブラリの実装者にとっては、ヘルパー関数の別実装も追加の規則に従うことが ABI によって義務付けられます。

特別なレジスタ規則を持つヘルパー関数は、PLT エントリを経由して呼び出すことはできません (セクション 6.5 を参照)。そのため、これらの定義は STV_INTERNAL または STV_HIDDEN としてマークし、共有ライブラリからインポートできないようにする必要があります。

表 8-9 には、レジスタ保存規則が変更されているヘルパー関数がまとめられています。表に載っている関数については、表に示されたレジスタのみが、関数の呼び出し時に変更されます。表にない関数は、通常の規則に従います。

表 8-9. C6000 におけるヘルパー関数のレジスタ規則

機能	変更されるレジスタ
_ _C6000_divi	A0、A1、A2、A4、A6、B0、B1、B2、B4、B5、B30、B31
_ _C6000_divu	A0、A1、A2、A4、A6、B0、B1、B2、B4、B30、B31
_ _C6000_remi	A1、A2、A4、A5、A6、B0、B1、B2、B4、B30、B31
_ _C6000_remu	A1、A4、A5、A7、B0、B1、B2、B4、B30、B31
_ _C6000_divremi	A1、A2、A4、A5、A6、B0、B1、B2、B4、B30、B31
_ _C6000_divremu	A0、A1、A2、A4、A6、B0、B1、B2、B4、B30、B31
_ _C6000_strasgi_64plus	A31、A30、B31、B30、ILC、RILC、B30、B31
_ _C6000_push_rts	A15、A3、B3、B30、B31
_ _C6000_pop_rts	B10、B11、B12、B13、B14、B30、B31
_ _C6000_call_stub	A3～A5、A8、A9、A16～A31、B8、B9、B16～B31、ILC、RILC

B30 と B31 は、呼び出し先で使用されていない場合でも、呼び出しによって変更されるものとして扱われます。これは、トランポリンでスクラッチ レジスタとして利用できるようにするためです。セクション 3.7 を参照してください。

8.4 複素数型のヘルパー関数

これらの関数では、複素数型の乗算と除算をサポートしています。この動作は C99 標準の付録 G で規定されています。

表 8-10. 複素数型のヘルパー関数

シグネチャ	説明
float64 complex **_ _C6000_mpycd**(float64 complex x, float64 complex y);	倍精度の複素数乗算
float32 complex **_ _C6000_mpycf**(float32 complex x, float32 complex y);	単精度の複素数乗算
float64 complex **_ _C6000_divcd**(float64 complex x, float64 complex y);	倍精度の複素数除算 (x/y)
float32 complex **_ _C6000_divcf**(float32 complex x, float32 complex y);	単精度の複素数除算 (x/y)

8.5 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 8-11. 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 _ _**C6000_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 _ _**C6000_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 _ _**C6000_isinf**(float64 x);	x が「無限大」を表す場合に限り真となります
int32 _ _**C6000_isinff**(float32 x);	x が「無限大」を表す場合に限り真となります
int32 _ _**C6000_isnan**(float64 x);	x が「非数」を表す場合に限り真となります
int32 _ _**C6000_isnanf**(float32 x);	x が「非数」を表す場合に限り真となります
int32 _ _**C6000_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 _ _**C6000_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 _ _**C6000_fpclassify**(float64 x);	浮動小数点値を分類します
int32 _ _**C6000_fpclassifyf**(float32 x);	浮動小数点値を分類します

関数 **_ _C6000_fpclassify** は浮動小数点数の分類に使用されます。処理は次のとおりです。

```
int32 __C6000_fpclassify(float64 x)
{
   if      (isnormal(x)) return 3;
   else if (isinf(x))    return 1;
   else if (isnan(x))    return 2;
   else                  return 4;
}
```

以下の C99 関数は、0 に向けて丸めおよび切り捨てを実行します。

表 8-12. 予約済み浮動小数点丸め関数

シグネチャ	説明
float64 _ _**C6000_nround**(float64 x);	最も近い整数に丸めます
float32 _ _**C6000_roundf**(float32 x);	最も近い整数に丸めます
float64 _ _**C6000_trunc**(float64 x);	0 に向けて切り捨てます
float32 _ _**C6000_truncf**(float32 x);	0 に向けて切り捨てます

Chapter 9

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 C6000 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。

9.1 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 13.4.4 に示すシンボル、またはセクション 13.1 に示す接頭辞のあるシンボルも予約されています。

9.2 <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 __c6xabi_abort_msg 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 8.2 を参照してください。

```
    void __c6xabi_abort_msg(const char *);
```

9.3 <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 C6000 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。

9.4 <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)

9.5 <errno.h> の実装

errno は、次のように関数呼び出しを含む式に展開されるマクロです。

```
#define errno (*__c6xabi_errno_addr())
extern int *__c6xabi_errno_addr(void);
```

この定義は、C6000 のスレッドローカル サポートの影響を受けます。Chapter 7 を参照してください。

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   33
#define ERANGE 34
#define ENOENT 2
#define EFPOS  152
#define EILSEQ 88
```

9.6 <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。

9.7 <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。

9.8 <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。

9.9 <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
```

9.10 <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;
```

9.11 <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。

9.12 <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

jmp_buf のサイズと整列は、13 個の「整数」の配列と同じです (すなわち 32 ビット x 13)。

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。

9.13 <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
```

9.14 <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。

9.15 <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。

9.16 <stddef.h> の実装

また、size_t と ptrdiff_t の各型は stddef.h に定義されています。セクション 2.1 を参照してください。

9.17 <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。

9.18 <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。C6000 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 int として定義されます。

9.19 <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。この関数はスレッドローカルである必要があります。Chapter 7 を参照。

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。

9.20 <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。

9.21 <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。

9.22 <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。

9.23 <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

9.24 <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;
```

Chapter 10

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 C6000 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。

10.1 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。C6000 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。

10.2 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。

10.3 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 2.7 で取り上げています。

10.4 初期化ガード変数 (GC++ABI 2.8)

ガード変数は、32 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 32 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;
}
```

10.5 コンストラクタ戻り値 (GC++ABI 3.1.5)

C6000 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、C6000 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。

10.6 ワンタイム構築 API (GC++ABI 3.3.2)

ガード変数は、32 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 10.4 を参照してください。

10.7 オブジェクト構成順序を制御する (GC++ ABI 3.3.4)

C6000 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。

10.8 デマングラ API (GC++ABI 3.4)

C6000 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。

10.9 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。C6000 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。

10.10 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。C6000 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。

10.11 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 11 に記載されています。

Chapter 11

例外処理

C6000 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。

11.1 概要

C6000 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、C6000 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。

11.2 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_C6000_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。

11.3 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。

11.3.1 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

11.3.2 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)	

11.3.3 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 11.4 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'

11.4 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 11-1 に示します。

11.4.1 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。

11.4.2 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)			

11.4.3 パーソナリティ ルーチン

C6000 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。最初の 3 つは 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 11-1. C6000 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)	_ _C6000_unwind_cpp_pr0	最大 3 つの 1 バイト命令	16	
0001	PR1 (Lu16)	_ _C6000_unwind_cpp_pr1	無制限の 1 バイト命令	16	
0010	PR2 (Lu32)	_ _C6000_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
0011	PR3	_ _C6000_unwind_cpp_pr3	24 ビット	16	最適化済みの C6x 固有のアンワインディング フォーマット
0100	PR4	_ _C6000_unwind_cpp_pr4	24 ビット	16	PR3 と同じですが、この関数のエピローグでは代替の C64x+ 用 *コンパクト フレーム レイアウト* を使用します。

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_C6000_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。

11.5 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 C6x 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。ただし、C64x+ 固有の _ _C6000_push_rts レイアウトが使用される場合は例外です。

11.5.1 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
   a. FP が使われている場合は、SP := FP
   b. それ以外の場合は、SP := SP + 定数
2. (任意) 呼び出し先保存レジスタから B3 を復元する
3. (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)
4. B3 経由でリターンする

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、FP の値をそのままコピーするか、定数でインクリメントすることで復元されます。後者の場合、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。SP が FP から復元される場合には、この追加のインクリメントは行われません。

ステップ 2：B3 を復元する

復帰を行う前に、復帰アドレスが必ず B3 に格納されている必要があります。呼び出し先保存レジスタ (たとえば「R」) に保存されている場合は、ステップ 3 で R を復元する前に、B3 を R から戻しておく必要があります。

ステップ 3：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 4.4.2) 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。TDEH では、C6000_push_rts レイアウトを使用する場合を除いて、この安全なデバッグ順が強制されます (セクション 4.4.4)。

コンパクト フレーム 方式 (セクション 4.4.4) を使用して作成されたスタック フレームでは、圧縮可能な命令を優先する最適化のため、保存レジスタ間に隙間が生じることがあります。アンワインダは、レジスタを配置するために使用されるアルゴリズムを理解し、それに応じてレジスタの位置を調整しなければなりません。

ビッグ エンディアン モードでは、LDDW および STDW の使用を容易にするため、ペアの 2 つのレジスタが同じ整列されたダブル ワードを占有する場合、そのペアの順序が入れ替えられます。これは、安全なデバッグ順を使用してレイアウトを決定した後に計算されるため、一部のレジスタ ペアは入れ替えられません。

一般に、SP (B15) は明示的なレジスタ復元によっては復元されません。FP が利用できない場合、DATA_MEM_BANK レイアウト (セクション 4.4.3) においてのみ明示的に復元されます。

ステップ 4：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 B3」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。

11.5.2 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 B3」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 11-2 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 11-2. スタック アンワインド命令

エンコード	手順	説明
00kk kkkk	SP += (k << 3) + 8	SP を小さな定数だけインクリメントします
1101 0010 kkkk kkkk . . .	SP += (ULEB128 << 3) + 0x408	SP を ULEB128 エンコードされた定数だけインクリメントします
1000 0000 0000 0000	CANTUNWIND	関数をアンワインドすることはできませんが、例外を捕捉する可能性があります
100x xxxx xxxx xxxx	POP ビットマスク	1 つ以上のレジスタを POP [x != 0]
101x xxxx xxxx xxxx	POP ビットマスク	C64x+ のコンパクト フレームから 1 つ以上のレジスタを POP [x != 0]
1100 nnnn xxxx xxxx . ..	POP レジスタ	n は POP するレジスタの数を表し、次の 4 ビット ニブルでエンコードされます
1101 0000	MV FP、SP	SP をインクリメントするのではなく 'FP から SP を復元します
1101 0001	_ _C6000_pop_rts	_ _C6000_pop_rts への呼び出しをシミュレートします
1110 0111	RET B3	このフレームのアンワインドの完了
1110 xxxx	復帰するか、B3 を復元します	B3 := レジスタ x (x != B3)

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。

小さなインクリメント

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

POP ビットマスク

7	6	5	4	3	2	1	0
1	**0**	**0**	A15	B15	B14	B13	B12
B11	B10	B3	A14	A13	A12	A11	A10

この 2 バイトの命令は、ビットマスクで指定された最大 13 個の呼び出し先保存レジスタを仮想スタックから POP する必要があることを示します。レジスタは、*安全なデバッグ*順序で定められた順番に従って復元する必要があります。

「POP ビットマスク」命令を使用して任意のレジスタが POP されると、SP はまず、呼び出し先保存レジスタ領域のサイズ分 (8 バイトに切り上げ)、**暗黙的にインクリメント**されます。これは、任意の明示的な SP インクリメント命令に加わるものです。ただし、「MV FP、SP」命令が使用されている場合、「POP ビットマスクが SP を暗黙的にインクリメントすることは**ありません**。

POP ビットマスク、C64x+ コンパクト フレーム

7	6	5	4	3	2	1	0
1	**0**	**1**	A15	B15	B14	B13	B12
B11	B10	B3	A14	A13	A12	A11	A10

POP ビットマスクと同じですが、C64x+ *コンパクト フレーム* レイアウトの使用を示し、このレイアウトは SP 自動デクリメントストアの使用を優先するためにスタックにホールを残すことがあります。アンワインダは、ホールを配置するアルゴリズムを理解し、その仕組みに合わせて適切に調整する必要があります。

POP レジスタ

7	6	5	4	3	2	1	0
1	**1**	**0**	**0**	n			
r0				r1			
r2				...			

コンパイラが安全なデバッグ順序を維持できない場合、または異なるレイアウトを選択するコンパイラの場合、それぞれの呼び出し先保存レジスタは個別に POP できます。4 ビットのオペコードの後の最初の 4 ビットは、POP されるレジスタの数を示します。その後の各 4 ビット ニブルは、呼び出し先保存レジスタのコード、または *ホール* を示す特別な値 0xF を表します。ホールが示されている場合、仮想 SP はデクリメントされるべきですが、レジスタは POP されるべきではありません。

4 ビット レジスタ エンコーディングは次のとおりです。

表 11-3. アンワインド命令におけるレジスタ エンコーディング

エンコード	登録	エンコード	登録
0000	A15	1000	A14
0001	B15	1001	A13
0010	B14	1010	A12
0011	B13	1011	A11
0100	B12	1100	A10
0101	B11	1101	予約済み
0110	B10	1110	予約済み
0111	B3	1111	「ホール」

MV FP、SP

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

この命令は、SP をインクリメントする代わりに、FP (A15) から SP を復元します。FP が利用可能な場合、SP の値を単に FP から復元する方が簡単です。DATA_MEM_BANK レイアウトでは、SP を復元する手段はこれしかない場合があります。

_ _C6000_pop_rts

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

この命令は、すべてのレジスタ復元が _ _C6000_pop_rts への呼び出しによって行われることを示します。この関数の動作は、アンワインダによってシミュレートされる必要があります。_ _C6000_pop_rts は B3 を暗黙的に復元し、RET B3 を実行します。

B3 を復元する

7	6	5	4	3	2	1	0
1	**1**	**1**	**0**	r	r	r	R

r が B3 以外のレジスタを表す場合、この命令は「MV reg, B3」をエンコードし、これによって「reg」から B3 が復元されます。POP がレジスタを上書きする場合、POP 命令の前にこれを実行する必要があります。

RET B3

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

この命令はシミュレーションされた *リターン* をエンコードし、このフレームのアンワインドが完了したことを示します。エンコードは「Restore B3」と同じですが、ソース レジスタは B3 自体として示されます。

アンワインド命令の各シーケンスは、明示的または暗黙的な「RET B3」で終了します。この命令は明示的なアンワインド命令から省略することができ、アンワインダが暗黙的にこの命令を追加します。

11.5.3 24 ビットのアンワインド エンコーダ

PR3 と PR4 は最適化されたエンコーディングを使用します。ほとんどの関数は PR3 を使用します。C64x+ コードをサイズに最適化する場合は PR4 を使用します。

31	30-28	27-24	23-17	16-4	3-0
1	**000**	Index	Stack increment	Register bitmask	Return register

スタック インクリメントは、バイト エンコードされた小さな定数インクリメントと類似していますが、8 によってバイアスされません。特殊なインクリメント値 0x7F は、「MV FP、SP」をエンコードするために使用されます。スタック インクリメントの値が 0x7F でない場合、SP は (value << 3) によってインクリメントされます。

復帰レジスタ フィールドは、表 11-3 のエンコーディングを使用して、復帰アドレスが格納されるレジスタを符号化します。このレジスタが B3 自身以外の任意のレジスタである場合、POP 操作を実行する前に、このレジスタから B3 を復元しなければなりません (次の段落)。

ビットマスクは、バイト エンコードされた POP ビットマスク命令と同じように解釈されます。パーソナリティ ルーチンが PR3 である場合、非コンパクト POP 命令が実行されます。パーソナリティ ルーチンが PR4 である場合、コンパクト フレーム POP 命令が実行されます。これには、SP の暗黙的なインクリメントが含まれる場合があります。

11.6 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			

11.6.1 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 11.6.5) 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。特別な再配置タイプ R_C6000_EHTYPE が EXTAB 内の type_info 参照をマークするために使用されます。

リンカは、type_info フィールドを type_info オブジェクトへの DP 相対オフセットとしてエンコードし、これによりテーブルの位置の独立性を保持します。オフセットは、参照されるキャッチ句または例外仕様を含む関数を定義するモジュールの静的ベースを基準にしています。

11.6.2 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、PR1、PR3、PR4 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 11-1. 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 11-2. 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。

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

11.6.3 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。

31-0

Scope (long or short form)	
0	PREL31 program address of landing pad

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。

11.6.4 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。

31-0

Scope (long or short form)		0
R	PREL31 program address of landing pad	
Type		

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 (R_C6000_EHTYPE 再配置によって再配置される)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。

11.6.5 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_C6000_NONE 再配置が含まれている必要があります。

11.7 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_C6000_UNWIND タイプの C6000.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 C6000.exidx セクションを 1 つの連続した C6000.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 C6000.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。動的リンク モジュールの場合、位置独立性を容易にするために、テーブルはコードと同じセグメントに配置する必要があります。

11.8 C++ 以外のコードとの相互作用

11.8.1 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。

11.8.2 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。

11.9 システム機能との相互作用

11.9.1 共有ライブラリ

例外処理テーブルは、実行可能ファイルまたは共有ライブラリ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。

11.9.2 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。

11.9.3 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。

11.10 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)。このフィールドは、R_C6000_EHTYPE で再配置されます。

$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。

Chapter 12

DWARF

C6000 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、C6000 に固有の表現を定義しています。

12.1 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 12-1 では DWARF3 レジスタ番号/名前から C6000 レジスタへのマッピングを定義します。

表 12-1. C6000 の DWARF3 レジスタ番号

DWARF 名	C6000 ISA レジスタ	説明
0-15	A0-A15	
16-31	B0-B15	
32	予約済み	
33	PCE1	E1 位相プログラム カウンタ
34	IRP	割り込み復帰ポインタ レジスタ
35	IFR	割り込みフラグ レジスタ
36	NRP	NMI 復帰ポインタ レジスタ
37-52	A16-A31	
53-68	B16-B31	
69	AMR	アドレス モード レジスタ
70	CST	制御ステータス レジスタ
71	ISR	割り込みセット レジスタ
72	ICR	割り込みクリア レジスタ
73	IER	割り込みイネーブル レジスタ
74	ISTP	割り込みサービス テーブル ポインタ レジスタ
75	IN	文書化されていない制御レジスタ
76	OUT	文書化されていない制御レジスタ
77	ACR	文書化されていない制御レジスタ
78	ADR	文書化されていない制御レジスタ
79	FADCR	浮動小数点加算器構成レジスタ
80	FAUCR	浮動小数点補助構成レジスタ
81	FMCR	浮動小数点逓倍器構成レジスタ
82	GFPGFR	ガロア フィールド多項式ジェネレータ関数レジスタ
83	DIER	文書化されていない制御レジスタ
84	REP	制限付きエントリ ポイント レジスタ
85	TSCL	タイム スタンプ カウンタ - 下半分
86	TSCH	タイム スタンプ カウンタ - 上半分
87	ARP	文書化されていない制御レジスタ
88	ILC	SPLOOP 内部ループ カウント レジスタ
89	RILC	SPLOOP リロード内部ループカウント レジスタ
90	DNUM	DSP コア番号レジスタ
91	SSR	飽和ステータス レジスタ
92	GPLYA	GMPY 多項式 - A 側レジスタ
93	GPLYB	GMPY 多項式 - B 側レジスタ
94	TSR	タスク状態レジスタ
95	ITSR	割り込みタスク状態レジスタ
96	NTSR	NMI/例外タスク状態レジスタ
97	EFR	例外フラグ レジスタ
98	ECR	例外クリア レジスタ
99	IERR	内部例外レポート レジスタ

表 12-1. C6000 の DWARF3 レジスタ番号 (続き)

DWARF 名	C6000 ISA レジスタ	説明
100	DMSG	文書化されていない制御レジスタ
101	CMSG	文書化されていない制御レジスタ
102	DT_DMA_ADDR	文書化されていない制御レジスタ
103	DT_DMA_DATA	文書化されていない制御レジスタ
104	DT_DMA_CNTL	文書化されていない制御レジスタ
105	TCU_CNTL	文書化されていない制御レジスタ
106	RTDX_REC_CNTL	文書化されていない制御レジスタ
107	RTDX_XMT_CNTL	文書化されていない制御レジスタ
108	RTDX_CFG	文書化されていない制御レジスタ
109	RTDX_RDATA	文書化されていない制御レジスタ
110	RTDX_WDATA	文書化されていない制御レジスタ
111	RTDX_RADDR	文書化されていない制御レジスタ
112	RTDX_WADDR	文書化されていない制御レジスタ
113	MFREG0	文書化されていない制御レジスタ
114	DBG_STAT	文書化されていない制御レジスタ
115	BRK_EN	文書化されていない制御レジスタ
116	HWBP0_CNT	文書化されていない制御レジスタ
117	HWBP0	文書化されていない制御レジスタ
118	HWBP1	文書化されていない制御レジスタ
119	HWBP2	文書化されていない制御レジスタ
120	HWBP3	文書化されていない制御レジスタ
121	OVERLAY	文書化されていない制御レジスタ
122	PC_PROF	文書化されていない制御レジスタ
123	ATSR	文書化されていない制御レジスタ
124	TRR	文書化されていない制御レジスタ
125	TCRR	文書化されていない制御レジスタ
126	DESR	文書化されていない制御レジスタ
127	DETR	文書化されていない制御レジスタ
128	STRM_HOLD	文書化されていない制御レジスタ
129	PDATA_O	文書化されていない制御レジスタ
130	TCR	文書化されていない制御レジスタ

12.2 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 12-1 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 C6000 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 (B15) の値として定義しています。

DWARF3 標準 (セクション 6.4.4) で提案されている仮想復帰アドレス用の専用列は、アンワインド テーブルには設けられていません。呼び出し規則に従い、復帰アドレスはアンワインド テーブルの B3 列で表されます。

アンワインド テーブルには、C6000 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。

12.3 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 C6000 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)

12.4 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、C6000 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。

C6000 向けの TI ベンダー固有の DIE タグは、表 12-2 に定義されています。TI ベンダー固有の属性は、表 12-2 に定義されています。

表 12-2. 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 12-3. 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_plt_entry	0x2012	フラグ	関数は PLT エントリ
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。

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_plt_entry:この属性は、プロシージャ リンケージ テーブルのエントリに対応する DW_TAG_subprogram DIE に、ゼロ以外の値とともに追加されます。その意味は DW_AT_trampoline 属性に類似しています。

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。

Chapter 13

ELF オブジェクト ファイル (プロセッサ補足)

C6000 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 C6000 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 14 に記載されています。

13.1 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 13-1 に記載されています。

表 13-1. 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。
c6xabi、_ _c6xabi	C6000 EABI で指定されるシンボルの共通名前空間。
C6000	C6000 で指定される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、C6000 では SHT_C6000_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。

13.2 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、C6000 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 13-2」では、C6000 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 13-2. 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
	ELFDATA2MSB	2	ビッグ エンディアン
EI_VERSION	EV_CURRENT	1	
EI_OSABI	ELFOSABI_C6000_ELFABI	64	ベアメタル動的リンク プラットフォーム
EI_OSABI	ELFOSABI_C6000_LINUX	65	MMU を使用しない Linux プラットフォーム
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。ベアメタル動的リンク モデル (セクション 14.4) や Linux (Chapter 15) では、このフィールドに固有の値が定義されています。

ELFOSABI_NONE 以外の値は、指定された値に対応する特定の ABI バリアントの規則にファイルが準拠していることを表します。特定のプラットフォームで有効なのは、規定に沿ったファイルだけです。ABI で定義された特定のバリアント以外のプラットフォーム向けにオブジェクトを構築することもできますですが、その場合は ELFOSABI_NONE と識別され、特定の規則に準拠していることを示さないことを意味します。こうしたファイルが特定の環境と互換性があるかどうかの判断は、ABI とは独立して行われます。

e_type

現時点では、C6000 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。

e_machine

本仕様に準拠するオブジェクト ファイルは、EM_TI_C6000 (140, 0x8c) 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。C6000 向けには 1 つ固有のフラグがあります。

名称	値	コメント
EF_C6000_REL	0x1	ファイルは静的再配置情報を含みます

EF_C6000_REL フラグは、実行可能ファイル (ET_EXEC) や共有オブジェクト (ET_DYN) に静的再配置情報が含まれていることを示します。このような静的再配置情報を持つ共有オブジェクトは *再配置可能モジュール* と呼ばれ、静的リンクと動的リンクのどちらでも利用できるライブラリに一般的に使用されます。

13.3 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。

13.3.1 セクション インデックス

ABI は、次のような特別なセクション インデックスを 1 つ定義します。

名称	値	コメント
SHN_C6000_SCOMMON	0xFF00	ニア DP アドレッシングを使用する共通ブロック シンボル

SHN_C6000_SCOMMON インデックスは、ニア DP アドレッシングでアドレス指定された共通ブロック シンボルを識別します。セクション 13.4.2 を参照してください。

13.3.2 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 13-3 のとおりです。

これらすべてのセクション タイプが C6000 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 C6000 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 13-3. 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_C6000_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_C6000_PREEMPTMAP	0x70000002	DLL 動的リンク プリエンプション マップ
SHT_C6000_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_PHATTRS	0x7F000004	拡張プログラム ヘッダー属性
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_C6000_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 11 を参照してください。

SHT_C6000_PREEMPTMAP は、C6000 DLL 動的リンク プリエンプション マップを含むセクションを識別します。

SHT_C6000_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 17 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 18 を参照してください。

SHT_TI_PHATTRS は、実行可能ファイルまたは共有オブジェクト ファイルのプログラム セグメントに関する追加のプロパティを含むセクションを識別します。Chapter 19 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 C6000 では使用されません。

13.3.3 拡張セクション ヘッダー属性

プロセッサ固有のセクション属性フラグ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 ヘッダー属性については、Chapter 19 に記載されています。

13.3.4 サブセクション

C6000 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 13.3.5 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。

13.3.5 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 13-4 には、C6000 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 13-4. C6000 特別なセクション

接頭辞	タイプ	属性
		コード セクション

表 13-4. C6000 特別なセクション (続き)

接頭辞	タイプ	属性
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.plt	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
	ニア データ セクション	
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.neardata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.rodata	SHT_PROGBITS	SHF_ALLOC
	ファー データ セクション	
.far	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.fardata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.const	SHT_PROGBITS	SHF_ALLOC
.fardata:const	SHT_PROGBITS	SHF_ALLOC
	動的データ セクション	
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.dsbt	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
	例外処理データ セクション	
.C6000.exidx	SHT_C6000_UNWIND	SHF_ALLOC + SHF_LINK_ORDER
.C6000.extab	SHT_PROGBITS	SHF_ALLOC
	初期化セクションと終了セクション	
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC + SHF_WRITE
.init_array	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE
.fini_array	SHT_FINI_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
	動的ローディング構造	
.dynamic (1)	SHT_DYNAMIC	SHF_ALLOC
.dynsym (1)	SHT_DYNSYM	SHF_ALLOC
.dynstr (1)	SHT_STRTAB	SHF_ALLOC + SHF_STRINGS
.hash (1)	SHT_TAB	SHF_ALLOC
.interp	SHT_PROGBITS	なし
	ビルド属性	
.C6000.attributes	SHT_C6000_ATTRIBUTES	なし
	シンボリック デバッグ セクション	
.debug (2)	SHT_PROGBITS	なし
	シンボル バージョニング セクション(1)	
.gnu.version	SHT_GNU_versym	SHF_ALLOC
.gnu.version_d	SHT_GNU_verdef	SHF_ALLOC
.gnu.version_r	SHT_GNU_verneed	SHF_ALLOC
	スレッドローカル ストレージのセクション	

表 13-4. C6000 特別なセクション (続き)

接頭辞	タイプ	属性
.tbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.TI.tls_init	SHT_PROGBITS	SHF_ALLOC
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
.cio	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.phattrs	SHT_TI_PHATTRS	なし
.TI.preempt.map	SHT_C6000_PREEMPTMAP	SHF_ALLOC
.TI.xref	SHT_TI_XREF	なし
.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、C6000 EABI によって使用されていないもの		
.comment		
.data		
.data1		
.line		
.rodata1		

(1) .dynamic セクションや関連セクションをメモリに割り当てるかどうかは、プラットフォームに依存します。
(2) .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、C6000 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、C6000 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。

スレッドローカル ストレージの詳細については、Chapter 7 を参照してください。

さらに、.common と .scommon はリンカで使用されるセクション名です。これらはオブジェクト ファイルに存在する実際のセクションではなく、抽象的なセクションです。名前はリンカ コマンド ファイル内で変数を配置するための規則として用いられています。これらのセクションは、他の目的には使用しないでください。

13.3.6 セクションの整列

C6000 のコードを含むセクションは、少なくとも 32 バイトに整列され、32 バイト境界までパディングされる必要があります。後者の要件は、C64+ 以降のアーキテクチャで隣接するデータをフェッチ パケット ヘッダとして誤って認識しないようにすることです。

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)。

13.4 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。

C6000 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。

13.4.1 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。

さらに、スレッドローカル シンボルはシンボル タイプ STT_TLS を持ちます。

13.4.2 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。この C6000 ABI では、共通ブロックの仕組みを拡張し、ニア データ アクセスとファー データ アクセスの両方に対応できるようにしています。共通ブロック シンボルがニア DP 相対アドレッシングで参照される場合、セクション インデックスにはプロセッサ固有の値 SHN_C6000_SCOMMON が割り当てられる必要があります。リンカはこのようなシンボルをニア データ セクションに割り当て、通常は .bss に置かれます。

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。これらのシンボルはファー データ セクションに配置され、通常は .far に置かれます。

13.4.3 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。

C6000 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

13.4.4 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 13-1 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 4.5 に記載)

13.4.5 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。

13.5 再配置

C6000 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 C6000 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。

13.5.1 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 13-5 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 13-6 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 13-5 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 13-6 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。
FP(x)	アドレス x にある命令を含むフェッチ パケットのアドレス。すなわち、FP(x) := x & 0xFFFFFFE0。
P	再配置される命令のフェッチ パケットアドレス。すなわち、P := FP(PC)。
B	現在のロード モジュールにおけるデータ セグメントのベース アドレス。この位置はシンボル _ _C6000_DSBT_BASE によってマークされ、プログラム実行時には DP レジスタの値となります。
GOT(S)	再配置に関連付けられたシンボル (S) のグローバル オフセット テーブル (GOT) エントリのアドレス。
TBR(x)	スレッドローカル ストレージ (TLS) ブロック ベースからの x のオフセット。スレッドローカル ストレージの詳細については、Chapter 7 を参照してください。
TPR(x)	スレッドポインタ (TP) からの x のオフセット。
TLS(x)	x の TLS ディスクリプタ。これは module-id および x の TBR オフセットを含みます。
TLSMOD(x)	x を定義するモジュールの TLS モジュール識別子。

表 13-5. C6000 再配置タイプ

名称	値	動作	制約
R_C6000_NONE	0		
R_C6000_ABS32	1	S + A	
R_C6000_ABS16	2	S + A	
R_C6000_ABS8	3	S + A	
R_C6000_PCR_S21	4	S + A – P	
R_C6000_PCR_S12	5	S + A – P	
R_C6000_PCR_S10	6	S + A – P	

表 13-5. C6000 再配置タイプ (続き)

名称	値	動作	制約
R_C6000_PCR_S7	7	S + A – P	
R_C6000_ABS_S16	8	S + A	
R_C6000_ABS_L16	9	S + A	
R_C6000_ABS_H16	10	S + A	Rela のみ
R_C6000_SBR_U15_B	11	S + A – B	
R_C6000_SBR_U15_H	12	S + A – B	
R_C6000_SBR_U15_W	13	S + A – B	
R_C6000_SBR_S16	14	S + A – B	
R_C6000_SBR_L16_B	15	S + A – B	
R_C6000_SBR_L16_H	16	S + A – B	
R_C6000_SBR_L16_W	17	S + A – B	
R_C6000_SBR_H16_B	18	S + A – B	Rela のみ
R_C6000_SBR_H16_H	19	S + A – B	Rela のみ
R_C6000_SBR_H16_W	20	S + A – B	Rela のみ
R_C6000_SBR_GOT_U15_W	21	GOT(S) + A – B	
R_C6000_SBR_GOT_L16_W	22	GOT(S) + A – B	
R_C6000_SBR_GOT_H16_W	23	GOT(S) + A – B	Rela のみ
R_C6000_DSBT_INDEX	24	この静的リンク ユニットの DSBT インデックス	
R_C6000_PREL31	25	S + A – PC	
R_C6000_COPY	26	プリエンプトされたシンボルのロード時コピー	ET_EXEC のみ
R_C6000_JUMP_SLOT	27	S + A	ET_EXEC / ET_DYN
R_C6000_EHTYPE	28	S + A - B	
R_C6000_PCR_H16	29	S - FP(P - A)	Rela のみ
R_C6000_PCR_L16	30	S - FP(P - A)	Rela のみ
予約済み	31		
予約済み	32		
R_C6000_TBR_U15_B	33	TBR(S)	静的のみ
R_C6000_TBR_U15_H	34	TBR(S)	静的のみ
R_C6000_TBR_U15_W	35	TBR(S)	静的のみ
R_C6000_TBR_U15_D	36	TBR(S)	静的のみ
R_C6000_TPR_S16	37	TBR(S)	
R_C6000_TPR_U15_B	38	TPR(S)	
R_C6000_TPR_U15_H	39	TPR(S)	
R_C6000_TPR_U15_W	40	TPR(S)	
R_C6000_TPR_U15_D	41	TPR(S)	
R_C6000_TPR_U32_B	42	TPR(S)	動的のみ
R_C6000_TPR_U32_H	43	TPR(S)	動的のみ
R_C6000_TPR_U32_W	44	TPR(S)	動的のみ
R_C6000_TPR_U32_D	45	TPR(S)	動的のみ
R_C6000_SBR_GOT_U15_W_TLSMOD	46	GOT(TLSMOD(S)) + A – B	静的のみ
R_C6000_SBR_GOT_U15_W_TBR	47	GOT(TBR(S)) + A - B	静的のみ
R_C6000_SBR_GOT_U15_W_TPR_B	48	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_U15_W_TPR_H	49	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_U15_W_TPR_W	50	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_U15_W_TPR_D	51	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_L16_W_TLSMOD	52	GOT(TLSMOD(S)) + A - B	静的のみ

表 13-5. C6000 再配置タイプ (続き)

名称	値	動作	制約
R_C6000_SBR_GOT_L16_W_TBR	53	GOT(TBR(S)) + A - B	静的のみ
R_C6000_SBR_GOT_L16_W_TPR_B	54	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_L16_W_TPR_H	55	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_L16_W_TPR_W	56	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_L16_W_TPR_D	57	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_H16_W_TLSMOD	58	GOT(TLSMOD(S)) + A - B	静的のみ
R_C6000_SBR_GOT_H16_W_TBR	59	GOT(TBR(S)) + A - B	静的のみ
R_C6000_SBR_GOT_H16_W_TPR_B	60	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_H16_W_TPR_H	61	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_H16_W_TPR_W	62	GOT(TPR(S))+A-B	静的のみ
R_C6000_SBR_GOT_H16_W_TPR_D	63	GOT(TPR(S))+A-B	静的のみ
R_C6000_TLSMOD	64	TLSMOD(S)	動的のみ
R_C6000_TBR_U32	65	TBR(S)	動的のみ
R_C6000_ALIGN	253	なし	ET_REL のみ
R_C6000_FPHEAD	254	なし	ET_REL のみ
R_C6000_NOCMP	255	なし	ET_REL のみ

R_NONE 再配置では、操作は実行されません。これは、参照側のセクションがリンクされる場合には参照先も同様に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_C6000_ABS8/16/32 再配置は、シンボルの再配置後のアドレスを 8 ビット、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。これらは通常、命令ではなく初期化データに使用されます。フィールドの符号属性は未定義です。すなわち、符号付き値にも符号なし値にも使用されます。

```
        .field X,32         ; R_C6000_ABS32
        .field X,16         ; R_C6000_ABS16
        .field X,8          ; R_C6000_ABS8
```

PCR 再配置は、符号付き PC 相対分岐ディスプレイスメントをエンコードします。それらは 32 ビット (ワード) 単位にスケーリングされます。ディスプレイスメントはソース命令のフェッチ パケットを基準として計算されます。

```
        B       func        ; R_C6000_PCR_S21
        CALLP   func,B3     ; R_C6000_PCR_S21
        BNOP    func        ; R_C6000_PCR_S12
        BPOS    func,A10    ; R_C6000_PCR_S10
        BDEC    func,A1     ; R_C6000_PCR_S10
        ADDKPC  func,B3,4   ; R_C6000_PCR_S7
```

名前に L16 を含む再配置は、32 ビットアドレスまたはオフセットの下位 16 ビットをエンコードします。H16 を含むものは上位 16 ビットをエンコードし、常に Rela です。S16 を含む再配置は符号付き 16 ビット値 (一般的にアドレスの一部ではありません) をエンコードします。U15 を含むものは符号なし 15 ビットの DP 相対ディスプレイスメントをエンコードします。

```
        MVHL    sym,A0      ; R_C6000_ABS_L16
        MVKH    sym,A0      ; R_C6000_ABS_H16
        MVK     const16,A0  ; R_C6000_ABS_S16  sign extend const16 into A0
        MVKLH   const16,A0  ; R_C6000_ABS_L16  move const16 into A0[16:31]
```

PCR_L16 および PCR_H16 再配置は、それぞれ、ターゲット アドレスと参照 PC (「ベース PC」) のフェッチ パケット アドレスとの間の PC 相対オフセットの下位ビットおよび上位ビットをエンコードします。現在の命令のフェッチ パケットからベース PC までのオフセットは加数フィールドにエンコードされます。すなわち、A := (P-base) です。再配置はその後、S-FP(P-A) を計算し、その結果 S と FP(base) の間のオフセットが得られます。これらの再配置は、セクション 5.1 に記載

されているように、PC 相対アドレッシングを使用して異なるセクション内のオブジェクトをアドレッシングするために使用されます。

```
        MVK     $PCR_OFFSET(sym,base),A0     ; R_C6000_PCR_L16
        MVKH    $PCR_OFFSET(sym,base),A0     ; R_C6000_PCR_H16
```

SBR_U15 再配置は、ニア DP データ アドレッシング用に 15 ビットの符号なし DP 相対オフセットをエンコードします。それらはアクセス幅に応じてスケーリングされます。(32 ビットワード (_W)、16 ビット ハーフワード (_H)、またはバイト (_B))。

```
        LDB     *+DP(sym),A1 ; R_C6000_SBR_U15_B
        ADDAB   DP,sym,A2    ; R_C6000_SBR_U15_B
        LDH     *+DP(sym),A1 ; R_C6000_SBR_U15_H
        ADDAH   DP,sym,A2    ; R_C6000_SBR_U15_H
        LDW     *+DP(sym),A1 ; R_C6000_SBR_U15_W
        ADDAW   DP,sym,A2    ; R_C6000_SBR_U15_W
```

その他の SBR 再配置は、ファー DP 相対アドレッシング用に 32 ビット DP 相対オフセットの上位部と下位部をエンコードするために使用されます。以下の例では、

- $bss はデータ セグメントのベース アドレスを表します。これは _ _C6000_DSBT_BASE (DP に格納される値) に対応します
- $DPR_byte(sym) はバイト単位の DP 相対オフセットを表します
- $DPR_hword(sym) は DP 相対オフセットを 2 で割った値を表します
- $DPR_word(sym) は DP 相対オフセットを 4 で割った値を表します

```
        MVK     (sym - $bss),A0      ; R_C6000_SBR_S16
        MVKL    $DPR_byte(sym),A0    ; R_C6000_SBR_L16_B
        MVKH    $DPR_byte(sym),A0    ; R_C6000_SBR_H16_B
        MVKL    $DPR_hword(sym),A0   ; R_C6000_SBR_L16_H
        MVKH    $DPR_hword(sym),A0   ; R_C6000_SBR_H16_H
        MVKL    $DPR_word(sym),A0    ; R_C6000_SBR_L16_W
        MVKH    $DPR_word(sym),A0    ; R_C6000_SBR_H16_W
```

SBR_GOT 再配置は、SBR 再配置と同じ命令およびエンコーディングに対応しますが、シンボル自体ではなく、参照されるシンボルの DP 相対 GOT アドレスを参照します。通常、GOT はニア DP 相対アドレッシングでアクセスされるため、R_C6000_DBR_GOT_U15_W が使用されます。GOT がファーの場合、オフセットは他の 2 つの再配置とともに MVKL/MVKH で生成されます (セクション 6.6 を参照)。以下の例では、

- GOT(sym) は、sym の GOT エントリの DP 相対オフセット (バイト単位) です
- $DPR_GOT(sym) は、sym の GOT エントリの DP 相対オフセット (ワード単位) です

```
        LDW     *+DP[GOT(sym)],A0    ; R_C6000_SBR_GOT_U15_W
        MVKL    $DPR_GOT(sym), A0    ; R_C6000_SBR_GOT_L16_W
        MVKH    $DPR_GOT(sym), A0    ; R_C6000_SBR_GOT_H16_W
```

R_C6000_DSBT_INDEX は、現在のロード モジュールのデータ セグメント ベース テーブルへのインデックスをエンコードします。これは、位置の独立性のために DSBT モデルを使用するファイルにのみ存在します。セクション 6.7 を参照してください。

```
        LDW     *+DP($DSBT_INDEX(__C6000_DSBT_BASE)),DP   ; R_C6000_DSBT_INDEX
```

R_C6000_COPY は、セクション 15.9 に記載されている import-as-own 規則の下で、ライブラリの定義をプリエンプトする実行可能ファイルに定義された重複シンボルをマークするために使用されます。実行可能ファイルがロードされると、動的ローダはライブラリの定義から実行可能ファイルの定義へ初期値をコピーする必要があります。この再配置タイプは、実行可能ファイル (ET_EXEC) の動的再配置テーブルにのみ存在します。

R_6000_JUMP_SLOT は、インポートされた関数を参照し、かつ PLT エントリからのみ参照される GOT エントリをマークするために使用され、そのためセクション 15.6 に記載されているように遅延バインディングの対象となります。

R_C6000_JUMP_SLOT 再配置は、実行可能ファイルおよび共有オブジェクト内でのみ発生し、動的再配置テーブルの DT_JMPREL セクション内にのみ存在します。

R_C6000_PREL31 は、例外処理テーブル内のコード アドレスをエンコードするために使用されます。
R_C6000_EHTYPE は、例外処理テーブル内の typeinfo 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 11.2 を参照してください。

値が 33 から 65 の再配置は、スレッドローカル ストレージ (TLS) と共に使用されます。これらの再配置には、R_C6000_TBR_*、R_C6000_TPR_*、R_C6000_SBR_GOT_*_W_T*、R_C6000_TLSMOD、R_C6000_TBR_U32 の再配置が含まれます。スレッドローカル ストレージの詳細については、Chapter 7 を参照してください。これらの TLS 再配置を使用する例については、セクション 7.5 を参照してください。

R_C6000_ALIGN と R_C6000_FPHEAD は、C64+ 圧縮機のマーカーとして使用されます。これらは ABI の下では何の効果も持ちません。部分リンクのように再配置可能ファイル (ET_REL) を他の再配置可能ファイルに結合する下流のツールは、それらを保持するか、またはそれらが現れるセクションを R_C6000_NOCMP でマークしなければなりません。

R_C6000_NOCMP は、セクションを圧縮不可能としてマークします。

13.5.2 再配置操作

表 13-6 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。
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。

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。
- R_C6000_DSBT_INDEX 再配置は、エンコードされた値がモジュールの DSBT テーブルのサイズ以上である場合にオーバーフローします。

表 13-6. C6000 再配置操作

再配置名	符号属性	フィールド [CS, 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_C6000_NONE	なし	[32, 0, 32]	なし	なし	なし	なし
R_C6000_ABS32	いずれでもよい	[32, 0, 32]	F	S + A	なし	R
R_C6000_ABS16	いずれでもよい	[16, 0, 16]	SE(F)	S + A	あり	R
R_C6000_ABS8	いずれでもよい	[8, 0, 8]	SE(F)	S + A	あり	R
R_C6000_PCR_S21	符号付き	[32, 7, 21]	SE(F << 2)	S + A – P	あり	R >> 2
R_C6000_PCR_S12	符号付き	[32, 16, 12]	SE(F << 2)	S + A – P	あり	R >> 2
R_C6000_PCR_S10	符号付き	[32, 13, 10]	SE(F << 2)	S + A – P	あり	R >> 2
R_C6000_PCR_S7	符号付き	[32, 16, 7]	SE(F << 2)	S + A – P	あり	R >> 2

表 13-6. C6000 再配置操作 (続き)

再配置名	符号属性	フィールド [CS, 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_C6000_ABS_S16	符号付き	[32, 7, 16]	SE(F)	S + A	あり	R
R_C6000_ABS_L16	なし	[32, 7, 16]	F	S + A	なし	R
R_C6000_ABS_H16	なし	[32, 7, 16]	r_addend	S + A	なし	R >> 16
R_C6000_SBR_U15_B	符号なし	[32, 8, 15]	ZE(F)	S + A – B	あり	R
R_C6000_SBR_U15_H	符号なし	[32, 8, 15]	ZE(F << 1)	S + A – B	あり	R >> 1
R_C6000_SBR_U15_W	符号なし	[32, 8, 15]	ZE(F << 2)	S + A – B	あり	R >> 2
R_C6000_SBR_S16	符号付き	[32, 7, 16]	SE(F)	S + A – B	あり	R
R_C6000_SBR_L16_B	符号なし	[32, 7, 16]	ZE(F)	S + A – B	なし	R
R_C6000_SBR_L16_H	符号なし	[32, 7, 16]	ZE(F << 1)	S + A – B	なし	R >> 1
R_C6000_SBR_L16_W	符号なし	[32, 7, 16]	ZE(F << 2)	S + A – B	なし	R >> 2
R_C6000_SBR_H16_B	符号なし	[32, 7, 16]	r_addend	S + A – B	なし	R >> 16
R_C6000_SBR_H16_H	符号なし	[32, 7, 16]	r_addend	S + A – B	なし	R >> 17
R_C6000_SBR_H16_W	符号なし	[32, 7, 16]	r_addend	S + A – B	なし	R >> 18
R_C6000_SBR_GOT_U15_W	符号なし	[32, 8, 15]	ZE(F << 2)	GOT(s) + A – B	あり	R >> 2
R_C6000_SBR_GOT_L16_W	符号なし	[32, 7, 16]	ZE(F << 2)	GOT(s) + A – B	なし	R >> 2
R_C6000_SBR_GOT_H16_W	符号なし	[32, 7, 16]	r_addend	GOT(s) + A – B	なし	R >> 18
R_C6000_DSBT_INDEX	符号なし	[32, 8, 15]	なし	DSBT インデックス	あり	R
R_C6000_PREL31	なし	[32, 0, 31]	SE(F << 1)	S + A - PC	なし	R >> 1
R_C6000_COPY	なし	[32, 0, 32]	なし	F	なし	F
R_C6000_JUMP_SLOT	いずれでもよい	[32, 0, 32]	F	S + A	なし	R
R_C6000_EHTYPE	いずれでもよい	[32, 0, 32]	F	S + A - B	なし	R
R_C6000_PCR_H16	符号付き	[32, 7, 16]	r_addend	S - FP(P - A)	なし	R >> 16
R_C6000_PCR_L16	なし	[32, 7, 16]	r_addend	S - FP(P - A)	なし	R
R_C6000_TBR_U15_B	符号なし	[32,8,15]	ZE(F)	TBR(S)	あり	R
R_C6000_TBR_U15_H	符号なし	[32,8,15]	ZE(F<<1)	TBR(S)	あり	R >> 1
R_C6000_TBR_U15_W	符号なし	[32,8,15]	ZE(F<<2)	TBR(S)	あり	R >> 2
R_C6000_TBR_U15_D	符号なし	[32,8,15]	ZE(F<<3)	TBR(S)	あり	R >> 3
R_C6000_TPR_S16	符号付き	[32,7,16]	SE(F)	TBR(S)	あり	R
R_C6000_TPR_U15_B	符号なし	[32,8,15]	ZE(F)	TPR(S)	あり	R
R_C6000_TPR_U15_H	符号なし	[32,8,15]	ZE(F<<1)	TPR(S)	あり	R >> 1
R_C6000_TPR_U15_W	符号なし	[32,8,15]	ZE(F<<2)	TPR(S)	あり	R >> 2
R_C6000_TPR_U15_D	符号なし	[32,8,15]	ZE(F<<3)	TPR(S)	あり	R >> 3
R_C6000_TPR_U32_B	符号なし	[32,0,326]	ZE(F)	TPR(S)	なし	R
R_C6000_TPR_U32_H	符号なし	[32,0,326]	ZE(F<<1)	TPR(S)	なし	R >> 1
R_C6000_TPR_U32_W	符号なし	[32,0,326]	ZE(F<<2)	TPR(S)	なし	R >> 2
R_C6000_TPR_U32_D	符号なし	[32,0,326]	ZE(F<<3)	TPR(S)	なし	R >> 3
R_C6000_SBR_GOT_U15_W_TLSMOD	符号なし	[32,8,15]	ZE(F<<2)	GOT(TLSMOD(S)) + A - B	あり	R >> 2
R_C6000_SBR_GOT_U15_W_TBR	符号なし	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	あり	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_B	符号なし	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	あり	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_H	符号なし	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	あり	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_W	符号なし	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	あり	R >> 2

表 13-6. C6000 再配置操作 (続き)

再配置名	符号属性	フィールド [CS, 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_C6000_SBR_GOT_U15_W_TPR_D	符号なし	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	あり	R >> 2
R_C6000_SBR_GOT_L16_W_TLSMOD	符号なし	[32,7,16]	ZE(F<<2)	GOT(TLSMOD(S)) + A - B	なし	R >> 2
R_C6000_SBR_GOT_L16_W_TBR	符号なし	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	なし	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_B	符号なし	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	なし	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_H	符号なし	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	なし	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_W	符号なし	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	なし	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_D	符号なし	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	なし	R >> 2
R_C6000_SBR_GOT_H16_W_TLSMOD	符号なし	[32,7,16]	ZE(F<<2)	GOT(TLSMOD(S)) + A - B	なし	R >> 18
R_C6000_SBR_GOT_H16_W_TBR	符号なし	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	なし	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_B	符号なし	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	なし	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_H	符号なし	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	なし	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_W	符号なし	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	なし	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_D	符号なし	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	なし	R >> 18
R_C6000_TLSMOD	符号なし	[32,0,32]	F	TLSMOD(S)	なし	R
R_C6000_TBR_U32	符号なし	[32,0,32]	F	TBR(S)	なし	R
R_C6000_FPHEAD	なし	なし	なし	なし	なし	なし
R_C6000_NOCMP	なし	なし	なし	なし	なし	なし

13.5.3 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。

- 絶対再配置型 (R_C6000_ABS*) で使用される場合、参照は 0 に解決されます。
- ベース相対再配置型 (R_C6000_SBR*) で使用される場合、参照は静的ベース アドレス (B) に解決されます。

R_C6000_PCR_S21 再配置で使用され、かつ再配置対象の命令が次の形式を持つ場合:

```
      B.S2   sym      ; R_C6000_PCRS21
```

その命令は次のように置き換えられます。

```
      B.B2   B3
```

その他すべてのケースは、ABI に準拠しません。

注

この仕様の他の箇所で要求されているように、弱シンボルが解決され、かつ 21 ビットの PC 相対アドレスがターゲットの宛先に到達できない場合、リンカはトランポリンを生成して再配置を実装する必要があります。

Chapter 14

ELF プログラムのロードと 動的 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 (動的リンクの場合は複数の ELF ファイル) 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。

動的リンクとは、個別に構築されたコンポーネントを組み合わせてプログラムを構成し、ロード時にリンクや再配置を行うことで、それらを複数の実行可能ファイル間で共有可能にする仕組みの総称です。

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。たとえば、1 つのプロセスのみを実行するベアメタル プラットフォームでは、動的リンクやロードは必要でも、位置の独立性や共有オブジェクトは不要なことがあります。

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、C6000 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。さらに、この節ではベアメタル動的リンク モデルという特定のプロファイルを定義しています。

この ABI で定義されるもう 1 つのプロファイルは Linux モデルです。Linux 向けの System V ABI 標準に対するプロセッサ固有の補足については、「Chapter 15.」を参照してください。

14.1 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

C6000 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セグメント型を 1 つ定義します。

名称	値	コメント
PT_C6000_PHATTR	0x70000000	拡張セグメント属性

PT_C6000_PHATTR セグメント タイプは、そのセグメントが、プログラム内の他の PT_LOAD セグメントに関する追加的な記述情報を含んでいることを示します。このセグメントには、SHT_TI_PHATTRS 型のセクションが 1 つ含まれています。プログラム ヘッダー属性の詳細については、Chapter 19 に記載されています。

p_vaddr、p_paddr

C6000 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

C6000 に定義されるプロセッサ固有のセグメント フラグが 1 つあります。

名称	値	コメント
PF_C6000_DPREL	0x10000000	DP 相対アドレッシングを使用してアクセス

PF_C6000_DPREL フラグは、DP 相対アドレッシングを使用してアクセスされるセグメントを識別します。そのため、リンク後の配置による制約を受けます。位置独立のモジュールは、通常 DP 相対アドレッシングに対して動的再配置を含みません。複数の DP 相対セグメントが存在する場合、それらの DP に対する位置 (したがって互いの相対的な位置) を維持する必要があります。このフラグは、動的ローダまたはその他のリンク後エージェントが、これらのセグメントの割り当てを調整できるように、セグメントを識別するためのものです。

TI ツールチェーンでは、いくつかの補助的なセグメント属性も使用されます。使用可能なフラグ数が限られているため、追加のセグメント属性を表現する代替手段として、プログラム ヘッダー属性テーブルが定義されています (Chapter 19 を参照)。

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、C6000 における p_align の意味および設定は未定義です。

14.1.1 ベース アドレス

位置独立コードは、プログラム ヘッダーの p_vaddr フィールドで指定されたアドレスである必要はなく、ロード時の再配置を必要とせずに任意のアドレスにロードして実行できます。ただし、セグメント同士が相対オフセットを用いて互いを参照している場合、実際のロード先が p_vaddr に指定された場所と異なっていても、その相対位置は維持されなければなりません。System V ABI では、セグメント間の指定アドレスと実際のアドレスのディスプレイスメントを *ベース アドレス* と呼びます。

位置独立なセグメントを異なるアドレスにロードできる範囲は、プラットフォームに依存します。ただし、次のような普遍的なルールがあります。

- 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。
- PHA_BOUND 属性を持つセグメントは、指定されたアドレスにロードする必要があります。

14.1.2 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、C6000 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 DP 相対セグメント、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。

14.1.3 バインドされたセグメントと読み取り専用セグメント

セクション 19.2 で説明したように、セグメントに追加のプロパティを注釈として付与するメカニズムがあります。このメカニズムは、ROM ベースのセグメントに適用されるプロパティを表すために使用されます。

PHA_BOUND 属性でマークされたセグメントは、指定されたアドレスにバインドされており、後続の再リンク、動的リンク、または動的ロードの過程で変更できません。このプロパティは、セグメント自体が ROM に配置されている場合、または ROM 内のコードから絶対アドレスを使用して参照される場合に適用されます。

PHA_READBONLY 属性でマークされたセグメントは、その内容がロックされており、再配置やその他の後続の変更の影響を受けません。このプロパティは、ROM に配置されたセクションに適用されます。動的ローダは、この情報をヒントとして使用し、そのようなセグメントの再配置処理を回避できます。

PHA_READONLY セグメントと、プログラム ヘッダのセグメント権限が PF_R (読み取り専用) であるセグメントとの違いは、通常、PF_R セグメントはローダによって変更できますが、プログラム自体から変更できないのに対し、PHA_READONLY セグメントはどちらからも変更できないという点です。

14.1.4 スレッドローカル ストレージ

スレッドローカル ストレージ (TLS) は、プログラムが静的ストレージ期間を持つスレッド固有の変数を定義できるようにするストレージ クラスです。TLS 変数または thread-local 変数は、スレッドごとに 1 回だけインスタンス化されるグローバル / 静的変数です。スレッドローカル ストレージの詳細については、Chapter 7 を参照してください。

C6000 EABI は TLS をサポートしますが、これはランタイム オペレーティング システムのスレッド ライブラリが __c6xabi_get_tp() 関数および TLS サポートのその他の側面を実装しているかどうかに依存します。

スレッドローカル変数は、ELF オブジェクト ファイルおよびモジュールにおいて静的データと同様に表現されます。ただし、ELF では再配置可能ファイルの場合、これらのスレッドローカル変数を SHF_TLS フラグ付きのセクションに割り当てる必要があります。初期化済みのスレッドローカル ストレージには .tdata を、未初期化のものには .tbss をセクション名として用いることが規定されています。これらのセクションは読み取り/書き込みの権限を持ちます。

モジュールでは TLS セグメントを PT_TLS というセグメント タイプによって示すことが ELF によって求められています。このセグメントは読み取り専用です。PT_TLS セグメントは TLS イメージです。

スレッドローカル シンボルはシンボル タイプ STT_TLS を持ちます。

14.2 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 C6000 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 14-1、表 14-2、表 14-3 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。「DL のみ」とラベル付けされた列は、動的リンクまたはロードを使用するシステムにのみ適用される手順を示します。

表 14-1. 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		DL のみ
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 14.1 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。	
4.	依存ライブラリのプロセス イメージを作成します。依存ライブラリは、動的セクション内の DT_NEEDED エントリによって識別されます。ライブラリは、ターゲット プロセッサ、ABI、OS、DSBT インデックスに関して互換性があるか確認する必要があります。	✓
5.	このモジュールおよびすべての依存ライブラリに対して DSBT インデックスを割り当てます。インデックスは、実行可能ファイルおよびそのすべてのライブラリの間で一意でなければなりません。あるライブラリのインスタンスは、複数のプログラムで共有される場合でも、1 つのインデックスのみを持つ必要があります。セクション 6.7 を参照してください。	✓
6.	インポートされたシンボルとエクスポートされたシンボルの間でシンボリック参照を解決します。動的リンクを持つシンボルは、動的セクション内の DT_SYMTAB タグによって識別される動的シンボル テーブルに表示されます。可視性が STV_DEFAULT のエクスポートされたシンボルは、親ファイルの定義によってプリエンプトされることがあります。バージョン情報を持つシンボル (動的セクション内の DT_SYMVER タグによって識別) については、参照が適切な定義と一致するようにローダが保証する必要があります。	✓
7.	必要に応じて再配置を実行します。ロード時の再配置は、動的セクション内の DT_REL または DT_RELA タグによって示されます。再配置は、セクション 13.5 に記載された仕様に従って処理されます。	✓
8.	実行可能ファイルおよび依存ライブラリの DSBT エントリを初期化します。この手順は 2 つの部分からなります。最初に、現在の実行可能ファイルの DSBT は、すべてのロード済みモジュール (自身を含む、インデックス 0) の静的ベース アドレスで初期化される必要があります。次に、他のすべてのロード済みモジュールの DSBT は、このモジュールのベース アドレスで更新され、このモジュールにステップ 5 で割り当てられたインデックスに設定される必要があります。	✓
9.	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 14-2. 実行環境を初期化する手順

ステップ		DL のみ
10.	SP を設定します。SP (B15) 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。	
11.	DP を設定します。DP (B14) は、任意の DP 相対セグメントの最小アドレスに対応するシンボル _ _C6000_DSBT_BASE の値に設定されなければなりません。	

表 14-2. 実行環境を初期化する手順 (続き)

ステップ		DL のみ
12.	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 18 に記載されています。	
13.	事前初期化呼び出しを実行します。これらは依存ライブラリの初期化関数よりも前に実行されるよう定義された初期化関数への呼び出しです。事前初期化呼び出しは、System V ABI で規定されているように、動的セクション内の DT_PREINIT_ARRAY タグによって識別されます。	✓
14.	依存ライブラリに対して、System V ABI の「初期化関数および終了関数」に関するセクションで定義された順序に従い、初期化呼び出し (ステップ 15) を再帰的に実行します。	✓
15.	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。これらは依存ライブラリの初期化の *後に* 実行されます。初期化関数へのポインタはテーブルに格納されます。動的情報を持つファイルでは、そのテーブルは DT_INIT_ARRAY タグまたは DT_INIT タグによって識別されます。その他のファイルでは、テー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。	
16.	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 14-3. 終了手順

ステップ		DL のみ
17.	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。	
18.	System V ABI の「初期化関数および終了関数」のセクションで定義された順序に従い、依存ライブラリの終了呼び出し (ステップ 19) を再帰的に実行します。	✓
19.	現在のモジュールの終了関数を呼び出します。これは、DT_FINI タグまたは DT_FINI_ARRAY タグによって識別されます。	✓

14.3 動的リンク

動的リンクは、プログラムが個別に構築されたコンポーネントで構成されるようにする関連メカニズムのセットです。メカニズムは以下で構成されます。

- **リンケージ メカニズム**: 個別にリンクされたオブジェクト間の参照をサポートします。これらは、主に動的セクションと、動的シンボル テーブルや動的再配置などの関連するサブコンポーネントで構成されます。
- **共有メカニズム**: コードを共有する各アプリケーションでは、異なる場所にデータのプライベート コピーを保持できます。MMU を使用するシステムは通常、仮想アドレスから物理アドレスへの変換に依存します。C6000 には MMU がなく、Chapter 6 で規定されているデータ セグメント ベース テーブルと呼ばれるメカニズムに依存しています。
- **アドレッシング メカニズム**: リンケージと共有をサポートします。これらも、一般には Chapter 6 で規定されています。

システムでは、特定の要件に応じてメカニズムのサブセットを使用できます。たとえば、1 つのプロセスのみを実行するベアメタル プラットフォームでは、動的なリンクとローディングが必要ですが、位置の独立性や共有オブジェクトは必要ありません。

ABI では現在、異なるレベルの機能を持つ 2 つの特定のプロファイルを定義しています。1 つ目は、セクション 14.4 で説明されているベアメタル動的リンク モデルです。もう 1 つは、Chapter 15 で説明されている Linux モデルです。

14.3.1 プログラム インタプリタ

セクション 14.2 で説明されているように、プログラムのロードは外部のエージェントによって実行されます。Linux やおそらく他の OS ベースのシステムでは、この機能を実行するエージェントは、プログラム ヘッダーの PT_INTERP タグとして実行可能ファイル自体に格納されます。通常これは動的ローダ (ld.so など) です。

ベアメタル実行可能ファイルはインタプリタに依存せず、システム側がプログラムのロード方法を把握している必要があります。ベアメタル動的実行可能ファイルは、PT_DYNAMIC セグメントに動的情報を含むことがありますが、PT_INTERP エントリに持つことはありません。

14.3.2 動的セクション

System V ABI の規定に従い、動的リンクされたプログラムのプログラム ヘッダーには PT_DYNAMIC 型のエントリが含まれます。このエントリは .dynamic という特別なセクションを指し、セクション タイプは SHT_DYNAMIC で、動的リンクおよびロードに関する情報を保持します。この動的セクションは、動的シンボル テーブル セクションや動的再配置セクションなどの他のセクションを参照しています。これらはまとめて *動的情報* と呼ばれます。

動的情報は、プラットフォームごとの規則によって、プログラムのロード可能イメージ (1 つ以上の PT_LOAD セグメント) に含まれる場合と含まれない場合があります。動的情報がロード可能ではない場合、オブジェクト コンポーネントを参照する動的タグは仮想アドレスではなくファイル オフセットで表されます。

動的セクション自体は System V ABI で規定されています。C6000 固有の動的タグが定義されており、その一覧は表 14-4 に記載されています。

表 14-4. C6000 動的タグ

名称	値	d_un	実行可能ファイル	共有オブジェクト
DT_C6000_GSYM_OFFSET	0x6000000D	d_val	オプション	オプション
DT_C6000_GSTR_OFFSET	0x6000000F	d_val	オプション	オプション
DT_C6000_PRELINKED	0x60000011	d_val	オプション	オプション
DT_C6000_DSBT_BASE	0x70000000	d_ptr	必須 (DSBT モデルの場合)	必須 (DSBT モデルの場合)
DT_C6000_DSBT_SIZE	0x70000001	d_val	必須 (DSBT モデルの場合)	必須 (DSBT モデルの場合)
DT_C6000_PREEMPTMAP	0x70000002	d_ptr	オプション	オプション
DT_C6000_DSBT_INDEX	0x70000003	d_val	オプション	オプション

グローバル シンボル マーカー タグ

動的シンボル テーブル内のシンボルは、ローカルまたはグローバルとして指定されます。ローカル シンボルは、それを含むモジュールの再配置にのみ必要であり、動的シンボル解決には関与しないため、モジュールを再配置した後に動的ローダがそれらを破棄できます。動的シンボル テーブル内でローカル シンボルをグローバル シンボルよりも前にまとめておくことで、ベアメタル プラットフォームにおいて動的ローダが効率的に処理できるようになります。
DT_C6000_GSYM_OFFSET タグは、動的シンボル テーブル (.dynsym) 内の最初のグローバル シンボルのオフセットを保持します。DT_C6000_GSTR_OFFSET タグは、動的文字列テーブル (.dynstr) 内の最初のグローバル シンボル名のオフセットを保持します。

ローカル シンボルは、タグで示された位置の後に依然として存在する場合がありますが、指定された位置より前にグローバル シンボルが存在することはありません。

DT_C6000_PRELINKED

このタグは、ベアメタル ロード モジュールでのみ使用されます。このタグは、ファイルに仮想アドレスが割り当て済みであることを示します。割り当てはプレリンカや類似ツールによって行われる場合があります。その値はタイムスタンプを意味します。

DT_C6000_PRELINKED は Linux のプレリンカで使用される DT_GNU_PRELINKED タグと似ていますが、ベアメタルでのプレリンキングは完全に同一ではないため、別のタグが定義されています。

DSBT タグ

これらのタグは、位置の独立性を実現するために DSBT モデルを利用するロード モジュールで使用されます (セクション 6.7 を参照)。DT_C6000_DSBT_BASE タグは、データ セグメントの静的リンクされた位置を示し、_ _c6xabi_DSBT_BASE シンボルに対応します。ロード モジュールではシンボル テーブルを必ずしも含む必要がないため、その値がこのタグに保持されます。

DT_C6000_DSBT_SIZE タグは、DSBT テーブル用に予約されているサイズを示します。すべてのロード モジュールは、その中で最大の DSBT インデックス番号以上のサイズを持つテーブルを備えていなければなりません。モジュールが小さすぎるテーブルや大きすぎるインデックスでロードされた場合、ローダはそのモジュールをロードできません。

セクション 6.7 に記載されているように、モジュールの DSBT インデックスはリンカによって静的に、またはローダによって動的に割り当てられます。ロード モジュールに静的なインデックスが割り当てられている場合は、DT_C6000_DSBT_INDEX タグにその値が記録されます。同じプロセス内で他の動的リンク モジュールが同じインデックスを使用することはできません。動的にインデックスが割り当てられるモジュールでは、このタグは省略されます。

DT_PREEMPTMAP

このタグは、静的バインディングを利用してシンボルのプリエンプションを事前に解決するプラットフォーム向けに、プリエンプション マップのファイル オフセットを保持します。

DT_PLTGOT

このタグは、グローバル オフセット テーブル (GOT) の仮想アドレスを保持します。

動的再配置タグ

System V ABI では、オブジェクト ファイルにおける動的再配置の位置と種類を特定する、以下の 7 種類のタグが定義されています。

- DT_RELA、DT_RELASZ - 動的再配置の開始位置とサイズを示します。
- DT_PLTREL - テーブルの DT_JMPREL セクションに含まれる再配置の種類を示します。C6000 では、この値は必ず DT_RELA になります。
- DT_JMPREL、DT_PLTRELSZ - DT_RELA テーブルの一部範囲を示し、その範囲には PLT エントリ経由でのみ参照されるシンボルの再配置が含まれます。
- DT_REL、DT_RELSZ - これらのタグは C6000 では使用されません。

基本仕様では、DT_RELA と DT_RELASZ で定義される動的再配置に、DT_JMPREL と DT_PLTRELSZ で定義される PLT 固有の再配置が含まれるかどうかが明確にされていません。C6000 ABI では、DT_RELA テーブルに DT_JMPREL テーブルを含めるものとしています。

14.3.3 共有オブジェクトの依存関係

実行可能ファイルはライブラリに依存することがあり、そのライブラリがさらに他のライブラリに依存することもあります。これらの依存関係は、動的セクション内の DT_NEEDED エントリにエンコードされます。実行可能ファイルまたはライブラリが別のライブラリに依存する場合、依存するライブラリは、参照元の動的セクション内の DT_NEEDED エントリによって名前が付けられます。動的リンカは、依存するライブラリを見つけて、それをロードする必要があります (セクション 14.2 を参照)。

Linux などの一部のプラットフォームには、依存ライブラリを探索するための標準化された検索メカニズムがあります。たとえば環境変数 LD_LIBRARY_PATH がその一例です (System V ABI に記載)。ベアメタル プラットフォームには標準化されたガイドラインは存在しません。いずれの場合も、シンボルの解決は System V ABI で説明されている幅優先の方法で進められます。

14.3.4 グローバル オフセット テーブル

複数の実行可能ファイル間で共有されるライブラリを含むいくつかのコンテキストでは、位置独立アドレッシングが必要となります。コード セグメントに位置依存のアドレスを埋め込まないように、そのようなアドレスは、各静的リンク ユニットのデータ セグメントの一部であるグローバル オフセット テーブル (GOT) と呼ばれるテーブルに生成されます。オブジェクトに直接アクセスする代わりに、プログラムは GOT から変数のアドレスを読み取り、その変数を間接的に参照します。GOT はデータ セグメントの一部であり、常に静的リンク時に固定されたオフセットを使用して DP 相対でアドレッシングされます。これは、コンパイラによって出力される特別な GOT 生成再配置に応じてリンカによって生成されます。GOT 内のアドレスは、動的リンク時に、実際にそのシンボルが配置されたアドレスが判明した時点でパッチされます。

コンパイラは特別な再配置エントリを使用して GOT を参照します。静的リンカは、この特別な再配置に応じてテーブルそのものを生成します。テーブル エントリ自体には (動的) 再配置が含まれており、動的ローダはこれを用いて、参照されるオブジェクトの最終的に解決されたアドレスをパッチします。GOT ベースのアドレッシングについては、セクション 6.6 を参照してください。GOT エントリに適用される再配置については、セクション 13.5.1 を参照してください。

ベアメタル モデルを使用する実行可能ファイルおよびライブラリは、GOT ベースのアドレッシングを必要とする場合もあれば、必要としない場合もあります。

14.3.5 プロシージャ リンケージ テーブル

セクション 6.5 で説明しているように、プロシージャ リンケージ テーブル (PLT) は、あるロード モジュールから別のモジュール内のインポート関数への呼び出しを接続するスタブの集合です。インポート関数のアドレスは静的リンク時には不明なため、静的リンカは代わりにその関数を呼び出すために位置に依存しないスタブを生成し、元の呼び出しがこのスタブを経由するようにパッチを適用します。このスタブは、呼び出し先の動的リンク時アドレスに従ってロード時に再配置されます。

PLT はコード セグメントの一部です。PLT エントリは、位置独立性が必要かどうかに応じて、呼び出し先を指定する際に絶対アドレッシングまたは GOT ベースのアドレッシングを使用できます。

14.3.6 プリエンプション

プリエンプションは、ライブラリ内で定義されたシンボルが *それより前にある* 実行可能ファイルまたはライブラリ内の定義によってマスクされるときに発生します。ここでの「それより前にある」とは、実行可能ファイルとその依存ライブラリによって構成される依存関係ツリーによって確立される幅優先順位による順序を指します。

シンボルがプリエンプトされるのは、そのシンボルへのすべての参照 (それを定義しているモジュール内のものも含む) が GOT ベースのアドレッシングを使用している場合に限られます。動的リンカは、GOT の適切なスロットに優先するシンボルのアドレスをパッチするだけで、プリエンプションを実行します。

14.3.7 初期化と終了

ロード モジュールでは、参照または呼び出される前に、モジュール内の静的オブジェクトに対する C++ コンストラクタなどの初期化コードの実行が必要になる場合があります。同様に、モジュールが終了する際には、デストラクタなどの終了コードが必要となる場合があります。

モジュールは、System V ABI に規定されているように、動的セクション内の DT_INIT、DT_INIT_ARRAY、DT_PREINIT_ARRAY、DT_FINI、DT_FINI_ARRAY のエントリを使用して、必要な初期化および終了を指定します。

初期化の場合と同様に、ローダまたは実行環境は、モジュール依存関係によって課される順序の制約に従って、終了関数を実行する責任を負います。

14.4 ベアメタル動的リンク モデル

ベアメタル動的リンク モデルは、特定のオペレーティング システムの規則に縛られず、個別にリンクされたコンポーネントを必要とするアプリケーション向けのプラットフォーム非依存モデルです。DSBT モデルおよび GOT ベースのアドレッシングはオプションで除外することができ、その場合、動的リンクによる実行時の性能低下はほぼゼロになりますが、配置およびアドレッシング方式により多くの制約が生じます。

最小構成では、DSBT を使用せず、位置独立性も持たない形で、モデルはライブラリの *動的リンクおよびロード* をサポートしますが、異なる実行可能ファイル間でのライブラリ *共有* はサポートされません。言い換えると、GOT および DSBT を使用しない場合、ベアメタル動的リンク モデルは 1 つの静的にリンクされたベアメタル実行可能ファイルのアドレッシング方式をそのまま使用し、柔軟性を犠牲にして大きな性能上の利点を得ます。

より柔軟性が必要な場合は、DSBT をオプションで有効にすることができ、その結果、個別に構築されたライブラリが独自のデータ セグメントを持つことが可能となります。同様に、位置の独立性をオプションで有効にすることもでき、その場合、ライブラリを複数の実行可能ファイル間で共有できるようになります。

14.4.1 ファイル形式

プログラムは、実行可能ファイル (ファイル形式 ET_EXEC) および依存ライブラリ (ファイル形式 ET_DYN) として個別にリンクされる場合があります。このモデルでは、ファイルはそれぞれ *ベアメタル動的実行可能ファイル* および *ベアメタル動的ライブラリ* と呼ばれます。これらのファイルには、PT_DYNAMIC プログラム ヘッダーを介してダイナミック セクションを通じて参照される通常の動的情報が含まれます。プログラムとそのライブラリは、必要に応じてロード時に動的に再配置できます。

14.4.2 ELF 識別

このモデルに準拠する実行可能ファイルと共有オブジェクトは、ELF ヘッダの EI_OSABI フィールドで ELFOSABI_C6000_ELFABI で識別されます。再配置可能なファイルは ELFOSABI_NONE として識別されます。

14.4.3 可視性とバインディング

グローバル シンボルのデフォルトの可視性は STV_INTERNAL です。つまり、インポートまたはエクスポートされるシンボルは、そのように明示的に宣言される必要があります。シンボルのプリエンプションはサポートされていません。共有オブジェクト間において、曖昧なリンケージ (vtbls、rtti 型情報など) を持つシンボルについては、One Definition Rule は適用されません。ベアメタル モデルでは、強制的な静的バインディングが使用されます。つまり、リンカはインポートされた当該の参照を静的リンク中に定義に強制的にバインドします。

動的シンボル テーブルでは、STV_DEFAULT の可視性を持つシンボルはすべて STB_GLOBAL としてマークされます。つまり、デフォルトの可視性を持つ弱シンボルは、グローバル シンボルに変換されます。これはローダの実装を簡単にするためです。

14.4.4 データ アドレッシング

ベアメタル動的リンク モデルにおいて、DSBT モデルの使用は任意です。DSBT がない場合、プログラムは 1 つの DP を持ち、それは実行ファイルのデータ セグメントのベース アドレス (最初の DP 相対セグメント) を指します。実行可能ファイル自身は、自身のデータを参照するためにニア DP 相対アドレッシングを使用できます。ライブラリ内のデータは、ファー アドレッシング モード (ファー DP 相対または絶対) を使用してアドレス指定されなければなりません。これは、実行可能ファイルがインポートされたデータをアドレッシングする場合と、ライブラリが自身のデータをアドレッシングする場合 (DP は実行ファイルに属するため) に適用されます。DSBT がない場合、ライブラリは .bss、.neardata、.rodata のセクションを持つことができません。

DSBT が有効化されると、個別にビルドされた各コンポーネントは自身の DP 相対セグメントを持つことができます。

ベアメタル動的リンク モデルにおいては、GOT ベースのアドレッシングによる位置独立データもまた任意です。GOT ベースのアドレッシングを使用しない場合、インポートされたアドレスへの参照は、コード セグメント内に絶対アドレスとして、あるいは、DSBT 以外の実行可能ファイルに対してはオプションで、実行可能ファイルの DP からのオフセットとして符号化されます。そのようなコードは、ライブラリのデータ セグメントの個別の事前処理コピーにアクセスできないため、個別に

リンクされたライブラリはサポートされますが、共有ライブラリはサポートされません。位置の独立性を持たないコードは、ロード時に修正を必要とする可能性があります。

リンカは、DSBT および GOT モデルの一貫した使用を強制します。

14.4.5 コードのアドレッシング

インポートされた関数への呼び出しは、コンパイラまたは静的リンカによって生成される PLT エントリを経由して行われます。遅延バインディングには対応していません。PLT は、必要な位置の独立性の度合いに応じて、絶対指定、PC 相対指定、GOT ベースのアドレッシングを使用して関数を参照します。

14.4.6 動的情報

動的タグでは、動的情報を参照するために (System V ABI で指定された仮想アドレスではなく) ファイル オフセットを使用します。動的セグメントはプログラムのロード イメージの一部ではありません。つまり、PT_DYNAMIC と関連するセクションはどの PT_LOAD セグメントにも含まれていません。

表 14-5 ではベアメタル ダイナミック リンク モデルの特性をまとめており、2 つのベアメタル ファイル タイプを比較しています。

表 14-5. ベアメタル ダイナミック リンク ファイル

特性	ベアメタル動的実行可能ファイル	ベアメタル動的ライブラリ
ELF ファイル タイプ (e_type)	ET_EXEC	ET_DYN
ELF 識別 (e_ident)	ELFOSABI_C6X_ELFABI	
ロード可能な動的セクション	なし	
独自のデータをアドレス指定	.bss、.neardata、および .rodata を指定でき、ニア DP 相対アドレッシングを使用してアクセス可能	DSBT あり:実行可能ファイルと同じ DSBT なし:ファー (DP 相対、絶対、または GOT)
インポートされたデータのアドレス指定	ファー (DP 相対、絶対、または GOT)	
PT_DYNAMIC セグメントあり	あり	
Has PT_INTERP	なし	
シンボルをインポート/エクスポート可能	○ (明示的なディレクティブを使用)	
ロード時に再配置可能	省略可	あり
エントリ ポイント	必須	オプション

Chapter 15

Linux ABI

このセクションでは、C6000 Linux ベースシステムのアドレッシング、動的リンク、プログラムのロードに関する規則について説明します。可能な限り、他の組込み MMU レス Linux システムの規則に従うことを方針としています。

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。このセクションは、この ABI の他の部分とともに、特に C6000 の Linux で動作するプログラム向けに、その標準に対するプロセッサ固有の補足情報となっています。

これらの規則はユーザー空間アプリケーション プログラムに適用されます。カーネルは独立しており、実装固有のガイドラインに従うことがあります。

15.1 ファイル形式

プログラムは、実行可能ファイル (ファイル形式 ET_EXEC) および共有ライブラリ (ファイル形式 ET_DYN) で構成されるように個別にリンクされる場合があります。これらのファイルには、PT_DYNAMIC プログラム ヘッダーを介してダイナミック セクションを通じて参照される通常の動的情報が含まれます。プログラムとそのライブラリは、必要に応じてロード時に動的に再配置できます。

共有ライブラリは位置独立であることが要求されます。実行可能ファイルは位置独立であってもなくてもかまいません。位置依存の実行可能ファイルは、ロード時にコード セグメントへの再配置を必要とし、システム リソースを非効率に消費する可能性があります。

15.2 ELF 識別

Linux ABI に準拠する実行可能ファイルと共有オブジェクトは、ELF ヘッダの EI_OSABI フィールドで ELFOSABI_C6000_LINUX で識別されます。再配置可能なファイルは ELFOSABI_NONE として識別されます。

ELFOSABI_Linux の代わりにベンダー固有の ELFOSABI_C6000_Linux 値を指定する理由は、MMU レス Linux (uClinux) に対応するこの ABI バリアントを将来の MMU 対応バリアントと区別するためです。

15.3 プログラム ヘッダーとセグメント

以下はプログラム ヘッダーおよびセグメントです。

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。Linux ABI においては、p_align は 0x1000 に設定されるものと規定されています。

PT_INTERP セグメント

PT_INTERP セグメントには、動的ローダを含むオブジェクト ファイルの名前が格納されます。この本ドキュメントで説明される ELF 実行可能ファイルでは、インタプリタは通常 ld.so です。

読み取り専用セグメント

共有オブジェクトおよび実行可能ファイルには、モジュールのプログラム コードと共有可能な定数を含む、Read+Execute 権限を持つ PT_LOAD セグメントが必要です。共有可能な定数とは、書き込み可能ではなく、その値がアドレスで構成されない任意のオブジェクトを指します。このセグメントには、ファイル ヘッダー、PT_INTERP、PT_PHDR、PT_DYNAMIC、PT_NOTE (存在する場合)、PT_PHATTR セグメントなど、プログラムのロードおよび実行に必要な ELF 構造も含まれます。

位置依存型実行可能ファイルには、内容が未定義の追加の読み取り専用または Read+Execute セグメントが含まれる場合があります。このようなセグメントが複数ある場合、それらの間での PC 相対参照を持つことは許可されません。ローダがそれらを再配置する場合、相互の相対位置を保持する必要はありません。

データ セグメント

共有オブジェクトと実行可能ファイルには、モジュールの DSBT、GOT、読み書き可能データを含む、Read+Write 権限を持つ PT_LOAD セグメントが 1 つ必要です。このセグメントは DP 相対アドレッシングを使ってアドレッシングされるため、PF_C6000_DPREL フラグが付けられます。

ELF では、セグメント内の初期化されていないデータはすべての初期化データの後に配置される必要があります。しかし、DP 相対セグメントに初期化されていないニア データ (.bss など) と初期化済みのファー データ (.fardata など) の両方が含まれる場合、初期化されていないデータを先に配置しないと DP の範囲に収まらない場合があります。この場合、リンカはセグメント内の初期化されていない領域を 0 で埋める必要があります。

共有オブジェクト実行可能ファイルには、追加の Read+Write セグメントが存在する場合があります。位置の独立性を確保するため、これらのセクションは GOT を用いたアドレス指定でのみアドレッシングする必要があります。位置依存型実行可能ファイルでは、絶対アドレッシングを使用できます。

スタック セグメント

C6000 Linux ABI は、実行可能ファイルに対して最小限のスタック割り当てをツールチェーンが指定できるようにするために、追加のセグメント型を定義して、一般的な規則に従います。

名称	値	コメント
PT_GNU_STACK	0x6474E551	スタック サイズおよび権限

p_flags メンバーは、スタックを含むセグメントに対する権限を指定し、スタックを実行可能とするかどうかを示すために使用されます。

このヘッダーが存在しない場合、スタックのサイズと権限は未指定のままとなります。

バインドされたセグメントと読み取り専用セグメント

Linux の実行と可能ファイルおよび共有オブジェクトには、セクション 14.1.3 で説明されているように、バインドまたは読み取り専用としてマークされたセグメントを含めてはなりません。

15.4 データ アドレッシング

共有ライブラリは完全に位置独立でなければなりません。すなわち、読み取り専用セグメントに対するロード時の再配置は存在しません。可視性が STV_DEFAULT の任意のオブジェクトは GOT を通してアドレス指定されなければなりません。その他のすべての静的データは DP 相対でアドレス指定されなければなりません。

実行可能ファイルは、オプションで位置独立としてビルドできます。実行可能ファイルは、STV_DEFAULT 変数に対して GOT ベースのアドレッシングを使用せずに済むよう、セクション 15.9 に記載されている import-as-own プリエンプション メカニズムを使用できます。

位置依存の実行可能ファイルは、位置独立アドレッシングと位置依存 (絶対) アドレッシングの組み合わせを使用できます。絶対アドレッシングを使用する実行可能ファイルは、ロード時に再配置を必要とします。

15.4.1 データ セグメント ベース テーブル (DSBT)

Linux の実行可能ファイルおよび共有オブジェクトは、セクション 6.7 に記述されている DSBT モデルに準拠する必要があります。ニア DP セグメントには、プログラムを構成するすべてのモジュールの中で最大の DSBT インデックスと同じ数以上のエントリを持つ DSBT テーブルが含まれている必要があります。ライブラリ ベンダ間の一貫性を保つため、ABI は DSBT テーブルのデフォルト サイズを 64 エントリに標準化しています。DSBT テーブルの存在およびサイズは、セクション 14.3.2 に規定されている C6000 固有の動的タグによって示されます。

DSBT インデックス 0 は実行可能ファイル用に予約されています。DSBT インデックス 1 はプログラム インタープリタ用に予約されています。ライブラリは 2 から始まるユニークな DSBT インデックスが静的に割り当てられます。位置の独立性の規則を満たすために、DSBT インデックスの動的再割り当てはサポートされていません。

15.4.2 グローバル オフセット テーブル (GOT)

セクション 6.6 で説明されているとおり、コードおよびデータの位置の独立性はグローバル オフセット テーブル (GOT) を通じて実現されます。GOT はデータ セグメントの一部であり、常に静的リンク時に固定されたオフセットを使用して DP 相対でアドレッシングされます。GOT は動的に割り当てられるアドレスを格納する 4 バイト *スロット* で構成されます。Linux 実行可能ファイルまたは共有オブジェクトには、少なくとも 2 つのスロット (8 バイト) を持つ GOT が必要です。GOT エントリは、動的シンボルを参照する動的再配置でマークされます。GOT エントリは静的リンカによって次のように初期化されます。

- R_C6000_JUMP_SLOT 再配置でマークされた GOT エントリは、遅延バインディング リゾルバ スタブのアドレスで初期化されます (セクション 15.6 を参照)。
- その他のすべての GOT エントリは 0 に初期化されます。

静的リンカは、遅延バインダが使用できるように GOT 内の最初の 2 つのスロットを予約する必要があります。セクション 15.6 を参照してください。

15.5 コード アドレッシング

インポートされた関数、またはインポートされた可能性のある関数を呼び出す場合、コンパイラまたはリンカは セクション 6.5 で説明している Procedure Linkage Table Entry と呼ばれるスタブを生成します。

PLT を介したパッチ適用が必要な呼び出しは、次のすべての条件を満たす再配置型でマークされます。

- 再配置型が R_C6000_PCR21 である。
- 参照されるシンボルの表示設定が STV_DEFAULT である。
- 参照されるシンボルの型が STT_FUNC か STT_NONE である。

実行可能ファイルの場合にはさらに次の条件が適用されます。

- 呼び出しを含むスタティック リンク ユニットでシンボルが未定義である。

共有オブジェクトまたは位置非依存実行ファイル内の PLT エントリでは、呼び出し先にアドレスを指定するために、位置非依存 (GOT ベース) アドレッシングを使用する必要があります。この場合、PLT エントリは セクション 15.6 で説明されている遅延バインディング規則に従う必要があります。つまり、PLT の最初の命令では、呼び出し先の GOT エントリを B0 にマークする R_C6000_JUMP_SLOT 再配置エントリのバイト オフセットをロードする必要があります。

位置依存実行ファイル内の PLT エントリでは絶対アドレッシングを使用できます。C6000 では、他のアーキテクチャに共通する規則を採用していません。この規則では、関数のアドレスへの参照を PLT エントリに対して静的に解決できます。セクション 6.7.3 を参照してください。

15.6 遅延バインディング

大規模なプログラムの場合、ロード時のシンボル解決によってプログラムの起動時間が大幅に低下することがあります。遅延バインディングは、関数がプログラムによって実際に呼び出されるまで、その関数シンボルの解決を遅らせるメカニズムです。実際に呼び出される関数のみを解決すればよいため、起動時間が短縮され、結果として全体的なパフォーマンスが向上します。

一般的なアプローチは、PLT を介した最初の呼び出しが、動的リンカ内のレゾルバ関数を介して制御を誘導することです。これにより解決が実行され、その後の呼び出しが直接その関数自体に再ルーティングされます。

レゾルバには 2 つの引数が必要です。1 つ目は、現在のモジュール (参照を含むモジュール) を識別するモジュール ID です。モジュール ID の表現は ABI によって規定されておらず、ローダによって決定されます。第 2 引数は、ターゲット関数に対応する再配置エントリを指定します。再配置エントリは、ターゲット シンボルの名前と GOT 内の参照位置の両方を提供します。再配置エントリは、オブジェクト ファイル内で、そのファイルの .dynamic セクションにある DT_RELPLT タグのアドレスからのバイト オフセットによって指定されます。

これらはすべて呼び出し元の背後で行われるため、このメカニズムによって、標準的な関数呼び出しインターフェイスに影響を与えるあらゆる状態が保持される必要があります。特に、引数の受け渡しに使用されるレジスタや復帰アドレス レジス

タを妨害してはならず、変更した呼び出し先保存レジスタも保持する必要があります。通常の引数レジスタに妨害されないように、レゾルバの 2 つの引数は B0 と B1 に渡されます。

グローバル オフセット テーブル内の 2 つのスロットは、遅延バインディングを実装するために動的ローダが使用できるよう予約されています。GOT[0] は、レゾルバ関数のアドレスを保持するためにローダによって使用されます。GOT[1] は、モジュール ID を保持するために使用されます。

次の手順で、このメカニズムについて説明します。

1. 静的リンカは遅延バインディングの候補を特定します。候補とは、PLT エントリによってのみ参照される GOT エントリです。つまり、インポートされた関数の呼び出しにのみ使用されます。
2. 静的リンカは、特別なレゾルバ スタブを生成するか、ライブラリから取得します。この説明では、このスタブは PLT0 と呼びますが、ABI ではその名前や位置は指定されません。
3. 静的リンカは、候補の GOT エントリを PLT0 のアドレスで初期化し、それらを R_C6000_JUMP_SLOT 再配置でマークします。リンカは、DT_JMPREL タグでマークされた動的再配置テーブルのセクション内からこのような再配置を見つけます。
4. PLT エントリは、遅延解決で使用する追加命令を含めて生成されます。この命令はレゾルバの 2 つの引数のうちの第 1 引数、すなわち DT_REL[A] タグで示される動的再配置テーブルに対する R_C6000_JUMP_SLOT 再配置エントリのバイト オフセットをレジスタ B0 にロードします。続いて、通常の方法で GOT からターゲット アドレスをロードし、そのアドレスにジャンプします。ステップ 3 の初期化の結果、このジャンプが最初に行われると、制御が PLT0 に転送されます。
5. PLT0 は、レゾルバの 2 つの引数うちの第 2 引数、すなわち現在のモジュールを識別するローダ定義の値を GOT[1] から B1 にロードします。続いて、GOT[0] からローダのレゾルバ関数のアドレスをロードし、それらをテールコールします。
6. レゾルバ関数は 2 つの引数を使用して、モジュール ID で指定されたオブジェクト ファイル内から該当する動的再配置を見つけます。動的シンボル テーブル内でシンボルを検索して関数の実際のアドレスを取得し、GOT エントリをそのアドレスに置き換えます。最後にそのアドレスにジャンプして、事実上ターゲット関数をテールコールします。
7. 以後の呼び出しで PLT エントリに入ると、GOT は実際のアドレスで更新されているため、制御は直接その関数に渡されます。

遅延バインディング PLT エントリ

```
$sym$plt:
      MVK   reloc_offset(sym),B0   ;byte offset of GOT reloc entry from DT_RELPLT
      MVKH  reloc_offset(sym),B0
      LDW   *+DP($GOT(sym)),tmp    ;&PLT0 first time, &sym after that
      B     tmp
```

レゾルバ スタブ - PLT0

```
PLT0:
      LDW   *+DP($GOT(0)),tmp      ; address of resolver
      LDW   *+DP($GOT(4)),B1       ; module id
      B tmp                        ; tail-call resolver
```

グローバル オフセット テーブル

```
              ; $GOT(0) reserved, initialized to module id
              ; $GOT(4) reserved, initialized to &resolver function
              ; ...
              ; $GOT(sym) R_C6000_JUMP_SLOT initialized to &PLT0
              ; updated to &sym by resolver
```

15.7 可視性

Linux では、グローバル シンボルのデフォルトの可視性は STV_DEFAULT です。共有オブジェクトでは、STV_DEFAULT の可視性を持つ定義済みシンボルは他の定義によって置き換えられる可能性があるため、インポート シンボルと同じようにアドレッシングする必要があります。実行可能ファイルでは、import-as-own 規則 (セクション 15.9 を参照) により、STV_DEFAULT の可視性を持つ定義済み変数および未定義変数を STV_INTERNAL のようにアドレッシングすることができます。すなわち、DP 相対アドレッシングを使用します。

ツールチェーンは、デフォルトの可視性ルールを変更するベンダー固有のオプションまたは拡張を実装することができます。それらも影響を受けるシンボルの可視性フラグに標準値を使用して反映させる必要があります。

15.8 プリエンプション

Linux では、シンボルの解決に関して、動的リンクが静的リンクの動作を維持するという規則が採用されています。プリエンプションは、同じシンボルに対して複数の定義が存在する場合、すなわちライブラリ内で定義されたシンボルが、*それより前にある* 実行可能ファイルまたはライブラリ内の定義によってマスクされるときに発生します。ここでの「それより前にある」とは、実行可能ファイルとその依存ライブラリによって構成される依存関係ツリーによって確立される幅優先順位による順序を指します。

シンボルがプリエンプトされるのは、そのシンボルへのすべての参照 (それを定義しているモジュール内の参照も含む) が GOT ベースのアドレッシングを使用している場合に限られます。動的リンカは、GOT の適切なスロットに優先するシンボルのアドレスをパッチするだけで、プリエンプションを実行します。

15.9 Import-as-Own プリエンプション

Linux では、外部シンボルは通常 STV_DEFAULT の可視性を持ち、特に明示的に宣言されない限りプリエンプションの対象となります。これは通常、同じモジュール内で定義されたものを含む、ほとんどすべての変数参照に対して GOT ベースのアドレッシングが発生することになります。言い換えれば、Linux モジュールは、extern 変数へのすべての参照を、実際に該当しない場合でも、インポートされたものとして扱うことが要求されます。その結果生じる性能上のペナルティを回避するために、実行可能ファイルはそれを回避できる特別な規則を採用しています。

実行可能ファイルは、任意の変数参照を、まるでその変数が自分自身 (すなわち、実行可能ファイル内で定義されているもの) であるかのように扱うことができ、その場合コンパイラは効率的な DP 相対アドレッシングを生成できます。静的リンク時に、その変数が実際にはインポートされたものであると判明した場合、実行可能ファイル内にその変数の重複する定義が作られます。動的ロード時に、その重複した定義はライブラリ内の元の定義をプリエンプトし、初期化子がプリエンプトされた定義から新しい定義へコピーされます。

重複定義のサイズは、ソース定義の st_size フィールドによって指定されます。重複定義の最小整列は次のように与えられます。

- *max* を、ソース モジュール内で与えられたサイズのオブジェクトに必要となる可能性のある最大整列とします。この値は、オブジェクトのサイズ、Chapter 2 の整列要件、TAG_ABI_array_object_alignment ビルド属性で指定された整列に応じて決まります。
- *vaddr* にはソース モジュール内のオブジェクトの仮想アドレスを指定します。
- 重複オブジェクトの整列は、*vaddr* と *max* の最大公約数です。

(つまり、重複オブジェクトは、少なくとも元のオブジェクトと同等に整列され、必要に応じて最大の整列条件まで満たすように定義されます)。

プロセス イメージが作成されたときに元のシンボルに格納されていた任意の初期値は、重複オブジェクトに伝播される必要があります。R_C6000_COPY 再配置がこの目的を果たします。リンカは実行可能ファイル内の重複定義を R_C6000_COPY でマークします。ロード時に、動的ローダはライブラリ内の参照先シンボルを見つけ、その位置のデータを実行可能ファイル内の重複定義へコピーします。

このようにして、実行可能ファイルは動的リンクに対してペナルティを課されません。代わりに、そのペナルティはライブラリが負担します。ライブラリは、すべての extern 変数をインポート済みであると仮定しなければなりませんが、プリエンプションの仕組みにより、これはいずれにしても必要な処理になります。

15.10 プログラムのロード

Linux カーネルは、プログラムのロード処理を開始するにあたり、ロード セグメントと、その PT_INTERP ヘッダーで指定されたインタプリタ プログラムのロード セグメントの両方をメモリにコピーまたはマップします。その後カーネルはインタプリタ内のエントリ ポイントにジャンプし、インタプリタがロード処理を完了します。ELF 実行可能ファイルの場合、インタプリタは通常動的ローダ ld.so です。

インタプリタが最初に呼び出されるとき、インタプリタ自身の動的再配置を処理することによって自己ブートストラップする必要があります。その後、依存ライブラリをロードし、動的シンボル解決を行い、プログラム自身の動的再配置を処理する必要があります。

カーネルは、ロードマップと呼ばれる初期化済みデータ構造を介して、起動情報をインタプリタに伝えます。その宣言は次のとおりです:

プログラム ロード マップ データ構造

```
struct elf32_dsbt_loadmap
{
   /* Protocol version number, must be zero. */
   Elf32_Word version;
   /* Pointer to DSBT */
   unsigned   *dsbt_table;
   unsigned    dsbt_size;
   unsigned    dsbt_index;
   /* Number of segments */
   Elf32_Word  nsegs;
   /* The actual memory map. */
   struct elf32_dsbt_loadseg segs[nsegs];
};
struct elf32_dsbt_loadseg
{
   /* Core address to which the segment is mapped. */
   Elf32_Addr addr;
   /* Virtual address recorded in the program header. */
   Elf32_Addr p_vaddr;
   /* Size of this segment in memory. */
   Elf32_Word p_memsz;
};
```

カーネルは、4 つの引数をレジスタに、それ以外をスタックに置いて呼び出します。レジスタ引数は以下のとおりです:

B4	実行可能ファイルのロード マップのアドレス
A6	インタプリタのロード マップのアドレス
B6	インタプリタの動的セクションのアドレス
B14 (DP)	インタプリタの _ _c6xabi_DSBT_BASE

カーネルはプロセスにスタックを割り当て、SP を初期化します。スタックの初期内容は、プログラムのコマンドライン引数と環境変数を提供します。

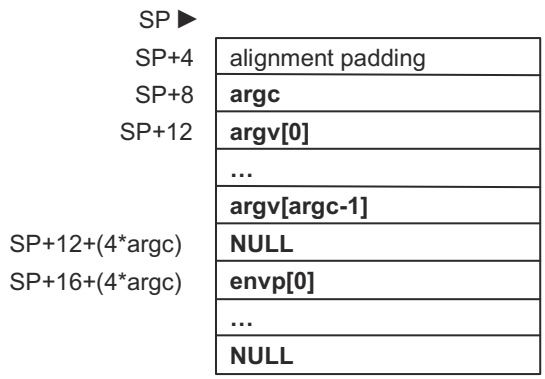


図 15-1. プログラム ロード マップ データ構造 のスタックの初期内容

その後カーネルは、_start というシンボルでラベル付けされたインタプリタのエントリ ポイントにジャンプします。

15.11 動的情報

動的セグメントには、プログラムのロードと動的リンクに関する情報が含まれています。これは System V ABI で指定されています。C6000 固有の動的タグの値と意味は「セクション 14.3.2」で指定されています。Linux モジュールには、グローバル シンボル マーカー タグ DT_C6000_GSYM_OFFSET と DT_C6000_GSTR_OFFSET が含まれていません。

Linux ABI では、すべての動的リンク メタデータがプログラムのロード イメージの一部です。つまり、PT_DYNAMIC セグメントと関連セクションは読み取り専用 PT_LOAD セグメントに含まれています。したがって、アドレス値 (d_ptr) がある動的タグは、ベアメタル ABI のようにファイル オフセットではなく仮想アドレスとして表現されます。

15.12 初期化関数と終了関数

System V ABI は、実行可能ファイルおよび共有オブジェクトに対して、グローバル オブジェクトのコンストラクタなどの関数を main を呼び出す前に呼び出すことができる初期化シーケンスを規定しています。同様に、main が戻った後に呼び出される関数を定義するための仕組みもあります。これらの仕組みは、DT_INIT* 動的タグおよび DT_FINI* 動的タグでマークされた関数ポインタのテーブルを使用します。

GC++ ABI のセクション 3.3.5 は、終了メカニズムを拡張して、使用しているプログラムが終了する前に共有オブジェクトがアンロードされたときに呼び出されるデストラクタを C++ プログラムが適切に登録できるようにしています。この仕組みでは、C++ コンパイラ サポート ライブラリ内の _ _cxa_atexit と呼ばれる API 関数を使用し、次のように呼び出されます。

```
__cxa_atexit(dtor, obj, &__dso_handle);
```

(ここで dtor はデストラクタ関数へのポインタ、obj はオブジェクトへのポインタです)

3 番目の引数である _ _dso_handle は、その共有オブジェクトを識別する一意のアドレスです。C6000 ABI は、その値をモジュールのニア DP セグメントのアドレスであると定義します。

もう 1 つの関数 _ _cxa_finalize は、共有オブジェクトがアンロードされるときに登録された関数を呼び出す機能を実装します。この関数は次のように呼び出されます。

```
__cxa_finalize(&__dso_handle);
```

リンカは、この呼び出しが最初の終了処理アクションとして行われるようにしなければならず、通常は DT_FINI* テーブルが使用されます。_ _cxa_finalize には引数がありますが、DT_FINI 関数は引数なしで呼ばれるため、リンカはこの呼び出しのために引数なしのラッパー関数を生成する必要があります。

この規則の要件をまとめると、静的リンカには以下の責務があります。

- ニア DP セグメントのアドレスを持つ非公開シンボル _ _dso_handle を生成すること。
- _ _cxa_finalize を前述のように呼び出す、引数なしのラッパー関数を生成すること。

- DT_FINI 動的タグまたは DT_FINI_ARRAY 動的タグでマークされた終了関数リストの最初の呼び出しとして、ラッパー関数を登録すること。

これらの要件は、_ _cxa_atexit を呼び出す任意の実行可能ファイルまたは共有オブジェクトを生成するときに適用されます。

15.13 Linux モデルの概要

表 15-1. Linux プログラム ファイル

特性	位置依存実行可能ファイル	位置独立実行可能ファイル	共有オブジェクト
ELF ファイル形式 (e_type)	ET_EXEC		ET_DYN
ELF 識別 (e_ident)	ELFOSABI_C6X_LINUX		
読み取り専用セグメント	複数可	1	
DP 相対データ セグメント	1		
その他の読み取り / 書き込みセグメント	絶対	GOT のみ	
コードのアドレッシング	PC 相対または絶対	PC 相対または GOT	
独自の非公開データのアドレッシング	DP 相対または絶対	DP 相対	
インポートされた STV_DEFAULT データのアドレッシング	ファー (DP 相対、絶対、GOT)	DP 相対または GOT	GOT
DSBT モデル	必須		
読み取り専用セグメントへのロード時再配置が必要	あり	なし	
外部シンボルのデフォルトの可視性	STV_DEFAULT		
ロード セグメントにはメタデータ (PT_INTERP、PT_PHDR、PT_DYNAMIC) が含まれる	あり		

Chapter 16

シンボルのバージョン管理

シンボルのバージョン管理は、共有ライブラリ内で複数のバージョンのシンボルをサポートし、動的にリンクされたコンポーネント間の互換性を確保するメカニズムを提供します。C6000 の実装は GNU ツールチェーンで使用されているものに基づいており、Sun Microsystems から採用されています。GNU におけるシンボルのバージョン管理のサポートに関する参考文書は、http://people.redhat.com/drepper/symbol-versioning の Ulrich Drepper による論文です。当社が知る限り、C6000 固有の追加や変更はありません。このドキュメントの説明は、参照目的でこのメカニズムをまとめたものです。

シンボルのバージョン管理を使用する実行可能ファイルは、ELF ヘッダ内の EI_OSABI フィールドを適切なオペレーティング システム固有の値に設定する必要があります。

16.1 ELF シンボル バージョニングの概要

GNU シンボル バージョニングを使用すると、ユーザーは DSO からエクスポートされたシンボルに対してバージョン名を指定できます。これにより、1 つの DSO 内に同じシンボル定義の複数のバージョンを持つことができます。そのうち 1 つのみがデフォルトとして指定されます。このシンボル定義にリンクされると、シンボル参照のバインドには、常にデフォルト バージョンが使用されます。

たとえば、あるライブラリ実装者が codec_1_0.dso 内に API 関数 api_do_encode を定義すると仮定します。当初は、1 つのバージョンしか存在しません。仮にこれを VER1 としますアプリケーションがこの DSO にリンクすると、api_do_encode へのすべての参照は api_do_encode の VER1 によって解決されます。その後、実装者は api_do_encode の新しいバージョンを追加して API を拡張します。この新バージョンは古いものと互換性はありませんが、従来の API を利用する既存アプリケーションも引き続きサポートする必要があります。そこで実装者は、新しい codec_2_0.dso を作成し、元の VER1 api_do_encode と同じシンボルの新しい VER2 の定義を同時に含めます。以降、VER2 がデフォルトのバージョンとして指定されます。新しいアプリケーションが codec_2_0.dso とリンクすると、api_do_encode への参照は VER2 api_do_encode によって解決されます。元の VER1 api_do_encode は、codec_1_0.dso に基づいてビルドされた古いアプリケーションからの参照を解決するために依然として利用可能です。

シンボル バージョンを指定する仕組みの詳細については、Drepper の論文を参照してください。

GNU のシンボル バージョニングに関する情報は、次の 3 つの ELF セクションに記録されます。

- **バージョン定義セクション**

  このセクションでは、この実行可能ファイルからエクスポートされるシンボルに関連付けられたバージョン名を定義します。また、ファイルのバージョンもここで定義されます。

  このセクションは、動的セクション内の DT_VERDEF タグ エントリを通じて位置を参照できます。DT_VERDEFNUM タグには、このセクションに含まれるバージョン定義の数が格納されます。バージョン定義セクションのセクション タイプは SHT_TI_verdef です。このセクション タイプの値 0x6FFFFFFD は SHT_GNU_verdef と同じものになります。この仕様では、このセクション名として .gnu.version_d を推奨しますが、このセクションの識別にはセクション タイプのみを使用し、名前は使用しないでください。
- **要求されるバージョン セクション**

  このセクションには、この実行可能ファイル内の未定義シンボル参照によって要求されるバージョンが記録されます。各エントリは DSO を指定し、そこから必要とされるバージョンのリストを参照します。動的リンカが実行可能ファイルをロードする際には、必要なすべての DSO を探して読み込みます。DSO を公開する前には、実行可能ファイルが要求するバージョンが、対象の DSO のバージョン定義で満たされているかを動的リンカが確認します。この要求されるバージョンに関する情報は、静的リンカが DSO 内の定義に参照をバインドする際に記録されます。
- **バージョン セクション**

  このセクションは、動的シンボル エントリにバージョン番号を追加して、動的シンボル テーブルを拡張します。セクション内のエントリ数は、動的シンボル テーブルのエントリ数と同じです。シンボル ID は、このバージョン番号テーブルを参照するインデックスとして使用されます。シンボルが未定義の場合、そのバージョン番号は要求されるセクションのエントリに対応します。シンボルが定義済みの場合、そのバージョン番号はバージョン定義セクションのエントリに対応します。ビット 15 がクリアされていると、バージョン定義は *デフォルト* とみなされます。

ELF には、ELF 実行可能ファイル内でこれらのシンボル バージョン セクションを特定・識別する仕組みがあります。これらのセクションは動的セクションのタグによって位置付けられ、特定のセクション タイプによって識別されます。

たとえば、バージョン定義セクションは動的タグ DT_VERDEF によって指定されます。DT_VERDEFNUM タグには、そのバージョン定義セクションに含まれるバージョン定義の数が記録されます。このセクションのタイプは SHT_GNU_verdef (0x6FFFFFFD) でなければなりません。このセクションの名前は形式上 .gnu.version_d ですが、実装では名前ではなくセクション タイプを基準とすることを推奨します。

16.2 バージョン セクションの識別

表 16-1 には、シンボル バージョニングに関連する 3 種類の ELF セクションについて、それぞれのタグ、セクション タイプ、セクション名が示されています。

表 16-1. バージョン セクションの識別

ELF セクション	動的タグ	セクション タイプ	セクション名
バージョン定義	DT_VERDEF (0x6FFFFFFC) DT_VERDEFNUM (0x6FFFFFFD)	SHT_GNU_verdef (0x6FFFFFFD)	.gnu.version_d
必要なバージョン	DT_VERNEED (0x6FFFFFFE) DT_VERNEEDNUM (0x6FFFFFFF)	SHT_GNU_verneed (0x6FFFFFFE)	.gnu.version_r
バージョン	DT_VERSYM (0x6FFFFFF0)	SHT_GNU_versym (0x6FFFFFFF)	.gnu.versym

Chapter 17
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。C6x 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。

17.1 C6000 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 C6000 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。

C6000 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。

- 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。

17.2 ビルド属性タグ

Tag_ISA (=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 C6000 ISA を指定します。次の値が定義されています。

0	ISA の指定なし
1	C62x
2	予約済み
3	C67x
4	C67x+
5	予約済み
6	C64x
7	C64x+
8	C6740
9	Tesla
10	C6600

このタグはオブジェクトの互換性を次のように決定します。ここで、推移的関係 A < B は、B が A と互換性を持つことを意味します。すなわち、B は A または B のいずれかに生成されたコードを実行できます。属性を組み合わせる際には、両方を実行できる *最大の* ISA (この場合は B) を使用する必要があります。

- Tesla は他の ISA リビジョンとの互換性を持ちません。
- C62x < Tesla を除くすべての ISA
- C67x < C67x+
- C67x+ < C6740
- C64x < C64x+
- C64x+ < C6740
- C6740 < C6600

C6000 ISA の互換性は、次の有向グラフによって示されており、その中でエッジ A→B は互換関係 A < B を表します。

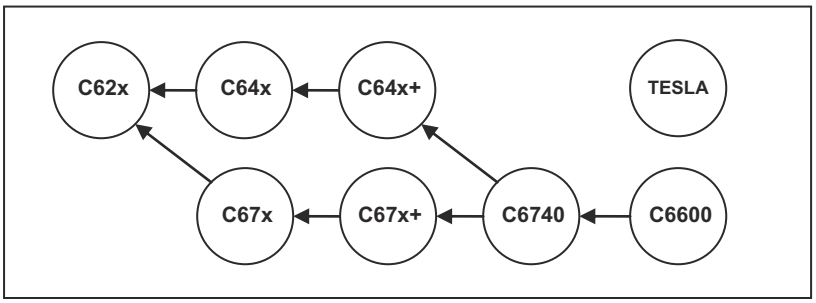


図 17-1. C6000 ISA の互換性グラフ

Tag_ABI_wchar_t, (=6), ULEB128

0	wchar_t は使用しない。
1	wchar_t のサイズは 2 バイト。
2	wchar_t のサイズは 4 バイト。

セクション 2.1 は wchar_t を符号なし整数として指定します。ただし、状況によっては、TI ツールチェーンは wchar_t を符号なし短整数と定義します。このタグを使用すると、この違反に起因する非互換性を検出できます。

Tag_ABI_stack_align_needed, (=8), ULEB128

0	コードには、機能境界で 8 バイトのスタック整列が必要です。
1	コードには、機能境界で 16 バイトのスタック整列が必要です。

Tag_ABI_stack_align_preserved, (=10), ULEB128

0	コードには、機能境界で 8 バイトのスタック整列が必要です。
1	コードには、機能境界で 16 バイトのスタック整列が必要です。

現在サポートされているすべての ISA は、8 バイトのスタック整列を使用します。将来的な ISA では、16 バイトの整列が予定されています。

16 バイトのスタック整列を必要とするコードは、8 バイトの整列のみを保持するコードとは互換性がありません。タグをマージする場合、結果は TAG_ABI_stack_align_preserved で指定される最小の整列と、TAG_ABI_stack_align_needed で指定される最大の整列を反映したものになる必要があります。

Tag_ABI_DSBT, (=12), ULEB128

0	DSBT アドレッシングは使用しない。
1	DSBT アドレッシングを使用。

Tag_ABI_PID, (=14), ULEB128

0	データ アドレッシングは位置に依存。
1	データ アドレッシングは位置に依存せず、GOT はニア DP アドレッシングを使用してアクセス。
2	データ アドレッシングは位置に依存せず、GOT はファー DP アドレッシングを使用してアクセス。

ゼロ以外の Tag_ABI_PID を持つオブジェクト ファイルは、データに絶対アドレッシングを使用しません。すべてのデータは、DP 相対、GOT、または読み取り専用定数の場合は PC 相対アドレッシングを使用してアドレス指定されます。このようなオブジェクトには、共有オブジェクトが必要とする再配置を必要とせずに、DP 相対データ セグメントの位置を動的に割り当てることができます。

値が 2 の場合、そのオブジェクトがファー GOT ベースのアドレッシングに依存していることを示す (「セクション 6.6」を参照)。つまり、GOT 自体がファーになります。

Tag_ABI_PIC, (=16), ULEB128

0	アドレッシング規則が、共有オブジェクトには適していません。
1	アドレッシング規則が、共有オブジェクトに適しています。

Tag_ABI_PIC は、そのオブジェクトが共有オブジェクトに必要とされるアドレッシング規則に従っていることを示します。特に、インポートされた変数へのすべての参照が GOT を介してアドレッシングされることを示します。

共有ライブラリをリンクするとき、リンカは、そのライブラリを構成するすべてのオブジェクトにこのタグの存在を強制する必要があります。

Tag_ABI_PIC という名前は、誤解を招く可能性があります。「位置の独立性」という用語は、いくつかの関連する特性を指す場合がありますが、それが共有オブジェクトに必要とされる特性と一致するとは限りません。したがって、この属性は後者の集合に基づいて定義されます。

Tag_ABI_array_object_alignment, (=18), ULEB128

0	配列変数は 8 バイト境界に整列されます。
1	配列変数は 4 バイト境界に整列されます。
2	配列変数は 16 バイト境界に整列されます。

Tag_ABI_array_object_align_expected,(=20), ULEB128

0	コードは配列変数に 8 バイトの整列を想定します。
1	コードは配列変数に 4 バイトの整列を想定します。
2	コードは配列変数に 16 バイトの整列を想定します。

前述の 2 つのタグは、外部から参照可能な配列変数に適用されます。これは、セクション 2.6 で説明したとおりです。互換性のために、TAB_ABI_array_align_expected タグで示される整列値は、TAG_ABI_array_object_alignment タグで示される整列値以下である必要があります。タグをマージする場合、結果は TAG_ABI_array_object_alignment で指定される最小の整列と、TAG_ABI_array_object_align_expected で指定される最大の整列を反映したものになる必要があります。

Tag_ABI_compatibility, (=32), ULEB128, char[]

ベンダは ABI の範囲を超えた独自の互換性規則を定めることができます。ULEB128 フラグと NULL で終了する文字列の 2 つのオペランドがあります。文字列は、取り決めたベンダによって定義された、ABI 外規則の名前を指定します。フラグは、その規則に関してオブジェクトを特徴付けます。以下の説明における *ABI 互換* という用語は、この ABI に準拠していること、さらには本書で定められた条件 (たとえば他のビルド属性タグ) に基づいて互換性があることを意味します。フラグ値は以下のとおりです。

0	このオブジェクトにはツールチェーン固有の互換性要件がなく、他のすべての ABI 互換オブジェクトと互換性があります。
1	このオブジェクトは、指定された規則に準拠したツールチェーンで処理される場合に、他の ABI 互換オブジェクトと互換性を持ちます。たとえば、その規則がベンダを指定している場合は、そのベンダのツールチェーンが該当します。
N が 1 より大きい	このオブジェクトは ABI とは互換性がありませんが、指定された規則の下では他のオブジェクトと互換性を持つ場合があります。この場合、フラグの解釈はその規則によって定義されます。

この文字列が識別するのは ABI 外規則であり、必ずしもファイルを生成したツールチェーンとはなるわけではありません。

ABI 互換性タグが省略された場合は、フラグ値 0、すなわち追加の互換性要件がないタグと同じ意味になります。

Tag_ABI_conformance, (=67), char[]

このタグはオブジェクトが準拠する ABI のバージョンを指定します。タグ値は ABI のバージョンを含む NULL で終了する文字列になります。この規格で指定されているバージョンは「1.0」です。小数点の後に続く数字は情報提供のみを目的とし、互換性チェックには影響しません。

利用者が認識しやすいように、ファイル全体が ABI に準拠する一般的なケースでは、このタグを C6000 サブセクション内の最初の属性ベクトルの先頭に配置する必要があります。

表 17-1 は、ABI によって定義されるビルド属性タグの概要です。

表 17-1. C6000 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	
Tag_ISA	4	ULEB128	前述の説明を参照してください
Tag_ABI_wchar_t	6	ULEB128	0 以外の場合は、正確に一致する必要があります
Tag_ABI_stack_align_needed	8	ULEB128	Tag_ABI_stack_align_preserved と互換性を持つ必要があります。 最大値を使用して結合します。
Tag_ABI_stack_align_preserved	10	ULEB128	Tag_ABI_stack_align_needed と互換性を持つ必要があります。 最小値を使用して結合します。
Tag_ABI_DSBT	12	ULEB128	完全一致
Tag_ABI_PID	14	ULEB128	異なる場合は警告されます。最小値を使用して結合します。
Tag_ABI_PIC	16	ULEB128	共有ライブラリを構築するときに存在しない場合は警告されます。最小値を使用して結合します。
TAG_ABI_array_object_alignment	18	ULEB128	少なくとも、TAG_ABI_array_object_align_expected の整列である必要があります。 最大の整列を使用して結合します。
TAG_ABI_array_object_align_expected	20	ULEB128	TAG_ABI_array_object_alignment の整列以下である必要があります。 最小の整列を使用して結合します。
Tag_ABI_compatibility	32	ULEB128 char[]	本文の説明を参照してください。
Tag_ABI_conformance	67	char[]	未指定

Chapter 18

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。

18.1 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バイトで指定する 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;
   uint32  run_addr;
   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 データの最初のバイ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* バイトをロード 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 データの最初のバイ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。

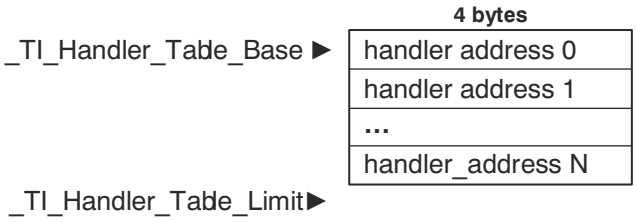


図 18-1. ハンドラ テーブルのフォーマット

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 8 ビット インデックスの後のバイ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。

「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);
      }
   }
}
```

18.2 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。

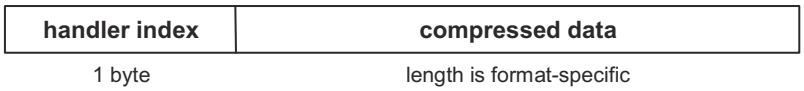


図 18-2. 圧縮ソース データ フォーマット

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。

18.2.1 RLE

8 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。C6000 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. 最初のバイトを読み取り、それを区切り文字 (D) として割り当てます。
2. 次のバイト (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. 次のバイト (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、次のバイト (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. 次の 16 ビット (LL) を読み取ります。
8. 次のバイト (C) を読み取ります。
9. C != 0 の場合、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。

18.2.2 LZSS フォーマット

8 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。

18.3 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 18 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 18-3 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。

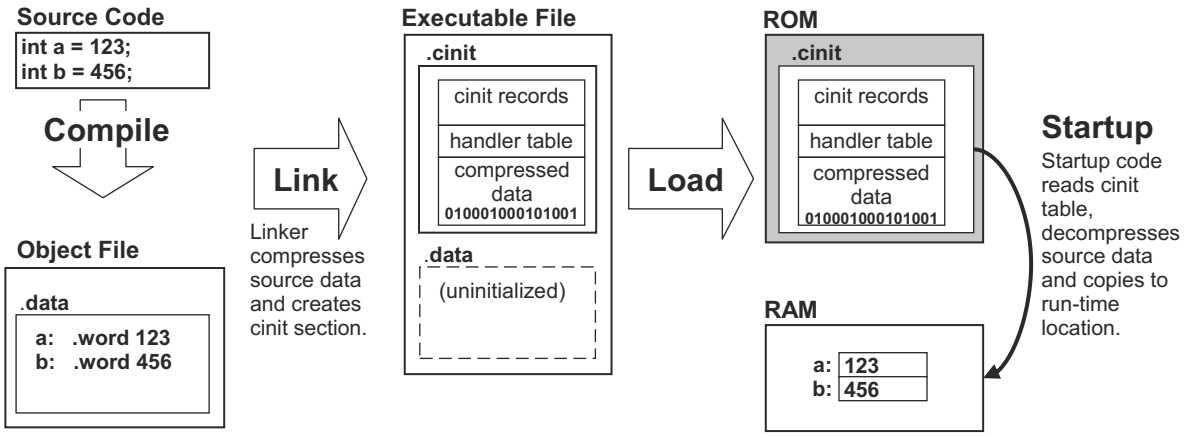


図 18-3. cinit を介した ROM ベースの変数の初期化

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 18.1 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 18-4 は、.cinit セクションの概略図です。

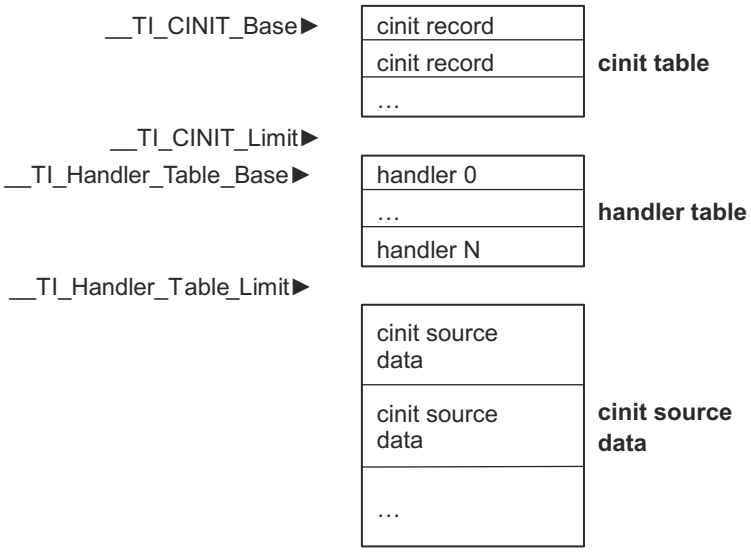


図 18-4. .cinit セクション

.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。

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 1 バイト先を指します。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。

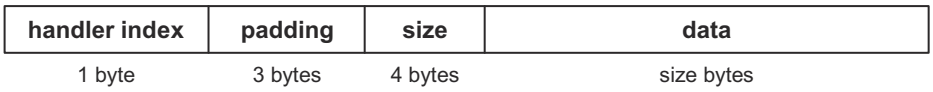
cinit テーブル内のレコードは次の形式を取ります。

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

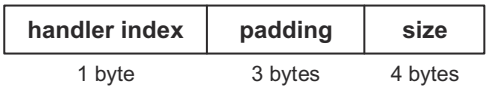
- **source_data** フィールドは .cinit セクション内のソース データを指します。
- **dest** フィールドは宛先アドレスを指します。コピー テーブルのレコードとは異なり、cinit レコードにはサイズ フィールドがありません。サイズは常にソース データ内にエンコードされます。

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 18.1 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。



  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 4 バイ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バイト数を指定します。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* のバイト数が、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。



  サイズ フィールドは、ハンドラ インデックスの次に来る 4 バイ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* 分のバイトをゼロで埋めます。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。

Chapter 19

拡張プログラム ヘッダー属性

ELF 実行可能ファイルおよび共有ライブラリは、プログラム ヘッダー テーブルを含んでいます。プログラム テーブルの各エントリは、1 つのセグメントを表します。他のメタデータとあわせて、プログラム テーブルはセグメント属性のプロセッサ固有拡張を限定的に可能にします。最大で、OS 固有のフラグが 8 個、プロセッサ固有のフラグが 4 個まで許可されます。

これらのフラグは、プロセッサ固有 ABI によって追加のセグメント プロパティを表現するために使用できます。

ただし、使用可能なフラグの数は非常に少なく、パラメータを持つ属性を表現するためには使用できません。TI では、ELF のプログラム ヘッダー テーブルにおいて、システム / デバイス / アプリケーション固有のセグメント属性の追加指定が必要になると想定しています。セグメント フラグでは TI が必要とするすべての属性を表現するには不十分であるため、ELF フォーマットを拡張し、*拡張プログラム ヘッダー属性* を導入しました。C6000 EABI に準拠したツールは、実装品質の観点から、拡張プログラム ヘッダー属性のサポートを選択して実装できます。拡張プログラム ヘッダー属性への対応は、C6000 EABI 準拠の要件ではありません。

拡張プログラム ヘッダー属性は、SHT_TI_PHATTRS (0x7F000004) 型で名前が .TI.phattrs のプロセッサ固有セクションにエンコードされます。このセクションは、PT_TI_PHATTRS (0x70000000) 型のセグメントによって指定されたセグメントに含まれます。

19.1 エンコード

プログラム ヘッダ属性は <セグメント ID, タグ, 値> トリプレットでエンコードされます。これは次のように表現できます。

```
typedef struct
{
   Elf32_Half pha_seg_id;       /* Segment id */
   Elf32_Half pha_tag_id;       /* Attribute kind id */
   Union
   {
      Elf32_Off pha_offset;     /* byte offset within the .TI.phattrs section */
      Elf32_Word pha_value;     /* Constant tag value */
   }  pha_un;
}  Elf32_TI_PHAttrs;
```

セグメント ID とタグ ID はどちらも、ELF ファイルのバイト順序で 2 バイト符号なし整数としてエンコードされます。共用体 pha_un のフィールドは、ELF ファイルのバイト順序で 4 バイト符号なし整数としてエンコードされます。この表現は、動的タグの <タグ, 値> 表現に基づいてモデル化されています。

タグの値は、インライン 32 ビット定数、または固定長バイナリ データ (FLBD) かヌル終端バイト文字列 (NTBS) を指す .TI.phattrs セクションへのオフセットにすることができます。固定長バイナリ データのサイズは 32 ビット アラインにする必要があります。

拡張プログラム ヘッダ属性セグメントが存在する場合は、PHA_NULL タグによって終了します。

属性タグの値とプロパティは TI によって割り当ておよび管理されており、プロセッサ固有です。すべての未定義値は将来の使用のために予約されています。

属性タグでは、pha_un の値がどのように解釈されるかを決定します。各属性には事前定義された動作があります。pha_un フィールドは pha_value または pha_offset として解釈できます。また、使用しない場合もあります。pha_offset を使用する場合、値は NTBS または FLBD を指します。pha_offset が FLBD として解釈される場合、フィールドの長さは事前に定義されている必要があります。

19.2 属性タグの定義

テキサス インスツルメンツは、ネイティブの ROMing サポートに対応するために、2 つの属性を導入しました。

表 19-1. ROMing サポート属性

名称	タグ ID	d_un	長さ
PHA_NULL	0x0	無視	なし
PHA_BOUND	0x1	無視	なし
PHA_READONLY	0x2	無視	なし

- 属性 **PHA_BOUND** は、そのセグメントのアドレスが最終アドレスにバインドされ、ダウンストリームでの再リンク、動的リンク、動的ロードの段階において変更できないことを示します。この特性は、セグメント自身が ROM に配置されている場合、または ROM 内のコードから絶対アドレスによって参照される場合に適用されます。

  PHA_BOUND はまた、静的リンカまたは動的リンカに対して、このアドレスが割り当て済みであり、さらなる割り当てには利用できないことを示します。
- **PHA_READONLY** は、そのセクションが *真の* 定数データを含むこと、すなわち静的リンカおよび動的リンカがその内容に対して再配置を行ったり、いかなる変更も加えてはならないことを示します。PHA_READONLY セグメントは、再配置エントリを一切持ってはなりません。動的ローダはこれをヒントとして、そのようなセグメントに対する再配置処理を回避できます。

19.3 拡張プログラム ヘッダ属性セクションのフォーマット

拡張プログラム ヘッダ属性セクションは次の 3 つのパートに分かれています。

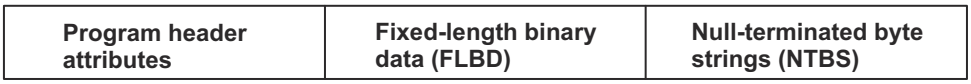


図 19-1. 拡張プログラム ヘッダ属性セクションのフォーマット

最初のパートは、PHA_NULL で終端された Elf32_TI_PHAttrs のベクトルです。次に、FLBD パートと NTBS パートが続きます。使用する場合、pha_un.pha_offset は、セクションの先頭を基準にしたバイト オフセットを使用して FLBD パーツまたは NTBS パーツを指します。pha_offset フィールドにアクセスするタグがない場合は、FLBD と NTBS を空にしても構いません。

次の表には、2011 年 9 月から 2014 年 3 月までの間に行われた変更が記載されています。左側の列には、その特定の変更が反映されたこのドキュメントのバージョンが示されています。

表 20-1. 改訂履歴

	ロケーション	追加 / 変更 / 削除
SPRAB89A	Chapter 7	スレッドローカル ストレージ (TLS) について説明する新しいセクション。
SPRAB89A	セクション 13.5.1	スレッドローカル ストレージ再配置タイプが追加。
SPRAB89A	セクション 13.5.2	スレッドローカル ストレージ再配置操作が追加。
SPRAB89A	セクション 14.1.4	スレッドローカル ストレージに関する新しいセクションを参照。
SPRAB89A	表 8-8	表の後の解説が、スレッドローカル ストレージに関する新しいセクションを参照。
SPRAB89A	セクション 13.3.5	スレッドローカル ストレージのセクションに関する情報。

Changes from MARCH 31, 2014 to AUGUST 6, 2025 (from Revision A (March 2014) to Revision B (August 2025)) **Page**

- OpenMP は C6000 ターゲットではサポートされなくなりました.....59
- OpenMP は C6000 ターゲットではサポートされなくなりました.....60
- R_C6000_JUMP_SLOT 再配置操作の綴りを修正しました。..... 129

重要なお知らせと免責事項

テキサス・インスツルメンツは、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、テキサス・インスツルメンツ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 テキサス・インスツルメンツ製品の選定、(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 テキサス・インスツルメンツ製品を使用するアプリケーションの開発の目的でのみ、テキサス・インスツルメンツ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。テキサス・インスツルメンツ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、テキサス・インスツルメンツおよびその代理人を完全に補償するものとし、テキサス・インスツルメンツは一切の責任を拒否します。

テキサス・インスツルメンツの製品は、テキサス・インスツルメンツの販売条件、または ti.com やかかる テキサス・インスツルメンツ製品の関連資料などのいずれかを通じて提供する適用可能な条項の下で提供されています。テキサス・インスツルメンツがこれらのリソースを提供することは、適用されるテキサス・インスツルメンツ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。

お客様がいかなる追加条項または代替条項を提案した場合でも、テキサス・インスツルメンツはそれらに異議を唱え、拒否します。

郵送先住所:Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

重要なお知らせと免責事項

テキサス・インスツルメンツは、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、 テキサス・インスツルメンツ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 テキサス・インスツルメンツ製品の選定、(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 テキサス・インスツルメンツ製品を使用するアプリケーションの開発の目的でのみ、 テキサス・インスツルメンツ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。 テキサス・インスツルメンツ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、 テキサス・インスツルメンツおよびその代理人を完全に補償するものとし、 テキサス・インスツルメンツは一切の責任を拒否します。

テキサス・インスツルメンツの製品は、 テキサス・インスツルメンツの販売条件、または ti.com やかかる テキサス・インスツルメンツ製品の関連資料などのいずれかを通じて提供する適用可能な条項の下で提供されています。 テキサス・インスツルメンツがこれらのリソースを提供することは、適用される テキサス・インスツルメンツ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。

お客様がいかなる追加条項または代替条項を提案した場合でも、 テキサス・インスツルメンツはそれらに異議を唱え、拒否します。

郵送先住所：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265