サイプレスはインフィニオン テクノロジーズになりました

この表紙に続く文書には「サイプレス」と表記されていますが、これは同社が最初にこの製品を開発したからです。新規および既存のお客様いずれに対しても、引き続きインフィニオンがラインアップの一部として当該製品をご提供いたします。

文書の内容の継続性

下記製品がインフィニオンの製品ラインアップの一部として提供されたとしても、それを理由としてこの文書に変更が加わることはありません。今後も適宜改訂は行いますが、変更があった場合は文書の履歴ページでお知らせします。

注文時の部品番号の継続性

インフィニオンは既存の部品番号を引き続きサポートします。ご注文の際は、データシート記載の注文部品番号をこれまで通りご利用下さい。

AN60486

PSoC® 1 M8C ImageCraft C コード最適化

Archana Yarlagadda
関連プロジェクト: なし
関連製品ファミリ: CY8C2xxxx (すべての PSoC 1 デバイス)
ソフトウェア バージョン: PSoC Designer™ 5.4
関連アプリケーション ノート: AN75320、AN60630、AN2017

本アプリケーション ノートについて、ご質問がある場合または支援が必要な場合は、keri@cypress.com までお問い合わせください。

AN60486 は、PSoC® 1 C コードを最適化して高速化および小型化する方法を説明し、そのコード最適化に役立つ PSoC Designer プロジェクトおよび ImageCraft コンパイラの設定を対象としています。 また、効率的なコーディングのためのガイドラインも含みます。本アプリケーション ノートでは、ユーザーが PSoC 1、PSoC Designer 統合設計環境 (IDE)、C 言語によるプログラミングに精通していることを前提とします。PSoC 1 の入門については、「AN75320 - PSoC 1 入門」を参照してください。

目次

はじめに 1
プロジェクト レベルの最適化 2
設定 1: 再配置可能なコード開始アドレス 2
設定 2: コンフィギュレーションの初期化 3
設定 3: サブリメーション (Sublimation) とコンデンセーション (Condensation) 4
設定 4: ROM 定数対 RAM 定数 6
ImageCraft Pro コンパイラ オプション 6
ヒントおよびガイドライン 6
ガイドライン 1: 割り込みサービス ルーチンにおける関数呼び出しを回避 6
ガイドライン 2: 算術関数の使用を制限 8
ガイドライン 3: 配列インデックス対ポインタ 10
ガイドライン 4: スイッチ文と if-else 文の使用 11
ガイドライン 5: アセンブラでコードの一部を書く 13
ガイドライン 6: PSoC 1 のビットを操作 13
ガイドライン 7: C コードのフラッシュ使用率および実行時間の計算 14
結論 15

はじめに

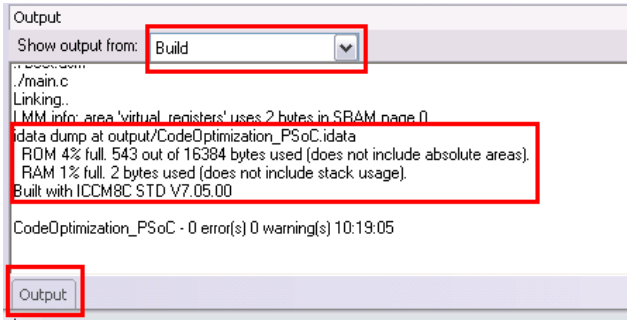
PSoC Designer プロジェクトに使用されるフラッシュ (ROM) 容量を削減する方法はいくつかあります。これにより、より小さい容量の PSoC デバイスを使用することが可能になり、コスト削減に繋がることがあります。

ImageCraft コンパイラの Standard 版は PSoC Designer と共に含まれています。より多く最適化を提供する Pro 版を ImageCraft のウェブサイトから購入することもできます。本アプリケーションノートに付属するコードスニペットは、両方のコンパイラに使用できます。 表示されている例は、Standard 版でコンパイルされました。Pro 版を使用して得られる追加の最適化についても、本文書で説明されています。

本書で説明する技法を使用するには、PSoC Designer プロジェクトの開発およびビルド方法を理解し、C 言語プログラミングの基礎知識を身に付ける必要があります。 PSoC 1 入門は AN75320 を参照して下さい。

PSoC Designer プロジェクトを開発中に、どの程度の ROM スペースを使用しているかを確認するには、図 1 に示されているように **Output** ステータスウィンドウの **Build** タブをご覧ください。この例で使用されているコード容量は 543 バイトです。

図 1. ROM および RAM 使用量を表示する PSoC Designer ビルド メッセージ



次のセクションでは、プロジェクト レベルの最適化設定、その後いくつかのコーディング ガイドラインについて説明します。

プロジェクト レベルの最適化

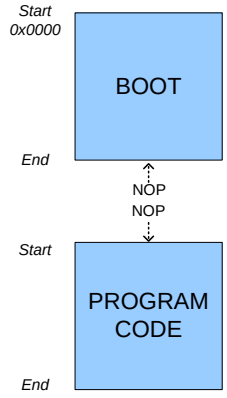
Project>Settings から、複数の PSoC Designer コード最適化設定を利用可能です。

設定 1: 再配置可能なコード開始アドレス

PSoC Designer プロジェクトがビルドされる際に、ImageCraft コンパイラは C ファイルをアセンブリ ファイルに変換します。そしてアセンブラがそれらアセンブリ ファイルを再配置可能なオブジェクト ファイルに変換します。 最後に、リンカーが再配置可能なオブジェクト ファイルを結合し、実行可能な*.hex* ファイルを作成します。

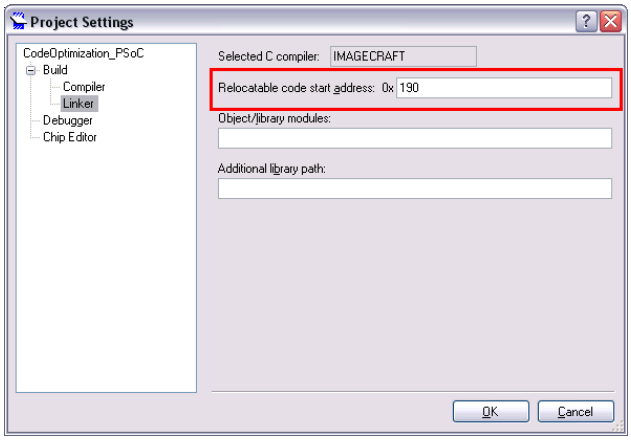
ユーザー コードに加えて、*.hex* ファイルには自動生成された初期化コード (ブートコード) が含まれています。 続いて、ブート コードの拡張を可能にするための一部のフラッシュ バイトを予約するために機能する、一連の NOP 命令があります。図 2 を参照してください。 ブート コードの直後にコードを強制的に配置することにより、フラッシュ スペースを節約できます。

図 2. PSoC Designer プロジェクト用のフラッシュ メモリ マップ



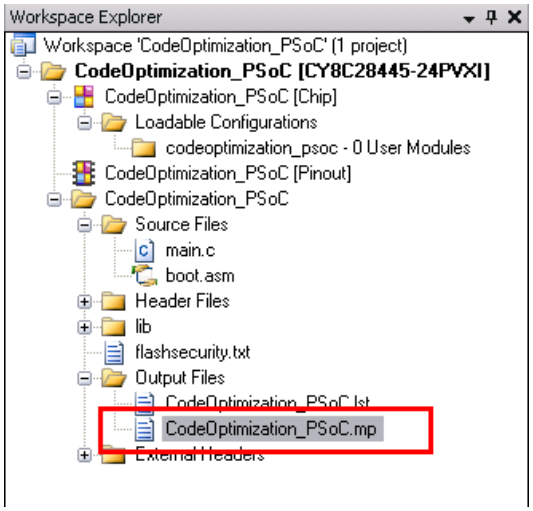
Project>Settings を選択して PSoC Designer により、ユーザー コードの開始アドレスを指定できます。図 3 に示すようにポップアップするウィンドウで、「Linker」を選択します。

図 3. 再配置可能なコード開始アドレスの選択



テキスト ボックス内に入力するアドレスは、図 4 で示したように、マップ ファイル (*.mp*) で見つけられるブート コードのサイズに基づいています。

図 4. PSoC Designer におけるマップ ファイル



マップ ファイルは、図 5 に示されているように、様々なコード エリアの開始アドレスおよび終了アドレスを表示します。ブート コードは「TOP」エリアにあり、ご使用のコードは「lit」エリアにあります。図 3 の**再配置可能なコード開始アドレス**を「TOP」エリアの終了アドレスに設定してフラッシュ スペースを最も効率的に使用できます。 たとえば、デフォルトの設定は 0x190 (「lit」エリアの冒頭のアドレス) です。 これを 0x151 (「TOP」エリアの最終のアドレス) に変更すると、ROM にさらに 63 (0x190 – 0x151 = 0x3F) バイトを取得できます。

図 5. *.mp* ファイルにおける「lit」および「TOP」エリア

```
Project:      output/CodeOptimization_PSoC
Build Number: 0
Date:         Mon Mar 19 10:19:03 2012
Compiler:     ICCM8C
Version:      7.05.00

Area                              Addr   Size   Decimal Bytes (Attributes)
--------------------------------  ----   ----   ------- ----- ------------
                                  lit    0190   00F2 =    242. bytes (rel,con,rom,lit)

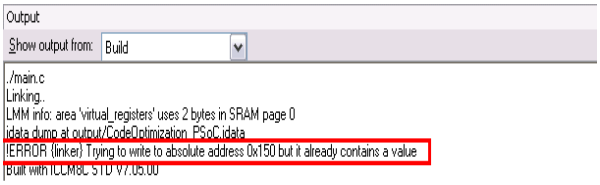
       Addr  Global Symbol
      -----  --------------------------------
       0190  LoadConfigTBL_codeoptimization_psoc_Bank0
       01F3  LoadConfigTBL_codeoptimization_psoc_Bank1
       0282  __lit_end

Area                              Start  End    Decimal Bytes (Attributes)
--------------------------------  ----   ----   ------- ----- ------------
                                  TOP    0000   0151 =    337. bytes (abs,ovr,rom,code)

       Addr  Global Symbol
      -----  --------------------------------
       0080  __Start
       014E  _bGetPowerSetting
       014E  bGetPowerSetting
```

再配置可能なコード開始アドレスが「TOP」の最終のアドレスよりも低い値に設定されると、エラー メッセージが表示されます。たとえば、値を 0x150 に設定し、ブート コードが 0x151 で終わる場合、図 6 に示すようにエラー メッセージが表示されます。

図 6. 再配置可能なコード アドレスがブート コードのサイズよりも低い場合のエラー



設定 2: コンフィギュレーションの初期化

コード最適化のもう 1 つのオプションは、起動時の PSoC レジスタの初期化です。図 7 に示すように、レジスタの値は **Loop (ループ)** または **Direct write (直接書き込み)** という 2 方式を使用して設定できます。 これらオプションを使用するには、**Project＞Settings** を選択し、表示するウィンドウで「Chip Editor」をクリックしてください。

Loop 方式を使用すると、レジスタ アドレスおよび値の表が作成され、専用関数により表内の値が対応アドレスに書き込まれます。**Direct write** 方式を使用すると、レジスタ当たり 1 つの MOV 命令という形式でレジスタへ直接書き込みます。

選択された方法は、自動生成されたファイル *PSOCConfigTBL.asm* で符号化されます。これら 2 方式のコードは、図 8 に示されます。

図 7. コンフィギュレーション初期化の選択

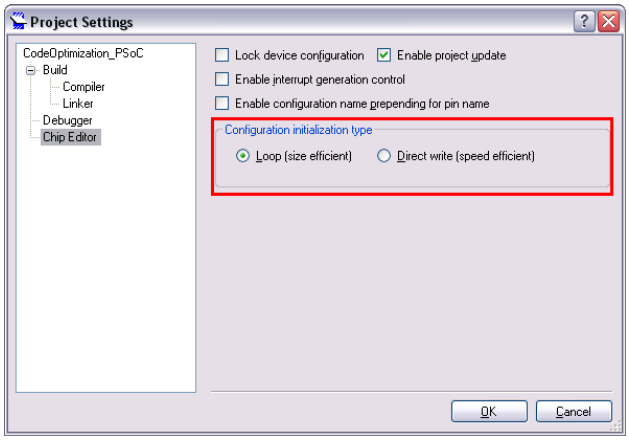


図 8. 「Loop」と「Direct write」のコードの違い

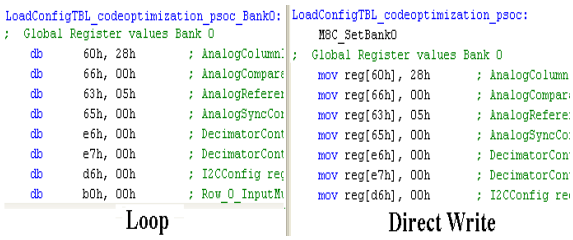
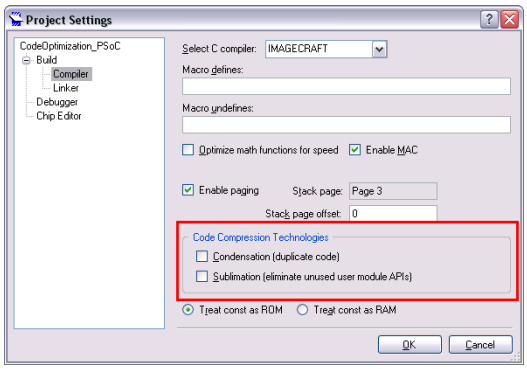


図 8 を見ると、各レジスタ書き込み用のコードが分かるようになります。**Direct write** 方式で使用されるコード (MOV reg[expr]、expr = 3 バイト) は **Loop** 方式のコード (addr、値 = 2 バイト) より 1 バイト多いです。しかし、**Loop** 方式にも 94 バイト使用する表読み出し関数が含まれています。 そのため、ロードするレジスタが 94 個以上の場合 (これは、最も単純な Designer プロジェクトを除くすべての場合に該当する)、**Loop** オプションを選択すると使用するフラッシュ容量は少なくなります。図 8 に示されているコンフィギュレーション表を使用して初期化されるレジスタの数は、設計に使用されるリソース (ユーザー モジュール) の数に依存します。リソース数を表す値が大きいほど、多くのレジスタが初期化されます。 そのため、デザインにフラッシュ節約のためにユーザー モジュールが多く含まれており、起動に少しの時間かかってもかまわない場合、**Loop** 方式を使用します。 デザインに多くのユーザー モジュールを持っていない、またはより高速なデバイスの起動を希望するなら、**Direct write** 方式を選択します。

設定 3: サブリメーション (Sublimation) とコンデンセーション (Condensation)

サブリメーション (sublimation) およびコンデンセーション (condensation) は、ImageCraft Standard 版が提供する圧縮技術です。図 9 に示されているように **Project>Settings** を選択し、ポップアップ ウィンドウで「Compiler」をクリックすることで設定を行うことができます。

図 9. サブリメーション (sublimation) とコンデンセーション (condensation) のオプション



サブリメーション (Sublimation)

「**Sublimation**」オプションを選択する場合、コンパイラがユーザー モジュール API (インターフェース コード) における未使用の関数を削除し、それによりスペースを節約します。

たとえば、プロジェクトに PGA および PWM ユーザー モジュールを設置し、両方のユーザー モジュールに「Start」関数のみを呼び出すことができます。図 10 および図 11 に示されているように、これらのユーザー モジュール API において不使用の関数を排除することにより 142 バイトのフラッシュ容量を節約できます。

図 10. サブリメーションを使用しない場合の ROM 使用状況

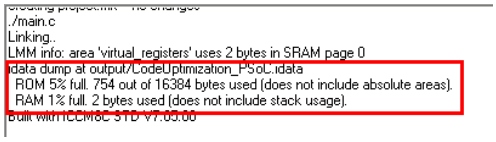
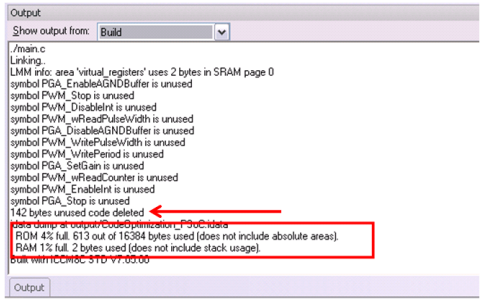
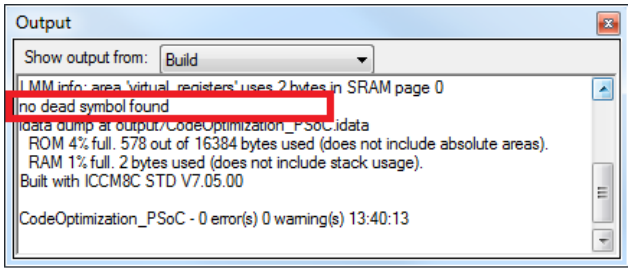


図 11. サブリメーションを使用する場合の ROM 使用状況



すべてのユーザー モジュール API の全関数がプロジェクトで使用されている場合、図 12 に示されているように、「no dead symbol found (デッド シンボルが見つかりませんでした)」というメッセージが表示されます。

図 12. 「no dead symbol found」メッセージが表示されるサブリメーション



コンデンセーション (Condensation)

Condensation オプションを選択する場合、プロジェクトで繰り返されるコードのセグメント用に関数が形成され、そのコードの各インスタンスが関数呼び出しに置き換えられます。コード 1 に示されている簡単な例では、プロジェクトの様々な場所で同じコードが 3 回繰り返されています。コンデンセーションが有効にされると、コードの単一インスタンスは関数に変換され、そのインスタンスが発生するいかなる箇所で、図 13 に示されているようにその関数への関数呼び出しに置き換えられます。そのようなすべての作成された関数は、図 13 に示されているように、コードの終わりおよび「<created procedures>」ラベルの下に配置されます。図に示すように、コード 1 用に作成された関数は「<created procedures>」ラベルで始まります。そのため、インスタンスが使用されるいかなる箇所で、「<created procedures>」ラベルの呼び出しに置き換えられます。「<created procedures>」ラベル下のロケーション「0520」および「052B」における関数は、呼び出し可能な関数に変換される、プロジェクトにおけるその他の重複コードです。

コード 1

```
/* Instance one */
shadowRegs[PORT_2] |= 0x01;
PRT2DR = shadowRegs[PORT_2];

/* Instance two */
shadowRegs[PORT_2] |= 0x01;
PRT2DR = shadowRegs[PORT_2];

/* Instance three */
shadowRegs[PORT_2] |= 0x01;
PRT2DR = shadowRegs[PORT_2];
```

図 13. *.lst* ファイルにおけるコード 1 のコンデンセーション

```
1984        0454: 90 C2     CALL   <created procedures>
1985   (0029)
1986   (0030)  /* Instance one */
1987   (0031)  shadowRegs[PORT_2] |= 0x01;
1988   (0032)  PRT2DR = shadowRegs[PORT_2];
1989   (0033)


2050        04AD: 90 69     CALL   <created procedures>
2051   (0060)
2052   (0061)
2053   (0062)  /* Instance two */
2054   (0063)  shadowRegs[PORT_2] |= 0x01;
2055   (0064)  PRT2DR = shadowRegs[PORT_2];


2139   (0108)       /* Instance three */
2140   (0109)       shadowRegs[PORT_2] |= 0x01;
2141        0505: 62 D0 00 MOV    REG[0xD0],0x0
2142        0508: 90 0E     CALL   <created procedures>
2143   (0110)       PRT2DR = shadowRegs[PORT_2];


2218   <created procedures>:
2219        0518: 2E 02 01 OR     [shadowRegs+2],0x1
2220        051B: 51 02     MOV    A,[shadowRegs+2]
2221        051D: 60 08     MOV    REG[0x8],A
2222        051F: 7F        RET
2223        0520: 62 D0 00 MOV    REG[0xD0],0x0
2224        0523: 26 02 FE AND    [shadowRegs+2],0xFE
2225        0526: 51 02     MOV    A,[shadowRegs+2]
2226        0528: 60 08     MOV    REG[0x8],A
2227        052A: 7F        RET
2228        052B: 71 10     OR     F,0x10
2229        052D: 43 08 01 OR     REG[0x8],0x1
2230        0530: 41 09 FE AND    REG[0x9],0xFE
2231        0533: 70 CF     AND    F,0xCF
2232        0535: 7F        RET
```

Condensation オプションにより、図 17 に示されているように相当量の ROM スペースを節約できます。コードにコンデンセーションを適用するには、「text」エリアにおけるコード (図 14 に示している *.mp* ファイルに見ることができます) が 256 バイトを超える必要があります。「text」エリアが 256 バイト未満の場合、「program code in 'text' area too small for worthwhile code compression」(「text」エリア内のプログラム コードは、コード圧縮には小さ過ぎます) というエラー メッセージが表示されます (図 15)。コンデンセーションを行うための重複コードのインスタンスが見つからない場合、「no worthwhile duplicate found」(有効な重複が見つかりませんでした) というエラー メッセージが表示されます (図 16)。

注 図 15、図 16、図 17 に表示される情報は、コード コンデンセーション中の異なるシナリオをエミュレートするための異なるコードのものです。

図 14. *.mp* ファイル内の「text」エリア

```
Area                              Addr   Size   Decimal Bytes (Attributes)
--------------------------------  ----   ----   ------- ----- ------------
                                  text   0445   0143 =   323. bytes (rel,con,rom,code)

       Addr  Global Symbol
      -----  --------------------------------
       0445  __text_start
       0445  _main
       0588  __xidata_start
       0559  <created procedures>
       0588  __text_end
       058E  __xidata_end
```

図 15. 「text」エリアが 256 バイト未満の場合のコンデンセーション

```
Output
Show output from: Build
Starting MAKE...
creating project.mk -- no changes
./main.c
Linking..
LMM info: area 'virtual_registers' uses 2 bytes in SRAM page 0
program code in 'text' area too small for worthwhile code compression
idata dump at output/CodeOptimization_PSoC.idata
  ROM 6% full. 863 out of 16384 bytes used (does not include absolute areas).
  RAM 1% full. 2 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

図 16. 重複コードが見つからない場合のコンデンセーション

```
Output
Show output from: Build
Starting MAKE...
creating project.mk -- no changes
./main.c
Linking..
LMM info: area 'virtual_registers' uses 2 bytes in SRAM page 0
no worthwhile duplicate found
idata dump at output/CodeOptimization_PSoC.idata
  ROM 7% full. 1055 out of 16384 bytes used (does not include absolute areas).
  RAM 1% full. 2 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

図 17. コード 1 におけるコンデンセーション (コード 1 は text エリアが 256 バイトを超える大きいプロジェクトの一部)

```
Output
Show output from: Build
Starting MAKE...
creating project.mk -- no changes
./main.c
Linking..
LMM info: area 'data' item of 5 bytes allocated in SRAM page 0
LMM info: area 'virtual_registers' uses 4 bytes in SRAM page 0
374 bytes before Code Compression, 323 after. 12% reduction.
idata dump at output/CodeOptimization_PSoC.idata
  ROM 7% full. 1063 out of 16384 bytes used (does not include absolute areas).
  RAM 2% full. 9 bytes used (does not include stack usage).
Built with ICCM8C STD V7.05.00
```

注 コンデンセーションが相当量の ROM スペースを減少するが、重複コードを関数呼び出しで置き換えるため、プログラム実行に遅延を加えます。

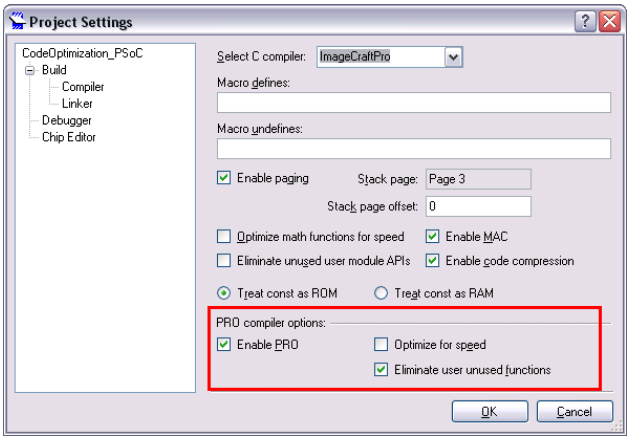
設定 4: ROM 定数対 RAM 定数

このオプション (図 9 を参照) は、フラッシュ (ROM) に定数を置くことで RAM の使用容量を削減しています。フラッシュの使用量を大幅に削減するわけではありません。 大部分のプロジェクトには **Treat const as ROM** (定数を ROM で扱う) オプションを選択します。**Treat const as RAM** (RAM の定数として扱う) オプションは、主に ImageCraft コンパイラの旧版との下位互換性のために使用されます。

ImageCraft Pro コンパイラ オプション

ImageCraft Standard コンパイラで提供される最適化オプションに加えて、ImageCraft Pro は図 18 に示されるようにその他のオプションを提供します。

図 18. ImageCraft Pro コンパイラ オプション



ImageCraft Standard コンパイラで提供される **Sublimation** および **Condensation** オプションは、それぞれ ImageCraft Pro コンパイラの「**Eliminate unused user module APIs**」(未使用のユーザー モジュール API を削除する) オプションおよび「**Enable Code Compression**」(コード圧縮を有効にする) オプションと同じです。

ImageCraft Pro は、「**Eliminate user unused functions**」オプションを提供しています。これは、ユーザーのプロジェクトが巨大でユーザーが使用していない関数を発見することが困難な場合に非常に役に立ちます。 わかりやすいシナリオはプロジェクトがリビジョンを重ねることで、未使用の API がコードに残存し、ROM スペースを消費する傾向があります。 このオプションが有効にされると、コンパイラは、未使用の API がコードに設置されていないこと、そして ROM スペースがその他の用途に使用可能であることを確認します。

ImageCraft Pro で使用可能なもう 1 つのオプションは、「**Optimize for speed** 」(コードの速度を最適化) です。このオプションを有効にすると、コードはより高速に実行するようにコンパイルされます。 最適化が適用される場所に応じて ROM の使用量が増加、または減少します。 たとえば、コードにおける小さいループは連続命令へと拡張されるため、ROM の使用量が増加します。 冗長な割り当てまたは転送命令が削除されるため、ROM の使用が減少します。

ヒントおよびガイドライン

いくつかのコーディング技術によりユーザーのファームウェアをより効率的にすることができます。これらの方法は PSoC Designer プロジェクトおよび PSoC 1 で使用可能であり、また、PSoC 3 および PSoC Creator™などの類似する 8 ビット プロセッサ、IDE およびコンパイラに適用可能です。

ガイドライン 1: 割り込みサービス ルーチンにおける関数呼び出しを回避

割り込みサービス ルーチン (ISR) の C コードをコンパイルする場合、ImageCraft コンパイラは ISR が使用する仮想レジスタ (一時値を保存するためにコンパイラが使用するレジスタ) をすべてスタックにプッシュします。 ISR コードに関数呼び出しが含まれている場合、コンパイラは呼び出された関数により修正されるレジスタを見分けられません。 ImageCraft C コンパイラは、最大で 15 の仮想レジスタを使用して一時データをスタックに保存します。

256 バイトを超える RAM を備えた PSoC チップでは、これら 15 の仮想レジスタと共に 4 つのページ ポインタも保存および復元されます。コード 2 の例では、コンパイラが仮想レジスタを保存する必要はありません (図 19)。コード 3 に示されているように関数呼び出しを使用して実装する場合、図 20 に示されているように、同じ機能でも 15 の仮想レジスタを余計に使用します。 各レジスタは、次の追加のオーバーヘッド コードを必要とします: MOV [2 バイト] + PUSH [1 バイト] + POP [1 バイト] + MOV [2 バイト]で、レジスタごとに合計 6 バイト。コード 3 は、これらの多くの MOV/PUSH 命令のため、さらに 90 バイトを使用し、遅延、長期の ISR 実行を必要とします。

コード 2

```
BYTE bVar1;

#pragma interrupt_handler
SleepTimerHandler;

void SleepTimerHandler(void)
{
    bVar1 = 1;
}
```

図 19. 関数呼び出しを伴わない ISR 用に生成されたコード (コード 2)

```
(0012) #pragma interrupt_handler SleepTimerHandler;
(0013)
(0014) void SleepTimerHandler(void)
(0015) {
_SleepTimerHandler:
    0445: 71 C0     OR    F,0xC0
    0447: 08        PUSH  A
    0448: 5D D0     MOV   A,REG[0xD0]
    044A: 08        PUSH  A
(0016)      bVar1 = 1;
    044B: 62 D0 00  MOV   REG[0xD0],0x0
    044E: 55 09 01  MOV   [bVar1],0x1
    0451: 18        POP   A
    0452: 60 D0     MOV   REG[0xD0],A
    0454: 18        POP   A
    0455: 7E        RETI
(0017) }
```

コード 3

```
BYTE bVar1;
void TestFunc()
{
    bVar1 = 1;
}

#pragma interrupt_handler
SleepTimerHandler;

void SleepTimerHandler(void)
{
    TestFunc();
}
```

図 20. 関数呼び出しを伴った ISR 用に生成されたコード (コード 3)

```
_SleepTimerHandler:
    02C7: 08        PUSH  A
    02C8: 51 10     MOV   A,[__r0]
    02CA: 08        PUSH  A
    02CB: 51 0F     MOV   A,[__r1]
    02CD: 08        PUSH  A
    02CE: 51 0E     MOV   A,[__r2]
    02D0: 08        PUSH  A
    02D1: 51 0D     MOV   A,[__r3]
    02D3: 08        PUSH  A
    02D4: 51 0C     MOV   A,[__r4]
    02D6: 08        PUSH  A
    02D7: 51 0B     MOV   A,[__r5]
    02D9: 08        PUSH  A
    02DA: 51 0A     MOV   A,[__r6]
    02DC: 08        PUSH  A
    02DD: 51 09     MOV   A,[__r7]
    02DF: 08        PUSH  A
    02E0: 51 08     MOV   A,[__r8]
    02E2: 08        PUSH  A
    02E3: 51 07     MOV   A,[__r9]
    02E5: 08        PUSH  A
    02E6: 51 06     MOV   A,[__r10]
    02E8: 08        PUSH  A
    02E9: 51 05     MOV   A,[__r11]
    02EB: 08        PUSH  A
    02EC: 51 04     MOV   A,[__rX]
    02EE: 08        PUSH  A
    02EF: 51 03     MOV   A,[__rY]
    02F1: 08        PUSH  A
    02F2: 51 02     MOV   A,[__rZ]
    02F4: 08        PUSH  A
(0032)      TestFunc();
    02F5: 9F CC     CALL  TestFunc|__text_start|_TestFunc
    02F7: 18        POP   A
    02F8: 53 02     MOV   [__rZ],A
    02FA: 18        POP   A
    02FB: 53 03     MOV   [__rY],A
    02FD: 18        POP   A
    02FE: 53 04     MOV   [__rX],A
    0300: 18        POP   A
    0301: 53 05     MOV   [__r11],A
    0303: 18        POP   A
    0304: 53 06     MOV   [__r10],A
    0306: 18        POP   A
    0307: 53 07     MOV   [__r9],A
    0309: 18        POP   A
    030A: 53 08     MOV   [__r8],A
    030C: 18        POP   A
    030D: 53 09     MOV   [__r7],A
    030F: 18        POP   A
    0310: 53 0A     MOV   [__r6],A
    0312: 18        POP   A
    0313: 53 0B     MOV   [__r5],A
    0315: 18        POP   A
    0316: 53 0C     MOV   [__r4],A
    0318: 18        POP   A
    0319: 53 0D     MOV   [__r3],A
    031B: 18        POP   A
    031C: 53 0E     MOV   [__r2],A
    031E: 18        POP   A
    031F: 53 0F     MOV   [__r1],A
    0321: 18        POP   A
    0322: 53 10     MOV   [__r0],A
    0324: 18        POP   A
    0325: 7E        RETI
```

ガイドライン 2: 算術関数の使用を制限

多くの場合、C コンパイラはインライン アセンブラ コードを使用して簡単な算術計算を実行します。しかし、複雑な計算については、代わりに 1 つまたはそれ以上の算術ライブラリ関数をコードに追加することがあります。 これは即座には明らかにならない可能性があります。 処理および変数の種類によりますが、「+」、「-」、「*」、「/」、「%」、「>」、および「<」のような単純な C 演算子でさえもライブラリ関数を実行用に必要とすることがあります。

それらライブラリ関数は極めて効率的で有益ですが、ライブラリ関数はあなたが予想できない多くのコード スペースを占めることがあります。 多くの場合、データ型やコード演算子を注意深く管理することにより、ライブラリ関数の過度の使用を回避できます。

プログラムが整数演算を使用する場合、使用される変数のサイズおよび種類 (8、16 または 32 ビット、符号付きまたは符号なし) に応じて算術ライブラリ関数が追加されます。 様々な関数のバイト使用量に関する詳細は、**Help > Documentation > Designer Specific Documents** の下の *Libraries User Guide* に記載されています。

基礎算術関数 (すなわち常に使用されている関数) には、整数の加算、減算およびシフトが含まれています。 乗算などのその他の演算が使用される場合、これらの演算のコードも含まれます。

乗除算の代わりにシフトおよび加算を使用

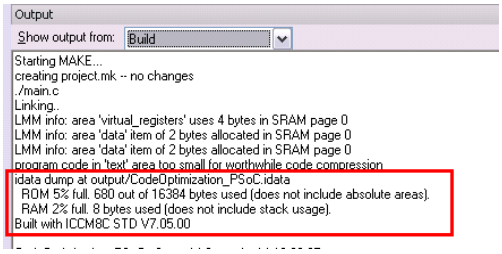
符号なし整数の乗除算の代わりにビット シフトおよび加算のような小技を使用することは、コード スペースを節約します。 符号なし整数では、単一のビット単位のシフト右が 2 による除算に相当し、シフト左が 2 による乗算に相当します。コード 5 に示すように、シフトおよび加算を使用することで、乗除算ライブラリ関数を含む必要はありません。

同じ結果をもたらす次の 2 つのコードのスニペットでは、図 21 と図 22 に示されているように、コード 4 の実装はコード 5 よりも 55 バイト多く使用します。この違いは、コードに「__mul16」関数が追加されているためです。

コード 4

```
unsigned int iTest1, iTest2;
void main(void)
{
    iTest1 = iTest2 *3;
}
```

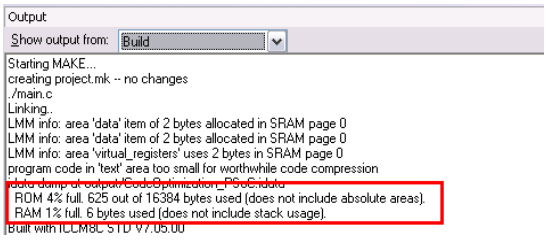
図 21. コード 4 の ROM 使用量



コード 5

```
unsigned int iTest1, iTest2;
void main(void)
{
    iTest1 = (iTest2 << 1) + iTest2;
}
```

図 22. コード 5 の ROM 使用量



冪関数を避ける

冪関数も算術演算ライブラリ関数を追加します。 冪関数を通分して乗算または除算演算を実行することもできます。例えば、4^2= 4×4 です。 前のセクションで説明したように、乗除算は、シフトおよび加算を使用して実行できます。 そのため、これら関数の実装をできる限りシフトおよび加算に換算することにより、限られた量の冪関数を有するプログラムにおいてコードス ペースを節約できます。

浮動小数点計算の回避

8 ビット プロセッサの浮動小数点演算は、ほぼ常にライブラリ関数を必要とします。 基礎算術演算実行用の関数に加えて、四捨五入、正規化および特別な条件の照合のための共用関数もコードに追加される可能性があります。 浮動小数点ライブラリの詳細については、「*Arithmetic Libraries User Guide*」(演算ライブラリ ユーザー ガイド) を参照してください。 概算として参考までに、次の浮動小数点関数に必要なコード スペースは以下の通りです (「*Arithmetic Libraries User Guide* 」(演算ライブラリ ユーザー ガイド) にも記載されています)。

比較 (*_fpcmp) = 109–125 バイト

加算 (*_fpadd) = 461–478 バイト

減算 (*_fpsub) = 468–485 バイト

乗算 (*_fpmul) = 406–558 バイト

除算 (*_fpdiv) = 432–449 バイト

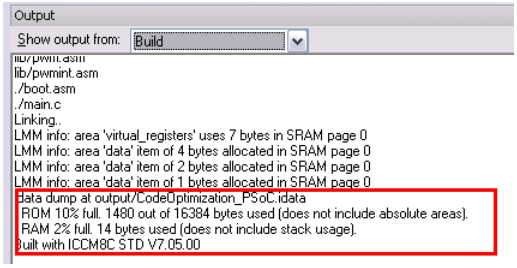
浮動小数点算術関数は、整数算術ライブラリをベースとして使用します。 これは、浮動小数点演算を使用する際は、整数算術ライブラリも必要となることを意味します。 浮動小数点演算を使用する代わりに、変数の範囲が既知の場合は、変数をスケールして整数演算を直接使用することができます。

たとえば、以下の 2 つのコードでは、変数の範囲は小数点以下第二位までであることが判明しています。 そのため、すべての浮動小数点数に 100 を掛けると、浮動小数点演算の代わりに整数演算を使用できます。 この例では、図 23 および図 24 に示されているように、整数演算法のコード 7 の使用量は、内容が同じだが浮動小数点を用いたコード 6 より 761 バイト少なくなります。

コード 6

```
int iTest2, iTest3;
float fTest1;
void main(void)
{
    fTest1 = iTest2 * 2.42;
    if(fTest1 > 7.5)
    {
        iTest3 = 2;
    }
    else
    {
        iTest3 = 1;
    }
}
```

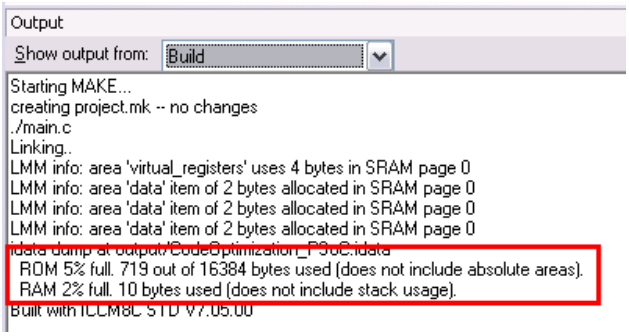
図 23. コード 6 の ROM 使用量



コード 7

```
int iTest1, iTest2, iTest3;
void main(void)
{
    iTest1 = iTest2 * 242;
    if(iTest1 > 750)
    {
        iTest3 = 2;
    }
    else
    {
        iTest3 = 1;
    }
}
```

図 24. コード 7 の ROM 使用量



計算の代わりにルックアップ テーブルを使用

計算よりもルックアップ テーブル (LUT) を使用したほうが効率的な場合があります。 速度、正確性およびコード スペースなど、複数のトレードオフがあります。 アプリケーションの種類に応じて選択します。

たとえば、AN2017 – PSoC 1 のサーミスタによる温度測定 で提供されるプロジェクトは、浮動小数点および LUT 法で実装するオプションを提供しています。 このプロジェクトでは浮動小数点演算の代わりに、LUT を使用することによりメモリを 4187 バイト (ルックアップ テーブル用の 3779 バイト、浮動小数点方程式用の 7966 バイト) 節約しますが、精度のトレードオフがあります。

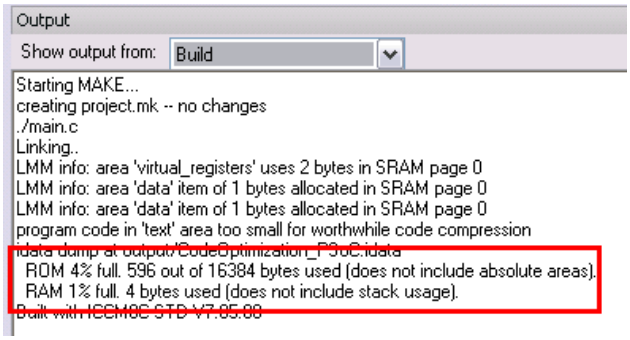
データ型の一貫性の維持

変数のデータ型を選択する際は、関数に必要とされる最も小さいデータ型を選択します。 たとえば、変数の最大値が 256 を超えない場合は int の代わりに BYTE を選択します。 同様に、適用可能な場合では、符号付き変数の代わりに符号なし変数を選択します。 例えば、プログラムのにおいて変数「iTest1」と「iTest2」が 255 を超えない場合は、コード 4 を検討します。この場合、コード 8 に示すように「unsigned int」の代わりに「unsigned char」として宣言することができます。 ROM 使用量は図 25 に示されています。コード 8 はコード 4 よりも 84 バイト少なく消費します。

コード 8

```
unsigned char cTest1, cTest2;
void main(void)
{
    cTest1 = cTest2 *3;
}
```

図 25. コード 8 の ROM 使用量



符号付き変数の正および負の範囲が判明した場合、その変数をずらして正値にし、符号なしの算術関数を使用できます。 同様に、変数が–10～+10 の範囲内の値を有することが見込まれる場合、コード 0 が–10 に対応してコード 20 が+10 に対応するように、値を 10 ずらします。

コード スペースという点で型キャスティングは高価であるため、使用されるデータ型の一貫性を維持します。 プログラムにおいて型キャスティングの使用が必須の場合、算術ライブラリのコードスペースを節約するために、変数の型をキャスティングして同じ算術ライブラリを使用します。

ガイドライン 3: 配列インデックス対ポインタ

配列インデックス法では、ImageCraft はインデックスを使用して各アドレスに直接アクセスします。 アドレスは定数のままとなるため、計算は必要ありません。 ポインタ アクセスの場合、各アクセスは変数であるポインタに基づいています。 したがって、コンパイラはアドレスを入手するために、より多くの計算を実行する必要があります。

アクセス法が繰り返し使用される場合、アクセスの種類の違いがメモリ使用量に大幅な変化をもたらします。

たとえば、以下の 2 つのコード例について検討します。図 26 および図 27 に示されているように、コード 9 はコード 10 より 25 バイト多く使用します。 そのため、使用されるアクセス方法の種類 (配列インデックス対ポインタ) を注意深く観察することは、コードの最適化にとって重要です。 アクセスの種類および変数の種類には数多くのバリエーションがあります。 アドレスが定数である (その場合にはコンパイラがアドレスを直接に置き換える) ことの確認がキーとなります。 実行中にアドレス計算が必要な場合、コンパイラはより多くのコード スペースを使用します。 一般に、「->」演算子はコード スペースという点で高価です。

コード 9

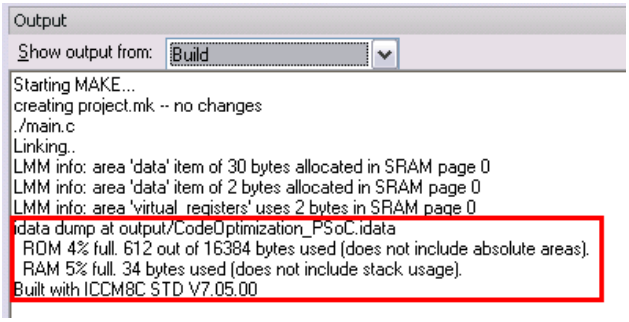
```
typedef struct
{
    int iData;
    BYTE bData;
}sData;

typedef struct
{
    sData myArray[10];
}sArray;

sArray myTest;
sData* myPtr;

void main(void)
{
    myPtr = myTest.myArray;
    myPtr->iData = 100;
}
```

図 26. コード 9 の ROM 使用量



コード 10

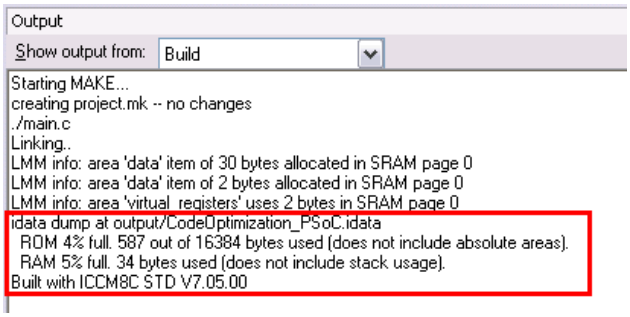
```
typedef struct
{
    int iData;
    BYTE bData;
}sData;

typedef struct
{
    sData myArray[10];
}sArray;

sArray myTest;
sData* myPtr;

void main(void)
{
    myTest.myArray[1].iData = 100;
}
```

図 27. コード 10 の ROM 使用量



ガイドライン 4: スイッチ文と if-else 文の使用

条件判断は、スイッチ文または if-else 文のどちらかを使用できます。2 文の相違点は、スイッチ文が常に 16 ビットの比較をするのに対して、if-else 文が変化する型に基づいて比較を行います。1 バイトの変数 (BYTE) の場合、ImageCraft コンパイラは、if-else 文を使用することで、スイッチ文と比べてより効率的なコードを作り出します。これは、8 ビットの比較が必要とするコード スペースは 16 ビットの比較より少ないためです。

スイッチ文は追加で 9 バイト、さらに各ケースにつき 5 バイトを使用します。たとえば、コード 11 に示されているようなデフォルトの項目を備えた 4 ケースのスイッチ文は、;

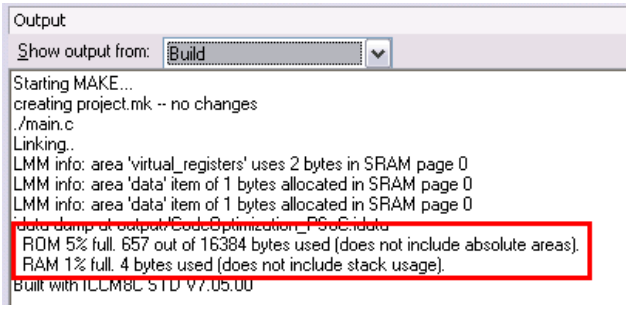
```
            break;
        }
        case 3:
        {
            bTest2 = 2;
            break;
        }
        case 2:
        {
            bTest2 = 3;
            break;
        }
        default:
        {
            bTest2 = 4;
        }
    }
}
```

図 28 および図 29 に示されるように、同様のコード 12 よりも (9 + 4 × 4) = 25 バイト多く使用します。

コード 11

```
BYTE bTest1, bTest2;
void main(void)
{
    switch(bTest1)
    {
        case 4:
        {
            bTest2 = 1;
            break;
        }
        case 3:
        {
            bTest2 = 2;
            break;
        }
        case 2:
        {
            bTest2 = 3;
            break;
        }
        default:
        {
            bTest2 = 4;
        }
    }
}
```

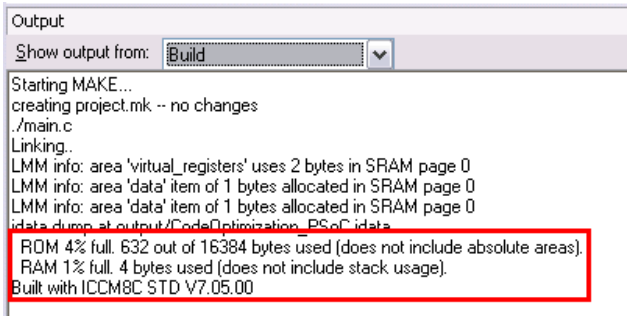
図 28. コード 11 の ROM 使用量



コード 12

```
BYTE bTest1, bTest2;
void main(void)
{
    if(bTest1 == 4)
    {
        bTest2=1;
    }
    else if(bTest1 == 3)
    {
        bTest2 = 2;
    }
    else if(bTest1 ==2)
    {
        bTest2 = 3;
    }
    else
    {
        bTest2=4;
    }
}
```

図 29. コード 12 の ROM 使用量



条件判断が 2 バイト変数 (WORD) 向けである場合、結果的なコード サイズはいずれの実装においてもほぼ同一です。

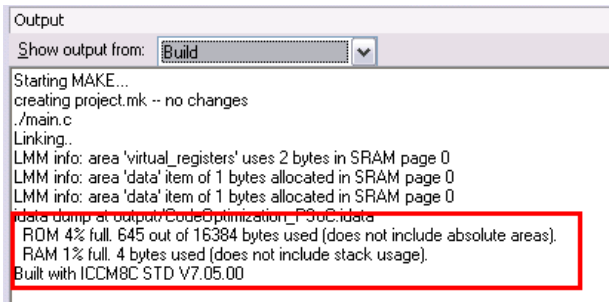
ガイドライン 5: アセンブラでコードの一部を書く

アセンブラでプログラムを書くことにより、コンパイラの誤解釈を回避し、完全なユーザー最適化を可能にします。 アセンブラでプログラム全体を書くことは退屈で煩わしいですが、コードの一部だけをアセンブリ言語に変換することによりコードのサイズやパフォーマンスを最適化できます。コード 11 のように、アセンブリで bTest2 変数への割り当てを伴うコード 13 を検討します。図 30 のように、コード 13 の消費量はコード 11 より 12 バイト少なくなります。

コード 13

```
BYTE bTest1, bTest2;
void main(void)
{
    switch(bTest1)
    {
        case 4:
        {
            asm("MOV [_bTest2], 0x1");
            break;
        }
        case 3:
        {
            asm("MOV [_bTest2], 0x2");
            break;
        }
        case 2:
        {
            asm("MOV [_bTest2], 0x3");
            break;
        }
        default:
        {
            asm("MOV [_bTest2], 0x4");
        }
    }
}
```

図 30. コード 13 の ROM 使用量



ガイドライン 6: PSoC 1 のビットを操作

PSoC1 では「ビット」変数向けに定義されたデータ型がありません。変数のビット／複数ビットを操作するためには、変数のマスクを論理演算子によりまたは変数に直接割り当てて使用することができます。 マスクは、操作要件によって定数になる場合があります。コード 14 とコード 15 は、変数内のビットを操作する方法を示しています。

コード 14

```
BYTE bTest1, bTest2, bTest3;
void main(void)
{
  /* To set first, sixth and eighth bit
  of the variable */
  bTest1 = (0xA1);

  /*  To  clear  the  second  and  the
  seventh bits the variable */
  bTest2 = bTest2 & (0xBD);

  /* To invert the second, fifth and
  eighth bits in the variable */
  bTest3 = bTest3 ^ (0x92);
}
```

変数は、複数のビットの定義を有し、1 つ以上のビットが変数内の他のビットに影響を与えずに操作しなければならない場合、ビットを操作するために論理演算子を使用します。 これらの操作は、XOR、AND、および OR 命令を使用します。 コード 14 に該当する.*asm* コードは以下のように<*プロジェクト名*>.*lst* ファイル (**Workspace＞Output Files**) に生成されます。

```
  /* To set first, sixth and eighth bit
  of the variable */
  bTest1 = bTest1 | (0xA1);
  __text_start|_main|_main:
  62 D0 00 MOV   REG[0xD0],0x0
  2E 03 A1 OR    [bTest1],0xA1

  /* To clear the second and the
  seventh bits the variable */
  bTest2 = bTest2 & (0xBD);
  62 D0 00 MOV   REG[0xD0],0x0
  26 02 BD AND   [bTest2],0xBD

  /* To invert the second, fifth and
  eighth bits in the variable */
  bTest3 = bTest3 ^ (0x92);
  62 D0 00 MOV   REG[0xD0],0x0
  51 04    MOV   A,[bTest3]
  31 92    XOR   A,0x92
  53 04    MOV   [bTest3],A
```

```
  address: 8F FF     JMP    address
```

注 MOV REG[0xD0],0x0 命令は、グローバル変数「bTest1/bTest2/bTest3」の RAM ページ番号をセットします。

変数のサイズが単一のビットであるか、または変数で定義されているすべてのビットが同時に割り当て／操作されている場合は、コード15のように、直接割り当てを使用して操作します。この場合、XOR、AND、およびORよりも1サイクル速いMOV命令を使用します。

注 命令の実行に要するサイクル数を調べるには**「Help」>「Documentation 」「 Compiler and Programming Documents」**にある *Assembly Language User Guide*、PSoC Designer の付録 A5、「Instruction Set Summary」を参照してください。

コード15

```
BYTE bTest1, bTest2;
void main(void)
{
  /* To clear the first and the second
  bits and set the third bit of a three
  bits variable */
  bTest1 = 0x04;

 /* To invert the variable */
 bTest2 = ~ (bTest2);
}
```

該当する *.asm* コードは以下の通りです。

```
BYTE bTest1, bTest2;
void main(void)
{
  /* To clear the first and the second
  bits and set the third bit of a three
  bits variable */
  bTest1 = 0x04;
  __text_start|_main|_main:
  62 D0 00 MOV    REG[0xD0],0x0
  55 02 04 MOV    [bTest1],0x4

  /* To invert the variable */
  bTest2 = ~ (bTest2);
  62 D0 00 MOV    REG[0xD0],0x0
}
  51 03     MOV    A,[bTest2]
  73        CPL    A
  53 03     MOV    [bTest2],A
  address: 8F FF     JMP    address
```

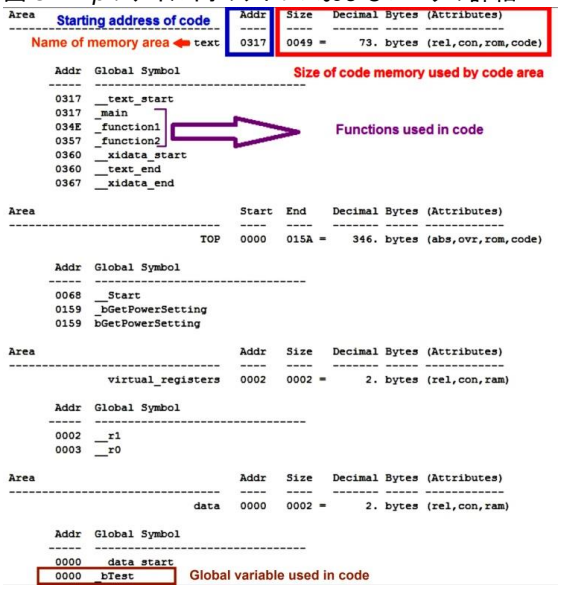
注 MOV REG[0xD0],0x0 命令は、「bTest1/bTest2/bTest3」変数の RAM ページ番号を設定します。

ガイドライン 7: C コードのフラッシュ使用率および実行時間の計算

コード メモリ使用量およびプログラム実行時間が関心事である場合、関数のフラッシュの使用量やコードとそのコードの実行時間を知るのは手動による最適化に役立ちます。

関数のコード サイズを調べるには **Workspace > Output Files** にある*<プロジェクト名>.mp* というファイルを参照してください。*.mp* ファイルは、コード エリアとそのエリアで定義された関数／定数によってコード メモリの使用状況に関する詳細を提供します。また、図 31 に示すように、RAM エリアとそれらのエリアで定義されたグローバル変数のアドレスで、RAM の使用状況に関する詳細も提供します。

図 31.*.mp* ファイル内のアドレスおよびメモリの詳細



注 コード／RAM エリアの詳細については、C 言語コンパイラ ユーザー ガイド、セクション 6.9、、およびアセンブリ言語のユーザー ガイド、セクション 5.1 を参照してください。

関数のコード サイズを調べるには、*.mp* ファイル内の関数名を検索します。カスタム エリアで関数を配置していない場合、デフォルトでは、すべてのユーザー定義関数が「text」コード エリアに配置されます (図 32)。図 32 では、コード 16 で定義された「main」、「function1」、および「function2」の開始アドレスを示しています。「main」のコード サイズを調べるためには、「main」のアドレスから以下の関数 (この場合「function1」) のアドレスを引きます。「main」のサイズは 55 バイト (0x034E – 0x0317 = 0x0037) で、「function1」は 9 バイト (0x0357 – 0x034E = 0x0009)で、「function2」は 9 バイト (0x0360 – 0x0357 = 0x0009)です。

コード 16

```
int bTest;
void main(void)
{
bTest = ++bTest;
}

void function1(int a)
{      a++;
       return;
}

void function2(int b)
{      b--;
       return;
}
```

図 32..*mp* ファイルを使用したフラッシュ計算

```
Area                              Addr   Size
--------------------------------  ----   ----
                            text  0317   0049

      Addr  Global Symbol
     -----  --------------------------------
      0317    text start
      0317  _main
      034E  _function1
      0357  _function2
      0360    xidata start
      0360  __text_end
      0367  __xidata_end
```

コード実行時間を計算するには、個別の命令の実行時間を把握する必要があります。個別の命令の実行時間を調べるには **Help ＞ Documentation ＞ Compiler and Programming Documents」**にある *Assembly Language User Guide*、PSoC Designer の付録 A5、「Instruction Set Summary」を参照してください。 表の「CPU クロック サイクル」の欄は、命令を実行するのに必要な CPU クロック数を提供します。 命令の実行期間を調べるには、CPU のクロック周期で、その値を掛けます。 以下の実施例は、異なるコードでの実行時間の計算を示します。

例 1: 次のように空「while (1)」ループを取り上げます。

```
void main(void)
{
while (1)
       {
       }
}
```

.asm<プロジェクト名>.lst ファイルで生成された「while (1)」ループに該当する.asm コードは以下の通りです。

```
while (1)
{
}
address: 8X XX     JMP    address
```

JMP命令は、実行に5 CPUサイクルかかります。 CPUクロックが24MHzであれば、JMP命令の実行時間は以下になります。

$$5 \times \frac{1}{24}\ \mu s = 0.208\ \mu s$$

この速度では、CPUは、約0.208μ*S*あたり1 (JMP) 命令を実行し、つまり1秒あたり(1/0.208) = 480万の命令を実行します。

例 2:
次のように単純な加算の例を検討します。

```
void main(void)
{
BYTE bTest1, bTest2;
while(1)
{
       bTest1 = bTest1 + bTest2;
}
}
```

対応.*asm*コードは以下のとおりです。

```
BYTE bTest1, bTest2;
while(1)
{
       bTest1 = bTest1 + bTest2;
    address1: 52 01     MOV    A,[X+1]
    address2: 05 00     ADD    [X+0],A
}
    address3: 8X XX     JMP    address1
```

前コードの実行は6 (MOV) + 8 (ADD) + 5 (JMP) = 19クロック サイクルかかります。 この場合、平均1命令当たりは19/3=6.33サイクルです。 CPUクロックが24MHzであれば、コード全体の実行時間は以下になります。

$$\frac{19}{24{,}000{,}000} s = 0.792\ \mu s$$

このレートでは、CPUは、0.792/3 = 0.264μSあたり1命令を実行し、つまり1秒あたり (1/0.264) = 37.9万の命令を実行します。

結論

上記のガイドラインはコードの最適化に役立ちます。その中には、ImageCraft コンパイラ特有のものもあれば、一般的なものもあります。 ファームウェアの開発中または開発後にこれら

の提案に従うことは、コード スペースの最適化に役立ちます。すべてのコード スニペットおよびスナップショットは、CY8C28xxx デバイスおよび ImageCraft Standard コンパイラ 7.0.5 版についてテストおよびキャプチャされました。その他の PSoC1 デバイスの場合でも結果が同様でしょう。

これらのガイドラインに加えて、小フラッシュ容量のデバイスを取り扱っている場合、良い習慣としてはプロジェクトの開発中にコード サイズの増加を監視することです。コード スペースが大幅に増加した場合は常にマップ ファイル (図 4) を検証し、コードの最新部分が何らかの予想外の関数を使用しているかを確認してください。フラッシュの増加を引き起こすコードの正確な部分を知ることにより、コードの最適化が簡単になります。

コンパイラおよびプロジェクトの最適化オプションの詳細については、*ImageCraft C Compiler Guide* および *ImageCraft Assembly Language Guide* を参照してください。

著者について

名前:	Archana Yarlagadda
役職:	アプリケーション エンジニア
経歴:	サイプレスのアプリケーション エンジニア、PSoC 担当 ノックスビルのテネシー大学においてアナログ VLSI の修士号取得
連絡先:	msur@cypress.com

ドキュメントの改版履歴

文書名: PSoC® 1 M8C ImageCraft C コード最適化 – AN60486

文書番号: 001-82519

版	ECN	変更者	発行日	変更内容
**	3732388	HZEN	09/03/2012	これは英語版 001-60486 Rev. *B からを翻訳した日本語版 001-82519 Rev. **です。
*A	4536599	HZEN	10/13/2014	これは英語版 001-60486 Rev. *E からを翻訳した日本語版 001-82519 Rev. *A です。

ワールドワイドな販売と設計サポート

サイプレスは、事業所、ソリューション センター、メーカー代理店、および販売代理店の世界的なネットワークを保持しています。お客様の最寄りのオフィスについては、サイプレスのロケーション ページをご覧ください。

製品

車載用	cypress.com/go/automotive
クロック＆バッファ	cypress.com/go/clocks
インターフェース	cypress.com/go/interface
照明 ＆電源管理	cypress.com/go/powerpsoc cypress.com/go/plc
メモリ	cypress.com/go/memory
PSoC	cypress.com/go/psoc
タッチ センシング	cypress.com/go/touch
USB コントローラー	cypress.com/go/usb
ワイヤレス／RF	cypress.com/go/wireless

PSoC®ソリューション

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

サイプレス開発者コミュニティ

コミュニティ | フォーラム | ブログ | ビデオ | トレーニング

テクニカル サポート

cypress.com/go/support

PSoC は、サイプレス セミコンダクタ社の登録商標です。PSoC Creator および PSoC Designer は、サイプレス セミコンダクタ社の商標です。本書で言及するその他すべての商標または登録商標は、各社の所有物です。

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation、2010-2014. 本文書に記載される情報は、予告なく変更される場合があります。Cypress Semiconductor Corporation (サイプレス セミコンダクタ社) は、サイプレス製品に組み込まれた回路以外のいかなる回路を使用することに対して一切の責任を負いません。 サイプレス セミコンダクタ社は、特許またはその他の権利に基づくライセンスを譲渡することも、または含意することもありません。 サイプレス製品は、サイプレスとの書面による合意に基づくものでない限り、医療、生命維持、救命、重要な管理、または安全の用途のために使用することを保証するものではなく、また使用することを意図したものでもありません。 さらにサイプレスは、誤動作や故障によって使用者に重大な傷害をもたらすことが合理的に予想される生命維持システムの重要なコンポーネントとしてサイプレス製品を使用することを許可していません。 生命維持システムの用途にサイプレス製品を供することは、製造者がそのような使用におけるあらゆるリスクを負うことを意味し、その結果サイプレスはあらゆる責任を免除されることを意味します。
このソースコード (ソフトウェアおよび／またはファームウェア) はサイプレス セミコンダクタ社 (以下「サイプレス」) が所有し、全世界の特許権保護 (米国およびその他の国)、米国の著作権法ならびに国際協定の条項により保護され、かつそれらに従います。 サイプレスが本書面によりライセンシーに付与するライセンスは、個人的、非独占的かつ譲渡不能のライセンスであり、適用される契約で指定されたサイプレスの集積回路と併用されるライセンシーの製品のみをサポートするカスタム ソフトウェアおよび／またはカスタム ファームウェアを作成する目的に限って、サイプレスのソースコードの派生著作物をコピー、使用、変更そして作成するためのライセンス、ならびにサイプレスのソース コードおよび派生著作物をコンパイルするためのライセンスです。 上記で指定された場合を除き、サイプレスの書面による明示的な許可なくして本ソース コードを複製、変更、変換、コンパイル、または表示することは全て禁止します。
免責条項: サイプレスは、明示的または黙示的を問わず、本資料に関するいかなる種類の保証も行いません。これには、商品性または特定目的への適合性の黙示的な保証が含まれますが、これに限定されません。 サイプレスは、本文書に記載される資料に対して今後予告なく変更を加える権利を留保します。 サイプレスは、本文書に記載されるいかなる製品または回路を適用または使用したことによって生ずるいかなる責任も負いません。 サイプレスは、誤動作や故障によって使用者に重大な傷害をもたらすことが合理的に予想される生命維持システムの重要なコンポーネントとしてサイプレス製品を使用することを許可していません。 生命維持システムの用途にサイプレス製品を供することは、製造者がそのような使用におけるあらゆるリスクを負うことを意味し、その結果サイプレスはあらゆる責任を免除されることを意味します。
ソフトウェアの使用は、適用されるサイプレス ソフトウェア ライセンス契約によって制限され、かつ制約される場合があります。