

Application Note

MSPM0 マイコンによるサイズ重視の C コード最適化: ヒントとコツ**概要**

マイコン (MCU) を選択する際、多くの場合、利用可能なコード空間と不揮発性メモリが意思決定プロセスで重要な要素になります。ターゲット デバイスの制約内で機能を最大化するために、開発者はソフトウェア開発およびコンパイル時に特定の戦略を利用して、コード サイズを効率化できます。このアプリケーション ノートでは、MSPM0 デバイスと開発する際の Code Composer Studio™ (CCS) のさまざまな最適化構成について紹介し、コード フットプリントを大幅に削減できるコンパイラ構成について解説します。さらに、このドキュメントでは、よりコンパクトでサイズが最適化された MSPM0 アプリケーション用コードを作成するための、コーディング上のベスト プラクティスや手法についても解説します。

目次

1 概要.....	2
2 C コンパイラの最適化.....	3
2.1 最適化設定.....	3
2.2 ランタイム モデル オプション.....	7
3 コーディング手法.....	9
3.1 変数と定数に可能な限り小さい型を使用する.....	9
3.2 乗算と除算を避ける.....	9
3.3 手動計算の代わりにルックアップ テーブルを使用する.....	9
3.4 機能を効果的に使用する.....	9
3.5 ダウン カウント ループを実装する.....	10
3.6 __even_in_range() 組み込み関数を使用する.....	10
3.7 volatile キーワード.....	10
4 例.....	11
4.1 最適化レベル.....	11
4.2 LTO.....	11
4.3 ランタイム モデル オプション.....	11
5 まとめ.....	12
6 参考資料.....	13

商標

Code Composer Studio™ is a trademark of Texas Instruments.

すべての商標は、それぞれの所有者に帰属します。

1 概要

コンパイラの設定を戦略的に選択し、サイズの最適化を意識してコードを記述すると、コード サイズを大幅に削減できる場合があります。このアプリケーション ノートで例として使用されているコード サイズは、16kB のフラッシュと 1kB の SRAM を含む MSPM0C1104 マイコン用に構築可能なサンプル コード `i2c_controller_target_dynamic_switching` を使用しています。記載されているコード サイズは、CCS バージョン 20.4 で TI Clang バージョン 4.0.4.LTS を使用してビルドされたものです。

2 C コンパイラの最適化

一般的に、読みやすさの観点からはアセンブリより C が好まれますが、コード空間が限られたデバイスでは、C によって大きなオーバーヘッドが生じる可能性があります。ただし、コンパイラの最適化を適切に行い、グローバル変数などの機能を戦略的に活用することで、C でもアセンブリレベルに近い効率を実現できます。そのうえ、手作業でアセンブリをチューニングする必要がなく、コンパイラによる自動最適化の恩恵も受けられます。

2.1 最適化設定

CCS では、最適化レベルの設定は、「Project」(プロジェクト) > 「Project Properties...」(プロジェクト プロパティ...) > 「Build」(ビルド) > 「Arm Compiler」(Arm コンパイラ) > 「Optimization」(最適化) の順にアクセスします。

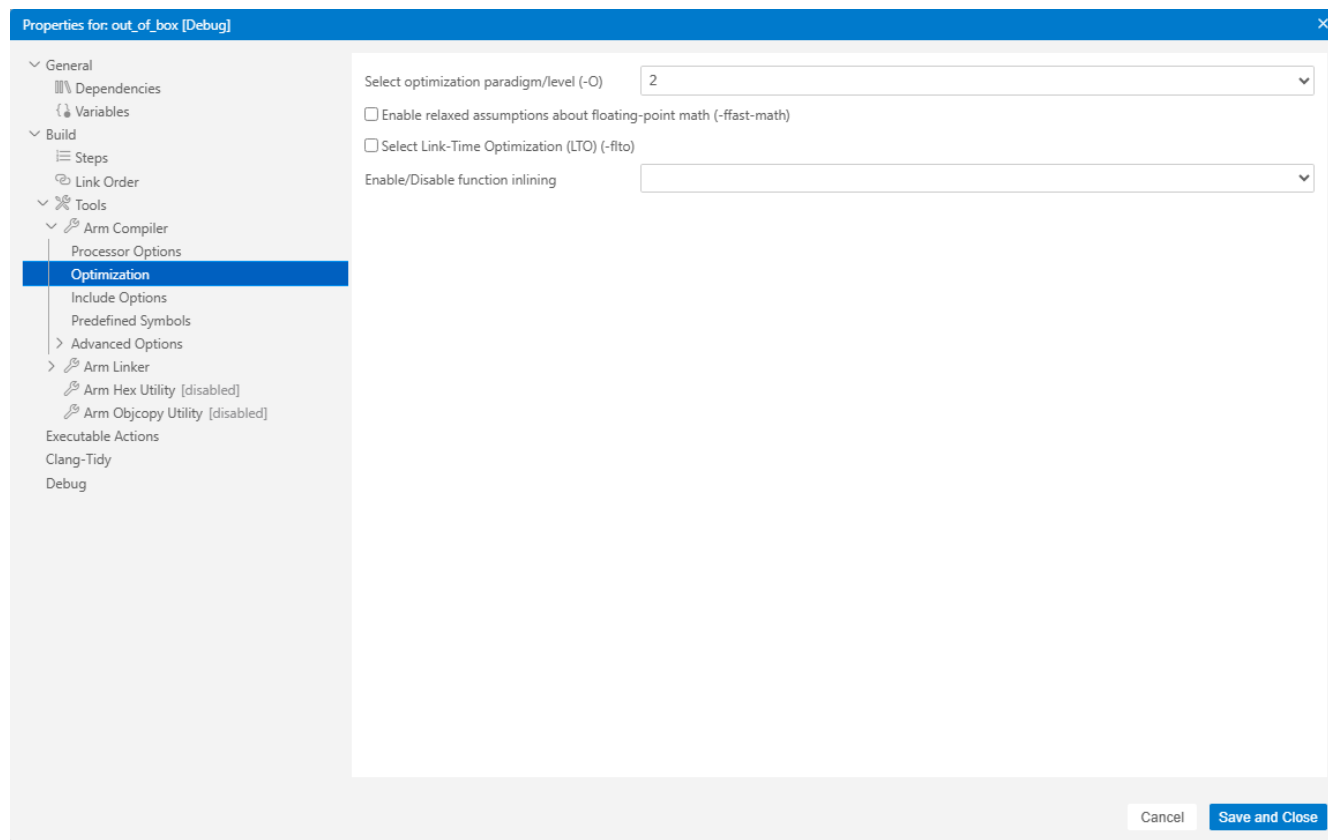


図 2-1. 最適化設定

2.1.1 最適化レベル

最適化レベル フラグは、パフォーマンスを犠牲にしてコンパイラ生成のコード サイズを小さくすることも、コード サイズを大きくすることでパフォーマンスを向上させるように設定することもできます。コンパイラの選択は、最適化アルゴリズムとコード生成方法が異なるため、コード サイズに直接影響することに注意してください。

表 2-1. 最適化レベル

		コード最適化レベル							
IDE	Keil	O0	O1	O2	O3	Ofast	Os (バランス型)	Oz (画像サイズ)	Og
	CCS	O0	O1	O2	O3	Ofast	Os	Oz	/
	IAR	なし	低	中	高	高速	バランス重視	サイズ重視	/

表 2-2. 最適化の概要

最適化レベル オプション	最適化の概要	実施した最適化
-O0	最適化を実行しません。このオプションは、デバッグのしやすさを最大限に高めるために推奨されています。	なし
-O1, -O	制限付きの最適化を有効にします。コード サイズとデバッグのしやすさのバランスがとれています。	- 制御フローの単純化 - 隣接する <code>icmp</code> を <code>memcpy</code> にマージします <code>memcpy/memset/memcmp</code> のインライン化 - 定数ホイスティング - ライブラリ関数への部分的なインライン呼び出し <code>always_inline</code> 関数のインライン - グローバル変数のマージ - 分離スタック スロットをマージ - ループ強度の低減 - ループ不変コード移動 - 共通部分式除去 - 不要な引数の除去 - マシン コードのシンク - ピープホルの最適化 - 末尾述語化 - 末尾複製 - ロード / ストアの最適化 - 簡単なレジスタ統合 - コピー伝播 - 条件付き定数伝播 - 呼び出し値伝播 - 制御フローの最適化 - If 変換 - Thumb2 命令サイズの縮小 - デッド コードの除去 - ループ ベクトル化 <code>printf</code> 関数の特殊化 - 小サイズの <code>memcpy/memset</code> 関数の特殊化 - 不要なライブラリ呼び出しの条件付き除去 - ループ回転 - ループ展開

表 2-2. 最適化の概要 (続き)

最適化レベル オプション	最適化の概要	実施した最適化
-O2	ほとんどの最適化を有効にします。コンパイラ生成のコードサイズが大きくなる可能性がある一部の最適化を無効にします。	- すべての (-O1) 最適化を実行し、さらに次のことを実行します。 - 関数の統合 / インライン化 - 命令投機 - 値の伝播 - ジャンプ スレディング (DFA 以外) - 末尾呼び出し除去 - ロード / スタ移動のマージ - グローバル値の番号付け - メモリ依存関係解析 - デッド スタ除去 - スーパーワードレベル並列性 (SLP) ベクトル化 - 冗長な命令の結合 - 不要なグローバル要素の除去 - 重複するグローバル定数のマージ - 高速 memcpy/memset 関数の特殊化 - ループのターゲット境界を 16 バイトに整列 (Cortex-R4/R5)
-O3	-O2 で利用可能なすべての最適化に加えて、コンパイラが生成するコード サイズを増加させる可能性のある、その他の最適化も有効にします。このオプションは性能を最適化するために推奨されますが、コード サイズが大きくなる可能性があります。	- 速度に合わせて調整されたすべての (-O2) 最適化を実行します。さらに次のことを実行します。 - 関数のサポートされている組み込み関数への置き換え - 追加のエイリアス解析とループ最適化 - 積極的な関数のインライン化 - 呼び出しサイトの分割 「参照渡し」の引数のスカラへの昇格 - パターン ベースの式の組み合わせ
-Og	デバッグ機能を維持しながら、制限付きの最適化を有効にします。-O2 からの最適化を追加して、-O1 で使用可能なすべての最適化を有効にします。	- すべての (-O2) 最適化を実行しますが、次のことは無効にします。 - ループ ベクトル化なし - always_inline 関数を除く関数のインライン化なし - 命令投機なし - ジャンプ スレディングなし - 値の伝播なし - 末尾呼び出し除去なし - ロード / スタ移動のマージなし - グローバル値の番号付けなし - メモリ依存関係解析なし - スーパーワードレベル並列性 (SLP) ベクトル化なし - 不要なグローバル要素の除去なし - 重複するグローバル定数のマージなし

表 2-2. 最適化の概要 (続き)

最適化レベル オプション	最適化の概要	実施した最適化
-Os	O2 で利用可能なすべての最適化と追加の最適化設計を有効にして、コード サイズを縮小します。 -Oz オプションと比較して、 -Os オプションはパフォーマンスに悪影響を及ぼす可能性のあるコード サイズの最適化を回避します。	- コード サイズに合わせて調整されたすべての (-O2) 最適化を実行します。さらに次のことを実行します。 - これまでに有効になっている、コード サイズに合わせて調整された最適化 - 小サイズの <code>memcpy/memset</code> 関数の特殊化 - 最小限の <code>memcpy/memset/memcmp</code> のインライン化 - 不要なライブラリ呼び出しの条件付き除去をしない - ループのターゲット境界を 16 バイトに整列しない (Cortex-R4/R5)
-Oz	-O2 で利用可能なすべての最適化と追加の最適化設計を有効にして、コード サイズを縮小します。 -Os オプションと比較して、 -Oz オプションは、パフォーマンスに悪影響を与える可能性のある追加のコード サイズの最適化を実行します。この最適化設定は、コード サイズの小ささを優先する場合に推奨されます。	- すべての (-Os) 最適化を実行し、さらに次のことを実行します。 Machine Outlining (マシン アウトライニング) - ループ ベクトル化なし - コード サイズに影響する最適化をより控えめにする - ループのターゲット境界を 16 バイトに整列しない (Cortex-R4/R5)
-Ofast	このオプションは非推奨です。このオプションは、 -O3 で利用可能なすべての最適化に加えて、言語標準への厳密な準拠が保証されない、より積極的な最適化を有効にします。この動作が必要な場合は、 -O3 オプションを -fast-math と組み合わせて使用することをお勧めします。	- すべての (-O3) 最適化を実行します - さらなる積極的な最適化

さまざまな最適化レベルでのアセンブリ コードの影響を示す例については、『[TI Arm Clang コンパイラ ツール ユーザーガイド](#)』を参照してください。

2.1.2 Optnone

optnone 関数属性を関数に適用して、関数のコード生成時に最適化を適用しないようコンパイラに指示できます。これは、アプリケーション全体に対するコマンドライン オプション **-O0** と同じ効果を、単一の関数に対して及ぼします。

シンタックス

```
<return type> symbol (<arguments>) __attribute__((optnone));
```

例

次の C の例では、**park_and_wait** 関数に空の **while** ループが含まれています。**optnone** 属性を **park_and_wait** に適用しない場合、コンパイラはこのループを最適化によって削除し、さらにこの関数への参照も削除します。

```
void check_peripheral();
__attribute__((optnone)) void park_and_wait();
void run_a_check_on_peripheral(void) {
    check_peripheral();
}
void check_peripheral() {
    if (*(unsigned long *) (268612608)) != 286529877) {
        park_and_wait();
    }
}
__attribute__((optnone)) void park_and_wait() {
    while(1) {
        ;
    }
}
```

上記のコードのコンパイラ生成アセンブリは、-O2 最適化オプションが指定されていても、**park_and_wait** への参照はそのままであることを示しています。optnone 属性が **park_and_wait** 関数に適用されていない場合、オブティマイザは **park_and_wait** のループが空であることを検出し、**check_peripheral** の参照を削除します。

```
%> tiarmclang -mcpu=cortex-m4 -O2 -S optnone_function_attr.c
%> cat optnone_function_attr.s
...
.section .text.check_peripheral,"ax",%progbits
.hidden check_peripheral @ -- Begin function check_peripheral
.globl check_peripheral
.p2align 2
.type check_peripheral,%function
.code 16 @ @check_peripheral
.thumb_func
check_peripheral:
movw r0, #46080
movt r0, #4098
ldr r0, [r0]
movw r1, #6485
movt r1, #4372
cmp r0, r1
it eq
bxeq lr
.LBB1_1: @ %if.then
bl park_and_wait
...
```

2.1.3 リンク時最適化 – LTO

LTO を使用すると、コンパイラは各ファイルを個別に最適化するのではなく、リンク フェーズ中に複数のソース ファイル間でコードを最適化します。LTO は **-flto** コンパイラ フラグで有効にできます。

Cortex-M0+ プロセッサの場合、-Oz オプションを使用してコンパイルした 191 M0+ ドライバライブラリ アプリケーションでテストした結果、LTO はコード サイズを 23 ~ 25% 削減できました。コード サイズの削減率は、次の式で計算できます。

$$\text{コードサイズ削減率 (\%)} = (1 - (\text{LTO コード サイズ} / \text{非 LTO コード サイズ})) \times 100$$

LTO を有効にすることで、大幅な高速化も可能になります。Cortex-M0+ プロセッサの場合、-O3 オプションでコンパイルされた 15 個の EEMBC AutoBench アプリケーションを用いてテストした結果、最大で 1.20 倍の高速化と、実行時間 / サイクル数 3.8% の削減が確認されました。以下の式で、これらの指標を計算できます。

$$\text{高速化倍率} = (\text{非 LTO サイクル} / \text{LTO サイクル})$$

$$\text{サイクル削減率 (\%)} = ((\text{非 LTO サイクル} - \text{LTO サイクル}) / \text{非 LTO サイクル}) \times 100$$

2.2 ランタイム モデル オプション

CCS では、ランタイム モデル オプションの設定は、「Project」(プロジェクト) > 「Project Properties...」(プロジェクト プロパティ...) > 「Build」(ビルド) > 「Arm Compiler」(Arm コンパイラ) > 「Runtime Model Options」(ランタイム モデル オプション) の順にアクセスします。

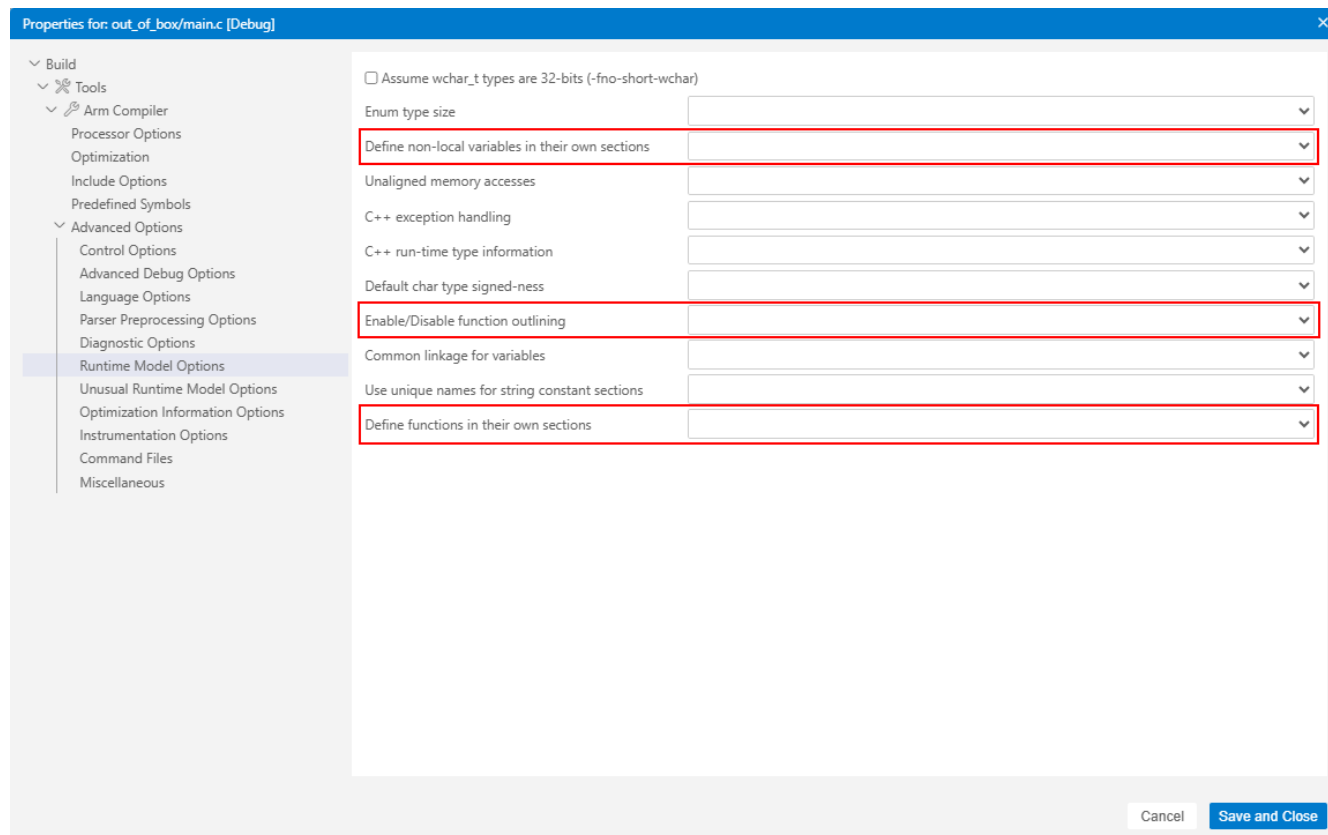


図 2-2. ランタイム モデル オプション

2.2.1 データおよび関数セクション

デフォルトでは、`-fdata-sections` および `-function-sections` オプションが有効になっています。これらのフラグを指定すると、コンパイラはデータ定義および関数定義ごとに、それぞれ独立したセクションにコードを生成します。その結果、これらのフラグを使用すると、よりきめ細かいデッド コードの除去が可能になります。

2.2.2 Function Outlining (関数アウトライニング)

Function Outlining (関数アウトライニング) (別名 Machine Outlining) は、マシン コードの繰り返しシーケンスを識別し、各インスタンスをそれらの操作を含む新しい関数の呼び出しに置き換える最適化です。`-Oz` オプションを選択すると、Function Outlining (関数アウトライニング) が有効になります。

表 2-3. Function Outlining (関数アウトライニング)

関数アウトライニング オプション	有効
<code>-moutline</code>	関数内で machine outlining (マシン アウトライニング) を実行します。ネット コード サイズが小さくなることが保証されている場合にのみ適用されます。
<code>-moutline-inter-function</code>	より積極的な function outlining (関数アウトライニング) が可能ですが、必ずしもコード サイズの縮小を保証するとは限りません。 <code>-Oz</code> オプションと組み合わせて指定すると、関数間のアウトライニングが有効になります。
<code>-mno-outline</code>	<code>-Oz</code> オプションを使用すると、指定されたコンパイル単位の function outlining (関数アウトライニング) を無効化します。

3 コーディング手法

3.1 変数と定数に可能な限り小さい型を使用する

定数は、コードと同様に不揮発性メモリに格納されます。そのため、定数を定義するときに可能な限り小さい型を使用すると、コードの無駄なスペースを減らすことができます。たとえば、ルックアップ テーブルに **0 ~ 255** の値しか含まれていない場合、定数テーブルを宣言するときに符号なし **8 ビット型**を使用すると、**16 ビット int 型**と比べてスペースを半分に減らせます。この概念は、**PERSISTENT** キーワードを使用してフラッシュに格納されている変数にも当てはまります。変数を **SRAM** ではなくフラッシュに格納すると、その変数によって、本来ならコード用に使用できたフラッシュの領域が消費されます。したがって、この場合も小さな型を使用することが重要です。

アプリケーションに十分な精度を維持していれば、ルックアップ テーブルに使用される数値が小さいコードを記述することもできます。一例として、タイマ **PWM** 出力用の値を含む定数配列が挙げられます。タイマのソースが **32768Hz** で、タイマ出力が **60Hz** しかない場合、タイマ周期またはデューティ サイクルの最大カウント値は $32768/60 = 545$ にしかありません。この値は、**8 ビット**の変数には格納できません。ただし、クロック モジュールまたはタイマ内の内部クロック分周器を使用してタイマ ソースを **4 分の 1** に分周すると、デューティ サイクルの最大カウント値は $8192/60 = 136$ になり、**8 ビット**値に格納できるほど小さい値になります。これにより、デューティ サイクル設定において十分な精度を提供できるため、この簡単な変更により、タイマ カウント設定を含む **const** ルックアップ テーブルのサイズを半減させることができます。値の格納方法を賢く選択することで、配列とルックアップ テーブルのサイズを大きく変えられます。

可能な限り最小のデータ型を選択するだけでなく、複数のブール型フラグを個別の変数として扱う代わりにビットフィールドを使用して **1 バイト**に詰め込むことで、さらなる最適化が可能です。読み取り専用のデータには **const** 修飾子を付けて、**SRAM** の領域を消費せず、フラッシュ メモリに格納されるようにすることも重要です。さらに、文字列リテラルは利用可能なメモリを急速に消費する可能性があるため、コード内での使用は避けるようにしましょう。

3.2 乗算と除算を避ける

乗算や除算の処理は実行に多くのサイクルを要し、それらを行うためにより多くのコードを必要とします。したがって、不要な乗算や除算をコードから取り除く方法を見つけることは、コードのスペースと実行時間の両方を節約する優れた方法となります。

2 の累乗 (たとえば、**2, 4, 8, 16, ...**) による乗算や除算を行う場合は、代わりにビット シフトを使用できます。**C** 言語でビット シフトを行うには、右シフトには **>>** 左シフトには **<<** を使用し、その後にシフトするビット数を指定します。**CPU** 内にはビット シフト用のアセンブリ命令やハードウェアがあるため、これは、乗算や除算よりもはるかに効率的です。

別の方法として、乗算や除算を実行時ではなく、コンパイル時に計算できるかどうかを判断する方法があります。たとえば、変数を持たない **2** つの定数を乗算する場合、コンパイラはコンパイル中に結果を計算して計算値を直接使用することができるため、メモリと実行時間の両方を消費する乗算コードを必要としません。

変数を含む乗算などで事前にすべての計算を行えない場合でも、その変数の値の範囲が既知であれば、計算結果の候補を格納したルックアップ テーブルを作成できます。

3.3 手動計算の代わりにルックアップ テーブルを使用する

異なる値に対して複雑な計算を繰り返し実行する必要がある場合は、ルックアップ テーブルの方が適していないか検討してみてください。これは浮動小数点計算において特に当てはまります。計算に入力される変数の値の範囲が既知であれば、起こり得る結果を格納したルックアップ テーブルを作成できます。ルックアップ テーブルは、テーブル自体もデバイスの不揮発性メモリに格納する必要があるため、必ずしもサイズを小さくできるとは限らないことに注意してください。

3.4 機能を効果的に使用する

関数はコードを読みやすくすることができます。ただし、関数は実行速度やコードサイズの面でオーバーヘッドを生じさせることもあります。関数が **1** 回だけ呼び出された場合、特に小さい関数の場合は、余分なオーバーヘッドは通常価値がありません。読みやすさのために小さな関数を残しつつ、関数宣言に **inline** (または **__inline**) キーワードを使用してインライン化することで、余分なオーバーヘッドを回避できます。この宣言は、実際の関数呼び出しを実行する代わりに、関数が呼び出されるすべての場所にその関数の内容を挿入するようコンパイラに指示します。

ただし、コード内で何かを何度か実行する必要がある場合もあります。この場合、同じコードを複数の異なる場所で再利用できるため、関数はよりコード サイズの効率的なソリューションを作成できます。意図的に再利用性を考慮してコードを記述すると、コード サイズにも大きな違いが生じる可能性があります。

3.5 ダウン カウント ループを実装する

繰り返しのアクションやループを使用するアプリケーションでは、適切なループ構造を選択するとコード サイズに影響を与えることがあります。ダウン カウント ループは、プロセッサ内蔵のゼロ フラグを利用するため、一般的にアップ カウント ループよりも効率的です。ループがゼロまでカウント ダウンされると、デクリメント命令は自動的にゼロ フラグをセットし、追加の比較命令なしでループ終了条件をチェックできるようにします。これに対して、アップ カウント ループは明示的な比較命令を必要とし、反復のたびにターゲット値をロードする必要があります。

3.6 `__even_in_range()` 組み込み関数を使用する

`__even_in_range(x, NUM)` 組み込み関数は、`switch` 文において、変数 `x` の値が常に 0 から `NUM` の範囲内の偶数値であることをコンパイラに示唆します。これにより、コンパイラは `x` の値についてより多くの前提条件を設けることができ、通常よりも高度な最適化が可能になります。割り込みベクタの値は常に特定の範囲に収まり、かつ常に偶数であるため、この関数は一般的に割り込みサービス ルーチン (ISR) で使用されます。

ただし、`__even_in_range()` の用途は ISR に限定されません。`switch` 文の制御変数が常に偶数であり、その値に上限が設けられていることがコード上で保証されていれば、コード内の他の `switch` 文でもこの組み込み関数を使用できます。その一例として、1 ではなく 2 ずつ増加するバイト カウンタ変数が挙げられます。この変数は、コードによって最大値が制御され、0 または 2 にロール オーバーするように設計されています。

3.7 `volatile` キーワード

`volatile` キーワードは C 言語規格の一部です。tiarmclang コンパイラは、すべての言語モードでこのキーワードをサポートしています。

`volatile` キーワードは、変数へのアクセス方法に関して、コンパイラがその変数を含む式に対して過度な最適化を行ってはならないことをコンパイラに示します。たとえば、変数は外部プログラム、割り込み、別のスレッド、またはペリフェラル デバイスからもアクセスされる可能性があります。

コンパイラは、データフロー解析を用いて最適化が可能かどうかを判断し、可能な限り冗長なメモリ アクセスを排除します。しかし、メモリ アクセスの中にはコンパイラが認識できないような特殊なものがあります。そのような場合には、コンパイラが重要なものを最適化によって削除してしまわないよう、`volatile` キーワードを使用する必要があります。`volatile` と宣言された変数へのアクセスは、コンパイラによって最適化されることはありません。`volatile` 変数の読み取りや書き込みの回数は、C/C++ コード上の記述と完全に一致し、増減せず、その順序も維持されます。

プログラムの通常の制御フローとは異なる要因 (割り込みサービス ルーチンなど) によって変更される可能性のある変数は、すべて `volatile` と宣言する必要があります。これにより、割り込み関数がいつでもその変数の値を変更する可能性があることがコンパイラに伝わり、コンパイラはその変数へのアクセス回数や順序を変更するような最適化を行わなくなります。これが、`volatile` キーワードの主な目的です。

4 例

最適化レベル -O2 で i2c_controller_target_dynamic_switching のサンプルをビルドした場合、メモリ使用量はフラッシュが約 2.5kB、SRAM が 117B となります。



図 4-1. メモリの割り当て

4.1 最適化レベル

表 4-1 に、異なる最適化レベルを適用した場合の例のコード サイズを示します。

表 4-1. コードのサイズ

最適化レベル	コードのサイズ
-O0	4,112B
-O1	2,208B
-O2	2,464B
-O3	2,456B
-Og	2,208B
-Os	2,160B
-Oz	2,128B

4.2 LTO

-Oz 最適化レベルで LTO を有効にすると、コード サイズは 1,896B に減少しますが、SRAM の使用量はわずかに増加して 128B になります。



4.3 ランタイム モデル オプション

最適化レベル -O2 を使用して以下のテストを実施しました。

表 4-2. ランタイム モデル オプション

LTO	アウトライニング	データ / 関数セクション	コード サイズ フラッシュ / SRAM
なし	-moutline	デフォルト (有効)	2,456B/105B
なし	-moutline-inter-function	デフォルト	2,456B/105B
なし	-mno-outline	デフォルト	2,464B/105B
なし	-moutline	-fno-data-sections -fno-function-sections	2,520B/105B
あり	-moutline	デフォルト	2,328B/105B

5 まとめ

多くの場合、最適なコーディング ソリューションを見つけるには、さまざまなアプローチをテストするのが最善の方法です。バージョン管理システムを活用すると、プロジェクトの別々のブランチまたはコピーを作成して、さまざまな最適化方法を試すことができます。その後、結果を比較して、どの結果が最適かを確認できます。

6 参考資料

1. テキサス・インスツルメンツ、『[TI Arm Clang コンパイラ ツール ユーザー ガイド - 5.0.0.STS](#)』
2. テキサス・インスツルメンツ、『[MSP430™ マイコンによるサイズ重視の C コード最適化:ヒントとコツ](#)』アプリケーション レポート

重要なお知らせと免責事項

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

これらのリソースは、TI 製品を使用する設計の経験を積んだ開発者への提供を意図したものです。(1) お客様のアプリケーションに適した TI 製品の選定、(2) お客様のアプリケーションの設計、検証、試験、(3) お客様のアプリケーションに該当する各種規格や、その他のあらゆる安全性、セキュリティ、規制、または他の要件への確実な適合に関する責任を、お客様のみが単独で負うものとし、TI は一切の責任を拒否します。

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

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

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

Copyright © 2026, Texas Instruments Incorporated

最終更新日：2025 年 10 月