ARM® コンパイラ

バージョン 6.6

『ソフトウェア開発ガイド』

ARM

ARM DUI0773GJ

ARM® コンパイラ

『ソフトウェア開発ガイド』

リリース情報

ドキュメント履歴

発行	日付	機密保持ステータス	変更点
A	14 3 月 2014	非機密扱い	ARM コンパイラ v6.00 リリース
B	15 12 月 2014	非機密扱い	ARM コンパイラ v6.01 リリース
C	30 6 月 2015	非機密扱い	ARM コンパイラ v6.02 リリース
D	18 11 月 2015	非機密扱い	ARM コンパイラ v6.3 リリース
E	24 2 月 2016	非機密扱い	ARM コンパイラ v6.4 リリース
F	29 6 月 2016	非機密扱い	ARM コンパイラ v6.5 リリース
G	04 11 月 2016	非機密扱い	ARM コンパイラ v6.6 リリース

非機密著作権情報

機密保持ステータス

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

無制限アクセスは、ARM 社内による分類です。

製品ステータス

本書の情報は最終版であり、開発済み製品に対応しています。

Web アドレス

http://www.jp.arm.com

目次
ARM® コンパイラ『ソフトウェア開発ガイド』

図の一覧
ARM® コンパイラ『ソフトウェア開発ガイド』

表の一覧
ARM® コンパイラ『ソフトウェア開発ガイド』

序章

この前書きでは、次について紹介します。*ARM® コンパイラ『ソフトウェア開発ガイド』*.

このドキュメントは、次で構成されています。

- *本書について*(9 ページ).

本書について

『ARM® コンパイラソフトウェア開発ガイド』には、さまざまな ARM アーキテクチャベースプロセッサのコードを開発するためのチュートリアルと例が記載されています。

本書の構成

本書は以下の章から構成されています。

第 章 1 ツールチェインの概要
ARM コンパイルツールについての概要、および簡単なコード例をコンパイルする方法が収録されています。

第 章 2 診断
コンパイラツールチェイン診断メッセージの形式と診断出力の制御方法について説明します。

第 章 3 C および C++ コードのコンパイル
`armclang` を使用して C および C++ コードをコンパイルする方法について説明します。

第 章 4 アセンブリコードのアセンブル
`armclang` および `armasm` を使用してアセンブリソースコードをアセンブルする方法について説明します。

第 章 5 オブジェクトファイルをリンクして実行可能ファイルを生成する
`armlink` を使用してオブジェクトファイルをリンクし、実行可能イメージを生成する方法について説明します。

第 章 6 最適化
`armclang` を使用してコードサイズとパフォーマンスのいずれかに最適化する方法と、デバッグ機能に対する最適化レベルの影響について説明します。

第 章 7 コード作成時の注意事項
プログラミング方法やテクニックを用いて、C および C++ ソースコードの移植性、効率性、および堅牢性を高める方法について説明します。

第 章 8 コードとデータのターゲット メモリへのマッピング
ターゲット ハードウェアの正しいメモリ領域にコードおよびデータを配置する方法を説明します。

第 章 9 ARMv8-M セキュリティ拡張機能を使用したセキュア イメージおよび非セキュア イメージのビルド
ARMv8-M セキュリティ拡張機能を使用してセキュア イメージをビルドする方法と、非セキュア イメージによってセキュア イメージを呼び出すことができるようにする方法について説明します。

用語集

「ARM 用語集」は、ARM マニュアルで使用されている用語とその定義のリストです。一般に認められている意味と ARM での意味が異なる場合を除いて、「ARM 用語集」に業界標準の用語は含まれていません。

詳細については、「*ARM 用語集*」を参照して下さい。

表記規則

italic
重要用語、相互参照、引用箇所を示します。

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

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

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

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

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

`<and>`
: コードまたはコードの一部のアセンブラ構文で置換可能な項が使用されている場合に、その項を囲みます。以下はその一例です。

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

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

ご意見、ご感想

本製品に関するフィードバック

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

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

内容に関するフィードバック

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

- タイトル *ARM コンパイラ『ソフトウェア開発ガイド』*。
- 文書番号（ARM DUI0773GJ）。
- 該当する場合は、問題のあるページ番号。
- 問題点の簡潔な説明

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

—— **注** ——

ARM では、この PDF を Adobe Acrobat および Acrobat Reader でのみテストしており、その他の PDF リーダーを使用した場合の表示品質については、保証いたしかねます。

その他の情報

- *ARM Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *ARM Glossary*.

第 1 章
ツールチェインの概要

ARM コンパイルツールについての概要、および簡単なコード例をコンパイルする方法が収録されています。

以下のセクションから構成されています。

1.1 ツールチェインの概要

ARM® コンパイラ 6 コンパイルツールは、実行可能イメージ、部分的にリンクされたオブジェクトファイル、そして共有オブジェクトファイルの構築、そしてイメージの異なる形式への変換を可能にします。

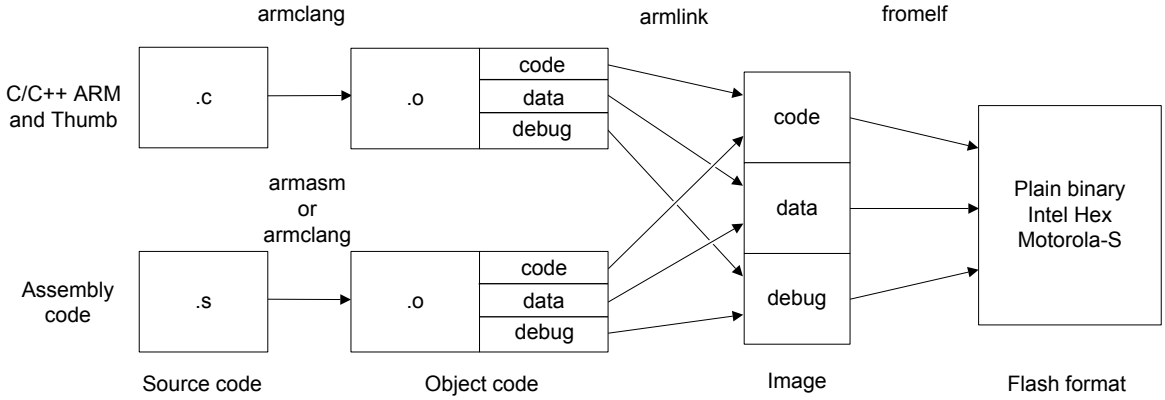


図 1-1 コンパイラツールチェーン

ARM コンパイラツールチェーンは以下のツールで構成されています。

armclang

`armclang` コンパイラおよびアセンブラ。C および C++ コードをコンパイルし、A64、A32、および T32 GNU 構文のアセンブリ コードをアセンブルします。

armasm

レガシーアセンブラ。これは、ARM 構文を使用して、A32、A64、および T32 アセンブリコードをアセンブルします。

レガシー ARM 構文アセンブリコードには `armasm` のみを使用します。すべての新しいアセンブリファイルには `armclang` アセンブラと GNU 構文を使用します。

armlink

リンカ。1 つまたは複数のオブジェクトファイルの内容と、1 つまたは複数のオブジェクトライブラリから選択された部分を結合し、実行可能プログラムを生成します。

armar

ライブラリアン。ELF オブジェクト ファイルをアーカイブやライブラリにまとめて保存するために使用します。これにより、複数の ELF ファイルの代わりにライブラリやアーカイブをリンカに渡すことができます。アーカイブをサードパーティに配布して、さらに高度なアプリケーションの開発に使用してもらうこともできます。

fromelf

イメージ変換ユーティリティ。逆アセンブリおよびコードサイズやデータサイズなど、入力イメージに関するテキスト情報を生成することもできます。

——— **注** ———

逆アセンブリは、ARM アセンブラ構文で生成され、GNU アセンブラ構文では生成されません。

———————————

関連タスク

1.5 "Hello world" の例(1-21 ページ).

関連参照

1.4 共通の ARM コンパイラツールチェーンのオプション(1-18 ページ).

1.2 サポートレベルの定義

ここでは、さまざまな ARM コンパイラ 6 の機能のサポート レベルについて説明します。

ARM コンパイラ 6 は Clang および LLVM テクノロジ上に構築されており、本書に記載されている製品機能セットよりも多くの機能が備えられています。それらの機能で見込まれている、機能のサポート レベルと保証内容を以下の定義で明らかにしていきます。

ARM では、ARM コンパイラ 6 のあらゆる機能の使用に関するフィードバックをお待ちしております。また、ユーザがその機能に適したレベルに到達できるようサポートいたします。サポートについては、*http://www.arm.com/support* でお問い合わせいただけます。

ドキュメント内での識別

明記されている場合を除き、ARM コンパイラ 6 のマニュアルに記載されているすべての機能は製品機能です。製品以外の機能の制限については明記されています。

製品機能
製品機能は運用環境での使用に適しています。機能性は十分にテストされ、機能や更新リリースにおける安定性が見込まれています。

- ARM では、製品機能に対する有意な機能変更について前もってお知らせするよう努めています。
- サポートおよびメンテナンス契約を結んでいらっしゃるお客様には、すべての製品機能のご利用に対して徹底したサポートを提供いたします。
- ARM では製品機能に関するフィードバックをお待ちしております。
- ARM が発見または把握している製品機能の問題については、ARM コンパイラの今後のバージョンでの修正が検討されます。

完全サポート対象の製品機能に加えて、アルファ品質またはベータ品質の製品機能も一部あります。

ベータ製品機能
ベータ製品機能は完全に実装されていますが、運用環境での使用に適していると見なされるまでのテストが十分に行われていません。
ベータ製品機能には [BETA] と記されています。

- ARM では、ベータ製品機能の既知の制限を文書化するよう努めています。
- ベータ製品機能は、最終的には ARM コンパイラ 6 の今後のリリースで製品機能となることが見込まれています。
- ARM ではベータ製品機能のご利用を推奨しており、また、これらの機能に関するフィードバックをお待ちしております。
- ARM が発見または把握しているベータ製品機能の問題については、ARM コンパイラ の今後のバージョンでの修正が検討されます。

アルファ製品機能
アルファ製品機能は完全に実装されておらず、今後のリリースで変更される場合があるため、安定性レベルはベータ製品機能よりも低くなります。
アルファ製品機能には [ALPHA] と記されています。

- ARM では、アルファ製品機能の既知の制限を文書化するよう努めています。
- ARM ではアルファ製品機能のご利用を推奨しており、また、これらの機能に関するフィードバックをお待ちしております。
- ARM が発見または把握しているアルファ製品機能の問題については、ARM コンパイラの今後のバージョンでの修正が検討されます。

コミュニティ機能

ARM コンパイラ 6 は LLVM テクノロジ上に構築され、可能な場合はそのテクノロジの機能性が保持されています。つまり、本書に記載はされていなくても ARM コンパイラで利用できる機能が他にもあるということです。このような追加機能は、コミュニティ機能と呼ばれています。コミュニティ機能の詳細については、*Clang/LLVM プロジェクトのマニュアル*を参照して下さい。

本マニュアル内でコミュニティ機能を参照できる箇所には [COMMUNITY] と記されています。

- 本書で明記している場合を除き、ARM ではこれらの機能の品質レベルまたは機能水準について一切言及いたしません。
- 機能は、機能リリースによって大幅に変更される場合があります。
- コミュニティ機能の変更は見込まれておりませんが、ARM では更新リリースでコミュニティ機能が引き続き機能するかどうかについては保証いたしません。

今後、一部のコミュニティ機能がコミュニティ機能になるかもしれませんが、この件に関して ARM が指針を提供することはありません。ARM ではこのような機能の利用状況を把握したいと考えています。機能に関するフィードバックをお待ちしております。サポート対象の機能については、ARM は、お客様が機能をご利用するにあたって最善のサポートを提供できるように努めます。これらの機能に関する欠陥レポートの受け付けは行いますが、その問題を今後のリリースで修正することについては保証いたしません。

コミュニティ機能の使用に関するガイダンス

動作しているコミュニティ機能の可能性を評価する場合は、いくつかの要素が検討されます。

- 以下の図は、ARM コンパイラ 6 のツールチェーンの構造を示しています。

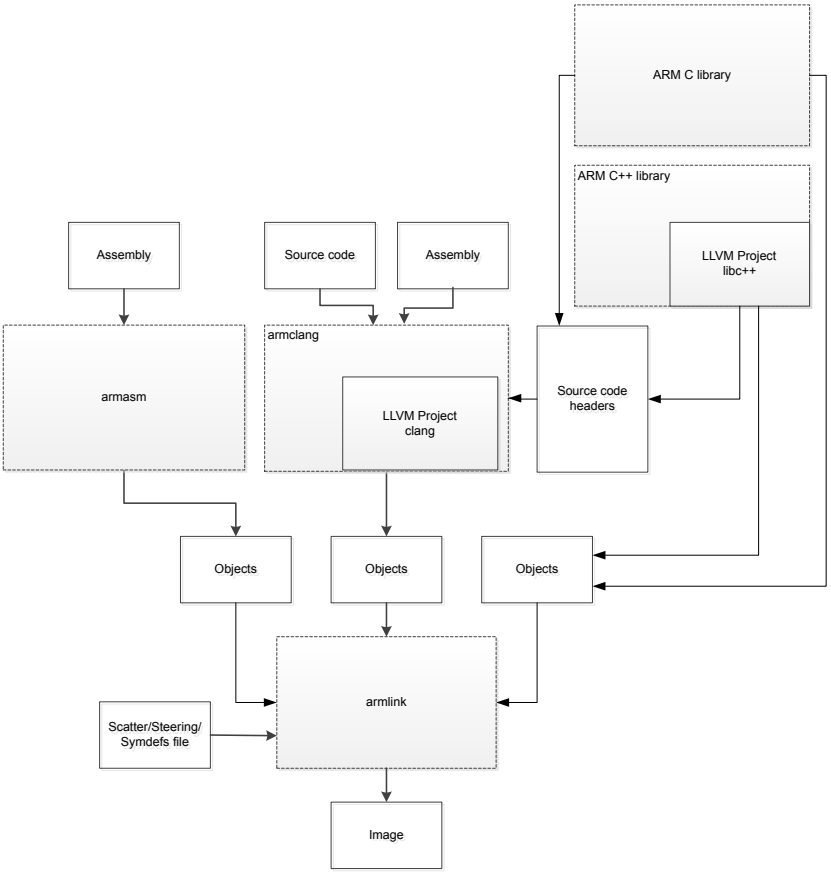


図 1-2 ARM コンパイラ 6 の統合境界

破線のボックスはツールチェーン コンポーネントで、それらのコンポーネント間の関係が統合境界です。統合境界に広がるコミュニティ機能には、機能上、有意な制限が存在する場合があります。ARM コンパイラ 6 のサポート対象になっている規格のいずれかで関係が符号化されている場合は例外的に制限がありません。「*ARM® アーキテクチャ用アプリケーションバイナリインタフェース*

(ABI)」を参照して下さい。統合境界をまたがないコミュニティ機能は、正しく機能する可能性が高くなります。
- Linux や BSD などのホステッド環境をターゲットとする場合に主に使用する機能には有意な制限があるか、ベアメタル環境をターゲットとする場合は適用できない可能性があります。
- コンパイラ機能の Clang 実装の中でも、特に、他のツールチェーンに長期間存在しているものは完成度が高い可能性があります。新しい言語機能のサポートなど、新機能の機能性は完成度がさほど高くないため、機能が制限されることが多くあります。

サポート対象外機能

コミュニティ機能とコミュニティ機能の双方のカテゴリを使用した場合、特定の機能やユースケースが正常に機能しないことが確認されています。また、この機能やユースケースは ARM コンパイラ 6 で使用することを目的としていません。

製品機能の制限はマニュアルに記載されています。完全なサポート対象外機能のリストやコミュニティ機能のユースケースを ARM が提供することはできませんが、コミュニティ機能の既知の制限については「*コミュニティ機能*(1-13 ページ)」に列挙してあります。

既知のサポート対象外機能の一覧

以下の一覧はサポート対象外機能の一部です。時間の経過とともに内容が変更される可能性があります。
- Clang オプション `-stdlib=libstdc++` はサポートされていません。
- ローカル変数の C++ のスタティックな初期化は、標準 C++ ライブラリにリンクされる場合スレッドセーフではありません。スレッドセーフティを実現するには、『*標準 C++ ライブラリの実装定義*』で説明されているようにスレッドセーフな関数を独自に実装する必要があります。

  注

  この制限は、[ALPHA] でサポートされているマルチスレッド C++ ライブラリには適用されません。詳細については、ARM のサポート窓口までお問い合わせください。
- C11 ライブラリ機能の使用はサポート対象外です。
- 非 ARM アーキテクチャのみに関連するコミュニティ機能はすべて、ARM コンパイラ 6 のサポート対象外です。
- ARMv7 または ARMv6-M よりも古いアーキテクチャを実装したターゲットのコンパイルはサポートされていません。

1.3 LLVM コンポーネントのバージョンと言語の互換性

armclang は、LLVM コンポーネントに基づき、異なるソース言語基準に対して異なるレベルのサポートを提供します。

——— 注 ———

このトピックでは、[ALPHA] 機能および [COMMUNITY] 機能について説明します。詳細については、「*サポートレベルの定義* (1-13 ページ)」を参照して下さい。

———————————

ベース LLVM コンポーネント

ARM コンパイラ 6 は以下の LLVM コンポーネントに基づきます。

表 1-1 LLVM コンポーネントのバージョン

コンポーネント	バージョン	その他の情報
Clang	3.8	*http://clang.llvm.org*

言語のサポートレベル

libc++ を使用する ARM コンパイラ 6 は、異なるソース言語標準に対して異なるレベルのサポートを提供します。

表 1-2 言語のサポートレベル

言語標準	サポートレベル
C90	サポートされています。
C99	複素数以外でサポート
[COMMUNITY] C11	ベース Clang コンポーネントは、C11 言語機能を提供します。ただし、ARM はこれらの機能の個別のテストを実行していないため、これらはコミュニティ機能となっています。C11 ライブラリ機能の使用はサポート対象外です。 C11 は C コードのデフォルトの言語標準です。ただし、新しい C11 言語機能の使用はコミュニティ機能となっています。必要に応じて、`-std` オプションを使用して言語標準を制限して下さい。`-Wc11-extensions` オプションを使用すると、C11 固有の機能に関する警告を表示できます。
C++98	C++ 例外の使用を含めてサポートされます。 `-fno-exceptions` に対するサポートは、制限されています。 例外のサポートの詳細については、*『ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド』*の*「標準 C++ ライブラリの実装定義」*を参照して下さい。

表 1-2 言語のサポートレベル（続き）

言語標準	サポートレベル
C++11	以下の例外でサポートされています。 [ALPHA] 以下の標準ライブラリ ヘッダによって使用できる並列構造が [ALPHA] でサポートされます。 • `<thread>` • `<mutex>` • `<shared_mutex>` • `<condition_variable>` • `<future>` • `<chrono>` • `<atomic>` • 詳細については、ARM のサポート窓口までお問い合わせください。 詳細については、*『ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド』*の*「標準 C++ ライブラリの実装定義」*を参照して下さい。
[COMMUNITY] C++14	ベース Clang および libc++ コンポーネントは、C++14 言語機能を提供します。ただし、ARM はこれらの機能の個別のテストを実行していないため、これらはコミュニティ機能となっています。

追加情報

ARM 特定の言語拡張の詳細については、*『armclang リファレンスガイド』*を参照して下さい。

libc++ のサポートの詳細については、*『ARM C ライブラリ、C++ ライブラリ、および浮動小数点サポートユーザガイド』*の*「標準 C++ ライブラリの実装定義」*を参照して下さい。

Clang ドキュメントには、言語の互換性に関する追加情報が記載されています。

- 言語の互換性:

  http://clang.llvm.org/compatibility.html
- 言語拡張機能:

  http://clang.llvm.org/docs/LanguageExtensions.html
- C++ のステータス:

  http://clang.llvm.org/cxx_status.html

関連情報

armclang リファレンスガイド.

1.4 共通の ARM コンパイラツールチェーンのオプション

ARM コンパイラツールチェーンの各ツールのコマンドラインオプションの最も一般的な使用方法をリストします。

armclang の共通オプション

armclang コマンドラインオプションの詳細については、*『armclang リファレンスガイド』*を参照して下さい。

一般的に使用される armclang オプションは次のとおりです。

表 1-3 armclang の共通オプション

オプション	説明
-c	リンク手順ではなく、コンパイル手順を実行します。
-x	例えば、-xc inputfile.s または -xc++ inputfile.s などの後続のソースファイルの言語を指定します。
-std	コンパイルする言語標準を -std=c90 などのように指定します。
--target=*arch-vendor-os-abi*	選択された実行状態（AArch32 または AArch64）向けのコードを生成します。例えば、--target=aarch64-arm-none-eabi または --target=arm-arm-none-eabi のようになります。
-march=*name*	指定されたアーキテクチャ向けのコードを生成します。例えば、-mcpu=armv8-a または -mcpu=armv7-a のようになります。
-march=list	ターゲットのサポートされているすべてのアーキテクチャのリストが表示されます。
-mcpu=*name*	指定されたプロセッサ向けのコードを生成します。たとえば、-mcpu=cortex-a53、-mcpu=cortex-a57、または -mcpu=cortex-a15 のようになります。
-mcpu=list	ターゲットのサポートされているすべてのプロセッサのリストが表示されます。
-marm	A32 命令セットをターゲットとするようにコンパイラに要求します。例えば、--target=arm-arm-none-eabi -march=armv7-a -marm のようになります。 -marm オプションは、AArch64 ターゲットでは有効でありません。AArch64 ターゲットでは、コンパイラは -marm オプションを無視し、警告を生成します。
-mthumb	T32 命令セットをターゲットとするようにコンパイラに要求します。例えば、--target=arm-arm-none-eabi -march=armv8-a -mthumb のようになります。 -mthumb オプションは、AArch64 ターゲットでは有効でありません。AArch64 ターゲットでは、コンパイラは -mthumb オプションを無視し、警告を生成します。
-g	DWARF デバッグテーブルを生成します。
-E	プリプロセッサの手順のみを実行します。
-I	指定したディレクトリを、インクルードファイルの検索場所リストに追加します。
-o	出力ファイルの名前を指定します。
-O*num*	ソースファイルのコンパイル時に使用するパフォーマンス最適化レベルを指定します。

表 1-3 armclang の共通オプション（続き）

オプション	説明
-Os	コード速度に対するコードサイズのバランスを取ります。
-Oz	コードサイズを最適化します。
-S	コンパイラによって生成されたマシンコードの逆アセンブリを出力します。
-###	コンパイラとリンカを呼び出すために使用されるオプションを示す診断出力が表示されます。コンパイル手順もリンク手順のいずれも実行されません。

armlink の共通オプション

armlink コマンドラインオプションの詳細については、*『armlink ユーザガイド』*を参照して下さい。

一般的に使用される armlink オプションは次のとおりです。

表 1-4 armlink の共通オプション

オプション	説明
--ro_base	RO 出力セクションを含んでいる領域のロードアドレスと実行アドレスを指定のアドレスに設定します。
--rw_base	RW 出力セクションを含んでいる領域の実行アドレスを指定のアドレスに設定します。
--scatter	このオプションを使用すると、指定のファイルに含まれる分散ロード記述子を使用して、イメージのメモリマップが作成されます。
--split	RO 出力セクションと RW 出力セクションを含んでいるデフォルトのロード領域を別々の領域に分割します。
--entry	イメージ固有の初期エントリポイントを指定します。
--info	リンカ処理に関する情報を表示します。例えば、--info=exceptions と指定すると、例外テーブルの生成と最適化に関する情報が表示されます。
--list=*filename*	--info や --map を含むオプションの診断出力を指定のファイルに転送します。
--map	メモリマップを表示します。このマップには、イメージの各ロード領域、実行領域、および入力セクション（リンカによって生成された入力セクションを含む）のアドレスとサイズが含まれます。
--symbols	リンク手順で使用される各ローカルシンボルとグローバルシンボル、およびその値を一覧表示します。

armar の共通オプション

armar コマンドラインオプションの詳細については、*『armar ユーザガイド』*を参照して下さい。

一般的に使用される armar オプションは次のとおりです。

表 1-5 armar の共通オプション

オプション	説明
--debug_symbols	ライブラリにデバッグシンボルを含めます。
-a *pos_name*	ライブラリ内の新しいファイルをファイル *pos_name* の後に配置します。
-b *pos_name*	ライブラリ内の新しいファイルをファイル *pos_name* の前に配置します。

表 1-5 armar の共通オプション（続き）

オプション	説明
`-d file_list`	指定されたファイルをライブラリから削除します。
`--sizes`	ライブラリ内のメンバごとに、Code、RO Data、RW Data、ZI Data、および Debug のサイズが一覧表示されます。
`-t`	ライブラリの内容を表すテーブルを出力します。

fromelf の共通オプション

`fromelf` コマンドラインオプションの詳細については、*『fromelf ユーザガイド』*を参照して下さい。

一般的に使用される `fromelf` オプションは次のとおりです。

表 1-6 fromelf の共通オプション

オプション	説明
`--elf`	ELF 出力モードを選択します。
`--text [options]`	イメージ情報をテキスト形式で表示します。 オプションの *options* を使用して、イメージ情報に含める追加の情報を指定することができます。有効な *options* には、コードを逆アセンブルする `-c` や、シンボルとバージョン管理テーブルを出力するための `-s` などがあります。
`--info`	特定のトピックに関する情報を表示します。例えば、`--info=totals` では、イメージ内の各入力オブジェクトおよびライブラリメンバのコード、RO データ、RW データ、ZI データ、およびデバッグサイズの一覧が表示されます。

armasm の共通オプション

`armasm` コマンドラインオプションの詳細については、*『armasm ユーザガイド』*を参照して下さい。

——— 注 ———

ARM 構文を使用してレガシーアセンブリコードをアセンブルするには、`armasm` のみを使用します。新しいアセンブリファイルには GNU 構文を使用して、`armclang` アセンブラでアセンブルします。

———————

一般的に使用される `armasm` オプションは次のとおりです。

表 1-7 armasm の共通オプション

オプション	説明
`--cpu=name`	ターゲットのプロセッサを設定します。
`-g`	DWARF デバッグテーブルを生成します。
`--fpu=name`	ターゲットの浮動小数点ユニット（FPU）アーキテクチャを選択します。
`-o`	出力ファイルの名前を指定します。

1.5 "Hello world" の例

この例は、簡単な C プログラム hello_world.c を armclang および armlink を使用してビルドする方法を示しています。

手順

1. 以下のコンテンツを使用して、C ファイル hello_world.c を作成します。

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

2. C ファイル hello_world.c を次のコマンドでコンパイルします。

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c hello_world.c
```

-c オプションは、コンパイル手順のみを実行するようにコンパイラに指示します。-march=armv8-a オプションは、ARMv8-A アーキテクチャおよび --target=aarch64-arm-none-eabi ターゲットの AArch64 状態をターゲットにするようにコンパイラに指示します。

コンパイラはオブジェクト ファイル hello_world.o を作成します。

3. 次のように、ファイルをリンクします。

```
armlink -o hello_world.axf hello_world.o
```

-o オプションは、出力イメージに hello_world.axf という名前を付け、デフォルトのイメージ名 __image.axf を使用しないようにリンカに指示します。

4. DWARF 4 と互換性のあるデバッガを使用して、イメージをロードして実行します。

コンパイラは、DWARF 4 標準と互換性のあるデバッグ情報を生成します。

1.6 コンパイラからリンカにオプションを渡す

armclang を実行した場合、デフォルトでコンパイラは自動的にリンカである armlink を呼び出します。

多数の armclang オプションで、リンカの動作を制御します。これらのオプションは、同等の armlink オプションに変換されます。

表 1-8 armclang リンカ制御オプション

armclang オプション	armlink オプション	説明
-e	--entry	イメージ固有の初期エントリポイントを指定します。
-L	--userlibpath	リンカがユーザライブラリの検索に使用するパスのリストを指定します。
-l	--library	指定されたライブラリを検索対象ライブラリの一覧に追加します。
-u	--undefined	未定義のシンボルがある場合に、指定されたシンボルを削除できないようにします。

さらに、-Xlinker オプションと -Wl オプションを使用することで、コンパイラコマンドラインから直接リンカにオプションを渡すことができます。これらのオプションを使用することで、同じ機能が実行されますが、使用される構文は異なります。

- -Xlinker オプションは、1 つのオプション、1 つの引数、または 1 つの option=argument のペアを指定します。複数のオプションを渡す場合は、複数の -Xlinker オプションを使用します。
- -Wl, オプションでは、カンマ区切りのオプション、引数、または option=argument のペアの一覧を指定します。

例えば次の例では、armlink は 1 つのオプション --list=diag.txt と 2 つのオプション --list diag.txt を同等に扱うため、すべて同等です。

```
-Xlinker --list -Xlinker diag.txt -Xlinker --split

-Xlinker --list=diag.txt -Xlinker --split

-Wl,--list,diag.txt,--split

-Wl,--list=diag.txt,--split
```

注

-### コンパイラオプションは、コンパイラとリンカを呼び出す正しい方法を示す診断出力を生成し、各ツールのオプションを示します。-### オプションを使用した場合、armclang ではこの診断出力のみが表示されます。ソースファイルがコンパイルされたり、armlink が呼び出されたりすることはありません。

次の例は、-Xlinker オプションを使用して --split オプションをリンカに渡し、RO 出力セクションと RW 出力セクションを含むデフォルトのロード領域を別々の領域に分割する方法を示しています。

```
armclang hello.c --target=aarch64-arm-none-eabi  -Xlinker --split
```

fromelf --text を使用して、イメージコンテンツの相違を比較することができます。

```
armclang hello.c --target=aarch64-arm-none-eabi -o hello_DEFAULT.axf
armclang hello.c --target=aarch64-arm-none-eabi -o hello_SPLIT.axf -Xlinker --split

fromelf --text hello_DEFAULT.axf &gt; hello_DEFAULT.txt
fromelf --text hello_SPLIT.axf &gt; hello_SPLIT.txt
```

UNIX diff ツールなどのファイル比較ツールを使用して、hello_DEFAULT.txt および hello_SPLIT.txt ファイルを比較します。

第 2 章
診断

コンパイラツールチェイン診断メッセージの形式と診断出力の制御方法について説明します。

以下のセクションから構成されています。

- *2.1 診断結果の解釈*(2-24 ページ).
- *2.2 armclang による診断の制御オプション*(2-26 ページ).
- *2.3 armclang を使用して診断を制御するためのプラグマ*(2-27 ページ).
- *2.4 他のツールによる診断の制御オプション*(2-28 ページ).

2.1 診断結果の解釈

ARM コンパイラ 6 ツールチェーンのすべてのツールは詳細な診断メッセージを生成して、情報の出力量の制御を可能にします。

armclang での診断メッセージの形式と診断出力の制御メカニズムは、ツールチェインの他のツールと異なります。

armclang のメッセージ形式

armclang では、次の形式のメッセージが生成されます。

```
file:line:col:タイプ:message
```

各項目には以下の意味があります。

file
: メッセージを生成したファイル名。

行
: メッセージを生成した行番号。

列
: メッセージを生成した列番号。

タイプ
: メッセージの種類です。エラーや警告などがあります。

message
: メッセージのテキスト。

以下に例を示します。

```
hello.c:7:3:error:use of undeclared identifier 'i'
i++;
^
1 error generated.
```

他のツールのメッセージ形式

ツールチェインのその他のツール（armasm および armlink など）では、次の形式のメッセージが生成されます。

```
type:prefixidsuffix:message_text
```

各項目には以下の意味があります。

タイプ
: 次のいずれかを指定します。

内部エラー
: ツールで内部的な問題が発生していることを示します。フィードバック情報を用意して購入元にお問い合わせ下さい。

エラー
: ツールが停止する原因となる問題があることを示します。

警告
: 問題を引き起こす可能性のある例外的な状況を示す警告ですが、ツールは引き続き実行されます。

注釈
: ツールの使用方法が、一般的ではあるが場合によっては特殊であることを示します。この診断情報はデフォルトでは表示されません。ツール実行は継続されます。

prefix
: メッセージを生成したツールとして、以下のいずれかを示します。

- A - armasm
- L - armlink または armar
- Q - fromelf

id
一意の数値によるメッセージ識別子。

suffix
メッセージのタイプを、以下のいずれかから示します。

- E - エラー
- W - 警告
- R - 注釈

message_text
メッセージのテキスト。

以下に例を示します。

```
エラー:L6449E:While processing /home/scratch/a.out:I/O error writing file '/home/scratch/
a.out':パーミッションが拒否されました
```

関連概念

2.2 armclang による診断の制御オプション(2-26 ページ).
2.4 他のツールによる診断の制御オプション(2-28 ページ).

2.2 armclang による診断の制御オプション

多数のオプションで、armclang コンパイラによる診断の出力を制御します。

armclang を使用した診断の制御の詳細については、*「Clang Compiler User's Manual」*の*「Controlling Errors and Warnings」*を参照してください。

診断を制御する際に一般的に使用されるオプションのいくつかを以下に示します。

`-Werror`
: 警告をエラーに変換します。

`-Werror=foo`
: 警告 foo をエラーに変換します。

`-Wno-error=foo`
: -Werror が指定されていても、警告 foo を警告のまま残します。

`-Wfoo`
: 警告 foo を有効にします。

`-Wno-foo`
: 警告 foo を抑制します。

`-w`
: すべての警告を抑制します。

`-Weverything`
: すべての警告を有効にします。

メッセージを抑制できるところでは、コンパイラは診断出力に適切な抑制フラグを提供します。

例えば、デフォルトでは armclang は printf() ステートメントの形式を調べて、% 形式指定子の数が、データ引数の数と一致することを確認します。以下のコードによって警告が生成されます。

```
printf("Result of %d plus %d is %d\n", a, b);
```

```
armclang --target=aarch64-arm-none-eabi -c hello.c
hello.c:25:36: warning: more '%' conversions than data arguments [-Wformat]
  printf("Result of %d plus %d is %d\n", a, b);
```

この警告を抑制するには、-Wno-format を使用します。

```
armclang --target=aarch64-arm-none-eabi -c hello.c -Wno-format
```

関連参照

章 7 コード作成時の注意事項(7-54 ページ).

関連情報

LLVM コンパイラインフラストラクチャプロジェクト.
Clang コンパイラユーザガイド.

2.3 armclang を使用して診断を制御するためのプラグマ

ソース コード内のプラグマによって armclang コンパイラからの診断の出力を制御できます。

armclang を使用した診断の制御の詳細については、*「Clang Compiler User's Manual」*の*「Controlling Errors and Warnings」*を参照してください。

以下に、診断を制御する一般的なオプションの一部を示します。

`#pragma clang diagnostic ignored "-W`*name*`"`
: *name* で指定された診断メッセージを無視します。

`#pragma clang diagnostic warning "-W`*name*`"`
: *name* で指定された診断メッセージの重大度を警告に設定します。

`#pragma clang diagnostic error "-W`*name*`"`
: *name* で指定された診断メッセージの重大度をエラーに設定します。

`#pragma clang diagnostic fatal "-W`*name*`"`
: *name* で指定された診断メッセージの重大度を致命的エラーに設定します。

`#pragma clang diagnostic push`
: 復元できるように診断状態を保存します。

`#pragma clang diagnostic pop`
: 最後に保存された診断状態を復元します。

コンパイラは、診断の出力に適切な診断名を指定します。

注

または、コマンドライン オプション `-W`*name* を使用してメッセージの重大度を非表示にしたり変更することができます。ただし、変更はコンパイル全体に適用されます。

関連情報

-W.

2.4 他のツールによる診断の制御オプション

多数のさまざまなオプションを使用して、armasm、armlink、armar、および fromelf などのツールの診断を制御します。

以下のオプションを使用して、診断を制御します。

--brief_diagnostics
: armasm のみ。短い形式の診断出力を使用します。この形式では、元のソース行は表示されず、1 行に収まらないエラーメッセージは折り返されません。

--diag_error=*tag*[,*tag*]...
: 指定された診断メッセージをエラーの重大度に設定します。すべての警告をエラーとして取り扱うには、--diag_error=warning を使用します。

--diag_remark=*tag*[,*tag*]...
: 指定された診断メッセージの注釈の重要度を設定します。

--diag_style=arm|ide|gnu
: 診断メッセージの表示スタイルを指定します。

--diag_suppress=*tag*[,*tag*]...
: 指定された診断メッセージを非表示にします。ダウンロード可能なすべてのエラーを非表示にするには --diag_suppress=error を使用し、すべての警告を非表示にするには --diag_suppress=warning を使用します。

--diag_warning=*tag*[,*tag*]...
: 指定された診断メッセージを警告の重大度に設定します。ダウンロード可能なすべてのエラーを警告に設定するには、--diag_warning=error を使用します。

--errors=filename
: 診断メッセージの出力を、指定されたファイルに転送します。

--remarks
: armlink 専用です。注釈メッセージの表示を有効にします(--diag_remark を使用して注釈の重要度を再指定したすべてのメッセージを含む)。

tag は 4 桁の診断番号、 *nnnn* で、ツールの接頭文字は指定されていますが、重大度を示す接尾文字はありません。

例えば、警告メッセージを注釈の重大度に降格させる場合に使用します。

```
$ armasm test.s --cpu=8-A.32
"test.s", line 55: Warning: A1313W: ファイルの終わりに END ディレクティブがありません
0 Errors, 1 Warning

$ armasm test.s --cpu=8-A.32 --diag_remark=A1313
"test.s", line 55: ファイルの終わりに END ディレクティブがありません
```

第 3 章
C および C++ コードのコンパイル

armclang を使用して C および C++ コードをコンパイルする方法について説明します。

以下のセクションから構成されています。

3.1 ターゲットアーキテクチャ、プロセッサ、命令セットの指定

コードをコンパイルする際、コンパイラはターゲットとするアーキテクチャまたはプロセッサ、使用可能なオプションのアーキテクチャ機能、および使用する命令セットを認識する必要があります。

概要

ある特定のプロセッサ上でのみコードを実行する場合は、その特定のプロセッサをターゲットに指定できます。パフォーマンスは最適化されますが、コードの動作はそのプロセッサ上でしか保証されません。

さまざまなプロセッサ上でコードを実行する場合は、アーキテクチャをターゲットに指定できます。コードはターゲットアーキテクチャに実装されたどのプロセッサ上でも動作しますが、パフォーマンスが影響を受ける場合があります。

ターゲットの指定には、以下のオプションがあります。

1. `--target` オプションを使用して、実行状態を指定します。

   実行状態は、プロセッサに応じて AArch64 または AArch32 になります。
2. 以下のいずれかをターゲットにします。
   - `-march` オプションを使用したアーキテクチャ。
   - `-mcpu` オプションを使用した特定のプロセッサ。
3. （AArch32 ターゲットのみ）`-mfpu` オプションを使用してターゲットで使用可能な浮動小数点ハードウェアを指定するか、省略してターゲットにデフォルトを使用します。
4. (AArch32 ターゲットのみ) ARM と Thumb の両方をサポートするプロセッサでは、`-marm` または `-mthumb` を使用して命令セットを指定するか、省略してデフォルトの `-marm` にします。

ターゲット実行状態の指定

`armclang` でターゲット実行状態を指定するには、`--target` コマンドラインオプションを使用します。

`--target=`*`arch-vendor-os-abi`*

サポートされているターゲットは以下のとおりです。

`aarch64-arm-none-eabi`
: AArch64 状態の A64 命令を生成します。`-mcpu` が指定されている場合を除き、`-march=armv8-a` を指定したと見なされます。

`arm-arm-none-eabi`
: AArch32 状態の A32/T32 命令を生成します。`-march`（アーキテクチャをターゲットにする場合）または `-mcpu`（プロセッサをターゲットにする場合）と共に使用する必要があります。

——— 注 ———

`--target` オプションは、`armclang` オプションです。その他すべてのツール（`armasm` や `armlink` など）では、`--cpu` および `--fpu` オプションを使用して、ターゲットプロセッサやターゲットアーキテクチャを指定して下さい。

———————

——— 注 ———

`--target` オプションは必須です。ターゲット実行状態を必ず指定して下さい。

———————

ターゲットアーキテクチャの指定

`--target` および `-march` を使用してターゲットアーキテクチャを指定すると、そのアーキテクチャを持つどのプロセッサ上でも動作する汎用コードが生成されます。

サポートされているアーキテクチャをすべて表示するには、-march=list オプションを使用します。

注

-march オプションは、armclang オプションです。その他すべてのツール（armasm や armlink など）では、--cpu および --fpu オプションを使用して、ターゲットプロセッサやターゲットアーキテクチャを指定して下さい。

特定のプロセッサの指定

--target および -mcpu でプロセッサをターゲットすると、指定したプロセッサのコードが最適化されます。

サポートされているプロセッサをすべて表示するには、-mcpu=list オプションを使用します。

-mcpu および -march を使用して、機能の修飾子を指定できます。たとえば、-mcpu=cortex-a57+nocrypto と指定できます。

ターゲットで使用可能な浮動小数点ハードウェアの指定

-mfpu オプションは、ターゲットアーキテクチャまたはプロセッサによって暗示されるデフォルトの FPU オプションをオーバーライドします。

注

-mfpu オプションは、ARMv8-A の AArch64 ターゲットでは無視されます。AArch64 ターゲットのデフォルトの FPU をオーバーライドするには、-mcpu オプションを使用します。たとえば、AArch64 ターゲットに対して暗号拡張機能を使用しないようにするには、-mcpu=*name*+nocrypto オプションを使用します。

命令セットの指定

以下のようにアーキテクチャによって異なる命令セットがサポートされています。

- AArch64 状態の ARMv8-A プロセッサは、A64 命令を実行します。
- AArch32 状態の ARMv8-A プロセッサ、ならびに ARMv7 以前の A プロファイルプロセッサと R プロファイルプロセッサは、A32（以前の ARM）および T32（以前の Thumb）命令を実行します。
- M プロファイルプロセッサは、T32（以前の Thumb）命令を実行します。

ターゲット命令セットを指定するには、次のコマンドラインオプションを使用します。

- -marm は、A32（以前の ARM）命令セットをターゲットとします。これは、ARM 命令または A32 命令をサポートするすべてのターゲットのデフォルトです。
- -mthumb は、T32（以前の Thumb）命令セットをターゲットとします。これは、Thumb または T32 命令のみをサポートしているすべてのターゲットに対するデフォルトです。

注

-marm および -mthumb オプションは、AArch64 ターゲットでは有効でありません。AArch64 ターゲットでは、コンパイラは -marm および -mthumb オプションを無視し、警告を生成します。

コマンドラインの例

以下の例は、アーキテクチャ、プロセッサ、および命令セットのさまざまな組み合わせ向けにコンパイルする方法を示しています。

表 3-1 アーキテクチャ、プロセッサ、および命令セットのさまざまな組み合わせ向けのコンパイル

アーキテクチャ	プロセッサ	命令セット	armclang コマンド
ARMv8-A の AArch64 状態	汎用	A64	`armclang --target=aarch64-arm-none-eabi test.c`
ARMv8-A の AArch64 状態	Cortex® -A57	A64	`armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 test.c`
ARMv8-A の AArch32 状態	汎用	A32	`armclang --target=arm-arm-none-eabi -march=armv8-a test.c`
ARMv8-A の AArch32 状態	Cortex -A53	A32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 test.c`
ARMv8-A の AArch32 状態	Cortex -A57	T32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-a57 -mthumb test.c`
ARMv7-A	汎用	A32	`armclang --target=arm-arm-none-eabi -march=armv7-a test.c`
ARMv7-A	Cortex -A9	A32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-r7 test.c`
ARMv7-A	Cortex -A15	T32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-r7 -mthumb test.c`
ARMv7-R	Cortex -R7	A32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-r7 test.c`
ARMv7-R	Cortex -R7	T32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-r7 -mthumb test.c`
ARMv7-M	汎用	T32	`armclang --target=arm-arm-none-eabi -march=armv7-m test.c`
ARMv6-M	Cortex -M0	T32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-m0 test.c`
ARMv8-M.Mainline	汎用	T32	`armclang --target=arm-arm-none-eabi -march=armv8-m.main test.c`
ARMv8-M.Baseline	汎用	T32	`armclang --target=arm-arm-none-eabi -march=armv8-m.base test.c`

関連情報

-mcpu.
--target.
-marm.
-mthumb.
-mfpu.

3.2 インラインアセンブリコードの使用

コンパイラのインラインアセンブラを使用することにより、最適なアセンブリ言語ルーチンを記述し、C または C++ で利用できないターゲットプロセッサの機能にアクセスできます。

__asm キーワードを使用して、インライン GCC 構文アセンブリコードを関数に組み込むことができます。以下に例を示します。

```
#include <stdio.h>

int add(int i, int j)
{
  int res = 0;
  __asm (
    "ADD %[result], %[input_i], %[input_j]"
    :[result] "=r" (res)
    :[input_i] "r" (i), [input_j] "r" (j)
  );
  return res;
}

int main(void)
{
  int a = 1;
  int b = 2;
  int c = 0;

  c = add(a,b);

  printf("Result of %d + %d = %d\n", a, b, c);
}
```

注

インラインアセンブラは ARM アセンブラ構文で記述されたレガシーアセンブリコードをサポートしてません。ARM 構文アセンブリコードの GCC 構文への移行の詳細については、*『移行と互換性ガイド』*を参照して下さい。

__asm インラインアセンブリステートメントの一般的な形式は以下のとおりです。

__asm(*code [:output_operand_list [:input_operand_list [:clobbered_register_list]]]*);

code はアセンブリコードです。この例では、これは "ADD %[result], %[input_i], %[input_j]" です。

output_operand_list は、コンマで区切られた、出力オペランドのオプションのリストです。各オペランドは、角括弧で囲まれたシンボリック名、制約文字列、括弧で囲まれた C 式で構成されます。この例として、[result] "=r" (res) という 1 つの出力オペランドがあります。

input_operand_list は、コンマで区切られたオプションの入力オペランドのリストです。入力オペランドは、出力オペランドと同じ構文を使用します。この例では、2 つの入力オペランドがあります。[input_i] "r" (i), [input_j] "r" (j).

clobbered_register_list は、上書きされたレジスタのオプションリストです。この例では、これは省略します。

関連情報

ARM 構文アセンブリコードから GNU 構文への移行.

3.3 組み込み関数の使用

コンパイラ組み込み関数は、コンパイラによって提供される関数です。コンパイラ組み込み関数を使用すると、アセンブリ言語での複雑な実装手段をとらずに、C および C++ ソースコードでドメイン固有の演算を簡単に組み込むことができます。

C 言語および C++ 言語は、広範囲に及ぶタスクに適していますが、例えば*デジタル信号処理*(DSP)などのアプリケーションの特定エリアのサポートは組み込まれていません。

特定のアプリケーションドメイン内には、通常、頻繁に実行する必要のあるドメイン固有のさまざまな演算があります。ただし、これらの演算は C または C++ では効率的に実装できない場合もよくあります。代表的な例としては、通常 DSP プログラミングで使用する 2 の補数の 32 ビット符号付き整数のサチュレート加算があります。サチュレート加算に C を実装した例を以下に示します。

```
#include <limits.h>
int L_add(const int a, const int b)
{
    int c;
    c = a + b;
    if (((a ^ b) & INT_MIN) == 0)
    {
        if ((c ^ a) & INT_MIN)
        {
            c = (a < 0) ?INT_MIN :INT_MAX;
        }
    }
    return c;
}
```

コンパイラ組み込み関数を使用すると、コンパイラの命令選択よりも包括的にターゲットアーキテクチャの命令を実行できます。

コンパイラ組み込み関数は、C または C++ の関数呼び出しのように見えますが、コンパイル時に特定のシーケンスのローレベルの命令によって置換されます。以下の例は、__qadd サチュレート加算組み込み関数へのアクセス方法を示しています。

```
#include <arm_acle.h>    /* ACLE 組み込み関数をインクルードする */

int foo(int a, int b)
{
  return __qadd(a, b);  /* a と b のサチュレート加算 */
}
```

コンパイラ組み込み関数を使用すると、多くのパフォーマンス上の利点が得られます。

- コンパイラ組み込み関数をローレベルのコンパイラ組み込み関数に置き換えると、C または C++ での対応する実装よりも効率が良くなり、命令数とサイクル数の両方が削減されます。コンパイラ組み込み関数を実装するために、コンパイラでは指定されたターゲットアーキテクチャに対して最適な命令のシーケンスが自動的に生成されます。例えば __qadd 組み込み関数は、A32 アセンブリ言語命令 qadd に直接対応します。

  ```
  QADD r0, r0, r1    /* エントリ時に r0 = a、r1 = b と想定 */
  ```

- コンパイラには、ベースとなっている C および C++ 言語の伝達能力を超えた多くの情報が渡されます。これにより、コンパイラは通常であれば実行不可能な最適化を実行し、命令シーケンスを生成します。

パフォーマンス上のこれらの利点は、リアルタイム処理アプリケーションにとっては重要です。ただし、コンパイラ組み込み関数を使用するとコードの移植性が低下するため、注意して使用する必要があります。

3.4 浮動小数点命令とレジスタ使用の防止

浮動小数点命令または浮動小数点レジスタを使用しないように、コンパイラに指示することができます。

浮動小数点の計算とリンケージ

浮動小数点計算は、以下の方法で実行できます。

- ハードウェアコプロセッサによって実行される浮動小数点命令。結果のコードは、ベクタ浮動小数点(VFP)コプロセッサハードウェアを搭載したプロセッサ上でのみ実行できます。
- 浮動小数点ライブラリ `fplib` を使用したソフトウェアライブラリ関数。このライブラリには、追加のハードウェアを使用することなく、浮動小数点演算を実行するために呼び出すことができる関数が含まれています。

ハードウェア浮動小数点命令を使用するコードの方が、ソフトウェアで浮動小数点演算を行うコードより小型であり、パフォーマンスも優れています。ただし、ハードウェア浮動小数点命令には VFP コプロセッサが必要です。

浮動小数点リンケージは、浮動小数点パラメータと戻り値を渡すために使用するレジスタを制御します。

- ソフトウェア浮動小数点リンケージは、関数のパラメータと戻り値が r0 ～ r3 の ARM 整数レジスタおよびスタックを使用して渡されることを意味します。ソフトウェア浮動小数点リンケージを使用する利点は次のとおりです。
  - VFP コプロセッサの使用にかかわらず、コードをプロセッサ上で実行できる。
  - ソフトウェア浮動小数点リンケージ向けにコンパイルされたライブラリに対してコードをリンクできる。
- ハードウェア浮動小数点リンケージは、VFP コプロセッサレジスタを使用して引数と戻り値を渡します。ハードウェア浮動小数点リンケージを使用する利点は、ソフトウェア浮動小数点リンケージよりも効率がよいことですが、VFP コプロセッサが必要です。

浮動小数点命令とレジスタの使用の設定

AArch64 状態向けにコンパイルする場合は、以下のようになります。

- デフォルトでは、コンパイラはハードウェア浮動小数点命令およびハードウェア浮動小数点リンケージを使用します。
- `-mcpu=`*name*`+nofp+nosimd` オプションを使用して、浮動小数点命令と浮動小数点レジスタの両方を使用しないようにします。

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53+nofp+nosimd test.c
```

  このモードで、その後に浮動小数点データ型を使用することはできません。

AArch32 状態向けにコンパイルする場合は、以下のようになります。

- `--target=arm-arm-none-eabi` を使用すると、コンパイラはハードウェア浮動小数点命令およびソフトウェア浮動小数点リンケージを使用します。これは、オプション `-mfloat-abi=softfp` に対応します。
- `-mfloat-abi=soft` オプションを使用して、浮動小数点演算およびソフトウェア浮動小数点リンケージ向けのソフトウェア ライブラリ関数を使用します。

```
armclang --target=arm-arm-none-eabi -march=armv8-a -mfloat-abi=soft test.c
```

- `-mfloat-abi=hard` オプションを使用して、ハードウェア浮動小数点命令およびハードウェア浮動小数点リンケージを使用します。

```
armclang --target=arm-arm-none-eabi -march=armv8-a -mfloat-abi=hard test.c
```

関連情報

-mcpu.

-mfloat-abi.

-mfpu.
浮動小数点サポートについて.

3.5 ベアメタル位置非依存実行可能ファイル

ベアメタル位置非依存実行可能ファイル(PIE)は、特定のアドレスで実行する必要はありませんが、適切に整列されたアドレスで実行できる実行可能ファイルです。

——— 注 ———

- ベアメタル PIE のサポートは廃止される予定です。
- armclang では -fropi および -frwpi がサポートされています。これらのオプションを使用して、ベアメタル位置非依存実行可能ファイルを作成できます。

———————————

位置非依存コードは、可能であれば PC 相対アドレシングモードを使用します。可能でない場合は、*グローバルオフセットテーブル*(GOT)を介してグローバルデータにアクセスします。GOT 内のアドレスエントリとデータ領域内の初期化されたポインタは、実行可能ファイルの初回実行時に、実行可能ファイルのロードアドレスによって更新されます。

イメージにリンクされているすべてのオブジェクトとライブラリをコンパイルして位置非依存にする必要があります。

ベアメタル PIE のコンパイルとリンク

以下の簡単なサンプルコードを考えてみます。

```
#include <stdio.h>

int main(void)
{
  printf(“hello\n”);
  return 0;
}
```

このコードをベアメタル PIE 用にコンパイルし、自動的にリンクするには、armclang と共に -fbare-metal-pie オプションを使用します。

```
armclang -fbare-metal-pie --target=arm-arm-none-eabi -march=armv8-a hello.c -o hello
```

または、別個の手順として、armclang -fbare-metal-pie を使用してコンパイルし、armlink --bare_metal_pie を使用してリンクすることができます。

```
armclang -fbare-metal-pie --target=arm-arm-none-eabi -march=armv8-a -c hello.c
armlink --bare_metal_pie hello.o -o hello
```

結果の実行可能ファイル hello は、ベアメタル位置非依存実行可能ファイルです。

——— 注 ———

ベアメタル PIE に含めるように armcc を使用してコンパイルされるレガシーコードは、オプション --apcs=/fpic を使用してコンパイルする必要があります。または、グローバルデータへの参照が含まれていない場合は、オプション --apcs=/ropi を使用してコンパイルできます。

———————————

リンク時最適化を使用する場合は、以下のように armlink --lto_relocation_model=pic オプションを使用して、位置非依存コードを生成するようにリンク時オプティマイザに指示します。

```
armclang -flto -fbare-metal-pie --target=arm-arm-none-eabi -march=armv8-a -c hello.c -o
hello.bc
armlink --lto --lto_relocation_model=pic --bare_metal_pie hello.bc -o hello
```

制約条件

ベアメタル PIE 実行可能ファイルは、以下に従っている必要があります。

- AArch32 状態のみ。
- .got セクションは、書き込み可能な領域内になければなりません。
- シンボルへのすべての参照は、リンク時に解決される必要があります。

- イメージは、0x0 のベースアドレスを持つ位置非依存コードにリンクする必要があります。
- コードとデータは、互いに固定オフセットでリンクする必要があります。
- スタックは、ランタイム再配置ルーチン `__arm_relocate_pie_` が呼び出される前に設定する必要があります。つまり、スタック初期化コードがイメージコードの一部である場合、PC 相対アドレシングのみを使用する必要があります。
- ターゲットプラットフォームは、PIE をロードして ZI 領域が確実にゼロで初期化されるようにする必要があります。
- 位置非依存のアセンブリコードを記述する場合は、一部の命令（例えば、`LDR`）を使用するとラベルの形式で PC 相対アドレスを指定できることに注意してください。例えば、

  ```
  LDR r0,=__main
  ```

  シンボルが読み出し専用のセクションにあるため、これは `--bare-metal-pie` でビルドする場合にリンク手順が失敗する原因になります。回避方法として、書き込み可能なセクションに間接的にシンボルを指定することができます。以下に例を示します。

  ```
      LDR r0, __main_addr
  ...
      AREA WRITE_TEST, DATA, READWRITE
  __main_addr DCD __main
      END
  ```

スキャッタファイルの使用

スキャッタファイルの例を示します。

```
LR 0x0 PI
{
    er_ro +0 { *(+RO) }
    DYNAMIC_RELOCATION_TABLE +0 { *(DYNAMIC_RELOCATION_TABLE) }

    got +0 { *(.got) }
    er_rw +0 { *(+RW) }
    er_zi +0 { *(+ZI) }

    ; ユーザ指定のスタック/ヒープ初期化ルーチンによって要求された
    ; スタックおよびヒープセクションをここに追加
}
```

リンカによって `DYNAMIC_RELOCATION_TABLE` セクションが生成されます。このセクションは、`DYNAMIC_RELOCATION_TABLE` という実行領域に配置する必要があります。これにより、C ライブラリで提供されているランタイム再配置ルーチン `__arm_relocate_pie_` は、シンボル `Image$$DYNAMIC_RELOCATION_TABLE$$Base` および `Image$$DYNAMIC_RELOCATION_TABLE$$Limit` を使用してテーブルの始点と終点を探すことができます。

スキャッタファイルと C ライブラリで提供されているデフォルトのエントリコードを使用する場合、ユーザはスタックとヒープを初期化するための独自のルーチンをリンカで指定する必要があります。このユーザ指定のスタックおよびヒープルーチンは `__arm_relocate_pie_` ルーチンの前に実行されるため、このルーチンが PC 相対アドレシングのみを使用するようにする必要があります。

関連情報

--fpic.
--pie.
--bare_metal_pie.
--ref_pre_init.
-fbare-metal-pie.
-fropi.
-frwpi.

3.6 実行専用メモリマップ

実行専用メモリ(XOM) では命令フェッチしかできません。読み出しと書き込み アクセスはできません。

実行専用メモリは、実行可能コードをユーザに読まれないようにして、知的所有権を保護します。例えば、ファームウェアを実行専用メモリに入れて、ユーザコードとドライバを別に読み込むことができます。ファームウェアを実行専用メモリに入れておくと、ユーザは手軽にコードを読み取ることができません。

注

ARM アーキテクチャは実行専用メモリを直接サポートしません。実行専用メモリはメモリデバイスレベルでサポートされます。

3.7 実行専用メモリ用のアプリケーションのビルド

コードを実行専用メモリに入れておくと、ユーザは手軽にそのコードを読み取ることができません。

——— 注 ———

リンク時最適化は armclang -mexecute-only オプションを実行しません。armclang -flto オプションまたは -Omax オプションを使用すると、コンパイラは実行専用コードを生成できません。

———————————

コードを実行専用メモリに入れたアプリケーションをビルドするには

手順

1. -mexecute-only オプションを使用して C または C++ コードをコンパイルします。
   ```
   armclang --target=arm-arm-none-eabi -march=armv7-m -mexecute-only -c test.c -o test.o
   ```
   -mexecute-only オプションは、コンパイラがコード セクションにデータ アクセスを生成することを防ぎます。

   コードとデータを別々のセクションに保持するために、コンパイラはコードによるリテラル プール インラインの配置を無効にします。

   ELF オブジェクト ファイルのコンパイルされた実行専用コード セクションは、SHF_ARM_NOREAD フラグでマークされます。
2. 以下のいずれかを使用してリンカへのメモリマップを指定します。
   - スキャッタ ファイルの +XO セレクタ。
   - コマンドラインの armlink --xo-base オプション

   ```
   armlink --xo-base=0x8000 test.o -o test.axf
   ```
   XO 実行領域は、RO、RW、および ZI 実行領域の 別々のロード領域に配置されます。

   ——— 注 ———

   --xo-base を指定しない場合、デフォルトでは以下のようになります。
   - XO 実行領域は RO 実行領域の直前に配置されます。アドレスは 0x8000 です。
   - すべての実行領域は同じロード領域内にあります。

   ———————————

第 4 章
アセンブリコードのアセンブル

armclang および armasm を使用してアセンブリソースコードをアセンブルする方法について説明します。

以下のセクションから構成されています。

- *4.1 ARM 構文と GNU 構文のアセンブリコードをアセンブルする*(4-42 ページ).
- *4.2 アセンブリコードの前処理*(4-44 ページ).

4.1 ARM 構文と GNU 構文のアセンブリコードをアセンブルする

The ARM コンパイラ 6 ツールチェーンでは、ARM 構文と GNU 構文の両方のアセンブリ言語ソースコードをアセンブルできます。

ARM と GNU は、アセンブリ言語ソースコードに使用される 2 つの異なる構文です。この 2 つは似ていますが、いくつかの相違点があります。例えば、ARM 構文では行の先頭の位置によってラベルが識別されますが、GNU 構文ではコロンの存在によってラベルが識別されます。

—————— 注 ——————

*『GNU Binutils - 用途』*ドキュメントでは、GNU 構文のアセンブリコードについて詳しく説明されています。

*『移行と互換性ガイド』*では、従来のアセンブリコードの移行に役立つよう、ARM 構文および GNU 構文のアセンブリの相違点について詳しく説明されています。

————————————

次の例は、ループ内でレジスタをインクリメントするための ARM 構文と GNU 構文の同等のアセンブリコードを示しています。

ARM 構文アセンブリ:

```
; 単純な ARM 構文のサンプル
;
; ループのラウンドを 10 回反復し、1 をレジスタに毎回加算します。

        AREA ||.text||, CODE, READONLY, ALIGN=2

main PROC
        MOV      w5,#0x64       ; W5 = 100
        MOV      w4,#0          ; W4 = 0
        B        test_loop      ; test_loop に分岐
loop
        ADD      w5,w5,#1       ; 1 を W5 に加算
        ADD      w4,w4,#1       ; 1 を W4 に加算
test_loop
        CMP      w4,#0xa        ; W4 &lt; 10 なら、分岐して loop に戻る
        BLT      loop
        ENDP

        END
```

ARM 構文を使用するレガシーアセンブリソースファイルがある場合があります。レガシー ARM 構文アセンブリコードをアセンブルするには armasm を使用します。通常、armasm アセンブラを起動するには以下のように入力します。

```
armasm --cpu=8-A.64 -o file.o file.s
```

GNU 構文アセンブリ:

```
// 単純な GNU 構文のサンプル
//
// ループのラウンドを 10 回反復し、1 をレジスタに毎回加算します。

        .section .text,"x"
        .balign 4

main:
        MOV      w5,#0x64       // W5 = 100
        MOV      w4,#0          // W4 = 0
        B        test_loop      // test_loop に分岐
loop:
        ADD      w5,w5,#1       // 1 を W5 に加算
        ADD      w4,w4,#1       // 1 を W4 に加算
test_loop:
        CMP      w4,#0xa        // W4 &lt; 10 なら、分岐して loop に戻る
        BLT      loop
        .end
```

新しく作成したアセンブリファイルには、GNU 構文を使用します。armclang アセンブラを使用して、GNU アセンブリ言語ソースコードをアセンブルします。通常、armclang アセンブラを起動するには以下のように入力します。

```
armclang --target=aarch64-arm-none-eabi -c -o file.o file.s
```

関連情報

GNU Binutils - 用途.
ARM 構文アセンブリコードから GNU 構文への移行.

4.2 アセンブリコードの前処理

C プリプロセッサは、C ディレクティブ (#include または #define など) を含むアセンブリ コードを解決してからアセンブルする必要があります。

デフォルトで、armclang はアセンブリ コードのソース ファイルの接尾辞を使用して、C プリプロセッサを実行するかどうかを決定します。

- .s (小文字) 接尾辞は、前処理が不要なアセンブリ コードを示します。
- .S (大文字) 接尾辞は、前処理が必要なアセンブリ コードを示します。

-x オプションを使用すると、ファイル拡張子から言語を推測する代わりに、後続のソースファイルの言語を指定することにより、デフォルトをオーバーライドすることができます。具体的にいうと、-x assembler-with-cpp は、C ディレクティブおよび armclang を含むアセンブリコードが C プリプロセッサを実行する必要があることを示しています。-x オプションは、コマンドライン上でその後に続く入力ファイルのみに適用されます。

注

.ifdef アセンブラ ディレクティブとプリプロセッサの #ifdef ディレクティブを混同しないでください。

- プリプロセッサの #ifdef ディレクティブは、プリプロセッサ マクロの有無を確認します。これらのマクロは、#define プリプロセッサ ディレクティブまたは armclang -D コマンドライン オプションを使用して定義されます。
- armclang 統合アセンブラの .ifdef ディレクティブは、コード シンボルを確認します。これらのシンボルは、ラベルまたは .set ディレクティブを使用して定義されます。

プリプロセッサは最初に実行され、ソース コードにテキストが代入されます。この段階で、#ifdef ディレクティブが処理されます。その後、.ifdef ディレクティブが処理されるときに、ソース コードがアセンブラに渡されます。

アセンブリコードのソースファイルを前処理するには、以下のいずれかを実行します。

- アセンブリコードのファイル名に .S の接尾辞があることを確認します。

  以下に例を示します。

  ```
  armclang --target=arm-arm-none-eabi -march=armv8-a -E test.S
  ```

- -x assembler-with-cpp オプションを使用して、アセンブリ ソース ファイルに前処理が必要なことを armclang に知らせます。このオプションは、小文字の拡張子 .s が付いた既存のソース ファイルがある場合に役立ちます。

  以下に例を示します。

  ```
  armclang --target=arm-arm-none-eabi -march=armv8-a -E -x assembler-with-cpp test.s
  ```

注

-E オプションは、armclang でプリプロセッサ処理のみを実行するよう指定します。

関連情報

アセンブリ ソース コードを前処理するためのコマンドライン オプション.
-E armclang オプション.
-x armclang オプション.

第 5 章
オブジェクトファイルをリンクして実行可能ファイルを生成する

armlink を使用してオブジェクトファイルをリンクし、実行可能イメージを生成する方法について説明します。

以下のセクションから構成されています。

5.1 オブジェクトファイルをリンクして実行可能ファイルを生成する

リンカは、1 つまたは複数のオブジェクトファイルの内容を必要なオブジェクトライブラリの選択した部分と結合して、実行可能イメージ、部分的にリンクされたオブジェクトファイル、または共有オブジェクトファイルを生成します。

リンカを起動するコマンドは以下のとおりです。

`armlink` *`options input-file-list`*

各項目には以下の意味があります。

`options`
はリンカコマンドラインオプションです。

`input-file-list`
は、オブジェクト、ライブラリ、または*シンボル定義*(symdefs)ファイルの空白で区切られたリストです。

例えば、オブジェクトファイル `hello_world.o` を実行可能イメージ `hello_world.axf` にリンクさせるには、以下のように入力します。

```
armlink -o hello_world.axf hello_world.o
```

第 6 章
最適化

armclang を使用してコードサイズとパフォーマンスのいずれかに最適化する方法と、デバッグ機能に対する最適化レベルの影響について説明します。

以下のセクションから構成されています。

- *6.1 コード サイズまたはパフォーマンスの最適化*(6-48 ページ).
- *6.2 リンク時最適化を使用したモジュール間の最適化*(6-49 ページ).
- *6.3 最適化によるデバッグ機能への影響*(6-53 ページ).

6.1 コード サイズまたはパフォーマンスの最適化

コンパイラと関連ツールでは、コードを最適化するために数々の技法を使用します。使用することでコードのパフォーマンスが改善される技法もあれば、コードのサイズが縮小される技法もあります。

この最適化技法は、しばしば相反する働きをすることがあります。つまり、コードのパフォーマンスが改善される技法を使用すると、コードサイズが増え、コードサイズを減らす技法を使用すると、パフォーマンスが低下する結果になることがあります。例えば、コンパイラで小規模なループを展開してパフォーマンスを高めることができますが、コードサイズが増大するというデメリットが伴います。

デフォルトでは、armclang で最適化が実行されることはありません。つまり、デフォルトの最適化レベルが -O0 になります。

次の armclang オプションを使用すると、コードパフォーマンスの最適化に役立ちます。

-O0 | -O1 | -O2 | -O3
: ソースファイルをコンパイルする際に使用する最適化レベルを指定します。-O0 は最小レベル、-O3 は最大レベルを示します。

-Ofast
: 言語標準への厳密な準拠に違反する可能性のあるその他の強力な最適化と共に、-O3 の最適化をすべて有効にします。

armclang 用の以下のオプションは、コードサイズに最適化する際に役立ちます。

-Os
: 実行時間をできるだけ長くしてイメージ サイズを小さくするように最適化します。このオプションは、パフォーマンスに対するコード サイズのバランスを調整します。

-Oz
: より小さいコード サイズ向けに最適化します。

注

armlink オプションの --lto_level を使用して最適化レベルを設定することもできます。レベルは armclang の最適化レベルに対応します。

armclang 用の以下のオプションは、コードサイズとコードパフォーマンスの両方に最適化する際に役立ちます。

-flto
: リンク時最適化を有効にします。これにより、リンカは複数のソースファイルで追加の最適化を行うことができます。

加えて、コード作成時に決定する選択肢によって、最適化が影響を受けます。以下に例を示します。

- ループ終了条件を最適化すると、コードサイズとパフォーマンスの両方が最適化されます。特に、ゼロにデクリメントするカウンタを備えたループでは、通常、インクリメントするカウンタを持つループよりも小規模で高速なコードが生成されます。
- ループの繰り返し数を減らすことによって、ループを手動で展開すると同時に、繰り返しのたびごとに行われる作業の量を増やすことによって、パフォーマンスを改善することができますが、コードサイズは大きくなります。
- オブジェクトおよびライブラリにおけるデバッグ情報を削減することによって、イメージのサイズが縮小します。
- インライン関数を使用すると、コードサイズとパフォーマンスのトレードオフをとることができます。
- 組み込み関数を使用することで、パフォーマンスを向上させることができます。

6.2 リンク時最適化を使用したモジュール間の最適化

異なるモジュールからのソースコードを同時に最適化できるため、リンク時にも最適化を行うことができます。

デフォルトでは、コンパイラは各ソース モジュールを個別に最適化し、C または C++ ソース コードをオブジェクト コードを含む ELF ファイルに変換します。シンボルの参照と再配置を解決することによって、リンカはリンク時にすべての ELF オブジェクト ファイルを実行可能ファイルに結合します。各ソース ファイルを個別にコンパイルすると、モジュール間のインライン展開のような最適化の機会が失われる可能性があります。

リンク時最適化が有効な場合、コンパイラはビットコードと呼ばれる中間形式にソース コードを変換します。リンカは、リンク時にビットコードを含むすべてのファイルを収集し、リンク時オプティマイザ (`libLTO`) に送信します。モジュールを収集するとモジュール間の依存関係に関する情報が増えるため、リンク時オプティマイザはより多くの最適化を実行できます。次に、リンク時オプティマイザは 1 つの ELF オブジェクト ファイルをリンカに返送します。最後に、リンカはすべてのオブジェクトとライブラリ コードを結合して、実行可能ファイルを作成します。

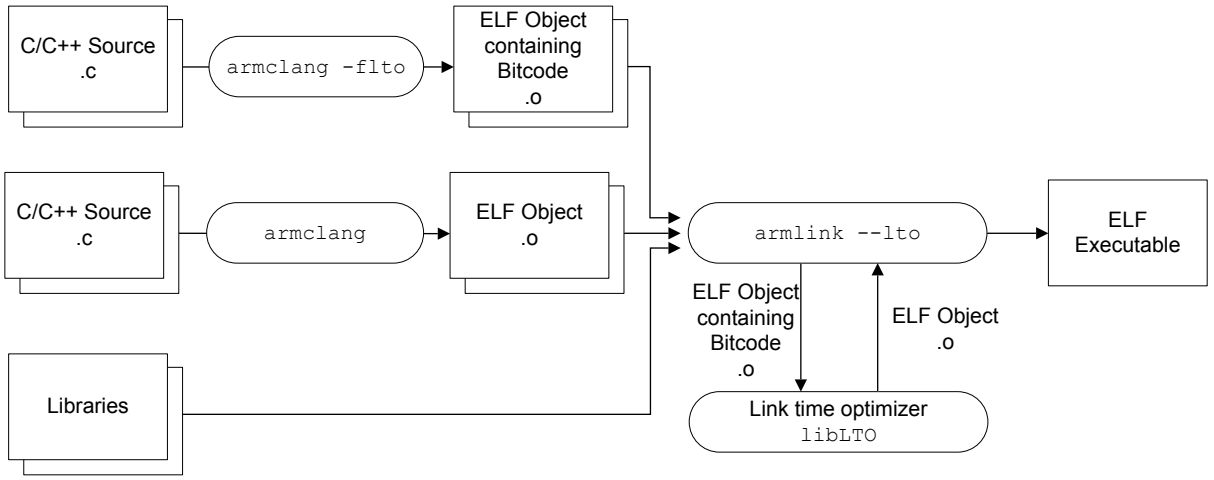


図 6-1 リンク時最適化

注

この図では、ビットコードを含む ELF オブジェクトは標準コードおよびデータを含まない ELF ファイルです。代わりに、LLVM ビットコードを保持する `.llvmbc` というセクションを含みます。

セクション `.llvmbc` は予約されています。`__attribute__((section(".llvmbc")))` などを使用して `.llvmbc` セクションを作成することはできません。

注意

リンク時最適化は、強力な最適化を実行します。これによって、大きいコード チャンクが削除されることがあります。

以下のサブセクションから構成されています。

- *6.2.1 リンク時最適化の有効化*(6-49 ページ).
- *6.2.2 リンク時最適化の制約条件*(6-50 ページ).

6.2.1 リンク時最適化の有効化

`armclang` と `armlink` の両方でリンク時最適化を有効にする必要があります。

リンク時最適化を有効にするには、以下の手順を実行します。

1. コンパイル時に、armclang オプションの -flto を使用して、リンク時最適化に適した ELF ファイルを生成します。これらの ELF ファイルの .llvmbc セクションにはビットコードが含まれます。
2. リンク時に、armlink の --lto オプションを使用して、指定したビットコードファイルのリンク時最適化を有効にします。

   注

   armclang は、-flto オプションが -c オプションなしで使用された場合、--lto オプションを自動的に armlink に渡します。

例 1:すべてのソースファイルの最適化

以下の例は、すべてのソースファイルでリンク時最適化を実行します。

```
armclang --target=arm-arm-none-eabi -march=armv8-a -flto src1.c src2.c src3.c -o output.axf
```

この例では、以下の操作が行われます。

1. armclang は、C ソース ファイル src1.c、src2.c、および src3.c を ELF ファイル src1.o、src2.o、および src3.o にコンパイルします。これらの ELF ファイルにはビットコードが含まれます。
2. armclang は、--lto オプションを使用して armlink を自動的に呼び出します。
3. armlink は、ビットコードファイル src1.o、src2.o、および src3.o をリンク時オプティマイザに渡して、1 つの最適化された ELF オブジェクトファイルを生成します。
4. armlink は、ELF オブジェクトファイルから実行可能ファイル output.axf を作成します。

例 2:ソースファイルのサブセットの最適化

以下の例は、ソースファイルのサブセットに対してリンク時最適化を実行します。

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c src1.c -o src1.o
armclang --target=arm-arm-none-eabi -march=armv8-a -c -flto src2.c -o src2.o
armclang --target=arm-arm-none-eabi -march=armv8-a -c -flto src3.c -o src3.o
armlink --lto src1.o src2.o src3.o -o output.axf
```

この例では、以下の操作が行われます。

1. armclang は、C ソースファイル src1.c を ELF オブジェクトファイル src1.o にコンパイルします。
2. armclang は、C ソース ファイル src2.c および src3.c を ELF ファイル src2.o および src3.o にコンパイルします。これらの ELF ファイルにはビットコードが含まれます。
3. armlink は、ビットコード ファイル src2.o および src3.o をリンク時オプティマイザに渡して、最適化された 1 つの ELF オブジェクト ファイルを生成します。
4. armlink は、ELF オブジェクトファイル src1.o をリンク時オプティマイザによって生成されたオブジェクトファイルと結合して、実行可能ファイル output.axf を作成します。

関連参照

6.2.2 リンク時最適化の制約条件(6-50 ページ).

関連情報

-flto.

--lto armlink オプション.

--keep armlink オプション.

6.2.2 リンク時最適化の制約条件

ARM コンパイラ 6 では、リンク時オプティマイザにいくつかの制約条件あります。今後のリリースでは制約条件が少なくなり、機能が増える予定です。リンク時最適化のユーザ インタフェースは、今後のリリースで変更される場合があります。

ビットコードライブラリなし

armlink では、コマンドライン上のビットコード オブジェクトのみがサポートされています。ライブラリからのビットコード オブジェクトはサポートされていません。armlink は、ライブラリからのロード中にビットコードを含むファイルを検出すると、エラー メッセージを出力します。

armar は armclang -flto を使用して生成された ELF ファイルをそのまま受け入れますが、現在これらのファイルには適切なシンボル テーブルがありません。そのため、生成されたアーカイブのインデックス情報は正しくなく、armlink はこのアーカイブでシンボルを検出することはできません。

部分リンク

armlink オプションの --partial は ELF ファイルに対してのみ機能します。リンカは、ビットコードを含むファイルを検出すると、エラー メッセージを出力します。

スキャッタロード

リンク時オプティマイザの出力は、1 つの ELF オブジェクト ファイルです。デフォルトで一時的なファイル名が付けられています。この ELF オブジェクト ファイルには、他の ELF オブジェクト ファイルと同様にセクションとシンボルが含まれており、これらは入力セクション セレクタによって通常どおりにマッチングが行われます。

ELF オブジェクト ファイルの出力に名前を付けるには、armlink オプションの --lto_intermediate_filename を使用します。スキャッタ ファイルで、この ELF ファイル名を参照できます。

リンク時最適化は、*(+RO) や .ANY(+RO) などの通常の入力セクション セレクタ (リンク時最適化によって生成されるセクションの選択に使用される) を使用して、スキャッタ ファイルに正確に配置する必要がないコードおよびデータにのみ実行することを推奨します。
スキャッタ ファイルでは、名前で .llvmbc セクションのビットコードをマッチングすることはできません。

——— 注 ———

スキャッタロード インタフェースは、ARM コンパイラ 6 の今後のバージョンで変更される場合があります。

実行可能ファイルとライブラリの互換性

armclang 実行可能ファイルと libLTO ライブラリは、同じ ARM コンパイラ 6 のインストールで取得する必要があります。ARM コンパイラ 6 で提供される libLTO 以外の使用はサポートされていません。

その他の制限

- ROPI/RWPI イメージのビルドにリンク時最適化を使用することはできません。
- リンク時最適化によって生成されるオブジェクト ファイルには、ターゲット アーキテクチャのデフォルトであるビルド属性が含まれています。リンク時最適化が有効な場合に armlink オプションの --cpu または --fpu を使用すると、armlink が、リンク時オプティマイザによって生成されたファイルの属性と指定された属性に互換性がないと間違って報告する可能性があります。
- リンク時最適化は armclang オプションの -ffunction-sections および -fdata-sections を認識しません。
- リンク時最適化は armclang -mexecute-only オプションを実行しません。armclang -flto オプションまたは -Omax オプションを使用すると、コンパイラは実行専用コードを生成できません。

関連参照

6.2.1 リンク時最適化の有効化(6-49 ページ).

関連情報

-flto.

--lto armlink オプション.
--keep armlink オプション.

6.3 最適化によるデバッグ機能への影響

コード最適化とデバッグ機能の間にはトレードオフがあります。

コンパイラによって正確な最適化を実行できるかどうかは、どの最適化レベルを選択するかだけでなく、パフォーマンスとコードサイズのどちらを最適化しようとしているかによっても決まります。

最も低い最適化レベル `-O0` では、生成されたコードの構造がソースコードに直接対応するため、最適なデバッグ機能が得られます。

最適化レベルを高くすると、オブジェクトコードとソースコードのマッピングがあいまいになる場合があるため、デバッグビューの質が大幅に低下します。コンパイラは、デバッグ情報で記述できない最適化を実行できることがあります。

関連情報

-O.

第 7 章
コード作成時の注意事項

プログラミング方法やテクニックを用いて、C および C++ ソースコードの移植性、効率性、および堅牢性を高める方法について説明します。

以下のセクションから構成されています。

7.1 C コードのループ終了の最適化

ループは、大半のプログラムに含まれている一般的な構文の 1 つです。ループの実行には多大な時間が費やされることが多いので、時間制限の厳しいループに注意を払うことを推奨します。

ループ終了条件は、多大なオーバーヘッドの原因となりかねないので、その作成にあたっては注意が必要です。可能な限り以下を実践して下さい。

- 単純な終了条件を使用する。
- ゼロまでカウントダウンするループを作成する。
- **unsigned int** 型のカウンタを使用する。
- ゼロと等しいかどうかをテストする。

任意のガイドラインまたはすべてのガイドラインに関して、個別にまたはそれらを組み合わせて従うことで、より優れたコードを作成できます。

次の表に、n! を計算するルーチンの実装例を 2 つ示します。これらの例は共に、ループ終了のオーバーヘッドを示しています。最初の実装では、インクリメントループを使用して n! が計算されるのに対し、2 番目の実装では、デクリメントループを使用して n! が計算されます。

表 7-1 インクリメントループとデクリメントループを表す C コード

インクリメントループ

```
int fact1(int n)
{
    int i, fact = 1;
    for (i = 1; i <= n; i++)
        fact *= i;
    return (fact);
}
```

デクリメントループ

```
int fact2(int n)
{
    unsigned int i, fact = 1;
    for (i = n; i != 0; i--)
        fact *= i;
    return (fact);
}
```

以下のテーブルは、上記の各サンプル実装に対して、
armclang -Os -S --target=arm-arm-none-eabi -march=armv8-a により生成されたマシンコードの対応する逆アセンブリを示します。

表 7-2 インクリメントループとデクリメントループを表す C 逆アセンブリコード

インクリメントループ

```
fact1:
        mov     r1, r0
        mov     r0, #1
        cmp     r1, #1
        bxlt    lr
        mov     r2, #0
.LBB0_1:
        add     r2, r2, #1
        mul     r0, r0, r2
        cmp     r1, r2
        bne     .LBB0_1
        bx      lr
```

デクリメントループ

```
fact2:
        mov     r1, r0
        mov     r0, #1
        cmp     r1, #0
        bxeq    lr
.LBB1_1:
        mul     r0, r0, r1
        subs    r1, r1, #1
        bne     .LBB1_1
        bx      lr
```

逆アセンブリを比較すると、インクリメントループ逆アセンブリの命令ペア ADD および CMP が、デクリメントループ逆アセンブリの単一の SUBS 命令で置換されていることがわかります。SUBS 命令によって、Z フラグを含むステータスフラグが更新されるため、明示的な CMP r1,r2 命令に対する要件は発生しません。

ループ内で命令を節約したことに加えて、変数 n がループの存続期間中ずっと使用可能である必要はないので、維持する必要があるレジスタの数が少なくなります。レジスタの割り当てが容易になりま

す。これは、元の終了条件に関数呼び出しが含まれる場合には、より重要になります。以下に例を示します。

```
for (...; i < get_limit(); ...);
```

ループカウンタを必要な繰り返し数に初期化して、ゼロにデクリメントするテクニックは、 **while** ステートメントと **do** ステートメントにも適用されます。

7.2 C コードのループの展開

ループは、大半のプログラムに含まれている一般的な構文の 1 つです。ループの実行には多大な時間が費やされることが多いので、時間制限の厳しいループに注意を払うことを推奨します。

小規模なループを展開すると、コードサイズが増大しますが、パフォーマンスを向上できます。ループが展開されると、ループ カウンタの更新頻度が低くなり、実行される分岐数が少なくなります。ループが数回しか繰り返されない場合は、ループを完全に展開して、ループオーバーヘッドを完全になくすことができます。コンパイラは -O3 で自動的にループを展開します。それ以外の場合は、何らかの展開をソースコードで行う必要があります。

注

ループを手動で展開すると、ループの自動再展開をはじめ、コンパイラによるループ最適化を妨げる場合があります。

ループ展開のメリットとデメリットは、次の表に示す 2 つのサンプルルーチンを使用して説明できます。どちらのルーチンも、最下位ビットを抽出してカウントし、シフトアウトさせることにより、シングルビットを効率的にテストします。

最初のルーチンでは、ビットをカウントするためにループが使用されます。2 番目のルーチンは、最初の実装を 4 回展開し、n の 4 シフト分を 1 つにまとめて最適化が適用されたものです。

頻繁に展開すると、最適化の機会が増えます。

表 7-3 未展開および展開されたビットカウントループを表す C コード

ビットカウントループ

```
int countbit1(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}
```

展開されたビットカウントループ

```
int countbit2(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        if (n & 2) bits++;
        if (n & 4) bits++;
        if (n & 8) bits++;
        n >>= 4;
    }
    return bits;
}
```

以下の表は、C コードの各実装が armclang -Os -S --target=arm-arm-none-eabi -march=armv8-a を使用してコンパイルされた、上記の実装例の各コンパイラによって生成されたマシンコードの対応する逆アセンブリを示しています。

表 7-4 未展開および展開されたビットカウントループを表す逆アセンブリコード

ビットカウントループ

```
countbit1:
        mov     r1, r0
        mov     r0, #0
        cmp     r1, #0
        bxeq    lr
        mov     r2, #0
.LBB0_1:
        and     r3, r1, #1
        cmp     r2, r1, lsr #1
        add     r0, r0, r3
        lsr     r3, r1, #1
        mov     r1, r3
        bne     .LBB0_1
        bx      lr
```

展開されたビットカウントループ

```
countbit2:
        mov     r1, r0
        mov     r0, #0
        cmp     r1, #0
        bxeq    lr
        mov     r2, #0
.LBB1_1:
        and     r3, r1, #1
        cmp     r2, r1, lsr #4
        add     r0, r0, r3
        ubfx    r3, r1, #1, #1
        add     r0, r0, r3
        ubfx    r3, r1, #2, #1
        add     r0, r0, r3
        ubfx    r3, r1, #3, #1
        add     r0, r0, r3
        lsr     r3, r1, #4
        mov     r1, r3
        bne     .LBB1_1
        bx      lr
```

展開されたバージョンのビットカウントループの方が元のバージョンよりも高速ですが、コードサイズは大きくなります。

7.3 コンパイラの最適化における volatile キーワードの影響

コンパイラが最適化してはならない変数を宣言する場合は、volatile キーワードを使用します。必要な場合に volatile キーワードを使用しなければ、コンパイラは変数へのアクセスを最適化して、意図しないコードを生成したり意図された機能を削除することがあります。

volatile の意味

変数を volatile として宣言すると、以下によって実装の外部でその変数をいつでも変更することができるとコンパイラに伝達されます。

- オペレーティング システム。
- 割り込みルーチンまたはシグナル ハンドラなどの別の実行スレッド。
- ハードウェア。

これにより、この変数が未使用または未変更という前提に基づいて、コンパイラは変数の使用を最適化しません。

volatile の使用に適した状況

実装の外部で変更される可能性がある変数に volatile キーワードを使用します。

たとえば、関数内の変数は外部プロセスによって更新される可能性があります。しかし、変数が未変更の場合、コンパイラはメモリから変数値にアクセスするのではなくレジスタに保存された古い変数値を使用することがあります。変数を volatile として宣言すると、コンパイラは、コード内で変数を参照するときは常にメモリからこの変数にアクセスするようになります。これによって、コードは必ずメモリから更新済みの変数値を使用するようになります。

別の例では、変数がスリープまたはタイマ遅延を実装するために使用されることがあります。変数が未使用の場合、変数が volatile として宣言されていなければ、コンパイラがタイマ遅延コードを削除する可能性があります。

実際の環境では、以下の場合に変数を **`volatile`** として宣言する必要があります。

- メモリマップされた周辺装置にアクセスする場合
- グローバル変数を複数のスレッド間で共有する場合
- 割り込みルーチンまたはシグナルハンドラでグローバル変数にアクセスする場合

volatile を使用しない場合に発生する可能性がある問題

volatile 変数が volatile として宣言されていない場合、コンパイラはその値を実装の外部で変更できないものと見なします。そのため、コンパイラは不要な最適化を実行する可能性があります。このような問題は、いろいろな形で明らかになります。

- ハードウェアをポーリングする際に、コードがループから抜け出せなくなる可能性があります。
- マルチスレッド コードが異常な動作を示すことがあります。
- 最適化によって、意図的なタイミングの遅延を実装するコードが削除されることがあります。

volatile キーワードを使用しない場合の無限ループの例

`volatile` キーワードの使用法を、以下の表の 2 つのルーチンの例で示します。

表 7-5 非揮発バッファループと揮発バッファループを表す C コード

バッファループの非揮発バージョン	バッファループの揮発バージョン

`int buffer_full;`
`int read_stream(void)`
`{`
`  int count = 0;`
`  while (!buffer_full)`
`  {`
`  count++;`
`  }`
`  return count;`
`}`
	`volatile int buffer_full;`
	`int read_stream(void)`
	`{`
	`  int count = 0;`
	`  while (!buffer_full)`
	`  {`
	`  count++;`
	`  }`
	`  return count;`
	`}`

いずれのルーチンも、ステータス フラグ `buffer_full` が true に設定されるまでループのカウンタがインクリメントします。`buffer_full` の状態は、プログラムフローとは非同期に変化する可能性があります。

左側の例は変数 `buffer_full` を volatile として宣言していないため、正しくありません。右側の例は変数 `buffer_full` を volatile として宣言しています。

以下の表は、コンパイラによって生成されたマシン コードが前述の各例に対してどのように逆アセンブルされるかを示したものです。各例の C コードは、
`armclang --target=arm-arm-none-eabi -march=armv8-a -Os -S` を使用してコンパイルされています。

表 7-6 非揮発バッファループと揮発バッファループを表す逆アセンブリコード

バッファループの非揮発バージョン	バッファループの揮発バージョン

`read_stream:`
`  movw  r0, :lower16:buffer_full`
`  movt  r0, :upper16:buffer_full`
`  ldr  r1, [r0]`
`  mvn  r0, #0`
`.LBB0_1:`
`  add  r0, r0, #1`
`  cmp  r1, #0`
`  beq  .LBB0_1  ; infinite loop`
`  bx  lr`
	`read_stream:`
	`  movw  r1, :lower16:buffer_full`
	`  mvn  r0, #0`
	`  movt  r1, :upper16:buffer_full`
	`.LBB1_1:`
	`  ldr  r2, [r1]  ; buffer_full`
	`  add  r0, r0, #1`
	`  cmp  r2, #0`
	`  beq  .LBB1_1`
	`  bx  lr`

volatile を使用しない例の逆アセンブリでは、ステートメント `LDR r1, [r0]` が `buffer_full` の値を `.LBB0_1` というループの外側にあるレジスタ `r1` にロードします。`buffer_full` は **`volatile`** として宣言されていないので、コンパイラはその値をプログラムの外部で変更できないものと見なします。コンパイラは、`buffer_full` の値を `r0` に既に読み込んでおり、この変数の値が変化する可能性はないので、最適化が有効にされている場合は、変数の再ロードを省略します。その結果、`.LBB0_1` という無限ループが生成されます。

volatile を使用する例の逆アセンブリでは、`buffer_full` の値がプログラムの外部で変化する可能性があるものとコンパイラによって見なされ、最適化は実行されません。その結果、`buffer_full` の値は、`.LBB1_1` というループの内部にあるレジスタ `r2` にロードされます。その結果、ループ `.LBB1_1` に生成されたアセンブリ コードは正しいコードです。

7.4 C および C++ のスタックの使用

C と C++ では、いずれもスタックが多く使用されます。

例えば、スタックは以下の項目を格納します。

- 関数の復帰アドレス
- *ARM 64 ビットアーキテクチャ向けプロシージャコール標準*(AAPCS64)によって定められた、保持する必要があるレジスタ。例えば、サブルーチンのエントリでレジスタの内容を保存する場合など。
- ローカル変数(ローカル配列を含む)、構造体、共用体、およびクラス(C++ の場合)

以下に示すような一部のスタック消費量が不明です。

- ローカルの整数または浮動小数点変数は、配置された場合(つまりレジスタに割り当てられていない場合)、スタックメモリが割り当てられます。
- 構造体は通常、スタックに割り当てられます。16 バイトの倍数にパディングされた `sizeof(struct)` と同等のスペースは、スタックに予約されます。コンパイラは、代わりにレジスタに構造体を割り当てようとします。
- 配列のサイズがコンパイル時にわかっている場合、コンパイラはスタック上のメモリを割り当てます。ここでも、16 バイトの倍数にパディングされている `sizeof(array)` に相当するスペースは、スタック上で予約されています。

  注

  可変長配列のメモリは、実行時にヒープに割り当てられます。

- 複数の最適化では、新しい一時変数を導入して、中間結果を保持することができます。最適化には、CSE の排除、ライブの範囲分割および構造体分割などがあります。コンパイラは、レジスタにこれらの一時変数を割り当てようとします。それ以外の場合は、スタックに配置されます。
- 通常、16 ビットでエンコードされた Thumb® 命令のみをサポートするプロセッサに対してコンパイルされたコードは、32 ビットでエンコードされた Thumb 命令をサポートするプロセッサに対してコンパイルされたコード、ARM コード、および A64 コードよりも多くのスタックを使用します。これは、ARM コードおよび 32 ビットでエンコードされた Thumb 命令に 14 つのレジスタがあるのに対して、16 ビットでエンコードされた Thumb 命令には、割り当てに利用できるレジスタが 8 つしかないためです。
- AAPCS64 では、タイプ、サイズ、順番に応じて、レジスタではなくスタックから関数引数を割り当てることが必要です。

スタック消費量の推定方法

スタック使用は、コードに依存するうえ、実行時にプログラムが使用するコードパスに応じて実行が変わってくるため、推測が困難です。ただし、スタックの消費量を手動で推定することは可能です。これには、以下の方法があります。

- `--callgraph` でリンクして、静的コールグラフを生成します。これにより、スタックの消費量をはじめとした、すべての関数に関する情報が示されます。

  `.debug_frame` セクションの DWARF フレーム情報を使用します。`-g` オプションを使用してコンパイルし、必要な DWARF 情報を生成します。
- `--info=stack` または `--info=summarystack` でリンクすることによって、すべてのグローバルシンボルのスタック消費量のリストを表示します。
- デバッガを使用して、スタック内の最後の使用可能な場所にウォッチポイントを設定し、そのウォッチポイントがヒットされるかどうかを調べます。`-g` オプションを使用してコンパイルし、必要な DWARF 情報を生成します。
- デバッガを使用し、
  1. 想定を大幅に上回るサイズの領域をスタックのメモリに割り当てます。
  2. `0xDEADDEAD` などの既知の値のコピーでスタックを埋めます。
  3. アプリケーションまたはその固定部分を実行します。テスト実行では、スタックスペースの可能な限り多くの部分を使用するようにして下さい。例えば、最も深いネスト構造の関数呼び出しと、静的解析で検出されたワーストケースパスを実行するとします。必要に応じて割り込みを生成し、割り込みがスタックトレースに記録されるようにします。

4. アプリケーションの実行が完了した後で、メモリのスタックスペースを検証して、上書きされた既知の値の数を調べます。スペースの使用済み部分にはガベージがあり、残りの部分には既知の値があります。
5. ガベージ値の数を数え、`sizeof(value)` だけ乗算すると、そのサイズがバイトで示されます。

計算の結果から、スタックのサイズがどの程度増大したかがバイト単位で示されます。

- 固定仮想プラットフォーム（FVP）を使用して、メモリのスタックのすぐ下のアクセスが許可されていないメモリ領域をマップファイルで定義します。アクセスが禁止された領域にスタックがオーバーフローすると、データアボートが発生し、デバッガによってトラップされる可能性があります。

スタック消費量を減らす方法

一般に、以下の方法でプログラムのスタック要件を引き下げることができます。

- 必要とする変数の数が少ない小さな関数を記述する。
- 大きなローカル構造体または配列を使用しない。
- 別のアルゴリズムを使用するなどして、再帰を避ける。
- 関数の任意のポイントで使用される変数の数を最小にする。
- C のブロック範囲を使用し、必要な場所でのみ変数を宣言して、別個の有効範囲で必要とされるメモリをオーバーラップする。

7.5 関数のパラメータ受け渡しに伴うオーバーヘッドを最小化するための方法

関数のパラメータ受け渡しに伴うオーバーヘッドは、いくつかの方法で最小限にできます。

以下に例を示します。

- AArch64 ステートでは、整数 8 と浮動小数点引数 8（合計 16 個）を効率的に渡すことができます。AArch 32 ステートでは、関数の各引数のサイズが 1 ワード以下の場合、引数が 4 個以下になることを確認します。C++ では、通常は暗黙の `this` ポインタ引数が `R0` で渡されるため、非スタティックメンバ関数に使用する引数が効率制限以下であることを確認して下さい。
- 関数で効率制限以上の引数が必要な場合は、スタックした引数の受け渡しコストをカバーできるように、その関数で相当な量の処理を行うようにして下さい。
- 関連性のある引数を 1 つの構造体にまとめ、その構造体を指すポインタを関数呼び出しで受け渡します。こうすることにより、パラメータ数を減らし、読みやすさを高めることができます。
- 32 ビットのアーキテクチャでは、`long long` パラメータでは、偶数レジスタインデクスで整列する必要のある引数ワードが 2 個使用されるため、これらのパラメータの数を最小限にします。
- 32 ビットのアーキテクチャでは、ソフトウェア浮動小数点を使用する場合は、`double` パラメータの数を最小限にします。

7.6 インライン関数

インライン関数を使用すると、コードサイズとパフォーマンスのトレードオフをとることができます。デフォルトで、コードをインライン展開すべきかどうかは、コンパイラによって自動的に決定されます。

インライン関数の詳細については、Clang のマニュアルを参照して下さい。

関連情報

言語互換性.

7.7 C コードのゼロによる整数除算エラー

ハードウェア除算命令（例えば、SDIV および UDIV）をサポートしないターゲットの場合は、ゼロによる整数除算エラーは、適切な C ライブラリヘルパ関数の __aeabi_idiv0() および __rt_raise() を試用することでトラップおよび識別できます。

__aeabi_idiv0() を使用したゼロによる整数除算エラーのトラップ

ゼロによる整数除算エラーは C ライブラリヘルパ関数の __aeabi_idiv0() を使用してトラップできるため、ゼロによる除算が発生したときにゼロなどの標準的な結果が返されるようにすることができます。

整数除算は、C ライブラリヘルパ関数 __aeabi_idiv() および __aeabi_uidiv() を使用してコードに実装されます。どちらの関数でもゼロ除算の有無がチェックされます。

ゼロによる整数除算が検出されると、__aeabi_idiv0() への分岐が作成されます。そのため、__aeabi_idiv0() 上にブレークポイントを設定するだけで、ゼロ除算をトラップできます。

ライブラリには、__aeabi_idiv0() の 2 種類の実装が用意されています。デフォルトの実行では何も行われないため、ゼロによる除算が検出されると除算関数はゼロを返します。ただし、信号処理を使用する場合は、__rt_raise(SIGFPE, DIVBYZERO) を呼び出す別の実装が選択されます。

独自のバージョンの __aeabi_idiv0() が指定された場合、除算関数がこの関数を呼び出します。__aeabi_idiv0() の関数プロトタイプは次のとおりです。

```
int __aeabi_idiv0(void);
```

__aeabi_idiv0() が値を返すと、その値は除算関数によって返される商として使用されます。

__aeabi_idiv0() の呼び出し時、リンクレジスタ LR には、アプリケーションコードの __aeabi_uidiv() 除算ルーチンの呼び出しの *後* に命令のアドレスが含まれています。

ソースコードで問題のある行を識別するには、LR によって指定されたアドレスにある C コード行をデバッガで参照します。

パラメータを検証し、__aeabi_idiv0 トラップ時の事後分析デバッグ用に保存するには、$Super$$ と $Sub$$ の各メカニズムを使用します。

1. パッチを適用していない元の関数 __aeabi_idiv0() を識別するには、__aeabi_idiv0() に接頭辞 $Super$$ を付けます。
2. 元の関数を直接呼び出すには、接頭辞 $Super$$ を付けた __aeabi_idiv0() を使用します。
3. __aeabi_idiv0() の元のバージョンの代わりに呼び出す新しい関数を識別するには、__aeabi_idiv0() に接頭辞 $Sub$$ を付けます。
4. 元の関数 __aeabi_idiv0() の前または後に処理を追加するには、__aeabi_idiv0() に接頭辞 $Sub$$ を付けます。

以下のサンプルでは、$Super$$ と $Sub$$ のメカニズムを使用して __aeabi_div0 をインターセプトする方法を示します。

```
extern void $Super$$__aeabi_idiv0(void);
/* この関数が元の関数 __aeabi_idiv0() の代わりに呼び出される */
void $Sub$$__aeabi_idiv0()
{
    // ゼロによる除算を処理するコードを挿入する
    ...
    // 元の __aeabi_idiv0 関数を呼び出す
    $Super$$__aeabi_idiv0();
}
```

__rt_raise() を使用したゼロによる整数除算エラーのトラップ

ゼロによる整数除算では、デフォルトでは 0 が返されます。したがって、ゼロによる除算をインターセプトするには、シグナルを処理するための C ライブラリヘルパ関数 __rt_raise() を再実装します。

__rt_raise() の関数プロトタイプを以下に示します。

```
void __rt_raise(int signal, int type);
```

__rt_raise() を再実装すると、__rt_raise() を呼び出す __aeabi_idiv0() の信号処理ライブラリのバージョンがライブラリによって自動的に指定され、__aeabi_idiv0() の該当のライブラリバージョンが最終イメージに含まれます。

その場合、ゼロによる除算のエラーが発生し、__aeabi_idiv0() によって __rt_raise(SIGFPE, DIVBYZERO) が呼び出されます。そのため、__rt_raise() を再実装する場合は、ゼロ除算が発生したかどうかを判断する際に (signal == SIGFPE) && (type == DIVBYZERO) をチェックする必要があります。

7.8 無限ループ

armclang は、C11 および C++11 標準に記載されているように、副作用のない無限ループを、定義されていない動作と見なします。特定の状況では、armclang は無限ループを削除または移動するため、最終的に終了するか、予期したとおりに動作しなくなります。

armclang での無限ループの記述方法

ループを無限に実行するために、ARM では無限ループを次の方法で記述することをお勧めします。

```
void infinite_loop(void) {
  while (1)
    asm volatile("");    // このラインには副作用があると見なされる
}
```

ループに副作用があるため、armclang はループを削除または移動しません。

第 8 章
コードとデータのターゲット メモリへのマッピング

ターゲット ハードウェアの正しいメモリ領域にコードおよびデータを配置する方法を説明します。

以下のセクションから構成されています。

8.1 ARM® コンパイラ のオーバーレイ サポート

メモリにいくつかのコードをロードした後に、それを別のコードに置換する必要が生じる場合があります。たとえば、システムのメモリに一度にすべてのコードをロードできないという制約がある場合がそれにあたります。

この問題を解決するには、オーバーレイ マネージャによって各オーバーレイ コードがアンロードおよびロードされるオーバーレイ領域を作成します。ARM コンパイラでは、以下をサポートしています。

- 自動オーバーレイ メカニズム。リンカがコード セクションをオーバーレイ領域にどのように割り当てるかを決定します。
- 手動オーバーレイ メカニズム。手動でコード セクションの割り当てを決定します。

関連概念

8.2 自動オーバーレイ サポート(8-70 ページ).
8.3 手動オーバーレイ サポート(8-75 ページ).

関連情報

__attribute__((section("name"))) 関数属性.
AREA.
実行領域の属性.
--emit_debug_overlay_section リンカ オプション.
--overlay_veneers リンカ オプション.

8.2 自動オーバーレイ サポート

リンカによってオーバーレイ領域にコード セクションを自動的に割り当てるには、C コードまたはアセンブリ コードを変更してオーバーレイされる部分を識別する必要があります。また、スキャッタ ファイルを設定してオーバーレイを検索する必要もあります。

自動オーバーレイ メカニズムは、以下で構成されます。

- コードをオーバーレイ済みとマークするためにオブジェクト ファイルで使用できる特別なセクション名。
- AUTO_OVERLAY 実行領域属性。これをスキャッタ ファイルで使用して、リンカが実行時にロードするオーバーレイ セクションを割り当てるメモリ領域を示します。
- コマンドライン オプション --overlay-veneers。オーバーレイ マネージャが正しいオーバーレイをアンロードおよびロードするように、リンカがオーバーレイとベニアの間でコールをリダイレクトします。
- リンカによって提供される一連のデータ テーブルおよびシンボル名。これを使用して、オーバーレイ マネージャを記述できます。
- armlink --emit_debug_overlay_section コマンドライン オプション。イメージにデバッグ情報を追加できます。このオプションを使用することで、現在有効なオーバーレイをオーバーレイに対応したデバッガで追跡できるようになります。

以下のサブセクションから構成されています。

8.2.1 オーバーレイ領域でのコード セクションの自動配置

ARM コンパイラは、コード セクションをオーバーレイ領域に自動配置することができます。

.ARM.overlay*N* 形式の名前を指定することで、オーバーレイになるコード内のセクションを識別します。ここで、 *N* は整数識別子です。その後、スキャッタ ファイルを使用して、armlink が実行時にロードするオーバーレイを割り当てるメモリ領域を示します。

各オーバーレイ領域は、スキャッタ ファイル内の AUTO_OVERLAY 属性が割り当てられた実行領域に対応します。armlink は、各オーバーレイ領域に 1 セットの整数識別子を割り当てます。また、オブジェクト ファイルで定義された .ARM.overlayN という名前の各オーバーレイ セクションに別のセットの整数識別子を割り当てます。

注

オブジェクト ファイルのオーバーレイ セクションに割り当てられる数値は、.ARM.overlay*N* セクション名で指定した数値とは一致しません。

手順

1. armlink の自動オーバーレイ メカニズムを処理する関数を宣言します。
   - C では、関数属性を使用します。以下に例を示します。

```
__attribute__((section(".ARM.overlay1"))) void foo(void) { ...}

__attribute__((section(".ARM.overlay2"))) void bar(void) { ...}
```

   - armclang の統合アセンブラ構文では、.section ディレクティブを使用します。以下に例を示します。

```
    .section    .ARM.overlay1,"ax",%progbits
    .globl    foo
    .p2align    2
    .type    foo,%function
foo:                                    @ @foo
```

```
    ...
    .fnend

    .section .ARM.overlay2,"ax",%progbits
    .globl    bar
    .p2align    2
    .type    bar,%function
bar:                                    @ @bar
    ...
    .fnend
```

- ARM 構文のアセンブリでは、AREA ディレクティブを使用します。以下に例を示します。

```
    AREA |.ARM.overlay1|,CODE
foo PROC
    ...
    ENDP

    AREA |.ARM.overlay2|,CODE
bar PROC
    ...
    ENDP
```

注

コード セクションをオーバーレイするかどうかを選択できます。データ セクションをオーバーレイすることはできません。

2. スキャッタ ファイル内でコード セクションをロードする場所を指定します。1 つまたは複数の実行領域で AUTO_OVERLAY を使用します。

   実行領域にセクション セレクタを含めることはできません。例えば、

```
OVERLAY_LOAD_REGION 0x10000000
{
    OVERLAY_EXECUTE_REGION_A 0x20000000 AUTO_OVERLAY 0x10000 { }
    OVERLAY_EXECUTE_REGION_B 0x20010000 AUTO_OVERLAY 0x10000 { }
}
```

   この例では、armlink によって、アドレス 0x10000000 で始まるすべてのオーバーレイ データをロードするプログラム ヘッダ テーブル エントリが生成されます。また、各オーバーレイは、アドレス 0x20000000 または 0x20010000 にコピーされた場合に正しく実行されるように再配置されます。armlink によって、これらのアドレスのいずれかが各オーバーレイに選択されます。

3. リンク時に、--overlay_veneers コマンドライン オプションを指定します。このオプションを使用すると、armlink は、オーバーレイ マネージャのエントリ ポイントを介して迂回するように、2 つのオーバーレイの間または非オーバーレイ コードとオーバーレイの間で関数呼び出しを調整します。

   オーバーレイに対応したデバッガを使用してアクティブなオーバーレイを追跡するには、armlink --emit_debug_overlay_section コマンドライン オプションを指定します。

関連情報

__attribute__((section("name"))) 関数属性.
AREA.
実行領域の属性.
--emit_debug_overlay_section リンカ オプション.
--overlay_veneers リンカ オプション.

8.2.2 オーバーレイ ベニア

armlink は、2 つのオーバーレイの間または非オーバーレイ コードとオーバーレイの間で、関数呼び出しごとにオーバーレイ ベニアを生成できます。

関数の呼び出しや戻りによって、2 つのオーバーレイの間または非オーバーレイ コードとオーバーレイの間で制御を転送できます。ターゲット関数が意図する実行アドレスに存在しない場合、ターゲット オーバーレイをロードする必要があります。

ターゲット オーバーレイが存在するかどうか検出するために、armlink は、オーバーレイ マネージャのエントリ ポイント __ARM_overlay_entry を介して迂回するように、このようなすべての関数呼び出しを調整できます。この機能を有効にするには、armlink コマンドライン オプション --overlay_veneers を使用します。このオプションによって、ターゲット関数の代わりに呼び出し命令 (通常は BL 命令) でベニアが示されるように、影響を受ける関数呼び出しごとにベニアが生成されます。ベニアは、いくつかのレジスタをスタックに保存し、そこに含まれるターゲット関数とオーバーレイに関するいくつかの情報をロードして、オーバレイ マネージャのエントリ ポイントに制御を転送します。その後、オーバーレイ マネージャは以下を実行する必要があります。

- 正しいオーバーレイがロードされ、ターゲット関数に制御が転送されていることを確認します。
- 元の BL 命令によって、スタックとレジスタが以前置かれていた状態にそれらを復元します。
- 関数呼び出しがオーバーレイ内で生成された場合、呼び出し先関数から復帰することによって復帰元のオーバーレイがリロードされることを確認します。

関連情報

--overlay_veneers リンカ オプション.

8.2.3 オーバーレイ データ テーブル

armlink は、読み出し専用データを示すさまざまなシンボル (ほぼ配列) を提供します。このデータは、イメージ内のオーバーレイおよびオーバーレイ領域のコレクションを説明します。

シンボルは、以下のとおりです。

Region$$Table$$AutoOverlay
: このシンボルは、オーバーレイ領域 1 つにつき 2 つの 32 ビット ポインタを含む配列を示します。各領域では、2 つのポインタがオーバーレイ領域の開始アドレスと終了アドレスを示します。開始アドレスは、領域の先頭のバイトです。終了アドレスは、領域の末尾の次にあるバイトです。オーバーレイ マネージャは、このシンボルを使用して、呼び出し元関数の復帰アドレスがオーバーレイ領域に含まれる時期を特定します。この場合、リターン thunk が必要になる可能性があります。

  注

  領域は必ず開始アドレスの昇順でソートされます。

Region$$Count$$AutoOverlay
: このシンボルは、オーバーレイ領域の合計数を指定する 1 つの 16 ビット整数 (符号なし短整数型) を示します。つまり、配列 Region$$Table$$AutoOverlay および CurrLoad$$Table$$AutoOverlay のエントリ数です。

Overlay$$Map$$AutoOverlay
: このシンボルは、オーバーレイ領域 1 つにつき 1 つの 16 ビット整数 (符号なし短整数型) を含む配列を示します。このテーブルには、オーバーレイごとに、オーバーレイが正しく実行されるためにロードされる必要があるオーバーレイ領域が示されます。

Size$$Table$$AutoOverlay
: このシンボルは、オーバーレイ 1 つにつき 1 つの 32 ビットワードを含む配列を示します。このテーブルには、オーバーレイごとにオーバーレイの正確なデータ サイズが示されます。通常、オーバーレイは領域を完全に埋めてしまうわけではないため、このサイズはオーバーレイ領域に含まれるサイズよりも小さい場合があります。

読み出し専用テーブルに加えて、armlink は読み出し-書き込みメモリも提供します。

CurrLoad$$Table$$AutoOverlay
: このシンボルは、オーバーレイ領域ごとに 16 ビット整数 (符号なし短整数型) を含む配列を示します。この配列は、オーバーレイ マネージャが各領域で現在ロードされているオーバーレイの識別子を保存することを意図したものです。オーバーレイ マネージャは、すでにロードされているオーバーレイのリロードを回避できます。

これらすべてのデータ テーブルはオプションです。コードが特定のテーブルを参照しない場合は、イメージから省略されます。

関連概念

8.2 自動オーバーレイ サポート(8-70 ページ).

8.2.4 自動オーバーレイ サポートの制限事項

自動オーバーレイ機能を使用する場合、いくつかの制限事項があります。

以下の制限事項が適用されます。

- 自動オーバーレイ機能は C++ をサポートしていません。
- 複数の関数を `.ARM.overlay`*N* という同じ名前のセクションに割り当てると、`armlink` はそれらを異なるオーバーレイとして扱います。`armlink` は各オーバーレイに異なる整数 ID を割り当てます。
- `armlink --any_placement` コマンドライン オプションは、自動オーバーレイ セクションに対しては現在無視されます。
- オーバーレイ システムは、オーバーレイ間および非オーバーレイ コードとオーバーレイ コードの間の直接的な呼び出し用にベニアを自動生成します。また、関数ポインタを介したオーバーレイ内の関数への間接的な呼び出しが機能するように自動的に調整します。ただし、間接的な関数呼び出しが正しく設定されないことがあります。具体的には、ユーザが非オーバーレイ関数へのポインタを取り、その関数を呼び出すオーバーレイにこのポインタを渡す場合です。その場合、`armlink` はオーバーレイ ベニアに呼び出しを挿入できません。そのため、オーバーレイ マネージャは、復帰時に呼び出し元関数の代わりにオーバーレイをリロードするように調整することはできません。

  単純なケースでは、機能します。しかし、非オーバーレイ関数が、呼び出し元関数のオーバーレイと競合する 2 つ目のオーバーレイから呼び出しを実行すると、ランタイム エラーが発生します。例えば、

```
__attribute__((section(".ARM.overlay1"))) void innermost(void)
{
    // 操作を実行する
}

void non_overlaid(void)
{
    innermost();
}

typedef void (*function_pointer)(void);

__attribute__((section(“.ARM.overlay2”))) void call_via_ptr(function_pointer f)
{
    f();
}

int main(void)
{
    // オーバーレイ関数 call_via_ptr() を呼び出して
    // non_overlaid() にポインタを渡す。そうすると non_overlaid() は
    // 別のオーバーレイで関数 innermost() を呼び出す。call_via_ptr() と innermost() が
    // リンカによって同じオーバーレイ領域に割り当てられている場合、
    // non_overlaid() から制御が戻るまで
    // call_via_ptr をリロードすることはできない。

    call_via_ptr(non_overlaid);
}
```

関連概念

8.2 自動オーバーレイ サポート(8-70 ページ).

8.2.5 自動配置されたオーバーレイ向けのオーバーレイ マネージャの記述

オーバーレイのロードおよびアンロードを処理するようにオーバーレイ マネージャを記述するには、オーバーレイ マネージャのエントリ ポイントを実装する必要があります。

オーバーレイ マネージャのエントリ ポイント `__ARM_overlay_entry` は、リンカが生成したベニアがジャンプすると予測される場所です。リンカは、ロードするオーバーレイおよびオーバーレイ領域をオーバーレイ マネージャが検索できるように、いくつかのデータ テーブルも提供します。

エントリ ポイントは、以下のようにリンカのオーバーレイ ベニアによって呼び出されます。

- r0 には、ターゲット関数を含むオーバーレイの整数識別子が含まれています。
- r1 には、ターゲット関数の実行アドレスが含まれています。つまり、オーバーレイがロードされるときに関数が表示されるアドレスです。
- オーバーレイ ベニアは 6 つの 32 ビットワードをスタックにプッシュします。これらのワードは、呼び出し元関数の r0、r1、r2、r3、r12、および lr レジスタの値で構成されます。呼出し命令が `BL` の場合、lr の値は `BL` の前の値ではなく、`BL` 命令によって lr に記述された値です。

オーバーレイ マネージャは、以下を実行する必要があります。

1. ターゲット オーバーレイのロード。
2. スタックからの 6 つすべてのレジスタの復元。
3. r1 で渡されるターゲット関数のアドレスへの制御の転送。

オーバーレイ マネージャは、リターン thunk ルーチンのポイントを示すために lr の呼び出し元関数に渡す値を変更する必要がある場合もあります。このルーチンは、呼び出し元関数のオーバーレイをリロードしてから、制御を呼び出し元関数の lr の元の値に戻します。

リターン thunk で使用する lr の元の値を保存できる適当な場所はありません。たとえば、スタックに値を含めることができる場所はありません。そのため、オーバーレイ マネージャは、独自のスタックによって組織化されたデータ構造を維持する必要があります。データ構造には、オーバーレイ マネージャが関数呼び出し時にリターン thunk を代わりに使用するたびに保存される lr 値とそれに対応するオーバーレイ ID が含まれおり、主要なコール スタックとの同期が維持されます。

注

この余分な並列スタックを維持する必要があるため、オーバーレイ マネージャの並列スタックの整合性を維持するようにカスタマイズしなければ、協調的またはプリエンプティブなスレッド切り替え、コルーチン、setjmp/longjmp などのスタック操作を使用することはできません。

`armlink --info=auto_overlays` オプションによって、リンカは、出力するイメージにオーバーレイのテキスト サマリーを書き出します。サマリーには、整数 ID、開始アドレス、および各オーバーレイのサイズが含まれます。この情報を使用して、イメージ (`fromelf --bin` 出力など) からオーバーレイを抽出できます。その後、それらを別のペリフェラル ストレージ システムに配置できます。そのため、オーバーレイ マネージャでそれらの 1 つをロードする必要がある場合に、データ チャンクに割り当てられたオーバーレイ ID がわかります。

関連概念

8.2 自動オーバーレイ サポート(8-70 ページ).

関連情報

__attribute__((section("name"))) 関数属性.
AREA.
実行領域の属性.
--emit_debug_overlay_section リンカ オプション.
--overlay_veneers リンカ オプション.

8.3 手動オーバーレイ サポート

コード セクションをオーバーレイ領域に手動で割り当てるには、スキャッタ ファイルを設定してオーバーレイを検索する必要があります。

手動オーバーレイ メカニズムは、以下で構成されます。

- ロード領域および実行領域の OVERLAY 属性。この属性をスキャッタ ファイルで使用して、リンカが実行時にロードするオーバーレイ セクションを割り当てるメモリ領域を示します。
- 以下の armlink コマンドライン オプション。イメージにデバッグ情報を追加できます。
  - --emit_debug_overlay_relocs.
  - --emit_debug_overlay_section.

  この追加のデバッグ情報によって、有効なオーバーレイをオーバーレイに対応したデバッガで追跡できるようになります。

以下のサブセクションから構成されています。

- *8.3.1 オーバーレイ領域でのコード セクションの手動配置*(8-75 ページ).

8.3.1 オーバーレイ領域でのコード セクションの手動配置

オーバーレイを使用して、複数の実行領域を同じアドレスに配置できます。

OVERLAY 属性を使用すると、複数の実行領域を同じアドレスに配置できます。実行領域を一度に 1 つだけインスタンス化するには、オーバーレイマネージャが必要です。ARM コンパイラ にオーバーレイ マネージャは付属していません。

以下の例では、RAM 内のスタティックセクションと、その後に続く一連のオーバーレイの定義を示しています。この例では、一度にこれらのセクションの 1 つのみをインスタンス化しています。

```
EMB_APP 0x8000
{
    …
    STATIC_RAM 0x0                  ; RW と ZI のコード/データのほとんどを含む
    {
            * (+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY    ; オーバーレイの開始アドレス…
    {
            module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
            module2.o (+RW,+ZI)
    }
    …                           ; 残りのスキャッタロード記述
}
```

起動時に C ライブラリは OVERLAY とマークされている領域を初期化しません。オーバーレイ領域が使用するメモリの内容は、オーバーレイマネージャが管理します。また、領域に初期化されたデータが含まれている場合は、NOCOMPRESS 属性を使用して RW データが圧縮されないようにします。

リンカ定義シンボルを使用すると、コードとデータのコピーに必要なアドレスを取得できます。

OVERLAY 属性は、別の領域とは異なるアドレスを持つ単一の領域で使用できます。したがって、C ライブラリ スタートアップ コードによる特定の領域の初期化を避ける方法として、オーバーレイ領域を使用できます。他のオーバーレイ領域と同様に、これらの領域はコード内で手動で初期化する必要があります。

オーバーレイ領域は相対ベースを持つことができます。*+offset* ベースアドレスを持つオーバーレイ領域の動作は、その前の領域と *+offset* の値に依存します。*+offset* の値が同じ場合、リンカは同じベース アドレスに連続する *+offset* 領域を配置します。

+offset の実行領域 ER が、オーバーレイ実行領域の連続する重複ブロックに続けて存在する場合、ER のベースアドレスは次のようになります。

`limit address of the overlapping block of overlay execution regions + offset`

以下の表には、`+offset` を OVERLAY 属性と共に使用した場合の影響を示しています。REGION1 は、スキャッタファイル内の REGION2 の直前に配置されます。

表 8-1 オーバーレイでの相対オフセットの使用

REGION1 を OVERLAY と共に設定する	+offset	REGION2 ベースアドレス
いいえ	<offset>	REGION1 のリミット + <offset>
はい	+0	REGION1 ベースアドレス
はい	<non-zero offset>	REGION1 のリミット + <non-zero offset>

以下の例では、相対オフセットをオーバーレイと共に使用した場合の実行領域のアドレスに対する影響を示しています。

```
EMB_APP 0x8000
{
    CODE 0x8000
    {
        *(+RO)
    }
    # REGION1 Base = CODE limit
    REGION1 +0 OVERLAY
    {
        module1.o(*)
    }
    # REGION2 Base = REGION1 Base
    REGION2 +0 OVERLAY
    {
        module2.o(*)
    }
    # REGION3 Base = REGION2 Base = REGION1 Base
    REGION3 +0 OVERLAY
    {
        module3.o(*)
    }
    # REGION4 Base = REGION3 Limit + 4
    Region4 +4 OVERLAY
    {
        module4.o(*)
    }
}
```

オーバーレイでない領域の長さが不明な場合は、ゼロ相対オフセットを使用してオーバーレイの開始アドレスを指定することで、オーバーレイをスタティック セクションの直後に配置できます。

関連情報

ロード領域の記述.
ロード領域の属性.
ロード領域のアドレス属性の継承規則.
ロード領域に相対アドレス +offset を使用する際の注意事項.
実行領域に相対アドレス +offset を使用する際の注意事項.
--emit_debug_overlay_relocs リンカ オプション.
--emit_debug_overlay_section リンカ オプション.
『ABI for the ARM Architecture: Support for Debugging Overlaid Programs』.

関連概念

8.1 ARM® コンパイラ のオーバーレイ サポート(8-69 ページ).

関連情報

実行領域の属性.

--emit_debug_overlay_relocs リンカ オプション.
--emit_debug_overlay_section リンカ オプション.

第 9 章
ARMv8-M セキュリティ拡張機能を使用したセキュア イメージおよび非セキュア イメージのビルド

ARMv8-M セキュリティ拡張機能を使用してセキュア イメージをビルドする方法と、非セキュア イメージによってセキュア イメージを呼び出すことができるようにする方法について説明します。

以下のセクションから構成されています。

9.1 セキュア イメージおよび非セキュア イメージのビルドの概要

ARM コンパイラ 6 のツールによって、ARMv8-M セキュリティ拡張機能のセキュア状態で実行されるイメージをビルドできます。それらのイメージでセキュア イメージを呼び出すために非セキュア イメージの開発者が必要とするインポート ライブラリ パッケージも作成できます。

注

ARMv8-M セキュリティ拡張機能は、*読み出し専用の位置非依存* (ROPI) イメージと *読み書き位置非依存* (RWPI) イメージのビルド時はサポートされません。

セキュア状態で実行されるイメージをビルドするには、コードに `<arm_cmse.h>` ヘッダを含めて、`armclang -mcmse` コマンドライン オプションを使用してコンパイルする必要があります。この方法でコンパイルすると、以下の機能を使用できるようになります。

- テスト ターゲット `TT` 命令。
- `TT` 命令の組み込み関数。
- 非セキュア関数ポインタ組み込み関数。
- `__attribute__((cmse_nonsecure_call))` 関数属性と `__attribute__((cmse_nonsecure_entry))` 関数属性。

スタートアップ時に、セキュア コードによって *セキュリティ属性ユニット* (SAU) が設定され、非セキュア スタートアップ コードが呼び出される必要があります。

セキュア コードおよび非セキュア コードをコンパイルする際の重要な要件

セキュア コードと非セキュア コードをコンパイルするときは、以下の点に注意してください。

- C または C++ でセキュアおよび非セキュア コードをコンパイルできますが、2 つの間の境界には C 関数呼び出しリンケージが必要です。
- セキュリティ境界をまたいで、クラスや参照などの C++ オブジェクトを渡すことはできません。
- セキュリティ境界をまたいで C++ 例外をスローすることはできません。
- `__ARM_FEATURE_CMSE` 定義済みマクロの値は、サポートされている ARMv8-M セキュリティ拡張機能を示します。
- ターゲットの最大限の機能を使用してセキュア コードをコンパイルします。たとえば、FPU を使用せずにコンパイルすると、セキュア関数は `__attribute__((cmse_nonsecure_entry))` として宣言された関数から復帰するときに浮動小数点レジスタをクリアしません。そのため、関数によって機密データが漏洩する可能性があります。
- パディングと半精度浮動小数点によって発生した未定義のビットを使用した構造体は、現在、セキュア関数の引数および戻り値としてサポートされていません。このような構造体を使用すると、機密情報が漏洩する可能性があります。ポインタによって渡されるのに十分な大きさの構造体もサポートされておらず、エラーが発生します。
- 以下のケースは `-mcmse` を使用してコンパイルするときはサポートされず、エラーが発生します。
  - 可変個引数エントリ関数。
  - レジスタに適さない引数が含まれたエントリ関数 (引数が多いか、引数の値が大きいため)。
  - レジスタに適さない引数が含まれた非セキュア関数呼び出し (引数が多いか、引数の値が大きいため)。

非セキュア イメージがベニアを使用してセキュア イメージを呼び出す方法

非セキュア イメージからセキュア イメージを呼び出すには、非セキュア状態からセキュア状態に移行する必要があります。移行は、セキュア ゲートウェイ ベニアを通して開始されます。セキュア ゲートウェイ ベニアは、セキュア コードの残りの部分からアドレスを分離させます。

セキュア イメージのエントリ ポイント *`entryname`* は、以下で識別されます。

```
__acle_se_entryname:
entryname:
```

呼び出しシーケンスは以下のとおりです。

1. 非セキュア イメージは分岐の BL 命令を使用して、セキュア イメージの必要なエントリ関数のセキュア ゲートウェイ ベニアを呼び出します。

```
bl    entryname
```

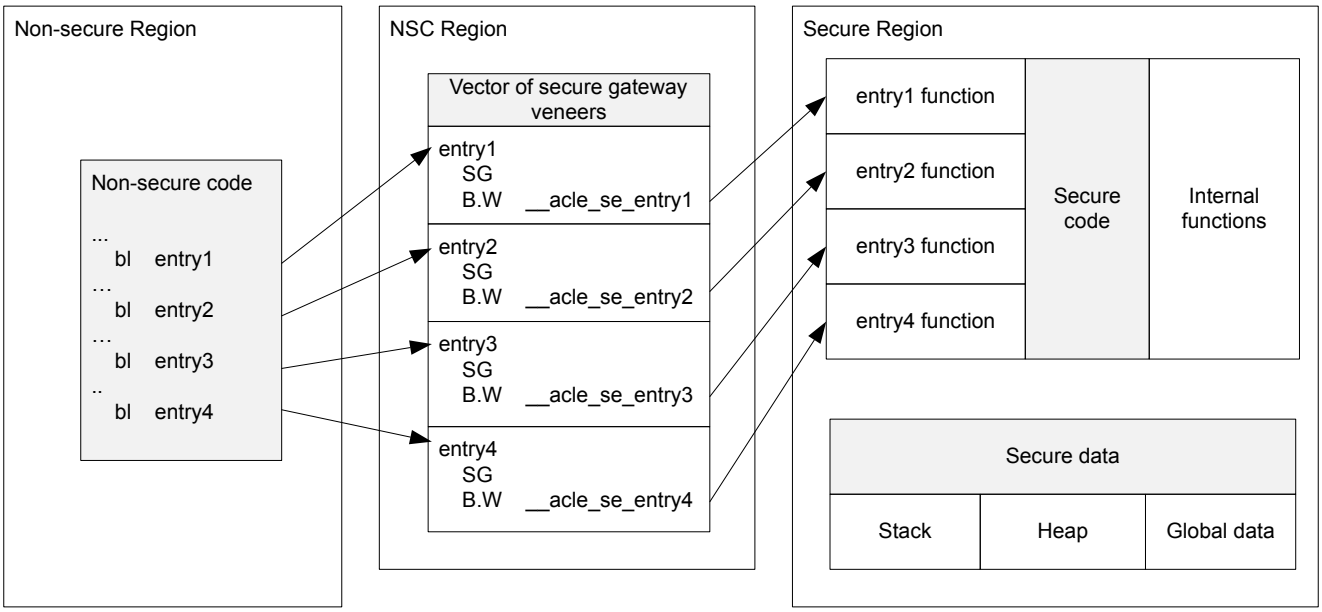
2. 非セキュア ゲートウェイ ベニアは、SG 命令と、B 命令を使用するセキュア イメージのエントリ関数への呼び出しで構成されます。

```
entryname    SG
    B.W       __acle_se_entryname
```

3. セキュア イメージは、BXNS 命令を使用してエントリ関数から復帰します。

```
bxns  lr
```

以下に呼び出しシーケンスを図で示します。ただし、わかりやすくするためにエントリ関数からの復帰は表示していません。



インポート ライブラリ パッケージ

インポート ライブラリ パッケージは、セキュア イメージで使用可能なエントリ関数を識別します。インポート ライブラリ パッケージの内容は、以下のとおりです。

- インタフェース ヘッダ ファイル (myinterface.h など)。このファイルは、テキスト エディタを使用して手動で作成します。
- インポート ライブラリ (importlib.o. armlink など)。このライブラリはセキュア イメージのリンク ステージで作成されます。

  注

  以下の理由から、コンパイル ステージとリンク ステージを分ける必要があります。
  — セキュア イメージをビルドする際にインポート ライブラリを作成するため。
  — 非セキュア イメージをビルドする際にインポート ライブラリを使用するため。

関連タスク

9.2 ARMv8-M セキュリティ拡張機能を使用したセキュア イメージのビルド(9-82 ページ).
9.4 以前生成したインポート ライブラリを使用したセキュア イメージのビルド(9-88 ページ).
9.3 セキュア イメージを呼び出すことができる非セキュア イメージのビルド(9-86 ページ).

関連情報

ホワイト ペーパー: ARMv8-M アーキテクチャの技術概要.
-mcmse.
__attribute__((cmse_nonsecure_call)) 関数属性.
__attribute__((cmse_nonsecure_entry)) 関数属性.
定義済みマクロ.
TT 命令の組み込み関数.
非セキュア関数ポインタ組み込み関数.
B 命令.
BL 命令.
BXNS 命令.
SG 命令.
TT、TTT、TTA、TTAT 命令.
セキュア イメージの CMSE ベニア セクションの配置.

9.2 ARMv8-M セキュリティ拡張機能を使用したセキュア イメージのビルド

セキュア イメージをビルドする場合は、セキュア イメージへのエントリ ポイントを指定するインポート ライブラリも生成する必要があります。インポート ライブラリは、セキュア イメージを呼び出す必要がある非セキュア イメージをビルドする場合に使用されます。

前提条件

以下の手順は、完全な例ではありません。また、コードによって *セキュリティ属性ユニット* (SAU) が設定され、非セキュア スタートアップ コードが呼び出されるものとします。

手順

1. インタフェース ヘッダ ファイル `myinterface_v1.h` を作成して、非セキュア コードで使用するための C リンケージを指定します。

```
#ifdef __cplusplus
extern "C" {
#endif

int entry1(int x);
int entry2(int x);

#ifdef __cplusplus
}
#endif
```

2. セキュア コード用の C プログラム `secure.c` に以下を含めます。

```
#include <arm_cmse.h>
#include "myinterface_v1.h"

int func1(int x) { return x; }
int __attribute__((cmse_nonsecure_entry)) entry1(int x) { return func1(x);  }
int __attribute__((cmse_nonsecure_entry)) entry2(int x) { return entry1(x); }

int main(void) { return 0; }
```

2 つのエントリ関数の実装に加えて、このコードはセキュア コードによってのみ呼び出される関数 `func1()` を定義します。

注

セキュア コードを C++ としてコンパイルしている場合、`__attribute__((cmse_nonsecure_entry))` として宣言された関数に `extern "C"` を追加する必要があります。

3. `armclang -mcmse` コマンドライン オプションを使用して、オブジェクト ファイルを作成します。

```
$ armclang -c --target arm-arm-none-eabi -march=armv8-m.main -mcmse secure.c -o secure.o
```

4. `armclang` によって生成されるマシン コードの逆アセンブリを確認するには、以下のコマンドを入力します。

```
$ armclang -c --target arm-arm-none-eabi -march=armv8-m.main -mcmse -S secure.c
```

逆アセンブリはファイル `secure.s` に保存されます。以下に例を示します。

```
    .text
    ...
    .code 16
    .thumb_func
    ...
func1:
    .fnstart
    ...
    bx lr
    ...
__acle_se_entry1:
entry1:
    .fnstart
@ BB#0:
    .save    {r7, lr}
```

```
    push    {r7, lr}
    ...
    bl func1
    ...
    pop.w {r7, lr}
    ...
    bxns lr
    ...
__acle_se_entry2:
entry2:
    .fnstart
@ BB#0:
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl entry1
    ...
    pop.w {r7, lr}
    bxns lr
    ...
main:
    .fnstart
@ BB#0:
    ...
    movs r0, #0
    ...
    bx lr
    ...
```

エントリ関数はセキュア ゲートウェイ (SG) 命令で始まっていません。2 つのシンボル `__acle_se_`*entry_name* および *entry_name* は、リンカへのエントリ関数の開始を示しています。

5. `Veneer$$CMSE` セレクタを含むスキャッタ ファイルを作成して、*非セキュア呼び出し可能* (NSC) メモリ領域にエントリ関数のベニアを配置します。

```
LOAD_REGION 0x0 0x3000
{
    EXEC_R 0x0
    {
        *(+RO,+RW,+ZI)
    }
    EXEC_NSCR 0x4000 0x1000
    {
        *(Veneer$$CMSE)
    }
    ARM_LIB_STACK 0x700000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
...
```

6. `armlink --import-cmse-lib-out` コマンドライン オプションとスキャッタ ファイルを使用してオブジェクト ファイルをリンクし、セキュア イメージを作成します。

```
$ armlink secure.o -o secure.axf --cpu 8-M.Main --import-cmse-lib-out importlib_v1.o --
scatter secure.scf
```

最終的なイメージに加えて、この例のリンクからは非セキュア イメージのビルド用のインポート ライブラリ `importlib_v1.o` も生成されます。ベニアを含むセクションがアドレス `0x4000` に配置されると仮定した場合、インポート ライブラリは以下のエントリを含むシンボル テーブルのみを含む再配置可能なファイルで構成されます。

シンボル型	Name	アドレス
STB_GLOBAL, SHN_ABS, STT_FUNC	entry1	0x4001
STB_GLOBAL, SHN_ABS, STT_FUNC	entry2	0x4009

このアセンブリ コードに対応する再配置可能なファイルをイメージにリンクすると、リンカによってエントリ ベニアのみを含むセクションにベニアが作成されます。

注

セキュア イメージを前回ビルドしたときのインポート ライブラリがある場合、セキュア イメージの新しいバージョンを作成する際に出力インポート ライブラリのアドレスを変更しないようにすることができます。アドレスを変更しないようにするには、`--import-cmse-lib-in` コマンドライン オプションを `--import-cmse-lib-out` オプションと一緒に指定します。ただし、入力ライブラリと出力ライブラリは必ず違う名前にしてください。

7. リンカによって生成されるエントリ ベニアを確認するには、以下のコマンドを入力します。

```
$ fromelf --text -s -c secure.axf
```

この例では、以下のエントリ ベニアが EXEC_NSCR *execute-only* (XO) 領域に作成されます。

```
...

** Section #3 'EXEC_NSCR' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....SG        ; [0x3e08]
        0x00004004:    f7fcb85e    ..^.B        __acle_se_entry1 ; 0xc4
    entry2
        0x00004008:    e97fe97f    ....SG        ; [0x3e10]
        0x0000400c:    f7fcb86c    ..l.B        __acle_se_entry2 ; 0xe8

...
```

ベニアを含むセクションは 32 バイト境界で整列しており、32 バイト境界にパディングが挿入されます。

スキャッタ ファイルを使用しない場合、エントリ ベニアは最初の実行領域として ER_XO セクションに配置されます。以下に例を示します。

```
...

** Section #1 'ER_XO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00008000

    $t
    entry1
        0x00008000:    e97fe97f    ....SG        ; [0x7e08]
        0x00008004:    f000b85a    ..Z.B.W      __acle_se_entry1 ; 0x80bc
    entry2
        0x00008008:    e97fe97f    ....SG        ; [0x7e10]
        0x0000800c:    f000b868    ..h.B.W      __acle_se_entry2 ; 0x80e0

...
```

後発条件

セキュア イメージをビルドした後、以下の手順を実行します。

1. セキュア イメージをデバイスに事前にロードします。
2. 事前にロードされたイメージを含むデバイスを、インポート ライブラリ パッケージと一緒に、このデバイスの非セキュア コードを開発する担当者に届けます。インポート ライブラリ パッケージの内容は、以下のとおりです。
   - インタフェース ヘッダ ファイル `myinterface_v1.h`。
   - インポート ライブラリ `importlib_v1.o`。

関連タスク

9.4 以前生成したインポート ライブラリを使用したセキュア イメージのビルド(9-88 ページ).
9.3 セキュア イメージを呼び出すことができる非セキュア イメージのビルド(9-86 ページ).

関連情報

ホワイト ペーパー: ARMv8-M アーキテクチャの技術概要.
-c armclang オプション.
-march armclang オプション.
-mcmse armclang オプション.
-S armclang オプション.
--target armclang オプション.
__attribute__((cmse_nonsecure_entry)) 関数属性.
SG 命令.
--cpu armlink オプション.
--import_cmse_lib_in armlink オプション.
--import_cmse_lib_out armlink オプション.
--scatter armlink オプション.
--text fromelf オプション.

9.3 セキュア イメージを呼び出すことができる非セキュア イメージのビルド

セキュア イメージを呼び出すための非セキュア イメージをビルドする場合、非セキュア コードを C で記述する必要があります。また、そのセキュア イメージ用に作成されたインポート ライブラリ パッケージも取得する必要があります。

前提条件

以下の手順では、*9.2 ARMv8-M セキュリティ拡張機能を使用したセキュア イメージのビルド* (9-82 ページ) で作成されるインポート ライブラリ パッケージを所有していることが前提となっています。このパッケージには、非セキュア コードを C または C++ としてコンパイルできる C リンケージが用意されています。

インポート ライブラリ パッケージによって、セキュア イメージのエントリ ポイントが識別されます。

手順

1. 非セキュア コード `nonsecure.c` の C プログラムにインタフェース ヘッダ ファイルを含めて、必要に応じてエントリ関数を使用します。以下に例を示します。

```
#include <stdio.h>
#include "myinterface_v1.h"

int main(void) {
    int val1, val2, x;

    val1 = entry1(x);
    val2 = entry2(x);

    if (val1 == val2) {
        printf("val2 is equal to val1\n");
    } else {
        printf("val2 is different from val1\n");
    }

    return 0;
}
```

2. オブジェクトファイル、`nonsecure.o` を作成します。

```
$ armclang -c --target arm-arm-none-eabi -march=armv8-m.main nonsecure.c -o nonsecure.o
```

3. 非セキュア イメージ用のスキャッタ ファイルを作成します。ただし、*非セキュア呼び出し可能* (NSC) メモリ領域は使用しません。以下に例を示します。

```
LOAD_REGION 0x8000 0x3000
{
    ER 0x8000
    {
        *(+RO,+RW,+ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
...
```

4. インポート ライブラリ `importlib_v1.o` とスキャッタ ファイルを使用してオブジェクト ファイルをリンクし、非セキュア イメージを作成します。

```
$ armlink nonsecure.o importlib_v1.o -o nonsecure.axf --cpu=8-M.Main --scatter
nonsecure.scat
```

関連タスク

9.2 ARMv8-M セキュリティ拡張機能を使用したセキュア イメージのビルド(9-82 ページ).

関連情報

ホワイト ペーパー: ARMv8-M アーキテクチャの技術概要.
-march armclang オプション.
--target armclang オプション.
--cpu armlink オプション.
--scatter armlink オプション.

9.4 以前生成したインポート ライブラリを使用したセキュア イメージのビルド

新しいバージョンのセキュア イメージをビルドして、以前のバージョンに存在したエントリ ポイントと同じアドレスを使用することができます。以前のバージョンのセキュア イメージ用に生成されたインポート ライブラリを指定して、新しいセキュア イメージ用に別のインポート ライブラリを生成します。

前提条件

以下の手順は、完全な例ではありません。また、コードによって *セキュリティ属性ユニット* (SAU) が設定され、非セキュア スタートアップ コードが呼び出されるものとします。

以下の手順では、*9.2 ARMv8-M セキュリティ拡張機能を使用したセキュア イメージのビルド* (9-82 ページ) で作成されるインポート ライブラリ パッケージを所有していることが前提となっています。

手順

1. インタフェース ヘッダ ファイル `myinterface_v2.h` を作成して、非セキュア コードで使用するための C リンケージを指定します。

```
#ifdef __cplusplus
extern "C" {
#endif

int entry1(int x);
int entry2(int x);
int entry3(int x);
int entry4(int x);

#ifdef __cplusplus
}
#endif
```

2. セキュア コード `secure.c` 用の C プログラムに以下を含めます。

```
#include <arm_cmse.h>
#include "myinterface_v2.h"

int func1(int x) { return x; }
int __attribute__((cmse_nonsecure_entry)) entry1(int x) { return func1(x);  }
int __attribute__((cmse_nonsecure_entry)) entry2(int x) { return entry1(x); }
int __attribute__((cmse_nonsecure_entry)) entry3(int x) { return func1(x) + entry1(x);  }
int __attribute__((cmse_nonsecure_entry)) entry4(int x) { return entry1(x) * entry2(x); }

int main(void) { return 0; }
```

2 つのエントリ関数の実装に加えて、このコードはセキュア コードによってのみ呼び出される関数 `func1()` を定義します。

注

セキュア コードを C++ としてコンパイルしている場合、`__attribute__((cmse_nonsecure_entry))` として宣言された関数に `extern "C"` を追加する必要があります。

3. `armclang -mcmse` コマンドライン オプションを使用して、オブジェクト ファイルを作成します。

```
$ armclang -c --target arm-arm-none-eabi -march=armv8-m.main -mcmse secure.c -o secure.o
```

4. `armclang` によって生成されるマシン コードの逆アセンブリを確認するには、以下を入力します。

```
$ armclang -c --target arm-arm-none-eabi -march=armv8-m.main -mcmse -S secure.c
```

逆アセンブリはファイル `secure.s` に保存されます。以下に例を示します。

```
    .text
    ...
    .code 16
    .thumb_func

    ...

func1:
```

```
    .fnstart
    ...
    bx lr

    ...

__acle_se_entry1:
entry1:
    .fnstart
@ BB#0:
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl func1
    pop.w {r7, lr}
    ...
    bxns lr

    ...

__acle_se_entry4:
entry4:
    .fnstart
@ BB#0:
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl entry1
    ...
    pop.w {r7, lr}
    bxns lr

    ...

main:
    .fnstart
@ BB#0:
    ...
    movs r0, #0
    ...
    bx lr

    ...
```

エントリ関数はセキュア ゲートウェイ (SG) 命令で始まっていません。2 つのシンボル `__acle_se_`*`entry_name`* および *`entry_name`* は、リンカへのエントリ関数の開始を示しています。

5. `Veneer$$CMSE` セレクタを含むスキャッタ ファイルを作成して、*非セキュア呼び出し可能* (NSC) メモリ領域にエントリ関数のベニアを配置します。

```
LOAD_REGION 0x0 0x3000
{
    EXEC_R 0x0
    {
        *(+RO,+RW,+ZI)
    }
    EXEC_NSCR 0x4000 0x1000
    {
        *(Veneer$$CMSE)
    }
    ARM_LIB_STACK 0x700000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
...
```

6. `armlink --import-cmse-lib-out` および `--import-cmse-lib-in` コマンドライン オプションと前処理済みのスキャッタ ファイルを使用してオブジェクト ファイルをリンクし、セキュア イメージを作成します。

```
$ armlink secure.o -o secure.axf --cpu 8-M.Main --import-cmse-lib-out importlib_v2.o --
import-cmse-lib-in importlib_v1.o --scatter secure.scf
```

最終的なイメージに加えて、この例のリンクからは非セキュア イメージのビルド用のインポート ライブラリ `importlib_v2.o` も生成されます。ベニアを含むセクションがアドレス `0x4000` に配置されると仮

定した場合、インポート ライブラリは以下のエントリを含むシンボル テーブルのみを含む再配置可能なファイルで構成されます。

シンボル型	Name	アドレス
STB_GLOBAL, SHN_ABS, STT_FUNC	entry1	0x4001
STB_GLOBAL, SHN_ABS, STT_FUNC	entry2	0x4009
STB_GLOBAL, SHN_ABS, STT_FUNC	entry3	0x4021
STB_GLOBAL, SHN_ABS, STT_FUNC	entry4	0x4029

このアセンブリ コードに対応する再配置可能なファイルをイメージにリンクすると、リンカによってエントリ ベニアのみを含むセクションにベニアが作成されます。

7. リンカによって生成されるエントリ ベニアを確認するには、以下のコマンドを入力します。

```
$ fromelf --text -s -c secure.axf
```

この例では、以下のエントリ ベニアが EXEC_NSCR *execute-only* (XO) 領域に作成されます。

```
...
** Section #3 'EXEC_NSCR' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 64 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....SG        ; [0x3e08]
        0x00004004:    f7fcb85e    ..^.B        __acle_se_entry1 ; 0xc4
    entry2
        0x00004008:    e97fe97f    ....SG        ; [0x3e10]
        0x0000400c:    f7fcb86c    ..l.B        __acle_se_entry2 ; 0xe8

...

    entry3
        0x00004020:    e97fe97f    ....SG        ; [0x3e28]
        0x00004024:    f7fcb872    ..r.B        __acle_se_entry3 ; 0x10c
    entry4
        0x00004028:    e97fe97f    ....SG        ; [0x3e30]
        0x0000402c:    f7fcb888    ....B        __acle_se_entry4 ; 0x140

...
```

ベニアを含むセクションは 32 バイト境界で整列しており、32 バイト境界にパディングが挿入されます。

スキャッタ ファイルを使用しない場合、エントリ ベニアは最初の実行領域として ER_XO セクションに配置されます。既存のエントリ ポイントのエントリ ベニアは、CMSE ベニア セクションに配置されます。例えば、

```
...
** Section #1 'ER_XO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00008000

    $t
    entry3
        0x00008000:    e97fe97f    ....SG        ; [0x7e08]
        0x00008004:    f000b87e    ..~.B.W      __acle_se_entry3 ; 0x8104
    entry4
        0x00008008:    e97fe97f    ....SG        ; [0x7e10]
        0x0000800c:    f000b894    ....B.W      __acle_se_entry4 ; 0x8138

...
** Section #4 'ER$$Veneer$$CMSE_AT_0x00004000' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR
+ SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
```

```
        0x00004000:    e97fe97f    ....SG        ; [0x3e08]
        0x00004004:    f004b85a    ..Z.B.W      __acle_se_entry1 ; 0x80bc
    entry2
        0x00004008:    e97fe97f    ....SG        ; [0x3e10]
        0x0000400c:    f004b868    ..h.B.W      __acle_se_entry2 ; 0x80e0

...
```

後発条件

更新されたセキュア イメージをビルドした後、以下の手順を実行します。

1. 更新されたセキュア イメージをデバイスに事前にロードします。
2. 事前にロードされたイメージを含むデバイスを、新しいインポート ライブラリ パッケージと一緒に、このデバイスの非セキュア コードを開発する担当者に届けます。インポート ライブラリ パッケージの内容は、以下のとおりです。
   - インタフェース ヘッダ ファイル `myinterface_v2.h`。
   - インポート ライブラリ `importlib_v2.o`。

関連タスク

9.2 ARMv8-M セキュリティ拡張機能を使用したセキュア イメージのビルド(9-82 ページ).
9.3 セキュア イメージを呼び出すことができる非セキュア イメージのビルド(9-86 ページ).

関連情報

ホワイト ペーパー: ARMv8-M アーキテクチャの技術概要.
-c armclang オプション.
-march armclang オプション.
-mcmse armclang オプション.
-S armclang オプション.
--target armclang オプション.
__attribute__((cmse_nonsecure_entry)) 関数属性.
SG 命令.
--cpu armlink オプション.
--import_cmse_lib_in armlink オプション.
--import_cmse_lib_out armlink オプション.
--scatter armlink オプション.
--text fromelf オプション.