PSoC™ Arm® Cortex®コードの最適化

本書について

適用範囲と目的

本書は、PSoC™ 4 および PSoC™ 5LP の Arm® Cortex® CPU 用に C およびアセンブラコードを最適化する方法を示します。CPU パフォーマンスを改善し、PSoC™メモリアーキテクチャを効果的に使用するためのコーディング技術が存在し、効率の向上と消費電力の削減につながります。本書では、PSoC™ 4 デバイスのフラッシュを最適化する方法に関する情報と推奨事項を提供します。

対象者

本書は、PSoC™ 4 および PSoC™ 5LP デバイスコードの Arm® Cortex® CPU および PSoC™ Creator および ModusToolbox™ソフトウェア用に C およびアセンブラコードを最適化したいお客様を対象とします。

目次

本書について 1
目次 2
1 はじめに 4
2 PSoC™ 4 および PSoC™ 5LP アーキテクチャ 5
2.1 レジスタセット 5
2.2 アドレスマップ 6
2.2.1 PSoC™ 4 アドレスマップ 7
2.2.2 PSoC™ 5LP アドレスマップ 8
2.3 割込み 9
3 コンパイラの一般的なトピック 10
3.1 コンパイラの定義済みマクロ 10
3.2 コンパイラ出力の表示 11
3.3 コンパイラの最適化 12
3.4 属性 13
4 変数へのアクセス 14
4.1 グローバル変数と静的変数 14
4.2 自動変数 15
4.3 関数の引数と結果 15
4.4 LDR および STR 命令 16
5 C とアセンブラコードの混合 18
5.1 構文 18
5.2 自動変数 20
5.3 グローバル変数と静的変数 22
5.4 関数の引数 25
6 特殊機能命令 27
6.1 サチュレーション命令 27
6.1.1 SSAT 命令 27
6.1.2 USAT 命令 28
6.1.3 構文 28
6.2 組込み関数 28
6.3 アセンブラ 29
7 パックおよびアンパックの構造体 30
7.1.1 コンパイラの考慮事項 31
8 コンパイラライブラリ 32
9 コードと変数の配置 35
9.1 リンカスクリプトファイル 35
9.1.1 GCC のリンカスクリプトファイル 36
9.1.2 MDK のリンカスクリプトファイル 39
9.1.3 IAR のリンカスクリプトファイル 41
9.1.4 変数の初期化 47
9.1.5 マップファイル 47
9.2 配置手順 48
9.2.1 カスタム領域の定義 48
9.2.2 関数と変数の宣言 49
9.2.3 リンカスクリプトファイルを変更する 49
9.3 例 52

目次

9.4 一般的な注意事項 54
9.5 EMIF の考慮事項 (PSoC™ 5LP のみ) 55
10 PSoC™ 4 デバイスコードを最適化するヒントとコツ 57
10.1 シンプルで高速な低電力モードエントリ機能の使用 57
10.2 アプリケーションで複数のペリフェラルが使用される場合のリソース設定の再利用 58
10.3 クロック, ペリフェラル, およびピン設定予約の最適化 59
10.4 デバイスコンフィギュレータの最適化 60
10.5 CAPSENSE™ミドルウェアライブラリの最適化 63
10.6 重要な注意事項 67
11 Cortex®-M3 ビットバンド (PSoC™ 5LP のみ) 68
12 DMA アドレス 70
13 まとめ 71
13.1 PSoC™デバイスのすべてのリソースの使用 71
14 付録: コンパイラ出力の詳細 73
14.1 アセンブラの例、Cortex®-M3 用 GCC 73
14.1.1 Cortex®-M3 用 GCC、なし, サイズ, および速度最適化 73
14.1.2 Cortex®-M3 用 GCC、デバッグ, 最小, および高最適化 80
14.2 アセンブラの例、Cortex®-M0/M0+ 用 GCC 86
14.2.1 Cortex®-M0/M0+用 GCC、なし, サイズ, および速度最適化 86
14.2.2 Cortex®-M0/M0+用 GCC、デバッグ, 最小, および高最適化 92
14.3 アセンブラ例、Cortex®-M3 用 MDK 97
14.4 アセンブラ例、Cortex®-M0/M0+ 用 MDK 103
14.5 アセンブラの例、Cortex®-M3 用 IAR 109
14.5.1 Cortex®-M3 用 IAR、なし, サイズ, および速度最適化 109
14.5.2 Cortex®-M3 用 IAR、低, 中, およびバランス最適化 115
14.6 アセンブラ例、Cortex®-M0/M0+ 用 IAR 120
14.6.1 Cortex®-M0/M0+用 IAR、なし, サイズ, および速度最適化 126
14.6.2 Cortex®-M0/M0+用 IAR、低, 中, およびバランス最適化 131
14.7 コンパイラテストプログラム 137
参考資料 143
改訂履歴 144
免責事項 145

1 はじめに

PSoC™ 4 および PSoC™ 5LP デバイスの Arm® Cortex® CPU は、非常に効率的な方法で C コードを実装するように設計されています。したがって、ほとんどの場合、PSoC™ 4 または PSoC™ 5LP の C プログラミングを行うために特別な知識は必要ありません。このアプリケーションノートは、より高度で固有の問題を解決するのに役立ちます。

- アプリケーションを少量のフラッシュまたは SRAM に適合させる
- 時間に制約のあるアプリケーション、つまりコードの速度と効率を最大化します

これらのタイプの問題を解決するために、いくつかの方法が提供されます。

このアプリケーションノートは、C 言語で組込みアプリケーションをプログラミングする方法を理解していることを前提としています。GCC (GNU Compiler Collection) または Keil MDK (Microcontroller Development Kit) C コンパイラの知識がある程度必要です。CPU で使用される Thumb-2 アセンブリ言語の知識も役立ちます。

PSoC™ 4 デバイスの場合、ModusToolbox™ ソフトウェアまたは PSoC™ Creator の使用方法を知っておく必要があります。PSoC™ 4 を初めて使用する場合は、AN79953 – Getting started with PSoC™ 4 MCU の使用開始でデバイスの概要を確認できます。ModusToolbox™ software を初めて使用する場合は、ModusToolbox™ software を参照してください。PSoC™ Creator を初めて使用する場合は、PSoC™ Creator を参照してください。

PSoC™ 5LP デバイスの場合、PSoC™ 3, PSoC™ 4, および PSoC™ 5LP 用の統合開発環境である PSoC™ Creator の使用方法を知っておく必要があります。PSoC™ 5LP を初めて使用する場合は、AN77759 – Getting started with PSoC™ 5LP でデバイスの概要を参照できます。PSoC™ Creator を初めて使用する場合は、PSoC™ Creator のホームページを参照してください。

Note: 多くの例では、Cortex®アセンブリ言語である Thumb-2 のコードを示しますが、このアプリケーションノートは、この言語のチュートリアルを意図したものではありません。Thumb-2 アセンブラの詳細とチュートリアルについては、Arm® Cortex®のドキュメントを参照してください。

PSoC™ 3 の 8051 CPU の C コードの最適化については、AN60630 - PSoC™ 3 8051 code and memory optimization を参照してください。

2 PSoC™ 4 および PSoC™ 5LP アーキテクチャ

このアプリケーションノートで説明されている方法を効果的に使用するには、それらが基づいているレジスタとアドレスのアーキテクチャを理解することが重要です。ここでは、これらのアーキテクチャについて説明します。

2.1 レジスタセット

Cortex®レジスタセットと命令セットは、非常に効率的な C コードを実装するための基盤です。Figure 1 に示すように、PSoC™ 4 Cortex®-M0 と PSoC™ 5LP Cortex®-M3 レジスタは非常に似ています。PSoC™ 4100PS / PSoC™ 4000S / 4100S デバイスの Cortex®-M0+レジスタは、以下の Cortex®-M0 レジスタに類似していることに注意してください。詳細については、Cortex®-M0+テクニカルリファレンスマニュアルの「Processor core registers summary」章を参照してください。

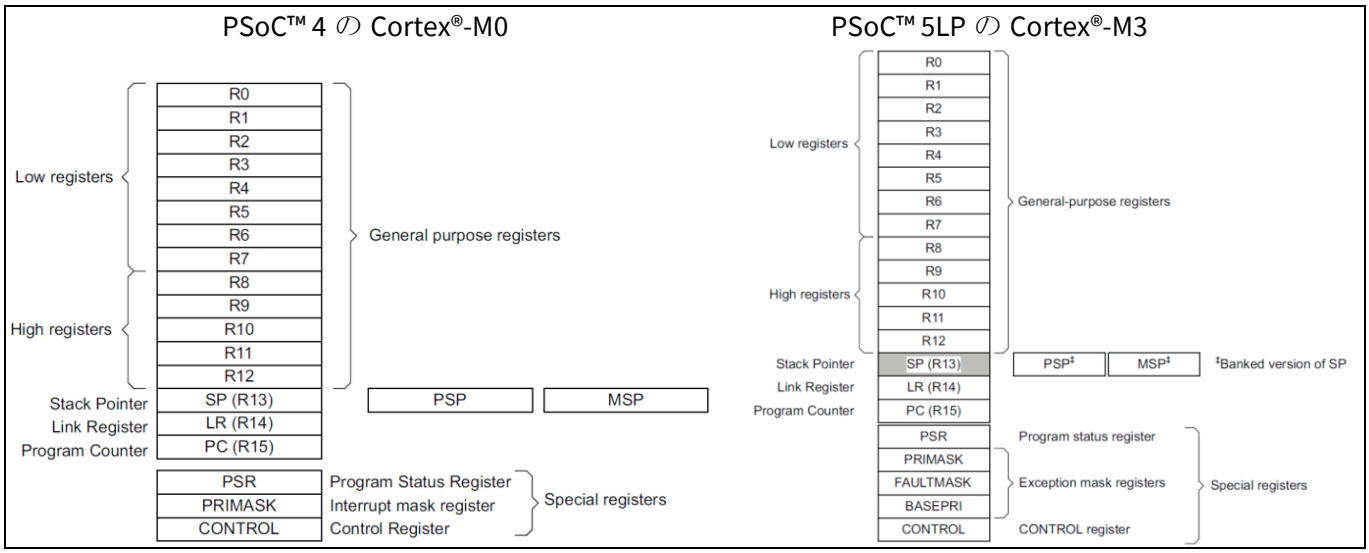


Figure 1 Cortex® CPU アーキテクチャ

すべてのレジスタは 32 ビットです。12 個の汎用レジスタがあります (下位レジスタ R0～R7 は、命令セットでより広範なサポートを備えています)。特殊レジスタには次のものがあります。

- リアルタイムオペレーティングシステム (RTOS) のより効率的な実装のためのデュアルスタックポインタ (R13)
- 関数呼び出しから高速に戻るためのリンクレジスタ (R14)
- プログラムカウンタ (R15)
- プログラムステータスレジスタ (PSR) には、ゼロやキャリーフラグなどの命令結果が含まれる
- 割込みマスクレジスタ (Cortex®-M0) / 例外マスクレジスタ (Cortex®-M3)
- 制御レジスタ

PSoC™ 5LP Cortex®-M3 には、スタック管理、および PSR、割込み、および制御レジスタに多くの機能があります。Cortex®-M3 には、除算 (UDIV、SDIV)、乗算と累算 (MLA、MLS)、サチュレーション (USAT、SSAT)、ビットフィールド命令など、より広範な命令セットもあります。これらの命令を利用する方法については、特殊機能命令を参照してください。

2.2 アドレスマップ

Figure 2 に示すように、Cortex®-M0/M0+と Cortex®-M3 は非常によく似たアドレスマップです。

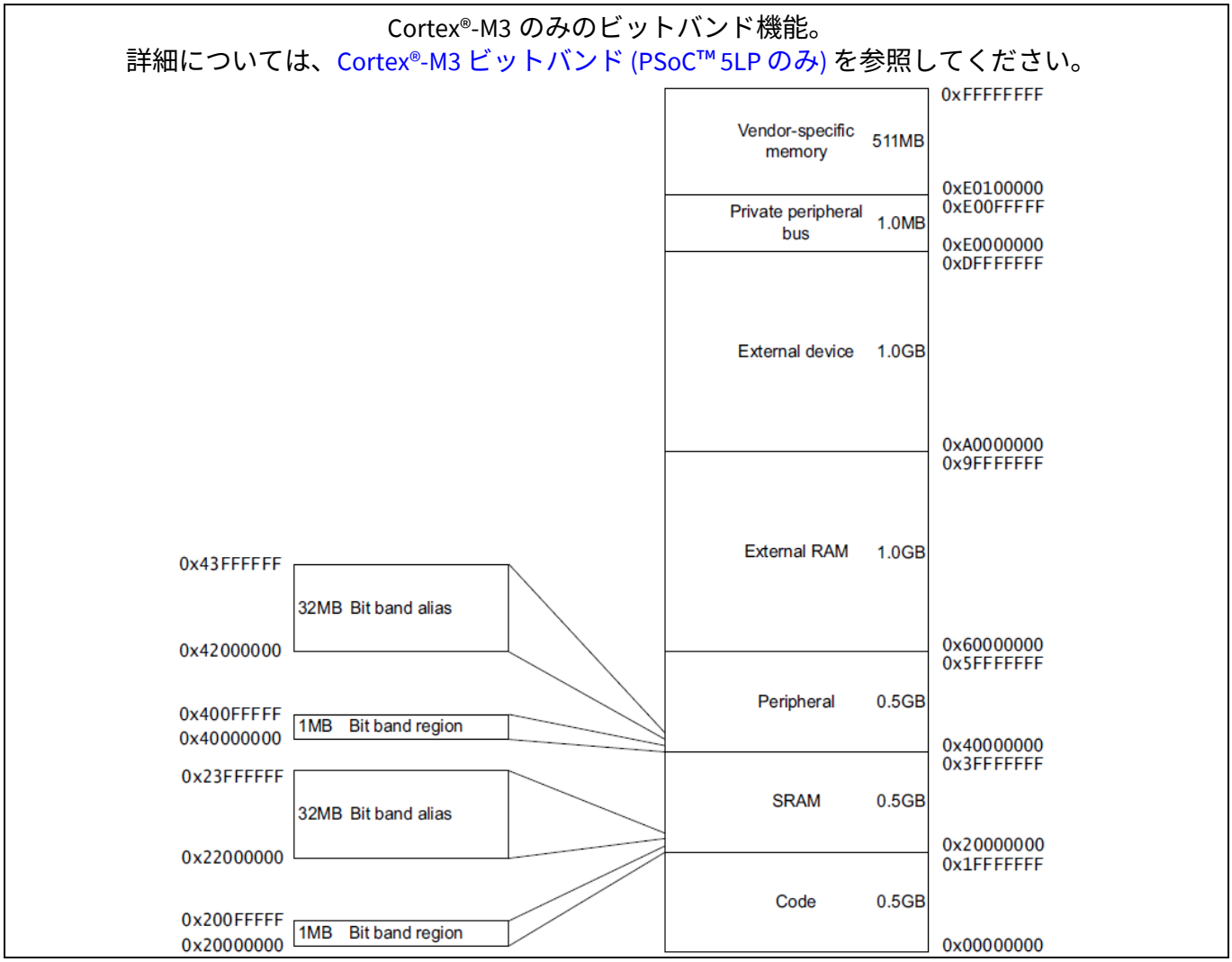


Figure 2 Cortex®アドレスマップ

アドレス空間は 4G バイト (32 ビットアドレッシング) で、Figure 2 に示すアクセス領域に分割されます。CPU は、コード、SRAM、および外部 RAM 領域で命令を実行できます。これらの領域のいずれかにコードまたはデータを配置できます。CPU には、命令の並列フェッチと実行を可能にする 3 命令パイプラインがあります。

PSoC™ 5LP Cortex®-M3 にはビットバンド機能があり、エイリアス領域でアドレスにアクセスすると、対応するビットバンド領域でビットレベルのアクセスが行われます。これにより、領域の最下部の 1MB のシングルビットをすばやく設定、クリア、またはテストできます。詳細については、Cortex®-M3 ビットバンド (PSoC™ 5LP のみ) を参照してください。

Cortex® CPU は 4GB のアドレス空間にアクセスできますが、PSoC™デバイス内では、これらのアドレスのごく一部のみが PSoC™メモリまたはレジスタにアクセスします。以下は、Cortex®アドレス空間のどこに PSoC™メモリとレジスタが配置されているかの概要です。詳細については、デバイスのデータシートまたはリファレンスマニュアル (RM) のメモリマップを参照してください。

2.2.1 PSoC™ 4 アドレスマップ

Figure 3 は、単一の Cortex®-M0/M0+バスであるシステムバスを使用して、アドレスマップのほとんどの領域にアクセスしていることを示します。

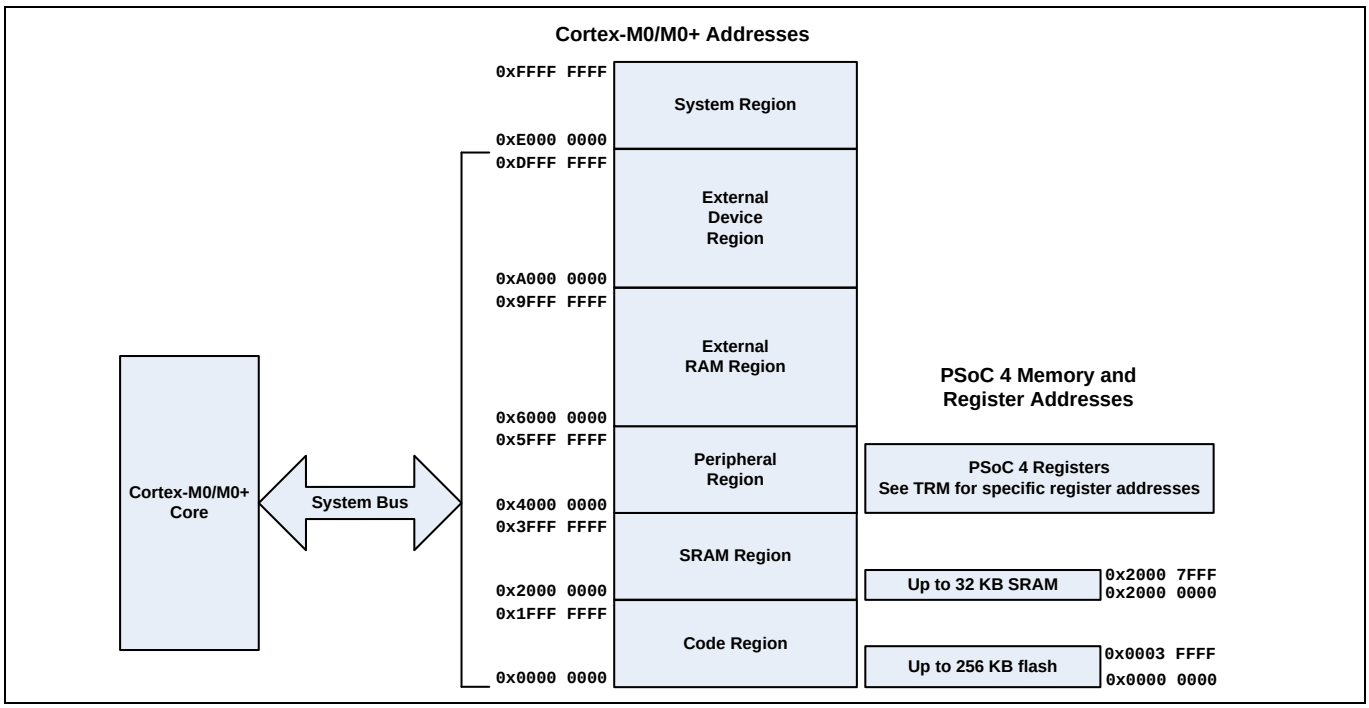


Figure 3 PSoC™ 4 アドレスマップ

PSoC™ 4 のメモリとレジスタは以下のようにアドレス指定されています。

- フラッシュは、Cortex®コード領域のアドレス 0 から始まります。フラッシュブロックには、読み出しアクセラレータが含まれます。詳細については、デバイスのデータシートを参照してください。
- SRAM は、Cortex® SRAM 領域のアドレス 0x20000000 から始まります。
- レジスタは、Cortex® Peripheral 領域の 0x40000000 からアドレス指定されます。特定のレジスタアドレスについては、リファレンスマニュアルを参照してください。

すべてのメモリアクセスは 32 ビットです。

コードは SRAM に配置できます。詳細については、コードと変数の配置を参照してください。

Note: PSoC™4 にはバスが1 つしかないため、コード実行とデータアクセスの速度と効率は、それらの領域を占有するメモリの速度にのみ依存します。通常、SRAM はフラッシュより高速です。ただし、Cortex®命令パイプラインとフラッシュ読み出しアクセラレータの組合せにより、コード領域へのアクセスが SRAM 領域へのアクセスとほぼ同じ速度になります。SRAM からコードを実行することは可能ですが、パフォーマンスの大幅な向上は必ずしも実現されない場合があります。

2.2.2 PSoC™ 5LP アドレスマップ

Figure 4 に示すように、PSoC™ 5LP / Cortex®-M3 アーキテクチャは、PSoC™ 4 よりも複雑で、多くの機能を備えています。Cortex®-M3 には、1 つではなく 3 つのバスがあります。

- I (命令) バスおよび D (データ) バス: コード領域からの命令の読み出しおよびデータへのアクセス用.PSoC™ 5LP では、I および D バスは、コード領域にアクセスするための単一の C (コード) バスに多重化されています。
- S (システム) バス: 他の領域からの命令の読み出しとデータへのアクセス用

C バスと S バスが分離されているため、Cortex®-M3 はコード領域と他の領域の同時並列アクセスを実行して、より効率的な操作を行うことができます。

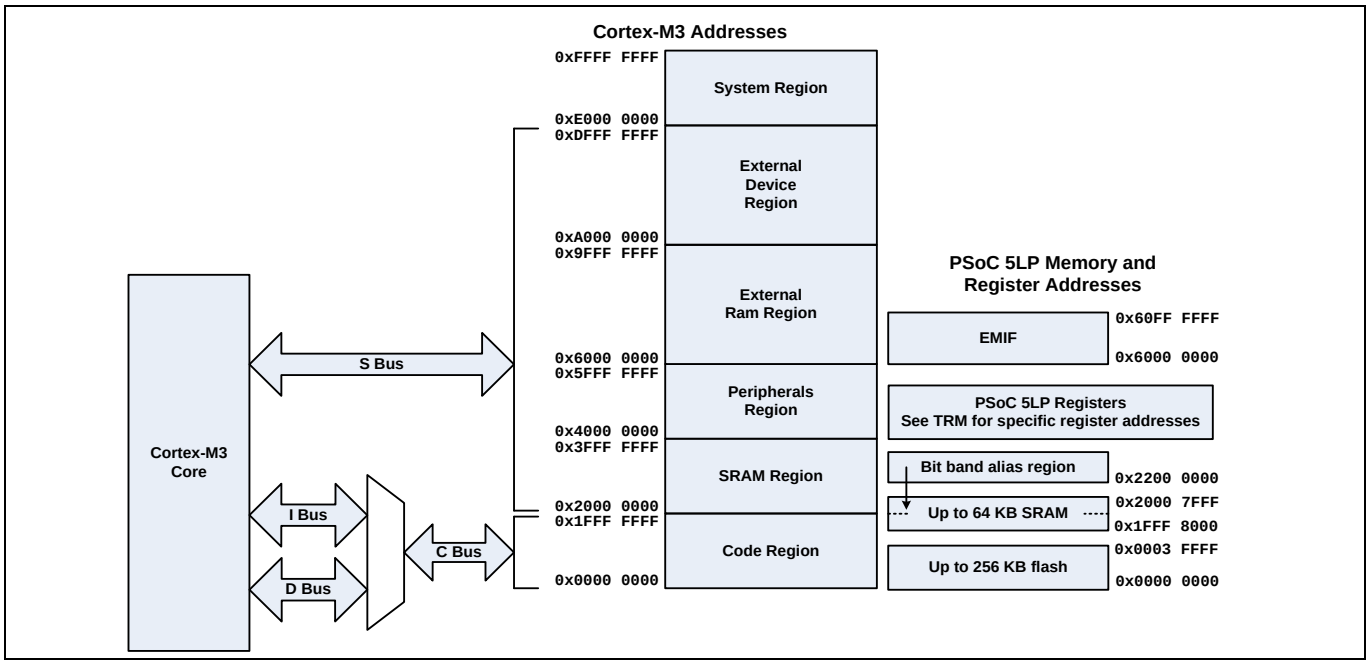


Figure 4 PSoC™ 5LP アドレスマップ

PSoC™ 5LP のメモリとレジスタは次のようにアドレス指定されています。

- PSoC™ 5LP フラッシュは、Cortex®コード領域のアドレス 0 から始まります。フラッシュキャッシュが含まれています。詳細については、PSoC™ 5LP デバイスのデータシートを参照してください。
- PSoC™ 5LP SRAM は論理的に半分に分割され、アドレス 0x20000000 を中心とします。例えば、64 KB の SRAM を備えたデバイスでは、SRAM の半分である 32 KB が 0x20000000 未満でアドレス指定され、残りの半分が 0x20000000 を超えてアドレス指定されます。したがって、SRAM アドレスの範囲は 0x1FFF8000～0x20007FFF です。32 KB SRAM のデバイスのアドレスは、0x1FFFC000～0x20003FFF の範囲です。

コード SRAM と呼ばれる SRAM の下半分は、Cortex®コード領域にあります。上位 SRAM と呼ばれる上半分は、Cortex® SRAM 領域にあります。Figure 4 に示すように、2 つの半分は異なるバスでアクセスされます。SRAM の半分をコード領域に配置すると、コードとデータを配置して、アクセスを高速化できます。詳細については、コードと変数の配置を参照してください。

Note: PSoC™5LP 内では、SRAM アクセスは通常フラッシュアクセスよりも高速ですが、Cortex®命令パイプラインとフラッシュキャッシュの組合せにより、フラッシュアクセスは SRAM アク

セスとほぼ同じ速度になります。コード SRAM または上位 SRAM からコードを実行することは可能ですが、パフォーマンスの大幅な向上は必ずしも実現されない場合があります。

上部 SRAM のみが Cortex®-M3 ビットバンド領域にあることに注意してください。

- PSoC™ 5LP レジスタは、Cortex ペリフェラル領域の 0x40000000 からアドレス指定されます。特定のレジスタアドレスについては、PSoC™ 5LP テクニカルリファレンスマニュアル (TRM) を参照してください。
- PSoC™ 5LP 外部メモリインターフェース (EMIF) アドレスは、Cortex®外部 RAM 領域の 0x60000000 から始まります。PSoC™ 5LP EMIF の詳細については、デバイスデータシート、リファレンスマニュアル、または External Memory Interface (EMIF) Component データシートを参照してください。

8 ビットまたは 16 ビットに設定できる EMIF を除いて、すべてのメモリアクセスは 32 ビットです。

PSoC™ 5LP には、ダイレクトメモリアクセス (DMA) コントローラも含まれています。バスアービトレーション技術を使用して、デュアルバスマスターとして CPU と帯域幅を共有します。詳細については、AN52705 - PSoC™ 3 and PSoC™ 5LP - Getting started with DMA を参照してください。このアプリケーションノートの DMA アドレスも参照してください。

2.3 割込み

両方の Cortex® CPU は、迅速かつ確定的な割込み処理のための高度なサポートを提供します。詳細については、デバイスデータシートまたは TRM、PSoC™ Creator 割込みコンポーネントデータシート、または AN54460 – PSoC™ 3 and PSoC™ 5LP interrupts/AN90799 – PSoC™ 4 interrupts を参照してください。

3 コンパイラの一般的なトピック

GCC, MDK, および IAR コンパイラの詳細な調査を開始する前に、いくつかの一般的なコンパイラトピックについて検討します。

Note: このアプリケーションノートに示されているすべての C サンプルコードは、PSoC ™Creator 4.3: GCC 5.4, ModusToolbox: GCC 10.3.1, Keil マイクロコントローラ開発キット (MDK) バージョン 5.03, および IAR 8.50.4 でサポートされている C コンパイラで使用するように設計されています。GCC 5.4 コンパイラは、PSoC ™Creator インストールに無料で含まれます。MDK を購入する必要がありますが、オブジェクトサイズが制限された評価版である MDK-Lite は、Keil から無料で入手できます。IAR も購入する必要がありますが、IAR Systems から機能が制限された評価版もいくつか無料で入手できます。特に明記されていない限り、コンパイラの最適化はオフになっています。

このアプリケーションノートの C サンプルコードはすべて、コンパイラ固有の拡張機能を除いて、ANSI 標準 C を使用しています。

3.1 コンパイラの定義済みマクロ

できるだけ多くの異なるコンパイラ間で直接移植できる C コードを記述することを推奨します。ただし、これが不可能な場合があり、複数のコンパイラで使用するためには、同じコードの複数のバージョンを記述しなければいけません。これを行う必要がある場合は、ほとんどのコンパイラで提供されている事前定義マクロを使用して、使用されているコンパイラを識別できます。これにより、使用しているコンパイラのコードのみをコンパイルできます。次に例を示します。

```
#if defined(MY_COMPILER_MACRO)
  /* put your compiler-unique code here */
#endif
```

この手法を ModusToolbox™プロジェクトに適用するには、MDK, GCC, IAR コンパイラにそれぞれ含まれている以下のマクロを使用します。ModusToolbox™ software プロジェクトの Makefile で Toolchain フィールドを変更して選択を反映させます。また、CY_COMPILER_PATH フィールドがコンパイラの.bin ファイルディレクトリを指していることを確認します。

MDK, GCC, および IAR コンパイラの場合、それぞれ__ARMCC_VERSION, __GNUC__, __ICCARM__が定義されているかどうかをチェックするだけで、そのコンパイラが使用されていることを示します。実際の値、つまりコンパイラのバージョンを気にする必要はありません。

```
#if defined(__ARMCC_VERSION)
/* put your MDK unique code here */
#elif defined (__GNUC__)
/* put your GCC unique code here */
#elif defined (__ICCARM__)
/* put your IAR unique code here */
#endif
```

この手法を PSoC™ Creator プロジェクトに適用するには、上記と同じマクロを使用できます。

3.2 コンパイラ出力の表示

さまざまな条件下でのコンパイラの動作を理解するには、出力アセンブラコードを確認する必要があります。ModusToolbox™ software でこれを行うには 2 つの方法があります。

1. Figure 5 に示すように、デバッガの逆アセンブリウィンドウで生成されたアセンブリを確認できます。
2. Makefile の CFLAGS または CXXFLAGS フィールドを変更して、コンパイラ コマンドにリスト ファイル生成オプションを含めることができます。これらのフラグは、使用するツールチェーンによって異なります。デフォルトの GCC コンパイラの場合、これらのオプションの例は次のようになります。

```
CFLAGS=-Wa,-adhlns=$@.lst
```

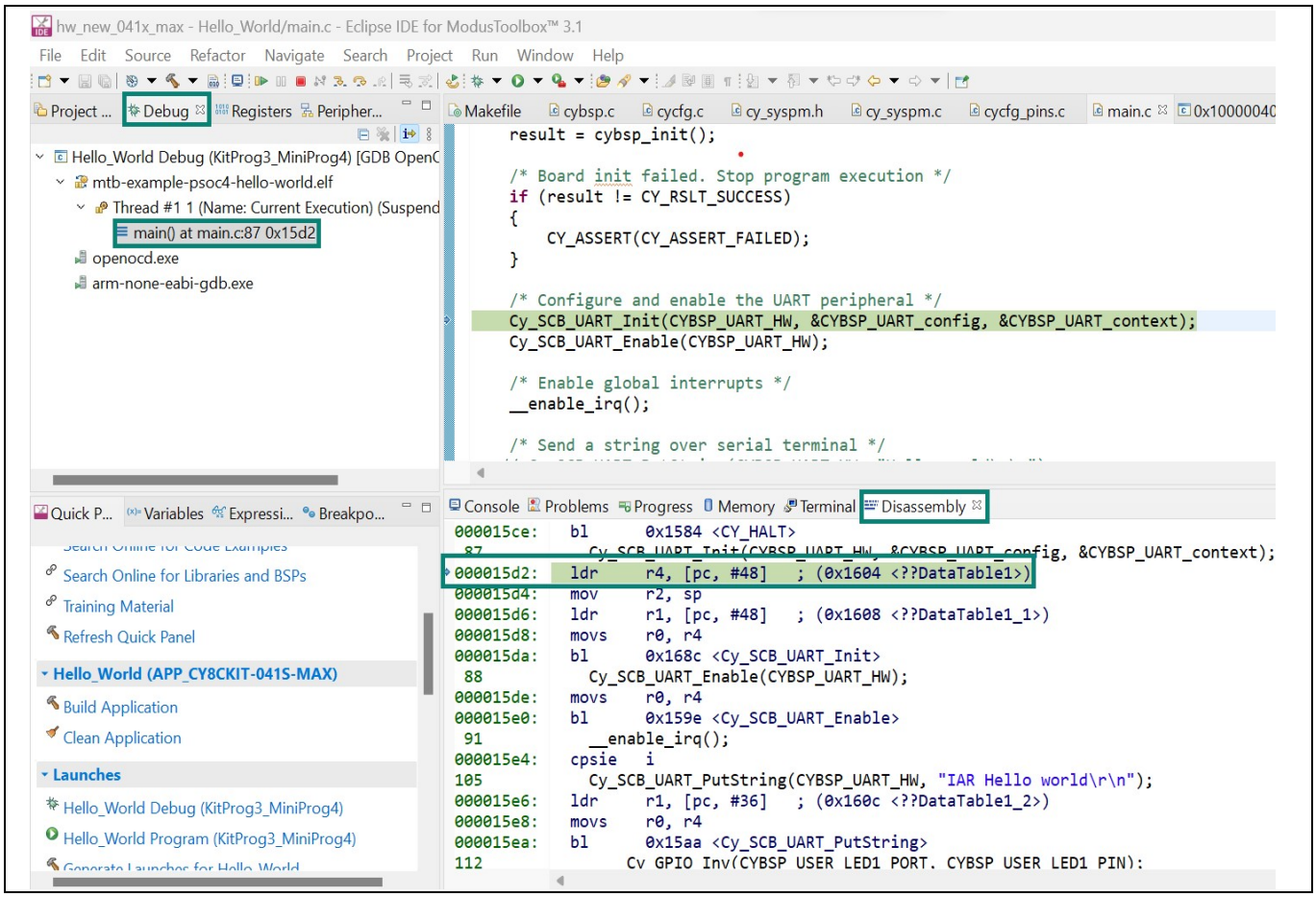


Figure 5 ModusToolbox™ソフトウェアの逆アセンブリウィンドウ用の Eclipse IDE

PSoC™ Creator で出力アセンブラ コードを表示するには、次の 2 つのオプションがあります。

1. Figur 6 に示すように、コンパイルされた C ファイル (filename.lst) に対応するリスト ファイルを開きます。デフォルトの PSoC™ Creator プロジェクト ビルド設定では、リスト ファイルを作成するように設定されています。**Project** > **Build Settings** > **Compiler** > **General** を選択します。
2. デバッガの逆アセンブリ ウィンドウを使用します (**Debug** > **Windows** > **Disassembly**)。そのウィンドウを右クリックすると、混合ソースとアセンブラを表示するオプションが表示されます。
もちろん、この方法には、デバッガを使用する前に、動作するターゲット ハードウェアと、正しくビルドされる PSoC™ Creator プロジェクトが必要であるという欠点があります。

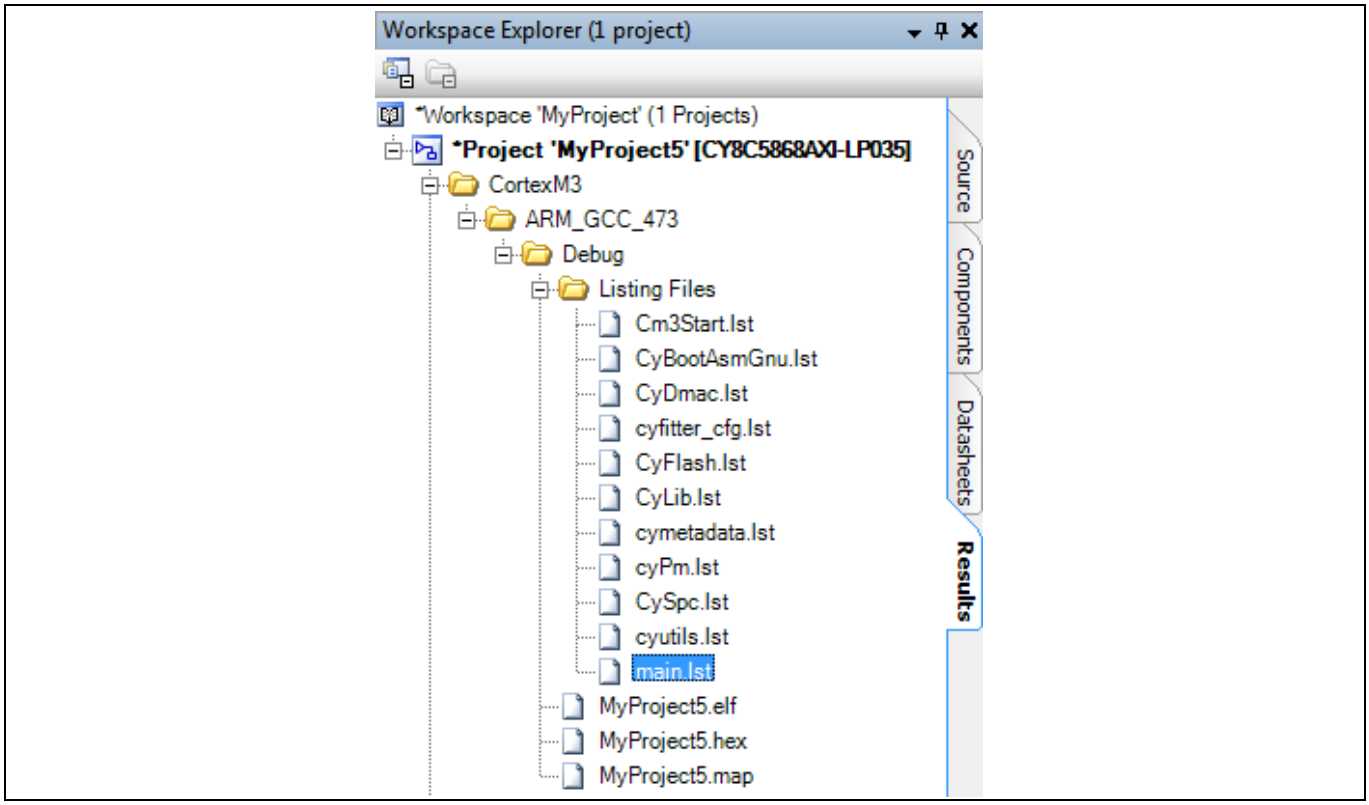


Figure 6 PSoC™ Creator でのファイルの一覧表示

Note: MDK の無料評価版である MDK-Lite では、.lst ファイルにアセンブラが含まれていないため、出力アセンブラ コードを表示するには方法2 を使用する必要があります。

3.3 コンパイラの最適化

最適化オプションをオンにすると、コンパイラは、コンパイル時間とプログラムをデバッグする機能を犠牲にして、C コードのパフォーマンスやサイズを改善しようとします。

ModusToolbox™ software では、Makefile を使用してプロジェクト全体のコンパイラ最適化を設定できます。`CONFIG` フィールドを `Custom` に設定し、`CFLAGS (C)`または `CXXFLAGS (C++)`フィールドにコンパイラ最適化オプションを追加すると、最適化が設定されます。IAR および MDK では、`#pragma` を使用して個々の関数の最適化を設定できます。詳細については、C ドキュメントを参照してください。

PSoC™ Creator では、**Project** > **Build Settings** > **Compiler** > **Optimization** でプロジェクト全体のコンパイラ最適化を設定できます。PSoC™ Creator が GCC に対して提供する最適化は、**none**, **debug**, **minimal**, **high**, **speed**, および **size** です。MDK の場合、PSoC™ Creator は、**none**, **size**, および **speed** の最適化を提供します。これは、Keil 8051 コンパイラに提供される 11 レベルの最適化とは異なります。IAR は同様のレベルの最適化を提供し、**none**, **low**, **medium**, **high-balanced**, **high-speed**, および **high-size** を提供します。

ビルド設定ダイアログで選択した最適化オプションは、プロジェクト内のすべての C ファイルに適用されます。個々の C ファイルに最適化を適用することもできます。ワークスペースエクスプローラーウィンドウでファイルを右クリックし、**Build Settings** を選択します。GCC では、特定の関数属性を使用する場合を除いて、個々の関数に最適化レベルを設定できません。

最適化を使用して C コードをコンパイルした後は、アセンブラ出力を注意深く確認し、それが期待どおりに動作していることを確認することを強く推奨します。デバッガでアセンブラコードをステップ実行

することも役立ちます。1 つの良い方法は、最適化なしで C コードを動作するようにしてから、最適化を使用して再構築し、テストを繰り返すことです。これは、PSoC™ Creator プロジェクトのビルド設定または ModusToolbox™ software の Makefile ビルド設定のデバッグおよびリリース構成を使用して実行できます。

さまざまな最適化オプションの具体的な例については、付録: コンパイラ出力の詳細を参照してください。

3.4 属性

GCC と MDK の両方でサポートされている C 言語の拡張機能は、関数、変数、および構造体型に属性を適用することです。属性は、例えば、以下を制御するために使用できます。

- 特定の機能の最適化
- 構造体がメモリを占有する方法 (パックおよびアンパックの構造体を参照してください)
- メモリ内の関数と変数の場所 (コードと変数の配置を参照してください)

構文は (「属性」の前後に 2 つの下線文字) です。

```
__attribute__ ((<attribute-list>))
```

IAR は、この構文を使用して、GCC スタイルの属性の限定された選択をサポートします。

特定の属性については、このアプリケーション ノートの以降のセクションで詳しく説明します。詳細については、C ドキュメントを参照してください。

4 変数へのアクセス

コンパイラの出力を確認するとき、最初に調べる領域の 1 つは、変数 (および配列と構造体) の読み出しと書き込みの方法です。アセンブリ言語の出力では、GCC, MDK, および IAR コンパイラの両方が、以下にアクセスするための特定の手法を実装しています。

- グローバル変数と静的変数
- 自動 (ローカル) 変数
- 関数の引数と関数の結果

これらのそれぞれがどのように行われるかを見てみましょう。

4.1 グローバル変数と静的変数

両方の Cortex® CPU で使用される Thumb-2 アセンブリ言語は、通常、32 ビットの即値をレジスタにロードすることをサポートしていません。(例外があります。例えば、小さな即値をロードして符号拡張することができます。) これにより、グローバル変数または静的変数のアドレスをロードしたり、一般にアドレスをロードしたりすることが難しくなります。Table 1 に、以下の C コードの例でこの問題を処理する 2 つの方法を示します。

```
/* loading a global variable */
uint32 myVar;
. . .
myVar = 7;
```

Table 1 CPU レジスタにアドレスをロードする方法の例

方法 1: 2 つの 16 ビット直接ロード	方法 2: PC 相対ロード

`/* rx = address of myVar load the lower and upper halves of the address */`
`movw rx, #<LS word> /* 32-bit instruction */`
`movt rx, #<MS word> /* 32-bit instruction */`
`. . .`
`movs ry, #7`
`str ry, [rx, #0] /* myVar = 7 */`
	`/* rx = address of myVar load the value stored in flash, below */`
	`ldr rx, [pc, #<offset>] /* 16-bit instruction */`
	`. . .`
	`movs ry, #7`
	`str ry, [rx, #0] /* myVar = 7 */`
	`. . .`
	`/* address value stored after the end of the function */`
	`.word <address of myVar> /* 32-bit value */`

一般的に、方法 2 はサイズが制限されたアプリケーションに適しています。これは、使用するバイト数が 2 つ少ないためです (1 つの 16 ビットワード)。ただし、方法 1 は、Cortex®の命令パイプラインが原因で、より高速に実行される可能性があります。PSoC™の場合、命令の実行速度はフラッシュキャッシュ (PSoC™ 5LP) またはアクセラレータ (PSoC™ 4) にも依存するため、必ずしも確定的ではないことに注意してください。詳細については、アドレスマップを参照してください。

異なるコンパイラ最適化は、これら 2 つの方法のどちらかを実装します。詳細な例については、付録: コンパイラ出力の詳細を参照してください。

グローバル変数の使用を最小限に抑えることがコーディングの良い方法です。PSoC™ Cortex® CPU でこれを行うと、メモリおよび PSoC™レジスタアドレスのロードを減らすことにより、コードサイズを減らすように動作する場合があります。

4.2 自動変数

C では、自動変数は関数内 (ローカル) で定義される変数です。Table 2 に示すように、関数のサイズと複雑さ、およびコンパイラ最適化設定に応じて、自動変数が CPU レジスタに割り当てられるか、またはスタックに保存されます。

Table 2 自動変数を使用する方法の例

C コード	アセンブラコード

`void MyFunc(void)`
`{`
`  uint8 i = 3;`
`  uint8 j = 10;`
`  . . .`
`}`
	`/* use rx as i, do NOT save it on the stack */`
	`movs  rx, #3  /* initialize i */`
	
	`/* store j on the stack, use ry to temporarily hold the initial value */`
	`movs  ry, #10  /* initialize j */`
	`strb  ry, [sp, #<offset>] /* on the stack */`

サイズと速度の両方の最適化により、自動変数のスタック使用量が減少する傾向があります。例については、付録: コンパイラ出力の詳細を参照してください。

4.3 関数の引数と結果

Procedure call standard for Arm® architecture は、引数を関数に渡すためにレジスタ R0～R3 を割り当て、関数の結果を渡すために R0 を割り当てます。引数の数が 4 より大きい場合、最初の 4 つの引数はレジスタに入れられ、残りはスタックにプッシュされます。

関数内では、引数はそれぞれのレジスタで維持されるか、他のレジスタに転送されるか、スタックに保存されます。関数の自動変数もスタックに格納される場合があるため (Table 2)、スタックの管理が複雑になる可能性があります。この複雑さを処理するために、Table 3 の例に示すように、**prolog** と **epilog** として知られる 2 つの命令シーケンスをコンパイル済み関数に含めることができます。

Table 3　Prolog および Epilog 命令を含む関数の例

C コード	アセンブラコード

/* Function with 6 arguments and a return value */
uint32 MyFunc(uint32 a, uint32 b, uint32 c,
uint32 d, uint32 e, uint32 f)
{
return a + b + c + d + e + f;
}
	/* function prolog */
	push {r7} /* make room on the stack */
	sub sp, sp, #20 /* for the arguments */
	add r7, sp, #0 /* use r7 as a base pointer */
	str r0, [r7, #12] /* save the arguments */
	str r1, [r7, #8] /* on the stack */
	str r2, [r7, #4]
	str r3, [r7, #0]
	
	/* function body */
	ldr r2, [r7, #12] /* build the sum */
	ldr r3, [r7, #8] /* in r2 and r3 */
	adds r2, r2, r3
	ldr r3, [r7, #4]
	adds r2, r2, r3
	ldr r3, [r7, #0]
	adds r2, r2, r3
	ldr r3, [r7, #24] /* argument e */
	adds r2, r2, r3
	ldr r3, [r7, #28] /* argument f */
	adds r3, r2, r3
	mov r0, r3 /* return value in r0 */
	
	/* function epilog */
	add r7, r7, #20 /* restore stack and r7 */
	mov sp, r7
	pop {r7}
	bx lr /* return */

コードサイズを最小化し、速度を最大化するには、関数の引数の数を 4 に制限する必要があります。

関数のサイズと複雑さに応じて、サイズの最適化は関数の prolog と epilog のコードを削減しようとします。例については、付録: コンパイラ出力の詳細を参照してください。

4.4　LDR および STR 命令

これらの命令は、メモリおよび PSoC™レジスタの読み出しおよび書き込みに使用されます。それらは非常に強力で、多くの柔軟なオプションを提供します。命令のバリアントは、バイトおよびハーフワード (16 ビット) アクセス、ゼロおよび符号拡張、および即値とレジスタオフセットをサポートします。Table 4 に示すように、オフセットオプションは、ポインタオフセットの処理や、配列や構造体のメンバーへのアクセスに特に役立ちます。

Table 4　LDR および STR 命令の使用例

C コード	アセンブラコード
/* loading an array member */ uint8 myArray[100]; . . . myArray[6] = 7;	ldr rx, [pc, #<offset>] /* rx = address of myArray */ movs ry, #7 strb ry, [rx, #6]
uint8 myArray[100]; . . . uint8 i; . . . myArray[i] = 7;	ldr rx, [pc, #<offset>] /* rx = address of myArray */ ldrb ry, [sp, #<offset>] /* ry = i (automatic variable) */ movs rz, #7 strb rz, [rx, ry]

4 変数へのアクセス

LDR および STR 命令は常にレジスタ相対であるため、LDR または STR 命令の前に、ターゲットアドレスをレジスタにロードする別の命令が常に存在することに注意してください。Table 1 を参照してください。これらの命令のバリエーションとオプションは、Cortex®-M0 (PSoC™ 4) と Cortex®-M3 (PSoC™ 5LP) では異なります。詳細については、Arm® Cortex®のドキュメントを参照してください。

5 C とアセンブラコードの混合

コードをより短く, より速く, より効率的にする最も効果的な方法の 1 つは、アセンブラでコードを記述することです。アセンブラを使用すると、CPU でサポートされているが C コンパイラでは使用されていない特殊関数命令を利用できる場合もあります。特殊機能命令を参照してください。ただし、アセンブラでのコーディングは、最小のアプリケーションを除くすべてにとって困難な作業であり、一旦コードを作成すると、保守や他のコンパイラや CPU への移植が容易ではありません。そのため、ほとんどのコードは C で記述されており、アセンブラを使用しても、いくつかの重要な機能にのみ使用されます。

アセンブラのもう 1 つの問題は、共存させる必要のある C ファイルとは別に、独自のファイルに書き込む必要があることです。これにより、コードの統合と保守が困難になる可能性があります。

これらの問題の両方の解決策は、**インラインアセンブラ**と呼ばれる C 言語の拡張機能です。インラインアセンブラでは、アセンブラコードを C ファイルに直接配置でき、別の C ステートメントとして扱われます。これにより、効率を上げるために必要な場所でのみアセンブラを使用できるようになり、C とアセンブラコードを簡単に組み合わせられます。GCC, MDK, および IAR コンパイラはどちらもインラインアセンブラをサポートしています。さらに、MDK は**組込みアセンブラ**と呼ばれる同様の機能をサポートしています。この機能では、関数は完全にアセンブラで記述されますが、C ファイルに含まれます。

ここでは、GCC, MDK, および IAR コンパイラで、C とアセンブラコードを組み合わせて使用する方法を示します。ここで説明する方法を効果的に使用するには、それらが基づいているレジスタキテクチャを理解することが重要です。詳細については、CPU レジスタキテクチャを参照してください。

Note: 次の例は、Cortex®-M3 のアセンブラを示します。Cortex®-M0/M0+は、Cortex®-M3 命令のより限定されたサブセットを使用します。詳細については、Arm® Cortex®のドキュメントを参照してください。

Note: ほとんどのアセンブラ命令は、Cortex®レジスタに作用します(レジスタセットを参照)。Procedure Call Standard for Arm Architecture では、これらのレジスタの一部を関数で保持する必要があります。必要に応じて、PUSH および POP 命令を使用して、レジスタをスタックに保存および復元します。

5.1 構文

インラインアセンブラの GCC の構文は次のとおりです。

```
asm("assembler instruction");
```

C コードに 1 行のアセンブラコードを追加します。例えば、以下は R0 レジスタをインクリメントします。

```
asm("ADD  r0, r0, #1"); /* R0 = R0 + 1 */
```

複数行インラインアセンブラの構文は次のとおりです。

```
asm("line 1\n"
    "line 2\n"
    "line 3\n"
    . . .
    "line n");
```

以下は例です。

5 C とアセンブラコードの混合

```
/* R0 = R0 + 1; R1 = R0 */
asm("ADD  r0, r0, #1\n"
    "MOV  r1, r0");
```

Note: キーワード asm の代わりに__asm__が使用できます。詳細については、C ドキュメントを参照してください。

Note: キーワード volatile を追加して、ステートメントがコンパイラによって最適化されないようにできます。

```
asm volatile(" ... ");
```

インラインアセンブラの MDK 構文は GCC の構文と同じですが、「asm」の前に 2 つのアンダースコア文字が付く点が異なります。

```
__asm("assembler instruction");


__asm("line 1\n"
      "line 2\n"
      "line 3\n"
      . . .
      "line n");
```

MDK 組込みアセンブラの構文は次のとおりです。

```
__asm return-type
  function-name(argument-list)
  {
    /* This is a C comment */
    instruction /* assembler comment */
    ...
    instruction
  }
```

以下は例です。

```
  __asm int DoSum(int x, int y)
  {
    ADD r0, r0, r1
    BX  lr
  }
```

IAR はキーワード__asm を使用しますが、asm も使用できます。オプション-strict を使用してコンパイルする場合、キーワード asm は使用できないことに注意してください。IAR インラインアセンブラの構文は GCC と同じです。

5.2 自動変数

GCC では、インラインアセンブラから自動 (またはローカル) 変数にアクセスするには、最初に変数に Rx レジスタを強制的に割り当てる必要があります。変数を次のように宣言します。

```
register int foo asm("r0"); /* foo occupies register R0 */
```

Note: xxxGCC は、実際には asm キーワードの C 表現オペランドの複雑な言語をサポートしています。この言語のチュートリアルは、このアプリケーションノートの範囲を超えています。詳細は C ドキュメント、特にセクション 6.41「GNU コンパイラコレクションの使用」を参照してください。

例として、2 つの自動変数‘foo’と‘bar’を定義して、それらの間で簡単な数学演算を実行します。

```
void main()
{
    register int foo asm("r0") = 5L; /*register variables can be initialized */
    register int bar asm("r1");
    bar = foo + 1; /* C code version */
    asm("ADD r1, r0, #1"); /* bar = foo + 1 */
}
```

上記の例では、C コードとインラインアセンブラは同じ操作を実行します。ただし、コンパイルされた C コード (最適化なし) は中間レジスタを使用し、その結果、2 倍のフラッシュメモリを使用して 3 倍の命令を生成します。これは、*.lst* ファイルからの抜粋です。

```
20:.\main.c       ****  bar = foo + 1;
42 0008 0346      mov  r3, r0
43 000a 03F10103  add  r3, r3, #1
44 000e 1946      mov  r1, r3

22:.\main.c      ****  asm("ADD r1, r0, #1"); /* bar = foo + 1 */
47 0010 00F10101  ADD  r1, r0, #1
```

関数のサイズと複雑さによって、コンパイラ最適化オプションを使用して中間レジスタを削除できる場合があります。

MDK を使用すると、自動 (ローカル) 変数がレジスタを占有するように強制する必要はありません。そのかわりに、変数に直接アクセスできます。

```
void main()
{
    /* no need to declare variables in registers */
    int foo = 5;
    int bar;

    bar = foo + 1; /* C code version */

    __asm("ADDS bar, foo, #1"); /* bar = foo + 1 */
}
```

5 C とアセンブラコードの混合

この例では、C コードとインラインアセンブラは同じ操作を行い、まったく同じコードを生成します。

IAR では、register キーワードは考慮されず、自動変数はレジスタの可用性に応じてレジスタ内またはスタック上に配置されます。IAR では、次の構文を使用してインラインアセンブラで変数を参照する方法が提供されています。

```
__asm("assembler instruction %output_operand, %input_operand1, …, %input_operandN"
      : "=constraint" (output_symbol)
      : "constraint1" (intput1_symbol), …, "constraintN" (inputn_symbol));
```

制約「r」を使用すると、式で汎用レジスタが使用されます。以下は例です。

```
void main()
{
    /* variables initialized without register keyword */
    int foo = 5L;
    int bar;

    /* C code version */
    bar = foo + 1;

    /* Assembly version */
    __asm("ADDS %0, %1, #1"
          : "=r" (bar)
          : "r" (foo));
}
```

この例は次のようにコンパイルされます。

```
/* variables initialized without register keyword */
int foo = 5L;
        MOVS     R1,#+5
int bar;

/* C code version */
bar = foo + 1;
        MOVS     R0,R1
  ADDS     R0,R0,#+1

/* Assembly version */
__asm("ADDS %0, %1, #1"
      : "=r" (bar)
      : "r" (foo));
  ADDS R0, R1, #1
```

この例の C コードバージョンでは、インラインアセンブリが 1 つの命令で実行する処理を 2 つの命令で実行します。他のコンパイラ最適化レベルでは異なる出力が生成され、C コードバージョンから余分な命令が削除される可能性があることに注意してください。

5.3 グローバル変数と静的変数

以前のメソッドは、グローバル変数および静的変数 (「グローバル」) でも使用できます。グローバルにアクセスする前に、変数のアドレスをレジスタにロードする必要があることに注意してください。詳細はグローバル変数と静的変数を参照してください。

GCC では、次の構文を使用してアドレスをロードします。

```
LDR  rx, =variable_name
LDR  ry, =0x1FFF9000 /* an address */
```

自動変数の代わりにグローバルを使用して前の例を繰り返しましょう。

```
int foo = 5L;
int bar;

void main()
{
    bar = foo + 1;

    /* bar = foo + 1 */
    asm("LDR r0, =foo\n"
        "LDR r1, =bar\n"
        "LDR r2, [r0]\n"
        "ADD r2, r2, #1\n"
       "STR r2, [r1]");
}
```

繰り返しますが、C コードとインラインアセンブラは同じ操作を行いますが、アドレスロード方法が異なるため (Table 1 を参照)、コンパイルされた C コード (最適化なし) は、インラインアセンブラよりも 2 つの命令を生成し、4 バイト多いメモリを使用します (Cortex®-M3)。次のデバッガの一部分に示します。

5 C とアセンブラコードの混合

```
bar = foo + 1;
    F248130C  movw   r3, #810c
    F6C173FF  movt   r3, #1fff
    681B      ldr    r3, [r3, #0]
    F1030201  add.w  r2, r3, #1
    F248132C  movw   r3, #812c
    F6C173FF  movt   r3, #1fff
    601A      str    r2, [r3, #0]

/* bar = foo + 1 */
asm("LDR r0, =foo\n"
    "LDR r1, =bar\n"
    "LDR r2, [r0]\n"
    "ADD r2, r2, #1\n"
    "STR r2, [r1]");
    4804      ldr    r0, [pc, #10]
    4905      ldr    r1, [pc, #14]
    6802      ldr    r2, [r0, #0]
    F1020201  add.w  r2, r2, #1
    600A      str    r2, [r1, #0]
```

これらの結果は、コンパイラの最適化が使用されている場合は異なる場合があります。

MDK では、グローバル変数にアクセスする方法が 2 つあります。最初の方法 (インラインアセンブラ) は、自動変数にアクセスする方法と似ています。

```
 /* bar = foo + 1 */
 __asm("ADDS bar, foo, #1");
```

ただし、アセンブラの出力はまったく異なります。

```
4805     ldr   r0, [pc, #14]
6800     ldr   r0, [r0, #0]
1C40     adds  r0, r0, #1
4905     ldr   r1, [pc, #14]
6008     str   r0, [r1, #0]
```

変数アドレスをロードするには、追加の命令が必要です。詳細については、グローバル変数と静的変数を参照してください。この場合、インラインアセンブラは実質的に疑似命令であり、実際のアセンブラ命令を 5 つ生成します。出力は C で記述した場合と同じなので、この場合、インラインアセンブラを使用する利点はありません。

埋め込みアセンブラメソッドは次のようになります。

5 C とアセンブラコードの混合

```
__asm void AddGlobals(void)
{
    extern foo
    extern bar

    LDR  r0, =foo
    LDR  r1, =bar
    LDR  r0, [r0]
    ADD  r0, r0, #1
    STR  r0, [r1]
    BX   lr
}
```

この場合、結果のコードはインラインアセンブラメソッドの場合と同じです。

IAR では、インラインアセンブラは自動変数にアクセスするのと同じ方法でグローバル変数にアクセスします。

```
/* bar = foo + 1; */
__asm("ADDS %0, %1, #1"
      : "=r" (bar)
      : "r" (foo));
```

ただし、MDK と似ていますが、出力アセンブリはまったく異なります。

```
__asm("ADDS %0, %1, #1"
      : "=r" (bar)
      : "r" (foo));

      LDR      R0,??main_0
      LDR      R0,[R0, #+0]
      ADDS     R0, R0, #1
      LDR      R1,??main_0+0x4
      STR      R0,[R1, #+0]
```

変数アドレスをロードするには、追加の命令が必要です。この場合、MDK の場合と同様に、インラインアセンブリ命令は一連の命令にコンパイルされる疑似命令として機能します。このステートメントの C バージョンは同じ命令にコンパイルされるため、インラインアセンブリを使用する利点はありません。

5.4 関数の引数

関数の引数と結果に記載されているように、Procedure Call Standard for Arm Architecture は、引数を関数に渡すためにレジスタ R0～R3 を割り当て、関数の結果を渡すために R0 を割り当てます。引数の数が 4 より大きい場合、最初の 4 つの引数はレジスタに入れられ、残りはスタックにプッシュされます。

したがって、引数の数を 4 つに制限すると、それらの引数を持つレジスタに直接アクセスするアセンブラコードを作成できます。次の例は、4 つの引数の合計を計算する関数を実装する複数の方法を示します。

```
uint32 addFunc(uint32 a, uint32 b, uint32 c, uint32 d)
{
    return a + b + c + d;
}
```

GCC:

```
uint32 addFunc(uint32 a, uint32 b, uint32 c, uint32 d)
{
    /* define return value in R0 */
    /* does not overwrite input argument 'a' in R0 */
    register uint32 rtnval asm("r0");
    /* the arguments are in registers R0 – R3 */
    asm volatile ("add r0, r0, r1\n"
                  "add r0, r0, r2\n"
                  "add r0, r0, r3");
    return rtnval; /* return value in R0 */
}
```

MDK の組込みアセンブラ:

```
__asm uint32 addFunc(uint32 a, uint32 b, uint32 c, uint32 d)
{
    /* the arguments are in registers R0 – R3 */
    ADD r0, r0, r1
    ADD r0, r0, r2
    ADD r0, r0, r3

    /* return value in R0 */
    BX  lr
}
```

5 C とアセンブラコードの混合

IAR:

```
uint32 addFunc(uint32 a, uint32 b, uint32 c, uint32 d)
{
    int32 rtnval;

    /* the arguments are in registers R0 – R3 */
    __asm volatile ("ADDS R0, R0, R1\n"
                    "ADDS R0, R0, R2\n"
                    "ADDS %0, R0, R3"
                    : "=r" (rtnval) : );

    return rtnval; /* Return value in R0 */
}
```

6 特殊機能命令

一部の CPU には、C コンパイラでは通常使用されない特殊機能命令があります。これらの命令には、C 組込み関数でアクセスできますが、場合によってはアセンブラを使用してのみアクセスできます。ここでは、C 組込み関数と、C とアセンブラの混合コードを使用して、これらの命令に簡単にアクセスする方法について説明します。

例として PSoC™ 5 Cortex®-M3 飽和命令を見てみましょう。その他の特殊命令機能の説明については、C ドキュメントを参照してください。

6.1 サチュレーション命令

Figure 7 に示すように、飽和は、信号が増幅される場合など信号処理で一般的に使用されます。16 ビット ADC を使用していて、12 LS ビットだけに関心があるとします。増幅後、未使用の MS ビットを単に削除して値を調整すると、オーバーフローによって結果の信号が著しく歪む可能性があります。飽和によりオーバーフローが回避され、歪みが軽減されます。

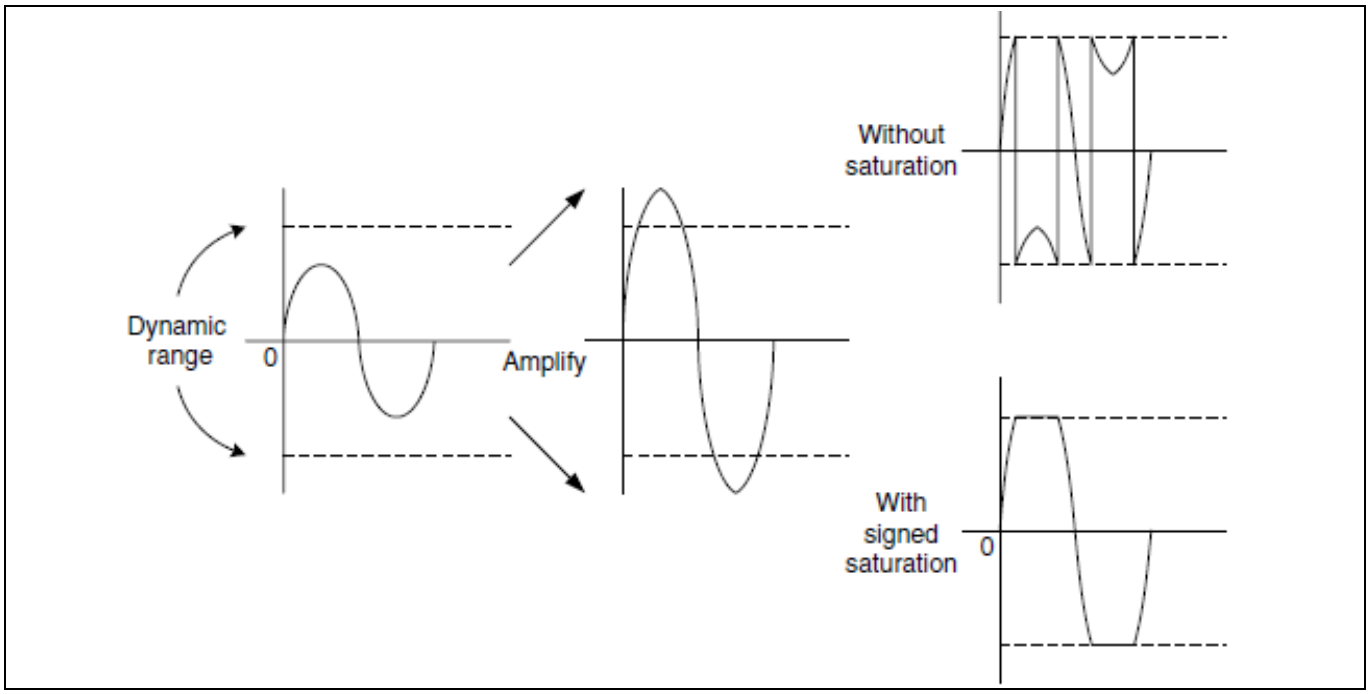


Figure 7 飽和演算

C では多重比較と `if-else` ステートメントを使用して飽和処理を実行できますが、Cortex®-M3 には、処理をはるかに効率的にする 2 つのアセンブラ命令 (それぞれ符号付きと符号なしの SSAT と USAT) があります。これらの命令は次のように動作します。

6.1.1 SSAT 命令

SSAT 命令は、符号付きの範囲 $-2^{n-1} \le x \le 2^{n-1}-1$ に飽和します。

- 飽和させる値が-2^{n-1} より小さい場合、返される結果は-2^{n-1} です。
- 飽和させる値が $2^{n-1}-1$ より大きい場合、返される結果は $2^{n-1}-1$ です。
- それ以外の場合、返される結果は飽和する値と同じです。

6.1.2 USAT 命令

USAT 命令は、符号なしの範囲 $0 \le x \le 2^{n-1}$ まで飽和します。

- 飽和させる値が 0 未満の場合、返される結果は 0 です。
- 飽和させる値が 2^{n-1} より大きい場合、返される結果は 2^{n-1} です。
- それ以外の場合、返される結果は飽和する値と同じです

6.1.3 構文

1op Rd, #n, Rm

ここでは以下のとおりです。

- op は次のいずれかです。
  - SSAT は符号付きの値を符号付きの範囲に飽和させます
  - USAT は符号付きの値を符号なしの範囲に飽和させます
- Rd はディスティネーションレジスタです。
- n は、飽和させるビット位置を指定します。
  - SSAT の場合、n の範囲は 1～32 です。
  - USAT の場合、n の範囲は 0～31 です。
- Rm は、飽和させる値を含むレジスタです。

Note: SSAT および USAT 命令は、入力レジスタの 32 ビット値で動作します。対応する C 変数は、`int`, `int32`, または `uint32` 型である必要があります。サチュレーション命令を実行する前に、符号拡張が必要な場合があります。

6.2 組込み関数

C では、組込み関数は関数呼び出しのように見えますが、コンパイル時に 1 つ以上のアセンブラ命令の特定のシーケンスに置き換えられます。Arm® Cortex® Microcontroller Software Interface Standard (CMSIS) ライブラリには、ほとんどの Cortex®特殊関数アセンブラ命令の組込み関数のセットが含まれます。PSoC™ Creator プロジェクトがビルドされた後、これらの関数はワークスペースエクスプローラーウィンドウのフォルダ Generated Source > PSoCx > cyboot > *core_cmInstr.h* にあります。ModusToolbox™ ソフトウェアでは、関数は Libs > PSoCxPDL > cmsis> include > *cmsis_<toolchain>.h* にあります。

飽和の組込み関数は次のようになります。

```
__SSAT(ARG1, ARG2)
__USAT(ARG1, ARG2)
```

ここで、ARG1 は飽和する入力値、ARG2 は飽和するビット位置です。次のように関数を呼び出します。

```
int data_unsat = -1L;
int data_sat = __USAT(data_unsat, 8);
```

この例では、`data_unsat` を符号なしの 8 ビットに飽和させます。`data_unsat` の値が 255 (0xFF) を超える場合、結果は 255 (0xFF) に飽和され、`data_sat` に格納されます。`data_unsat` の値が負の場合、0 が `data_sat` に格納されます。

6.3 アセンブラ

Table 5 に示すように、C とアセンブラコードの混合で説明されている手法を使用して、特殊関数命令を挿入することもできます。

Table 5 C とアセンブラの混合での飽和命令の使用

例	C とアセンブラの混合コード

GCC Example	`void main()`
	`{`
	`  register int data_unsat asm("r0");`
	`  register int data_sat asm("r3");`
	
	`  asm("ssat r3, 8, r0");`
	`  . . .`
	`}`

MDK Example	`void main()`
	`{`
	`  int data_unsat;`
	`  int data_sat;`
	
	`  __asm("ssat data_sat, 8, data_unsat");`
	`  . . .`
	`}`

IAR Example	`void main()`
	`{`
	`  int data_unsat;`
	`  int data_sat;`
	
	`  __asm("SSAT %0, #8, %1");`
	`  : "=r" (data_sat);`
	`  : "r" (data_unsat));`
	`}`

この例では、data_unsat を 8 ビットの符号付きに飽和させます。8 ビットの符号付き飽和では、値の範囲は-128～+127 になります。したがって、値が-128 より小さい場合、結果は-128 になり、値が+127 より大きい場合、結果は+127 になります。したがって、結果は正方向では 0x7F に飽和し、負方向では 0x80 に飽和します。

飽和命令を使用して、必要な飽和ビット数に値を飽和させることができます。USAT はシングルエンドモードで設定された ADC で使用でき、SSAT は差動モードで設定された ADC で使用できます。

7 パックおよびアンパックの構造体

ほとんどの組込みシステムでは、データはバイト単位で、例えば UART または I2C ポートを使用して送信されます (SPI プロトコルは例外です。詳細については、PSoC™ Creator Serial Peripheral Interface (SPI) コンポーネント データシートを参照してください)。8 ビット CPU を使用すると、複雑なデータ構造体をバイト単位で送受信でき、結果は元のデータと完全に一致します。しかしながら、より大きな CPU (16 ビット、32 ビットなど) では、これは必ずしも当てはまりません。PSoC™ 4 Cortex®-M0 および PSoC™ 5LP Cortex®-M3 CPU を例として使用して、これがそうである理由を詳しく調べてみましょう。

PSoC™の 32 ビット Cortex® CPU は 32 ビットワードとしてメモリにアクセスするため、データが 32 ビット境界に格納されている場合、つまり 2 つの LS アドレスビットがゼロである場合に最も効率的に機能します。例えば、16 ビットまたは 32 ビットの変数が奇数アドレスから保存され、アドレスの LS ビットが 1 の場合、それを読み出すには 2 回の 32 ビットメモリ読み出しが必要で、それを書き込むには 2 回の読み出し-変更-書き込みサイクルが必要です。これは実行速度に大きな影響を与える可能性があります。

残念ながら、C 言語では、単語が奇数境界にある構造を簡単に作成できます。次の例を検討してください。

```
struct myStruct
{
    uint8  m1; /* stored on a 32-bit boundary, address = ...xx00 */
    uint32 m2; /* stored on an 8-bit boundary, address = ...xx01 */
}
```

上記は、構造体がアドレス境界の考慮事項に関係なくバイト単位でメモリに配置されるため、**パック**構造体と呼ばれます。8 ビット CPU のコンパイラは通常、パック構造体を生成します。デフォルトでは、Cortex® CPU, GCC, および MDK コンパイラは、構造体を**アンパック**形式で保存します。アドレスは構造体メンバーのサイズによって決定されます。例えば以下のとおりです。

```
struct myUnpackedStruct
{
    uint8  m1; /* stored on a 32-bit boundary, address = ...xx00 */
    /* 3 unused filler bytes */
    uint32 m2; /* stored on a 32-bit boundary, address = ...yy00 */
}
```

アンパック構造体はより効率的にアクセスできますが、サイズが大きくなるため、SRAM が限られているデバイスでは問題になる場合があります。ただし、アンパック構造体がバイト単位で送信され、受信側がバイトをパック構造体に保存する場合など、より深刻な問題が発生する可能性があります。データが破損し、検出が困難なシステムレベルの欠陥が発生します。

コードでこの問題を修正する方法はいくつかあります。最も簡単なのは、単純に構造体メンバーの宣言の順序を最適化することです。例えば、先の構造体を次のように並べ替えることができます。

```
struct myStruct
{
    uint32 m2; /* stored on a 32-bit boundary, address = ...xx00 */
    uint8  m1; /* stored on a 32-bit boundary, address = ...yy00 */
}
```

これで構造体がパックされ、メンバーの各アドレスが 32 ビット境界にあります。

7.1.1 コンパイラの考慮事項

GCC, MDK, および IAR コンパイラの場合、デフォルトでは、次の規則に従って構造体がアンパックされます。

- char または uint8 (1 バイト) は 1 バイト境界で整列されます
- short または uint16 (2 バイト) は 2 バイト境界で整列されます。LS アドレスビットは 0 です。
- long または uint32 (4 バイト) は 4 バイト境界で整列されます。2 つの LS アドレスビットは 00 です。
- float (4 バイト) は 4 バイト境界で整列されます。2 つの LS アドレスビットは 00 です。
- char *、int * (4 バイト) などのポインタは 4 バイト境界で整列されます。2 つの LS アドレスビットは 00 です。

次の構文を使用して、構造体を強制的にパックできます。

Table 6 構文

GCC:	MDK:	IAR:

`struct myStruct`
`{`
`  . . .`
`} __attribute__ ((packed));`
	`__packed struct myStruct`
	`{`
	`  . . .`
	`};`
		`__packed struct myStruct`
		`{`
		`  . . .`
		`};`

Note: packed ステートメントは、構造体定義でのみ使用することを推奨します。実際の構造体変数の宣言や typedef 宣言では使用しないでください。

8 コンパイラライブラリ

GCC, MDK, および IAR コンパイラの場合、C 標準ライブラリ関数呼び出しを同等の C ステートメントに置き換えると、メモリ使用量を大幅に削減できます。例えば、次の C フラグメントを検討してください。

```
#include <math.h>
uint32 a, b;
a = 5;
b = pow(a,3);
```

Table 7 は、PSoC™ 5LP および PSoC™ 4 の両方のコンパイラのフラッシュおよび SRAM メモリ消費量を示します。

Table 7 pow( )関数呼び出しによるメモリ消費

	GCC	MDK	IAR
PSoC™ 5LP			
Flash	8939	7696	1498
SRAM	405	196	2241
PSoC™ 4			
Flash	14374	7700	1488
SRAM	364	312	1160

Table 8 に示すように、pow( )ライブラリ関数の呼び出しを次の同等のコードで置き換えると、使用するメモリが大幅に少なくなります。

```
b = a * a * a;
```

Table 8 pow( )関数呼び出しを使用しないメモリ消費

	GCC	MDK	IAR
PSoC™ 5LP			
Flash	1582 (-82.3%)	1444 (-81.2%)	982 (-34.4%)
SRAM	301 (-25.7%)	200 (-32.4%)	2241
PSoC™ 4			
Flash	1198 (-91.7%)	1004 (-87.0%)	992 (-33.3%)
SRAM	252 (-30.8%)	216 (-30.8%)	1160

サイズが小さくなる理由は、ANSI C の定義により、pow( )関数が double 型の引数を取り、double 型を返すためです。この関数を整数で呼び出すと、関数呼び出しの前後で自動的に適切な型にキャストされますが、実装には多くのコードが必要です。

PSoC™ Creator 4.3 では、GCC ライブラリの選択が利用可能です (newlib および newlib-nano)。newlib-nano ライブラリは、メモリ使用量を削減するために、標準 C ライブラリ関数から使用頻度の低い機能をいくつか削除します。

Note: newlib-nano から削除された機能の1 つは、printf() での浮動小数点サポートです。これは、浮動小数点値を表示する場合に問題を引き起こす可能性があります。例えば、次のコードフラグメントを考えてみます。

```
char My_String[30];
float My_Float = 3.14159;
sprintf(My_String, "Value of pi is: %.2f to 2dp", My_Float);
```

newlib-nano では、My_String に文字列"Value of pi is: %.2f to 2dp"が作成されます。期待値 3.14 は含まれていません。2 つの可能な回避策があります。

1. Figure 8 に示すように、newlib-nano で浮動小数点フォーマットのサポートを有効にします。この機能はデフォルトでは無効になっています。有効にすると、フラッシュの使用量が 10K～15K バイト増加します。

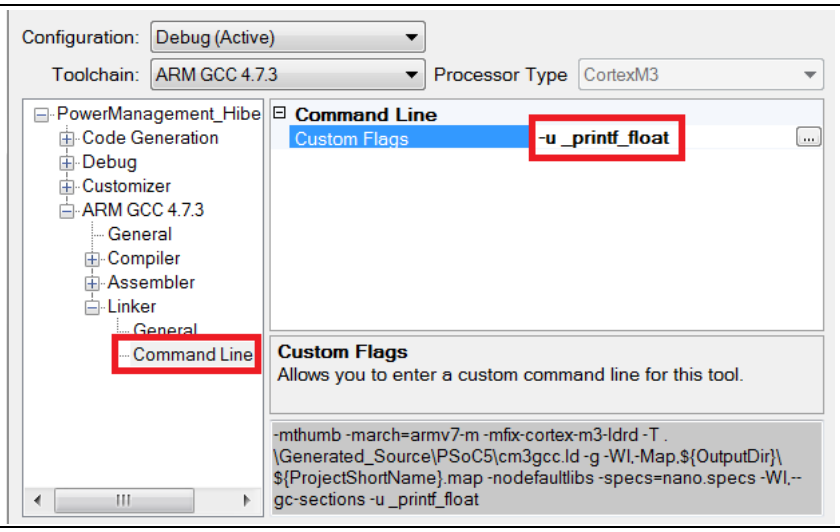


Figure 8 newlib-nano で浮動小数点フォーマットのサポートを有効にする

2. Figure 9 に示すように、ライブラリをフル機能の newlib に変更します。**Use Default Libraries** オプションも False に設定する必要があることに注意してください。デフォルトでは newlib-nano を使用します。newlib に変更すると、フラッシュの使用量が 25K～30K バイト増加し、SRAM の使用量が約 2K バイト増加します。

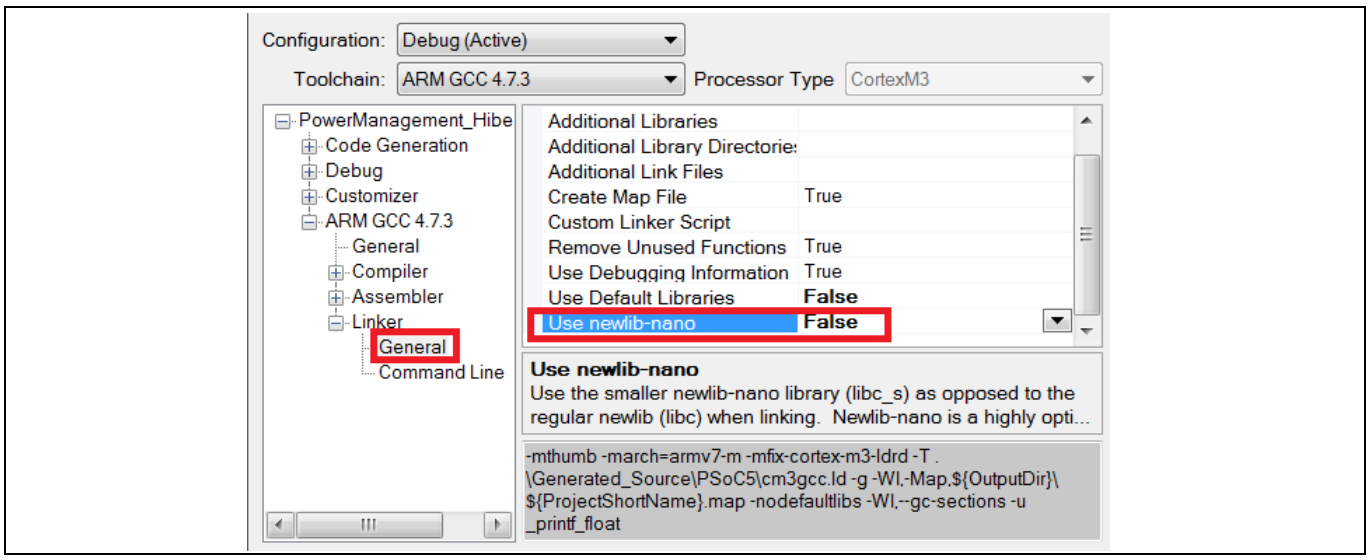


Figure 9 newlib-nano を無効にする

MDK は、Arm® C Micro-library と呼ばれる機能制限ライブラリも提供しています。詳細については、C ドキュメントを参照してください。

9 コードと変数の配置

ここでは、C コードと変数をメモリ内のカスタムの場所に配置する方法を示します。これを行う理由はいくつかあります。例については、カスタムの場所を定義します。を参照してください。

ここで説明する方法を効果的に使用するには、それらが基づいている CPU アーキテクチャを理解することが重要です。詳細については、アドレスマップを参照してください。

9.1 リンカスクリプトファイル

コードと変数をカスタムの場所に配置するには、リンカスクリプトファイルを変更する方法を知っている必要があります。ここでは、リンカスクリプトファイルが PSoC™ 4 および PSoC™ 5LP でのメモリの使用を制御する方法の基本を示します。詳細は、GCC または C ドキュメントを参照してください。

ModusToolbox™ では、デフォルトのリンカスクリプトファイルは、Figure 10 に示すように、*<Project Directory>\bsps\TARGET_APP_<Kit Name>\TOOLCHAIN_<Toolchain Name>* にあります。`LINKER_SCRIPT` Makefile 変数で使用するカスタムリンカファイルを指定することもできます。

Note: ModusToolbox™ ソフトウェア3.1 の場合、アプリケーションでカスタムリンカファイルを使用するには、Makefile の LINKER_SCRIPT にカスタムリンカスクリプトファイルのパスを指定する必要があります。そうしないと、デフォルトのリンカスクリプトが使用されます。また、カスタムリンクスクリプトをアプリケーションのテンプレートフォルダに配置すると、デフォルトのリンカスクリプトを上書きできます。詳細については、ModusToolbox™ ユーザー ガイドを参照してください。

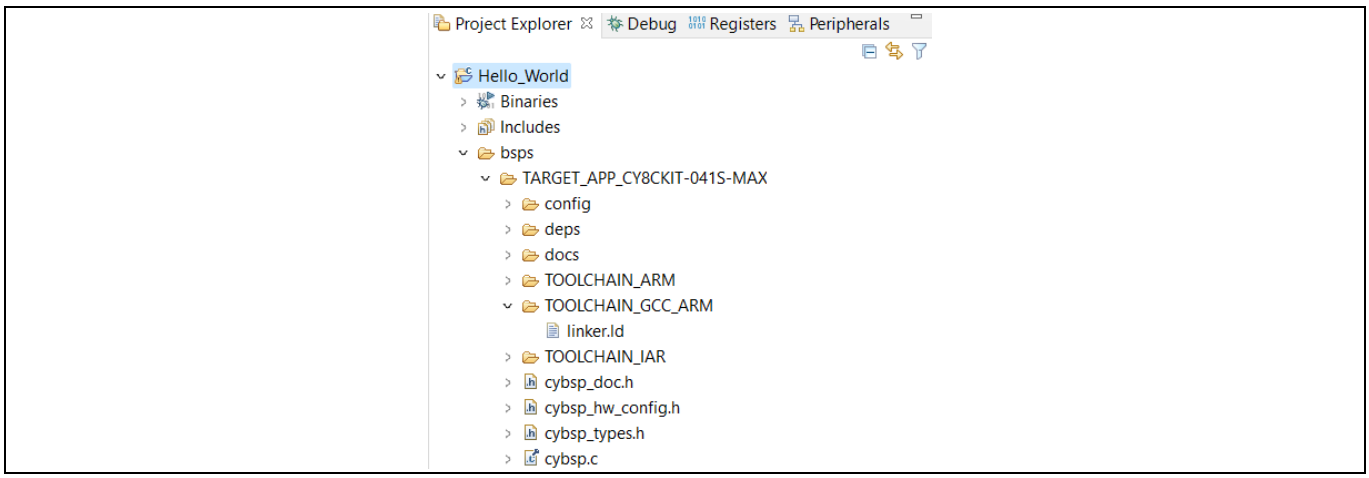


Figure 10 ModusToolbox™ソフトウェアリンカスクリプトファイル

PSoC™ Creator では、Figure 11 に示すように、プロジェクトのビルド後にデフォルトのリンカスクリプトファイルは *Generated Source* フォルダに保存されます。

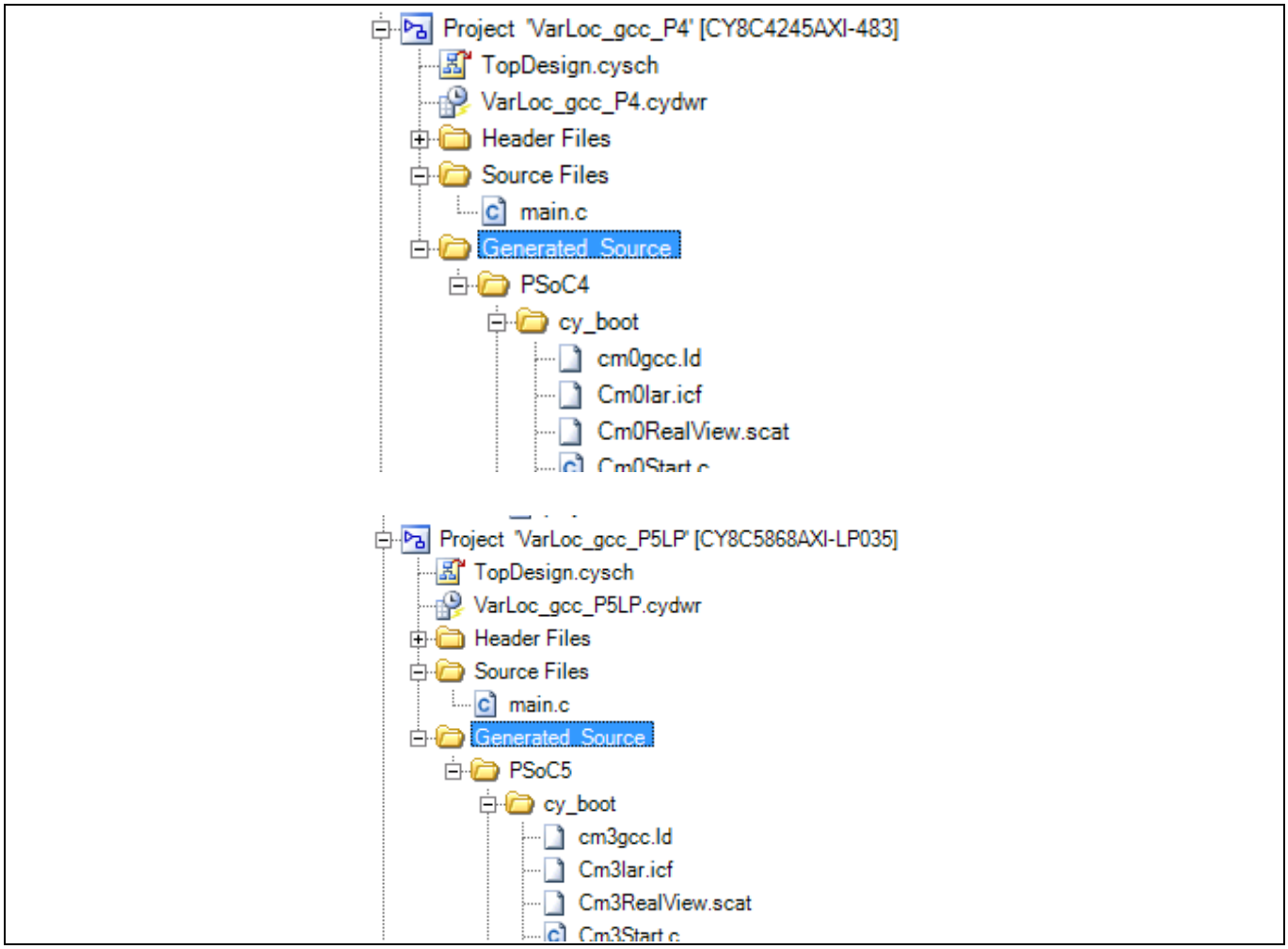


Figure 11 PSoC™ Creator リンカスクリプトファイル

GCC の場合、リンカスクリプトファイルのタイプは *.ld*、MDK の場合、リンカスクリプトファイルのタイプは *.scat* または *.sct* (「scatter」の略)、IAR の場合、リンカスクリプトファイルのタイプは *.icf* です。

Note: PSoC™Creator では、リンカスクリプトファイルはプロジェクトのビルド時に自動的に生成されます。これらのファイルに加えた変更は、次のビルドで上書きされる可能性があります。PSoC™Creator にカスタムスクリプトファイルを使用するように指示できます。
Project** > **Build Settings** > **Linker** > **General** > **Custom Linker Script

カスタムリンカスクリプトファイルを使用する場合は、それをプロジェクトに追加し (**Project** > **Existing Item...**)、プロジェクトフォルダに保存することを推奨します。カスタム *.scat* ファイルは、生成されたソースの下の PSoCx フォルダに保存する必要があります。

9.1.1 GCC のリンカスクリプトファイル

.ld ファイルには、MEMORY {}と SECTIONS {}の 2 つの主要なコマンドがあります。MEMORY コマンドは、PSoC™のすべての物理メモリのタイプ, 場所, および使用法を記述します。

例えば、ModusToolbox™ソフトウェア 3.1 で 384 KB フラッシュと 4 KB SRAM を備えた PSoC™ 4 の場合は以下のとおりです。

```
MEMORY
{
  FLASH (rx): ORIGIN = 0x00000000, LENGTH = 0x00060000
  RAM (rwx): ORIGIN = 0x20000000, LENGTH = 0x00008000

}
```

例えば、PSoC™ Creator で 32 KB フラッシュと 4 KB SRAM を備えた PSoC™ 4 の場合は以下のとおりです。

```
MEMORY
{
  rom (rx) : ORIGIN = 0x0, LENGTH = 32768
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 4096

}
```

rom 領域は PSoC™フラッシュを示し、領域 ram は PSoC™ SRAM を示します。「rwx」の文字は、それぞれ読み出し, 書き込み, 実行のメモリ属性インジケータです。すべての起点と長さの単位はバイト単位で、値は 10 進数または 16 進数にできます。PSoC™ 5LP も同様です。ram ORIGIN 値は、Cortex®-M3 コード/SRAM 領域境界を横切る SRAM を表します (Figure 4)。

例えば、PSoC™ Creator で 256 KB フラッシュと 64 KB SRAM を備えた PSoC™ 5LP の場合は以下のとおりです。

```
MEMORY
{
  rom (rx) : ORIGIN = 0x0, LENGTH = 262144
  ram (rwx) : ORIGIN = 0x20000000 - (65536 / 2), LENGTH = 65536
}
```

SECTIONS コマンドは、すべてのセクションをアドレス順にリストします。次に例を示します。

```
SECTIONS
{
  .text: { ... }
  .rodata: { ... }
  .ramvectors: { ... }
  .noinit: { ... }
  .data: { ... }
  .bss: { ... }
  .heap: { ... }
  .stack: { ... }
}
```

(すべてのセクションが上に示されているわけではなく、主要なセクションのみが示されています。) Figure 12 は、これらのセクションが PSoC™ 4 および PSoC™ 5LP のフラッシュと SRAM に配置される場所を示します。

- .text: 実行コード
- .rodata: const 変数; 初期データ
- .ramvectors: Cortex® 例外ベクタテーブル
- .noinit: 初期化されないデータ
- .data: 明示的に初期化される変数
- .bss: 0 に初期化される変数

- heap (ヒープ)
- stack (スタック)

リンカスクリプトファイルを詳しく調べると、セクションの多くが**領域**ステートメントで終わっていることがわかります。このステートメントは、セクションを配置するメモリ領域をリンカに指示します。次に例を示します。

```
.text: { ... } > FLASH
.data: { ... } > RAM AT>FLASH
.heap: { ... } > RAM
```

AT ステートメントは、変数の明示的な初期化を可能にします。変数の初期化を参照してください。

Note: PSoC™5LP の場合、SRAM 内のセクションの配置は、コード SRAM /上部 SRAM 境界の位置 (0x20000000、Figure 4 を参照) に対して不確定です。

Note: ほとんどのアプリケーションでは、スタックは上位 SRAM にあり、他のセクションはすべてコード SRAM にあると想定できます。これは変更できます。リンカスクリプトファイルを変更するを参照してください。

.ld ファイルの使用に関する完全なドキュメントは、GCC ドキュメントにあります。詳細は C ドキュメントを参照してください。

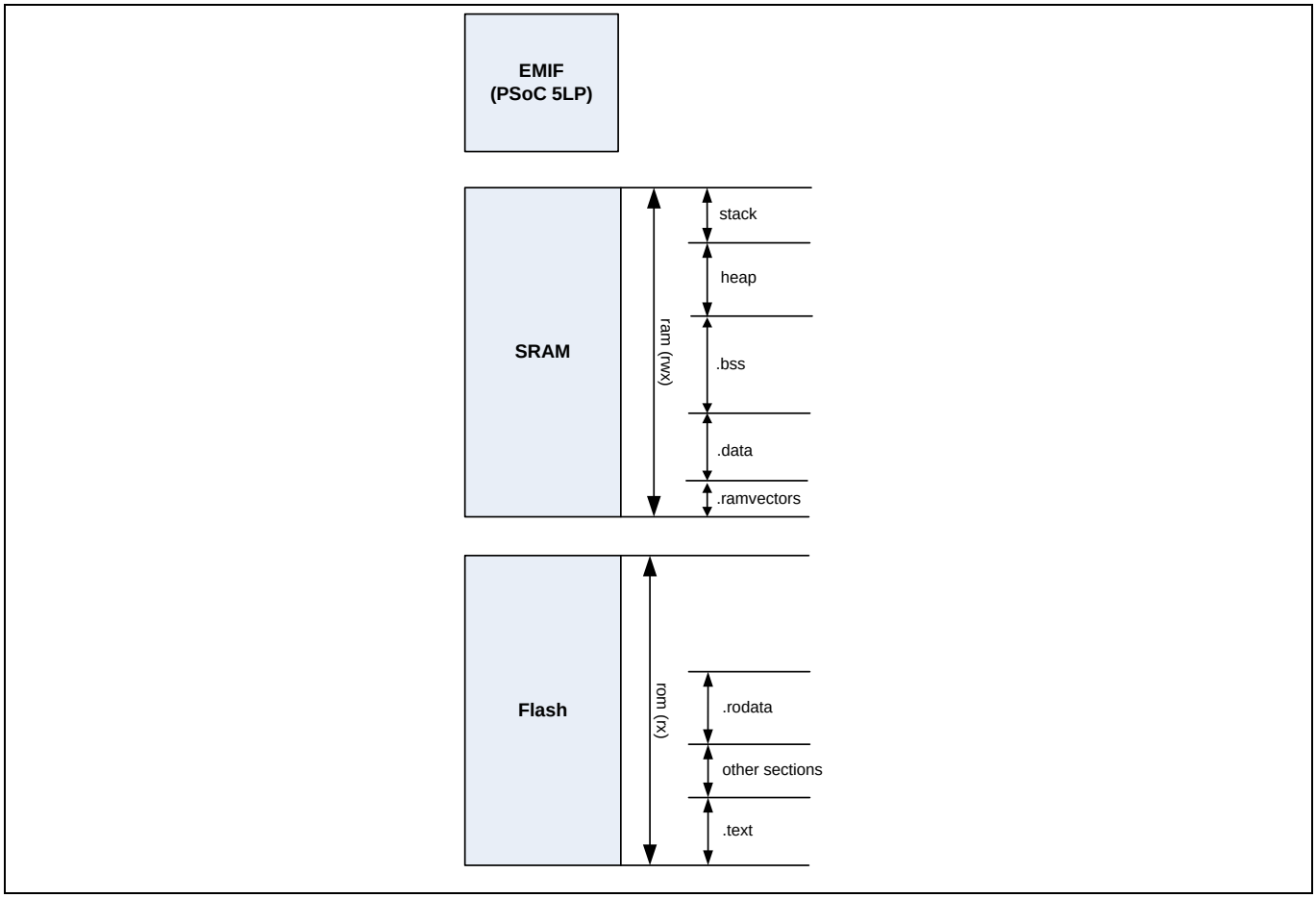


Figure 12 SECTIONS コマンドと PSoC™メモリ

9.1.2 MDK のリンカスクリプトファイル

.scat または *.sct* (「スキャッター」の略) ファイルにはコマンドがありません。かわりに、領域とセクションを定義します。それは**ロード領域**を持ちます。ロード領域にはいくつかの**実行領域**が含まれ、次に、1 つ以上の**セクション属性**が含まれます。次に例を示します。

ModusToolbox™ソフトウェア 3.1 の場合

```
LR_ROM ...     /* load region */
{
    ER_ROM ...   /* execution region */
    {
        *.oc (RESET, +First)
        *(InRoot$$Sections)
        .ANY (+RO)  /* section attribute */
        .ANY (+XO)
    }

    ER_RAM_VECTORS ...
    {
        * (.bss.RESET_RAM, +FIRST)
    }

    RW_RAM_DATA ...
    {
        * (+RW, +ZI)
    }
    RW_IRAM1 ...
    {
        * (.noinit)
    }
    ARM_LIB_HEAP ...
    {
    }

    ARM_LIB_STACK ...
    {
    }

}
```

PSoC™ Creator の場合

```
APPLICATION ... /* load region */
{
      CODE ... /* execution region */
      {
          * (+RO) /* section attribute */
      }
      ISRVECTORS ...
      {
          * (.ramvectors)
      }
      NOINIT_DATA ...
      {
          * (.noinit)
      }
      DATA ...
      {
```

9 コードと変数の配置

```
            .ANY (+RW, +ZI)
        }
        ARM_LIB_HEAP ... { }
        ARM_LIB_STACK ... { }

}
```

(すべての実行領域とセクション属性が上記に示されているわけではなく、主要な属性のみが示されています。) Figure 13 は、領域とセクションが PSoC™ 4 および PSoC™ 5LP フラッシュと SRAM に配置される場所を示します。特別セクション RO, RW, および ZI は次のように定義されます。

- RO: すべてのコード、const 変数、RW セクションの初期化バイト
- RW: 明示的に初期化されるすべての変数。変数の初期化を参照してください
- ZI: ゼロに初期化されるすべての変数。変数の初期化を参照してください

.noinit などの他の属性により、その属性に一致するコードまたは変数が配置されます。カスタム領域の定義を参照してください。

Note: PSoC™5LP の場合、SRAM 内の領域とセクションの配置は、コード SRAM /上部 SRAM 境界の位置 (0x20000000、Figure 13 を参照) に対して不確定です。

Note: ほとんどのアプリケーションでは、スタックとヒープは上位 SRAM にあり、他のセクションはすべてコード SRAM にあると想定できます。これは変更できます。リンカスクリプトファイルを変更するを参照してください。

.scat ファイルの使用に関する完全なドキュメントは、MDK ドキュメントにあります。詳細は C ドキュメントを参照してください。

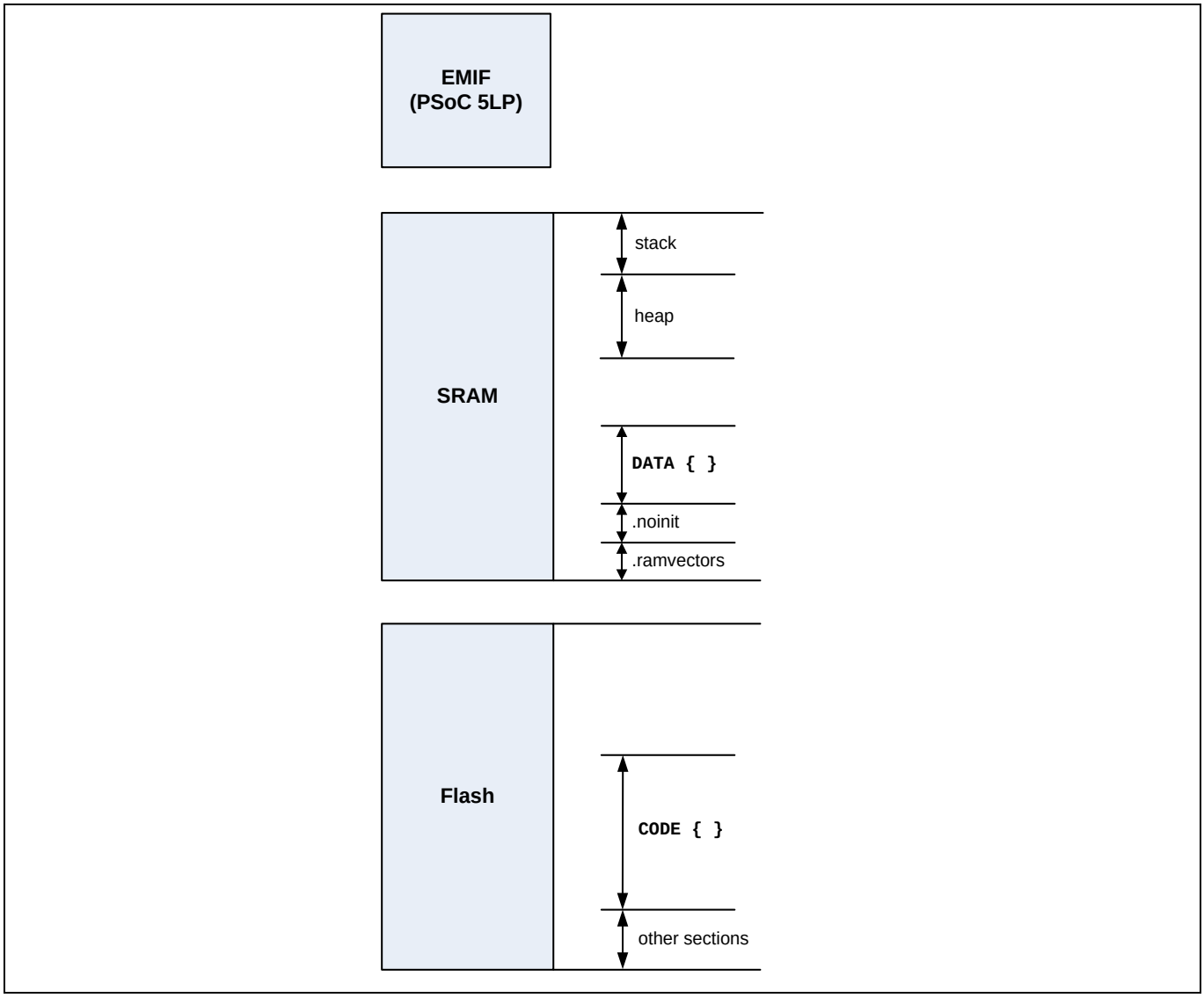


Figure 13　*.scat* ファイルセクションと PSoC™メモリ

9.1.3　IAR のリンカスクリプトファイル

IAR が使用する *.icf* ファイルは、IAR の ILINK ツールによって選択したパーツのデフォルト状態に初期化される設定ファイルです。デフォルトの *.icf* ファイルには、ファイルのヘッダに定義済みのメモリ領域があります。

ModusToolbox™ software 3.1 の場合

```
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_4.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x00000000;

/* The symbols below define the location and size of blocks of memory in the target.
 * Use these symbols to specify the memory regions available for allocation.
 */

/* The following symbols control RAM and flash memory allocation.
 * You can change the memory allocation by editing RAM and Flash symbols.
 */
/* RAM */
```

```
define symbol __ICFEDIT_region_IRAM1_start__ = 0x20000000;
define symbol __ICFEDIT_region_IRAM1_end__   = 0x20007FFF;
define symbol __ICFEDIT_region_IRAM2_start__ = 0x0;
define symbol __ICFEDIT_region_IRAM2_end__   = 0x0;
define symbol __ICFEDIT_region_IRAM3_start__ = 0x0;
define symbol __ICFEDIT_region_IRAM3_end__   = 0x0;
define symbol __ICFEDIT_region_ERAM1_start__ = 0x0;
define symbol __ICFEDIT_region_ERAM1_end__   = 0x0;
define symbol __ICFEDIT_region_ERAM2_start__ = 0x0;
define symbol __ICFEDIT_region_ERAM2_end__   = 0x0;
define symbol __ICFEDIT_region_ERAM3_start__ = 0x0;
define symbol __ICFEDIT_region_ERAM3_end__   = 0x0;

/* Flash */
define symbol __ICFEDIT_region_IROM1_start__ = 0x00000000;
define symbol __ICFEDIT_region_IROM1_end__   = 0x0005FFFF;
define symbol __ICFEDIT_region_IROM2_start__ = 0x0;
define symbol __ICFEDIT_region_IROM2_end__   = 0x0;
define symbol __ICFEDIT_region_EROM1_end__   = 0x0;
define symbol __ICFEDIT_region_EROM1_start__ = 0x0;
define symbol __ICFEDIT_region_EROM2_start__ = 0x0;
define symbol __ICFEDIT_region_EROM2_end__   = 0x0;
define symbol __ICFEDIT_region_EROM3_start__ = 0x0;
define symbol __ICFEDIT_region_EROM3_end__   = 0x0;

/*-Sizes-*/
if (!isdefinedsymbol(__STACK_SIZE)) {
define symbol __ICFEDIT_size_cstack__ = 0x0400;
} else {
define symbol __ICFEDIT_size_cstack__ = __STACK_SIZE;
}
define symbol __ICFEDIT_size_proc_stack__ = 0x0;

/* Defines the minimum heap size. The actual heap size will be expanded to the end of the
   stack region   */
if (!isdefinedsymbol(__HEAP_SIZE)) {
define symbol __ICFEDIT_size_heap__ = 0x0080;
} else {
define symbol __ICFEDIT_size_heap__ = __HEAP_SIZE;
}

/**** End of ICF editor section. ###ICF###*/
```

PSoC™ Creator の場合

```
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x00000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x0;
define symbol __ICFEDIT_region_ROM_end__ = 262144 - 1;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000 - (65536 / 2);
define symbol __ICFEDIT_region_RAM_end__ = 0x20000000 + (65536 / 2) - 1;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x0800; define symbol __ICFEDIT_size_heap__ = 0x80;

/**** End of ICF editor section. ###ICF###*/
```

これらの領域は、IAR ツールを使用して再設定できます。詳細については、IAR のドキュメントを参照してください。

.icf ファイルには、メモリサイズ, 領域アドレス, 初期化, および定義された領域の内容の属性の定義が含まれています。次に例を示します。

9 コードと変数の配置

ModusToolbox™ software 3.1 の場合

```
/* Allocate memory space of the maximum possible addressable size */
define memory mem with size = 4G;

/* Allocate memory regions (ROM and RAM) in the addressable space */
define symbol use_IROM1 = (__ICFEDIT_region_IROM1_start__ != 0x0 ||
__ICFEDIT_region_IROM1_end__ != 0x0);
define symbol use_IRAM1 = (__ICFEDIT_region_IRAM1_start__ != 0x0 ||
__ICFEDIT_region_IRAM1_end__ != 0x0);
define region IROM1_region = mem:[from __ICFEDIT_region_IROM1_start__ to
__ICFEDIT_region_IROM1_end__];
define region IRAM1_region = mem:[from __ICFEDIT_region_IRAM1_start__ to
__ICFEDIT_region_IRAM1_end__];

/* Create a stack and define stack properties */
define block CSTACK     with alignment = 8, size = __ICFEDIT_size_cstack__     { };
define block PROC_STACK with alignment = 8, size = __ICFEDIT_size_proc_stack__ { };

/* Create a heap and define properties */
define block HEAP       with expanding size, alignment = 8, minimum size =
__ICFEDIT_size_heap__ { };
define block HSTACK {block HEAP, block PROC_STACK, last block CSTACK};

/* handle initialization */
initialize by copy { readwrite };
do not initialize  { section <section name> };

/* Place sections in defined regions */
Place at start of <region> { section <section name> };
Place in <region> { <section name> };
```

PSoC™ Creator の場合

```
/* Allocate memory space of the maximum possible addressable size */
Define memory Mem with size = <size>;

/* Allocate memory regions (ROM and RAM) in the addressable space */
Define region ROM = Mem:[from <start_address> size <size>];
define region RAM = Mem:[from <start_address> size <size>];

/* Create a stack and define stack properties */
define block STACK with size = <size>, alignment = <alignment width> { };

/* Create a heap and define properties */
define block HEAP with size = <size>, alignment = <alignment width> { };

/* handle initialization */
Do not initialize ( section <noinit section name> );
Initialize by copy ( readwrite ); /* initialize read/write sections */

/* Place sections in defined regions */
Place at start of <region> ( section <section name> );

Place in <region> ( <section name> );
```

アクセス属性 (R/W または R/O) を使用して、コードまたは変数のセクションを領域に配置することができます。この場合の構文は次のようになります。

```
place in <region> ( <readwrite or readonly>)
```

次に例を示します。

```
place in IRAM1_region  { readwrite };
```

9 コードと変数の配置

Figure 14 に、これらの領域とセクションが PSoC™ 4 および PSoC™ 5LP メモリのどこに配置されているかを示します。

- `.text`: 実行可能コード
- `.rodata`: `const` 変数、初期化データ
- `.ramvectors`: Cortex® 例外ベクトル テーブル
- `.noinit`: 初期化されていない変数
- `.data`: 明示的に初期化されている変数
- `.bss`: 0 に初期化されている変数
- heap (ヒープ)
- stack (スタック)

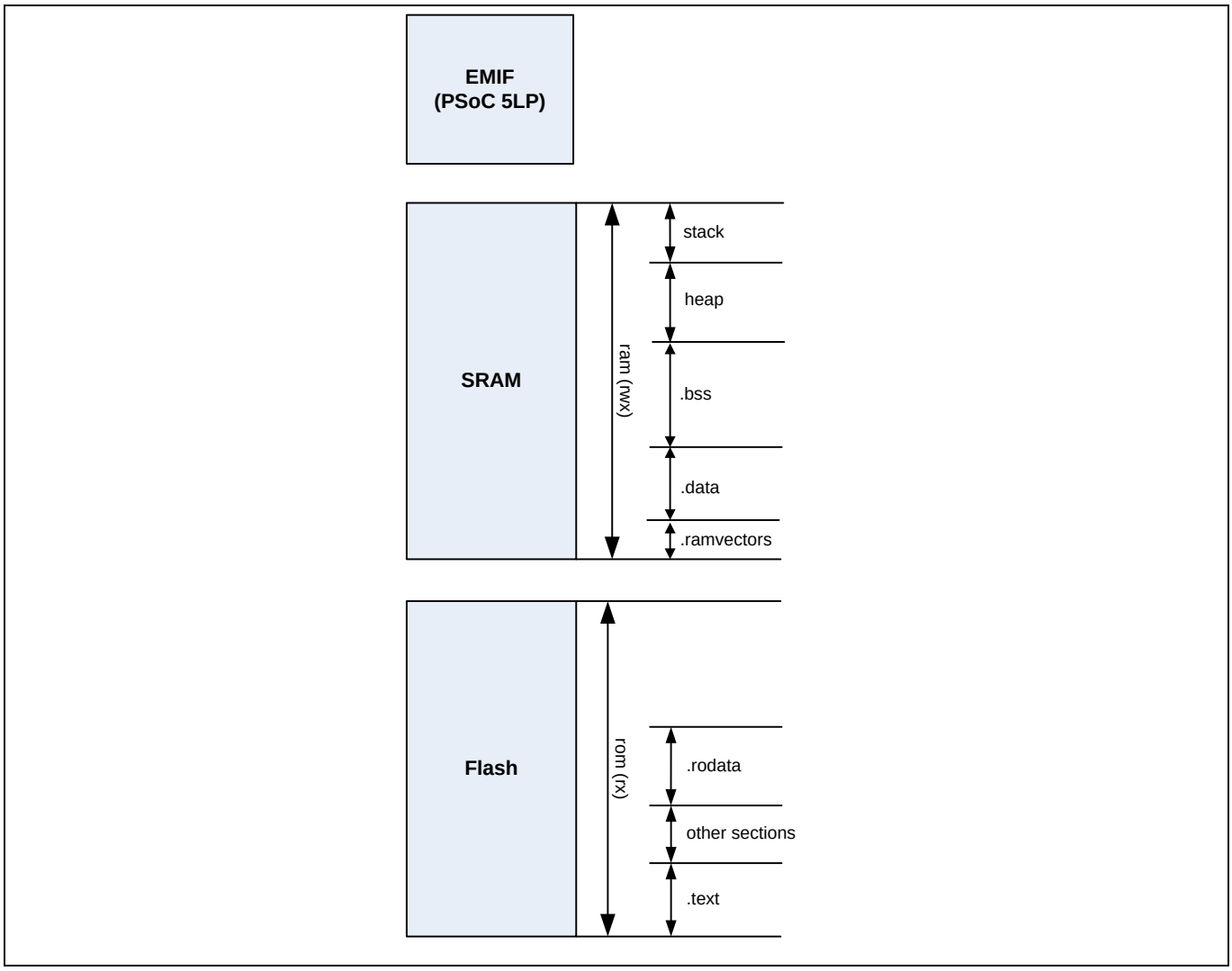


Figure 14 *.icf* ファイルセクションと PSoC™メモリ

以下の点は、3 つのリンカファイル タイプすべてに当てはまります。

1. グローバル変数と C 静的変数のみが含まれます。C 自動変数は別の方法で処理されます。自動変数を参照してください。
2. PSoC™ Creator では、ヒープとスタックのサイズは、プロジェクト DWR ウィンドウの **System** タブ、またはプロジェクトのカスタムリンカファイルで定義されます。ModusToolbox™ソフトウェアでは、

9 コードと変数の配置

ヒープとスタックのサイズはプロジェクトリンカファイルで定義されます。スタックポインタは、最上位の SRAM アドレスに 1 を加えた値に初期化され、スタックは下方向に増加します。malloc()や free()などの C 関数によって使用されるヒープは、ベースから上方向に増加します。

カスタムリンカスクリプトファイルでは、スタックとヒープのサイズをコンパイラごとに指定できます。次の例に、サイズ 0x0400 のスタックの設定を示します。ヒープサイズは、GCC および MDK コンパイラの RAM (サイズとオリジン) とスタックサイズに応じて異なります。IAR コンパイラでは、ヒープのサイズを直接設定できます。

ModusToolbox™ software 3.1 の場合

GCC *.ld*

```
__STACK_SIZE = 0x00000400;

.heap :
 {
    . = ALIGN(4);
     __HeapBase = .;
     __end__ = .;
     PROVIDE(end = .);
     . = ORIGIN(RAM) + LENGTH(RAM) - __STACK_SIZE;
     __HeapLimit = .;
 } > RAM

.stack :
 {
    . = ORIGIN(RAM) + LENGTH(RAM) - __STACK_SIZE;
    . = ALIGN(4);
    __StackLimit = .;
    . = . + __STACK_SIZE;
    . = ALIGN(4);
    __StackTop = .;
 } > RAM
```

MDK *.sct*

```
#define __STACK_SIZE    0x00000400

ARM_LIB_HEAP  +0 EMPTY ((__RAM_START+__RAM_SIZE)-AlignExpr(ImageLimit(RW_IRAM1),
8)-__STACK_SIZE)
{
}

ARM_LIB_STACK (__RAM_START+__RAM_SIZE) EMPTY -__STACK_SIZE
{
}
```

IAR *.icf*

```
define symbol __STACK_SIZE = 0x0400;
define symbol __HEAP_SIZE   = 0x0080;

/*-Sizes-*/
if (!isdefinedsymbol(__STACK_SIZE)) {
define symbol __ICFEDIT_size_cstack__ = 0x0400;
} else {
```

```
define symbol __ICFEDIT_size_cstack__ = __STACK_SIZE;
}
define symbol __ICFEDIT_size_proc_stack__ = 0x0;
/* Defines the minimum heap size. The actual heap size will be expanded to the end
of
   the stack region */
if (!isdefinedsymbol(__HEAP_SIZE)) {
define symbol __ICFEDIT_size_heap__ = 0x0080;
} else {
define symbol __ICFEDIT_size_heap__ = __HEAP_SIZE;
}

define block CSTACK     with alignment = 8, size = __ICFEDIT_size_cstack__
{ };
define block PROC_STACK with alignment = 8, size = __ICFEDIT_size_proc_stack__
{ };
define block HEAP       with expanding size, alignment = 8, minimum size =
__ICFEDIT_size_heap__ { };
define block HSTACK {block HEAP, block PROC_STACK, last block CSTACK};
define block RO     {first section .intvec, readonly};
```

PSoC™ Creator の場合

GCC *.ld*

```
.heap (NOLOAD) :
{
   . = _end;
   . += 0x80;
   __cy_heap_limit = .;
} >ram

.stack (__cy_stack - 0x0800) (NOLOAD) :
{
   __cy_stack_limit = .;
   . += 0x0800;
} >ram
```

MDK .scat

```
ARM_LIB_HEAP (0x2000000 + (65536 / 2) – 0x80 – 0x0800) EMPTY 0x80
{
}

ARM_LIB_STACK (0x2000000 + (65536 / 2).) EMPTY -0x0800
{
}
```

IAR *.icf*

```
define symbol size_stack__ = 0x0800;
define symbol size_heap__   = 0x80;

define block STACK with alignment = 8, size = size_stack__ {};
define block HEAP with alignment = 8, size = size_heap__ {};

/* Group stack and heap together, with stack last */
```

```
define block HSTACK(block HEAP, last block STACK};

/*Place HSTACK block in end of RAM */
“HSTACK”       : place at end of RAM_region { block HSTACK} ;
```

3. ヒープは PSoC™ Creator DWR ウィンドウで定義されたサイズを持ちますが、実際には SRAM のセクションとスタックポインタの現在の値の間の SRAM 全体を使用できます。同様に、スタックは定義されたスタックセクションを超えて下方向に拡大できます。スタックがその下のメモリ領域と重なり始めると、見つけにくい欠陥が発生する可能性があります。スタックオーバーフローを検出する 1 つの方法は、各関数にスタックポインタの現在の値を確認するコードを追加することです。

9.1.4 変数の初期化

グローバル変数と静的変数をカスタムの場所に配置する場合、それらがどのように初期化されるかを理解することが重要です。例えば、次のグローバル変数の定義を検討してください。

```
uint8 foo = 5;
uint16 myArray[10] = {1234, 12, ... };
```

これらの変数は SRAM にあるため、PSoC™に電源が投入されたときの値は未定義です。それらを適切に初期化するために、値はフラッシュに保存され、C スタートアップコード、つまり `main()`の前に実行されるコードは、フラッシュから SRAM に値をコピーします。フラッシュ内の値は、GCC の `.rodata` セクション (Figure 12) または MDK の `CODE (RO)`セクション (Figure 13) にあります。これらは、GCC および IAR の場合は `.data` セクション、MDK の場合は `DATA (RW)`セクションの変数にコピーされます。明示的に初期化された変数は、これらのセクションに必ず配置する必要があります。

明示的に初期化されていないグローバル変数と静的変数は、C スタートアップコードによってゼロに設定されます。これらは、`.bss` セクション (GCC および IAR) または `DATA (ZI)`セクション (MDK) に配置する必要があります。

上記のセクションの外にあるグローバル変数と静的変数は初期化されておらず、初期値は定義されていません。コードで初期化する必要があります。明示的な初期化は無視されます。

9.1.5 マップファイル

GCC リンカと MDK リンカのどちらにも、プロジェクトのビルド時に*.map* ファイルを生成するオプションがあります。PSoC™ Creator では、Figure 15 に示すように、このファイルは Workspace Explorer ウィンドウの **Results** タブにあります。ModusToolbox™ ソフトウェアでは、このファイルは *<Project Directory>\build\APP_<Kit Name>\<Config>* にあります。

.map ファイルは、すべてのコードモジュールと変数がリンカによってメモリ内のどこに配置されたかを示します。ビルド操作後にそれを確認し、以下を確認する必要があります。

- すべてのコードと変数が予想される場所に配置されています。
- セクションの重複はありません。

MDK の場合、リンカのデフォルトでは、シンボルのない*.map* ファイルが生成されるため、コードと変数が配置されている場所を特定することが難しくなります。シンボルを含めるには、リンカのコマンドラインに`--symbols` を追加します。

PSoC™ Creator メニューを使用して、**Project > Build Settings > Linker > Command Line > Custom Flags** に異動します。

ModusToolbox™ソフトウェアで、Makefile の LDFLAGS フィールドに`–symbols` フラグを追加します。

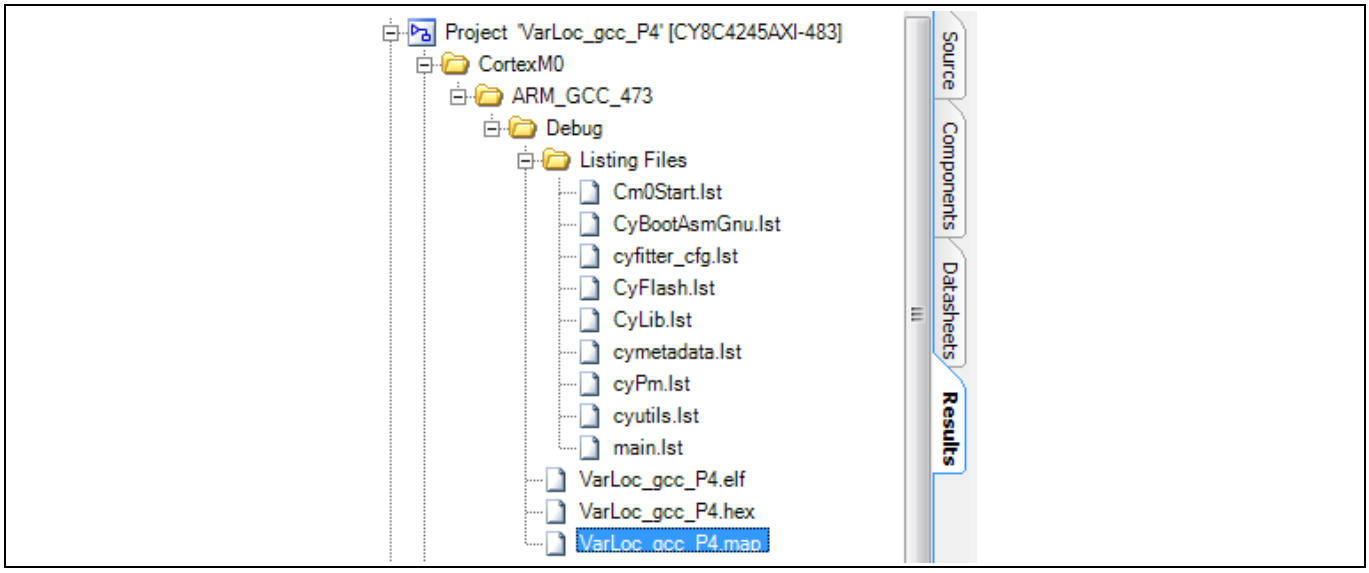


Figure 15 PSoC™ Creator の*.map* ファイル

リンカスクリプトファイルの動作の基本を確認したので、次に、リンカスクリプトファイルを使用してPSoC™メモリ内のカスタムの場所にコードまたは変数を配置する方法を検討します。

9.2 配置手順

C 関数または変数 (配列と構造体を含む) をカスタムの場所に配置するには、次の手順を実行します。

1. カスタムの場所を定義します。
2. C ソースコードで、カスタム セクションとともに、配置する関数と変数を宣言します。
3. プロジェクトをビルドします。生成されたリンカスクリプトファイルをカスタムファイルにコピーし、それを変更して手順 2 のセクションを追加して配置します。
4. プロジェクトを再ビルドします。*.map* ファイルを確認し、カスタムの場所が正しく入力され、セクションが重複していないことを確認します。

これらの各手順を詳しく見ていきましょう。

9.2.1 カスタム領域の定義

カスタムの場所を使用する前に、それらを使用する理由を明確に理解する必要があります。例えば、次の操作を行うかです。

- 関数または const 型の変数をフラッシュのカスタム領域に配置しますか?
- 関数を SRAM に配置して、実行を高速化しますか? その場合、PSoC™ 4 および PSoC™ 5LP でのフラッシュアクセスは SRAM アクセスとほぼ同じ速さなので、大幅なパフォーマンスの向上は実現されない可能性があることに注意してください。詳細については、PSoC™ 4 の Cortex®-M0/M0+および PSoC™ 5LP の Cortex®-M3 を参照してください。
- C スタートアップコードによって初期化されないように変数を配置しますか? これは通常、デバイスのリセット (パワーサイクルリセットを除く) を通じて変数の状態を維持するために使用されます。
- PSoC™ 5LP の上位 SRAM に変数を配置して、ビットバンドアクセスを可能にしますか?
- 変数を SRAM の他のカスタム領域に配置しますか?
- PSoC™ 5LP EMIF メモリに変数を配置しますか?

上記の質問に対する回答は、カスタムロケーションのアドレスを決定するのに役立ちます。

9.2.2 関数と変数の宣言

カスタムロケーションアドレスを決定したら、関数と変数を宣言します。宣言で、__attribute__キーワード (「attribute」の前後に 2 つのアンダースコア文字) を使用して、それらが存在するセクションを追加します。

```
uint8 foo __attribute__ ((section(".MY_section")));
```

このキーワードは、GCC, IAR, および MDK で使用できます。PSoC™ Creator および ModusToolbox™ソフトウェアは、次のようにこのステートメントを簡略化する便利なマクロ CY_SECTION を提供します。

```
uint8 foo CY_SECTION(".MY_section"); /* no explicit initialization, = 0 */
uint8 foo CY_SECTION(".MY_section") = 10; /* explicit initialization */
/* declare a function's section in the prototype only, and not in the
   actual function */
uint16 MyFunction(char *x) CY_SECTION(".MY_section");
/* CYISR is a PSoC Creator macro to define an interrupt handler function.
   See References for more information. */
CYISR(MyFunction) CY_SECTION(".MY_section");
```

PSoC™ Creator および ModusToolbox™ソフトウェアは、便利なマクロ CY_NOINIT も提供します。これは、変数を.noinit セクションに配置します。Figure 12 と Figure 13 を参照してください。

```
/* no initialization by C startup code, initial value is undefined */
        uint8 foo CY_NOINIT;
```

9.2.3 リンカスクリプトファイルを変更する

最後のタスクは、プロジェクトのリンカスクリプトファイルを変更して、以前に定義したセクションを配置する場所を宣言することです。

GCC の場合、リンカスクリプトの*.ld* ファイルを変更します。SECTIONS{}コマンドと、場合によってはMEMORY{}コマンドを変更します。これを変更する一般的な方法は、Table 9 に示すように、EMIF メモリのステートメントを追加するか、SRAM を分割することです。

Table 9 リンカスクリプトファイルを変更する

ModusToolbox™ software 3.1 の場合

```
MEMORY
   {
     FLASH (rx) : ORIGIN = 0x0, LENGTH = 262144
     CODERAM (rwx) : ORIGIN = 0x20000000 - (65536 / 2), LENGTH = (65536 / 2)
     UPPERRAM (rwx) : ORIGIN = 0x20000000, LENGTH = (65536 / 2)
     EMIF (rwx) : ORIGIN = 0x60000000, LENGTH = 0x1000000
   }
```

PSoC™ Creator の場合

```
MEMORY
   {
     rom (rx) : ORIGIN = 0x0, LENGTH = 262144
     coderam (rwx) : ORIGIN = 0x20000000 - (65536 / 2), LENGTH = (65536 / 2)
     upperram (rwx) : ORIGIN = 0x20000000, LENGTH = (65536 / 2)
     EMIF (rwx) : ORIGIN = 0x60000000, LENGTH = 0x1000000
   }
```

Note: SRAM 領域名を変更すると、デフォルトファイルの一部のセクション定義でエラーが発生するため、必要に応じて各セクション定義を変更する必要があります。例えば、PSoC™Creater では、.stack:{...}>ram を.stack:{...}>upperram に変更します。ModusToolbox™ソフトウェア3.1 では、.stack:{...}>RAM を.stack:{...}>UPPERRAM に変更します。

セクションを見つけるには、セクション定義を SECTIONS{}コマンドに追加します。セクション定義の構文は次のとおりです。

```
.MY_section <address> <(NOLOAD)> : <alignment>
{
  *(.MY_section)
} > <memory region>
```

セクション定義名とそのセクション内のセクション名は同じにすることができます。名前の最初の文字をピリオド「.」にすることは、必須ではありませんが、一般的な規則です。

前述のように、MEMORY{}コマンドのメモリ領域名を使用します。

Table 10 に示すように、既存のセクション内にセクションを配置することもできます。

Table 10 例

ModusToolbox™ software 3.1 の場合

```
.data:
{
  ...
  *(.MY_section)
  ...
} > RAM AT> FLASH
```

PSoC™ Creator の場合

```
.data:
{
  ...
  *(.MY_section)
  ...
} > ram AT> rom
```

場合によっては、セクションが最適化によって削除されることがあります。例えば、そのセクションにシンボルが存在しないとします。カスタム セクションが削除されないようにするには、キーワード KEEP を使用してセクションを「keep」するように指定できます。GCC および IAR では、KEEP 属性はリンカファイルに適用されます。MDK では、keep は Table 11 に示すように、リンカのコマンドラインオプションです。

Table 11　例

ModusToolbox™ software 3.1 の場合

GCC	IAR	MDK
.data: { ... KEEP(*(.MY_section)) ... } > RAM AT> FLASH	KEEP { section .MY_section };	/* In the command line */ --keep=.MY_section

PSoC™ Creator の場合

GCC	IAR	MDK
.data: { ... KEEP(*(.MY_section)) ... } > ram AT> rom	KEEP { section .MY_section };	/* In the command line */ --keep=.MY_section

MDK の場合は、リンカー スクリプトの.scat ファイルを変更します。手順は GCC の場合と似ていますが、より簡単です。変更する MEMORY{}または SECTIONS{}コマンドはありません。代わりに、Table 12 に示すように、実行領域を追加するだけです。

Table 12　例

GCC の場合

ModusToolbox™ software 3.1 の場合

```
MY_REGION <address> <UNINIT> <length>
{
  * (.MY_section)
}
```

既存のセクション内にセクションを配置することもできます。次に例を示します。

```
RW_RAM_DATA +0
{
  * (.MY_section)
  * (+RW, +ZI)
}
```

PSoC™ Creator の場合

```
MY_REGION <address> <UNINIT> <length>
{
  * (.MY_section)
}
```

既存のセクション内にセクションを配置することもできます。次に例を示します。

```
DATA
{
  * (.MY_section)
  .ANY (+RW, +ZI)
}
```

IAR の場合

ModusToolbox™ software 3.1 の場合

In IAR, modify the linker script *.icf* file. The procedure for adding a section is simple, just add your execution section as a new line. You can place the section in an absolute address in memory, for example:

IAR で、リンカー スクリプトの*.icf* ファイルを変更します。セクションを追加する手順は簡単で、実行セクションを新しい行として追加するだけです。セクションをメモリ内の絶対アドレスに配置できます。次に例を示します。

```
“.My_Section” : place at address mem : <address> { readonly section .My_Section };
```

You can also place the section within an existing section, for example:

```
/* places “My_Section” in “readwrite” section within RAM */
".My_Section"           : place in RAM_region { readwrite section .My_Section };
```

PSoC™ Creator の場合

IAR で、リンカー スクリプトの*.icf* ファイルを変更します。セクションを追加する手順は簡単で、実行セクションを新しい行として追加するだけです。セクションをメモリ内の絶対アドレスに配置できます。次に例を示します。

```
 “.My_Section”: place at address mem : <address> {<attribute>};
```

既存のセクション内にセクションを配置することもできます。次に例を示します。

```
/* places “My_Section” in “readwrite” section within RAM */
".My_Section"           : place in RAM_region { readwrite section readwrite };
```

9.3 例

前述のように、カスタムロケーションにはいくつかの異なるアプリケーションがあります。配置手順の例として、そのうちの 1 つを見てみましょう。この例では、配列を PSoC™ 5LP の上位 SRAM に配置して、Cortex®-M3 ビットバンド機能でアクセスできるようにします。詳細については Cortex®-M3 ビットバンド (PSoC™ 5LP のみ) を参照してください。

1. 「.bitband」と呼ばれるセクションを占めるように配列を定義します。

```
uint8 myArray[10] CY_SECTION(".bitband");
```

2. リンカスクリプトファイルを変更して、.bitband セクションをアドレス 0x20000000 から始まる上位 SRAM に配置するように指示する必要があります。Table 13 は、GCC (*.ld* ファイル) または MDK (*.scat* ファイル) でこれを行う方法を示します。

Table 13　リンカスクリプトファイルの変更例

	PSoC™ Creator の場合

GCC の例, *.ld* file	`/* put our .bitband section between the .heap section in code SRAM`
	`   and the .stack section in upper SRAM */`
	`.heap (NOLOAD) :`
	`{`
	`   . = _end;`
	`   . += 256;`
	`   __cy_heap_limit = .;`
	`} >ram`
	
	`.bitband 0x20000000 (NOLOAD) :`
	`{`
	`   *(.bitband)`
	`} >ram`
	
	`.stack (__cy_stack - 256) (NOLOAD) :`
	`{`
	`   __cy_stack_limit = .;`
	`   . += 256;`
	
	`}>ram`

MDK の例, *.scat* file	`/* put our BITBAND execution region between the DATA execution region`
	`   in code SRAM and the heap in upper SRAM */`
	`DATA +0`
	`{`
	`   .ANY (+RW, +ZI)`
	`}`
	
	`BITBAND 0x20000000 UNINIT`
	`{`
	`   * (.bitband)`
	`}`
	
	`ARM_LIB_HEAP (0x20000000 + (65536 / 2) - 256 - 256) EMPTY 256`
	`{`
	`}`
	
	`ARM_LIB_STACK (0x20000000 + (65536 / 2)) EMPTY -256`
	`{`
	
	`}`

IAR の例, *.icf* file

```
/****** Initializations ******/
initialize by copy { readwrite };
do not initialize { section .noinit };
do not initialize { readwrite section .ramvectors };
do not initialize { section .bitband };

/******** Placements *********/
if (CY_APPL_LOADABLE)
{
".cybootloader" : place at start of ROM_region {block
LOADER};
}
"APPL" : place at start of APPL_region {block APPL};

"RAMVEC" : place at start of RAM_region { readwrite
section .ramvectors };
"readwrite" : place in RAM_region { readwrite };
"HSTACK" : place at end of RAM_region { block HSTACK};
“.bitband" : place in RAM_region { readwrite
section .bitband };
keep { section .cybootloader,
       section .cyloadermeta,
       section .cyloadablemeta,
       section .cyconfigecc,
       section .cy_checksum_exclude,
       section .cycustnvl,
       section .cywolatch,
       section .cyeeprom,
       section .cyflashprotect,
       section .cymeta,
       section .bitband };
```

3. プロジェクトをビルドし、*.map* ファイルを確認して、配列が正しく配置され、セクションの重複がないことを確認します。

myArray の初期値は未定義であることに注意してください。コード内で初期化する必要があります。詳細については、IAR のリンカスクリプトファイルを参照してください。

ビットバンド領域にある変数のビットレベルアクセスを行う方法の詳細については、Cortex®-M3 ビットバンド (PSoC™ 5LP のみ) を参照してください。

9.4 一般的な注意事項

カスタムセクションでコードと変数を宣言するときは、次の点に注意してください。

- 明示的に初期化された変数は、.data セクション (GCC および IAR、Figure 12) または DATA 実行領域 (MDK、Figure 13) に配置する必要があります。詳細については、変数の初期化を参照してください。同様に、明示的な初期化がなく、ゼロに自動初期化されると予想される変数は、.bss セクション (GCC および IAR) または DATA 実行領域 (MDK) に配置する必要があります。(MDK コンパイラは、変数が明示的に初期化されているかどうかに応じて、変数に RW または ZI セクション属性を自動的に与えます。)
  上記のセクションに配置されていない変数は初期化されません。明示的な初期化は無視されます。
- SRAM に配置される関数は、.data セクション (GCC および IAR、Figure 12) または DATA 実行領域 (MDK、Figure 13) に配置する必要があります。
- 配列内の固定データテーブルなどのすべての定数データをフラッシュに配置する方が効率的ですが、デフォルトでは SRAM に配置されます。変数をフラッシュに配置するには、その型を const に設定し、明示的に初期化します。次に例を示します。

```
uint32 const var_in_flash = 0x12345678;
```

- フラッシュでカスタムの場所を使用している場合、PSoC™ Creator および ModusToolbox™ソフトウェアのブートローダは、フラッシュの一番上の 1 行または 2 行を使用して、ブートローダブルファイルに関する情報を保存することに注意してください。詳細については、ブートローダコンポーネントのデータシートおよび ModusToolbox™の DFU ミドルウェアを参照してください。
- MDK では、任意のアドレスに変数を配置する最も簡単な方法は、以下のとおりです。

```
__attribute__((at(address))) variable attribute. For example:
uint32 const var_in_flash[] __attribute__((at(0x300))) = { . . . };
```

次の.*map* ファイルのプログラムコードが示すように、リンカは特別なセクションを定義し、変数を目的のアドレスに配置し、必要に応じて他のコードと変数の配置を調整します。

```
Symbol Name                 Value        Ov Type  Size  Object(Section)
. . .
.text                       0x000002f4   Section     0  indicate_semi.o(.text)
.text                       0x000002f4   Section     0  exit.o(.text)
.ARM.__AT_0x00000300        0x00000300   Section    12  main.o(.ARM.__AT_0x00000300)
.text                       0x0000030c   Section     0  init_alloc.o(.text)
.text                       0x00000394   Section     0  h1_free.o(.text)
. . .
```

詳細については、C ドキュメントを参照してください。

- `.ramvectors` セクションは常に SRAM の下部にあり、スタックセクションは常に SRAM の上部にある必要があります。

.ld ファイルの使用に関する完全なドキュメントは、PSoC™ Creator インストールフォルダにあります。通常は次のとおりです。
C:\Program Files (x86)\Cypress\PSoC Creator\4.3\PSoC Creator\import\gnu\arm\5.4.1\share\doc\gcc-arm-none-eabi\pdf\ld.pdf.

.scat ファイルの使用に関する完全なドキュメントは、MDK インストールフォルダにあります。通常は次のとおりです。
C:\Keil\ARM\Hlp\armlink.chm and *C:\Keil\ARM\Hlp\armlinkref.chm*.

.icf ファイルの使用に関する完全なドキュメントは、IAR インストールフォルダにあります。通常は次のとおりです。
C:\Program Files (x86)\IAR Systems\Embedded Workbench 8.4\arm\doc\EWARM_DevelopmentGuide.ENU.pdf

9.5 EMIF の考慮事項 (PSoC™ 5LP のみ)

前述の手法を使用して、PSoC™ 5LP がサポートする外部メモリ (EMIF) に変数を配置することは可能ですが、いくつかの制限があります。

- PSoC™ Creator プロジェクトの回路図に EMIF コンポーネントを配置する必要があります。外部メモリアドレス, データ, および制御ラインは、かなりの数のデバイスピンを使用できることに注意してください。それに応じて設計を計画してください。詳細については、外部メモリインターフェース (EMIF) コンポーネントのデータシートを参照してください。

9 コードと変数の配置

- EMIF API 関数 EMIF_Start()が呼び出されるまで、外部メモリにアクセスできません。したがって、C スタートアップで EMIF 変数を初期化することはできません。コードが main()に達し、EMIF_Start()が呼び出された後で、それらを初期化する必要があります。
- EMIF は 8 ビットおよび 16 ビットのメモリをサポートしています。異なるサイズの変数の配置とアクセスが考慮事項になる場合があります。16 ビットと 32 ビットの変数と構造体のメンバーを、それぞれ 2 バイトと 4 バイトの境界に揃えることを推奨します。
- コードは EMIF から実行できますが、16 ビットの外部メモリでのみ実行できます。コードは、デバイスの内部フラッシュまたは SRAM からよりもはるかに遅く実行されます。また、外部メモリのコードを初期化することも困難です。一般に、外部メモリにコードを置くことは推奨しません。

10 PSoC™ 4 デバイスコードを最適化するヒントとコツ

10.1 シンプルで高速な低電力モードエントリ機能の使用

CPU をスリープまたはディープスリープ電源モードにするには、次の 4 つの関数があります。

- `Cy_SysPm_CpuEnterSleep()`
- `Cy_SysPm_CpuEnterDeepSleep()`
- `Cy_SysPm_CpuEnterSleepNoCallbacks()`
- `Cy_SysPm_CpuEnterDeepSleepNoCallbacks()`

関数 `Cy_SysPm_CpuEnterSleep()`と `Cy_SysPm_CpuEnterDeepSleep()`は、登録されたコールバック関数を自動的に実行して、CPU をそれぞれスリープおよびディープスリープ電源モードにします。

`Cy_SysPm_CpuEnterSleep()`関数を呼び出す際、スリープモードに入る前に、移行前に登録されたコールバックが実行され、周辺機器が CPU スリープの準備をできるようにします。その後、CPU はスリープモードに入ります。これは CPU 中心の電源モードです。つまり、CPU はスリープモードに入り、メインクロックが削除されています。周辺機器の観点からは、CPU アクティブモードと同じです。有効な割込みは、スリープモードからウェイクアップを引き起こす可能性があります。CPU スリープからのウェイクアップ後、遷移登録されたコールバックが実行され、周辺機器がアクティブ操作に戻ります。

`Cy_SysPm_CpuEnterDeepSleep()`関数の呼び出し中、ディープスリープモードに入る前に、遷移登録されたコールバックが実行され、周辺機器が CPU ディープスリープの準備ができるようになります。その後、ディープスリープモードに入ります。有効な割込みは、ディープスリープモードからウェイクアップを引き起こす可能性があります。ディープスリープモードからのウェイクアップ後、遷移登録されたコールバックが実行され、周辺機器がアクティブ操作に戻ります。

関数 `Cy_SysPm_CpuEnterSleepNoCallbacks()`および `Cy_SysPm_CpuEnterDeepSleepNoCallbacks()`は、登録されたコールバック関数を呼び出さずに、CPU をそれぞれスリープおよびディープスリープ電力モードにします。アプリケーションは、デバイスを低電力モードに入るように準備し、ウェイクアップ時に目的の状態に戻す必要があります。

したがって、コールバックなし関数を使用すると、登録されたコールバックを実行せずに起動時に即座に低電力モードに切り替えることでデバイスの総電力消費を削減し、フラッシュメモリを節約できるという利点があります。

Table 14 低電力モードエントリ関数のフラッシュメモリ消費量

関数	CY8CKIT-041S-MAX キットのフラッシュメモリ消費量 (バイト単位)					
	GCC_ARM		Arm®		IAR	
	デバッグ	リリース	デバッグ	リリース	デバッグ	リリース
`Cy_SysPm_CpuEnterSleep()`	332	292	324	260	392	416
`Cy_SysPm_CpuEnterDeepSleep()`	360	316	356	284	420	308
`Cy_SysPm_CpuEnterSleepNoCallbacks()`	20	28	24	24	24	24
`Cy_SysPm_CpuEnterDeepSleepNoCallbacks()`	44	52	40	40	40	40

Cy_SysPm_CpuEnterSleepNoCallbacks()を使用すると、Table 14 に示すように、GCC_ARM デバッグモードで Cy_SysPm_CpuEnterSleep()と比較して 312 バイトを節約できます。

Cy_SysPm_CpuEnterDeepSleepNoCallbacks()を使用すると、Table 14 に示すように、GCC_ARM デバッグモードで Cy_SysPm_CpuEnterDeepSleep()と比較して 316 バイトを節約できます。

Note: 関数の詳細については、PDL ライブラリの一部であるソースファイル cy_syspm.c を参照してください。

10.2 アプリケーションで複数のペリフェラルが使用される場合のリソース設定の再利用

アプリケーションで複数の周辺機器を使用し、それらすべてが同じリソース設定を必要とする場合、リソース設定は、周辺機器ごとに個別に作成する必要はなく、再利用できます。

例えば、デバイスコンフィギュレータを使用して GPIO ピンを LED 出力として設定すると、LED 出力ピンごとに個別のリソース設定が作成されます。最も一般的なケースでは、リソース設定の構造は同じである必要があります。その結果、すべてのユーザーLED 初期化でリソース設定を再利用できます。

デフォルトでは、*cycfg_pins.c* ファイルで LED 出力ピンごとに個別のリソース設定が作成されます。このファイルは、次のパスにあります。 *<project_directory>/bsps/<target>/config/GeneratedSource*

Table 15 現在のフローと設定を再利用するための回避策フロー

現在のフロー

```
#define CYBSP_LED1_HSIOM HSIOM_SEL_GPIO
#define CYBSP_LED2_HSIOM HSIOM_SEL_GPIO

const cy_stc_gpio_pin_config_t CYBSP_LED1_config =
{
.outVal = 1,
.driveMode = CY_GPIO_DM_STRONG_IN_OFF,
.hsiom = CYBSP_LED1_HSIOM,
.intEdge = CY_GPIO_INTR_DISABLE,
.vtrip = CY_GPIO_VTRIP_CMOS,
.slewRate = CY_GPIO_SLEW_FAST,
};

const cy_stc_gpio_pin_config_t CYBSP_LED2_config =
{
.outVal = 1,
.driveMode = CY_GPIO_DM_STRONG_IN_OFF,
.hsiom = CYBSP_LED2_HSIOM,
.intEdge = CY_GPIO_INTR_DISABLE,
.vtrip = CY_GPIO_VTRIP_CMOS,
.slewRate = CY_GPIO_SLEW_FAST,
};

Cy_GPIO_Pin_Init(CYBSP_LED2_PORT, CYBSP_LED2_PIN,
&CYBSP_LED2_config);

Cy_GPIO_Pin_Init(CYBSP_LED1_PORT, CYBSP_LED1_PIN,
&CYBSP_LED1_config);
```

回避策フロー

```
#define CYBSP_LED_HSIOM HSIOM_SEL_GPIO

    const cy_stc_gpio_pin_config_t CYBSP_LED_config =
```

```
    {
    .outVal = 1,
    .driveMode = CY_GPIO_DM_STRONG_IN_OFF,
    .hsiom = CYBSP_LED_HSIOM,
    .intEdge = CY_GPIO_INTR_DISABLE,
    .vtrip = CY_GPIO_VTRIP_CMOS,
    .slewRate = CY_GPIO_SLEW_FAST,
     };

     Cy_GPIO_Pin_Init(CYBSP_LED2_PORT, CYBSP_LED2_PIN,
&CYBSP_LED_config);

     Cy_GPIO_Pin_Init(CYBSP_LED1_PORT, CYBSP_LED1_PIN,
&CYBSP_LED_config);
```

Table 16　回避策フローを使用したフラッシュメモリの節約

CY8CKIT-041S-MAX キットのフラッシュメモリ節約 (バイト単位)					
GCC_ARM		**Arm®**		**IAR**	
デバッグ	リリース	デバッグ	リリース	デバッグ	リリース
24	24	24	24	24	24

回避策フローを使用すると、既存のリソース構成を再利用することで、各ペリフェラルの初期化ごとに Table 16 に記載されているバイト数を節約できます。この回避策を適用するには複数の方法があります。詳細については、重要な注意事項を参照してください。

10.3　クロック, ペリフェラル, およびピン設定予約の最適化

ModusToolbox™ では、BSP の初期化中に、`cycfg_config_reservations()`関数がデフォルトで呼び出されます。ただし、この関数は、アプリケーションで HAL を使用してクロック, 周辺機器, およびピン設定を予約する場合にのみ呼び出す必要があります。

HAL ベースのクロック, 周辺機器, およびピン設定の予約は、アプリケーションで HAL が使用されていない場合にのみ無効にする必要があります。

`cycfg_config_reservations()`関数は次のパスにあります。

<project_directory>/bsps/<target>/config/GeneratedSource/cycfg.c.

Table 17　HAL ベースの設定予約の最適化のための現在のフローと回避策

Current flow	`void init_cycfg_all(void)`
	`{`
	`  cycfg_config_init();`
	`  cycfg_config_reservations();`
	`}`

Workaround flow	`void init_cycfg_all(void)`
	`{`
	`  cycfg_config_init();`
	`}`

Table 18　回避策フローを使用したフラッシュメモリの節約

GCC_ARM		Arm®		IAR	
デバッグ	リリース	デバッグ	リリース	デバッグ	リリース
32	24	24	28	28	20

回避策フローを使用すると、Table 18 に記載されているバイト数を節約できます。この回避策を適用する方法は複数あります。詳細については、重要な注意事項を参照してください。

10.4　デバイスコンフィギュレータの最適化

エンドレベルのアプリケーションを開発した後、デバッグ ポートを無効にできます。次のように、システム設定でデバッグ チェック ボックスをオフにして、デバッグ オプションを無効にします。

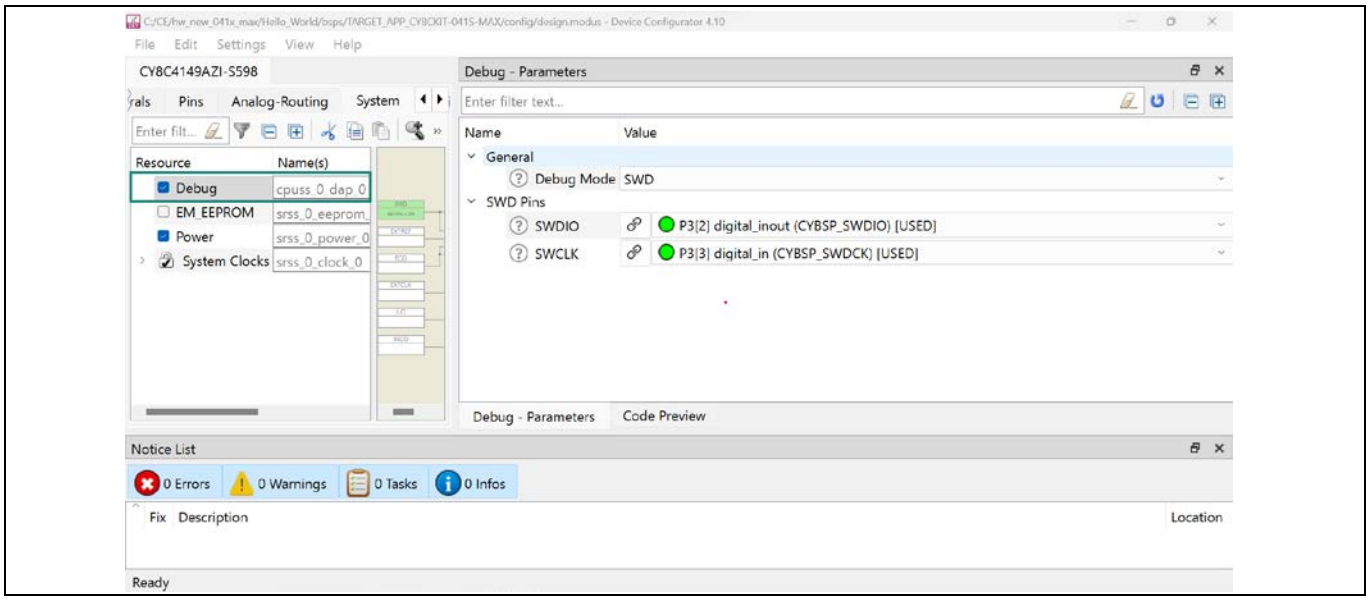


Figure 16　デフォルトでデバッグが有効

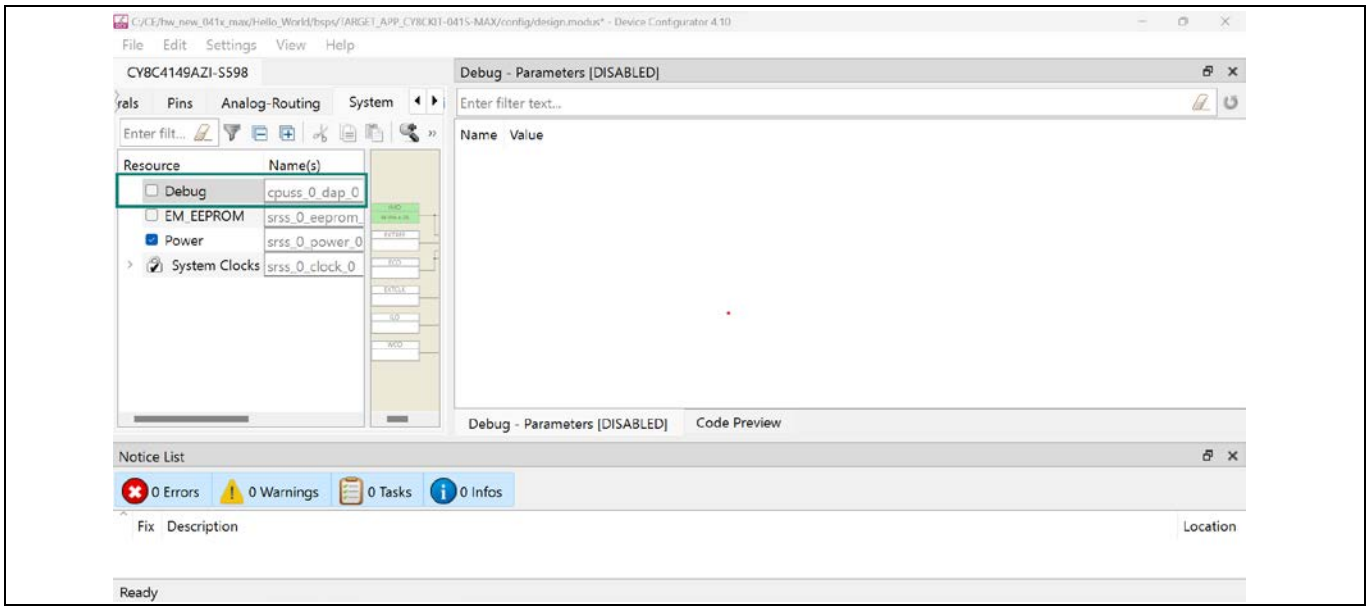


Figure 17　デバッグが無効

10 PSoC™ 4 デバイスコードを最適化するヒントとコツ

Table 19　デバッグオプションを無効にした場合のフラッシュメモリの節約

CY8CKIT-041S-MAX キットのフラッシュメモリの節約 (バイト単位)					
GCC_ARM		Arm®		IAR	
デバッグ	リリース	デバッグ	リリース	デバッグ	リリース
88	80	84	80	84	76

デバッグ オプションを無効にすると、Table 19 に記載されているバイト数を節約できます。

アプリケーションで未使用のピン、周辺機器、および DMA がデフォルトで有効になっている場合は、それらを無効にします。

例: I2C, ユーザー LED 2, および CAPSENSE™ 関連ピンは、CE230635 - PSoC™ 4: Hello world code example ではデフォルトで有効になっていますが、アプリケーションでは使用されません。そのため、これらの設定を無効にできます。

Figure 19 に示すように、MSC 0 および MSC 1 のチェック ボックスをオフにして、CAPSENSE™パーソナリティを無効にします。

Figure 21 に示すように、ピン設定で未使用の各ピンのチェック ボックスをオフにして、それらの未使用のピンを無効にします。

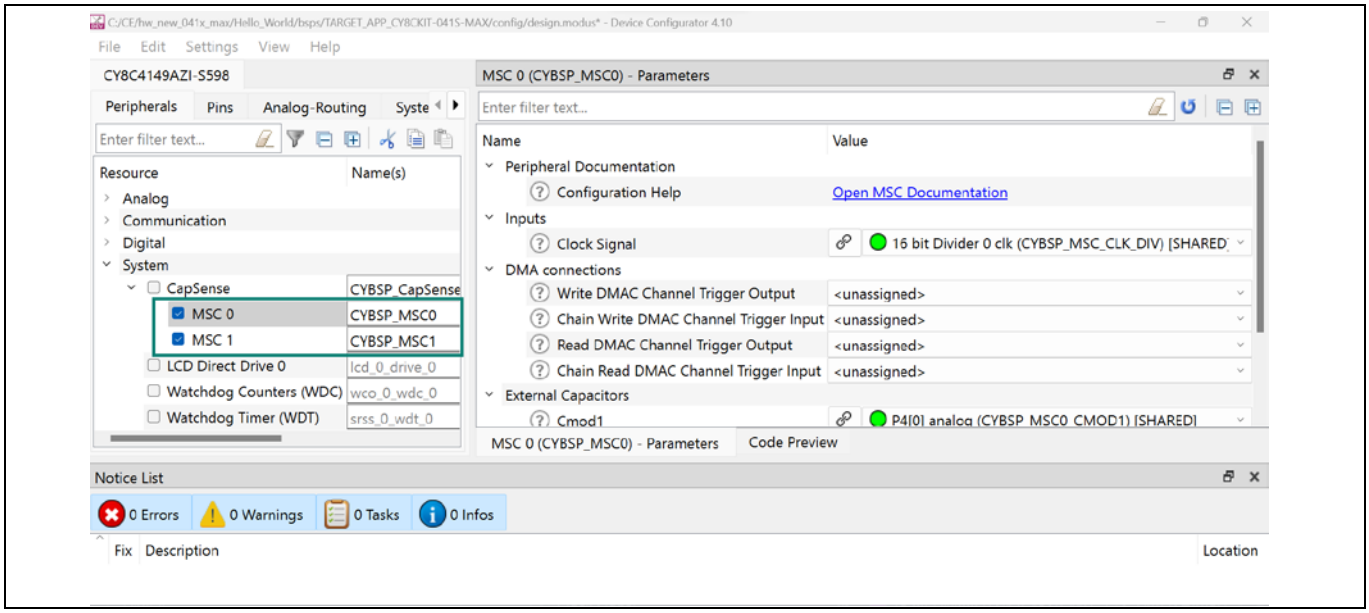


Figure 18　未使用のパーソナリティはデフォルトで有効

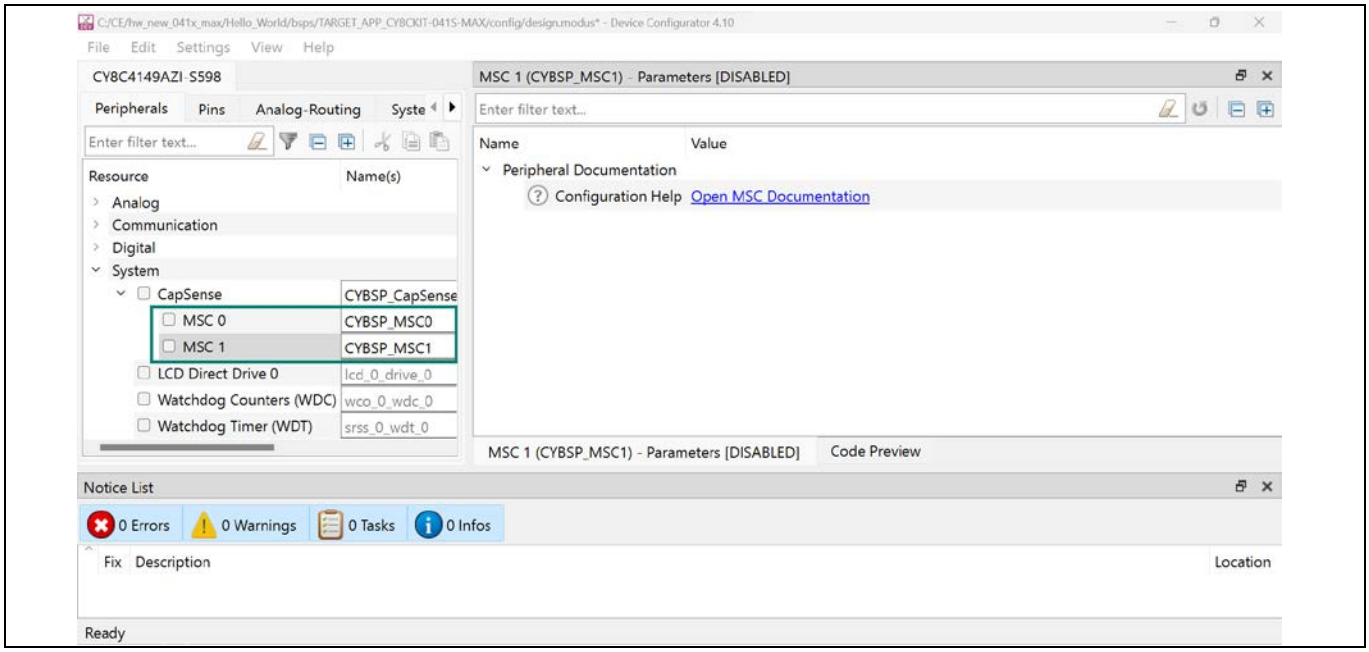


Figure 19　未使用のパーソナリティは無効

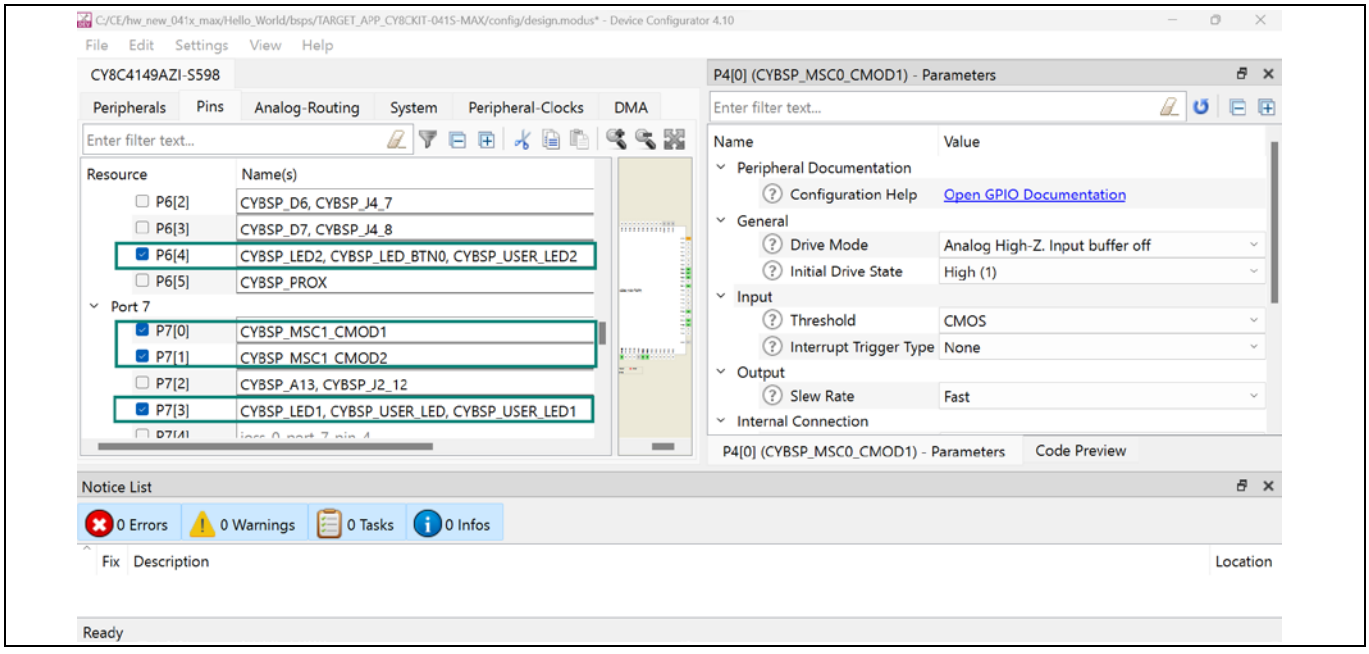


Figure 20　未使用のピンはデフォルトで有効

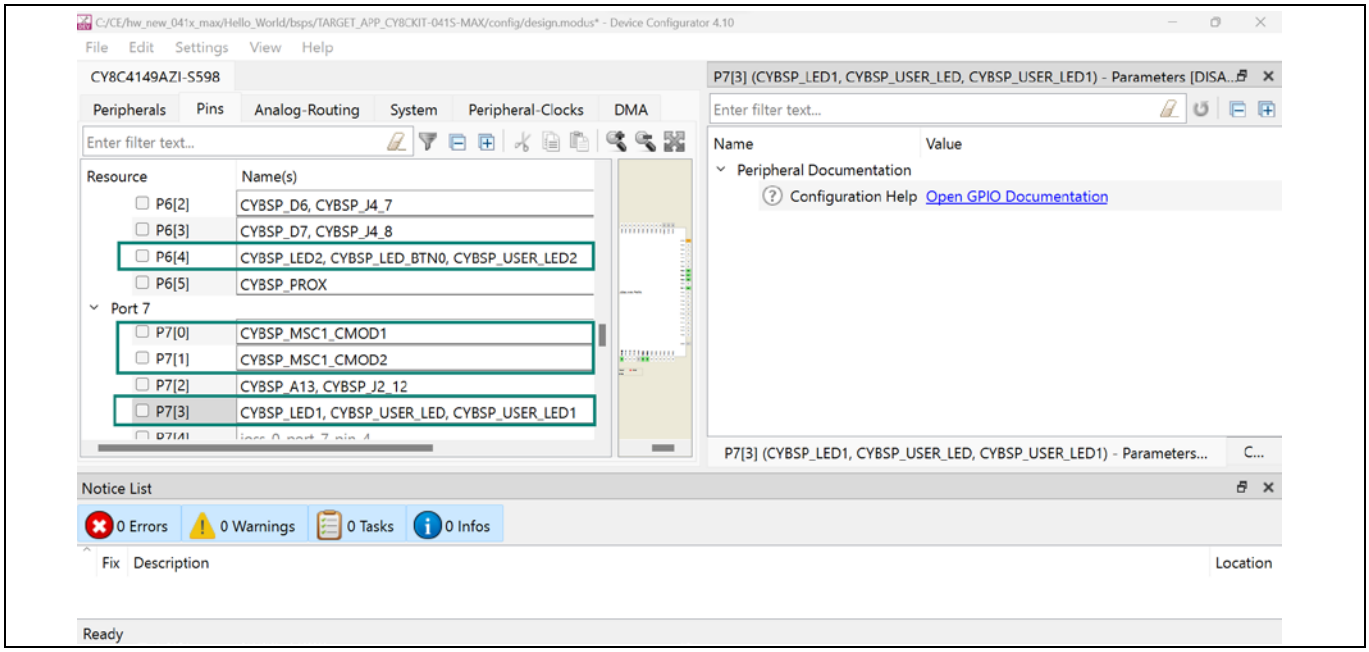


Figure 21　未使用のピンは無効

アプリケーションで未使用のピン, ペリフェラル, および DMA を無効にすることで、フラッシュバイトを数バイト節約できます。

10.5　CAPSENSE™ミドルウェアライブラリの最適化

ウォッチドッグタイマー値は CAPSENSE™ミドルウェアで `uint64_t` を使用しています。ただし、最初に 'periClkHz' を 'CY_CAPSENSE_CONVERSION_MEGA' 除算で実行するようにウォッチドッグタイマー値の計算操作を再配置して、`uint64_t` の使用を回避できます。

計算操作の再配置により、*cy_capsense_csd_v2.c* ファイルで `uint64_t` を `uint32_t` に置き換えることができます。

Table 20 CAPSENSE™ミドルウェア *cy_capsense_csd_v2.c* ファイル最適化のための現在のフローと回避策フロー

現在のフロー	`uint64_t isBusyWatchdogTimeUs;`
	
	`isBusyWatchdogTimeUs = (uint64_t)CY_CAPSENSE_CONFIGURED_FREQ_NUM * ptrWdCxt->maxRawCount;`
	`isBusyWatchdogTimeUs *= (uint64_t)modClkDivider * CY_CAPSENSE_CONVERSION_MEGA;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz;`
	
	`#if (CY_CAPSENSE_ENABLE == CY_CAPSENSE_MULTI_FREQUENCY_SCAN_EN)`
	`  isBusyWatchdogTimeUs  = (uint64_t)CY_CAPSENSE_MAX_SUPPORTED_FREQ_NUM * ptrWdCfg->numSns * ptrWdCfg->ptrWdContext->maxRawCount;`
	`#else`
	`  isBusyWatchdogTimeUs  = (uint64_t)ptrWdCfg->numSns * ptrWdCfg->ptrWdContext->maxRawCount;`
	`#endif`
	
	`isBusyWatchdogTimeUs *= (uint64_t)modClkDivider * CY_CAPSENSE_CONVERSION_MEGA;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz;`

回避策フロー	`uint32_t isBusyWatchdogTimeUs;`
	
	`isBusyWatchdogTimeUs = (uint32_t)CY_CAPSENSE_CONFIGURED_FREQ_NUM * ptrWdCxt->maxRawCount;`
	`isBusyWatchdogTimeUs *= (uint32_t)modClkDivider;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz / CY_CAPSENSE_CONVERSION_MEGA;`
	
	`#if (CY_CAPSENSE_ENABLE == CY_CAPSENSE_MULTI_FREQUENCY_SCAN_EN)`
	`  isBusyWatchdogTimeUs  = (uint32_t)CY_CAPSENSE_MAX_SUPPORTED_FREQ_NUM * ptrWdCfg->numSns * ptrWdCfg->ptrWdContext->maxRawCount;`
	`#else`
	`  isBusyWatchdogTimeUs  = (uint32_t)ptrWdCfg->numSns * ptrWdCfg->ptrWdContext->maxRawCount;`
	`#endif`
	
	`isBusyWatchdogTimeUs *= (uint32_t)modClkDivider;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz / CY_CAPSENSE_CONVERSION_MEGA;`

Table 21　回避策フローを使用したフラッシュメモリの節約

CY8CKIT-041S-MAX キットのフラッシュメモリの節約 (バイト単位)					
GCC_ARM		**Arm®**		**IAR**	
デバッグ	リリース	デバッグ	リリース	デバッグ	リリース
268	52	40	48	60	48

回避策フローを使用すると、Table 21 に記載されているバイト数を節約できます。

計算操作の再配置により、*cy_capsense_csx_v2.c* ファイル内の `uint64_t` を `uint32_t` に置き換えることができます。

Table 22　CAPSENSE™ミドルウェア *cy_capsense_csx_v2.c* ファイル最適化のための現在のフローと回避策フロー

現在のフロー	`uint64_t isBusyWatchdogTimeUs;`
	
	`#if (CY_CAPSENSE_ENABLE == CY_CAPSENSE_MULTI_FREQUENCY_SCAN_EN)`
	`  isBusyWatchdogTimeUs  = (uint64_t)totalSns * CY_CAPSENSE_MAX_SUPPORTED_FREQ_NUM * ptrWdCfg->ptrWdContext->resolution;`
	`#else`
	`  isBusyWatchdogTimeUs  = (uint64_t)totalSns * ptrWdCfg->ptrWdContext->resolution;`
	`#endif`
	
	`isBusyWatchdogTimeUs *= (uint64_t)snsClkDivider * modClkDivider * CY_CAPSENSE_CONVERSION_MEGA;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz;`

回避策フロー	`uint32_t isBusyWatchdogTimeUs;`
	
	`#if (CY_CAPSENSE_ENABLE == CY_CAPSENSE_MULTI_FREQUENCY_SCAN_EN)`
	`  isBusyWatchdogTimeUs  = (uint32_t)totalSns * CY_CAPSENSE_MAX_SUPPORTED_FREQ_NUM  * ptrWdCfg->ptrWdContext->resolution;`
	`#else`
	`  isBusyWatchdogTimeUs  = (uint32_t)totalSns * ptrWdCfg->ptrWdContext->resolution;`
	`#endif`
	
	`isBusyWatchdogTimeUs *= (uint32_t)snsClkDivider * modClkDivider;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz / CY_CAPSENSE_CONVERSION_MEGA;`

Table 23　回避策フローを使用したフラッシュメモリの節約

CY8CKIT-041S-MAX キットのフラッシュメモリの節約 (バイト単位)					
GCC_ARM		**Arm®**		**IAR**	
デバッグ	リリース	デバッグ	リリース	デバッグ	リリース
268	268	268	268	268	268

回避策フローを使用すると、Table 23 に記載されているバイト数を節約できます。

計算操作の再配置により、*cy_capsense_selftest_v2.c* ファイル内の `uint64_t` を `uint32_t` に置き換えることができます。

Table 24　CAPSENSE™ミドルウェア *cy_capsense_selftest_v2.c* ファイル最適化のための現在のフローと回避策フロー

現在のフロー	`uint64_t isBusyWatchdogTimeUs;`
	
	`isBusyWatchdogTimeUs  = (uint64_t)ptrScanConfig->convNum;`
	`isBusyWatchdogTimeUs *= (uint64_t)snsClkDivider * modClkDivider * CY_CAPSENSE_CONVERSION_MEGA;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz;`
	
	`isBusyWatchdogTimeUs  = (uint64_t)((uint64_t)0x01 << context->ptrBistContext->eltdCapResolution);`
	`isBusyWatchdogTimeUs *= (uint64_t)modClkDivider * CY_CAPSENSE_CONVERSION_MEGA;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz;`

回避策フロー	`Uint32_t isBusyWatchdogTimeUs;`
	
	`isBusyWatchdogTimeUs  = (uint32_t)ptrScanConfig->convNum;`
	`isBusyWatchdogTimeUs *= (uint32_t)snsClkDivider * modClkDivider;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz / CY_CAPSENSE_CONVERSION_MEGA;`
	
	`isBusyWatchdogTimeUs  = (uint32_t)((uint32_t)0x01 << context->ptrBistContext->eltdCapResolution);`
	`isBusyWatchdogTimeUs *= (uint32_t)modClkDivider;`
	`isBusyWatchdogTimeUs /= context->ptrCommonConfig->periClkHz / CY_CAPSENSE_CONVERSION_MEGA;`

Table 25　回避策フローを使用したフラッシュメモリの節約

CY8CKIT-041S-MAX キットのフラッシュメモリの節約 (バイト単位)					
GCC_ARM		Arm®		IAR	
デバッグ	リリース	デバッグ	リリース	デバッグ	リリース
128	52	52	72	64	60

回避策フローを使用すると、Table 25 に記載されているバイト数を節約できます。

10.6　重要な注意事項

フラッシュメモリの消費量と節約数はデバイスファミリによって異なり、コンパイラの最適化レベルにも依存します。

このドキュメントで説明するアプローチでは、デバッグ モードとリリースモードのすべての 3 つのコンパイラに対して ModusToolbox™ソフトウェア 3.1 のデフォルトのコンパイラ最適化レベルを使用して、フラッシュメモリの消費量を測定します。これについては、以下で説明します。

- GCC_ARM コンパイラバージョン v11.3.1 では、デバッグモードのコンパイラ最適化レベルは最小 (-Og) で、リリースモードのコンパイラ最適化レベルはサイズ (-Os) です。
- Arm®コンパイラバージョン v6.16 では、デバッグモードのコンパイラ最適化レベルは最小 (-O1) で、リリースモードのコンパイラ最適化レベルは小さいコードサイズ (-Oz) です。
- IAR コンパイラバージョン v9.30.1 では、デバッグモードのコンパイラ最適化レベルは最小 (-O1) で、リリースモードのコンパイラ最適化レベルはサイズ (-Ohs) です。

生成されたファイル (*cycfg_pins.c* など) に変更を加える場合は、設定をアプリケーションレベルのファイル (*main.c* など) にコピーして、生成されたファイルの代わりに使用するか、アプリケーション開発の最終段階で生成されたファイルでこれらの変更を行い、*design.modus* ファイルの保存中に発生したファイルの再生成によるファイルの変更の損失を回避してください。生成されたファイルに回避策を適用し、*design.modus* ファイルを変更して保存した場合は、これらの回避策を再度適用する必要があることに注意してください。詳細については、ModusToolbox™ツール パッケージ ユーザー ガイドを参照してください。

11 Cortex®-M3 ビットバンド (PSoC™ 5LP のみ)

アドレスマップ, Figure 2, および Figure 4 に示されているように、PSoC™ 5LP Cortex®-M3 には「ビットバンド」機能があり、**エイリアス領域**のアドレスにアクセスすると、対応するビットバンド領域でビットレベルのアクセスが行われます。これにより、領域の最初の 1 MB の 1 つのビットをすばやく設定, クリア, またはテストできます。特定のバイトアドレスの特定のビットの場合、対応するエイリアスアドレスの式は次のとおりです。

alias_address = 0x22000000 + 32 * (byte_address – 0x20000000) + 4 * bit_number

したがって、例えば、アドレス 0x20000001 のビット 5 を設定するためには、アドレスに「1」を書き込んでください。

0x22000000 + 32 * 1 + 4 * 5 = 0x22000034

同様に、ビットをクリアするためには、エイリアスアドレスに「0」を書き込んでください。ビットをテストするためには、エイリアスアドレスを読み出し、ビット 0 をテストします。

Note: SRAM 領域に加えて、Cortex®-M3 は、すべての PSoC™5LP レジスタが配置されている周辺領域のビットバンドをサポートします。ただし、周辺領域のビットバンドは PSoC™5LP ではサポートされていません。周辺領域のビットバンドエイリアス領域 (0x42000000～0x43FFFFFF) に書き込むと、PSoC™5LP レジスタで予期しない結果が生じる可能性があるため、推奨されません。

変数でビットバンド機能を使用するには、まずコードと変数の配置で説明されている手法を使用して、変数を上位 SRAM に配置します。次に、計算するマクロを定義し、ビットバンドエイリアス領域で対応するアドレスを使用します。

```
        #define  BIT_BAND_ALIAS_BASE  0x22000000
        /* 'byte' should be a number 0x20000000 to 0x200FFFFF
           'bit' should be a number 0 to 7 */
        #define  BIT_BAND_ALIAS_ADDR(byte, bit)  ((BIT_BAND_ALIAS_BASE + \
                                                  32 * ((uint32)(byte) – \
                                                  CYREG_SRAM_DATA_MBASE) + \
                                                  4 * (uint8)(bit))
        /* 'a' should be an address (uint32 *) */
        #define  GET_BIT(a, bit)        *(uint32 *)BIT_BAND_ALIAS_ADDR(a, bit)
        /* 'val' should be 0 or 1 */
        #define  SET_BIT(a, bit, val)   GET_BIT(a, bit) = (uint32)(val)
#define TEST_BIT(a, bit, val)  (GET_BIT(a, bit) == (uint32)(val))
```

次に、マクロを使用してビットを設定またはテストできます。

```
        SET_BIT(&foo, 5, 1); /* set bit 5 of foo */
        if (TEST_BIT(&foo, 5, 1)) { ... } /* test bit 5 */
```

一般に、Table 26 に示すように、変数の読み出し, 変更, 書き込みよりも、ビットバンド手法を使用してビットを設定またはクリアする方が効率的です。ビットバンドを使用すると、読み出し, 変更, 書き込みのサイクルが CPU によって内部的に実行されるため、命令が 1 つ節約されます。

11 Cortex®-M3 ビットバンド (PSoC™ 5LP のみ)

Table 26　ビットバンドと直接セットのアセンブリ言語

C コード	アセンブラコード

`/* direct set bit */`
`foo |= (1 << 5);`
	`/* R3 = address of foo */`
	`ldr  r2, [r3]`
	`orr  r2, r2, #32`
	`str  r2, [r3]`

`/* use bit band */`
`SET_BIT(&foo, 5, 1);`
	`/* R3 = bit band alias address for foo bit 5 */`
	`mov  r2, #1`
	`str  r2, [r3]`

ビットバンディングを使用してビットを切り替える効率的な方法はありません。次の操作を実行できます。

```
SET_BIT(&foo, 5, GET_BIT(&foo, 5) ^ 1);
```

ただし、次のようにするとより簡単かつ効率的です。

```
foo ^= (1 << 5);
```

12 DMA アドレス

ここでは、PSoC™ 5LP または PSoC™ 4 のダイレクトメモリアクセス (DMA) コントローラの使用方法を理解していることを前提としています。DMA コントローラは、CPU の介入なしにソースから宛先にデータを転送できます。これにより、DMA がデータ転送を行っている間に CPU が他のタスクを処理できるため、「マルチプロセッシング」環境が実現します。

DMA コントローラは非常に柔軟で、PSoC™メモリとオンチップ周辺機器 (ADC, DAC, デジタルフィルタブロック (DFB) (PSoC™ 5LP), USB, UART, SPI など) 間の複雑なデータ転送を実行できます。PSoC™ 5LP には 24 個の独立した DMA チャネルがあり、PSoC™ 4 には最大 32 個の DMA チャネルがあります。詳細については、AN52705 - PSoC™ DMA 入門を参照してください。

PSoC™ 5LP では、DMA は Cortex®-M3 S バス (Figure 4) を CPU と共有します。ただし、S バスはコード領域にアクセスしないため、DMA はコード SRAM (0x1FFF8000 ～ 0x1FFFFFFF) に直接アクセスできません。PSoC™ 5LP は、Figure 22 に示すように、DMA が対応するアドレス 0x20008000 ～ 0x2000FFFF にアクセスしてコード SRAM にアクセスできるように再マッピングを実装することでこれを処理します。

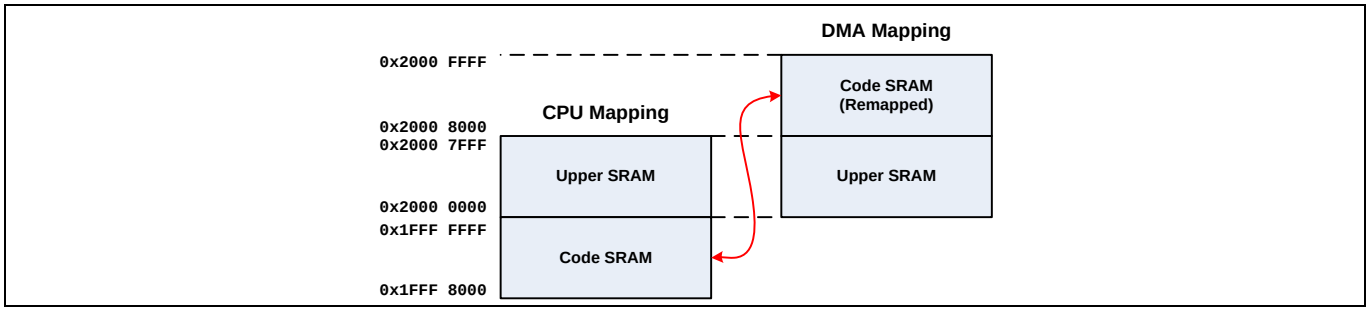


Figure 22 コード SRAM の DMA 再マッピング

DMA は 16 ビットのサブシステムであるため、アドレスを増分すると、ロールオーバーにより下位 16 ビットのみが増分されます。したがって、0x2000FFFF (0x1FFFFFFF にマップされます) の次の DMA アドレスは 0x20000000 です。つまり、SRAM は DMA 用の連続した 64 KB のメモリブロックとして機能します。これは、SRAM が常に 0x20000000 を中心とするため、64K 未満の SRAM を搭載したデバイスにも当てはまります。

DMA チャネルの設定に使用される PSoC™ Creator DMA コンポーネント API 関数が、再マッピングを処理します。API を使用しない場合は、実際のアドレスに関係なく、SRAM の DMA アドレスの上位 16 ビットを常に 0x2000 に設定してください。

13 まとめ

このアプリケーションノートでは、PSoC™ 4 および PSoC™ 5LP の Cortex® CPU の C コードの効率を高めるためのいくつかの方法を紹介しました。PSoC™ Creator および ModusToolbox™ソフトウェアでサポートされている GCC, MDK, および IAR コンパイラは、これらの手法を使用しなくてもほとんどのアプリケーションで適切に機能します。これらの手法は、コードサイズまたは実行速度の要件を満たすための特別な問題で必要です。

紹介されている方法 (順不同) は次のとおりです。

- 関数の引数の数を 4 以下に制限します。関数の引数と結果を参照してください。
- グローバル変数と静的変数の数を最小限に抑えます。これはコーディングのベストプラクティスであるだけでなく、アドレスロード操作の数を減らすことでコードサイズを縮小できます。グローバル変数と静的変数を参照してください。
- 重要なセクションで効率を最大化するために、インラインアセンブラまたは組込みアセンブラを使用します。C とアセンブラコードの混合を参照してください。この手法または組込み関数を使用して、特に Cortex®-M3 で特別な命令を利用することもできます。特殊機能命令を参照してください。効率的なアセンブラコードの書き方の例については、付録: コンパイラ出力の詳細も参照してください。
- 標準コンパイラライブラリは大量のメモリを使用する可能性があるため、使用には注意してください。代わりにインラインコードの使用を検討してください。コンパイラライブラリを参照してください。
- 構造体を使用する場合は、パックするかアンパックするかに注意してください。それぞれに利点と欠点があります。パックおよびアンパックの構造体を参照してください。
- 速度が重要なコードを SRAM に配置します。コードと変数の配置を参照してください。この手法を使用した場合、速度の向上が実現されない可能性があることに注意してください。
- PSoC™ 5LP Cortex®-M3 のビットバンド機能を利用するために変数を配置します。Cortex®-M3 ビットバンド (PSoC™ 5LP のみ) および例を参照してください。

13.1 PSoC™デバイスのすべてのリソースの使用

コードサイズを削減する方法がもう 1 つあります。これは、PSoC™が柔軟なデバイスとして設計されており、プログラム可能なアナログブロックとデジタルブロックにカスタム機能を構築できるという事実に基づいています。例えば、PSoC™ 5LP には、「コプロセッサ」として機能できる次の周辺機器があります。

- DMA コントローラ。最も一般的な CPU アセンブラ命令は MOV, LDR, および STR であるため、CPU はバイトの移動だけで多くのサイクルを費やします。代わりに DMA コントローラにその処理を実行させます。
- デジタルフィルタブロック (DFB) – 高度な 24 ビット積和計算機
- ユニバーサルデジタルブロック (UDB)。最大 24 個の UDB があり、各 UDB には 8 ビットのデータパスがあり、加算, 減算, ビット単位の演算, シフト, 巡回冗長検査 (CRC) を実行できます。データパスを連結してワード全体の計算を行うことができます。CPU の計算をデータパスにオフロードすることを検討してください。
- UDB には、ステート マシンの構築に使用できるプログラマブルロジックデバイス (PLD) もあります。ルックアップテーブル (LUT) コンポーネントデータシートを参照してください。LUT は、C のスイッチ/ケース ステートメントを使用して CPU でステートマシンをプログラミングする代わりに、効果的な方法です。

13 まとめ

- ADC, DAC, コンパレータ, オペアンプ, およびプログラマブルスイッチドキャパシタ/連続時間 (SC/CT) ブロックを含むアナログコンポーネント。これらからプログラマブルゲインアンプ (PGA), トランスインピーダンスアンプ (TIA), ミキサーを作成できます。デジタルドメインではなくアナログドメインで処理することを検討してください。

PSoC™ Creator は、これらの周辺機器にさまざまな機能を実装するための多数のコンポーネントを提供しています。同様に、ModusToolbox™は、周辺機器に機能を実装するための多くのリソースを提供しています。これにより、CPU から多くの機能をオフロードして、1 つのチップで効果的なマルチプロセッシングシステムを開発できます。これにより、コードサイズを縮小できるだけでなく、CPU が実行する必要があるタスクの数を減らすことで、CPU 速度を下げ、電力を削減できます。

例えば、PSoC™ 5LP を使用すると、多重化された ADC 入力を制御し、DMA とインターフェースしてデータを SRAM に保存し、CPU をまったく使用しない高度なアナログデータ収集システムを作成するようにデジタルシステムを設計できます。

インフィニオンは、PSoC™周辺機器に関する広範なアプリケーションノートのサポート、およびデバイスデータシートとテクニカルリファレンスマニュアル (TRM) の詳細なデータを提供しています。詳細については、参考資料を参照してください。

14 付録: コンパイラ出力の詳細

ここでは、PSoC™ Creator でサポートされている両方のコンパイラ (GCC および MDK) と両方の PSoC™ CPU (Cortex®-M0/M0+ および Cortex®-M3) のアセンブラ出力を、最適化ありと最適化なしの両方で詳細に示します。詳細は、次のように構成された複数の表に示されています。

- Cortex®-M3 CPU 用 GCC コンパイラのコンパイラ出力の詳細
- Cortex®-M3 CPU 用 GCC コンパイラのコンパイラ出力の詳細
- Cortex®-M0/M0+ CPU 用 GCC コンパイラのコンパイラ出力の詳細
- Cortex®-M0/M0+ CPU 用 GCC コンパイラのコンパイラ出力の詳細
- Cortex®-M3 CPU 用 MDK コンパイラのコンパイラ出力の詳細
- Cortex®-M0/M0+ CPU 用 MDK コンパイラのコンパイラ出力の詳細
- Cortex®-M3 CPU 用 IAR コンパイラのコンパイラ出力の詳細
- Cortex®-M3 CPU 用 IAR コンパイラのコンパイラ出力の詳細
- Cortex®-M0/M0+ CPU 用 IAR コンパイラのコンパイラ出力の詳細
- Cortex®-M0/M0+ CPU 用 IAR コンパイラのコンパイラ出力の詳細

C コードをコンパイルしたときに得られるものとまったく同じではないかもしれませんが、表のアセンブラコードは、コードに組み込むことができる便利な例として役立ちます。詳細については、C とアセンブラコードの混合を参照してください。

表を生成するために使用されたテストプログラムは、コンパイラテストプログラムにあります。

14.1 アセンブラの例、Cortex®-M3 用 GCC

14.1.1 Cortex®-M3 用 GCC、なし, サイズ, および速度最適化

Table 27 に、Cortex®-M3 用の GCC コンパイラについて、さまざまな最適化オプションのコンパイラ出力例を示します。これらの例は、コンパイラによって生成された.lst ファイルから抽出されたものです。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

Table 27 Cortex®-M3 CPU 用 GCC コンパイラのコンパイラ出力の詳細

C コード	GCC, Cortex®-M3, 最適化なし	GCC, Cortex®-M3, 最適化あり	GCC, Cortex®-M3, 速度最適化

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`bl LCD_Start`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`; conditional flags are NOT`
	`; updated by mov`
	`mov r0, #80`
	`bl LCD_PrintInt8`
		`; R0 = first argument`
		`; conditional flags ARE updated`
		`; by movs`
		`movs r0, #80`
		`bl LCD_PrintInt8`
			`; same as for size optimization`

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`mov r0, #0`
		`; R0 = first argument`
		`; R1 = second argument`
		`movs r1, #2`
			`; same as for size optimization`

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, 最適化なし	GCC, Cortex®-M3, 最適化あり	GCC, Cortex®-M3, 速度最適化
	mov r1, #2 bl LCD_Position	movs r0, #0 bl LCD_Position	

// For loop:
void ForLoop(uint8 i)
{
for(i = 0; i < 10; i++)
{
LCD_PrintInt8(i);
}
}
	; function prolog
	; i is saved on the stack
	sub sp, sp, #16
	add r7, sp, #0
	mov r3, r0
	strb r3, [r7, #15]
	
	; i = 0
	mov r3, #0
	strb r3, [r7, #15]
	b .L2
	
	; do the function call with
	; i as the argument in R0
	.L3:
	ldrb r3, [r7, #15]
	uxth r3, r3 ; sign extend
	mov r0, r3
	bl LCD_PrintInt8
	
	; i++
	ldrb r3, [r7, #15]
	add r3, r3, #1
	strb r3, [r7, #15]
	
	; check i 10, by comparing
	; it with 9
	.L2:
	ldrb r3, [r7, #15]
	cmp r3, #9
	bls .L3
	
	; function epilog
	add r7, r7, #16
	mov sp, r7
	pop {r7, pc} ; return
		; function prolog
		push {r4, lr}
		
		; R4 = i
		movs r4, #0
		
		.L2:
		; do the function call with
		; i as the argument in R0
		mov r0, r4 ; sign extend
		adds r4, r4, #1 ; i++
		uxtb r4, r4
		bl LCD_PrintInt8
		
		; check i not equal to 10
		cmp r4, #10
		bne .L2
		
		; function epilog
		pop {r4, pc} ; return
			; function prolog
			push {r3, lr}
			
			; unroll the loop
			; do the function call
			; 10 times
			; i as the argument in R0
			movs r0, #0
			bl LCD_PrintInt8
			
			movs r0, #1
			bl LCD_PrintInt8
			
			. . .
			
			movs r0, #9
			pop {r3, lr}
			; function returns back to
			; caller of this function
			b LCD_PrintInt8

// While loop
// i is type automatic, see
// Accessing Automatic Variables
// for details
	; prolog not shown
	; i = 0
	mov r3, #0
	strb r3, [r7, #7]
	b .L5
		; prolog not shown
		; i = 0
		movs r4, #0
		
		.L6:
		mov r0, r4
			; function prolog
			push {r3, lr}
			
			; unroll the loop
			; do the function call

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, 最適化なし	GCC, Cortex®-M3, 最適化あり	GCC, Cortex®-M3, 速度最適化

uint8 i = 0;
 	
while (i < 10)
{
  LCD_PrintInt8(i);
  i++;
}
	.L6:
	; LCD_PrintInt8(i)
	ldrb r3, [r7, #7]
	uxth r3, r3
	mov r0, r3
	bl LCD_PrintInt8
	
	; i++
	ldrb r3, [r7, #7]
	add r3, r3, #1
	strb r3, [r7, #7]
	
	.L5:
	; while(i 10)
	ldrb r3, [r7, #7]
	cmp r3, #9
	bls .L6
	
	; epilog not shown
		adds r4, r4, #1 ;
		i++
		uxtb r4, r4
		; LCD_PrintInt8(i)
		bl LCD_PrintInt8
		
		; check i not equal
		to 10
		cmp r4, #10
		bne .L6
		
		; epilog not shown
			; 10 times
			; i as the argument
			in R0
			movs r0, #0
			bl LCD_PrintInt8
			
			movs r0, #1
			bl LCD_PrintInt8
			
			. . .
			
			movs r0, #9
			pop {r3, lr}
			; function returns
			back to
			; caller of this
			function
			b LCD_PrintInt8

// Conditional
statement
void Conditional(uint8
i, uint8 j)
{
  if(j == 1)
  {
  LCD_PrintInt8(i);
  }
  else
  {
  LCD_PrintInt8(i +
1);
  }
}
	; prolog not shown
	; if(j == 1)
	ldrb r3, [r7, #6]
	cmp r3, #1
	bne .L5
	
	; LCD_PrintInt8(i)
	ldrb r3, [r7, #7]
	uxth r3, r3
	mov r0, r3
	bl LCD_PrintInt8
	b .L4
	
	.L5:
	; LCD_PrintInt8(i +
	1)
	ldrb r3, [r7, #7]
	uxth r3, r3
	add r3, r3, #1
	uxth r3, r3
	mov r0, r3
	bl LCD_PrintInt8
	
	.L4:
	; epilog not shown
		; no prolog
		; if(j == 1)
		cmp r1, #1
		beq .L10
		
		; LCD_PrintInt8(i)
		adds r0, r0, #1
		uxtb r0, r0
		
		.L10:
		; function returns
		back to
		; caller of this
		function
		b LCD_PrintInt8
			; same as for size
			optimization

// Switch case
statements
void SwitchCase(uint8
j)
{
  switch(j)
	; prolog not shown
	; switch(j)
	ldrb r3, [r7, #7]
	cmp r3, #0
	beq .L9
	cmp r3, #1
		; no prolog
		; switch(j)
		cbz r0, .L13
		cmp r0, #1
		bne .L15
			; no prolog
			; switch(j)
			cbnz r0, .L12
			movs r0, #1 ; case 0
			; no epilog
			b LCD_PrintInt8

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, 最適化なし	GCC, Cortex®-M3, 最適化あり	GCC, Cortex®-M3, 速度最適化

{
case 0:
LCD_PrintInt8(1);
break;
 	
case 1:
LCD_PrintInt8(2);
break;
 	
default:
LCD_PrintInt8(0);
break;
}
}
	beq .L10
	b .L12
	
	.L9: ; case 0
	mov r0, #1
	bl LCD_PrintInt8
	b .L7 ; break
	
	.L10: ; case 1
	mov r0, #2
	bl LCD_PrintInt8
	b .L9 ; break
	
	.L12: ; default
	mov.w r0, #0
	bl LCD_PrintInt8
	nop ; break
	
	.L7:
	; epilog not shown
		movs r0, #2 ; case 1
		b .L16
		
		.L13:
		movs r0, #1 ; case 0
		b .L16
		
		.L15:
		movs r0, #0 ; default
		
		.L16:
		; no epilog
		b LCD_PrintInt8
			
			.L12:
			cmp r0, #1
			beq .L13
			movs r0, #0 ; default
			; no epilog
			b LCD_PrintInt8
			
			movs r0, #2 ; case 1
			; no epilog
			b LCD_PrintInt8

// Ternary operator
void Ternary(uint8 i)
{
LCD_PrintInt8(
(i == 1) ? 80 : 100);
}
	; prolog not shown
	; check value of i
	ldrb r3, [r7, #7]
	cmp r3, #1
	bne .L17
	
	mov r3, #80
	b .L18
	
	.L17:
	mov r3, #100
	
	.L18:
	mov r0, r3
	bl LCD_PrintInt8
	; epilog not shown
		; no prolog
		; check value of i
		cmp r0, #1
		
		; “ite” stands for if-then-
		; else instruction
		; “ne” condition checks
		; if the previous compare
		; instruction has cleared the
		; “equal to” flag
		ite ne
		
		; mov if the result of the
		; previous “ite” instruction is
		; “not equal”
		movne r0, #100
		
		; mov if the result of the
		; previous “ite” instruction is
			; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, 最適化なし	GCC, Cortex®-M3, 最適化あり	GCC, Cortex®-M3, 速度最適化
		; “equal” moveq r0, #80 ; no epilog b LCD_PrintInt8	

// Addition operation
int DoAdd(int x, int y)
{
return x + y;
}
	; prolog not shown
	ldr r2, [r7, #4]
	ldr r3, [r7, #0]
	adds r3, r2, r3
	mov r0, r3 ; return value
	; epilog not shown
		; no prolog
		adds r0, r0, r1
		bx lr ; return with result
			; same as for size optimization

// Subtraction operation
int DoSub(int x, int y)
{
return x - y;
}
	; prolog not shown
	ldr r2, [r7, #4]
	ldr r3, [r7, #0]
	subs r3, r2, r3
	mov r0, r3 ; return value
	; epilog not shown
		; no prolog
		subs r0, r0, r1
		bx lr ; return with result
			; same as for size optimization

// Multiplication
int DoMul(int x, int y)
{
return x * y;
}
	; prolog not shown
	ldr r3, [r7, #4]
	ldr r2, [r7, #0]
	mul r3, r2, r3
	mov r0, r3 ; return value
	; epilog not shown
		; no prolog
		muls r0, r1, r0
		bx lr ; return with result
			; same as for size optimization

// Division
int DoDiv(int x, int y)
{
return x / y;
}
	; prolog not shown
	ldr r2, [r7, #4]
	ldr r3, [r7, #0]
	sdiv r3, r2, r3
	mov r0, r3 ; return value
	; epilog not shown
		; no prolog
		sdiv r0, r0, r1
		bx lr ; return with result
			; same as for size optimization

// Modulo operator
int DoMod(int x, int y)
{
return x % y;
}
	; prolog not shown
	ldr r3, [r7, #4]
	ldr r2, [r7, #0]
	; truncated quotient
	sdiv r2, r3, r2
	; quotient * divisor
	ldr r1, [r7,#0]
	mul r2, r1, r2
	; remainder = dividend -
	; (quotient * divisor)
	subs r3, r3, r2
	mov r0, r3 ; return value
	; epilog not shown
		; no prolog
		sdiv r3, r0, r1
		; multiply and subtract instruction
		; implements remainder =
		; dividend - (quotient * divisor)
		mls r0, r3, r1, r0
		bx lr ; return with result
			; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, 最適化なし	GCC, Cortex®-M3, 最適化あり	GCC, Cortex®-M3, 速度最適化

`// Pointer`
`void Pointer(uint8 x, uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
`  LCD_PrintInt8(*ptr);`
`}`
	`; *ptr = *ptr + x`
	`ldr  r3, [r7, #0] ; ptr`
	`ldrb  r2, [r3, #0]`
	`ldrb  r3, [r7, #7] ; x`
	`adds  r3, r2, r3`
	`uxtb  r2, r3`
	`ldr  r3, [r7, #0] ; ptr`
	`strb  r2, [r3, #0]`
	
	`; ptr++`
	`ldr  r3, [r7, #0] ; ptr`
	`add  r3, r3, #1`
	`str  r3, [r7, #0]`
	
	`; LCD_PrintInt8(*ptr)`
	`ldr  r3, [r7, #0]`
	`ldrb  r3, [r3, #0]`
	`mov  r0, r3`
	`bl  LCD_PrintInt8`
		`; *ptr = *ptr + x`
		`ldrb  r3, [r1, #0] ; R1 = ptr`
		`adds  r0, r0, r3  ; R0 = x`
		`strb  r0, [r1, #0]`
		
		`; ptr++`
		`; LCD_PrintInt8(*ptr)`
		`ldrb  r0, [r1, #1]`
		`b  LCD_PrintInt8`
			`; same as for size optimization`

`// Function pointer`
`void FuncPtr(uint8 x, void *fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; (*fptr)(x)`
	`ldrb  r2, [r7, #7] ; x`
	`ldr  r3, [r7, #0] ; fptr`
	`mov  r0, r2`
	`; in a blx instruction, the`
	`; LS bit of the register`
	`; must be 1 to keep the CPU`
	`; in Thumb mode, or an`
	`; exception occurs`
	`blx  r3`
		`; (*fptr)(x)`
		`; in a blx instruction, the LS`
		`; bit of the register must be`
		`; 1 to keep the CPU in Thumb`
		`; mode, or an exception occurs`
		`blx  r1`
			`; same as for size optimization`

`// Packed structures`
`struct FOO_P`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`} __attribute__ ((packed));`
 	
`extern struct FOO_P`
	`; membera = 5`
	`movw  r3, #:lower16:myfoo_p`
	`movt  r3, #:upper16:myfoo_p`
	`mov  r2, #5`
	`strb  r2, [r3, #0]`
	`; memberb = 10`
	`movw  r3, #:lower16:myfoo_p`
	`movt  r3,`
		`ldr  r3, [pc, .L28]`
		`movs  r2, #5`
		`movs  r0, #10`
		`strb  r2, [r3, #0] ; membera = 5`
		`strb  r0, [r3, #1] ; memberb = 10`
		`movs  r2, #0`
		`movs  r1, #15`
		`movs  r0, #20`
		`strb  r1, [r3, #2] ;`
			`movw  r3, #:lower16:myfoo_p`
			`movt  r3, #:upper16:myfoo_p`
			`movs  r1, #5`
			`movs  r0, #10`
			`movs  r2, #0`
			`strb  r1, [r3, #0] ; membera = 5`
			`strb  r0, [r3, #1] ; memberb = 10`

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, 最適化なし	GCC, Cortex®-M3, 最適化あり	GCC, Cortex®-M3, 速度最適化

myfoo_p;
 	
void
PackedStruct(void)
{
  myfoo_p.membera =
5;
  myfoo_p.memberb =
10;
  myfoo_p.memberc =
15;
  myfoo_p.memberd =
20;
}
	#:upper16:myfoo_p
	mov r2, #10
	strb r2, [r3, #1]
	; memberc = 15
	movw r3,
	#:lower16:myfoo_p
	movt r3,
	#:upper16:myfoo_p
	mov r2, #0
	orr r2, r2, #15
	strb r2, [r3, #2]
	mov r2, #0
	strb r2, [r3, #3]
	mov r2, #0
	strb r2, [r3, #4]
	mov r2, #0
	strb r2, [r3, #5]
	; memberd = 20
	movw r3,
	#:lower16:myfoo_p
	movt r3,
	#:upper16:myfoo_p
	mov r2, #0
	orr r2, r2, #20
	strb r2, [r3, #6]
	mov r2, #0
	strb r2, [r3, #7]
		memberc = 15
		strb r2, [r3, #3]
		strb r2, [r3, #4]
		strb r2, [r3, #5]
		strb r0, [r3, #6] ;
		memberd = 20
		strb r2, [r3, #7]
		bx lr
		
		.L28:
		.word myfoo_p
			movs r1, #15
			movs r0, #20
			strb r1, [r3, #2] ;
			memberc = 15
			strb r2, [r3, #3]
			strb r2, [r3, #4]
			strb r2, [r3, #5]
			strb r0, [r3, #6] ;
			memberd = 20
			strb r2, [r3, #7]
			bx lr

// unpacked structures
struct FOO
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
};
 	
extern struct FOO
myfoo;
 	
void
PackedStruct(void)
{
  myfoo.membera = 5;
  myfoo.memberb = 10;
  myfoo.memberc = 15;
  myfoo.memberd = 20;
}
	; membera = 5
	movw r3,
	#:lower16:myfoo
	movt r3,
	#:upper16:myfoo
	mov r2, #5
	strb r2, [r3, #0]
	; memberb = 10
	movw r3,
	#:lower16:myfoo
	movt r3,
	#:upper16:myfoo
	mov r2, #10
	strb r2, [r3, #1]
	; memberc = 15
	movw r3,
	#:lower16:myfoo
	movt r3,
	#:upper16:myfoo
	mov r2, #15
	str r2, [r3, #4]
	; memberd = 20
	movw r3,
		ldr r3, [pc, .L31]
		movs r2, #5
		strb r2, [r3, #0] ;
		membera = 5
		movs r0, #10
		movs r1, #15
		movs r2, #20
		strb r0, [r3, #1] ;
		memberb = 10
		str r1, [r3, #4] ;
		memberc = 15
		strh r2, [r3, #8] ;
		memberd = 20
		bx lr
		
		.L31:
		.word myfoo
			movw r3,
			#:lower16:myfoo
			movt r3,
			#:upper16:myfoo
			movs r2, #5
			strb r2, [r3, #0] ;
			membera = 5
			movs r0, #10
			movs r1, #15
			movs r2, #20
			strb r0, [r3, #1] ;
			memberb = 10
			str r1, [r3, #4] ;
			memberc = 15
			strh r2, [r3, #8] ;
			memberd = 20
			bx lr

C コード	GCC, Cortex®-M3, 最適化なし	GCC, Cortex®-M3, 最適化あり	GCC, Cortex®-M3, 速度最適化

	`#:lower16:myfoo`
	`movt r3,`
	`#:upper16:myfoo`
	`mov r2, #20`
	`strh r2, [r3, #8]`		

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`bl LCD_Start`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`; conditional flags are NOT`
	`; updated by mov`
	`mov r0, #80`
	`bl LCD_PrintInt8`
		`; R0 = first argument`
		`; conditional flags ARE updated`
		`; by movs`
		`movs r0, #80`
		`bl LCD_PrintInt8`
			`; same as for size optimization`

14.1.2 Cortex®-M3 用 GCC、デバッグ, 最小, および高最適化

Table 28 に、Cortex-M3 の GCC コンパイラについて、さまざまな最適化オプションのコンパイラ出力の例を示します。これらの例は、コンパイラによって生成された.lst ファイルから抽出されたものです。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

Table 28 Cortex®-M3 CPU 用 GCC コンパイラのコンパイラ出力の詳細

C コード	GCC, Cortex®-M3, デバッグ最適化	GCC, Cortex®-M3, 最小最適化	GCC, Cortex®-M3, 高最適化

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`bl LCD_Start`
		`; same as for Debug optimization`	`; same as for Debug optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`; conditional flags are NOT`
	`; updated by mov`
	`r0, #128`
	`bl LCD_PrintInt8`
		`; same as for Debug optimization`	`; same as for Debug optimization`

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`mov r1, #2`
	`mov r0, #0`
	`bl LCD_Position`
		`; same as for Debug optimization`	`; same as for Debug optimization`

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, デバッグ最適化	GCC, Cortex®-M3, 最小最適化	GCC, Cortex®-M3, 高最適化

// For loop:
void ForLoop(uint8 i)
{
  for(i = 0; i < 10; i++)
  {
  LCD_PrintInt8(i);
  }
}
	; function prolog
	push {r4, lr}
	
	; i = 0
	mov r4, #0
	b .L2
	
	; do the function call with
	; i as the argument in R0
	.L3:
	mov r0, r4
	bl LCD_PrintInt8
	adds r4, r4, #1
	uxtb r4, r4 ; sign extend
	
	; check i 10, by comparing
	; it with 9
	.L2:
	cmp r4, #9
	bls .L3
	
	pop {r4, pc}; return
		; function prolog
		push {r4, lr}
		
		; R4 = i
		movs r4, #0
		
		.L2:
		; do the function call with
		; i as the argument in R0
		uxtb r0, r4 ; sign extend
		bl LCD_PrintInt8
		adds r4, r4, #1 ; i++
		
		; check i not equal to 10
		cmp r4, #10
		bne .L2
		
		; function epilog
		pop {r4, pc} ; return
			; function prolog
			push {r4, lr}
			
			; R4 = i
			movs r4, #0
			
			.L2:
			; do the function call with
			; i as the argument in R0
			uxtb r0, r4 ; sign extend
			adds r4, r4, #1 ; i++
			bl LCD_PrintInt8
			
			; check i not equal to 10
			cmp r4, #10
			bne .L2
			
			; function epilog
			pop {r4, pc} ; return

// While loop
// i is type automatic, see
// Accessing Automatic Variables
// for details
uint8 i = 0;
 	
while (i < 10)
{
  LCD_PrintInt8(i);
  i++;
}
	; prolog not shown
	; i = 0
	mov r4, #0
	b .L6
	
	.L7:
	; LCD_PrintInt8(i)
	mov r0, r4
	bl LCD_PrintInt8
	adds r4, r4, #1 ; i++
	bl LCD_PrintInt8
	
	.L6
	; i++
	cmp r4, #9
	bls .L7
	
	; epilog not shown
		; prolog not shown
		; i = 0
		movs r4, #0
		
		.L6:
		uxtb r0, r4
		bl LCD_PrintInt8
		adds r4, r4, #1 ; i++
		
		; check i not equal to 10
		cmp r4, #10
		bne .L6
		
		; epilog not shown
			; prolog not shown
			
			movs r4, #0
			.L7:
			uxtb r0, r4
			adds r4, r4, #1
			bl LCD_PrintInt8
			
			cmp r4, #10
			bne .L7
			
			; epilog not shown

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, デバッグ最適化	GCC, Cortex®-M3, 最小最適化	GCC, Cortex®-M3, 高最適化

`// Conditional statement`
`void Conditional(uint8 i, uint8 j)`
`{`
`  if(j == 1)`
`  {`
`  LCD_PrintInt8(i);`
`  }`
`  else`
`  {`
`  LCD_PrintInt8(i + 1);`
`  }`
`}`
	`; prolog not shown`
	`; if(j == 1)`
	`cmp r1, #1`
	`bne .L10`
	
	`; LCD_PrintInt8(i)`
	`bl LCD_PrintInt8`
	`pop {r3, pc}.L10:`
	`adds r0, r0, #1`
	`uxtb r0, r0`
	`; LCD_PrintInt8(i)`
	`bl LCD_PrintInt8`
	`pop {r3, pc}`
	`; epilog not shown`
		`; no prolog`
		`; if(j == 1)`
		`cmp r1, #1`
		`itt ne`
		`addne r0, r0, #1`
		
		`uxtbne r0, r0`
		`; LCD_PrintInt8(i)`
		`bl LCD_PrintInt8`
			`; no prolog`
			`; if(j == 1)`
			`cmp r1, #1`
			`itt ne`
			`addne r0, r0, #1`
			
			`uxtbne r0, r0`
			`; LCD_PrintInt8(i)`
			`b LCD_PrintInt8`

`// Switch case statements`
`void SwitchCase(uint8 j)`
`{`
`  switch(j)`
`  {`
`  case 0:`
`  LCD_PrintInt8(1);`
`  break;`
 	
`  case 1:`
`  LCD_PrintInt8(2);`
`  break;`
 	
`  default:`
`  LCD_PrintInt8(0);`
`  break;`
`  }`
`}`
	`; prolog not shown`
	`; switch(j)`
	`cbz r0, .L15`
	`cmp r0, #1`
	`beq .L16`
	`b .L18`
	
	`.L15: ; case 0`
	`mov r0, #1`
	`bl LCD_PrintInt8`
	`b .L7 ; break`
	
	`.L16: ; case 1`
	`movs r0, #2`
	`bl`
	`LCD_PrintInt8`
	`; break`
	
	`.L18: ; default`
	`movs r0, #0`
	`bl LCD_PrintInt8`
	`pop {r3, pc}; break`
	
	`; epilog not shown`
		`; no prolog`
		`; switch(j)`
		`cbz r0, .L15`
		`cmp r0, #1`
		`beq .L16`
		
		`; default`
		`b .L18`
		
		`.L15: ; case 0`
		`movs r0, #1`
		`bl LCD_PrintInt8`
		
		`.L16: ; case 1`
		`movs r0, #2`
		`bl LCD_PrintInt8`
		
		`.L18`
		`movs r0, #0`
		`bl LCD_PrintInt8`
			`; no prolog`
			`; switch(j)`
			`cbz r0, .L15`
			`cmp r0, #1`
			`bne .L19`
			
			`; case 1`
			`movs r0, #2`
			`b LCD_PrintInt8`
			
			`; default`
			`.L19:`
			`movs r0, #0`
			`b LCD_PrintInt8`
			
			`; case 0`
			`.L15:`
			`movs r0, #1 ;`
			`b LCD_PrintInt8`

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, デバッグ最適化	GCC, Cortex®-M3, 最小最適化	GCC, Cortex®-M3, 高最適化

`// Ternary operator`
`void Ternary(uint8 i)`
`{`
`  LCD_PrintInt8(`
`  (i == 1) ? 80 : 100);`
`}`
	`; prolog not shown`
	`; check value of i`
	`cmp  r0, #1`
	`bne  .L22`
	`movs  r0, #80`
	`b  .L21`
	
	`.L22:`
	`movs  r0, #100`
	
	`.L21:`
	`bl  LCD_PrintInt8`
	`; epilog not shown`
		`; no prolog`
		`; check value of i`
		`cmp  r0, #1`
		
		`; “ite” stands for if-then-`
		`; else instruction`
		`; “eq” condition checks`
		`; if the previous compare`
		`; instruction has set the`
		`; “equal to” flag`
		`ite  eq`
		
		`; mov if the result of the`
		`; previous “ite” instruction is`
		`; “equal”`
		`moveq  r0, #80`
		
		`; mov if the result of the`
		`; previous “ite” instruction is`
		`; “not equal”`
		`movne  r0, #100`
		
		`; no epilog`
		`bl  LCD_PrintInt8`
			`; same as for minimal optimization`

`// Addition operation`
`int DoAdd(int x, int y)`
`{`
`  return x + y;`
`}`
	`; no prolog`
	`add  r0, r0, r1`
	`bx  lr ; return with result`
		`; same as for Debug optimization`	`; same as for Debug optimization`

`// Subtraction operation`
`int DoSub(int x, int y)`
`{`
`  return x - y;`
`}`
	`; no prolog`
	`subs  r0, r0, r1`
	`bx  lr ; return with result`
		`; same as for Debug optimization`	`; same as for Debug optimization`

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, デバッグ最適化	GCC, Cortex®-M3, 最小最適化	GCC, Cortex®-M3, 高最適化

`// Multiplication`
`int DoMul(int x, int y)`
`{`
`  return x * y;`
`}`
	`; no prolog`
	`mul  r0, r1, r0`
	`bx    lr ; return with result`
		`; same as for Debug optimization`	`; same as for Debug optimization`

`// Division`
`int DoDiv(int x, int y)`
`{`
`  return x / y;`
`}`
	`; no prolog`
	`sdiv  r0, r0, r1`
	`bx    lr ; return with result`
		`; same as for Debug optimization`	`; same as for Debug optimization`

`// Modulo operator`
`int DoMod(int x, int y)`
`{`
`  return x % y;`
`}`
	`; no prolog`
	`sdiv  r3, r0, r1`
	`; multiply and subtract instruction`
	`; implements remainder =`
	`; dividend - (quotient * divisor)`
	`mls   r0, r3, r1, r0`
	`bx    lr ; return with result`
		`; same as for Debug optimization`	`; same as for Debug optimization`

`// Pointer`
`void Pointer(uint8 x, uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
 	
`LCD_PrintInt8(*ptr);`
`}`
	`; *ptr = *ptr + x`
	`ldrb  r3, [r1]; R1 = ptr`
	`add   r0, r0, r3; R0 = x`
	`; ptr++`
	`; LCD_PrintInt8(*ptr)`
	`ldrb  r0, [r1, #1]`
	`bl    LCD_PrintInt8`
		`; *ptr = *ptr + x`
		`ldrb  r3, [r1] ; R1 = ptr`
		`add   r0, r0, r3; R0 = x`
		`strb  r0, [r1]`
		
		`; ptr++`
		`; LCD_PrintInt8(*ptr)`
		`ldrb  r0, [r1, #1]`
		`b     LCD_PrintInt8`
			`; same as for minimal optimization`

`// Function pointer`
`void FuncPtr(uint8 x, void *fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; (*fptr)(x)`
	`; in a blx instruction, the LS`
	`; bit of the register must be`
	`; 1 to keep the CPU in Thumb`
	`; mode, or an exception occurs`
	`blx   r1`
		`; same as for debug optimization`	`; same as for debug optimization`

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M3, デバッグ最適化	GCC, Cortex®-M3, 最小最適化	GCC, Cortex®-M3, 高最適化

`// Packed structures`
`struct FOO_P`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`} __attribute__`
`((packed));`
 	
`extern struct FOO_P`
`myfoo_p;`
 	
`void`
`PackedStruct(void)`
`{`
`  myfoo_p.membera =`
`5;`
`  myfoo_p.memberb =`
`10;`
`  myfoo_p.memberc =`
`15;`
`  myfoo_p.memberd =`
`20;`
`}`
	`ldr  r3, .L34`
	`movs  r2, #5`
	`strb  r2, [r3];`
	`memberb = 10`
	`movs  r2, #10`
	`strb  r2, [r3, #1]`
	`; memberc = 15`
	`movs  r2, #0`
	`movs  r1, #15`
	`strb  r1, [r3, #2]`
	`strb  r2, [r3, #3]`
	`strb  r2, [r3, #4]`
	`strb  r2, [r3, #5]`
	`; memberd = 20`
	`movs  r1, #20`
	`strb  r1, [r3, #6]`
	`strb  r2, [r3, #7]`
	`bx  lr`
	
	`.L34:`
	`.word  myfoo_p`
		`; same as for Debug`
		`optimization`
			`ldr  r3, .L32`
			`movs  r5, #5`
			`; memberb = 10`
			`movs  r4, #10`
			`strb  r5, [r3]`
			`; memberc = 15`
			`movs  r2, #0`
			`movs  r0, #15`
			`strb  r4, [r3, #1]`
			`; memberd = 20`
			`movs  r1, #20`
			`strb  r0, [r3, #2]`
			`strb  r1, [r3, #6]`
			`strb  r2, [r3, #3]`
			`strb  r2, [r3, #4]`
			`strb  r2, [r3, #5]`
			
			`strb  r2, [r3, #7]`
			`bx  lr`
			
			`.L32:`
			`.word  myfoo_p`

`// unpacked`
`structures`
`struct FOO`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`};`
 	
`extern struct FOO`
`myfoo;`
 	
`void`
`PackedStruct(void)`
`{`
`  myfoo.membera =  5;`
`  myfoo.memberb = 10;`
`  myfoo.memberc = 15;`
`  myfoo.memberd = 20;`
`}`
	`; membera = 5`
	`ldr  r3, .L37`
	`movs  r2, #5`
	`strb  r2, [r3]`
	`; memberb = 10`
	`movs  r2, #10`
	`strb  r2, [r3, #1]`
	`; memberc = 15`
	`movs  r2, #15`
	`str  r2, [r3, #4];`
	`memberd = 20`
	`movs  r2, #20`
	`strh  r2, [r3, #8]`
	`bx  lr`
	
	`.L37:`
	`.word  myfoo`
		`; same as for Debug`
		`optimization`
			`; membera = 5`
			`ldr  r3, .L36`
			`push  {r4}`
			`movs  r4, #5`
			`; memberb = 10`
			`movs  r0, #10`
			`strb  r4, [r3]`
			`; memberc = 15`
			`movs  r1, #15`
			`strb  r0, [r3, #1];`
			`memberd = 20`
			`movs  r2, #20`
			`str  r1, [r3, #4]`
			`strh  r2, [r3, #8]`
			`pop  {r4}`
			`bx  lr`
			
			`.L36:`
			`.word  myfoo`

14.2 アセンブラの例、Cortex®-M0/M0+ 用 GCC

14.2.1 Cortex®-M0/M0+用 GCC、なし, サイズ, および速度最適化

Table 29 に、Cortex-M0/M0+の GCC コンパイラについて、さまざまな最適化オプションのコンパイラ出力の例を示します。これらの例は、コンパイラによって生成された.lst ファイルから抽出されたものです。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

Table 29 Cortex®-M0/M0+ CPU 用 GCC コンパイラのコンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, 最適化なし	GCC, Cortex®-M0/M0+, サイズ最適化	GCC, Cortex®-M0/M0+, 速度最適化

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`bl LCD_Start`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`; conditional flags are NOT`
	`; updated by mov`
	`r0, #128`
	`bl LCD_PrintInt8`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`mov r1, #2`
	`mov r0, #0`
	`bl LCD_Position`
		`; same as for no optimization`	`; same as for no optimization`

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`for(i = 0; i < 10; i++)`
`{`
`LCD_PrintInt8(i);`
`}`
`}`
	`; function prolog`
	`; i is saved on the stack`
	`push {r7, lr}`
	`sub sp, sp, #16`
	`add r7, sp, #0`
	`mov r2, r0`
	`add r3, r7, #7`
	`strb r2, [r3]`
	
	`; i = 0`
	`mov r3, r7`
	`add r3, r3, #15`
	`mov r2, #0`
	`strb r2, [r3]`
	`b .L2`
	
	`; do the function call with`
	`; i as the argument in R0`
	`.L3:`
		`; function prolog`
		`push {r4, lr}`
		
		`; R4 = i`
		`mov r4, #0`
		
		`.L2:`
		`; do the function call with`
		`; i as the argument in R0`
		`mov r0, r4 ; sign extend`
		`add r4, r4, #1 ; i++`
		`uxtb r4, r4`
		`bl LCD_PrintInt8`
		
		`; check i not equal to 10`
		`cmp r4, #10`
		`bne .L2`
			`; function prolog`
			`push {r3, lr}`
			
			`; unroll the loop`
			`; do the function call`
			`; 10 times`
			`; i as the argument in R0`
			`mov r0, #0`
			`bl LCD_PrintInt8`
			
			`mov r0, #1`
			`bl LCD_PrintInt8`
			
			`. . .`
			
			`mov r0, #9`
			`bl LCD_PrintInt8`
			
			`pop {r3, pc} ; return`

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, 最適化なし	GCC, Cortex®-M0/M0+, サイズ最適化	GCC, Cortex®-M0/M0+, 速度最適化

	mov r3, r7
	add r3, r3, #15
	ldrb r3, [r3]
	mov r0, r3
	bl LCD_PrintInt8
	
	; i++
	mov r3, r7
	add r3, r3, #15
	mov r2, r7
	add r2, r2, #15
	ldrb r2, [r2]
	add r2, r2, #1
	strb r2, [r3]
	
	; check i 10, by comparing
	; it with 9
	.L2:
	mov r3, r7
	add r3, r3, #15
	ldrb r3, [r3]
	cmp r3, #9
	bls .L3
	
	; function epilog
	mov sp, r7
	add sp, sp, #16
	pop {r7, pc}
		pop {r4, pc} ;
		return	

// While loop
// i is type automatic, see
// Accessing Automatic Variables
// for details
uint8 i = 0;
 	
while (i < 10)
{
LCD_PrintInt8(i);
i++;
}
	; prolog not shown
	; i = 0
	add r3, r7, #7
	mov r2, #0
	strb r2, [r3]
	b .L5
	
	.L6:
	; LCD_PrintInt8(i)
	add r3, r7, #7
	ldrb r3, [r3]
	mov r0, r3
	bl LCD_PrintInt8
	
	; i++
	add r3, r7, #7
	add r2, r7, #7
	ldrb r2, [r2]
	add r2, r2, #1
	strb r2, [r3]
		; prolog not shown
		; i = 0
		movs r4, #0
		
		.L6:
		mov r0, r4
		add r4, r4, #1 ;
		i++
		uxtb r4, r4
		; LCD_PrintInt8(i)
		bl LCD_PrintInt8
		
		; check i not equal to 10
		cmp r4, #10
		bne .L6
		
		; epilog not shown
			; prolog not shown
			; unroll the loop
			; do the function call
			; 10 times
			; i as the argument in R0
			mov r0, #0
			bl LCD_PrintInt8
			
			mov r0, #1
			bl LCD_PrintInt8
			
			. . .
			
			mov r0, #9
			bl LCD_PrintInt8
			
			; epilog not shown

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, 最適化なし	GCC, Cortex®-M0/M0+, サイズ最適化	GCC, Cortex®-M0/M0+, 速度最適化

	.L5:
	; while(i 10)
	add r3, r7, #7
	ldrb r3, [r3]
	cmp r3, #9
	bls .L6
	
	; epilog not shown		

// Conditional statement
void Conditional(uint8 i, uint8 j)
{
  if(j == 1)
  {
  LCD_PrintInt8(i);
  }
  else
  {
  LCD_PrintInt8(i + 1);
  }
}
	; prolog not shown
	; if(j == 1)
	add r3, r7, #6
	ldrb r3, [r3]
	cmp r3, #1
	bne .L8
	
	; LCD_PrintInt8(i)
	add r3, r7, #7
	ldrb r3, [r3]
	mov r0, r3
	bl LCD_PrintInt8
	b .L7
	
	.L8:
	; LCD_PrintInt8(i + 1)
	add r3, r7, #7
	ldrb r3, [r3]
	add r3, r3, #1
	uxtb r3, r3
	mov r0, r3
	bl LCD_PrintInt8
	
	.L7:
	; epilog not shown
		; prolog not shown
		; if(j == 1)
		cmp r1, #1
		beq .L11
		
		; LCD_PrintInt8(i)
		add r0, r0, #1
		uxtb r0, r0
		
		.L11:
		bl LCD_PrintInt8
		
		; epilog not shown
			; same as for size optimization

// Switch case statements
void SwitchCase(uint8 j)
{
  switch(j)
  {
  case 0:
  LCD_PrintInt8(1);
  break;
 	
  case 1:
  LCD_PrintInt8(2);
  break;
 	
  default:
	; prolog not shown
	; switch(j)
	add r3, r7, #7
	ldrb r3, [r3]
	cmp r3, #0
	beq .L12
	cmp r3, #1
	beq .L13
	b .L15
	
	.L12 ; case 0
	mov r0, #1
	bl LCD_PrintInt8
	b .L10 ; break
		; prolog not shown
		; switch(j)
		cmp r0, #0
		beq .L14
		cmp r0, #1
		bne .L17
		
		mov r0, #2 ; case 1
		b .L18
		
		.L14:
		mov r0, #1 ; case 0
		b .L18
		
		.L17:
		mov r0, #0 ; default
			; prolog not shown
			; switch(j)
			cmp r0, #0
			bne .L14
			mov r0, #1 ; case 0
			bl LCD_PrintInt8
			
			.L8:
			pop {r3, pc} ; return
			
			.L14:
			cmp r0, #1
			beq .L15
			mov r0, #0 ; default
			bl LCD_PrintInt8
			b .L8

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, 最適化なし	GCC, Cortex®-M0/M0+, サイズ最適化	GCC, Cortex®-M0/M0+, 速度最適化

LCD_PrintInt8(0);
break;
}
}
	.L13: ; case 1
	mov r0, #2
	bl LCD_PrintInt8
	b .L10 ; break
	
	.L15: ; default
	mov r0, #0
	bl LCD_PrintInt8
	mov r8, r8 ; break - nop
	
	.L10:
	; epilog not shown
		.L18:
		bl LCD_PrintInt8
		; epilog not shown
			mov r0, #2 ; case 1
			bl LCD_PrintInt8
			b .L8

// Ternary operator
void Ternary(uint8 i)
{
LCD_PrintInt8(
(i == 1) ? 80 :
100);
}
	; prolog not shown
	; check value of i
	add r3, r7, #7
	ldrb r3, [r3]
	cmp r3, #1
	bne .L17
	
	mov r3, #80
	b .L18
	
	.L17:
	mov r3, #100
	
	.L18:
	mov r0, r3
	bl LCD_PrintInt8
	; epilog not shown
		; prolog not shown
		mov r3, #100
		cmp r0, #1
		bne .L20
		mov r3, #80
		
		.L20:
		mov r0, r3
		bl LCD_PrintInt8
		; epilog not shown
			; prolog not shown
			mov r3, #100
			cmp r0, #1
			beq .L19
			
			.L17:
			mov r0, r3
			bl LCD_PrintInt8
			pop {r3, pc} ; return
			
			.L19:
			mov r3, #80
			b .L17

// Addition operation
int DoAdd(int x, int y)
{
return x + y;
}
	; prolog not shown
	ldr r2, [r7, #4]
	ldr r3, [r7, #0]
	add r3, r2, r3
	mov r0, r3 ; return value
	; epilog not shown
		; no prolog
		add r0, r0, r1
		bx lr ; return with result
			; same as for size optimization

// Subtraction operation
int DoSub(int x, int y)
{
return x - y;
}
	; prolog not shown
	ldr r2, [r7, #4]
	ldr r3, [r7, #0]
	sub r3, r2, r3
	mov r0, r3 ; return value
	; epilog not shown
		; no prolog
		sub r0, r0, r1
		bx lr ; return with result
			; same as for size optimization

// Multiplication
int DoMul(int x, int y)
	; prolog not shown
	ldr r3, [r7, #4]
	ldr r2, [r7, #0]
		; no prolog
		mul r0, r1
			; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, 最適化なし	GCC, Cortex®-M0/M0+, サイズ最適化	GCC, Cortex®-M0/M0+, 速度最適化

{ return x * y; }	mul r3, r2
	mov r0, r3 ; return value
	; epilog not shown
		bx lr ; return with result	

// Division
int DoDiv(int x, int y)
{ return x / y; }
	; prolog not shown
	ldr r0, [r7, #4]
	ldr r1, [r7, #0]
	bl __aeabi_idiv
	mov r3, r0
	mov r0, r3 ; return value
	; epilog not shown
		push {r3, lr}
		bl __aeabi_idiv
		; return with result
		pop {r3, pc}
			; same as for size optimization

// Modulo operator
int DoMod(int x, int y)
{ return x % y; }
	; prolog not shown
	ldr r3, [r7, #4]
	mov r0, r3
	ldr r1, [r7]
	bl __aeabi_idivmod
	mov r3, r1
	mov r0, r3 ; return value
	; epilog not shown
		push {r3, lr}
		bl __aeabi_idivmod
		; return with result
		mov r0, r1
		pop {r3, pc}
			; same as for size optimization

// Pointer
void Pointer(uint8 x, uint8 *ptr)
{ *ptr = *ptr + x; ptr++; LCD_PrintInt8(*ptr); }
	; *ptr = *ptr + x
	ldr r3, [r7] ; ptr
	ldrb r2, [r3]
	add r3, r7, #7 ; x
	ldrb r3, [r3]
	add r3, r2, r3
	uxtb r2, r3
	ldr r3, [r7] ; ptr
	strb r2, [r3]
	
	; ptr++
	ldr r3, [r7] ; ptr
	add r3, r3, #1
	str r3, [r7]
	
	; LCD_PrintInt8(*ptr)
	ldr r3, [r7]
	ldrb r3, [r3]
	mov r0, r3
	bl LCD_PrintInt8
		; *ptr = *ptr + x
		ldrb r3, [r1] ; R1 = ptr
		add r0, r0, r3 ; R0 = x
		strb r0, [r1]
		
		; ptr++
		; LCD_PrintInt8(*ptr)
		ldrb r0, [r1, #1]
		bl LCD_PrintInt8
			; same as for size optimization

// Function pointer
void FuncPtr(uint8 x, void *fptr(uint8))
{
	; (*fptr)(x)
	add r3, r7, #7 ; x
	ldrb r2, [r3]
	ldr r3, [r7] ; fptr
		; (*fptr)(x)
		; in a blx instruction, the LS
		; bit of the register must be
			; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, 最適化なし	GCC, Cortex®-M0/M0+, サイズ最適化	GCC, Cortex®-M0/M0+, 速度最適化
(*fptr)(x); }	mov r0, r2 ; in a blx instruction, the ; LS bit of the register ; must be 1 to keep the CPU ; in Thumb mode, or an ; exception occurs blx r3	; 1 to keep the CPU in Thumb ; mode, or an exception occurs blx r1	

// Packed structures
struct FOO_P
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
} __attribute__ ((packed));
 	
extern struct FOO_P myfoo_p;
 	
void PackedStruct(void)
{
myfoo_p.membera = 5;
myfoo_p.memberb = 10;
myfoo_p.memberc = 15;
myfoo_p.memberd = 20;
}
	; membera = 5
	ldr r3, [pc, .L32]
	mov r2, #5
	strb r2, [r3]
	; memberb = 10
	ldr r3, [pc, .L32]
	mov r2, #10
	strb r2, [r3, #1]
	; memberc = 15
	ldr r3, [pc, .L32]
	ldrb r1, [r3, #2]
	mov r2, #0
	and r2, r1
	mov r1, #15
	orr r2, r1
	strb r2, [r3, #2]
	ldrb r1, [r3, #3]
	mov r2, #0
	and r2, r1
	strb r2, [r3, #3]
	ldrb r1, [r3, #4]
	mov r2, #0
	and r2, r1
	strb r2, [r3, #4]
	ldrb r1, [r3, #5]
	mov r2, #0
	and r2, r1
	strb r2, [r3, #5]
	; memberd = 20
	ldr r3, [pc, .L32]
	ldrb r1, [r3, #6]
	mov r2, #0
	and r2, r1
	mov r1, #20
	orr r2, r1
	strb r2, [r3, #6]
	ldrb r1, [r3, #7]
	mov r2, #0
	and r2, r1
		ldr r3, [pc, .L30]
		mov r2, #5
		mov r0, #10
		strb r2, [r3] ;
		membera = 5
		strb r0, [r3, #1] ;
		memberb = 10
		mov r2, #0
		mov r1, #15
		mov r0, #20
		strb r1, [r3, #2] ;
		memberc = 15
		strb r2, [r3, #3]
		strb r2, [r3, #4]
		strb r2, [r3, #5]
		strb r0, [r3, #6] ;
		memberd = 20
		strb r2, [r3, #7]
		bx lr
		
		.L30:
		.word myfoo_p
			; same as for size optimization

C コード	GCC, Cortex®-M0/M0+, 最適化なし	GCC, Cortex®-M0/M0+, サイズ最適化	GCC, Cortex®-M0/M0+, 速度最適化
	strb r2, [r3, #7] .L32: .word myfoo_p		

// unpacked structures
struct FOO
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
};
 	
extern struct FOO
myfoo;
 	
void
PackedStruct(void)
{
  myfoo.membera = 5;
  myfoo.memberb = 10;
  myfoo.memberc = 15;
  myfoo.memberd = 20;
}
	; membera = 5
	ldr r3, [pc, .L35]
	mov r2, #5
	strb r2, [r3]
	; memberb = 10
	ldr r3, [pc, .L35]
	mov r2, #10
	strb r2, [r3, #1]
	; memberc = 15
	ldr r3, [pc, .L35]
	mov r2, #15
	str r2, [r3, #4]
	; memberd = 20
	ldr r3, [pc, .L35]
	mov r2, #20
	strh r2, [r3, #8]
	
	.L35:
	.word myfoo
		ldr r3, [pc, .L33]
		mov r2, #5
		strb r2, [r3] ;
		membera = 5
		mov r0, #10
		mov r1, #15
		mov r2, #20
		strb r0, [r3, #1] ;
		memberb = 10
		str r1, [r3, #4] ;
		memberc = 15
		strh r2, [r3, #8] ;
		memberd = 20
		bx lr
		
		.L33:
		.word myfoo
			; same as for size optimization

14.2.2 Cortex®-M0/M0+用 GCC、デバッグ, 最小, および高最適化

Table 30 に、Cortex®-M0/M0+用の GCC コンパイラについて、さまざまな最適化オプションのコンパイラ出力の例を示します。これらの例は、コンパイラによって生成された.lst ファイルから抽出されたものです。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

Table 30 Cortex®-M0/M0+ CPU 用 GCC コンパイラのコンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, デバッグ最適化	GCC, Cortex®-M0/M0+, 最小最適化	GCC, Cortex®-M0/M0+, 高最適化

// Calling a function
// with no arguments
LCD_Start();
	; do the function call
	bl LCD_Start
		; same as for debug optimization	; same as for debug optimization

// Calling a function with
// one argument
LCD_PrintInt8(128);
	; R0 = first argument
	; conditional flags are NOT
	; updated by mov
	  r0, #128
	bl LCD_PrintInt8
		; same as for debug optimization	; same as for debug optimization

// Calling a function with	; R0 = first argument ; R1 = second	; same as for debug optimization	; same as for debug optimization

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, デバッグ最適化	GCC, Cortex®-M0/M0+, 最小最適化	GCC, Cortex®-M0/M0+, 高最適化

`// two arguments`
`LCD_Position(0, 2);`
	`argument`
	`mov  r1, #2`
	`mov  r0, #0`
	`bl   LCD_Position`		

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`  for(i = 0; i < 10; i++)`
`  {`
`  LCD_PrintInt8(i);`
`  }`
`}`
	`; function prolog`
	`push  {r4, lr}`
	
	`; i = 0`
	`movs  r4, #0`
	`b     .L2`
	
	`; do the function call with`
	`; i as the argument in R0`
	`.L3:`
	`movs  r0, r4`
	`bl    LCD_PrintInt8`
	`adds  r4, r4, #1 ; i++`
	`uxtb  r4, r4`
	
	`; check i  10, by comparing`
	`; it with 9`
	`.L2:`
	`cmp   r4, #9`
	`bls   .L3`
	
	`; function epilog`
	`pop   {r4, pc}`
		`; function prolog`
		`push  {r4, lr}`
		
		`; R4 = i`
		`movs  r4, #0`
		
		`.L2:`
		`; do the function call with`
		`; i as the argument in R0`
		`movs  r0, r4 ; sign extend`
		`bl    LCD_PrintInt8`
		`adds  r4, r4, #1 ; i++`
		`uxtb  r4, r4`
		
		`; check i not equal to 10`
		`cmp   r4, #10`
		`bne   .L2`
		
		`pop   {r4, pc} ; return`
			`; function prolog`
			`push  {r4, lr}`
			
			`; R4 = i`
			`movs  r4, #0`
			
			`.L2:`
			`; do the function call with`
			`; i as the argument in R0`
			`movs  r0, r4 ; sign extend`
			`adds  r4, r4, #1 ; i++`
			`uxtb  r4, r4`
			`bl    LCD_PrintInt8`
			
			`; check i not equal to 10`
			`cmp   r4, #10`
			`bne   .L2`
			
			`pop   {r4, pc} ; return`

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic Variables`
`// for details`
`uint8 i = 0;`
 	
`while (i < 10)`
`{`
`  LCD_PrintInt8(i);`
`  i++;`
`}`
	`; prolog not shown`
	`; i = 0`
	`movs  r4, #0`
	`b     .L5`
	
	`.L6:`
	`; LCD_PrintInt8(i)`
	`movs  r0, r4`
	`bl    LCD_PrintInt8`
	`adds  r4, r4, #1`
	`uxtb  r4, r4`
	
	`.L5:`
	`; while(i  10)`
	`cmp   r4, #9`
	`bls   .L6`
	
	`; epilog not shown`
		`; prolog not shown`
		`; i = 0`
		`movs  r4, #0`
		
		`.L5:`
		`movs  r0, r4`
		`; LCD_PrintInt8(i)`
		`bl    LCD_PrintInt8`
		`adds  r4, r4, #1 ; i++`
		`uxtb  r4, r4`
		
		`; check i not equal to 10`
		`cmp   r4, #10`
		`bne   .L5`
		
		`; epilog not shown`
			`; prolog not shown`
			`; i = 0`
			`movs  r4, #0`
			
			`.L6:`
			`movs  r0, r4`
			`adds  r4, r4, #1 ; i++`
			`uxtb  r4, r4`
			`; LCD_PrintInt8(i)`
			`bl    LCD_PrintInt8`
			
			`; check i not equal to 10`
			`cmp   r4, #10`
			`bne   .L6`
			
			`; epilog not shown`

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, デバッグ最適化	GCC, Cortex®-M0/M0+, 最小最適化	GCC, Cortex®-M0/M0+, 高最適化

`// Conditional statement`
`void Conditional(uint8 i, uint8 j)`
`{`
`  if(j == 1)`
`  {`
`  LCD_PrintInt8(i);`
`  }`
`  else`
`  {`
`  LCD_PrintInt8(i + 1);`
`  }`
`}`
	`; prolog`
	`push  {r4, lr}`
	`; if(j == 1)`
	`cmp  r1, #1`
	`bne  .L8`
	
	`; LCD_PrintInt8(i)`
	`bl  LCD_PrintInt8`
	`b  .L7`
	
	`.L8:`
	`; LCD_PrintInt8(i + 1)`
	`adds  r0, r0, #1`
	`uxtb  r0, r0`
	`bl  LCD_PrintInt8`
	
	`.L7:`
	`pop  {r4, pc}`
	`; epilog not shown`
		`; same as for debug optimization`	`; prolog`
			`push  {r4, lr}`
			`; if(j == 1)`
			`cmp  r1, #1`
			`beq  .L11`
			
			`.L11:`
			`; LCD_PrintInt8(i + 1)`
			`adds  r0, r0, #1`
			`uxtb  r0, r0`
			`bl  LCD_PrintInt8`
			
			`pop  {r4, pc}`
			`; epilog not shown`

`// Switch case statements`
`void SwitchCase(uint8 j)`
`{`
`  switch(j)`
`  {`
`  case 0:`
`  LCD_PrintInt8(1);`
`  break;`
 	
`  case 1:`
`  LCD_PrintInt8(2);`
`  break;`
 	
`  default:`
`  LCD_PrintInt8(0);`
`  break;`
`  }`
`}`
	`; prolog not shown`
	`; switch(j)`
	`cmp  r0, #0`
	`beq  .L12`
	`cmp  r0, #1`
	`beq  .L13`
	`b  .L15`
	
	`.L12  ; case 0`
	`mov  r0, #1`
	`bl  LCD_PrintInt8`
	`b  .L10  ; break`
	
	`.L13:  ; case 1`
	`mov  r0, #2`
	`bl  LCD_PrintInt8`
	`b  .L10  ; break`
	
	`.L15:  ; default`
	`mov  r0, #0`
	`bl  LCD_PrintInt8`
	`mov  r8, r8 ; break - nop`
	
	`.L10:`
	`; epilog not shown`
		`; ; prolog not shown`
		`; switch(j)`
		`cmp  r0, #0`
		`beq  .L12`
		`cmp  r0, #1`
		`beq  .L13`
		`b  .L15`
		
		`.L12  ; case 0`
		`mov  r0, #1`
		`bl  LCD_PrintInt8`
		`b  .L10  ; break`
		
		`.L13:  ; case 1`
		`movs  r0, #2`
		`bl  LCD_PrintInt8`
		`b  .L10  ; break`
		
		`.L15:  ; default`
		`movs  r0, #0`
		`bl  LCD_PrintInt8`
		`movs  r8, r8 ; break - nop`
		
		`.L10:`
		`; epilog not shown`
			`; ; prolog not shown`
			`; switch(j)`
			`cmp  r0, #0`
			`beq  .L14`
			`cmp  r0, #1`
			`beq  .L15`
			`movs  r0, #0 ; default`
			`bl  LCD_PrintInt8`
			
			`.L14  ; case 0`
			`mov  r0, #1`
			`bl  LCD_PrintInt8`
			`b  .L10  ; break`
			
			`.L15:  ; case 1`
			`movs  r0, #2`
			`bl  LCD_PrintInt8`
			`b  .L12  ; break`
			
			`.L12:`
			`; epilog not shown`

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, デバッグ最適化	GCC, Cortex®-M0/M0+, 最小最適化	GCC, Cortex®-M0/M0+, 高最適化

`// Ternary operator`
`void Ternary(uint8 i)`
`{`
`  LCD_PrintInt8(`
`  (i == 1) ? 80 :`
`100);`
`}`
	`; prolog not shown`
	`; check value of i`
	`cmp  r0, #1`
	`bne  .L18`
	`adds  r0, r0, #79`
	`mov  r3, #80`
	`b  .L17`
	
	`.L18:`
	`mov  r0, #100`
	
	`.L17:`
	`bl  LCD_PrintInt8`
	`; epilog not shown`
		`; prolog not shown`
		`cmp  r0, #1`
		`beq  .L18`
		`movs  r0, #100`
		`b  .L17`
		
		`.L18:`
		`movs  r0, #80`
		
		`.L17:`
		`bl  LCD_PrintInt8`
		`; epilog not shown`
			`; prolog not shown`
			`cmp  r0, #1`
			`beq  .L21`
			`movs  r0, #100`
			
			`.L21:`
			`movs  r0, #80`
			`b  .L20`
			
			`.L20:`
			`bl  LCD_PrintInt8`
			`; epilog not shown`

`// Addition operation`
`int DoAdd(int x, int`
`y)`
`{`
`  return x + y;`
`}`
	`; no prolog`
	`adds  r0, r0, r1`
	`bx  lr ; return`
	`with result`
		`; same as for debug`
		`optimization`
			`; same as for debug`
			`optimization`

`// Subtraction`
`operation`
`int DoSub(int x, int`
`y)`
`{`
`  return x - y;`
`}`
	`; no prolog`
	`subs  r0, r0, r1`
	`bx  lr ; return`
	`with result`
		`; same as for debug`
		`optimization`
			`; same as for debug`
			`optimization`

`// Multiplication`
`int DoMul(int x, int`
`y)`
`{`
`  return x * y;`
`}`
	`; no prolog`
	`muls r0, r1`
	`bx  lr ; return with`
	`result`
		`; same as for debug`
		`optimization`
			`; same as for debug`
			`optimization`

`// Division`
`int DoDiv(int x, int`
`y)`
`{`
`  return x / y;`
`}`
	`push  {r4, lr}`
	`; call into c helper`
	`function`
	`bl  __aeabi_idiv`
	`; return with result`
	`pop  {r4, pc}`
		`; same as for debug`
		`optimization`
			`; same as for debug`
			`optimization`

`// Modulo operator`
`int DoMod(int x, int`
`y)`
`{`
`  return x % y;`
`}`
	`push  {r4, lr}`
	`; call into c helper`
	`function`
	`bl  __aeabi_idivmod`
	`; return with result`
	`movs  r0, r1`
	`pop  {r4, pc}`
		`; same as for debug`
		`optimization`
			`; same as for debug`
			`optimization`

`// Pointer` `void Pointer(uint8 x,`	`; *ptr = *ptr + x` `ldrb  r3, [r1]  ; R1`	`; same as for debug` `optimization`	`; same as for debug` `optimization`

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, デバッグ最適化	GCC, Cortex®-M0/M0+, 最小最適化	GCC, Cortex®-M0/M0+, 高最適化

uint8 *ptr)
{
  *ptr = *ptr + x;
  ptr++;
  LCD_PrintInt8(*ptr);
}
	= ptr
	adds r0, r3, r0 ; R0 = x
	strb r0, [r1]
	
	; ptr++
	; LCD_PrintInt8(*ptr)
	ldrb r0, [r1, #1]
	bl LCD_PrintInt8		

// Function pointer
void FuncPtr(uint8 x, void *fptr(uint8))
{
  (*fptr)(x);
}
	; (*fptr)(x)
	; in a blx instruction, the LS
	; bit of the register must be
	; 1 to keep the CPU in Thumb
	; mode, or an exception occurs
	blx r1
		; same as for debug optimization	; same as for debug optimization

// Packed structures
struct FOO_P
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
} __attribute__ ((packed));
 	
extern struct FOO_P myfoo_p;
 	
void PackedStruct(void)
{
  myfoo_p.membera = 5;
  myfoo_p.memberb = 10;
  myfoo_p.memberc = 15;
  myfoo_p.memberd = 20;
}
	; membera = 5
	ldr r3, [pc, .L32]
	mov r2, #5
	strb r2, [r3]
	; memberb = 10
	ldr r3, [pc, .L32]
	mov r2, #10
	strb r2, [r3, #1]
	; memberc = 15
	ldr r3, [pc, .L32]
	ldrb r1, [r3, #2]
	mov r2, #0
	and r2, r1
	mov r1, #15
	orr r2, r1
	strb r2, [r3, #2]
	ldrb r1, [r3, #3]
	mov r2, #0
	and r2, r1
	strb r2, [r3, #3]
	ldrb r1, [r3, #4]
	mov r2, #0
	and r2, r1
	strb r2, [r3, #4]
	ldrb r1, [r3, #5]
	mov r2, #0
	and r2, r1
	strb r2, [r3, #5]
	; memberd = 20
	ldr r3, [pc, .L32]
	ldrb r1, [r3, #6]
		; membera = 5
		ldr r3, .L27
		movs r2, #5
		strb r2, [r3]
		; memberb = 10
		adds r2, r2, #5
		strb r2, [r3, #1]
		; memberc = 15
		movs r1, #15
		strb r1, [r3, #2]
		movs r1, #0
		strb r1, [r3, #3]
		strb r1, [r3, #4]
		strb r1, [r3, #5]
		; memberd = 20
		movs r2, #20
		strb r2, [r3, #6]
		movs r2, #0
		strb r2, [r3, #7]
		bx lr
		
		.L27:
		.word myfoo_p
			; membera = 5
			movs r2, #5
			ldr r3, .L30
			; memberd = 20
			movs r1, #20
			; membera = 5
			strb r2, [r3]
			; memberb = 10
			adds r2, r2, #5
			strb r2, [r3, #1]
			; memberd = 20
			adds r2, r2, #5
			strb r2, [r3, #2]
			movs r2, #0
			strb r1, [r3, #6]
			strb r2, [r3, #3]
			
			strb r2, [r3, #4]
			strb r2, [r3, #5]
			strb r2, [r3, #7]
			bx lr
			
			.L30:
			.word myfoo_p

14 付録: コンパイラ出力の詳細

C コード	GCC, Cortex®-M0/M0+, デバッグ最適化	GCC, Cortex®-M0/M0+, 最小最適化	GCC, Cortex®-M0/M0+, 高最適化

	mov r2, #0
	and r2, r1
	mov r1, #20
	orr r2, r1
	strb r2, [r3, #6]
	ldrb r1, [r3, #7]
	mov r2, #0
	and r2, r1
	strb r2, [r3, #7]
	
	.L32:
	.word myfoo_p		

// unpacked structures
struct FOO
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
};
 	
extern struct FOO
myfoo;
 	
void
PackedStruct(void)
{
myfoo.membera = 5;
myfoo.memberb = 10;
myfoo.memberc = 15;
myfoo.memberd = 20;
}
	; membera = 5
	ldr r3, [pc, .L35]
	mov r2, #5
	strb r2, [r3]
	; memberb = 10
	ldr r3, [pc, .L35]
	mov r2, #10
	strb r2, [r3, #1]
	; memberc = 15
	ldr r3, [pc, .L35]
	mov r2, #15
	str r2, [r3, #4]
	; memberd = 20
	ldr r3, [pc, .L35]
	mov r2, #20
	strh r2, [r3, #8]
	
	.L35:
	.word myfoo
		; membera = 5
		ldr r3, .L30
		movs r2, #5
		strb r2, [r3]
		; memberb = 10
		adds r2, r2, #5
		strb r2, [r3, #1]
		; memberc = 15
		adds r2, r2, #5
		str r2, [r3, #4]
		; memberd = 20
		adds r2, r2, #5
		strh r2, [r3, #8]
		bx lr
		
		.L30:
		.word myfoo
			; same as for minimal optimization

14.3 アセンブラ例、Cortex®-M3 用 MDK

Note: Table 31 に、Cortex®-M3 用の MDK コンパイラについて、さまざまな最適化オプションのコンパイラ出力の例を示します。MDK の無料評価版である MDK-Lite には.lst ファイルにアセンブラが含まれていないため、例は PSoC ™Creator のアセンブラレベルのデバッグウィンドウから抽出されました。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

14 付録: コンパイラ出力の詳細

Table 31 Cortex®-M3 CPU 用 MDK コンパイラのコンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`bl LCD_Start`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`movs r0, #80`
	`bl LCD_PrintInt8`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`movs r1, #2`
	`movs r0, #0`
	`bl LCD_Position`
		`; same as for no optimization`	`; same as for no optimization`

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`for(i = 0; i < 10; i++)`
`{`
`LCD_PrintInt8(i);`
`}`
`}`
	`; prolog`
	`push {r4, lr}`
	`mov r4, r0`
	
	`movs r4, #0 ; i = 0`
	`b.n <ForLoop+0x12>`
	
	`<ForLoop+0x8>:`
	`; do the function call with`
	`; i as the argument in R0`
	`mov r0, r4`
	`bl LCD_PrintInt8`
	
	`adds r0, r4, #1 ; i++`
	`uxtb r4, r0 ; sign extend`
	
	`<ForLoop+0x12>:`
	`cmp r4, #a ; i < 10`
	`blt.n <ForLoop+0x8>`
	
	`pop {r4, pc} ; return`
		`; prolog`
		`push {r4, lr}`
		
		`movs r4, #0 ; i = 0`
		
		`<ForLoop+0x4>:`
		`; do the function call with`
		`; i as the argument in R0`
		`mov r0, r4`
		`bl LCD_PrintInt8`
		
		`adds r4, r4, #1 ; i++`
		`uxtb r4, r4 ; sign extend`
		`cmp r4, #a ; i < 10`
		`bcc.n <ForLoop+0x4>`
		
		`pop {r4, pc} ; return`
			`; same as for size optimization`

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic Variables`
`// for details`
`uint8 i = 0;`
	`; prolog`
	`push {r4, lr}`
	
	`movs r4, #0 ; i = 0`
	`b.n <WhileLoop+0x10>`
		`; prolog`
		`push {r4, lr}`
		
		`movs r4, #0 ; i = 0`
		
		`<WhileLoop+0x4>:`
			`; same as for size optimization`

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化

`while (i < 10)`
`{`
`  LCD_PrintInt8(i);`
`  i++;`
`}`
	`<WhileLoop+0x6>:`
	`; do the function call with`
	`; i as the argument in R0`
	`mov  r0, r4`
	`bl  LCD_PrintInt8`
	
	`adds  r0, r4, #1 ; i++`
	`uxtb  r4, r0 ; sign extend`
	
	`<WhileLoop+0x10>:`
	`cmp  r4, #a  ; i < 10`
	`blt.n`
	`<WhileLoop+0x6>`
	
	`pop  {r4, pc};`
	`return`
		`; do the function call with`
		`; i as the argument in R0`
		`mov  r0, r4`
		`bl  LCD_PrintInt8`
		
		`adds  r4, r4, #1 ; i++`
		`uxtb  r4, r4 ; sign extend`
		
		`cmp  r4, #a  ; i < 10`
		`bcc.n`
		`<WhileLoop+0x4>`
		
		`pop  {r4, pc};`
		`return`	

`// Conditional statement`
`void Conditional(uint8 i, uint8 j)`
`{`
`  if(j == 1)`
`  {`
`  LCD_PrintInt8(i);`
`  }`
`  else`
`  {`
`  LCD_PrintInt8(i + 1);`
`  }`
`}`
	`; prolog`
	`push  {r4, r5, r6, lr}`
	
	`mov  r4, r0`
	`mov  r5, r1`
	`cmp  r5, #1 ; j == 1`
	`bne.n`
	`<Conditional+0x12>`
	
	`mov  r0, r4`
	`bl  LCD_PrintInt8`
	`b.n`
	`<Conditional+0x1a>`
	
	`<Conditional+0x12>:`
	`; LCD_PrintInt8(i + 1)`
	`adds  r1, r4, #1`
	`uxtb  r0, r1 ; sign extend`
	`bl  LCD_PrintInt8`
	
	`<Conditional+0x1a>:`
	`; return`
	`pop  {r4, r5, r6, pc}`
		`; no prolog`
		`; if(j == 1)`
		`cmp  r1, #1`
		`beq.n`
		`<Conditional+0x8>`
		
		`adds  r0, r0, #1 ; i + 1`
		`uxtb  r0, r0`
		
		`<Conditional+0x8>:`
		`; function returns back to`
		`; caller of this function`
		`b.w  LCD_PrintInt8`
			`; same as for size optimization`

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化

`// Switch case statements`
`void SwitchCase(uint8 j)`
`{`
`  switch(j)`
`  {`
`    case 0:`
`    LCD_PrintInt8(1);`
`    break;`
 	
`    case 1:`
`    LCD_PrintInt8(2);`
`    break;`
 	
`    default:`
`    LCD_PrintInt8(0);`
`    break;`
`  }`
`}`
	`; prolog`
	`push  {r4, lr}`
	
	`; switch(j)`
	`mov  r4, r0`
	`cbz  r4, <SwitchCase+0xc>`
	`cmp  r4, #1`
	`bne.n  <SwitchCase+0x1c>`
	`b.n  <SwitchCase+0x14>`
	
	`<SwitchCase+0xc>:`
	`movs  r0, #1 ; case 0`
	`bl  LCD_PrintInt8`
	`b.n  <SwitchCase+0x24>`
	
	`<SwitchCase+0x14>:`
	`movs  r0, #2 ; case 1`
	`bl  LCD_PrintInt8`
	`b.n  <SwitchCase+0x24>`
	
	`<SwitchCase+0x1c>:`
	`movs  r0, #0 ; default`
	`bl  LCD_PrintInt8`
	`nop`
	
	`<SwitchCase+0x24>:`
	`nop`
	`pop  {r4, pc} ; return`
		`; prolog not shown`
		`; switch(j)`
		`cbz  r0, <SwitchCase+0xa>`
		`cmp  r0, #1`
		`beq.n  <SwitchCase+0xe>`
		
		`movs  r0, #0 ; default`
		`b.n  <SwitchCase+0x10>`
		
		`<SwitchCase+0xa>:`
		`movs  r0, #1 ; case 0`
		`b.n  <SwitchCase+0x10>`
		
		`<SwitchCase+0xe>:`
		`movs  r0, #2 ; case 1`
		
		`<SwitchCase+0x10>:`
		`; function returns back to`
		`; caller of this function`
		`b.w  LCD_PrintInt8`
			`; same as for size optimization`

`// Ternary operator`
`void Ternary(uint8 i)`
`{`
`  LCD_PrintInt8(`
`    (i == 1) ? 80 : 100);`
`}`
	`; prolog`
	`push  {r4, lr}`
	
	`mov  r4, r0 ; i == 1`
	`cmp  r4, #1`
	`bne.n  <Ternary+0xc>`
	
	`movs  r1, #50`
	`b.n  <Ternary+0xe>`
	
	`<Ternary+0xc>:`
	`movs  r1, #64`
		`; no prolog`
		`cmp  r0, #1`
		`beq.n  <Ternary+0xa>`
		
		`<Ternary+0x6>:`
		`movs  r0, #64  ; 0x64`
		`; function returns back to`
		`; caller of this function`
		`b.w  LCD_PrintInt8`
			`; same as for size optimization`

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化
	<Ternary+0xe>: mov r0, r1 bl LCD_PrintInt8 pop {r4, pc} ; return	<Ternary+0xa>: movs r0, #50 ; 0x50 b.n <Ternary+0x6>	

// Addition operation
int DoAdd(int x, int
y)
{
return x + y;
}
	; no prolog
	mov r2, r0
	adds r0, r2, r1
	bx lr ; return
	with result
		; no prolog
		add r0, r1
		bx lr ; return
		with result
			; same as for size
			optimization

// Subtraction
operation
int DoSub(int x, int
y)
{
return x - y;
}
	; no prolog
	mov r2, r0
	subs r0, r2, r1
	bx lr ; return
	with result
		; no prolog
		subs r0, r0, r1
		bx lr ; return
		with result
			; same as for size
			optimization

// Multiplication
int DoMul(int x, int
y)
{
return x * y;
}
	; no prolog
	mov r2, r0
	mul.w r0, r2, r1
	bx lr ; return
	with result
		; no prolog
		muls r0, r1
		bx lr ; return
		with result
			; same as for size
			optimization

// Division
int DoDiv(int x, int
y)
{
return x / y;
}
	; no prolog
	mov r2, r0
	sdiv r0, r2, r1
	bx lr ; return
	with result
		; no prolog
		sdiv r0, r0, r1
		bx lr ; return
		with result
			; same as for size
			optimization

// Modulo operator
int DoMod(int x, int
y)
{
return x % y;
}
	; no prolog
	mov r2, r0
	; truncated quotient
	sdiv r0, r2, r1
	; multiply and
	subtract instruction
	; implements
	remainder =
	; dividend -
	(quotient * divisor)
	mls r0, r1, r0, r2
	bx lr ; return
	with result
		; no prolog
		; truncated quotient
		sdiv r2, r0, r1
		; multiply and
		subtract instruction
		; implements
		remainder =
		; dividend -
		(quotient * divisor)
		mls r0, r1, r2, r0
		bx lr ; return
		with result
			; same as for size
			optimization

// Pointer
void Pointer(uint8 x,
uint8 *ptr)
{
	; prolog
	push {r4, r5, r6,
	lr}
	mov r5, r0
		; no prolog
		; *ptr = *ptr + x
		ldrb r2, [r1, #0] ;
		R1 = ptr
			; same as for size
			optimization

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化

*ptr = *ptr + x;
ptr++;
LCD_PrintInt8(*ptr);
}
	mov r4, r1
	
	; *ptr = *ptr + x;
	ldrb r0, [r4, #0]
	add r0, r5
	strb r0, [r4, #0]
	
	adds r4, r4, #1 ;
	ptr++;
	
	ldrb r0, [r4, #0]
	bl LCD_PrintInt8
	
	pop {r4, r5, r6,
	pc} ; return
		add r0, r2 ; R0 =
		x
		strb r0, [r1, #0]
		
		; ptr++
		; LCD_PrintInt8(*ptr)
		ldrb r0, [r1, #1]
		
		; function returns
		back to
		; caller of this
		function
		b.w LCD_PrintInt8	

// Function pointer
void FuncPtr(uint8 x,
void
*fptr(uint8))
{
(*fptr)(x);
}
	; prolog
	push {r4, r5, r6,
	lr}
	mov r5, r0
	mov r4, r1
	
	; (*fptr)(x)
	mov r0, r5
	; in a blx
	instruction, the
	; LS bit of the
	register
	; must be 1 to keep
	the CPU
	; in Thumb mode, or
	an
	; exception occurs
	blx r4
	
	pop {r4, r5, r6,
	pc} ; return
		; (*fptr)(x)
		; in a bx
		instruction, the LS
		; bit of the register
		must be
		; 1 to keep the CPU
		in Thumb
		; mode, or an
		exception occurs
		
		; function returns
		back to
		; caller of this
		function
		bx r1
			; same as for size
			optimization

// Packed structures
struct FOO_P
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
} __attribute__
((packed));
 	
extern struct FOO_P
myfoo_p;
 	
void
	; no prolog
	; membera = 5
	movs r0, #5
	ldr r1, [pc, #14]
	strb r0, [r1, #0]
	; memberb = 10
	movs r0, #a
	strb r0, [r1, #1]
	; memberc = 15
	movs r0, #f
	; word unaligned
	access
	str.w r0, [r1, #2]
	; memberd = 20
		; no prolog
		ldr r0, [pc, #14]
		; membera = 5
		movs r1, #5
		strb r1, [r0, #0]
		; memberb = 10
		movs r1, #a
		strb r1, [r0, #1]
		; memberc = 15
		movs r1, #f
		; word unaligned
		access
		str.w r1, [r0, #2]
		; memberd = 20
			; same as for size
			optimization

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化

PackedStruct(void)
{
  myfoo_p.membera = 5;
  myfoo_p.memberb = 10;
  myfoo_p.memberc = 15;
  myfoo_p.memberd = 20;
}
	movs r0, #14
	strh r0, [r1, #6]
	bx lr ; return
	
	.word &myfoo_p
		movs r1, #14
		strh r1, [r0, #6]
		bx lr ; return
		
		.word &myfoo_p	

// unpacked structures
struct FOO
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
};
 	
extern struct FOO myfoo;
 	
void PackedStruct(void)
{
  myfoo.membera = 5;
  myfoo.memberb = 10;
  myfoo.memberc = 15;
  myfoo.memberd = 20;
}
	; no prolog
	; membera = 5
	movs r0, #5
	ldr r1, [pc, #10]
	strb r0, [r1, #0]
	; memberb = 10
	movs r0, #a
	strb r0, [r1, #1]
	; memberc = 15
	movs r0, #f
	str r0, [r1, #4]
	; memberd = 20
	movs r0, #14
	strh r0, [r1, #8]
	bx lr ; return
	
	.word &myfoo
		; no prolog
		ldr r0, [pc, #10]
		; membera = 5
		movs r1, #5
		strb r1, [r0, #0]
		; memberb = 10
		movs r1, #a
		strb r1, [r0, #1]
		; memberc = 15
		movs r1, #f
		str r1, [r0, #4]
		; memberd = 20
		movs r1, #14
		strh r1, [r0, #8]
		bx lr ; return
		
		.word &myfoo
			; same as for size optimization

14.4 アセンブラ例、Cortex®-M0/M0+ 用 MDK

Table 32 に、Cortex®-M0 の MDK コンパイラについて、さまざまな最適化オプションのコンパイラ出力の例を示します。MDK の無料評価版では使用可能な.lst ファイルは生成されないため、例は PSoC™ Creator のアセンブラレベルのデバッグウィンドウから抽出されました。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

Table 32 Cortex®-M0/M0+ CPU 用 MDK コンパイラのコンパイラ出力の詳細

C コード	MDK, Cortex®-M0/M0+, 最適化なし	MDK, Cortex®-M0/M0+, サイズ最適化	MDK, Cortex®-M0/M0+, 速度最適化

// Calling a function
// with no arguments
LCD_Start();
	; do the function call
	bl LCD_Start
		; same as for no optimization	; same as for no optimization

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M0/M0+, 最適化なし	MDK, Cortex®-M0/M0+, サイズ最適化	MDK, Cortex®-M0/M0+, 速度最適化

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`movs r0, #80`
	`bl LCD_PrintInt8`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`movs r1, #2`
	`movs r0, #0`
	`bl LCD_Position`
		`; same as for no optimization`	`; same as for no optimization`

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`for(i = 0; i < 10; i++)`
`{`
`LCD_PrintInt8(i);`
`}`
`}`
	`; prolog`
	`push {r4, lr}`
	`mov r4, r0`
	
	`movs r4, #0 ; i = 0`
	`b.n <ForLoop+0x12>`
	
	`<ForLoop+0x8>:`
	`; do the function call with`
	`; i as the argument in R0`
	`mov r0, r4`
	`bl LCD_PrintInt8`
	
	`adds r0, r4, #1 ; i++`
	`uxtb r4, r0 ; sign extend`
	
	`<ForLoop+0x12>:`
	`cmp r4, #a ; i < 10`
	`blt.n <ForLoop+0x8>`
	
	`pop {r4, pc} ; return`
		`; prolog`
		`push {r4, lr}`
		
		`movs r4, #0 ; i = 0`
		
		`<ForLoop+0x4>:`
		`; do the function call with`
		`; i as the argument in R0`
		`mov r0, r4`
		`bl LCD_PrintInt8`
		
		`adds r4, r4, #1 ; i++`
		`uxtb r4, r4 ; sign extend`
		`cmp r4, #a ; i < 10`
		`bcc.n <ForLoop+0x4>`
		
		`pop {r4, pc} ; return`
			`; same as for size optimization`

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic Variables`
`// for details`
`uint8 i = 0;`
 	
`while (i < 10)`
`{`
`LCD_PrintInt8(i);`
	`; prolog`
	`push {r4, lr}`
	
	`movs r4, #0 ; i = 0`
	`b.n <WhileLoop+0x10>`
	
	`<WhileLoop+0x6>:`
	`; do the function call with`
		`; prolog`
		`push {r4, lr}`
		
		`movs r4, #0 ; i = 0`
		
		`<WhileLoop+0x4>:`
		`; do the function call with`
		`; i as the argument in R0`
			`; same as for size optimization`

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M0/M0+, 最適化なし	MDK, Cortex®-M0/M0+, サイズ最適化	MDK, Cortex®-M0/M0+, 速度最適化

i++;
}
	; i as the argument in R0
	mov r0, r4
	bl LCD_PrintInt8
	
	adds r0, r4, #1 ; i++
	uxtb r4, r0 ; sign extend
	
	<WhileLoop+0x10>:
	cmp r4, #a ; i < 10
	blt.n <WhileLoop+0x6>
	
	pop {r4, pc};
	return
		mov r0, r4
		bl LCD_PrintInt8
		
		adds r4, r4, #1 ; i++
		uxtb r4, r4 ; sign extend
		
		cmp r4, #a ; i < 10
		bcc.n <WhileLoop+0x4>
		
		pop {r4, pc};
		return	

// Conditional statement
void Conditional(uint8 i, uint8 j)
{
if(j == 1)
{
LCD_PrintInt8(i);
}
else
{
LCD_PrintInt8(i + 1);
}
}
	; prolog
	push {r4, r5, r6, lr}
	
	mov r4, r0
	mov r5, r1
	cmp r5, #1 ; j == 1
	bne.n <Conditional+0x12>
	
	mov r0, r4
	bl LCD_PrintInt8
	b.n <Conditional+0x1a>
	
	<Conditional+0x12>:
	; LCD_PrintInt8(i + 1)
	adds r1, r4, #1
	uxtb r0, r1 ; sign extend
	bl LCD_PrintInt8
	
	<Conditional+0x1a>:
	; return
	pop {r4, r5, r6, pc}
		; prolog
		push {r4, lr}
		
		; if(j == 1)
		cmp r1, #1
		beq.n <Conditional+0xa>
		
		adds r0, r0, #1 ; i + 1
		uxtb r0, r0
		
		<Conditional+0xa>:
		bl LCD_PrintInt8
		
		; return
		pop {r4, pc}
			; same as for size optimization

// Switch case statements void SwitchCase(uint8 j)	; prolog push {r4, lr} ; switch(j)	; prolog push {r4, lr} ; switch(j)	; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M0/M0+, 最適化なし	MDK, Cortex®-M0/M0+, サイズ最適化	MDK, Cortex®-M0/M0+, 速度最適化

{
  switch(j)
  {
  case 0:
  LCD_PrintInt8(1);
  break;
 	
  case 1:
  LCD_PrintInt8(2);
  break;
 	
  default:
  LCD_PrintInt8(0);
  break;
  }
}
	mov r4, r0
	cmp r4, #0
	beq.n
	<SwitchCase+0xe>
	cmp r4, #1
	bne.n
	<SwitchCase+0x1e>
	b.n
	<SwitchCase+0x16>
	
	<SwitchCase+0xe>:
	movs r0, #1 ; case 0
	bl LCD_PrintInt8
	b.n
	<SwitchCase+0x26>
	
	<SwitchCase+0x14>:
	movs r0, #2 ; case 1
	bl LCD_PrintInt8
	b.n
	<SwitchCase+0x26>
	
	<SwitchCase+0x1e>:
	movs r0, #0 ; default
	bl LCD_PrintInt8
	nop
	
	<SwitchCase+0x26>:
	nop
	pop {r4, pc} ; return
		cmp r0, #0
		beq.n
		<SwitchCase+0xe>
		cmp r0, #1
		beq.n
		<SwitchCase+0x12>
		
		movs r0, #0 ; default
		b.n
		<SwitchCase+0x14>
		
		<SwitchCase+0xe>:
		movs r0, #1 ; case 0
		b.n
		<SwitchCase+0x14>
		
		<SwitchCase+0x12>:
		movs r0, #2 ; case 1
		
		<SwitchCase+0x14>:
		bl LCD_PrintInt8
		pop {r4, pc} ; return	

// Ternary operator
void Ternary(uint8 i)
{
  LCD_PrintInt8(
  (i == 1) ? 80 : 100);
}
	; prolog
	push {r4, lr}
	
	mov r4, r0 ; i == 1
	cmp r4, #1
	bne.n <Ternary+0xc>
	
	movs r1, #50
	b.n <Ternary+0xe>
	
	<Ternary+0xc>:
	movs r1, #64
	
	<Ternary+0xe>:
	mov r0, r1
		; prolog
		push {r4, lr}
		
		cmp r0, #1 ; i == 1
		beq.n <Ternary+0xe>
		
		movs r0, #64
		
		<Ternary+0x8>:
		bl LCD_PrintInt8
		pop {r4, pc} ; return
		
		<Ternary+0xe>:
			; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M0/M0+, 最適化なし	MDK, Cortex®-M0/M0+, サイズ最適化	MDK, Cortex®-M0/M0+, 速度最適化
	bl LCD_PrintInt8 pop {r4, pc} ; return	movs r0, #50 b.n <Ternary+0x8>	

// Addition operation
int DoAdd(int x, int y)
{
return x + y;
}
	; no prolog
	mov r2, r0
	adds r0, r2, r1
	bx lr ; return with result
		; no prolog
		adds r0, r1
		bx lr ; return with result
			; same as for size optimization

// Subtraction operation
int DoSub(int x, int y)
{
return x - y;
}
	; no prolog
	mov r2, r0
	subs r0, r2, r1
	bx lr ; return with result
		; no prolog
		subs r0, r0, r1
		bx lr ; return with result
			; same as for size optimization

// Multiplication
int DoMul(int x, int y)
{
return x * y;
}
	; no prolog
	mov r2, r0
	muls r0, r2, r1
	bx lr ; return with result
		; no prolog
		muls r0, r1
		bx lr ; return with result
			; same as for size optimization

// Division
int DoDiv(int x, in`t y)
{
return x / y;
}
	; prolog
	push {r4, r5, r6, lr}
	mov r4, r0
	mov r5, r1
	mov r1, r5
	mov r0, r4
	bl __aeabi_idiv
	; return with result
	pop {r4, r5, r6, pc}
		; prolog
		push {r4, lr}
		bl __aeabi_idiv
		; return with result
		pop {r4, pc}
			; same as for size optimization

// Modulo operator
int DoMod(int x, int y)
{
return x % y;
}
	; prolog
	push {r4, r5, r6, lr}
	mov r4, r0
	mov r5, r1
	mov r1, r5
	mov r0, r4
	bl __aeabi_idiv
	; return with result
	mov r0, r1
	pop {r4, r5, r6, pc}
		; prolog
		push {r4, lr}
		bl __aeabi_idiv
		; return with result
		mov r0, r1
		pop {r4, pc}
			; same as for size optimization

// Pointer void Pointer(uint8 x, uint8 *ptr)	; prolog push {r4, r5, r6, lr}	; prolog push {r4, lr}	; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M0/M0+, 最適化なし	MDK, Cortex®-M0/M0+, サイズ最適化	MDK, Cortex®-M0/M0+, 速度最適化

{
  *ptr = *ptr + x;
  ptr++;
  LCD_PrintInt8(*ptr);
}
	mov r5, r0
	mov r4, r1
	
	; *ptr = *ptr + x;
	ldrb r0, [r4, #0]
	adds r0, r0, r5
	strb r0, [r4, #0]
	
	adds r4, r4, #1 ;
	ptr++;
	
	ldrb r0, [r4, #0]
	bl LCD_PrintInt8
	
	pop {r4, r5, r6,
	pc} ; return
		; *ptr = *ptr + x
		ldrb r2, [r1, #0] ;
		R1 = ptr
		adds r0, r2, r0 ;
		R0 = x
		strb r0, [r1, #0]
		
		; ptr++
		; LCD_PrintInt8(*ptr)
		ldrb r0, [r1, #1]
		bl LCD_PrintInt8
		
		pop {r4, pc} ;
		return	

// Function pointer
void FuncPtr(uint8 x,
  void
*fptr(uint8))
{
  (*fptr)(x);
}
	; prolog
	push {r4, r5, r6,
	lr}
	mov r5, r0
	mov r4, r1
	
	; (*fptr)(x)
	mov r0, r5
	; in a blx
	instruction, the
	; LS bit of the
	register
	; must be 1 to keep
	the CPU
	; in Thumb mode, or
	an
	; exception occurs
	blx r4
	
	pop {r4, r5, r6,
	pc} ; return
		; (*fptr)(x)
		; in a bx
		instruction, the LS
		; bit of the register
		must be
		; 1 to keep the CPU
		in Thumb
		; mode, or an
		exception occurs
		
		; function returns
		back to
		; caller of this
		function
		bx r1
			; same as for size
			optimization

// Packed structures
struct FOO_P
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
} __attribute__
((packed));
 	
extern struct FOO_P
myfoo_p;
	; prolog
	push {r4, lr}
	; membera = 5
	movs r0, #5
	ldr r1, [pc, #18]
	strb r0, [r1, #0]
	; memberb = 10
	movs r0, #a
	strb r0, [r1, #1]
	; memberc = 15
	adds r1, r1, #2
	movs r0, #f
	bl __aeabi_uwrite4
		; prolog
		push {r4, lr}
		ldr r4, [pc, #18]
		; membera = 5
		movs r0, #5
		strb r0, [r4, #0]
		; memberb = 10
		movs r0, #a
		strb r0, [r4, #1]
		; memberc = 15
		adds r1, r4, #2
		movs r0, #f
		bl __aeabi_uwrite4
			; same as for size
			optimization

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M0/M0+, 最適化なし	MDK, Cortex®-M0/M0+, サイズ最適化	MDK, Cortex®-M0/M0+, 速度最適化

void
PackedStruct(void)
{
  myfoo_p.membera =
5;
  myfoo_p.memberb =
10;
  myfoo_p.memberc =
15;
  myfoo_p.memberd =
20;
}
	; memberd = 20
	movs r1, #14
	ldr r0, [pc,#8]
	strb r1, [r0, #6]
	movs r1, #0
	strb r1, [r0, #7]
	pop {r4, pc} ;
	return
	
	.word &myfoo_p
		; memberd = 20
		movs r0, #14
		strb r1, [r4, #6]
		movs r0, #0
		strb r1, [r4, #7]
		pop {r4, pc} ;
		return
		
		.word &myfoo_p	

// unpacked structures
struct FOO
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
};
 	
extern struct FOO
myfoo;
 	
void
PackedStruct(void)
{
  myfoo.membera = 5;
  myfoo.memberb = 10;
  myfoo.memberc = 15;
  myfoo.memberd = 20;
}
	; no prolog
	; membera = 5
	movs r0, #5
	ldr r1, [pc, #10]
	strb r0, [r1, #0]
	; memberb = 10
	movs r0, #a
	strb r0, [r1, #1]
	; memberc = 15
	movs r0, #f
	str r0, [r1, #4]
	; memberd = 20
	movs r0, #14
	strh r0, [r1, #8]
	bx lr ; return
	
	.word &myfoo
		; same as for no
		optimization
			; no prolog
			ldr r0, [pc, #10]
			; membera = 5
			movs r1, #5
			strb r1, [r0, #0]
			; memberb = 10
			movs r1, #a
			strb r1, [r0, #1]
			; memberc = 15
			movs r1, #f
			str r1, [r0, #4]
			; memberd = 20
			movs r1, #14
			strh r1, [r0, #8]
			bx lr ; return
			
			.word &myfoo

14.5 アセンブラの例、Cortex®-M3 用 IAR

14.5.1 Cortex®-M3 用 IAR、なし, サイズ, および速度最適化

Table 33 に、Cortex®-M3 用の IAR コンパイラについて、さまざまな最適化オプションのコンパイラ出力の例を示します。これらの例は、コンパイラによって生成された.lst ファイルから抽出されたものです。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

14 付録: コンパイラ出力の詳細

Table 33 Cortex®-M3 CPU 用 IAR コンパイラのコンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`BL LCD_Start`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`; conditional flags are NOT`
	`; updated by mov`
	`MOVS R0,#+128`
	`BL LCD_PrintInt8`
		`; same as for no optimization`	`; same as for size optimization`

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`MOVS R1,#+2`
	`MOVS R0,#+0`
	`BL LCD_Position`
		`; same as for no optimization`	`; same as for size optimization`

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`for(i = 0; i < 10; i++)`
`{`
`LCD_PrintInt8(i);`
`}`
`}`
	`; function prolog`
	`; i is saved on the stack`
	`PUSH {R4,LR}`
	`; R4 = i`
	`MOVS R4,#+0`
	`; do the function call with`
	`; i as the argument in R0`
	`??ForLoop_0:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; check i not equal to 10`
	`CMP R0,#+10`
	`BGE.N ??ForLoop_1`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
	`ADDS R4,R4,#+1 ; i++`
	`B.N ??ForLoop_0`
	`; function epilog`
	`??ForLoop_1:`
	`POP {R4,PC}; return`
		`; function prolog`
		`PUSH {R4,LR}`
		`B.N ?Subroutine0`
		`?Subroutine0: (+1)`
		`MOVS R4,#+0`
		`?Subroutine0_0: (+1)`
		`UXTB R0,R4`
		`BL LCD_PrintInt8`
		`ADDS R4,R4,#+1 ; i++`
		`UXTB R0,R4`
		`CMP R0,#+10 ; i = 10?`
		`BLT.N ??Subroutine0_0`
		`POP {R4,PC} ; return`
			`; function prolog`
			`PUSH {R4,LR}`
			`MOVS R4,#+0`
			`UXTB R0,R4`
			`BL LCD_PrintInt8`
			`ADDS R4,R4,#+1 ; i++`
			`UXTB R0,R4`
			`CMP R0,#+10 ; i = 10?`
			`BLT.N ??ForLoop_0`
			`POP {R4,PC} ; return`

`// While loop`
`// i is type automatic, see`
	`; prolog not shown`
	`; i = 0`
	`MOVS R4,#+0`
		`; prolog not shown`
		`; optimizes to match the for ; loop`
			`; function prolog`
			`PUSH {R4,LR}`
			`MOVS R4,#+0`

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化

// Accessing Automatic Variables // for details
uint8 i = 0;
 	
while (i < 10)
{
  LCD_PrintInt8(i);
  i++;
}
	??WhileLoop_0:
	MOVS R0,R4
	UXTB R0,R0
	; check i not equal to 10
	CMP R0,#+10
	BGE.N ??WhileLoop_1
	
	LCD_PrintInt8:
	MOVS R0,R4
	UXTB R0,R0
	; LCD_PrintInt8(i)
	BL LCD_PrintInt8
	
	ADDS R4,R4,#+1 ; i++
	B.N ??WhileLoop_0
	
	??WhileLoop_1:
	POP {R4,PC}
		function
		REQUIRE ?Subroutine0
		;; Fall through to label ?Subroutine0
		
		; epilog not shown
			UXTB R0,R4
			BL LCD_PrintInt8
			
			ADDS R4,R4,#+1 ; i++
			UXTB R0,R4
			CMP R0,#+10 ; i = 10?
			BLT.N ??WhileLoop_0
			
			POP {R4,PC} ; return

// Conditional statement
void Conditional(uint8 i, uint8 j)
{
  if(j == 1)
  {
 	
LCD_PrintInt8(i);
  }
  else
  {
  LCD_PrintInt8(i + 1);
  }
}
	; prolog not shown
	; if(j == 1)
	Conditional:
	PUSH {R3-R5,LR}
	MOVS R4,R0
	MOVS R5,R1
	
	MOVS R0,R5
	UXTB R0,R0
	CMP R0,#+1
	BNE.N ??Conditional_0
	
	; LCD_PrintInt8(i);
	MOVS R0,R4
	UXTB R0,R0
	BL LCD_PrintInt8
	B.N ??Conditional_1
	
	; LCD_PrintInt8(i + 1);
	??Conditional_0:
	ADDS R0,R4,#+1
	UXTB R0,R0
	BL LCD_PrintInt8
	
	??Conditional_1:
	POP {R0,R4,R5,PC}
		; no prolog
		; if(j == 1)
		CMP R1,#+1
		; if-then-then instruction,
		; if not equal, then add 1 to ; i and call LCd_PrintInt8
		; with i
		ITT NE
		ADDNE R0,R0,#+1
		UXTBNE R0,R0
		
		B.W LCD_PrintInt8
			; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化

// Switch case statements
void SwitchCase(uint8 j)
{
  switch(j)
  {
  case 0:
 	
LCD_PrintInt8(1);
  break;
 	
  case 1:
 	
LCD_PrintInt8(2);
  break;
 	
  default:
 	
LCD_PrintInt8(0);
  break;
  }
}
	; prolog not shown
	; switch(j)
	UXTB R0,R0
	CMP R0,#+0
	BEQ.N ??SwitchCase_0
	CMP R0,#+1
	BEQ.N ??SwitchCase_1
	B.N ??SwitchCase_2
	
	; case 0
	??SwitchCase_0:
	MOVS R0,#+1
	BL LCD_PrintInt8
	B.N ??SwitchCase_3
	
	; case 1
	??SwitchCase_1:
	MOVS R0,#+2
	BL LCD_PrintInt8
	B.N ??SwitchCase_3
	
	; default
	??SwitchCase_2:
	MOVS R0,#+0
	BL LCD_PrintInt8
	; no epilog
		; no prolog
		; switch(j)
		CBZ.N R0,??SwitchCase_0
		CMP R0,#+1
		BEQ.N ??SwitchCase_1
		B.N ??SwitchCase_2
		
		; case 0
		MOVS R0,#+1
		B.N ??SwitchCase_3
		
		; case 1
		MOVS R0,#+2
		B.N ??SwitchCase_3
		
		; case 2
		MOVS R0,#+0
		; default
		B.W LCD_PrintInt8
			; same as for size optimization

// Ternary operator
void Ternary(uint8 i)
{
  LCD_PrintInt8(
  (i == 1) ? 80 : 100);
}
	; prolog not shown
	; check value of i
	CMP R0,#+1
	; branch if not equal
	BNE.N ??Ternary_0
	MOVS R0,#+80
	B.N ??Ternary_1
	
	??Ternary_0:
	MOVS R0,#+100
	??Ternary_1:
	UXTB R0,R0
	BL LCD_PrintInt8
	; no epilog
		; no prolog
		; check value of i
		CMP R0, #1
		
		; “ite” stands for if-then-
		; else instruction
		; “ne” condition checks
		; if the previous compare
		; instruction has cleared the
		; “equal to” flag
		ITE NE
		
		; mov if the result of the
		; previous “ite”
			; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化

		instruction is
		; “not equal”
		MOVNE R0, #100
		
		; mov if the result of the
		; previous “ite” instruction is
		; “equal”
		MOVEQ R0, #80
		
		; no epilog
		B.W
		LCD_PrintInt8	

// Addition operation
int DoAdd(int x, int y)
{
return x + y;
}
	; no prolog
	ADDS R0,R1,R0
	BX LR 3 ; return value
		; same as for no optimization	; same as for size optimization

// Subtraction operation
int DoSub(int x, int y)
{
return x - y;
}
	; no prolog
	SUBS R0,R0,R1
	BX LR ; return value
		; same as for no optimization	; same as for size optimization

// Multiplication
int DoMul(int x, int y)
{
return x * y;
}
	; no prolog
	MULS R0,R0,R1
	BX LR ; return value
		; same as for no optimization	; same as for size optimization

// Division
int DoDiv(int x, int y)
{
return x / y;
}
	; no prolog
	SDIV R0,R0,R1
	BX LR ; return value
		; same as for no optimization	; same as for size optimization

// Modulo operator
int DoMod(int x, int y)
{
return x % y;
}
	; no prolog
	SDIV R2,R0,R1
	; multiply and subtract instruction
	; implements remainder =
	; dividend - (quotient * divisor)
	MLS R0,R1,R2,R0
		; same as for no optimization	; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化
	BX LR ; return value		

// Pointer
void Pointer(uint8 x, uint8 *ptr)
{
*ptr = *ptr + x;
ptr++;
 	
LCD_PrintInt8(*ptr);
}
	; *ptr = *ptr + x
	LDRB R0,[R1, #+0]
	ADDS R0,R4,R0
	STRB R0,[R1, #+0]
	
	; ptr++
	; LCD_PrintInt8(*ptr)
	ADDS R5,R1,#+1
	LDRB R0,[R5, #+0]
	BL LCD_PrintInt8
		; *ptr = *ptr + x
		LDRB R2, [R1, #+0] ; R1 = ptr
		ADDS R0, R0, R2 ; R0 = x
		STRB R0, [R1, #+0]
		
		; ptr++
		; LCD_PrintInt8(*ptr)
		LDRB R0, [R1, #+1]
		B.W LCD_PrintInt8
			; same as for size optimization

// Function pointer
void FuncPtr(uint8 x,
void *fptr(uint8))
{
(*fptr)(x);
}
	; (*fptr)(x)
	PUSH {R3-R5,LR}
	MOVS R4,R0
	MOVS R5,R1
	
	MOVS R0,R4
	UXTB R0,R0
	; in a blx instruction, the
	; LS bit of the register
	; must be 1 to keep the CPU
	; in Thumb mode, or an
	; exception occurs
	BLX R5
		; (*fptr)(x)
		; in a blx instruction, the LS
		; bit of the register must be
		; 1 to keep the CPU in Thumb
		; mode, or an exception occurs
		BX R1
			; same as for size optimization

// Packed structures
struct FOO_P
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
} __attribute__ ((packed));
 	
extern struct FOO_P myfoo_p;
 	
void PackedStruct(void)
{
myfoo_p.membera = 5;
myfoo_p.memberb = 10;
	; myfoo.membera = 5
	LDR.N R0,??DataTable2
	MOVS R1,#+5
	STRB R1,[R0, #+0]
	; myfoo_p.memberb = 10;
	MOVS R1,#+10
	STRB R1,[R0, #+1]
	; myfoo_p.memberc = 15;
	MOVS R1,#+15
	STR R1,[R0, #+2]
	; myfoo_p.memberd = 20;
	MOVS R1,#+20
	STRH R1,[R0, #+6]
	
	BX LR
		; myfoo.membera = 5
		LDR.N R0,??DataTable2
		MOVS R1,#+5
		STRB R1,[R0, #+0]
		; myfoo_p.memberb = 10;
		MOVS R2,#+10
		; myfoo_p.memberc = 15;
		MOVS R3,#+15
		; myfoo_p.memberd = 20;
		MOVS R1,#+20
		STRB R2,[R0, #+1]
		STR R3,[R0, #+2]
		STRH R1,[R0, #+6]
		
		BX LR
			; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	MDK, Cortex®-M3, 最適化なし	MDK, Cortex®-M3, サイズ最適化	MDK, Cortex®-M3, 速度最適化

myfoo_p.memberc = 15;
myfoo_p.memberd = 20;
}			

// unpacked structures
struct FOO
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
};
 	
extern struct FOO myfoo;
 	
void PackedStruct(void)
{
myfoo.membera = 5;
myfoo.memberb = 10;
myfoo.memberc = 15;
myfoo.memberd = 20;
}
	; myfoo.membera = 5
	LDR.N R0,??DataTable2_1
	MOVS R1,#+5
	STRB R1,[R0, #+0]
	; myfoo.memberb = 10;
	MOVS R1,#+10
	STRB R1,[R0, #+1]
	; myfoo.memberc = 15;
	MOVS R1,#+15
	STR R1,[R0, #+4]
	; myfoo.memberd = 20;
	MOVS R1,#+20
	STRH R1,[R0, #+8]
	
	BX LR
		; myfoo.membera = 5
		LDR.N R0,??DataTable2_1
		MOVS R1,#+5
		STRB R1,[R0, #+0]
		; myfoo.memberb = 10;
		MOVS R2,#+10
		; myfoo.memberc = 15;
		MOVS R3,#+15
		; myfoo.memberd = 20;
		MOVS R1,#+20
		STRB R2,[R0, #+1]
		STR R3,[R0, #+4]
		STRH R1,[R0, #+8]
		
		BX LR
			; same as for size optimization

14.5.2 Cortex®-M3 用 IAR、低, 中, およびバランス最適化

Table 34 に、Cortex®-M3 用の IAR コンパイラについて、さまざまな最適化オプションのコンパイラ出力の例を示します。これらの例は、コンパイラによって生成された.lst ファイルから抽出されたものです。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

Table 34 Cortex®-M3 CPU 用 IAR コンパイラのコンパイラ出力の詳細

C コード	IAR, Cortex®-M3, 低最適化	IAR, Cortex®-M3, 中最適化	IAR, Cortex®-M3, バランス最適化

// Calling a function
// with no arguments
LCD_Start();
	; do the function call
	BL LCD_Start
		; same as for no optimization	; same as for no optimization

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M3, 低最適化	IAR, Cortex®-M3, 中最適化	IAR, Cortex®-M3, バランス最適化

// Calling a function with
// one argument
LCD_PrintInt8(128);
 	
	; R0 = first argument
	; conditional flags are NOT
	; updated by mov
	MOVS R0,#+128
	BL LCD_PrintInt8
		; same as for no optimization	; same as for size optimization

// Calling a function with
// two arguments
LCD_Position(0, 2);
	; R0 = first argument
	; R1 = second argument
	MOVS R1,#+2
	MOVS R0,#+0
	BL LCD_Position
		; same as for low optimization	; same as for size optimization

// For loop:
void ForLoop(uint8 i)
{
for(i = 0; i < 10; i++)
{
 	
LCD_PrintInt8(i);
}
}
	; function prolog
	; i is saved on the stack
	PUSH {R4,LR}
	0; i++)
	MOVS R4,#+0
	B.N ??ForLoop_0
	
	??ForLoop_1:
	MOVS R0,R4
	UXTB R0,R0
	BL LCD_PrintInt8
	
	ADDS R4,R4,#+1
	??ForLoop_0
	MOVS R0,R4
	UXTB R0,R0
	CMP R0,#+10
	BLT.N ??ForLoop_1
	
	; function epilog
	POP {R4,PC};
	return
		; same as for low optimization	; function prolog
			PUSH {R4,LR}
			
			B.N ?Subroutine0
			
			?Subroutine0: (+1)
			MOVS R4,#+0
			
			?Subroutine0_0: (+1)
			UXTB R0,R4
			BL
			LCD_PrintInt8
			ADDS R4,R4,#+1 ;
			i++
			UXTB R0,R4
			CMP R0,#+10 ; i
			= 10?
			
			BLT.N ??Subroutine0
			_0
			POP {R4,PC} ;
			return

// While loop
// i is type automatic, see
// Accessing Automatic Variables
// for details
uint8 i = 0;
 	
while (i < 10)
{
 	
LCD_PrintInt8(i);
i++;
}
	; function prolog
	; i is saved on the stack PUSH {R4,LR}
	
	MOVS R4,#+0
	B.N ??WhileLoop_0
	
	??WhileLoop_1:
	MOVS R0,R4
	UXTB R0,R0
	BL LCD_PrintInt8
	
	ADDS R4,R4,#+1
	
	??WhileLoop_0:
		; same as for low optimization	; prolog not shown
			; optimizes to match the for ; loop function
			REQUIRE ?Subroutine0
			;; Fall through to label ?Subroutine0
			
			; epilog not shown

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M3, 低最適化	IAR, Cortex®-M3, 中最適化	IAR, Cortex®-M3, バランス最適化

	MOVS R0,R4
	UXTB R0,R0
	CMP R0,#+10
	BLT.N ??WhileLoop_1
	; function epilog
	POP {R4,PC};
	return		

// Conditional statement
void Conditional(uint8 i, uint8 j)
{
if(j == 1)
{
 	
LCD_PrintInt8(i);
}
else
{
 	
LCD_PrintInt8(i + 1);
}
}
	PUSH {R7,LR}
	; if(j == 1)
	UXTB R1,R1
	CMP R1,#+1
	BNE.N ??Conditional_0
	; LCD_PrintInt8(i);
	UXTB R0,R0
	BL LCD_PrintInt8
	B.N ??Conditional_1
	; LCD_PrintInt8(i + 1);
	Conditional_0:
	ADDS R0,R0,#+1
	UXTB R0,R0
	BL LCD_PrintInt8
	
	??Conditional_1:
	POP {R0,PC};
	return
		; no prolog
		; if(j == 1)
		CMP R1,#+1
		BNE.N ??Conditional_0
		
		; LCD_PrintInt8(i);
		B.W LCD_PrintInt8
		; LCD_PrintInt8(i + 1);
		??Conditional_0:
		ADDS R0,R0,#+1
		UXTB R0,R0
		B.W LCD_PrintInt8
			; no prolog
			; if(j == 1)
			CMP R1,#+1
			; if-then-then instruction,
			; if not equal, then add 1 to ; i and call LCd_PrintInt8
			; with i
			ITT NE
			ADDNE R0,R0,#+1
			UXTBNE R0,R0
			
			B.W LCD_PrintInt8

// Switch case statements
void SwitchCase(uint8 j)
{
switch(j)
{
case 0:
 	
LCD_PrintInt8(1);
break;
 	
case 1:
 	
LCD_PrintInt8(2);
break;
 	
default:
 	
LCD_PrintInt8(0);
break;
	; prolog not shown
	; switch(j)
	UXTB R0,R0
	CMP R0,#+0
	BEQ ??SwitchCase_0
	CMP R0,#+1
	BEQ ??SwitchCase_1
	B.N ??SwitchCase_2
	; case 0
	??SwitchCase_0:
	MOVS R0,#+1
	BL LCD_PrintInt8
	B.N ??SwitchCase_3
	
	; case 1
	??SwitchCase_1:
	MOVS R0,#+2
	BL LCD_PrintInt8
		; no prolog
		; switch(j)
		CMP R0,#+0
		BEQ.N ??SwitchCase_0
		CMP R0,#+1
		BEQ.N ??SwitchCase_1
		B.N ??SwitchCase_2
		
		case 0:
		??SwitchCase_0: (+1)
		MOVS R0,#+1
		B.W LCD_PrintInt8
		
		; case 1:
		??SwitchCase_1:
		MOVS R0,#+2
		B.W LCD_PrintInt8
		
		; default:
			; no prolog
			; switch(j)
			CBZ.N R0,??SwitchCase_0
			CMP R0,#+1
			BEQ.N ??SwitchCase_1
			B.N ??SwitchCase_2
			
			; case 0
			MOVS R0,#+1
			B.N ??SwitchCase_3
			
			; case 1
			MOVS R0,#+2
			B.N ??SwitchCase_3
			
			; case 2
			MOVS R0,#+0

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M3, 低最適化	IAR, Cortex®-M3, 中最適化	IAR, Cortex®-M3, バランス最適化
} }	B.N ??SwitchCase_3 ; default ??SwitchCase_2: MOVS R0,#+0 BL LCD_PrintInt8 ; epilog not shown	??SwitchCase_2: MOVS R0,#+0 B.W LCD_PrintInt8	; default B.W LCD_PrintInt8

// Ternary operator
void Ternary(uint8 i)
{
LCD_PrintInt8((i == 1) ? 80 : 100);
}
	; prolog not shown
	; check value of i
	CMP R0,#+1
	; branch if not equal
	BNE.N ??Ternary_0
	MOVS R0,#+80
	B.N ??Ternary_1
	
	??Ternary_0:
	MOVS R0,#+100
	??Ternary_1:
	UXTB R0,R0
	BL LCD_PrintInt8
	; no epilog
		; prolog not shown
		; check value of i
		CMP R0,#+1
		; branch if not equal
		BNE.N ??Ternary_0
		MOVS R0,#+80
		B.N ??Ternary_1
		
		??Ternary_0:
		MOVS R0,#+100
		??Ternary_1:
		BL LCD_PrintInt8
		; no epilog
			; prolog not shown
			; check value of i
			CMP R0,#+1
			; “ite” stands for if-then-
			; else instruction
			; “eq” condition checks
			; if the previous compare
			; instruction has set the
			; “equal to” flag
			ITE EQ
			MOVEQ R0,#+80
			MOVNE R0,#+100
			B.W LCD_PrintInt8

// Addition operation
int DoAdd(int x, int y)
{
return x + y;
}
	; no prolog
	ADDS R0,R1,R0
	BX LR ; return value
		; same as for low optimization	; same as for low optimization

// Subtraction operation
int DoSub(int x, int y)
{
return x - y;
}
	; no prolog
	SUBS R0,R0,R1
	BX LR ; return value
		; same as for low optimization	; same as for low optimization

// Multiplication
int DoMul(int x, int y)
{
return x * y;
}
	; no prolog
	MULS R0,R1,R0
	BX LR ; return value
		; same as for low optimization	; same as for low optimization

// Division
int DoDiv(int x, int y)
{
	; no prolog
	SDIV R0,R0,R1
	BX LR ; return value
		; same as for low optimization	; same as for low optimization

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M3, 低最適化	IAR, Cortex®-M3, 中最適化	IAR, Cortex®-M3, バランス最適化
return x / y; }			

// Modulo operator
int DoMod(int x,
int y)
{
return x % y;
}
	; no prolog
	SDIV R2,R0,R1
	; multiply and
	subtract instruction
	; implements remainder
	=
	; dividend - (quotient
	* divisor)
	MLS R0,R1,R2,R0
	BX LR ;
	return value
		; same as for low
		optimization
			; same as for low
			optimization

// Pointer
void Pointer(uint8
x, uint8 *ptr)
{
*ptr = *ptr + x;
ptr++;
 	
LCD_PrintInt8(*ptr
);
}
	; *ptr = *ptr + x
	LDRB R2,[R1, #+0]
	ADDS R0,R0,R2
	STRB R0,[R1, #+0]
	
	; ptr++
	; LCD_PrintInt8(*ptr)
	ADDS R0,R1,#+1
	LDRB R0,[R0, #+0]
	BL LCD_PrintInt8
		; *ptr = *ptr + x
		LDRB R2, [R1,
		#+0] ; R1 = ptr
		ADDS R0, R0, R2 ;
		R0 = x
		STRB R0, [R1, #+0]
		
		; ptr++
		; LCD_PrintInt8(*ptr)
		LDRB R0, [R1, #+1]
		B.W LCD_PrintInt8
			; same as for medium
			optimization

// Function
pointer
void FuncPtr(uint8
x,
void
*fptr(uint8))
{
(*fptr)(x);
}
	; (*fptr)(x)
	PUSH {R7,LR}
	UXTB R0,R0
	BLX R1
	POP {R0,PC}
		; (*fptr)(x)
		; in a blx
		instruction, the LS
		; bit of the register
		must be
		; 1 to keep the CPU in
		Thumb
		; mode, or an
		exception occurs
		BX R1
			; same as for medium
			optimization

// Packed
structures
struct FOO_P
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
} __attribute__
((packed));
 	
extern struct
FOO_P myfoo_p;
 	
void
	; myfoo.membera = 5
	LDR.N
	R0,??DataTable2
	MOVS R1,#+5
	STRB R1,[R0, #+0]
	; myfoo_p.memberb =
	10;
	MOVS R1,#+10
	STRB R1,[R0, #+1]
	; myfoo_p.memberc =
	15;
	MOVS R1,#+15
	STR R1,[R0, #+2]
	; myfoo_p.memberd =
	20;
		; same as for low
		optimization
			; myfoo.membera = 5
			LDR.N
			R0,??DataTable2
			MOVS R1,#+5
			STRB R1,[R0, #+0]
			; myfoo_p.memberb =
			10;
			MOVS R2,#+10
			; myfoo_p.memberc =
			15;
			MOVS R3,#+15
			; myfoo_p.memberd =
			20;
			MOVS R1,#+20
			STRB R2,[R0, #+1]

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M3, 低最適化	IAR, Cortex®-M3, 中最適化	IAR, Cortex®-M3, バランス最適化

PackedStruct(void)
{
  myfoo_p.membera
= 5;
  myfoo_p.memberb
= 10;
  myfoo_p.memberc
= 15;
  myfoo_p.memberd
= 20;
}
	MOVS R1,#+20
	STRH R1,[R0, #+6]
	
	BX LR
			STR R3,[R0, #+2]
			STRH R1,[R0, #+6]
			
			BX LR

// unpacked
structures
struct FOO
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
};
 	
extern struct FOO
myfoo;
 	
void
PackedStruct(void)
{
  myfoo.membera =
5;
  myfoo.memberb =
10;
  myfoo.memberc =
15;
  myfoo.memberd =
20;
}
	; myfoo.membera = 5
	LDR.N
	R0,??DataTable2_1
	MOVS R1,#+5
	STRB R1,[R0, #+0]
	; myfoo.memberb = 10;
	MOVS R1,#+10
	STRB R1,[R0, #+1]
	; myfoo.memberc = 15;
	MOVS R1,#+15
	STR R1,[R0, #+4]
	; myfoo.memberd = 20;
	MOVS R1,#+20
	STRH R1,[R0, #+8]
	
	BX LR
		; same as for low
		optimization
			; myfoo.membera = 5
			LDR.N
			R0,??DataTable2_1
			MOVS R1,#+5
			STRB R1,[R0, #+0]
			; myfoo.memberb = 10;
			MOVS R2,#+10
			; myfoo.memberc = 15;
			MOVS R3,#+15
			; myfoo.memberd = 20;
			MOVS R1,#+20
			STRB R2,[R0, #+1]
			STR R3,[R0, #+4]
			STRH R1,[R0, #+8]
			
			BX LR

14.6 アセンブラ例、Cortex®-M0/M0+ 用 IAR

Table 35 に、Cortex®-M0/M0+用の IAR コンパイラについて、さまざまな最適化オプションのコンパイラ出力の例を示します。これらの例は、コンパイラによって生成された.lst ファイルから抽出されたものです。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

14 付録: コンパイラ出力の詳細

Table 35 Cortex®-M0/M0+ CPU 用 IAR コンパイラのコンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`BL LCD_Start`
		`; same as for no optimization`	`; same as for no optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`; conditional flags are NOT`
	`; updated by mov`
	`MOVS R0,#+128`
	`BL LCD_PrintInt8`
		`; same as for no optimization`	`; same as for size optimization`

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`MOVS R1,#+2`
	`MOVS R0,#+0`
	`BL LCD_Position`
		`; same as for no optimization`	`; same as for size optimization`

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`  for(i = 0; i < 10; i++)`
`  {`
 	
`LCD_PrintInt8(i);`
`  }`
`}`
	`; function prolog`
	`; i is saved on the stack`
	`PUSH {R4,LR}`
	`; R4 = i`
	`MOVS R4,#+0`
	`; do the function call with`
	`; i as the argument in R0`
	`??ForLoop_0:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; check i not equal to 10`
	`CMP R0,#+10`
	`BGE ??ForLoop_1`
	
	`MOVS R0,R4`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
	
	`ADDS R4,R4,#+1 ; i++`
	`B ??ForLoop_0`
	
	`; function epilog`
	`??ForLoop_1:`
	`POP {R4,PC}; return`
		`; function prolog`
		`PUSH {R4,LR}`
		
		`B.N ?Subroutine0`
		
		`?Subroutine0:`
		`MOVS R4,#+0`
		
		`?Subroutine0_0:`
		`UXTB R0,R4`
		`BL LCD_PrintInt8`
		`ADDS R4,R4,#+1 ; i++`
		`UXTB R0,R4`
		`CMP R0,#+10 ; i = 10?`
		
		`BLT ??Subroutine0_0`
		`POP {R4,PC} ; return`
			`; function prolog`
			`PUSH {R4,LR}`
			`MOVS R4,#+0`
			
			`UXTB R0,R4`
			`BL LCD_PrintInt8`
			
			`ADDS R4,R4,#+1 ; i++`
			`UXTB R0,R4`
			`CMP R0,#+10 ; i = 10?`
			`BLT ??ForLoop_0`
			
			`POP {R4,PC} ; return`

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic Variables`
`// for details`
`uint8 i = 0;`
 	
`while (i < 10)`
`{`
`  LCD_PrintInt8(i);`
`  i++;`
`}`
	`; prolog not shown`
	`; i = 0`
	`MOVS R4,#+0`
	
	`??WhileLoop_0:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; check i not equal to 10`
	`CMP R0,#+10`
	`BGE ??WhileLoop_1`
	
	`LCD_PrintInt8:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; LCD_PrintInt8(i)`
	`BL LCD_PrintInt8`
	
	`ADDS R4,R4,#+1 ; i++`
	`B ??WhileLoop_0`
	
	`??WhileLoop_1:`
	`POP {R4,PC}`
		`; optimizes to match the for ; loop function`
		`REQUIRE ?Subroutine0`
		`;;  Fall through to label ?Subroutine0`
			`; function prolog`
			`PUSH {R4,LR}`
			`MOVS R4,#+0`
			
			`??WhileLoop_0:`
			`UXTB R0,R4`
			`BL LCD_PrintInt8`
			
			`ADDS R4,R4,#+1 ; i++`
			`UXTB R0,R4`
			`CMP R0,#+10 ; i = 10?`
			`BLT ??WhileLoop_0`
			
			`POP {R4,PC} ; return`

`// Conditional statement`
`void Conditional(uint8 i, uint8 j)`
`{`
`  if(j == 1)`
`  {`
 	
`LCD_PrintInt8(i);`
`  }`
`  else`
`  {`
`  LCD_PrintInt8(i + 1);`
`  }`
`}`
	`; prolog not shown`
	`; if(j == 1)`
	`Conditional:`
	`PUSH {R3-R5,LR}`
	`MOVS R5,R0`
	`MOVS R4,R1`
	
	`MOVS R0,R4`
	`UXTB R0,R0`
	`CMP R0,#+1`
	`BNE ??Conditional_0`
	
	`; LCD_PrintInt8(i);`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
	`B ??Conditional_1`
	
	`; LCD_PrintInt8(i + 1);`
	`??Conditional_0:`
	`ADDS R0,R4,#+1`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
		`; prolog not shown`
		`; if(j == 1)`
		`CMP R1,#+1`
		`BEQ ??Conditional_0`
		`ADDS R0,R0,#+1`
		`UXTB R0,R0`
		
		`??Conditional_0:`
		`BL LCD_PrintInt8`
			`; same as for size optimization`

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化
	??Conditional_1: POP {R0,R4,R5,PC}		

// Switch case statements
void SwitchCase(uint8 j)
{
  switch(j)
  {
  case 0:
 	
LCD_PrintInt8(1);
  break;
 	
  case 1:
 	
LCD_PrintInt8(2);
  break;
 	
  default:
 	
LCD_PrintInt8(0);
  break;
  }
}
	; prolog not shown
	; switch(j)
	UXTB R0,R0
	CMP R0,#+0
	BEQ ??SwitchCase_0
	CMP R0,#+1
	BEQ ??SwitchCase_1
	B ??SwitchCase_2
	
	; case 0
	??SwitchCase_0:
	MOVS R0,#+1
	BL LCD_PrintInt8
	B ??SwitchCase_3
	
	; case 1
	??SwitchCase_1:
	MOVS R0,#+2
	BL LCD_PrintInt8
	B ??SwitchCase_3
	
	; default
	??SwitchCase_2:
	MOVS R0,#+0
	BL LCD_PrintInt8
	; no epilog
		; prolog not shown
		; switch(j)
		CMP R0,#+0
		BEQ ??SwitchCase_0
		CMP R0,#+1
		BEQ ??SwitchCase_1
		B ??SwitchCase_2
		
		; case 0
		??SwitchCase_0:
		MOVS R0,#+1
		B ??SwitchCase_3
		
		; case 1
		??SwitchCase_1:
		MOVS R0,#+2
		B ??SwitchCase_3
		
		; case 2
		??SwitchCase_2:
		MOVS R0,#+0
		; default
		??SwitchCase_3:
		BL LCD_PrintInt8
		; epilog not shown
			; same as for size optimization

// Ternary operator
void Ternary(uint8 i)
{
  LCD_PrintInt8(
  (i == 1) ? 80 : 100);
}
	; prolog not shown
	; check value of i
	CMP R0,#+1
	; branch if not equal
	BNE ??Ternary_0
	MOVS R0,#+80
	B ??Ternary_1
	
	??Ternary_0:
	MOVS R0,#+100
	??Ternary_1:
	UXTB R0,R0
	BL LCD_PrintInt8
	; epilog not shown
		; prolog not shown
		; check value of i
		CMP R0, #1
		; branch if not equal
		BNE ??Ternary_0
		MOVS R0,#+80
		B ??Ternary_1
		
		??Ternary_0:
		MOVS R0,#+100
		
		??Ternary_1:
		BL LCD_PrintInt8
			; same as for size optimization

// Addition operation
int DoAdd(int x, int y)
{
  return x + y;
}
	; no prolog
	ADDS R0,R0,R1
	BX LR ; return value
		; same as for no optimization	; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化

`// Subtraction operation`
`int DoSub(int x, int y)`
`{`
`  return x - y;`
`}`
	`; no prolog`
	`SUBS R0,R0,R1`
	`BX LR ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Multiplication`
`int DoMul(int x, int y)`
`{`
`  return x * y;`
`}`
	`; no prolog`
	`MULS R0,R1,R0`
	`BX LR ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Division`
`int DoDiv(int x, int y)`
`{`
`  return x / y;`
`}`
	`; prolog not shown`
	`; Uses clib helper div function`
	`BL __aeabi_idiv`
	`POP {PC} ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Modulo operator`
`int DoMod(int x, int y)`
`{`
`  return x % y;`
`}`
	`; prolog not shown`
	`; Uses clib helper function`
	`BL __aeabi_idivmod`
	`MOVS R0,R1`
	`POP {PC} ; return value`
		`; same as for no optimization`	`; same as for size optimization`

`// Pointer`
`void Pointer(uint8 x, uint8 *ptr)`
`{`
`  *ptr = *ptr + x;`
`  ptr++;`
 	
`LCD_PrintInt8(*ptr);`
`}`
	`; *ptr = *ptr + x`
	`LDRB R0,[R1, #+0]`
	`ADDS R0,R0,R4`
	`STRB R0,[R1, #+0]`
	
	`; ptr++`
	`; LCD_PrintInt8(*ptr)`
	`ADDS R5,R1,#+1`
	`LDRB R0,[R5, #+0]`
	`BL LCD_PrintInt8`
		`; *ptr = *ptr + x`
		`LDRB R2, [R1, #+0] ; R1 = ptr`
		`ADDS R0, R2, R0 ; R0 = x`
		`STRB R0, [R1, #+0]`
		
		`; ptr++`
		`; LCD_PrintInt8(*ptr)`
		`LDRB R0, [R1, #+1]`
		`BL LCD_PrintInt8`
			`; same as for size optimization`

`// Function pointer`
`void FuncPtr(uint8 x,`
`  void *fptr(uint8))`
`{`
`  (*fptr)(x);`
`}`
	`; (*fptr)(x)`
	`PUSH {R3-R5,LR}`
	`MOVS R5,R0`
	`MOVS R4,R1`
	
	`MOVS R0,R5`
	`UXTB R0,R0`
	`; in a blx instruction, the`
	`; LS bit of the`
		`; (*fptr)(x)`
		`; in a blx instruction, the LS`
		`; bit of the register must be`
		`; 1 to keep the CPU in Thumb`
		`; mode, or an exception occurs`
		`BLX R1`
			`; same as for size optimization`

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化
	register ; must be 1 to keep the CPU ; in Thumb mode, or an ; exception occurs BLX R4		

// Packed structures
struct FOO_P
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
} __attribute__ ((packed));
 	
extern struct FOO_P myfoo_p;
 	
void PackedStruct(void)
{
myfoo_p.membera = 5;
myfoo_p.memberb = 10;
myfoo_p.memberc = 15;
myfoo_p.memberd = 20;
}
	; myfoo.membera = 5
	LDR R4,??DataTable2
	MOVS R0,#+5
	STRB R0,[R4, #+0]
	; myfoo_p.memberb = 10;
	MOVS R0,#+10
	STRB R0,[R4, #+1]
	; myfoo_p.memberc = 15;
	ADDS R1,R4,#+2
	MOVS R0,#+15
	BL __aeabi_uwrite4
	; myfoo_p.memberd = 20;
	MOVS R0,#+20
	STRB R0,[R4, #+6]
	LSRS R0,R0,#+8
	STRB R0,[R4, #+7]
	
	POP {R4,PC} ;
	return value
		; myfoo.membera = 5
		LDR R4,??DataTable2
		MOVS R0,#+5
		STRB R0,[R4, #+0]
		; myfoo_p.memberb = 10;
		MOVS R0,#+10
		STRB R0,[R4, #+1]
		; myfoo_p.memberc = 15;
		ADDS R1,R4,#+2
		MOVS R0,#+15
		BL __aeabi_uwrite4
		; myfoo_p.memberd = 20;
		MOVS R0,#+20
		STRB R0,[R4, #+6]
		MOVS R0,#+0
		STRB R0,[R4, #+7]
		
		POP {R4,PC} ;
		return value
			; same as for size optimization

// unpacked structures
struct FOO
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
};
 	
extern struct FOO myfoo;
 	
void PackedStruct(void)
{
myfoo.membera = 5;
	; myfoo.membera = 5
	LDR R0,??DataTable2_1
	MOVS R1,#+5
	STRB R1,[R0, #+0]
	; myfoo.memberb = 10;
	MOVS R1,#+10
	STRB R1,[R0, #+1]
	; myfoo.memberc = 15;
	MOVS R1,#+15
	STR R1,[R0, #+4]
	; myfoo.memberd = 20;
	MOVS R1,#+20
	STRH R1,[R0, #+8]
	
	BX LR
		; same as for no optimization	; same as for size optimization

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化

myfoo.memberb = 10;
myfoo.memberc = 15;
myfoo.memberd = 20;
}			

14.6.1 Cortex®-M0/M0+用 IAR、なし, サイズ, および速度最適化

Table 36 に、Cortex®-M0/M0+用の IAR コンパイラについて、さまざまな最適化オプションのコンパイラ出力の例を示します。例は、コンパイラによって生成された.lst ファイルから抽出されたものです。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

Table 36 Cortex®-M0/M0+ CPU 用 IAR コンパイラのコンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化

// Calling a function
// with no arguments
LCD_Start();
	; do the function call
	BL LCD_Start
		; same as for no optimization	; same as for no optimization

// Calling a function with
// one argument
LCD_PrintInt8(128);
	; R0 = first argument
	; conditional flags are NOT
	; updated by mov
	MOVS R0,#+128
	BL LCD_PrintInt8
		; same as for no optimization	; same as for size optimization

// Calling a function with
// two arguments
LCD_Position(0, 2);
	; R0 = first argument
	; R1 = second argument
	MOVS R1,#+2
	MOVS R0,#+0
	BL LCD_Position
		; same as for no optimization	; same as for size optimization

// For loop:
void ForLoop(uint8 i)
{
for(i = 0; i < 10; i++)
{
 	
LCD_PrintInt8(i);
}
}
	; function prolog
	; i is saved on the stack
	PUSH {R4,LR}
	; R4 = i
	MOVS R4,#+0
	; do the function call with
	; i as the argument in R0
	??ForLoop_0:
	MOVS R0,R4
	UXTB R0,R0
		; function prolog
		PUSH {R4,LR}
		
		B.N ?Subroutine0
		
		?Subroutine0:
		MOVS R4,#+0
		
		?Subroutine0_0:
		UXTB R0,R4
		BL LCD_PrintInt8
		ADDS R4,R4,#+1 ; i++
			; function prolog
			PUSH {R4,LR}
			MOVS R4,#+0
			
			UXTB R0,R4
			BL LCD_PrintInt8
			
			ADDS R4,R4,#+1 ; i++
			UXTB R0,R4
			CMP R0,#+10 ; i = 10?

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化

	; check i not equal to 10
	CMP R0,#+10
	BGE ??ForLoop_1
	
	MOVS R0,R4
	UXTB R0,R0
	BL LCD_PrintInt8
	
	ADDS R4,R4,#+1 ; i++
	B ??ForLoop_0
	
	; function epilog
	??ForLoop_1:
	POP {R4,PC};
	return
		UXTB R0,R4
		CMP R0,#+10 ; i = 10?
		
		BLT ??Subroutine0_0
		POP {R4,PC} ; return
			BLT ??ForLoop_0
			
			POP {R4,PC} ; return

// While loop
// i is type automatic, see
// Accessing Automatic Variables
// for details
uint8 i = 0;
 	
while (i < 10)
{
LCD_PrintInt8(i);
i++;
}
	; prolog not shown
	; i = 0
	MOVS R4,#+0
	
	??WhileLoop_0:
	MOVS R0,R4
	UXTB R0,R0
	; check i not equal to 10
	CMP R0,#+10
	BGE ??WhileLoop_1
	
	LCD_PrintInt8:
	MOVS R0,R4
	UXTB R0,R0
	; LCD_PrintInt8(i)
	BL LCD_PrintInt8
	
	ADDS R4,R4,#+1 ; i++
	B ??WhileLoop_0
	
	??WhileLoop_1:
	POP {R4,PC}
		; optimizes to match the for ; loop function
		REQUIRE ?Subroutine0
		;; Fall through to label ?Subroutine0
			; function prolog
			PUSH {R4,LR}
			MOVS R4,#+0
			
			??WhileLoop_0:
			UXTB R0,R4
			BL LCD_PrintInt8
			
			ADDS R4,R4,#+1 ; i++
			UXTB R0,R4
			CMP R0,#+10 ; i = 10?
			BLT ??WhileLoop_0
			
			POP {R4,PC} ; return

// Conditional statement
void Conditional(uint8 i, uint8 j)
{
if(j == 1)
{
	; prolog not shown
	; if(j == 1)
	Conditional:
	PUSH {R3-R5,LR}
	MOVS R5,R0
	MOVS R4,R1
	
	MOVS R0,R4
	UXTB R0,R0
		; prolog not shown
		; if(j == 1)
		CMP R1,#+1
		BEQ ??Conditional_0
		ADDS R0,R0,#+1
		UXTB R0,R0
		
		??Conditional_0:
			; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化

LCD_PrintInt8(i);
  }
  else
  {
  LCD_PrintInt8(i
+ 1);
  }
}
	CMP R0,#+1
	BNE ??Conditional_
	0
	
	; LCD_PrintInt8(i);
	MOVS R0,R4
	UXTB R0,R0
	BL LCD_PrintInt8
	B ??Conditional_
	1
	
	; LCD_PrintInt8(i + 1);
	??Conditional_0:
	ADDS R0,R4,#+1
	UXTB R0,R0
	BL LCD_PrintInt8
	
	??Conditional_1:
	POP {R0,R4,R5,PC}
		BL LCD_PrintInt8	

// Switch case
statements
void
SwitchCase(uint8 j)
{
  switch(j)
  {
  case 0:
 	
LCD_PrintInt8(1);
  break;
 	
  case 1:
 	
LCD_PrintInt8(2);
  break;
 	
  default:
 	
LCD_PrintInt8(0);
  break;
  }
}
	; prolog not shown
	; switch(j)
	UXTB R0,R0
	CMP R0,#+0
	BEQ ??SwitchCase_0
	CMP R0,#+1
	BEQ ??SwitchCase_1
	B ??SwitchCase_2
	
	; case 0
	??SwitchCase_0:
	MOVS R0,#+1
	BL LCD_PrintInt8
	B ??SwitchCase_3
	
	; case 1
	??SwitchCase_1:
	MOVS R0,#+2
	BL LCD_PrintInt8
	B ??SwitchCase_3
	
	; default
	??SwitchCase_2:
	MOVS R0,#+0
	BL LCD_PrintInt8
	; no epilog
		; prolog not shown
		; switch(j)
		CMP R0,#+0
		BEQ ??SwitchCase_0
		CMP R0,#+1
		BEQ ??SwitchCase_1
		B ??SwitchCase_2
		
		; case 0
		??SwitchCase_0:
		MOVS R0,#+1
		B ??SwitchCase_3
		
		; case 1
		??SwitchCase_1:
		MOVS R0,#+2
		B ??SwitchCase_3
		
		; case 2
		??SwitchCase_2:
		MOVS R0,#+0
		; default
		??SwitchCase_3:
		BL LCD_PrintInt8
		; epilog not shown
			; same as for size
			optimization

// Ternary operator
void Ternary(uint8
i)
{
  LCD_PrintInt8(
	; prolog not shown
	; check value of i
	CMP R0,#+1
	; branch if not equal
	BNE ??Ternary_0
		; prolog not shown
		; check value of i
		CMP R0, #1
		; branch if not equal
		BNE ??Ternary_0
			; same as for size
			optimization

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化

(i == 1) ? 80 : 100);
}
 	
	MOVS R0,#+80
	B ??Ternary_1
	
	??Ternary_0:
	MOVS R0,#+100
	??Ternary_1:
	UXTB R0,R0
	BL LCD_PrintInt8
	; epilog not shown
		MOVS R0,#+80
		B ??Ternary_1
		
		??Ternary_0:
		MOVS R0,#+100
		
		??Ternary_1:
		BL LCD_PrintInt8	

// Addition operation
int DoAdd(int x, int y)
{
return x + y;
}
	; no prolog
	ADDS R0,R0,R1
	BX LR ; return value
		; same as for no optimization	; same as for size optimization

// Subtraction operation
int DoSub(int x, int y)
{
return x - y;
}
	; no prolog
	SUBS R0,R0,R1
	BX LR ; return value
		; same as for no optimization	; same as for size optimization

// Multiplication
int DoMul(int x, int y)
{
return x * y;
}
	; no prolog
	MULS R0,R1,R0
	BX LR ; return value
		; same as for no optimization	; same as for size optimization

// Division
int DoDiv(int x, int y)
{
return x / y;
}
	; prolog not shown
	; Uses clib helper div function
	BL __aeabi_idiv
	POP {PC} ; return value
		; same as for no optimization	; same as for size optimization

// Modulo operator
int DoMod(int x, int y)
{
return x % y;
}
	; prolog not shown
	; Uses clib helper function
	BL __aeabi_idivmod
	MOVS R0,R1
	POP {PC} ; return value
		; same as for no optimization	; same as for size optimization

// Pointer
void Pointer(uint8 x, uint8 *ptr)
{
*ptr = *ptr + x;
	; *ptr = *ptr + x
	LDRB R0,[R1, #+0]
	ADDS R0,R0,R4
	STRB R0,[R1, #+0]
		; *ptr = *ptr + x
		LDRB R2, [R1, #+0] ; R1 = ptr
		ADDS R0, R2, R0 ; R0 = x
			; same as for size optimization

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化

ptr++;
 	
LCD_PrintInt8(*ptr);
}
	; ptr++
	; LCD_PrintInt8(*ptr)
	ADDS R5,R1,#+1
	LDRB R0,[R5, #+0]
	BL LCD_PrintInt8
		STRB R0, [R1, #+0]
		
		; ptr++
		; LCD_PrintInt8(*ptr)
		LDRB R0, [R1, #+1]
		BL LCD_PrintInt8	

// Function pointer
void FuncPtr(uint8 x,
void *fptr(uint8))
{
(*fptr)(x);
}
	; (*fptr)(x)
	PUSH {R3-R5,LR}
	MOVS R5,R0
	MOVS R4,R1
	
	MOVS R0,R5
	UXTB R0,R0
	; in a blx instruction, the
	; LS bit of the register
	; must be 1 to keep the CPU
	; in Thumb mode, or an
	; exception occurs
	BLX R4
		; (*fptr)(x)
		; in a blx instruction, the LS
		; bit of the register must be
		; 1 to keep the CPU in Thumb
		; mode, or an exception occurs
		BLX R1
			; same as for size optimization

// Packed structures
struct FOO_P
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
} __attribute__ ((packed));
 	
extern struct FOO_P myfoo_p;
 	
void PackedStruct(void)
{
myfoo_p.membera = 5;
myfoo_p.memberb = 10;
myfoo_p.memberc = 15;
myfoo_p.memberd = 20;
}
	; myfoo.membera = 5
	LDR R4,??DataTable2
	MOVS R0,#+5
	STRB R0,[R4, #+0]
	; myfoo_p.memberb = 10;
	MOVS R0,#+10
	STRB R0,[R4, #+1]
	; myfoo_p.memberc = 15;
	ADDS R1,R4,#+2
	MOVS R0,#+15
	BL __aeabi_uwrite4
	; myfoo_p.memberd = 20;
	MOVS R0,#+20
	STRB R0,[R4, #+6]
	LSRS R0,R0,#+8
	STRB R0,[R4, #+7]
	
	POP {R4,PC} ; return value
		; myfoo.membera = 5
		LDR R4,??DataTable2
		MOVS R0,#+5
		STRB R0,[R4, #+0]
		; myfoo_p.memberb = 10;
		MOVS R0,#+10
		STRB R0,[R4, #+1]
		; myfoo_p.memberc = 15;
		ADDS R1,R4,#+2
		MOVS R0,#+15
		BL __aeabi_uwrite4
		; myfoo_p.memberd = 20;
		MOVS R0,#+20
		STRB R0,[R4, #+6]
		MOVS R0,#+0
		STRB R0,[R4, #+7]
		
		POP {R4,PC} ; return value
			; same as for size optimization

C コード	IAR, Cortex®-M0/M0+, 最適化なし	IAR, Cortex®-M0/M0+, サイズ最適化	IAR, Cortex®-M0/M0+, 速度最適化

`// unpacked structures`
`struct FOO`
`{`
`  uint8  membera;`
`  uint8  memberb;`
`  uint32 memberc;`
`  uint16 memberd;`
`};`
 	
`extern struct FOO myfoo;`
 	
`void PackedStruct(void)`
`{`
`  myfoo.membera = 5;`
`  myfoo.memberb = 10;`
`  myfoo.memberc = 15;`
`  myfoo.memberd = 20;`
`}`
	`; myfoo.membera = 5`
	`LDR R0,??DataTable2_1`
	`MOVS R1,#+5`
	`STRB R1,[R0, #+0]`
	`; myfoo.memberb = 10;`
	`MOVS R1,#+10`
	`STRB R1,[R0, #+1]`
	`; myfoo.memberc = 15;`
	`MOVS R1,#+15`
	`STR R1,[R0, #+4]`
	`; myfoo.memberd = 20;`
	`MOVS R1,#+20`
	`STRH R1,[R0, #+8]`
	
	`BX LR`
		`; same as for no optimization`	`; same as for size optimization`

14.6.2 Cortex®-M0/M0+用 IAR、低, 中, およびバランス最適化

Table 37 に、Cortex®-M0/M0+用の IAR コンパイラについて、さまざまな最適化オプションのコンパイラ出力の例を示します。これらの例は、コンパイラによって生成された.lst ファイルから抽出されたものです。

コンパイラ関数でのレジスタの使用法とスタックの使用法の詳細については、関数の引数と結果を参照してください。

Table 37 Cortex®-M3 CPU 用 IAR コンパイラのコンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 低最適化	IAR, Cortex®-M0/M0+, 中最適化	IAR, Cortex®-M0/M0+, バランス最適化

`// Calling a function`
`// with no arguments`
`LCD_Start();`
	`; do the function call`
	`BL LCD_Start`
		`; same as for low`
		`; optimization`
			`; same as for low`
			`; optimization`

`// Calling a function with`
`// one argument`
`LCD_PrintInt8(128);`
	`; R0 = first argument`
	`; conditional flags are NOT`
	`; updated by mov`
	`MOVS R0,#+128`
	`BL LCD_PrintInt8`
		`; same as for low`
		`; optimization`
			`; same as for low`
			`; optimization`

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 低最適化	IAR, Cortex®-M0/M0+, 中最適化	IAR, Cortex®-M0/M0+, バランス最適化

`// Calling a function with`
`// two arguments`
`LCD_Position(0, 2);`
	`; R0 = first argument`
	`; R1 = second argument`
	`MOVS R1,#+2`
	`MOVS R0,#+0`
	`BL LCD_Position`
		`; same as for low`
		`; optimization`
			`; same as for low`
			`; optimization`

`// For loop:`
`void ForLoop(uint8 i)`
`{`
`for(i = 0; i < 10; i++)`
`{`
 	
`LCD_PrintInt8(i);`
`}`
`}`
	`; function prolog`
	`; i is saved on the stack`
	`PUSH {R4,LR}`
	`; R4 = i`
	`MOVS R4,#+0`
	`B ??ForLoop_0`
	
	`??ForLoop_1:`
	`; do the function call with`
	`; i as the argument in R0`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
	
	`ADDS R4,R4,#+1 ; i++`
	`B ??ForLoop_0`
	
	`??ForLoop_0:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`; check i not equal to 10`
	`CMP R0,#+10`
	`BLT ??ForLoop_1`
	
	`; function epilog`
	`POP {R4,PC};`
	`return`
		`; function prolog`
		`MOVS R4,#+0`
		`B ??ForLoop_0`
		
		`LCD_PrintInt8(i);`
		`??ForLoop_1: (+1)`
		`MOVS R0,R4`
		`UXTB R0,R0`
		`BL LCD_PrintInt8`
		
		`ADDS R4,R4,#+1`
		`??ForLoop_0:`
		`MOVS R0,R4`
		`UXTB R0,R0`
		`CMP R0,#+10`
		`BLT ??ForLoop_1`
		
		`; function epilog`
		`POP {R4,PC};`
		`return`
			`; function prolog`
			`PUSH {R4,LR}`
			
			`B.N ?Subroutine0`
			
			`?Subroutine0:`
			`MOVS R4,#+0`
			
			`?Subroutine0_0:`
			`UXTB R0,R4`
			`BL LCD_PrintInt8`
			`ADDS R4,R4,#+1 ; i++`
			`UXTB R0,R4`
			`CMP R0,#+10 ; i = 10?`
			
			`BLT ??Subroutine0_0`
			`POP {R4,PC} ; return`

`// While loop`
`// i is type automatic, see`
`// Accessing Automatic Variables`
`// for details`
`uint8 i = 0;`
 	
`while (i < 10)`
`{`
 	
`LCD_PrintInt8(i);`
	`; prolog not shown`
	`; i = 0`
	`??WhileLoop_1:`
	`MOVS R0,R4`
	`UXTB R0,R0`
	`BL LCD_PrintInt8`
	
	`ADDS R4,R4,#+1`
	
	`??WhileLoop_0:`
	`MOVS R0,R4`
	`UXTB R0,R0`
		`; prolog`
		`WhileLoop:`
		`PUSH {R4,LR}`
		`MOVS R4,#+0`
		`B ??WhileLoop_0`
		
		`LCD_PrintInt8(i);`
		`??WhileLoop_1:`
		`MOVS R0,R4`
		`UXTB R0,R0`
		`BL LCD_PrintInt8`
		`ADDS R4,R4,#+1`
			`; optimizes to match the for ; loop function`
			`REQUIRE ?Subroutine0`
			`;; Fall through to label ?Subroutine0`

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 低最適化	IAR, Cortex®-M0/M0+, 中最適化	IAR, Cortex®-M0/M0+, バランス最適化

i++;
}
	; check i not equal to 10
	CMP R0,#+10
	BLT ??WhileLoop_1
	
	; function epilog
	POP {R4,PC};
	return
		??WhileLoop_0:
		MOVS R0,R4
		UXTB R0,R0
		CMP R0,#+10
		BLT ??WhileLoop_1
		
		; function epilog
		POP {R4,PC};
		return	

// Conditional statement
void Conditional(uint8 i, uint8 j)
{
if(j == 1)
{
 	
LCD_PrintInt8(i);
}
else
{
 	
LCD_PrintInt8(i + 1);
}
}
	PUSH {R7,LR}
	; if(j == 1)
	UXTB R1,R1
	CMP R1,#+1
	BNE ??Conditional_0
	; LCD_PrintInt8(i);
	UXTB R0,R0
	BL LCD_PrintInt8
	B ??Conditional_1
	; LCD_PrintInt8(i + 1);
	Conditional_0:
	ADDS R0,R0,#+1
	UXTB R0,R0
	BL LCD_PrintInt8
	
	??Conditional_1:
	POP {R0,PC};
	return
		Conditional:
		PUSH {R7,LR}
		CMP R1,#+1
		BNE ??Conditional_0
		
		LCD_PrintInt8(i);
		BL LCD_PrintInt8
		POP {R0,PC}
		
		LCD_PrintInt8(i + 1);
		??Conditional_0:
		ADDS R0,R0,#+1
		UXTB R0,R0
		BL LCD_PrintInt8
		
		POP {R0,PC} ;
		return
			; prolog not shown
			; if(j == 1)
			CMP R1,#+1
			BEQ ??Conditional_0
			ADDS R0,R0,#+1
			UXTB R0,R0
			
			??Conditional_0:
			BL LCD_PrintInt8

// Switch case statements
void SwitchCase(uint8 j)
{
switch(j)
{
case 0:
 	
LCD_PrintInt8(1);
break;
 	
case 1:
 	
LCD_PrintInt8(2);
break;
	; prolog not shown
	; switch(j)
	UXTB R0,R0
	CMP R0,#+0
	BEQ ??SwitchCase_0
	CMP R0,#+1
	BEQ ??SwitchCase_1
	B ??SwitchCase_2
	; case 0
	??SwitchCase_0:
	MOVS R0,#+1
	BL LCD_PrintInt8
	
	B ??SwitchCase_3
		SwitchCase:
		PUSH {R7,LR}
		CMP R0,#+0
		BEQ ??SwitchCase_0
		CMP R0,#+1
		BEQ ??SwitchCase_1
		B ??SwitchCase_2
		
		case 0:
		LCD_PrintInt8(1);
		??SwitchCase_0:
		MOVS R0,#+1
		BL LCD_PrintInt8
		POP {R0,PC}
			; prolog not shown
			; switch(j)
			CMP R0,#+0
			BEQ ??SwitchCase_0
			CMP R0,#+1
			BEQ ??SwitchCase_1
			B ??SwitchCase_2
			
			; case 0
			??SwitchCase_0:
			MOVS R0,#+1
			B ??SwitchCase_3
			
			; case 1

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 低最適化	IAR, Cortex®-M0/M0+, 中最適化	IAR, Cortex®-M0/M0+, バランス最適化

default:
 	
LCD_PrintInt8(0);
break;
}
}
	; case 1
	??SwitchCase_1:
	MOVS R0,#+2
	BL LCD_PrintInt8
	B ??SwitchCase_
	3
	
	; default
	??SwitchCase_2:
	MOVS R0,#+0
	BL LCD_PrintInt8
	; epilog not shown
		case 1:
		LCD_PrintInt8(2);
		??SwitchCase_1:
		MOVS R0,#+2
		BL LCD_PrintInt8
		POP {R0,PC}
		
		default:
		LCD_PrintInt8(0);
		??SwitchCase_2:
		MOVS R0,#+0
		BL LCD_PrintInt8
		
		POP {R0,PC};
		return
			??SwitchCase_1:
			MOVS R0,#+2
			B ??SwitchCase_
			3
			
			; case 2
			??SwitchCase_2:
			MOVS R0,#+0
			; default
			??SwitchCase_3:
			BL LCD_PrintInt8
			; epilog not shown

// Ternary operator
void Ternary(uint8 i)
{
LCD_PrintInt8(
(i == 1) ? 80 : 100);
}
	; prolog not shown
	; check value of i
	CMP R0,#+1
	; branch if not equal
	BNE ??Ternary_0
	MOVS R0,#+80
	B ??Ternary_1
	
	??Ternary_0:
	MOVS R0,#+100
	??Ternary_1:
	UXTB R0,R0
	BL LCD_PrintInt8
	; no epilog
		; same as for low
		; optimization
			; prolog not shown
			; check value of i
			CMP R0, #1
			; branch if not equal
			BNE ??Ternary_0
			MOVS R0,#+80
			B ??Ternary_1
			
			??Ternary_0:
			MOVS R0,#+100
			
			??Ternary_1:
			BL LCD_PrintInt8

// Addition operation
int DoAdd(int x, int y)
{
return x + y;
}
	; no prolog
	ADDS R0,R1,R0
	BX LR ; return value
		; same as for low
		; optimization
			; same as for low
			; optimization

// Subtraction operation
int DoSub(int x, int y)
{
return x - y;
}
	; no prolog
	SUBS R0,R0,R1
	BX LR ; return value
		; same as for low
		; optimization
			; same as for low
			; optimization

// Multiplication
int DoMul(int x, int y)
{
	; no prolog
	MULS R0,R0,R1
	BX LR ;
	return value
		; same as for low
		; optimization
			; same as for low
			; optimization

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 低最適化	IAR, Cortex®-M0/M0+, 中最適化	IAR, Cortex®-M0/M0+, バランス最適化
return x * y; }			

// Division
int DoDiv(int x,
int y)
{
return x / y;
}
	; no prolog
	; call into c helper
	; function
	BL __aeabi_idiv
		; same as for low
		; optimization
			; same as for low
			; optimization

// Modulo operator
int DoMod(int x,
int y)
{
return x % y;
}
	; no prolog
	; call into c helper
	; function
	BL
	__aeabi_idivmod
	MOVS R0,R1
		; same as for low
		; optimization
			; same as for low
			; optimization

// Pointer
void Pointer(uint8
x, uint8 *ptr)
{
*ptr = *ptr + x;
ptr++;
 	
LCD_PrintInt8(*ptr
);
}
	; *ptr = *ptr + x
	LDRB R2,[R1, #+0]
	ADDS R0,R2,R0
	STRB R0,[R1, #+0]
	
	; ptr++
	; LCD_PrintInt8(*ptr)
	ADDS R0,R1,#+1
	LDRB R0,[R0, #+0]
	BL LCD_PrintInt8
		; *ptr = *ptr + x
		LDRB R2,[R1, #+0]
		ADDS R0,R2,R0
		STRB R0,[R1, #+0]
		
		; ptr++
		; LCD_PrintInt8(*ptr)
		LDRB R0,[R1, #+1]
		BL LCD_PrintInt8
			; same as for medium
			optimization

// Function
pointer
void FuncPtr(uint8
x,
void
*fptr(uint8))
{
(*fptr)(x);
}
	; (*fptr)(x)
	UXTB R0,R0
	; in a blx
	instruction, the
	; LS bit of the
	register
	; must be 1 to keep
	the CPU
	; in Thumb mode, or an
	; exception occurs
	BLX R1
		; (*fptr)(x)
		; in a blx
		instruction, the LS
		; bit of the register
		must be
		; 1 to keep the CPU in
		Thumb
		; mode, or an
		exception occurs
		BLX R1
			; same as for medium
			optimization

// Packed
structures
struct FOO_P
{
uint8 membera;
uint8 memberb;
uint32 memberc;
uint16 memberd;
} __attribute__
((packed));
 	
extern struct
FOO_P myfoo_p;
	; myfoo.membera = 5
	LDR
	R4,??DataTable2
	MOVS R0,#+5
	STRB R0,[R4, #+0]
	; myfoo_p.memberb =
	10;
	MOVS R0,#+10
	STRB R0,[R4, #+1]
	; myfoo_p.memberc =
	15;
	ADDS R1,R4,#+2
	MOVS R0,#+15
		; same as for low
		; optimization
			; myfoo.membera = 5
			LDR
			R4,??DataTable2
			MOVS R0,#+5
			STRB R0,[R4, #+0]
			; myfoo_p.memberb =
			10;
			MOVS R0,#+10
			STRB R0,[R4, #+1]
			; myfoo_p.memberc =
			15;
			ADDS R1,R4,#+2
			MOVS R0,#+15

14 付録: コンパイラ出力の詳細

C コード	IAR, Cortex®-M0/M0+, 低最適化	IAR, Cortex®-M0/M0+, 中最適化	IAR, Cortex®-M0/M0+, バランス最適化

void
PackedStruct(void)
{
  myfoo_p.membera
= 5;
  myfoo_p.memberb
= 10;
  myfoo_p.memberc
= 15;
  myfoo_p.memberd
= 20;
}
	BL
	__aeabi_uwrite4;
	myfoo_p.memberd = 20;
	MOVS R0,#+20
	STRB R0,[R4, #+6]
	LSRS R0,R0,#+8
	STRB R0,[R4, #+7]
	
	POP
	{R4,PC} ;return
			BL
			__aeabi_uwrite4;
			myfoo_p.memberd = 20;
			MOVS R0,#+20
			STRB R0,[R4, #+6]
			MOVS R0,#+0
			STRB R0,[R4, #+7]
			
			POP
			{R4,PC} ;return

// unpacked
structures
struct FOO
{
  uint8 membera;
  uint8 memberb;
  uint32 memberc;
  uint16 memberd;
};
 	
extern struct FOO
myfoo;
 	
void
PackedStruct(void)
{
  myfoo.membera =
5;
  myfoo.memberb =
10;
  myfoo.memberc =
15;
  myfoo.memberd =
20;
}
	; myfoo.membera = 5
	LDR
	R0,??DataTable2_1
	MOVS R1,#+5
	STRB R1,[R0, #+0];
	myfoo.memberb = 10;
	MOVS R1,#+10
	STRB R1,[R0, #+1];
	myfoo.memberc = 15;
	MOVS R1,#+15
	STR R1,[R0, #+4]
	; myfoo.memberd = 20;
	MOVS R1,#+20
	STRH R1,[R0, #+8]
	
	BX LR ;return
		; same as for low
		; optimization
			; same as for low
			optimization

14.7 コンパイラテストプログラム

前の表のコンパイラ出力を生成するために、次の C コードが使用されました。このコードは、PSoC™ 4 および PSoC™ 5LP, GCC, MDK, IAR 用に、最適化なしおよびサイズと速度の最適化ありでコンパイルされます。このコードは PSoC™ Creator プロジェクトに追加できますが、プロジェクトで次の操作も実行する必要があります。

- プロジェクトの回路図にキャラクターLCD コンポーネントを追加し、名前を「LCD」に変更します。
- PSoC™ 4 の場合、これらの低メモリパーツのヒープサイズとスタックサイズの設定を減らします。これは、デザイン全体のリソース (DWR) ウィンドウのシステムタブで行います。ヒープサイズとスタックサイズの値はそれぞれ 0x100 と 0x400 で、通常は適切です。

このコードは、*main.c* と *test.c* の 2 つのファイルにあります。これは *main.c* です。

```
#include <project.h>
extern void ForLoop(uint8);
extern void WhileLoop(void);
extern void Conditional(uint8, uint8);
extern void SwitchCase(uint8);
extern void Ternary(uint8);
extern int  DoAdd(int, int);
extern int  DoSub(int, int);
extern int  DoMul(int, int);
extern int  DoDiv(int, int);
extern int  DoMod(int, int);
extern void Pointer(uint8, uint8 *);
extern void FuncPtr(uint8, void (*)(uint8));
extern void PackedStruct(void);
extern void UnpackedStruct(void);

struct FOO /* structures are unpacked by default */
{
       uint8  membera;
     uint8  memberb;
     uint32 memberc;
     uint16 memberd;
};

struct FOO_P /* packed structure */
{
     uint8  membera;
```

```
        uint8  memberb;
        uint32 memberc;
        uint16 memberd;
    } __attribute__ ((packed));

    uint8 myData = 6;
    struct FOO_P myfoo_p;
    struct FOO   myfoo;


    int main()
    {
        /* Place your initialization/startup code here (e.g. MyInst_Start()) */
        LCD_Start();

        /* CyGlobalIntEnable; */ /* Uncomment this line to enable global
    interrupts. */
        for(;;)
        {
            LCD_PrintInt8(128);
            LCD_Position(0, 2);
            ForLoop(9);
            WhileLoop();
            Conditional(3, 4);
            SwitchCase(4);
            Ternary(5);
            LCD_PrintNumber((uint16)DoAdd(5, 4));
            LCD_PrintNumber((uint16)DoSub(5, 4));
            LCD_PrintNumber((uint16)DoMul(5, 4));
            LCD_PrintNumber((uint16)DoDiv(5, 4));
            LCD_PrintNumber((uint16)DoMod(5, 4));
            Pointer(4, &myData);
            FuncPtr(3, &LCD_PrintInt8);
            PackedStruct();
            UnpackedStruct();
        } /* end of for(;;) */
    } /* end of main() */
```

14 付録: コンパイラ出力の詳細

そして、これは test.c です。

```
#include <project.h>

struct FOO /* structures are unpacked by default */
{
    uint8  membera;
    uint8  memberb;
    uint32 memberc;
    uint16 memberd;
};

struct FOO_P /* packed structure */
{
    uint8  membera;
    uint8  memberb;
    uint32 memberc;
    uint16 memberd;
} __attribute__ ((packed));

extern struct FOO_P myfoo_p;
extern struct FOO   myfoo;

void ForLoop(uint8 i)
{
    for(i = 0; i < 10; i++)
    {
        LCD_PrintInt8(i);
    }
}

void WhileLoop(void)
{
    uint8 i = 0;
    while(i < 10)
    {
        LCD_PrintInt8(i);
        i++;
    }
}
```

14 付録: コンパイラ出力の詳細

```
void Conditional(uint8 i, uint8 j)
{
    if(j == 1)
    {
        LCD_PrintInt8(i);
    }
    else
    {
        LCD_PrintInt8(i + 1);
    }
}

void SwitchCase(uint8 j)
{
    switch(j)
    {
        case 0:
        LCD_PrintInt8(1);
        break;

        case 1:
        LCD_PrintInt8(2);
        break;

        default:
        LCD_PrintInt8(0);
        break;
    }
}
```

14 付録: コンパイラ出力の詳細

```
void Ternary(uint8 i)
{
    LCD_PrintInt8((i == 1) ? 80 : 100);
}

int DoAdd(int x, int y)
{
    return x + y;
}

int DoSub(int x, int y)
{
    return x - y;
}

int DoMul(int x, int y)
{
    return x * y;
}
int DoDiv(int x, int y)
{
    return x / y;
}

int DoMod(int x, int y)
{
    return x % y;
}

void Pointer(uint8 x, uint8 *ptr)
{
    *ptr = *ptr + x;
    ptr++;
    LCD_PrintInt8(*ptr);
}
```

14 付録: コンパイラ出力の詳細

```
void FuncPtr(uint8 x, void *fptr(uint8))
{
    (*fptr)(x);
}

void PackedStruct(void)
{
    myfoo_p.membera =  5;
    myfoo_p.memberb = 10;
    myfoo_p.memberc = 15;
    myfoo_p.memberd = 20;
}

void UnpackedStruct(void)
{
    myfoo.membera =  5;
    myfoo.memberb = 10;
    myfoo.memberc = 15;
    myfoo.memberd = 20;
}
```

参考資料

[1] アプリケーションノート

- AN77759 – PSoC™ 5LP 入門
- AN79953 – Getting Started with PSoC™ 4 MCU
- AN52705 – Getting Started with PSoC™ DMA
- AN54460 – PSoC™ 3 および PSoC™ 5LP の割込み
- AN90799 – PSoC™ 4 Interrupts
- AN60630 – PSoC™ 3 8051 Code and memory optimization

[2] C ドキュメント

- GCC ドキュメントは、PSoC™ Creator のインストールフォルダにあります。
  コンパイラ ドキュメント:
  C:\Program Files\Cypress\PSoC Creator\4.3\PSoC Creator\import\gnu_cs\arm\5.4.1\share\doc\gcc-arm-none-eabi\pdf\gcc\gcc.pdf
  リンカ スクリプト ファイル ドキュメント:
  C:\Program Files\Cypress\PSoC Creator\4.3\PSoC Creator\import\gnu_cs\arm\5.4.1\share\doc\gcc-arm-none-eabi\pdf\ld.pdf
- MDK の場合、ドキュメントは MDK インストール フォルダ (通常は C:\Keil\ARM\Hlp) にあります。armtools.chm から始めてください。コンパイラについては、armcc.chm および armccref.chm を参照してください。
  リンカ スクリプト ファイルのドキュメントは、armlink.chm および armlinkref.chm にあります。
- IAR の場合、ドキュメントは IAR インストール フォルダ (通常は C:\Program Files (x86)\IAR Systems\Embedded Workbench 8.4\arm\doc) にあります。コンパイラおよびリンカ スクリプトの情報は、EWARM_DevelopmentGuide.ENU.pdf にあります。

[3] Arm® Cortex®のドキュメント

Arm®は、Cortex®-M3 および Cortex®-M0/M0+ CPU に関する豊富な情報を自社の Web サイトで提供しています。

- Cortex®-M0 Instruction Set
- Cortex®-M0+ Instruction Set
- Cortex®-M3 Instruction Set
- Cortex® Microcontroller Software Interface Standard (CMSIS) library
- Arm® related books

改訂履歴

版数	発行日	変更内容
**	2020-10-01	本版は英語版 001-89610 Rev. *E について、CYPRESS DEVELOPER COMMUNITY の参画者によって日本語に翻訳されたドキュメントです。
*A	2024-08-05	これは英語版 001-289610 Rev. *H を翻訳した日本語版 002-31297 Rev. *A です。

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2024-08-05
Published by
Infineon Technologies AG
81726 Munich, Germany

© 2024 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference
002-31297 Rev. *A

重要事項
本手引書に記載された本製品の使用に関する手引きとして提供されるものであり、いかなる場合も、本製品における特定の機能性能や品質について保証するものではありません。本製品の使用の前に、当該手引書の受領者は実際の使用環境の下であらゆる本製品の機能及びその他本手引書に記された一切の技術的情報について確認する義務が有ります。インフィニオンテクノロジーズはここに当該手引書内で記される情報につき、第三者の知的所有権の不侵害の保証を含むがこれに限らず、あらゆる種類の一切の保証および責任を否定いたします。

本文書に含まれるデータは、技術的訓練を受けた従業員のみを対象としています。本製品の対象用途への適合性、およびこれら用途に関連して本文書に記載された製品情報の完全性についての評価は、お客様の技術部門の責任にて実施してください。

警告事項
技術的要件に伴い、製品には危険物質が含まれる可能性があります。当該種別の詳細については、インフィニオンの最寄りの営業所までお問い合わせください。
インフィニオンの正式代表者が署名した書面を通じ、インフィニオンによる明示の承認が存在する場合を除き、インフィニオンの製品は、当該製品の障害またはその使用に関する一切の結果が、合理的に人的傷害を招く恐れのある一切の用途に使用することはできないこと予めご了承ください。