PSoC® Creator

システム リファレンスガイド

cy_boot Component v2.40

Document Number: 001-80302 Rev. **

Cypress Semiconductor
198 Champion Court
San Jose、CA 95134-1709
電話 (米国):800.858.1810
Phone (米国外): 408.943.2600
http://www.cypress.com

Copyright © 2005-2012 Cypress Semiconductor Corporation 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 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も一切の責任を負いません。特許又はその他の権限下で、ライセンスを譲渡又は暗示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、又は安全の用途のために仕様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を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ツ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。

PSoC Designer™及び Programmable System-on-Chip™は、Cypress Semiconductor Corp.の商標、PSoC®は同社の登録商標です。本文書で言及するその他全ての商標又は登録商標は各社の所有物です。

全てのソースコード(ソフトウェア及び/又はファームウェア)は Cypress Semiconductor Corporation (以下「サイプレス」)が所有し、全世界(米国及びその他の国)の特許権保護、米国の著作権法並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る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って、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及び/又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複製、使用、変更、そして作成するためのライセンス、並びにサイプレスのソースコード及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、又は表示することは全て禁止されます。

免責条項:サイプレスは、明示的又は黙示的を問わず、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、商品性又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又は回路を適用又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ツ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。

目次

1 序論 5
表記 6
参考資料 6
修正履歴 6

2 標準的な型と定義 7
基本型 7
ハードウェア レジスタの型 7
コンパイラ定義 7
Keil 8051 互換の定義 8
リターンコード 8
インタラプト型とマクロ 9
CPU 固有定義 9
デバイス バージョン 定義 9

3 クロッキング 10
PSoC Creator クロッキングの実装 10
API 19

4 パワーマネージメント 29
クロックのコンフィグレーション 29
ウェイクアップ時間のコンフィグレーション 30
ウェイクアップ ソースのコンフィグレーション 30
API 31
インスタンスの低電力 API 38

5 インタラプト 40
API 40

6 キャッシュ 44
PSoC 3 キャッシュ機能 44
PSoC 5 キャッシュ機能 44

7 Pins 45
API 45

8 レジスタのアクセス 48
API 48
9 DMA 52
10 Flash および EEPROM 53
API 54
11 ブートローダ システム 59
ブートローダ コンポーネント 59
通信コンポーネント 60
カスタムブートホルダコンポーネント 61
ブートローダ プロジェクトのタイプ 63
メモリの使用 65
ブートローダのパラメータ 67
Bootloadable Parameters（ブートロード可能なパラメータ） 69
ブートローダの API 70
ブートローダの コマンド 70
ブートローダのパケット 75
ブートローダの ステータスコードおよびエラーコード 76
ブートローダのアプリケーションとコードデータのファイルフォーマット 76
ブートローダ ホストツール 77
12 システム関数 79
汎用 API 79
CyDelay APIs 80
13 起動とリンク 82
PSoC 3 82
PSoC 5 82
CMSIS サポート (PSoC 5) 83
リセット ステータスの保存 (PSoC 3 と PSoC 5) 83
14 ウォッチドッグ タイマ 84
API 84
15 cy_boot コンポーネントにおける変更 86
バージョン 2.40 86
バージョン 2.30 86
バージョン 2.21 87
バージョン 2.20 88
バージョン 2.10 88
バージョン 2.0 89

1 序論

このシステム リファレンス ガイドでは、PSoC Creator cy_boot コンポーネントにより提供される関数を解説しています。cy_boot コンポーネントは、プロジェクトがチップのリソースにアクセスしやすくなるための、システム機能を提供します。これらの関数はコンポーネントのライブラリの一部ではありませんが、使用することができます。必要なチップ関数を確実に実行するために、関数呼び出しを使用できます。

cy_boot は独特のコンポーネントです:

- 全てのプロジェクトに自動的にインクルードされます
- インスタンスは 1 つしか存在できません
- シンボルは表示されません
- コンポーネントカタログに存在しません（初期設定）

システムコンポーネントとして、cy_boot にはライブラリの機能をいくつか保有しています。このガイドは、以下の機能ごとにまとめられています。

- DMA
- フラッシュ
- クロッキング
- パワーマネージメント
- スタートアップ用コード
- 様々なライブラリ関数
- リンカ スクリプト

cy_boot コンポーネントは、ユーザーファームウェアがこのガイドに記載されたタスクを行えるようにする、API を提供します。複数の主要な機能領域があり、それぞれ個別に解説されています。

表記

以下の表は、このガイド全体で使用されている表記を示しています:

表記法	使用法
Courier New	ファイルのある場所やソースコードを示します: C:\ …cd\icc\, ユーザーが入力したテキスト
イタリック	ファイル名および参照する文書を示します: *sourcefile.hex*
[**括弧書き, 太字**]	手順における、キーボードに入力するコマンドを示します: [**Enter**] または [**Ctrl**] [**C**]
ファイル(File) > 新しいプロジェクト(New Project)	メニューのパスを示します: **ファイル(File) > 新しいプロジェクト(New Project) > 複製(Clone)**
太字	手順における、コマンド、メニューのパス、選択肢、およびアイコン名を示します: **Debugger** アイコンをクリックし、**Next** をクリックします。
灰色のボックス内のテキスト	PSoC Creator または PSoC デバイス固有の注意または警告を示します。

参考資料

このガイドは、PSoC Creator および PSoC デバイスに付随する文書の 1 つです。必要に応じて、下記の他の文書を参照してください。

- PSoC Creator ヘルプ
- PSoC Creator コンポーネントデータシート
- PSoC Creator コンポーネント作成者ガイド
- PSoC 3 および PSoC テクニカルリファレンスマニュアル (TRM)

修正履歴

ドキュメントのタイトル:PSoC® Creator™ System Reference Guide、 cy_boot Component v2.40 **文書番号:001-80302**		
版	**日付**	**変更内容**
**	06/06/2012	これは英語版001-73816 Rev **を翻訳した日本語版Rev. **です。

2 標準的な型と定義

様々な CPU やコンパイラで同じコードが動作できるようサポートするために、cy_boot コンポーネントは、プラットフォーム間で一貫した結果が結果が得られるようにするための型や定義を提供します。

基本型

種類	説明
char8	8 ビット (符号の有無はコンパイラのchar型の選択に依存)
uint8	8ビット 符号なし
uint16	16ビット 符号なし
uint32	32ビット 符号なし
int8	8ビット 符号付き
int16	16ビット 符号付き
int32	32ビット 符号付き

ハードウェア レジスタの型

ハードウェア レジスタは通常副作用があるので、揮発性(volatile)タイプとして参照されます。

定義	説明
reg8	volatile 8ビット 符号なし
reg16	volatile 16ビット 符号なし
reg32	volatile 32ビット 符号なし

コンパイラ定義

使用されているコンパイラは、特定のコンパイラの定義をテストすることで判断できます。たとえば、PSoC 3 Keil コンパイラをテストするには:

```
#if defined(__C51__)
```

定義	説明
__C51__	Keil 8051 コンパイラ

定義	説明
__GNUC__	ARM GCC コンパイラ
__ARMCC_VERSION	Keil MDK および RVDS ツールセットにより使用される、ARM Realview コンパイラ

Keil 8051 互換の定義

Keil 8051 コンパイラは、このプラットフォーム固有の型修飾子をサポートします。他のプラットフォームにおいては、これらの修飾子は存在してはいけません。互換性のために、これらの型は、Keil でコンパイルされた場合は適切な文字列に、他のプラットフォームでコンパイルされた場合は空文字列にマッピングする、定義としてサポートされています。これらの定義は、最適化された Keil 8051 コードを生成するために使用され、その一方で他のプラットフォームでコンパイルを可能にするためにも使用されます。

定義	Keil 型	その他のプラットフォーム
CYBDTA	bdata	
CYBIT	bit	uint8
CYCODE	code	
CYCOMPACT	compact	
CYDATA	データ	
CYFAR	far	
CYIDATA	idata	
CYLARGE	large	
CYPDATA	pdata	
CYREENTRANT	reentrant	
CYSMALL	small	
CYXDATA	xdata	

リターンコード

Cypress ルーチンのリターンコードは、8 ビット符号なし数値型である、cystatus 型として返されます。標準的な返り値:

定義	説明
CYRET_SUCCESS	成功
CYRET_UNKNOWN	理由不明の失敗
CYRET_BAD_PARAM	1つ以上の無効なパラメータ
CYRET_INVALID_OBJECT	無効なオブジェクトの指定
CYRET_MEMORY	メモリ関連の失敗
CYRET_LOCKED	リソースのロックによる失敗
CYRET_EMPTY	オブジェクトが存在しない
CYRET_BAD_DATA	正しくないデータが受信された (CRCもしくは他のエラーチェック)

定義	説明
CYRET_STARTED	動作開始。ただし、まだ完了しているとは限らない
CYRET_FINISHED	動作完了
CYRET_CANCELED	動作中止
CYRET_TIMEOUT	操作のタイムアウト
CYRET_INVALID_STATE	動作は開始していない、または不適切な状態にあります

インタラプト型とマクロ

型とマクロは、複数のコンパイラおよびプログラムにおいて、一貫したインタラプトサービスルーチンの定義を提供します。使用するマクロは関数の定義と関数のプロトタイプでは異なることにご注意ください。

関数定義例:

```
CY_ISR(MyISR)
{
      /* ISRコードが入る */
}
```

関数定義例:

```
CY_ISR_PROTO(MyISR);
```

インタラプトベクタのアドレス型

種類	説明
cyisraddress	インタラプトベクタ(ISR関数のアドレス)

CPU 固有定義

定義	説明
CY_NOP	プロセッサ NOP インストラクション

デバイス バージョン 定義

定義	説明
CY_PSOC3	PSoC 3 デバイス全て
CY_PSOC5	PSoC 5 デバイス全て
CY_PSOC3_ES3	PSoC 3 ES3 以降
CY_PSOC3_ES2	PSoC 3 ES2

3 クロッキング

PSoC Creator クロッキングの実装

PSoC Creator によりサポートされる PSoC デバイスは、柔軟なクロッキング機能を保有しています。これらのクロッキング機能は PSoC Creator において、Design-Wide Resources（設計全体リソース）設定内の選択や設計回路図のクロック信号の接続によって、またランタイム中にクロッキングを変更することができる API 呼び出しによって制御されます。

このセクションでは、PSoC Creator がデバイスにクロック信号をマッピングする方法を解説し、PSoC アーキテクチャに最適化されたクロッキング法のガイドを提供します。

クロックの接続性

PSoC アーキテクチャには、柔軟なクロック生成ロジックが含まれています。各デバイスにおける全てのクロッキング ソースの詳細な説明については、*テクニカルリファレンスマニュアル*を参照してください。これらの多様なクロッキング ソースの使用は、これらのクロックがデザインの要素にどのように接続されているかにより分類することができます。

BUS_CLK

これは特別なクロックです。MASTER_CLK に密接に関連してます。多くのデザインでは、MASTER_CLK と BUS_CLK は同じ周波数であり、同じクロックとみなされます。これらは、システムで最も周波数の高いクロックである必要があります。CPU は BUS_CLK をもとに動作し、全てのペリフェラルは、CPU および DMA に BUS_CLK を使用して通信します。クロックが同期される場合、MASTER_CLK に同期されます。ピンが同期化されると、BUS_CLK に同期化されます。

グローバルクロック

これは、グローバル 低スキュー デジタルクロックラインの一つに置かれるクロックです。これには、BUS_CLK も含まれます。クロックがクロックコンポーネントにより生成される場合、これはグローバルクロックとして生成されます。このクロックはクロック入力に直接接続されるか、反転してからクロック入力に接続される必要があります。グローバルクロック ラインは、PSoC のデジタル素子のクロック入力にのみ接続します。グローバルクロックラインがクロック入力以外(組み合わせ論理回路や、ピン）に接続されている場合、クロック信号は低スキュークロック ラインを使用して送信されません。

ルートクロック(Routed Clock)

グローバルクロックでない全てのクロックは、ルートクロックです。これは論理で発生するクロック（シングルインバーター以外）およびピンから由来するクロックを含みます。

クロックの同期

PSoC デバイス内の各クロックは、同期もしくは非同期です。BUS_CLK および MASTER_CLK が基準となります。PSoC は、同期システムとして動作するよう設計されてます。これはプログラマブル ロジックおよび CPU または DMA との間の通信を可能にするために行われました。これらが共通のクロックと同期していない場合、通信にクロッキング横断

(clocking crossing)回路を必要とします。非同期クロッキングは、CPU システムとやり取りしない PLD ロジックを除いて、一般的にサポートされていません。

同期クロック

同期クロック例としては次のようなものがあります:

- sync to MASTER_CLK オプションが設定されている、グローバルクロック。このオプションは[Clock component Configure(クロックコンポーネントの構成)]ダイアログの **Advanced(詳細)**タブにおいて初期設定として設定されています。
- [入力の同期(Input Synchronized)]オプションが選択された、入力ピンからのクロック。このオプションは、[Pins component Configure(ピンコンポーネントの構成)]ダイアログの **Input(入力)**タブにおいて、初期設定として設定されています。
- 同期されたクロック信号によってクロックされているレジスタからの信号の組み合わせロジックから供給されるクロック。

非同期クロック

非同期クロックは、同期していないクロックです。非同期クロックの例:

- 同期したピン以外以外のデジタルシステムインターコネクト(DSI)から送られる、全ての信号。これらの信号は、タイミングが保証されていないため、非同期とみなす必要があります。これには、以下のものが含まれます。
  - 通常、グローバルクロックとして扱われる(クロック入力に直結されていれば)が、クロックとして使用される前に論理回路を通って供給されるもの
  - 固定機能ブロック(カウンタ、タイマー、PWM)の出力
  - アナログブロックからの2値化信号
- 同期オプションが設定されていない、グローバルクロック
- [Input Synchronized(同期入力)]が選択されていない、入力ピンからのクロック
- 非同期信号と組み合わせて生成されたクロック

信号の同期化

クロック信号源ごとに異なる方法で、同期化することができます:

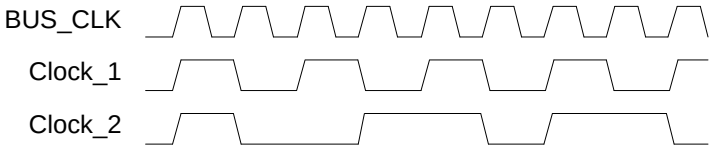
- [Clock component Configure(クロック コンポーネント構成)]ダイアログの **Sync with MASTER_CLK (MASTER_CLK と同期化)**オプションをチェックして、非同期なグローバル クロックを同期化することができます。
- ピンから入力されるルートクロック(Routed Clock)を同期化するには、[Pins component Configure(ピンコンポーネント構成)]ダイアログの **Input Synchronized(入力と同期)**オプションをチェックします(デフォルトでチェックされており、**[ピン]**タブ下に存在します)。
- どの信号も、Sync コンポーネントを使い、クロックシグナルと同期しているクロックを使用することで同期化することができます。

信号を同期化する場合:

- 同期化する信号に対し、同期元のクロックは 2 倍以上の周波数である必要があります。この規則に違反すると、入力するクロックのエッジを取りそこなうことがあり、同期化クロックの出力に反映されません。
- クロック信号の出力の遷移は、全て同期用クロックの立ち上がりエッジに起こります。
- クロック信号の出力のエッジは、元の(同期前のクロック)エッジタイミングからからずれます。

- クロック入力と同期元のクロックが直接関連していない場合、クロック信号の出力において High および Low のパルス幅に幅にばらつきが生じます。

以下の例は BUS_CLK に同期化された 2 つのクロックを示しています。Clock_1 の周期は、BUS_CLK のちょうど 2 倍です。Clock_2の周期は、BUS_CLKの約3倍です。この場合、ハイパルスおよびローパルスの幅が、1もしくは2BUS_CLK 周期の間で変化しています。どちらの場合においても、すべての遷移は BUS_CLK の立ち上がりエッジに起きます。



ルート（Routed）クロックの実装

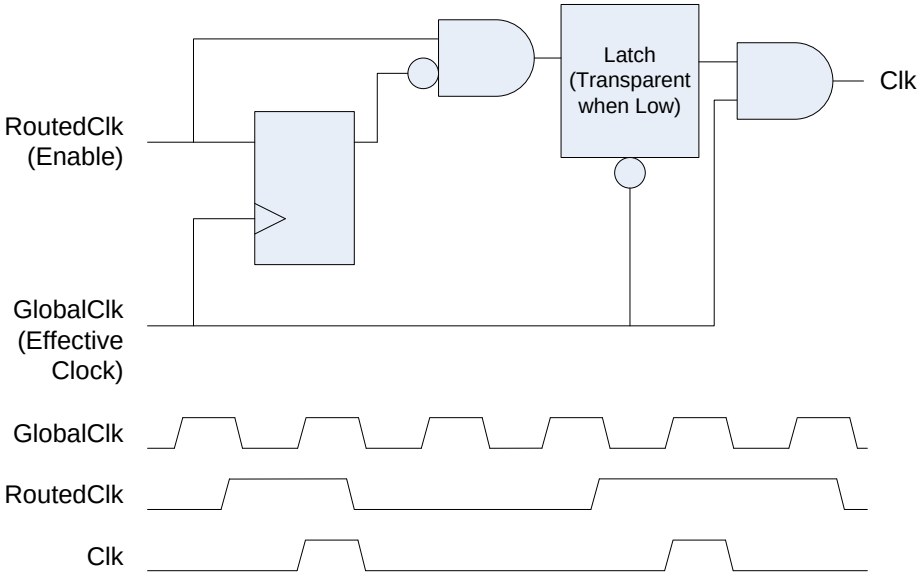
PSoC におけるクロックの実装では、グローバルクロックを、クロックを使用するデジタルロジックのクロック入力に直接接続します。これは、同期クロックと非同期クロックの双方に当てはまります。グローバルクロックは低スキュー クロックラインに配置されているため、同じグローバルクロックに接続されている、クロックを使用する要素は同時にクロックされます。

ルートクロックは、汎用デジタル配線網を通して分配されます。このためクロックは、各対象に対して異なる時刻に到着します。このクロック信号がクロックとして直接使用される場合、このクロックは強制的に非同期クロックとして扱われます。これは、遷移が BUS_CLK の立ち上がりエッジに合わせて起こることが保証されないからです。早く到着したクロックによりクロックされたレジスタ出力が、これよりも遅く到着した同一のクロックによってクロックされたレジスタで使用された場合、回路の異常動作を引き起こすことがあります。

いくつかの状況下では、PSoC Creator は、ルート（Routed）クロックを使用した回路を、グローバルクロックを利用したものに変換できます。ルート（Routed）クロックの全てのソースが、共通のグローバルクロックによりクロックされているレジスタの出力に辿れる場合、PSoC Creator はこの回路を自動的に変換します。これが可能なのは、次のような場合です：

- 全ての信号が、同じグローバルクロックから派生している時。このグローバルクロックは、非同期でも同期でも構いません。
- 全ての信号が、同期している複数のグローバルクロックから派生している時。この場合、共通のグローバルクロックが BUS_CLK です。

PSoC のクロッキングの実装は、この変換に利用される、組み込まれた(built-in)エッジ検出回路を含みます。この実装には、PLD リソースを必要としません。以下に、実装されたロジックと、得られるクロックのタイミング図を示します。



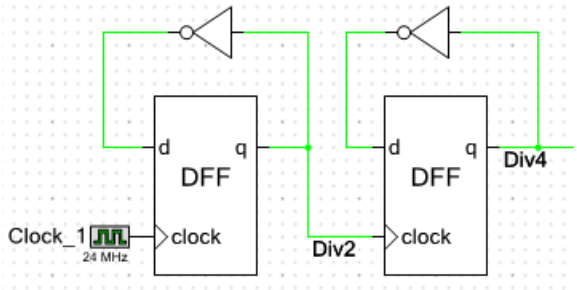
図にあるように、得られるクロックは、ルート（Routed）クロックの立ち上がり後の、最初のグローバルクロックと同期したクロックです。

ルート（Routed）クロックのソースをデザインの中から探す際に、変換された別のルート（Routed）クロックを発見できるかもしれません。この場合、その変換に使用されたグローバルクロックが、その信号のクロックソースになります。

ルート（Routed）クロックに対して行われた全てのクロック変換は、レポートファイルに記載されます。このファイルは、ビルドが成功した場合、Workspace Explorer 下の **Result（結果）**タブに存在します。詳細は、[Initial Mapping（初期マッピング）]の見出しの下に表示されます。各ルート（Routed）クロックは、"Effective Clock（実効クロック）"と"Enable Signal（イネーブル信号）"とともに表示されます。"実効クロック"は使用されているグローバルクロックであり、"イネーブル信号"はエッジ検出と、実効クロックのイネーブルとして使用されたルート（Routed）クロックです。

クロック分周を用いた例

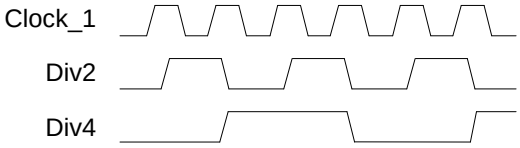
単純なクロック分周回路を用いて、変換がどのように行われるかを見ることができます。下図の回路では、最初のフリップフロップ(cydff_1)のクロックは、グローバルクロックです。これは、周波数が半分のクロックを生成します。この信号は、次のフリップフロップ(cydff_2)のクロックとなる、ルート（Routed）クロックとして使用されます。



レポートファイルは、1 つのグローバルクロックが使用され、1 つのルート（Routed）クロックが、グローバルクロックを実効クロックとして変換されたことを示します。

```
<CYPRESSTAG name="Tech mapping">
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
<CYPRESSTAG name="Global Clock Selection" icon="FILE_RPT_TECHM">
    Digital Clock 0: Automatic-assigning  clock 'Clock_1'. Fanout=1, Signal=tmp__cydff_1_clk
</CYPRESSTAG>
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: tmp__cydff_1_reg:macrocell.q
        Effective Clock: Clock_1
        Enable Signal: tmp__cydff_1_reg:macrocell.q
</CYPRESSTAG>
```

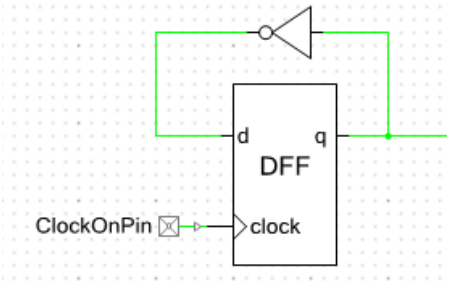
この回路により生成される信号は、下図のようになります。



Div4 が、Div2 信号の立ち下がりエッジにより生成されたように見えるかもしれません。しかし、これは違います。Div4 信号は、Div2 信号の立ち上がりエッジの後の、最初の Clock_1 立ち上がりエッジで生成されます。

ピンからのクロックの例

以下の回路では、クロックは、同期機能を ON にしたピンから得ています。ピンは既に BUS_CLK と同期済みなので変換された回路では実効クロックとして BUS_CLK を使用し、ピンの立ち上がりエッジをイネーブル信号として使用します。



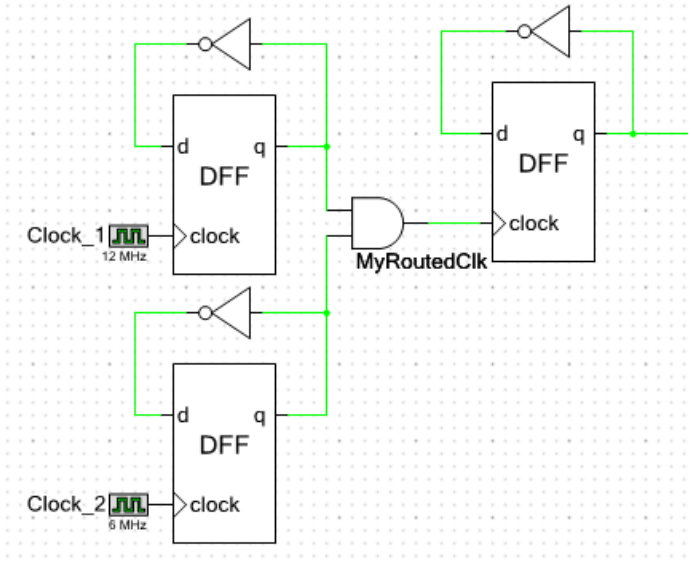

```
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
{Global Clock Selection}
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: ClockOnPin(0):iocell.fb
        Effective Clock: BUS_CLK
        Enable Signal: ClockOnPin(0):iocell.fb
</CYPRESSTAG>
```

ピンで入力同期が無効の場合、ルート（Routed）クロックを変換するグローバルクロックが存在しないため、ルート（Routed）クロックが直接使用されます。

```
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
<CYPRESSTAG name="Global Clock Selection" icon="FILE_RPT_TECHM">
</CYPRESSTAG>
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: ClockOnPin(0):iocell.fb
        Effective Clock: ClockOnPin(0):iocell.fb
        Enable Signal: True
</CYPRESSTAG>
```

複数のクロックソースを使用した例

この例では、ルート（Routed）クロックは、2 つの異なるクロックにより駆動するフリップフロップから生成されます。2 つのクロックは同期しているので、共通のグローバルクロックの BUS_CLK が実効クロックとなります。

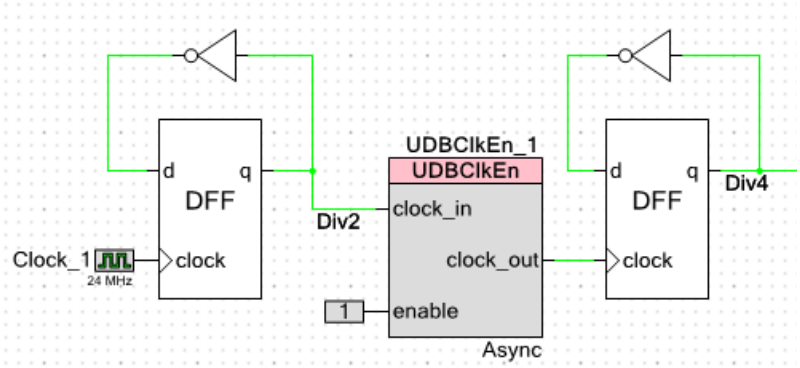



```
<CYPRESSTAG name="Tech mapping">
<CYPRESSTAG name="Initial Mapping" icon="FILE_RPT_TECHM">
<CYPRESSTAG name="Global Clock Selection" icon="FILE_RPT_TECHM">
    Digital Clock 0: Automatic-assigning  clock 'Clock_1'. Fanout=1, Signal=tmp__cydff_1_clk
    Digital Clock 1: Automatic-assigning  clock 'Clock_2'. Fanout=1, Signal=tmp__cydff_2_clk
</CYPRESSTAG>
<CYPRESSTAG name="UDB Routed Clock Assignment">
    Routed Clock: MyRoutedClk:macrocell.q
        Effective Clock: BUS_CLK
        Enable Signal: MyRoutedClk:macrocell.q
</CYPRESSTAG>
<CYPRESSTAG name="UDB Clock/Enable Remapping Results">
</CYPRESSTAG>
```

いずれかのクロックが非同期であった場合、ルート（Routed）クロックが直接使用されます。

ルート(Routed)クロック変換のオーバーライド

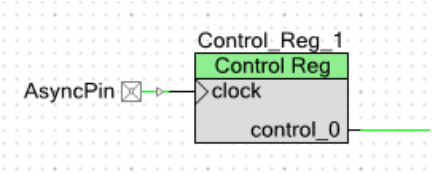
PSoC Creator がルート(Routed)クロックに対して行う自動変換は、通常使用されるべき実装です。しかし、ルート(Routed)クロックを強制的に使用させる方法が存在します。非同期モードにコンフィギュレーションした UDBClkEn コンポーネントは、以下の回路のようにルート(Routed)クロックの使用を強制します。

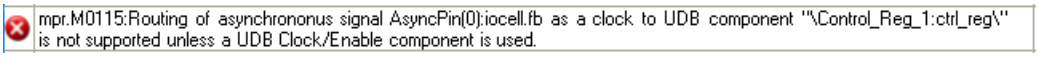


非同期クロックの使用

非同期クロックは、PLD ロジックとともに使用できます。しかし、CPU とのやりとりがある、コントロールレジスタ、ステータスレジスタ、およびデータパス要素は、自動的にサポートされません。多くの Cypress ライブラリコンポーネントは、同期クロックでしか動作しません。これらは、提供されるクロックが非同期の場合、自動的にシンクロナイザの挿入を要求します。SPI スレーブなどの、非同期クロックに駆動されるよう設計されたコンポーネントは、データシート内においてクロッキングの取り扱いを記載しています。

非同期クロックが PLD ロジック以外のものに直接接続されていた場合、デザインルールチェック(DRC)エラーが生成されます。例えば、非同期ピンがコントロールレジスタクロックに接続されていた場合、DRC エラーが生成されます。





エラーメッセージに記載されているように、エラーは非同期モードにて UDBClkEn コンポーネントを使用することで取り除けます。これは大元となる同期の問題を解決しませんが、デザインにおいて何らかの方法で同期の問題を解決した場合には、エラーを無視することができます。

クロックの交差(Clock Crossing)

デザイン内に複数のクロックドメインが必要になることが多くあります。多くの場合、これら複数のドメイン間にやりとりがないので、クロックの交差は起きません。一つのクロックドメインで生成された信号が、別のクロックドメインで使用される必要のある場合、特別な注意が必要です。2 つのクロックドメインが互いに非同期の場合と、2 つとも BUS_CLK に同期している場合があります。

2 つのクロックが BUS_CLK に同期している場合、遅いクロックドメインの信号は、もう一つのドメインで自由に使用できます。逆の場合、速いクロックドメインの信号が十分長い間アクティブであり、遅いクロックドメインでサンプルされるよう注意する必要があります。双方向ともに、タイミングの制約は、いずれかのクロックの速さではなく、BUS_CLK の速さによります。

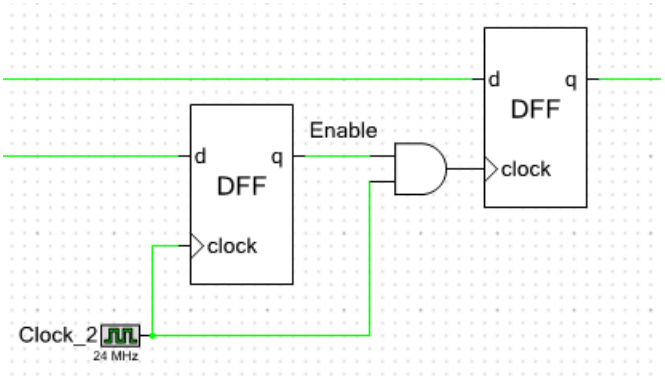
クロックドメイン間で保証されていることは、エッジが BUS_CLK の立ち上がりエッジで起きることだけです。このため、2 つのクロックドメインの立ち上がりエッジは、最小で 1 BUS_CLK 離れていることがあります。これは、あるクロックドメインがもう 1 つの整数倍のときも成立します。なぜなら、クロックデバイダがアラインされている必要がないからです。2 つのクロックドメイン間で組み合わせロジックが存在する場合、BUS_CLK 周波数を制限しないために、フリップフロップを挿入する必要があるかもしれません。フリップフロップの挿入により、1 つのクロックドメインからもう 1 つへの交差は、フリップフロップからフリップフロップに直接接続されたパスになります。

クロックドメインが互いに無関係の場合、クロックドメイン間でシンクロナイザを使う必要があります。同期化機能を実装するために、Sync コンポーネントを使用できます。このクロックには、到着先のクロックドメインのものを使用します。

Sync コンポーネントは、ダブルシンクロナイザを実装する、ステータスレジスタの特殊なモードを使用して実装されます。入力信号は、サンプリング用クロックの周期を越えるパルス幅を持つことが必要です。シンクロナイザを通過するときの実際の遅延の長さは、入力信号と同期用クロックのアライメントにより異なります。この長さは、1 クロック周期より僅かに長い時間から、2 クロック周期より僅かに長い時間の間です。複数の信号が同期化されている場合、2 つの信号がシンクロナイザに入るときと同じ 2 つの信号が出力されるときの時差は、どれがシンクロナイザーにより正しくサンプルされたかにより、最大 1 周期変化することがあります。

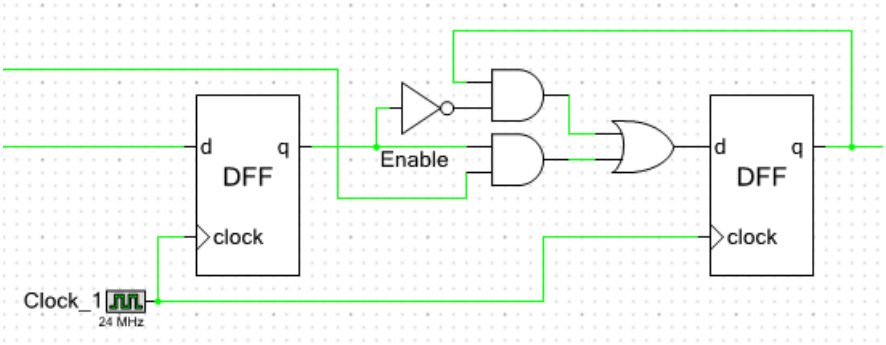
ゲートされたクロック

グローバルクロックは、回路を直接クロッキングする以外の用途で使用すべきではありません。グローバルクロックがロジックファンクションに使用された場合、信号はタイミングの保証がない、まったく異なるパス経路を通ります。タイミング分析を行うことができないため、次のような回路は回避する必要があります。



この回路は、ルート（Routed）クロックを用いて実装され、タイミング分析をサポートせず、クロックがイネーブルやディセーブルされる際に、クロック信号上にグリッジを発生しがちです。

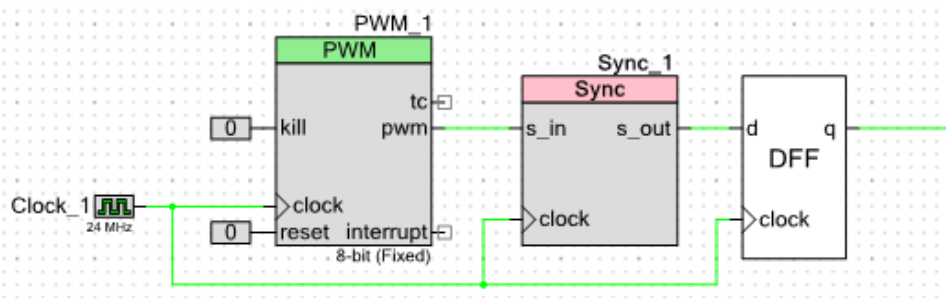
次の回路は上と等価の機能ですが、タイミング分析をサポートし、グローバルクロックのみを使用し、かつ信頼性に問題がありません。この回路はクロックをゲートしませんが、そのかわり、論理的に、新しいデータへの切替をイネーブルするか、現在のデータを保持させます。



クロックへのアクセスが必要な場合、たとえばピンに送信するクロックを生成する場合には、トグルフリップフロップをクロックするために、2 倍のクロックを使用すべきです。そのフリップフロップの出力に対し、必要なタイミング分析を行うことができます。

固定ファンクションのクロッキング

回路図上では、固定ファンクションのペリフェラルとUDB ベースのペリフェラルに送信されるクロック信号は、同じクロックに見えます。しかし、異なるペリフェラルのタイプに到着する時点の、クロック信号間のタイミング関係は保証されません。さらに、データ信号の配線上の遅延も保証されません。このため、固定ファンクションのペリフェラルが UDB アレイの信号に接続されている場合、信号は以下の例のように同期してある必要があります。固定ファンクションのペリフェラルから来る信号について、タイミングに関する仮定をしてはいけません。



API

uint8 CyPLL_OUT_Start(uint8 wait)

説明: PLLを有効化します。オプションにより、安定するまでウェイトします。最低250 マイクロ秒、もしくはPLLが安定したことを検知するまでウェイトします。

パラメータ: wait:

- 0: 設定直後に戻る
- 1: PLLロックまたはタイムアウトまでウェイト

返り値: ステータス

- CYRET_SUCCESS - 成功し完了
- CYRET_TIMEOUT - クロックの安定を検知する前にタイムアウト クロックの入力ソースにジッタが多い場合、ロックの検知ができないことがあります。しかし、タイムアウト終了後には、生成されたPLLクロックを使用することができます。

副作用と制限: wait が有効の場合:
高速タイムホイールを用いてウェイト時間を計測します。高速タイムホイールの他の用途における使用は、この関数の実行中は停止され、その後復元されます。
100KHz ILOを使用します。イネーブルされていない場合、この関数は実行している間、100KHz ILOをイネーブルにします。
この関数が実行している間、インタラプトルーチンによる、ILO、高速タイムホイール、セントラルタイムホイール、および Once Per Second（毎秒 1 回）インタラプトの、セットアップの変更はできません。ILO、セントラルタイムホイール、および Once Per Second（毎秒 1 回）インタラプトの現在の操作は、この関数の操作中も維持されます。ただし、パワーマネジャーインタラプトステータスレジスタ(Power Manager Interrupt Status Register)の読み込みが、CyPMReadStatus() 関数の使用以外の方法で行われない場合に限ります。

void CyPLL_OUT_Stop()

説明: PLLを無効化します

パラメータ: なし

返り値: なし

void CyPLL_OUT_SetPQ(uint8 P, uint8 Q, uint8 current)

説明: P分周器、Q分周器、およびチャージポンプ電流を設定します。出力周波数は、P/Q * 入力周波数 になります。この関数を呼び出す前に PLL をディスエーブルすることが必要です。

パラメータ: P: 有効範囲 [4 - 255]

Q: 有効範囲 [1 - 16] 入力周波数 / Q は、1MHz から 3MHz までの範囲に入る必要があります。

current: 有効範囲 [1 - 7] チャージポンプ電流(uA単位)。67 MHz 以下の出力周波数には 2uA が推奨され、より高い周波数には 3 uA が推奨されます。

返り値: なし

void CyPLL_OUT_SetSource(uint8 source)

説明: PLLの入力クロックソースを設定します。この関数を呼び出す前に PLL をディスエーブルすることが必要です。

パラメータ: source: 3つ存在するPLLクロックソースのうちの1つ

値	定義	Source (ソース)
0	CY_PLL_SOURCE_IMO	IMO
1	CY_PLL_SOURCE_XTAL	MHz 水晶振動子
2	CY_PLL_SOURCE_DSI	DSI

返り値: なし

void CyIMO_Start(uint8 wait)

説明: IMOを有効化します。オプションとして安定するまで 6us 以上ウェイトさせることができます。

パラメータ: wait:

- 0: 設定直後に戻る
- 1: IMOが安定するまで最低6usウェイトします。

返り値: なし

副作用と制限: wait が有効の場合:

高速タイムホイールを用いてウェイト時間を計測します。高速タイムホイールの他の用途における使用は、この関数の実行中は停止され、その後復元されます。

100KHz ILOを使用します。イネーブルされていない場合、この関数は実行している間、100KHz ILOをイネーブルにします。

この関数が実行している間、インタラプトルーチンによる、ILO、高速タイムホイール、セントラルタイムホイール、および Once Per Second（毎秒 1 回）インタラプトの、セットアップの変更はできません。ILO、セントラルタイムホイール、および Once Per Second（毎秒 1 回）インタラプトの現在の操作は、この関数の操作中も維持されます。ただし、パワーマネジャーインタラプトステータスレジスタ(Power Manager Interrupt Status Register)の読み込みが、CyPMReadStatus() 関数の使用以外の方法で行われない場合に限ります。

void CyIMO_Stop()

説明: IMOをディスエーブルにします。

パラメータ: なし

返り値: なし

void CyIMO_SetFreq(uint8 freq)

説明: IMOの周波数を設定します。IMOが実行中に、変更を加えることができます。USB設定が選択されると、USB クロック ロッキング回路はイネーブルになります。それ以外の場合では、この回路はディスエーブルになります。USB ブロックは USB 設定を選択する前に電源を入れなくてはなりません。IMOが現在マスタークロックを駆動している場合、この変更を行う前に、CyFlash_SetWaitCycles()を用いて、フラッシュウェイトステートを適切に設定する必要があります。

パラメータ: freq: IMOの動作周波数

値	定義	周波数
0	CY_IMO_FREQ_3MHZ	3 MHz
1	CY_IMO_FREQ_6MHZ	6 MHz
2	CY_IMO_FREQ_12MHZ	12 MHz
3	CY_IMO_FREQ_24MHZ	24 MHz
4	CY_IMO_FREQ_48MHZ	48 MHz
5	CY_IMO_FREQ_62MHZ	62.6 MHz
6	CY_IMO_FREQ_74MHZ	74.7 MHz (PSoC 5)
8	CY_IMO_FREQ_USB	24 MHz (USB動作用にトリミングされている)

返り値: なし

void CyIMO_SetSource(uint8 source)

説明: IMOブロックからのクロック出力のソースを設定します。IMOからの出力のデフォルトは、IMO自身です。オプションで、MHz水晶発振器もしくはDSI入力を、IMO出力のソースにすることが可能です。IMOが現在マスタークロックを駆動している場合、この変更を行う前に、CyFlash_SetWaitCycles()を用いて、フラッシウェイトステートを適切に設定する必要があります。

パラメータ: source: 3つ存在するIMO出力ソースのうちの1つ。

値	定義	Source (ソース)
0	CY_IMO_SOURCE_IMO	IMO
1	CY_IMO_SOURCE_XTAL	MHz 水晶振動子
2	CY_IMO_SOURCE_DSI	DSI

返り値: なし

void CyIMO_EnableDoubler()

説明: IMOダブラをイネーブルにします。2x周波数クロックを、24 MHz 入力を USB ブロックが使用するための48 MHz 出力に変換するために使用されます。

パラメータ: なし

返り値: なし

void CyIMO_DisableDoubler()

説明: IMOダブラをディスエーブルにします。

パラメータ: なし

返り値: なし

void CyMasterClk_SetSource(uint8 source)

説明: マスタクロックのソースを設定します。この関数を呼び出す前に、現在のソースと新しいソースが、共に安定に動作している必要があります。フラッシュウェイトステートは CyFlash_SetWaitCycles() を使用して変更する前に、適切に設定する必要があります。

パラメータ: source: 4つあるマスタークロックのソースの1つです。

値	定義	Source (ソース)
0	CY_MASTER_SOURCE_IMO	IMO
1	CY_MASTER_SOURCE_PLL	PLL
2	CY_MASTER_SOURCE_XTAL	MHz 水晶振動子
3	CY_MASTER_SOURCE_DSI	DSI

返り値: なし

void CyMasterClk_SetDivider(uint8 divider)

説明: マスタークロックを生成する際に使用する、分周器の値を設定します。フラッシュウェイトステートは CyFlash_SetWaitCycles() を使用して変更する前に、適切に設定する必要があります。

パラメータ: divider: 有効範囲 [0 - 255] クロックは、この値に1を加えた値で分周されます。例えば、2分周するには、このパラメーターを 1 に設定する必要があります。

返り値: なし

void CyBusClk_SetDivider(uint16 divider)

説明: バスクロックを生成する際に使用する、分周器の値を設定します。フラッシュウェイトステートは CyFlash_SetWaitCycles() を使用して変更する前に、適切に設定する必要があります。

パラメータ: divider: 有効範囲 [0 - 65535] クロックは、この値に1を加えた値で分周されます。例えば、2分周するには、このパラメーターを 1 に設定する必要があります。

返り値: なし

void CyCpuClk_SetDivider(uint8 divider)

説明: CPUクロックを生成する際に使用する、分周器の値を設定します。PSoC 3 のみに適用されます。

パラメータ: divider: 有効範囲 [0 - 15] クロックは、この値に1を加えた値で分周されます。例えば、2分周するには、このパラメーターを 1 に設定する必要があります。

返り値: なし

void CyUsbClk_SetSource(uint8 source)

説明: USBクロックのソースを設定します。

パラメータ: source: 4つあるUSBクロックのソースの1つです

値	定義	Source (ソース)
0	CY_USB_SOURCE_IMO2X	IMO 2x
1	CY_USB_SOURCE_IMO	IMO
2	CY_USB_SOURCE_PLL	PLL
3	CY_USB_SOURCE_DSI	DSI

返り値: なし

void CyILO_Start1K()

説明: ILO 1 KHz 発振器をイネーブルにします。

注 ILO 1 KHz 発振器はクロック エディターの選択に関わらず、初期設定において常にイネーブルになっています。そのため、この API は発振器が手動でオフにされたときのみ必要です。

パラメータ: なし

返り値: なし

void CyILO_Stop1K()

説明: ILO 1 KHz 発振器をディスエーブルにします。

注 ILO 1 KHz 発振器はスリープまたはハイバネート低電力モード API の使用が予想されている場合、イネーブルにする必要があります。詳細はこの文書のパワーマネージメントセクションを参照してください。

パラメータ: なし

返り値: なし

void CyILO_Start100K()

説明: ILO 100 KHz 発振器をイネーブルにします。

パラメータ: なし

返り値: なし

void CyILO_Stop100K()

説明: ILO 100 KHz 発振器をディスエーブルにします。

パラメータ: なし

返り値: なし

void CyILO_Enable33K()

説明: ILO 33 KHz 分周器をイネーブルにします。**注** 33 KHzクロックは100 KHz発振器から生成されるため、33 KHz出力を生成するには100 KHz発振器も動作している必要があります。

パラメータ: なし

返り値: なし

void CyILO_Disable33K()

説明: ILO 33 KHz 分周器をディスエーブルにします。33 KHzクロックは100 KHz発振器から生成されるものですが、このAPIは100 KHzクロックをディスエーブルにしないことに注意してください。

パラメータ: なし

返り値: なし

void CyILO_SetSource(uint8 source)

説明: ILOブロックからのクロック出力のソースを設定します。

パラメータ: source: 3つ存在するILO出力ソースのうちの1つ

値	定義	Source (ソース)
0	CY_ILO_SOURCE_100K	ILO 100 KHz
1	CY_ILO_SOURCE_33K	ILO 33 KHz
2	CY_ILO_SOURCE_1K	ILO 1 KHz

返り値: なし

uint8 CyILO_SetPowerMode(uint8 mode)

説明: パワーダウン時にILOが使用する電源モードを設定します。パワーダウン時の電力消費を減少させることが可能になりますが、この場合スタートアップ時間が長くなります。

パラメータ: mode:

値	定義	説明
0	CY_ILO_FAST_START	高速スタートアップ、パワーダウン時に内部バイアスがオン
1	CY_ILO_SLOW_START	低速スタートアップ、パワーダウン時に内部バイアスがオフ

返り値: 以前のパワーモード

void CyXTAL_32KHZ_Start()

説明: 32 KHz 水晶 発振器をイネーブルにします。

パラメータ: なし

返り値: なし

void CyXTAL_32KHZ_Stop()

説明: 32 KHz 水晶 発振器をディスエーブルします。

パラメータ: なし

返り値: なし

uint8 CyXTAL_32KHZ_ReadStatus()

説明: 32 KHz水晶発振器の2つのステータスビットを読み込みます。

パラメータ: なし

返り値: ステータス

値	定義	Source (ソース)
0x20	CY_XTAL32K_ANA_STAT	アナログ計測 1: 安定 0: 不安定
0x10	CY_XTAL32K_DIG_STAT	デジタル計測 (計測を行うためには 33 KHz ILOが必要) 1: 安定 0: 不安定

uint8 CyXTAL_32KHZ_SetPowerMode(uint8 mode)

説明: スリープモード時における、32 KHz水晶発振器のパワーモードを設定します。ノイズ源が少ないときに、スリープ中の電力消費を減少させることができます。アクティブモードの間は、発振器は常にハイパワーモードで動作します。

パラメータ: mode:

- 0: ハイパワーモード
- 1: スリープ中はローパワーモード

返り値: 以前のパワーモード

uint8 CyXTAL_Start(uint8 wait)

説明: MHz 水晶発振器をイネーブルにします。XERRビットがLow(エラーなし)になるまで、1ミリ秒もしくはwaitパラメータに指定されたミリ秒数経過するまでウェイトします。

パラメータ: wait: 有効範囲 [0 - 255] これは、ミリ秒単位のタイムアウト時間です。適当な値は、水晶発振器によります。

返り値: ステータス
CYRET_SUCCESS - 成功し完了
CYRET_TIMEOUT - XERRにおいて、Low値を検出せず、タイムアウト

副作用と制限: waitがイネーブルな場合(ゼロ以外のwaitの場合):
高速タイムホイールを用いてウェイト時間を計測します。高速タイムホイールの他の用途における使用は、この関数の実行中は停止され、その後復元されます。
100KHz ILOを使用します。イネーブルされていない場合、この関数は実行している間、100KHz ILOをイネーブルにします。
この関数が実行している間、インタラプトルーチンによる、ILO、高速タイムホイール、セントラルタイムホイール、および Once Per Second（毎秒 1 回）インタラプトの、セットアップの変更はできません。ILO、セントラルタイムホイール、および Once Per Second（毎秒 1 回）インタラプトの現在の操作は、この関数の操作中も維持されます。ただし、パワーマネジャーインタラプトステータスレジスタ(Power Manager Interrupt Status Register)の読み込みが、CyPMReadStatus() 関数の使用以外の方法で行われない場合に限ります。

void CyXTAL_Stop()

説明: メガヘルツ水晶発振器をディスエーブルにします。

パラメータ: なし

返り値: なし

void CyXTAL_EnableErrStatus()

説明: メガヘルツ水晶発振器のXERRステータスビットの生成をイネーブルにします。

パラメータ: なし

返り値: なし

void CyXTAL_DisableErrStatus()

説明: メガヘルツ水晶発振器のXERRステータスビットの生成をディスエーブルにします。

パラメータ: なし

返り値: なし

uint8 CyXTAL_ReadStatus()

説明: メガヘルツ水晶発振器のXERRステータスビットを読み込みます。このステータスビットはスティッキーで値読み出し時クリア(clear on read)です。

パラメータ: なし

返り値: ステータス:0: エラーなし、1:エラー

void CyXTAL_EnableFaultRecovery()

説明: 障害回復回路をイネーブルにします。この回路は、メガヘルツ水晶発振器に障害が出た際に、IMOに切り替えます。呼び出し直後の障害検出と切り替えを防ぐために、この関数を呼び出す前に、水晶発振器が安定動作しており、XERRビットが0である必要があります。

パラメータ: なし

返り値: なし

void CyXTAL_DisableFaultRecovery()

説明: メガヘルツ水晶発振器に障害が出た際に、IMOに切り替える、障害回復回路をディスエーブルにします。

パラメータ: なし

返り値: なし

void CyXTAL_SetStartup(uint8 setting)

説明: 水晶発振器のスタートアップ設定を設定します。

パラメータ: 設定:有効範囲 [0 - 31] 値は、使用されている水晶発振器の周波数と質に依存します。各水晶発振器における適切な値については、TRMを参照してください。

返り値: なし

void CyXTAL_SetFbVoltage(uint8 setting)

説明: PSoC 3 ES3 デバイス専用です。この関数は水晶回路で使用するフィードバックリファレンス電圧を設定します。

パラメータ: 設定:有効範囲 [0 - 15] 各電圧と設定値のマッピングの詳細については、TRMを参照してください。

返り値: なし

副作用と制限: フィードバック参照電圧は、ウォッチドッグ参照電圧より高い必要があります。

void CyXTAL_SetWdVoltage(uint8 setting)

説明: PSoC 3 ES3 デバイス専用です。この関数は水晶回路の故障を検出するためにウォッチドッグが使用するリファレンス電圧を設定します。

パラメータ: 設定：有効範囲 [0 - 7] 各電圧と設定値のマッピングの詳細については、TRMを参照してください。

返り値: なし

副作用と制限: フィードバック参照電圧は、ウォッチドッグ参照電圧より高い必要があります。

4 パワーマネージメント

電力消費および利用可能なリソース量を制御するために、PSoC デバイスがサポートするさまざまな電力モードがあります。PSoC 3 および PSoC 5 デバイスのどちらも、次の電力モードをサポートします（電力消費の降順）: アクティブ、代替アクティブ、スリープおよびハイバネート。

注 PSoC 5 はデバッガが実行中、低電力モードに入りません。

注 PSoC 3 および PSoC 5 において、パワー マネージャはシステム パフォーマンス コントローラー (SPC) がコマンドを実行している際、デバイスを低電力状態にしません。デバイスは SPC がコマンドの実行を完了してから、低電力モードに入ります。

スリープおよびハイバネート モードに入る前に、IMO 値は 12 MHz である必要があります。特定な低電力モードに入る直前に、IMO 周波数は 12 MHz に設定されます（フラッシュウェイトサイクル数を修正しない状態で）。IMO 周波数はウェイクアップ時に直ちに復元されます。マスクされている場合でも、デバイスを低電力モードに入れる前に、すべての保留中のインタラプトをクリアする必要があります。

クロックのコンフィグレーション

低電力モードエントリとウェイクアップを適切に行うには、いくつかのデバイス構成要件があります。

- スリープおよびハイバネート モードに入る前に、クロック システムを準備し、期待する通りにアクティブ モードと低電力モード間を切り替えることができるようにします。
- CyPmSaveClocks() および CyPmRestoreClocks() 関数は、それぞれ低電力モードに入る前、そしてアクティブ モードにウェイクアップした後で、クロックの構成を準備する責任があります。一般的に、CyPmSaveClocks() は構成を保存し、低電力モード エントリへの要件を設定します。CyPmRestoreClocks() はクロックの構成を元の状態に復元します。
- IMO はマスター クロックのソースになるために必要です。そのため、IMO クロック値は Design-Wide Resources System Editor（設計全体リソース システム エディタ）の「Enable Fast IMO During Startup（起動時に高速 IMO をイネーブルにする）」に対応するよう設定されます。このオプションがイネーブルの場合、IMO クロック周波数は 48 MHz に設定され、その他の場合は 12 MHz に設定されます。

  注 IMO クロック周波数は PSoC 5 では必ず 12 MHz に設定されます。PLL と MHz ECO は、IMO がマスター クロックのソースの場合、オフになります。
- バスとマスター クロックの分周器は分周比 1 に設定され、新しいフラッシュウェイトサイクルの値は、CPU 周波数の新しい値に一致するよう設定されます。詳細は CyFlash_SetWaitCycles() 関数の説明を参照してください。

スリープとハイバネートの低電力モードが適切に作動するためには、すべてのデバイスに対して 1 KHz ILO をイネーブルにする必要があります。PSoC 3 ES3 デバイスでは 1 KHz ILO を使用してリセット後のハイバネート/スリープレギュレーターセト

リング時間を測定します。この間、システムはこれらのモードに入る要求を無視します。ホールドオフ遅延は 1 kHz ILO の立ち上がりエッジを使用して測定されます。ターミナルカウントは、スリープ レギュレーター トリム レジスタによりセットされます。

注意 このレジスタは修正しないでください。

PSoC 5 デバイスでは、CPU がウェイクアップするためにインタラプトが必要です。パワーマネージメントの実装は、ウェイクアップ時間が別のコンポーネントによって構成されていることを前提とし（コンポーネントベースのウェイクアップ時間コンフィグレーション）、これによりインタラプトはターミナルカウントで発行されます。詳細は、次のセクションを参照してください。

ウェイクアップ時間のコンフィグレーション

デバイスを低電力モードからウェイクアップすることが出来るタイマーは 3 つあります。セントラル タイム ホイール (CTW)、高速タイマー ホイール (FTW) および one pulse per second（毎秒 1 パルス、One PPS）。これらのタイマーの詳細は、デバイスの TRM およびデータシートを参照してください。

ウェイクアップ時間をコンフィグレーションする方法は 2 つあります。

- パラメータをベースにしたウェイクアップ時間のコンフィグレーションは、希望するパラメータと共に CyPmSleep() と CyPmAltAct() 関数を呼び出して行います。このコンフィグレーション方法は PSoC 3 デバイスに対してのみ使用できます。
- コンポーネント ベースのウェイクアップ時間の構成 CTW ウェイクアップ間隔は、スリープ タイマー コンポーネントと共にコンフィグレーションされます。1 秒間隔は RTC コンポーネントと共にコンフィグレーションされます。

ハイバネート モードではウェイクアップ時間のコンフィグレーションはありません。

最初の CTW および FTW 間隔だけが指定されたものより小さくなることが保証されていることを、念頭においてください。後続する間隔が公称値を持つためには、対応するタイマーは CyPmSleep() と CyPmAltAct() 関数によりイネーブルされ、タイマーはイネーブルのままにしておきます。API によっては、このタイマーを使用することができます。このために、低電力モードエントリの前に、タイマーが常にイネーブル状態になることがあります（タイマーの間隔は対応するタイマーがディスエーブルのときのみ変更することができます）。そのため、ウェイクアップ間隔は必ず予想より小さくなります。

CyPmReadStatus() 関数はインタラプト ステータス ビットをクリアするため、対応するパラメータと共に（例えば、デバイスが CTW でウェイクアップするように構成されている場合、CY_PM_CTW_INT と共に）、ウェイクアップ直後に呼び出されなくてはなりません。

CTW をウェイクアップ タイマーとして使用する場合、ウェイクアップ後に CyPmReadStatus() 関数を必ず呼び出し（ウェイクアップがパラメーターまたはコンポーネントベースの方法で構成されているとき）、CTW インタラプトステータスビットをクリアする必要があります。CTW イベント発生後、1 ミリ秒(ILO の 1 クロックサイクル)以内に この関数を呼び出す必要があります。

ウェイクアップ ソースのコンフィグレーション

PSoC 3 ES3 デバイスでは、代替アクティブおよびスリープ低電力モードからデバイスをウェイクアップするウェイクアップソースをどれにするかコンフィギュレーションすることができます。PSoC 5 と PSoC 3 ES2 シリコンでは、ウェイクアップソースは指定できません。この場合、ウェイクアップソースの引数は無視され、利用可能なウェイクアップソースの全てが、デバイスをウェイクさせることができます。PSoC 5 シリコンでは CTW がスリープからの唯一のサポートされるウェイクアップソースです。

PSoC 3 代替アクティブ モード特有の問題

- インタラプト コントローラーにおいてイネーブルされたかどうかに関わらず、いずれのインタラプトも代替アクティブパワーモードからデバイスをウェイクします。

- エッジ検出器もバイパスされるので、ウェイクアップソースは必ずレベルトリガーされます。
- 直接接続されている DMA インタラプトでは、このモードからウェイクしません。ウェイクアップ状態になるようにするためには、DSI を通してルーティングしなくてはなりません。

API

void CyPmSaveClocks()

説明: この関数は、スリープもしくは休止のために、低電力モードに入る準備の際に呼び出されます。スリープ/ハイバネート中持続しない、あるいはスリープ/ハイバネートの準備のために修正する必要があるクロッキングシステムのすべての状態を保存します。デジタルおよびアナログクロック分周器を全てシャットダウンします。マスタクロックをIMOに変更し、PLLおよびMHz 水晶発振器をシャットダウンします。IMO 周波数はDesign-Wide Resources System Editor（設計全体リソースシステムエディタ）の「Enable Fast IMO During Startup（起動時に高速 IMO をイネーブル」と一致させて、12 MHz または 48 MHz に設定されます。ILOおよび32 KHz発振器には変更を加えません。現在のフラッシュウェイトステートの設定が保存され、その後現在のIMOの速度に合わせて変更されます。

注 マスタークロックソースが DSI 入力を通してルーティングされる場合、CyPmSaveClocks() / CyPmRestoreClocks() 関数を使用する前に、その他のソースに手動で設定する必要があります。

パラメータ: なし

返り値: なし

副作用と制限 すべてのペリフェラル クロックはこの API メソッド呼び出しの後、オフになります。

void CyPmRestoreClocks()

説明: CyPmSaveClocksの最後の呼び出し時に保存された状態を、全て復旧します。フラッシュウェイトステートの設定も復旧します。

注 マスタークロックソースが DSI 入力を通してルーティングされる場合、CyPmSaveClocks() / CyPmRestoreClocks() 関数を使用する前に、その他のソースに手動で設定する必要があります。

PSoC 3 ES3: メガヘルツ水晶がホールドオフタイムアウト後準備ができていないとき、マージ領域は状態を処理するのに使うことができます。

PSoC 5: メガヘルツ クリスタルは安定させるために最高 130 ms となりました。ホールドオフ タイムアウト後、準備が出来ているかの検証は行われません。

パラメータ: なし

返り値: なし

void CyPmAltAct(uint16 wakeupTime, uint16 wakeupSource)

説明: パーツを代替アクティブ(スタンバイ)状態にします。代替アクティブ状態では、デバイスがアクティブのときの機能がすべて使用できますが、関数の動作に、代替アクティブ状態ではCPUがディスエーブルにされていることことが影響します。設定コードとコンポーネントAPIは、代替アクティブ状態のテンプレートが代替アクティブ状態と同じになるように設定します。ただし、この例外として、CPUは代替アクティブ状態においてディスエーブルにされます。

注 この関数を呼び出す前に、ウェイクアップ タイマーとして使用されるタイマーのソース クロックの電力モードを手動で構成する必要があります。

PSoC 3: 代替アクティブに切り替える前に、NONE以外のwakeupTimeが指定されている場合、適切なタイマーの状態が設定され、かつそのタイマーのインタラプトがディスエーブルになります。ウェイクアップソースは、wakeupSourceに指定してある値の組み合わせ、およびwakeupTime 引数に指定してあるタイマーです。ウェイクアップ条件が満たされた場合、全ての保存された状態が復旧され、関数はアクティブ状態に戻ります。

注 wakeupTimeが、異なる値からなる場合、ウェイクアップ前までの時間が、指定された時間より著しく短くなることがあります。次の呼び出しが同じwakeupTimeの値で行われた場合、ウェイクアップが前回のウェイクアップ後に指定された時間に起こります。

NONE以外のwakeupTimeが指定されている場合、終了時に指定されたタイマーの状態は、wakeupTimeによりイマーをイネーブルにし、インタラプトをディスエーブルとして指定された状態のままになります。たとえばSleepTimerまたはRTCコンポーネントにより、CTW、FTW、もしくはOne PPSがウェイクアップ用に既に設定してある場合は、wakeupTimeにNONEを設定し、wakeupSourceに適切なソースを入力してください。

PSoC 5: どちらのパラメーターも PSoC 5 では使用されていません。イネーブル インタラプトが生じるまで、デバイスは代替アクティブ モードに入ります。

パラメータ: wakeupTime: タイマーのウェイクアップソースと、そのソースの周波数を指定します。PSoC 5 において、このパラメーターは無視されます。

値	定義	時間
0	PM_ALT_ACT_TIME_NONE	なし
1	PM_ALT_ACT_TIME_ONE_PPS	1PPS: 1 秒
2	PM_ALT_ACT_TIME_CTW_2MS	CTW: 2 ミリ秒
3	PM_ALT_ACT_TIME_CTW_4MS	CTW: 4 ミリ秒
	PM_ALT_ACT_TIME_CTW_8MS	CTW: 8 ミリ秒
5	PM_ALT_ACT_TIME_CTW_16MS	CTW: 16 ミリ秒
6	PM_ALT_ACT_TIME_CTW_32MS	CTW: 32 ミリ秒
7	PM_ALT_ACT_TIME_CTW_64MS	CTW: 64 ミリ秒
8	PM_ALT_ACT_TIME_CTW_128MS	CTW: 128 ミリ秒
9	PM_ALT_ACT_TIME_CTW_256MS	CTW: 256 ミリ秒
10	PM_ALT_ACT_TIME_CTW_512MS	CTW: 512 ミリ秒
11	PM_ALT_ACT_TIME_CTW_1024MS	CTW: 1024 ミリ秒
12	PM_ALT_ACT_TIME_CTW_2048MS	CTW: 2048 ミリ秒
13	PM_ALT_ACT_TIME_CTW_4096MS	CTW: 4096 ミリ秒

CyPmAltAct (続き)

パラメータ: wakeupTime（続き）タイマーのウェイクアップソースと、そのソースの周波数を指定します。PSoC 5 において、このパラメーターは無視されます。

値	定義	時間
14 - 269	PM_ALT_ACT_TIME_FTW(1-256)	FTW: 10 マイクロ秒から 2.56ミリ秒

マクロPM_ALT_ACT_TIME_FTW()は、遅延時間を10 マイクロ秒の倍数として指定する、引数を取ります、PSoC 3 ES2 において、値の有効範囲は 1 ～ 32 です。PSoC 3 ES3 シリコンにおいて、値の有効範囲は 1 ～ 256 です。

wakeupSource: ウェイクアップソースのビット単位のマスクを指定します。さらに、wakeupTimeが指定された場合、関連するタイマーがウェイクアップソースとして追加されます。ウェイクアップソース構成は関数終了前に復元されます。PSoC 5 において、このパラメーターは無視されます。

値	定義	Source (ソース)
0	PM_ALT_ACT_SRC_NONE	なし
1	PM_ALT_ACT_SRC_COMPARATOR0	コンパレータ 0
2	PM_ALT_ACT_SRC_COMPARATOR1	コンパレータ 1
4	PM_ALT_ACT_SRC_COMPARATOR2	コンパレータ 2
8	PM_ALT_ACT_SRC_COMPARATOR3	コンパレータ 3
16	PM_ALT_ACT_SRC_INTERRUPT	インタラプト
64	PM_ALT_ACT_SRC_PICU	PICU
128	PM_ALT_ACT_SRC_I2C	I2C
512	PM_ALT_ACT_SRC_BOOSTCONVERTER	ブースト コンバータ
1024	PM_ALT_ACT_SRC_FTW	高速タイムホイール
2048*	PM_ALT_ACT_SRC_CTW	セントラルタイムホイール
2048*	PM_ALT_ACT_SRC_ONE_PPS	1PPS
4096	PM_ALT_ACT_SRC_LCD	LCD

注 CTWおよび1PPSウェイクアップ信号は、同じマスクビットに含まれます。

コンパレータをwakeupSourceに指定する場合、インスタンスが指定するコンパレータを追跡する、インスタンス固有の定義を使用してください。たとえば、MyCompというコンパレータインスタンスを使用する場合、マスクにORする値は MyComp_ctComp__CMP_MASK になります。CTW、FTW または One PPS がウェイクアップソースとして使用された場合、CyPmReadStatus 関数をウェイクアップ時に対応するパラメーターと共に呼び出す必要があります。詳細は CyPmReadStatus API を参照してください。

返り値: なし

副作用と制限: PSoC 3 ES2 と PSoC 5 シリコンでは、ウェイクアップソースは指定できません。この場合、wakeupSource引数は無視され、利用可能なウェイクアップソースの全てが、デバイスをウェイクさせることができます。

NONE以外のwakeupTimeが指定されている場合、終了時に指定されたタイマー時の状態は、wakeupTimeにより指定された状態に、タイマーをイネーブルにし、インタラプトをディスエーブルにされた状態になります。さらに、ILO 1 KHz（もし CTW タイマーがウェイクアップタイムとして使用された場合）または ILO 100 KHz（FTW タイマーがウェイクアップタイムとして使用された場合）は開始されたままになります。

void CyPmSleep(uint8 wakeupTime, uint16 wakeupSource)

説明: パーツをスリープ状態にします。

注 この関数を呼び出す前に、ウェイクアップ タイマーとして使用されるタイマーのソース クロックの電力モードを手動で構成する必要があります。

PSoC 3: スリープに切り替える前に、NONE以外のwakeupTimeが指定されている場合、適切なタイマーの状態が設定され、かつそのタイマーのインタラプトがディスエーブルになります。ウェイクアップソースは、wakeupSourceに指定してある値の組み合わせ、およびwakeupTime 引数に指定してあるタイマーです。ウェイクアップ条件が満たされた場合、全ての保存された状態が復旧され、関数はアクティブ状態に戻ります。

注 wakeupTimeが、異なる値からなる場合、ウェイクアップ前までの時間が、指定された時間より著しく短くなることがあります。次の呼び出しが同じwakeupTimeの値で行われた場合、ウェイクアップが前回のウェイクアップ後に指定された時間に起こります。

NONE以外のwakeupTimeが指定されている場合、終了時に指定されたタイマー時の状態は、wakeupTimeにより指定されたまま、タイマーをイネーブルにし、インタラプトをディスエーブルにされた状態になります。たとえばSleepTimerまたはRTCコンポーネントにより、CTW、もしくは1PPSがウェイクアップ用に既に設定してある場合は、wakeupTimeにNONEを設定し、wakeupSourceに適切なソースを入力してください。

PSoC 5: この関数に対するいずれのパラメーターも PSoC 5 に使用されません。デバイスはセントラルタイムホイール (CTW) からのインタラプトによりウェイクされるまで、スリープ モードに入ります。CTW は、インタラプトを生成するためにあらかじめコンフィギュレーションされている必要があります。SleepTimerコンポーネントを使用して構成されています。デバイスをスリープモードからウェイクするには、CTW のみ使用することができます。この関数は他のインタラプトソースを自動的にディスエーブルし、次にデバイスが CTW にウェイクされてから復元します。

スリープ期間はデバイスがスリープしてからすぐにウェイクアップしない、あるいは長い間スリープ状態にしないために、制御する必要があります。1ms ～ 8ms の信頼できるスリープタイムをサポートすることができます。この要件は 4, 8 または 16 ms の CTW 設定で満足されます。スリープタイムを制御するため、デバイスをスリープする前に、CTW が自動的にリセットされます。その結果として、最初の ILO クロック エッジの到着時間による1msの不確実さにより、ウェイクアップタイムは CTW にプログラムされた期間の半分になります。例えば、4 ミリ秒の設定は、1 ミリ秒～ 2 ミリ秒の間のスリープ時間になります。

パラメータ: wakeupTime: タイマーのウェイクアップソースと、そのソースの周波数を指定します。PSoC 5 において、このパラメーターは無視されます。

値	定義	時間
0	PM_SLEEP_TIME_NONE	なし
1	PM_SLEEP_TIME_ONE_PPS	1PPS: 1 秒
2	PM_SLEEP_TIME_CTW_2MS	CTW: 2 ミリ秒
3	PM_SLEEP_TIME_CTW_4MS	CTW: 4 ミリ秒
4	PM_SLEEP_TIME_CTW_8MS	CTW: 8 ミリ秒
5	PM_SLEEP_TIME_CTW_16MS	CTW: 16 ミリ秒
6	PM_SLEEP_TIME_CTW_32MS	CTW: 32 ミリ秒
7	PM_SLEEP_TIME_CTW_64MS	CTW: 64 ミリ秒
8	PM_SLEEP_TIME_CTW_128MS	CTW: 128 ミリ秒
9	PM_SLEEP_TIME_CTW_256MS	CTW: 256 ミリ秒
10	PM_SLEEP_TIME_CTW_512MS	CTW: 512 ミリ秒

CyPmSleep (続き)

パラメータ: wakeupTime (続き)

値	定義	時間
11	PM_SLEEP_TIME_CTW_1024MS	CTW: 1024 ミリ秒
12	PM_SLEEP_TIME_CTW_2048MS	CTW: 2048 ミリ秒
13	PM_SLEEP_TIME_CTW_4096MS	CTW: 4096 ミリ秒

wakeupSource: ウェイクアップソースのビット単位マスクを指定します。さらに、wakeupTime が指定された場合、関連するタイマーがウェイクアップソースとして追加されます。ウェイクアップソースコンフィグレーションは関数終了前に復元されます。PSoC 5 において、このパラメーターは無視されます。

値	定義	Source (ソース)
0	PM_SLEEP_SRC_NONE	なし
1	PM_SLEEP_SRC_COMPARATOR0	コンパレータ 0
2	PM_SLEEP_SRC_COMPARATOR1	コンパレータ 1
4	PM_SLEEP_SRC_COMPARATOR2	コンパレータ 2
8	PM_SLEEP_SRC_COMPARATOR3	コンパレータ 3
64	PM_SLEEP_SRC_PICU	PICU
128	PM_SLEEP_SRC_I2C	I2C
512	PM_SLEEP_SRC_BOOSTCONVERTER	ブースト コンバータ
2048*	PM_SLEEP_SRC_CTW	セントラルタイムホイール
2048*	PM_SLEEP_SRC_ONE_PPS	1PPS
4096	PM_SLEEP_SRC_LCD	LCD

注 CTWおよび1PPSウェイクアップ信号は、同じマスクビットに含まれます。PSoC 5 において、これらは異なるビットです（値 1024）。

コンパレータをwakeupSourceに指定する場合、そのインスタンス用のコンパレータと共に用いられる、インスタンス固有の定義を使用してください。たとえば、MyCompというコンパレータインスタンスを使用する場合、マスクにORする値は MyComp_ctComp__CMP_MASK になります。

CTW または One PPS がウェイクアップソースとして使用された場合、CyPmReadStatus 関数をウェイクアップ時に対応するパラメーターと共に呼び出す必要があります。詳細は CyPmReadStatus API を参照してください。

返り値: なし

副作用と制限: NONE以外のwakeupTimeが指定されている場合、終了時に指定されたタイマー時の状態はタイマーをイネーブルにし、インタラプトをディスエーブルにされた、wakeupTimeにより指定された状態のままになります。さらに、ILO 1 KHz（もし CTW タイマーがウェイクアップタイムとして使用された場合）は開始されたままになります。

PSoC 3 ES2 シリコンには、デバイスが低電力モードに入ると、いくつかのアナログリソースへの接続が不安定になる欠陥があります。詳細はシリコンエラッタを参照してください。

1 kHz ILO クロックはPSoC3 ES3 シリコンに対してイネーブルされることが期待され、リセット後のハイバネート/スリープレギュレータセトリング時間を測定します。ホールドオフ遅延は 1 kHz ILO の立ち上がりエッジを使用して測定されます。

void CyPmHibernate()

説明: パーツをハイバネート状態にします。

PSoC 3: ハイバネートに切り替える前に、PICU ウェイクアップソースビットの現在の状態が保存され、次に設定されます。これでデバイスが構成され、PICU からウェイクアップします。PICU インタラプトを生成するには、1 つ以上のピンを構成してください。ピン Px.y に対しては、「PICU_INTTYPE_PICUx_INTTYPEy」が PICU 動作を制御します。TRM において、このレジスタは「PICU[0..15]_INTTYPE[0..7]」です。ピンコンポーネントデータシートにおいて、このレジスタは IRQ オプションとして参照されます。いったんウェイクアップが起きると、PICU ウェイクアップソースビットは復元され、PSoC はアクティブ状態に戻ります。

PSoC 5: ハイバネート状態からウェイクアップするためにサポートされている唯一の方法は、デバイスのハードウェアリセットです。PICU インタラプトソースは、デバイスをハイバネート状態にする前に、この関数により自動的にディスエーブルになります。

パラメータ: なし

返り値: なし

副作用と制限: アプリケーションは再度ハイバネートに入る前、またはハイバネートからウェイクアップした後スリープしてから 20 µs 待つ必要があります。この 20 µs でスリープレギュレーターは次のハイバネート/スリープイベントが起きる前に安定することができます。20 µs 要件はデバイスがウェイクアップしてから開始します。この要求事項が満たされたかを確認する、ハードウェアチェックは存在しません。指定された遅延は、ISRエントリ上で行われるべきです。

ウェイクアップ PICU インタラプトが生じた後、Pin_ClearInterrupt() 関数（ここでは「ピン」はピンコンポーネントのインスタンス名）はラッチされたピンイベントをクリアするために呼び出す必要があります。これにより、適切なハイバネートモードエントリが行われ、将来のイベントの検出が可能になります。

PSoC 3 ES2 シリコンには、デバイスが低電力モードに入ると、いくつかのアナログリソースへの接続が不安定になる欠陥があります。詳しくは、シリコンのエラッタを参照してください。

1 kHz ILO クロックはPSoC3 ES3 シリコンに対して、リセット後のハイバネート/スリープレギュレータセトリング時間を測定するためにイネーブルされることが望まれます。ホールドオフ遅延は 1 kHz ILO の立ち上がりエッジを使用して測定されます。

uint8 CyPmReadStatus(uint8 mask)

説明: パワーマネジャインタラプトステータスレジスタを管理します。レジスタには、1PPS、セントラルタイムホイール、および高速タイムホイールタイマーのインタラプトステータスが保持されます。このハードウェアレジスタは、読み込まれるとクリアします。必要なビットのみをクリアし、残りのビットを保全できるように、この関数は状態を保持するシャドーレジスタを使用します。この関数はステータスレジスタを読み込み、ステータスレジスタとシャドーレジスタをORします。この値が返されます。次に、セットされたマスクのビットがこの値からクリアされ、シャドーレジスタに書き戻されます。

注 この関数は CTW イベントが起きてから 1 ミリ秒以内（ILO の 1 クロックサイクル以内）に呼び出す必要があります。

パラメータ: mask: シャドーレジスタからクリアするビット

値	定義	Source (ソース)
1	CY_PM_FTW_INT	高速タイムホイール
2	CY_PM_CTW_INT	セントラルタイムホイール
4	CY_PM_ONEPPS_INT	1PPS

返り値: ステータス マスクパラメータに使用されたのと同じ、列挙数型ビット値。

インスタンスの低電力 API

多くのコンポーネントには、コンポーネントを低電力状態(スリープまたはハイバネート)にするための、インスタンス固有な低電力 API が存在します。これらの関数の一般例を以下に記載します。レジスタ保持に関する詳細情報が必要な場合、個別のデータシートを参照してください。

void `=instance_name`_Sleep (void)

説明: _Sleep()関数は、コンポーネントがイネーブルか否かを確認し、その状態を保存します。その後、_Stop()関数を呼び出し、ユーザーの設定を保存するために _SaveConfig() 関数を呼び出します。
CyPmSleep() および CyPmHibernate()関数を呼び出す前に、_Sleep()関数を呼び出してください。

パラメータ: なし

返り値: なし

副作用: なし

void `=instance_name`_Wakeup (void)

説明: _Wakeup()関数は、ユーザー設定を復旧させるために_RestoreConfig()関数を呼び出します。このコンポーネントが_Sleep()関数を呼び出す前にイネーブルにされた場合、_Wakeup() はコンポーネントを再度イネーブルにします。

パラメータ: なし

返り値: なし

副作用: 最初に _Sleep() または _SaveConfig() 関数を呼び出すことなく _Wakeup() 関数を呼び出すと、予期されない振る舞いにつながる可能性があります。

void `=instance_name`_SaveConfig(void)

説明: この関数は、コンポーネントの設定を保存します。保持されないレジスタも保存します。この関数は、[Configure] (設定) ダイアログで定義されている、または該当する API で変更される、現在のコンポーネント パラメータ値も保存します。この関数は、_Sleep()関数に呼び出されます。

パラメータ: なし

返り値: なし

副作用: なし

void `=instance_name`_RestoreConfig(void)

説明: この関数は、コンポーネントの設定を復元します。維持されないレジスタも復元します。この関数はまた、コンポーネントのパラメータ値を_Sleep()関数を呼び出す前の状態に復旧します。

パラメータ: なし

返り値: なし

副作用: _Sleep()または SaveConfig() 関数を呼び出す前に、この関数を呼び出した場合、予期しない挙動を示すことがあります。

5 インタラプト

この章の API は特に示されていない限り、すべてのアーキテクチャに適用されます。インタラプトについての詳細は、インタラプトコンポーネントデータシートも参照してください。

API

CyGlobalIntEnable

説明: グローバルインタラプトマスクを使用したインタラプトをイネーブルにする、マクロ文です。

CyGlobalIntDisable

説明: グローバルインタラプトマスクを使用したインタラプトをディスエーブルにする、マクロ文です。

uint32 CyDisableInts()

説明: 各インタラプトに対するインタラプト イネーブルをディスエーブルにします。

パラメータ: なし

返り値: 以前イネーブルだった、インタラプトの32ビットマスク

void CyEnableInts(uint32 mask)

説明: 32ビットマスクに指定された、全てのインタラプトをイネーブルにします。

パラメータ: mask: イネーブルにする、インタラプトの32ビットマスク

返り値: なし

注 インタラプトサービスルーチンは、エントリー時の状態に、CYDEV_INTC_CSR_ENレジスタビットとインタラプトイネーブル状態(EA)を復旧するというポリシーに従う必要があります。ISRは、呼び出しに対し、正しくネストされた CyEnterCriticalSection/CyExitCriticalSection 関数を使用する限り、特に何もする必要がありません。

void CyIntEnable(uint8 number)

説明: 指定されたインタラプト番号をイネーブルにします。

パラメータ: number: インタラプト番号 有効な範囲:[0-31]

返り値: なし

注 インタラプトサービスルーチンは、CYDEV_INTC_CSR_ENレジスタビットとインタラプトイネーブル状態(EA)を、エントリー時の状態に復旧するというポリシーに従う必要があります。ISRは、呼び出しに対し、正しくネストされたCyEnterCriticalSection/CyExitCriticalSection 関数を使用する限り、特に何もする必要がありません。

void CyIntDisable(uint8 number)

説明: 指定されたインタラプト番号をディスエーブルにします。

パラメータ: number: インタラプト番号 有効な範囲:[0-31]

返り値: なし

注 インタラプトサービスルーチンは、CYDEV_INTC_CSR_ENレジスタビットとインタラプトイネーブル状態(EA)を、エントリー時の状態に復旧するというポリシーに従う必要があります。ISRは、呼び出しに対し、正しくネストされたCyEnterCriticalSection/CyExitCriticalSection 関数を使用する限り、特に何もする必要がありません。

uint8 CyIntGetState(uint8 number)

説明: 指定されたインタラプト番号のイネーブル状態を取得します。

パラメータ: number: インタラプト番号 有効な範囲:[0-31]

返り値: イネーブル状態: イネーブルなら1、ディスエーブルなら0

cyisraddress CyIntSetVector(uint8 number, cyisraddress address)

説明: 指定されたインタラプト番号のインタラプトベクタを設定します。

パラメータ: number: インタラプト番号 有効な範囲:[0-31]

address: インタラプトサービスルーチンへのポインタ

返り値: 以前のインタラプトベクタ値

cyisraddress CyIntGetVector(uint8 number)

説明: 指定されたインタラプト番号のインタラプトベクタを取得します。

パラメータ: number: インタラプト番号 有効な範囲:[0-31]

返り値: インタラプトベクタ値

cyisraddress CyIntSetSysVector(uint8 number, cyisraddress address)

説明: この関数は ARM ベースのプロセッサにのみ適用されるためPSoC 3 デバイスには適用されません。指定された例外のインタラプトベクトルを設定します。ARM アーキテクチャのこれらの例外は、ユーザーインタラプトと同様に作動しますが、プロセッサのシステムアーキテクチャにより指定されます。各例外の数は固定されています。これらの例外の番号付けは、ユーザーインタラプトで使用されるものと別です。

パラメータ: number: 例外番号 有効な範囲:[0-15]。

address: インタラプトサービスルーチンへのポインタ

返り値: 以前のインタラプトベクタ値

cyisraddress CyIntGetSysVector(uint8 number)

説明: この関数は ARM ベースのプロセッサにのみ適用されるためPSoC 3 デバイスには適用されません。指定された例外のインタラプトベクトルを取得します。ARM アーキテクチャのこれらの例外は、ユーザーインタラプトと同様に作動しますが、プロセッサのシステムアーキテクチャによって定義されています。各例外の番号は固定されています。これらの例外の番号付けは、ユーザーインタラプトで使用されるものと別です。

パラメータ: number: 例外番号 有効な範囲:[0-15]。

返り値: インタラプトベクタ値

void CyIntSetPriority(uint8 number, uint8 priority)

説明: 指定されたインタラプト番号の優先度を設定します。

パラメータ: number: インタラプト番号 有効な範囲:[0-31]

priority: インタラプト優先度。0 が優先度の最高値です。有効な範囲:[0-7]

返り値: なし

uint8 CyIntGetPriority(uint8 number)

説明: 指定されたインタラプト番号の優先度を取得します。

パラメータ: number: インタラプト番号 有効な範囲:[0-31]

返り値: インタラプト優先度

void CyIntSetPending(uint8 number)

説明: 指定されたインタラプト番号を強制的にペンディングにします。

パラメータ: number: インタラプト番号 有効な範囲:[0-31]

返り値: なし

void CyIntClearPending(uint8 number)

説明: 指定されたインタラプト番号のペンディングをクリアします。

パラメータ: number: インタラプト番号 有効な範囲:[0-31]

返り値: なし

6 キャッシュ

PSoC 3 キャッシュ機能

PSoC 3 キャッシュは初期設定でイネーブルになっています。PSoC Creator Design-Wide Resources System Editor（PSoC Creator 設計全体リソースシステムエディタ）を使用してディスエーブルにすることができます。PSoC 3 のキャッシュの取り扱いについての、定義、関数およびマクロは存在しません。

PSoC 5 キャッシュ機能

void CyFlushCache()

説明: 全てのエントリを無効化することで、PSoC 5 キャッシュをフラッシュします。

パラメータ: なし

返り値: なし

7 Pins

Pins コンポーネントの一部として提供されたピン用の関数に加えて、ピン用マクロのライブラリが *cypins.h* ファイル内に提供されています。これらのマクロは、デバイス上の全てのピンに対して存在する、ポートピン設定レジスタを使用します。このレジスタのアドレスは、*cydevice_trm.h* ファイルに含まれます。これらのピン設定レジスタは次のような名称です:

```
CYREG_PRTx_PCy
```

x はポート番号で、y はポート内のピン番号です。

API

uint8 CyPins_ReadPin(uint16/uint32 pinPC)

説明: ピンの現在の値(ピン状態、PS)を読み込みます。

パラメータ: pinPC: ポートピン設定レジスタ(PSoC 3ではuint16、PSoC 5ではuint32)

返り値: ピン状態

0: ロジックロー

0以外: ロジックハイ

void CyPins_SetPin(uint16/uint32 pinPC)

説明: ピンのアウトプット値(データレジスタ、DR)をロジックハイにセットします。これは、ハードウェアにより駆動されていない、ソフトウェアピンに対してのみ有効です。

パラメータ: pinPC: ポートピン設定レジスタ(PSoC 3ではuint16、PSoC 5ではuint32)

返り値: なし

void CyPins_ClearPin(uint16/uint32 pinPC)

説明: ピンのアウトプット値(データレジスタ、DR)をロジックローにクリアします。これは、ハードウェアにより駆動されていない、ソフトウェアピンに対してのみ有効です。

パラメータ: pinPC: ポートピン設定レジスタ(PSoC 3ではuint16、PSoC 5ではuint32)

返り値: なし

void CyPins_SetPinDriveMode(uint16/uint32 pinPC, uint8 mode)

説明: ピンのドライブモード(DM)を設定します。

パラメータ: pinPC: ポートピン設定レジスタ(PSoC 3ではuint16、PSoC 5ではuint32)

mode: 設定する駆動モード

定義	Source (ソース)
PIN_DM_ALG_HIZ	アナログ HiZ
PIN_DM_DIG_HIZ	デジタルHiZ
PIN_DM_RES_UP	抵抗プルアップ
PIN_DM_RES_DWN	抵抗プルダウン
PIN_DM_OD_LO	オープンドレイン - ロー駆動
PIN_DM_OD_HI	オープンドレイン - ハイ駆動
PIN_DM_STRONG	ストロングCMOS出力
PIN_DM_RES_UPDWN	抵抗プルアップ/プルダウン

返り値: なし

uint8 CyPins_ReadPinDriveMode(uint16/uint32 pinPC)

説明: ピンのドライブモード(DM)を読み込みます。

パラメータ: pinPC: ポートピン設定レジスタ(PSoC 3ではuint16、PSoC 5ではuint32)

返り値: ピンの現在の駆動モード

定義	Source (ソース)
PIN_DM_ALG_HIZ	アナログ HiZ
PIN_DM_DIG_HIZ	デジタルHiZ
PIN_DM_RES_UP	抵抗プルアップ
PIN_DM_RES_DWN	抵抗プルダウン
PIN_DM_OD_LO	オープンドレイン - ロー駆動
PIN_DM_OD_HI	オープンドレイン - ハイ駆動
PIN_DM_STRONG	ストロングCMOS出力
PIN_DM_RES_UPDWN	抵抗プルアップ/プルダウン

void CyPins_FastSlew(uint16/uint32 pinPC)

説明: ピンのスルーレートを高速エッジレートにします。これは抵抗駆動モードではなく、ストロング出力駆動モードのピンにのみ適用されます。

パラメータ: pinPC: ポートピン設定レジスタ(PSoC 3ではuint16、PSoC 5ではuint32)

返り値: なし

void CyPins_SlowSlew(uint16/uint32 pinPC)

説明: ピンのスルーレートを低速エッジレートにします。これは抵抗駆動モードではなく、ストロング出力駆動モードのピンにのみ適用されます。

パラメータ: pinPC: ポートピン設定レジスタ(PSoC 3ではuint16、PSoC 5ではuint32)

返り値: なし

8 レジスタのアクセス

マクロのライブラリにより、デバイスのレジスタへの読み書きアクセスが提供されます。これらのマクロは自動生成された *cydevice.h*、*cydevice_trm.h* と *cyfitter.h* ファイルで利用できるようになった定義値と共に使用されます。レジスタへのアクセスには、これらのマクロを実装するのに使用した関数ではなく、これらのマクロを使用してください。これにより、デバイスに依存しないコードが生成されます。

PSoC 3 は 8-ビットアーキテクチャのため、プロセッサはエンディアンがありません。しかしながら、8 ビットアーキテクチャのコンパイラは、エンディアンを実装します。PSoC 3 では Keil コンパイラがビッグエンディアン（最も下位のアドレス中に MSB）で実装しています。PSoC 5 プロセッサのアーキテクチャは、リトルエンディアンを使用します。

PSoC 3 および PSoC 5 アーキテクチャの両者における SRAM と Flash ストレージは、アーキテクチャとコンパイラのエンディアンを使用して行われます。しかしながら、これらのチップの両方のレジスタは、リトルエンディアンの順番になっています。これらのマクロはこのリトルエンディアンの順番に一致するレジスタアクセスを許可します。これらのマクロを使用しないで、マルチバイトレジスタにおいて操作を行う場合、指定したアーキテクチャのバイトの順番を考慮する必要があります。例として、DMA を使用してメモリとレジスタ間での転送、そしてメモリでバイトのアレイで渡される関数の呼び出しなどがあります。

PSoC 3 は 8 ビットのプロセッサで、すべてのアクセスはバイトごとに行われます。PSoC 5 は適切な 8-、16- および 32-ビットアクセスを使用してアクセスします。PSoC 5 はこれらのアクセスをトランザクションの幅に合わせる必要がありません。

API

uint8 CY_GET_REG8(uint16/uint32 reg)

説明: 指定されたレジスタから8ビットの値を読み込みます。PSoC 3 において、アドレスは下位 64 K アドレスの範囲でなければなりません。

パラメータ: reg: レジスタアドレス(PSoC 3ではuint16、PSoC 5ではuint32)

返り値: 読み取り値

void CY_SET_REG8(uint16/uint32 reg, uint8 value)

説明: 指定されたレジスタに8ビットの値を書き込みます。PSoC 3 において、アドレスは下位 64 K アドレスの範囲でなければなりません。

パラメータ: reg: レジスタアドレス(PSoC 3ではuint16、PSoC 5ではuint32)

value: 書きこむ値

返り値: なし

uint16 CY_GET_REG16(uint16/uint32 reg)

説明: 指定されたレジスタから16ビットの値を読み込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3 において、アドレスは下位 64 K アドレスの範囲でなければなりません。

パラメータ: reg: レジスタアドレス(PSoC 3ではuint16、PSoC 5ではuint32)

返り値: 読み取り値

void CY_SET_REG16(uint16/uint32 reg, uint16 value)

説明: 指定されたレジスタに16ビットの値を書き込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3 において、アドレスは下位 64 K アドレスの範囲でなければなりません。

パラメータ: reg: レジスタアドレス(PSoC 3ではuint16、PSoC 5ではuint32)

value: 書きこむ値

返り値: なし

uint32 CY_GET_REG24(uint16/uint32 reg)

説明: 指定されたレジスタに24ビットの値を読み込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3 において、アドレスは下位 64 K アドレスの範囲でなければなりません。

パラメータ: reg: レジスタアドレス(PSoC 3ではuint16、PSoC 5ではuint32)

返り値: 読み取り値

void CY_SET_REG24(uint16/uint32 reg, uint32 value)

説明: 指定されたレジスタに24ビットの値を書き込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3 において、アドレスは下位 64 K アドレスの範囲でなければなりません。

パラメータ: reg: レジスタアドレス(PSoC 3ではuint16、PSoC 5ではuint32)

value: 書きこむ値

返り値: なし

uint32 CY_GET_REG32(uint16/uint32 reg)

説明: 指定されたレジスタから32ビットの値を読み込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3 において、アドレスは下位 64 K アドレスの範囲でなければなりません。

パラメータ: reg: レジスタアドレス(PSoC 3ではuint16、PSoC 5ではuint32)

返り値: 読み取り値

void CY_SET_REG32(uint16/uint32 reg, uint32 value)

説明: 指定されたレジスタに32ビットの値を書き込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3 において、アドレスは下位 64 K アドレスの範囲でなければなりません。

パラメータ: reg: レジスタアドレス(PSoC 3ではuint16、PSoC 5ではuint32)

value: 書きこむ値

返り値: なし

uint8 CY_GET_XTND_REG8(uint32 reg)

説明: 指定されたレジスタから8ビットの値を読み込みます。PSoC 3のアドレス空間全てをサポートしますが、標準のレジスタ取得関数より実行サイクル数が多いです。PSoC 5 において、CY_GET_REG8 と同一。

パラメータ: reg: レジスタのアドレス

返り値: 読み取り値

void CY_SET_XTND_REG8(uint32 reg, uint8 value)

説明: 指定されたレジスタに8ビットの値を書き込みます。PSoC 3のアドレス空間全てをサポートしますが、標準のレジスタ設定関数より実行サイクル数が多いです。PSoC 5 において、CY_SET_REG8 と同一。

パラメータ: reg: レジスタのアドレス

value: 書きこむ値

返り値: なし

uint16 CY_GET_XTND_REG16(uint32 reg)

説明: 指定されたレジスタから16ビットの値を読み込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3のアドレス空間全てをサポートしますが、標準のレジスタ取得関数より実行サイクル数が多いです。PSoC 5 において、CY_GET_REG16 と同一。

パラメータ: reg: レジスタのアドレス

返り値: 読み取り値

void CY_SET_XTND_REG16(uint32 reg, uint16 value)

説明: 指定されたレジスタに16ビットの値を書き込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3のアドレス空間全てをサポートしますが、標準のレジスタ設定関数より実行サイクル数が多いです。PSoC 5 において、CY_SET_REG16 と同一。

パラメータ: reg: レジスタのアドレス

value: 書きこむ値

返り値: なし

uint32 CY_GET_XTND_REG24(uint32 reg)

説明: 指定されたレジスタに24ビットの値を読み込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3のアドレス空間全てをサポートしますが、標準のレジスタ取得関数より実行サイクル数が多いです。PSoC 5 において、CY_GET_REG24 と同一。

パラメータ: reg: レジスタのアドレス

返り値: 読み取り値

void CY_SET_XTND_REG24(uint32 reg, uint32 value)

説明: 指定されたレジスタに24ビットの値を書き込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3のアドレス空間全てをサポートしますが、標準のレジスタ設定関数より実行サイクル数が多いです。PSoC 5 において、CY_SET_REG24 と同一。

パラメータ: reg: レジスタのアドレス

value: 書きこむ値

返り値: なし

uint32 CY_GET_XTND_REG32(uint32 reg)

説明: 指定されたレジスタから32ビットの値を読み込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3のアドレス空間全てをサポートしますが、標準のレジスタ取得関数より実行サイクル数が多いです。PSoC 5 において、CY_GET_REG32 と同一。

パラメータ: reg: レジスタのアドレス

返り値: 読み取り値

void CY_SET_XTND_REG32(uint32 reg, uint32 value)

説明: 指定されたレジスタに32ビットの値を書き込みます。このマクロは、正しい動作のためのバイトスワップを行います。PSoC 3のアドレス空間全てをサポートしますが、標準のレジスタ設定関数より実行サイクル数が多いです。PSoC 5 において、CY_SET_REG32 と同一。

パラメータ: reg: レジスタのアドレス

value: 書きこむ値

返り値: なし

9 DMA

DMA ファイルは、DMA コントローラ、DMA チャネル、および転送ディスクリプタ用の API 関数を提供します。この API は、ユーザーが DMA コンポーネントを回路図に配置した時に生成されるコードではなく、ライブラリ内のコードです。自動生成されたコードは、このモジュールの API を使用します。

詳細情報については、DMA コンポーネントデータシートを参照してください。

10 Flash および EEPROM

Flash と EEPROM は共通な関数セットを使用してプログラムされます。詳細情報については、EEPROM コンポーネントデータシートを参照してください。

Flash および EEPROM は、SPC (システムパフォーマンスコントローラ(System Performance Controller))呼び出しを通してプログラムされます。Flash/EEPROM 固有の API は、このことを単純にするために抽象化しています。

PSoC 3 パーツのみにエラー修正コード (ECC) ストレージがあります。これは、本来の ECC として構成しても、データの保存用として構成してもかまいません。データ保存用として ECC メモリーを使用する最も一般的な利用方法は、PSoC Creator により直接サポートされているコンフィギュレーションデータを保存することです。

ECC がディスエーブルになっている Flash をプログラミングする際（データ保存用に利用可能）、データ行を書き込むための方法がいくつかあります。

- ECC を含む行全体
- ECC を含まない行全体
- ECC メモリーのみ

コンフィギュレーションデータの保存用に ECC メモリーを使用する場合、コンフィギュレーションデータに使用されている ECC メモリーの領域を上書きしないようにしてください。

呼び出し関数は、最初に CySetTemp および CySetFlashEEBuffer 関数を呼び出す必要があります。温度は、フラッシュへの書き込み時間を最適化するのに必要です。バッファは、SPC と通信する間に、中間データを保持するのに使用します。SPC は、コマンドを送信し、データを読み出すためのレジスタを持つ、プッシュプルです。

Flash および EEPROM に対し、同じ関数「CyWriteRowData」を呼び出すことで、1 行ずつ書きこむことができます。最初のパラメータは、Flash または EEPROM アレイを指定します。フラッシュのアレイ数と、EEPROM のアレイ数は、選択されたパーツ固有のものです。どのアレイ ID が有効であるか、パーツをチェックしてください。

Flash アレイは最大 64 KB、さらに ECC バイトがあります。PSoC 3 アーキテクチャは 1 つの Flash アレイがあり、そのサイズは 16 KB、32 KB または 64 KB に ECC バイトを加えたものです。そのため、唯一の有効なアレイ ID は 0x00 です。

EEPROM アレイは最大 2 KB です。PSoC 3 と PSoC 5 デバイスは 1 つの EEPROM アレイがあり、そのサイズは 512 バイト、1 KB または 2 KB です。

API

cystatus CySetTemp()

説明: ダイの温度を取得し、FlashおよびEEPROM書き込み関数が使用する静的な場所に結果を保存します。この関数は一連のFlash / EEPROM 書き込み関数を実行する前に一度呼び出す必要があります。

パラメータ: なし

返り値: ステータス

値	説明
CYRET_SUCCESS	成功
CYRET_LOCKED	Flash / EEPROM書き込みが、既に使用されています
CYRET_UNKNOWN	失敗

副作用と制限: この関数の実行には、時間がかかります。
関数は SPC がアイドル状態になるまで戻りません。

cystatus CySetFlashEEBuffer(uint8 *buffer)

説明: FlashおよびEEPROM書き込み中に、フラッシュの1列分全て、および関連して使用されるECCを、一時的に保管するのに使われるバッファを設定します。このバッファはFlash ECC がディスエーブルのときのみ必要です。

パラメータ: uint8 *バッファ: (SIZEOF_FLASH_ROW + SIZEOF_ECC_ROW) バイトの、割り当てられたバッファです。

返り値: ステータス

値	説明
CYRET_SUCCESS	成功
CYRET_LOCKED	Flash / EEPROM書き込みが、既に使用されています

cystatus CyWriteRowFull(uint8 arrayId, uint16 rowAddress, uint8 *rowData, uint16 rowSize)

説明: 列を消去し、プログラムすることを可能にします。アレイがフラッシュアレイであり、ECCが設定の保存に使用されている場合、この関数は、列データとECCデータの両方が rowDataの一部として提供されていると仮定します。

パラメータ: uint8 arrayId: 書き込むアレイのIDです。書き込みタイプ(FlashもしくはEEPROM)は、アレイIDから判断されます。パーツ内のアレイは、各メモリタイプの最初のIDから連続して存在します。

種類	最初のID	アレイのサイズ
フラッシュ	FIRST_FLASH_ARRAYID	64 キロバイト
EEPROM	FIRST_EE_ARRAYID	2 キロバイト

uint16 rowAddress: 指定された arrayId 内の列のアドレス。

種類	アレイ内の列の数	列のサイズ(バイト単位)
Flash（ECC イネーブル）	256	SIZEOF_FLASH_ROW (256)
Flash（ECC ディスエーブル）	288	SIZEOF_FLASH_ROW + SIZEOF_ECC_ROW (288)
EEPROM	128	SIZEOF_EEPROM_ROW (16)

uint8 *rowData: プログラムするデータのアドレス この列のサイズは、「arrayId」によって、SIZEOF_FLASH_ROW、もしくは SIZEOF_EEPROM_ROW です。

注 このバッファは、SPCバッファとして渡されるバッファと同じであってはなりません。

uint16 rowSize: 行データのバイト数

返り値: ステータス

値	説明
CYRET_SUCCESS	成功
CYRET_LOCKED	Flash / EEPROM書き込みが、既に使用されています
CYRET_CANCELED	コマンドが受け付けられませんでした
他の0以外の値	失敗

cystatus CyWriteRowData(uint8 arrayId, uint16 rowAddress, uint8 *rowData)

説明: FlashもしくはEEPROMを1列書き込みます。Flashの場合、ECCがイネーブルならばECCが自動的に書き込まれます。ECCがディスエーブルならば、ECCメモリーの現在の内容は保持されます。

パラメータ: uint8 arrayId: 書き込むアレイのIDです。書き込みタイプ(FlashもしくはEEPROM)は、アレイIDから判断されます。パーツ内のアレイは、各メモリタイプの最初のIDから連続して存在します。

種類	最初のID	アレイのサイズ
フラッシュ	FIRST_FLASH_ARRAYID	64キロバイト
EEPROM	FIRST_EE_ARRAYID	2キロバイト

uint16 rowAddress: 指定された arrayId 内の列のアドレス。

種類	アレイ内の列の数	列のサイズ(バイト単位)
フラッシュ	256	SIZEOF_FLASH_ROW (256)
EEPROM	128	SIZEOF_EEPROM_ROW (16)

書き込む uint8 *rowData バイトのアレイ

返り値: ステータス

値	説明
CYRET_SUCCESS	成功
CYRET_LOCKED	Flash / EEPROM書き込みが、既に使用されています
CYRET_CANCELED	コマンドが受け付けられませんでした
他の0以外の値	失敗

cystatus CyWriteRowConfig(uint8 arrayId, uint16 rowAddress, uint8 *rowData)

説明: Flash列のECC部分を書き込みます。この関数は、ECCが無効であり、設定データの保存に使用されていない場合のみ存在します。この関数は、FlashアレイIDに対してのみ有効です(EEPROMにおいては無効です)。

パラメータ: arrayId: 書き込むアレイのIDです。パーツ内のアレイは、各メモリタイプの最初のIDから連続して存在します。

種類	最初のID	ECCアレイのサイズ
フラッシュ	FIRST_FLASH_ARRAYID	8キロバイト

rowAddress: 指定された arrayId 内の列のアドレス。

種類	アレイ内の列の数	列のサイズ(バイト単位)
フラッシュ	256	SIZEOF_ECC_ROW (32)

rowData: 書き込むバイトのアレイ。

返り値: ステータス

値	説明
CYRET_SUCCESS	成功
CYRET_LOCKED	Flash / EEPROM書き込みが、既に使用されています
CYRET_CANCELED	コマンドが受け付けられませんでした
他の0以外の値	失敗

void CyFlash_Start()

説明: Flashを有効にします。デフォルトでは、Flashは有効です。
PSoC 3 ES2およびそれ以前、そしてPSoC 5においては、同じビットがEEPROMとFlash両方を管理します。片方をスタートする、もしくは停止すると、もう片方もスタートもしくは停止します。

パラメータ: なし

返り値: なし

void CyFlash_Stop()

説明: Flashを無効化します。この設定は、CPUが動作中の間は無視されますです。これは、CPUが無効化された後に有効になります。
PSoC 3 ES2およびそれ以前、そしてPSoC 5においては、同じビットがEEPROMとFlash両方を管理します。片方をスタートする、もしくは停止すると、もう片方もスタートもしくは停止します。

パラメータ: なし

返り値: なし

void CyFlash_SetWaitCycles(uint8 freq)

説明: デバイスの動作周期に合わせて、フラッシュの適切なウェイトサイクルを設定します。この関数は、クロック周波数を引き上げる前に呼び出すべきです。クロック周波数を減らした後に、CPUパフォーマンスを向上させるために呼び出すこともできます。

パラメータ: freq: 動作周波数(メガヘルツ単位)

返り値: なし

void CyEEPROM_Start()

説明: EEPROMをイネーブルします。
PSoC 3 ES2およびそれ以前、そしてPSoC 5においては、同じビットがEEPROMとFlash両方を管理します。片方をスタートする、もしくは停止すると、もう片方もスタートもしくは停止します。これらのシリコンのバージョンでは、EEPROMが初期設定でイネーブルになっています。以降のシリコンのバージョンでは、EEPROM は別のビットで管理されており、使用前にスタートする必要があります。

パラメータ: なし

返り値: なし

void CyEEPROM_Stop()

説明: EEPROMをディスエーブルにします。
PSoC 3 ES2およびそれ以前、そしてPSoC 5においては、同じビットがEEPROMとFlash両方を管理します。片方をスタートする、もしくは停止すると、もう片方もスタートもしくは停止します。

パラメータ: なし

返り値: なし

void CyEEPROM_ReadReserve()

説明: EEPROMに対し読み込みアクセスをリクエストし、アクセス可能になるまでウェイトします。EEPROMへのアクセスは、EEPROMに書き込むコントローラと、EEPROMから読み込む通常アクセスの間で仲介されます。読み取りのために EEPRO へのアクセスを控える必要はありませんが、書き込みが未だにアクティブな場合で、読み取りが試みられると、フォールトが発生し、誤ったデータが戻されます。

パラメータ: なし

返り値: なし

void CyEEPROM_ReadRelease()

説明: EEPROMに対する読み込み予約を解除します。EEPROMに対する読み込み予約がある場合、EEPROMに追加の書き込みをする前に予約を解除する必要があります。

パラメータ: なし

返り値: なし

11 ブートローダ システム

PSoC Creator においては、ブートローダ システムが、デバイスのフラッシュメモリの内容を、新しいアプリケーションコードおよびデータ(「コード」)にアップデートするプロセスを管理します。このプロセスは、以下により行われます:

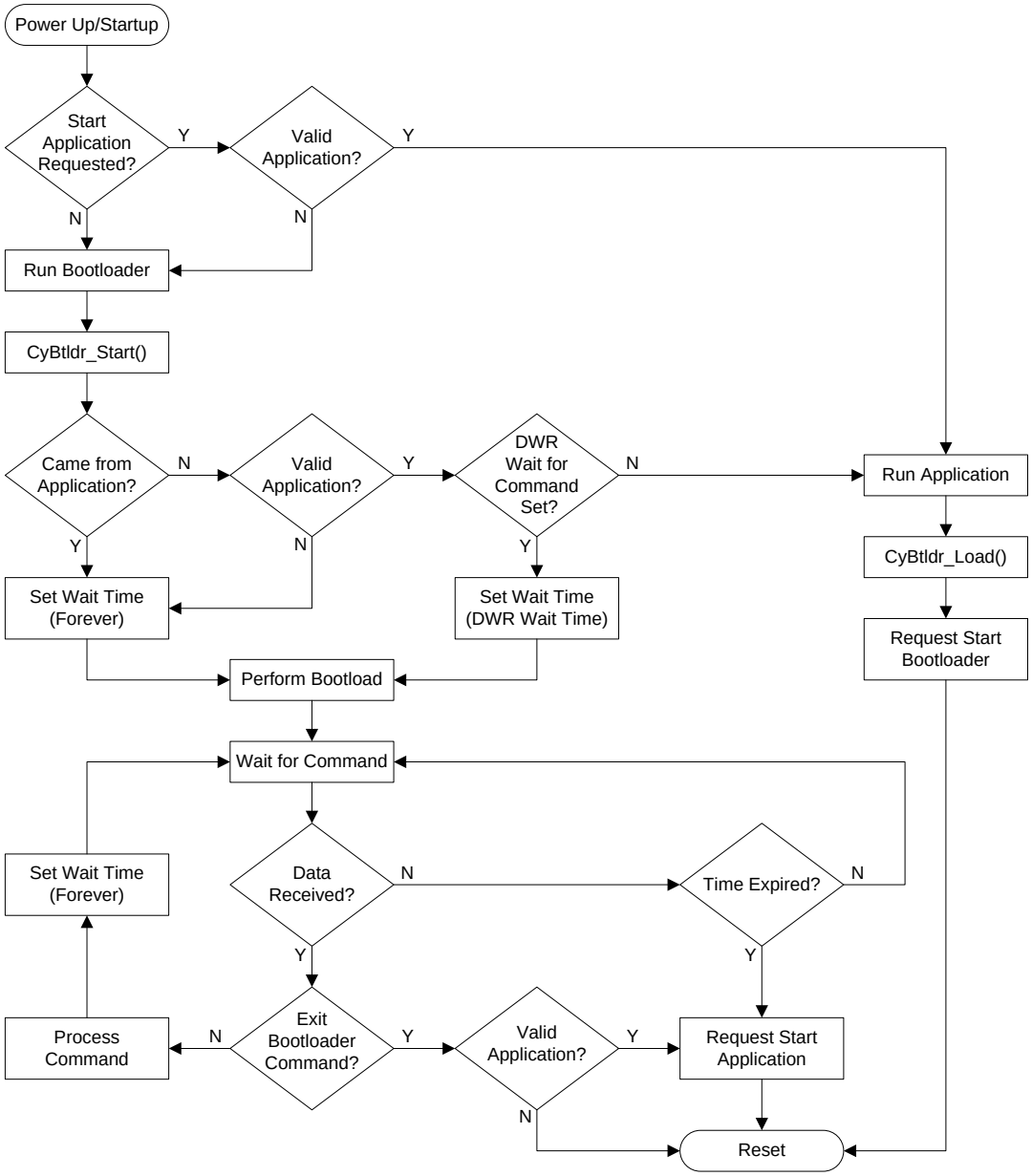
- ブートローダ コンポーネント
- 通信コンポーネント
- ブートローダ コンポーネントを生成するための、ブートローダ プロジェクト
- コードを生成するための、ブートロード可能なプロジェクト

以下の節では、ブートローダプロセスの各側面をより詳細に解説してあります。

ブートローダ コンポーネント

ブートローダ コンポーネントは、デバイスのフラッシュメモリの内容を、新しいコードにアップデートすることを可能にします。ブートローダは、コマンドを受け付けて実行し、コマンドに対する応答を通信コンポーネントに送信します。ブートローダは、受信したデータを収集し整理し、実際のフラッシュへの書き込みを、単純なコマンド/ステータスレジスタ インターフェースを通して管理します。ブートローダ コンポーネントは、典型的なコンポーネントではありません。コンポーネントカタログに記載されていません。しかし、ブートローダタイプのプロジェクトにおいては、裏に常に存在しています。

以下の図は、ブートローダがどのように動作するかを示しています。



通信コンポーネント

通信コンポーネントは、通信プロトコルを管理し、外部システムからコマンドを受け取り、これらのコマンドをブートローダに渡します。また、ブートローダからオフチップシステムへコマンドのレスポンスを返します。

注 ブートローダがサポートする通信方法は、USB および I^2C に限られます。適切な通信方法についての詳細に関しては、USBFS もしくは I^2C コンポーネントデータシートを参照してください。ブートホルダサポートを既存の通信コンポーネントに追加するカスタムオプションもあります。「カスタムブートホルダコンポーネント」 を参照してください。

また、どのような通信方法に対しても、独自のブートローダコンポーネントを自作することが可能です。自作についての情報および手順については、*Component Author Guide (コンポーネント作成者ガイド)*を参照してください。

カスタムブートホルダコンポーネント

任意の希望する通信コンポーネントを使用して、カスタムブートローダコンポーネントを作成することができます。DWR システムエディタ「IO このポーネント」の下で、「Custom_Interface」オプションを選択します。IO コンポーネント (IO Component)パラメータも参照してください。これによりアプリケーションコードに必要な任意の必要な方法で、必要な機能を実装することができます。次には、SPI 通信コンポーネント用に *main.c* ファイルに挿入されたコードの例を示します。CyBtldr_ API の詳細は *Component Author Guide (コンポーネント作者ガイド)*を参照してください。

```
void CyBtldrCommStart(void)
{
    SPIS_1_Start();
}

void CyBtldrCommStop (void)
{
    SPIS_1_Stop();
}

void CyBtldrCommReset(void)
{
}

cystatus CyBtldrCommWrite(uint8* buffer, uint16 size, uint16* count, uint8
timeOut)
{
    uint16 i;
    cystatus status = CYRET_EMPTY;

    uint8 intStatus = CyEnterCriticalSection();

    SPIS_1_ClearRxBuffer();
    SPIS_1_ClearTxBuffer();

    for (i = 0; i < size; i++)
    {
        SPIS_1_WriteTxData(buffer[i]);
    }

    CyExitCriticalSection(intStatus);

    while (timeOut-- > 0)
    {
        if ((SPIS_1_ReadTxStatus() & SPIS_1_STS_SPI_DONE) !=
           SPIS_1_STS_SPI_DONE)
        {
            *count = size;
            status = CYRET_SUCCESS;
            break;
        }

        CyDelay(10);
    }
```

```
    return status;
}

cystatus CyBtldrCommRead (uint8* buffer, uint16 size, uint16* count, uint8
timeOut)
{
    uint16 i = 0;
    cystatus status = CYRET_EMPTY;

    uint8 validData = 0;
    uint8 dataByte;

    while (timeOut-- > 0)
    {
        if (SPIS_1_GetRxBufferSize() > 0 && SPIS_1_GetTxBufferSize() == 0)
        {
            while (!validData && SPIS_1_GetRxBufferSize() > 0)
            {
                dataByte = SPIS_1_ReadRxData();

                validData = (1 == dataByte);
            }

            if (validData)
            {
                buffer[0] = dataByte;
                i = 1;
            }

            CyDelay(10);
            while (SPIS_1_GetRxBufferSize() > 0)
            {
                buffer[i++] = SPIS_1_ReadRxData();
            }

            if (i > 0)
            {
                while(buffer[--i] != 0x17);
                *カウント = i+1;
                status = CYRET_SUCCESS;
                break;
            }
        }
        CyDelay(10);
    }
    return status;
}
```

ブートローダ プロジェクトのタイプ

ブートローダ コンポーネントとコードの双方を実装するには、特殊な PSoC Creator プロジェクトのタイプである、ブートローダ プロジェクトと、ブートロード可能な（bootloadable）プロジェクトを作成する必要があります。

ブートローダ プロジェクト

ブートローダ コンポーネントは、ブートローダ型の PSoC Creator デザインプロジェクト内にのみ存在します。ブートローダ プロジェクトを作成すると、自動的にブートローダ コンポーネントがプロジェクト内に作成されます。

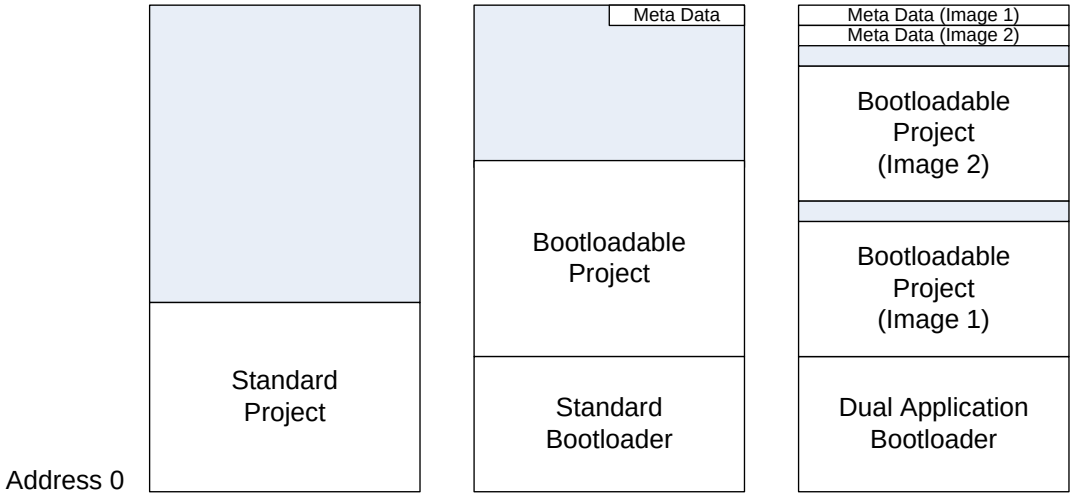
「Standard Bootloader（標準ブートローダ）」および「Dual Application Bootloader（デュアルアプリケーションブートローダ）」という 2 つのタイプのブートローダーが利用できます。「Standard Bootloader」オプションは、シングルアプリケーション コードが可能であり、「Dual Application Bootloader」は、フラッシュに 2 つのアプリケーションを可能にします。「Dual Application Bootloader」は、実行できる有効なアプリケーションが必ずあるという保証が必要な設計に役に立ちます。この保証には制限があり、各アプリケーションが「標準ブートローダ」プロジェクトで利用可能だったであろう、利用可能なフラッシュの半分を持ちます。

ブートローダデザインプロジェクトを完成させるには、一般的には回路図に通信コンポーネントをドラッグし、I/O をピンに繋ぎ、クロックをセットアップする、などの手順を行います。通信コンポーネントが存在するブートローダプロジェクトは、新しいコードを受け取りフラッシュに書き込む、基本的なブートローダ関数を実装します。基本的なブートローダプロジェクトにカスタム関数を追加するには、回路図に他のコンポーネントをドラッグするか、ソースコードを追加してください。

ブートロード可能なプロジェクト

ブートロード可能な（ bootloadable）プロジェクトが、実際のコードです。これは、「標準的な」デザインプロジェクトに酷似しているため、プロジェクトの型をデザインフェーズ中に簡単に切り替えることができます。主要な違いは、ブートロード可能なプロジェクトは、常にブートローダプロジェクトと関連付けられているということで、標準プロジェクトはブートローダプロジェクトと関連付けられることは絶対にありません。

また、標準的なプロジェクトは、フラッシュ内のアドレス 0 から存在します。しかし、ブートロード可能なプロジェクトは 0 より大きいアドレスからフラッシュメモリを占有します。関連付けられたブートローダプロジェクトがフラッシュ内のアドレス 0 からの領域を占有します。(以下を参照してください)



ブートローダおよびブートロード可能なプロジェクトの関数

ブートローダプロジェクトは、ブートローダプロジェクトの通信コンポーネントを通して、ブートロード可能なプロジェクト全体もしくは新しいコードを、フラッシュに転送します。転送後には、プロセッサは常にリセットされます。ブートローダプロジェクトは、リセット時に特定の条件をテストし、ブートロード可能なプロジェクトが存在しないか壊れている場合に、転送を自動的に開始する責任があります。

スタートアップ時に、ブートローダプロジェクトは、自分の設定用のコンフィギュレーションバイトを読み込みます。また、転送用にスタック等のリソースと周辺機器を初期化します。転送完了後に、ソフトウェアリセットを通して、制御をブートロード可能なプロジェクトに移します。

その後、ブートロード可能なプロジェクトは、自分の設定用のコンフィギュレーションバイトを読み込み、その機能用にスタック等のリソースと周辺機器を初期化します。ブートロード可能なプロジェクトは、ブートローダプロジェクト内の CyBtldr_Load()関数を呼び出し、転送を開始することができます。この場合、もう一度ソフトウェアリセットが行われます。

PSoC Creator プロジェクト出力ファイル

ブートローダまたはブートロード可能なプロジェクトのいずれかのプロジェクト型がビルドされた場合、そのプロジェクトに対し出力ファイルが作成されます。

さらに、ブートロード可能なプロジェクトがビルドされた際に、2 つのプロジェクト用の出力ファイル(組み合わせファイル)が作成されます。このファイルは、ブートローダおよびブートロード可能なプロジェクトを両方含みます。このファイルは、製品において、2 つのプロジェクトを、JTAG または SWD を経由して、フラッシュに容易にダウンロードするために使用されます。

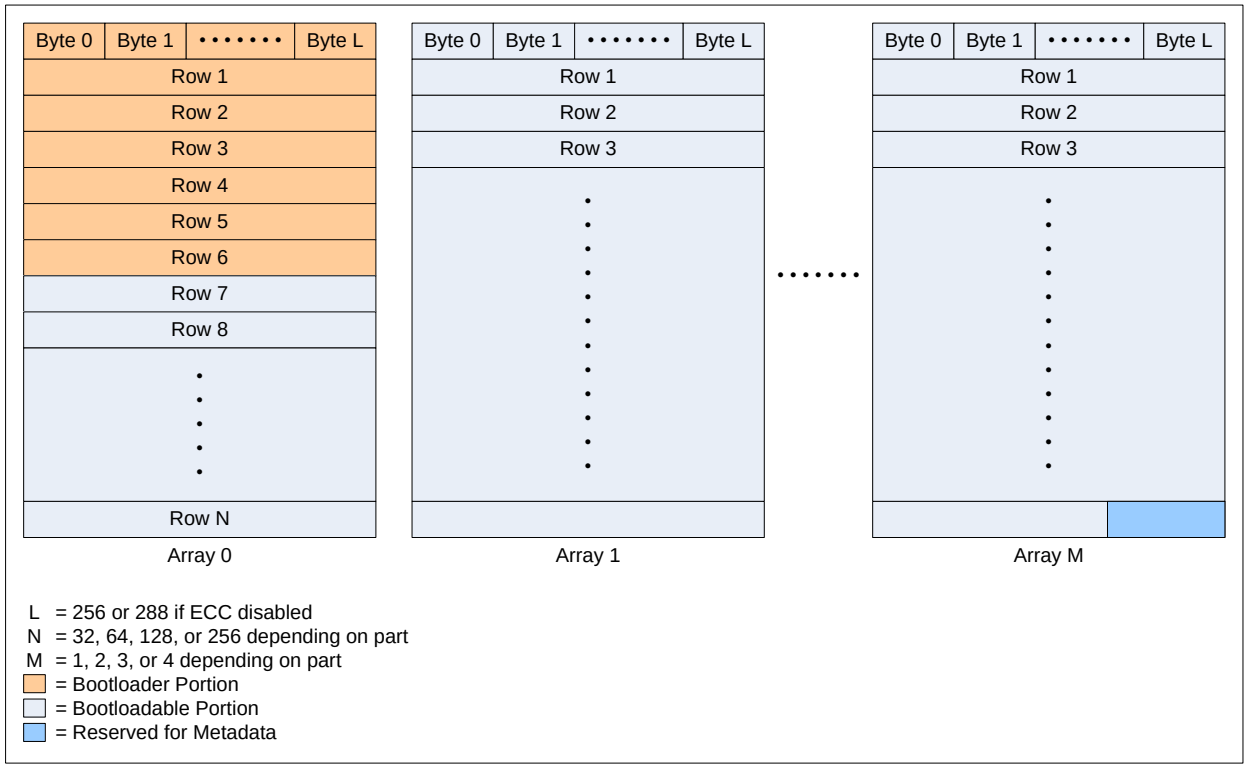
ブートローダプロジェクトにおいては、コンフィギュレーションバイトはブートローダが占有するメインフラッシュに保存され、ECC フラッシュには絶対に保存されません。

ブートロード可能なプロジェクトのコンフィギュレーションバイトは、主フラッシュおよび ECC フラッシュのどちらにも保存することができます。ブートロード可能なプロジェクトの出力ファイルのフォーマットは、デバイスの ECC バイトが無効化されている場合、転送操作がより短時間で終了するようになっています。これは、ブートロード可能な主フラッシュのアドレス空間のレコードと、ECC フラッシュのアドレス空間のレコードを交互配置することで可能になります。ブートローダは、この交互配置構造を利用し、関連するフラッシュ列ーこの列は、主フラッシュおよび ECC フラッシュのバイトを両方含みますーを 1 度でプログラムします。

各プロジェクトは、それぞれ独自のチェックサムを保有します。チェックサムは、プロジェクトのビルドタイムの出力ファイルに含まれます。

メモリの使用

次の図は、PSoC 3 と PSoC 5 のデバイスのフラッシュメモリーレイアウトを表示します。



ブートローダプロジェクトは、常にフラッシュの下位 256 バイトブロックを N 個占有します。N は、以下の用途のためのフラッシュが十分確保されるように設定されます:

- アドレス 0 から始まる、このプロジェクトのベクタテーブル(PSoC 3 以外)、
- ブートローダプロジェクトのコンフィギュレーションバイト、
- ブートローダプロジェクトのコードとデータ、および
- フラッシュのブートローダ部分のチェックサム。

ブートローダプロジェクトのコンフィギュレーションバイトは必ず主フラッシュに保存され、絶対に ECC フラッシュに保存されないことに留意してください。これに関連するオプションは、プロジェクトの .cydwr ファイルから削除されます。

フラッシュのブートローダ部分は保護されており、JTAG または SWD を経由したダウンロードでのみ上書きされます。

ブートロード可能なプロジェクトは、ブートローダの次にある最初の 256 バイト区切りからフラッシュを占有し、以下を含みます。

- このプロジェクトのベクタテーブル(PSoC 3 以外)、
- ブートロード可能なプロジェクトのコードとデータ、および
- ブートローダおよびブートロード可能の両者により使用される、メタデータを保存するために使用される、最後のフラッシュメモリアレイの一番最後の 64 バイトの予約領域のデータ。

ブートロード可能なプロジェクトのコンフィギュレーションバイトは、標準的なプロジェクトと同じ方法、すなわちプロジェクトの.cydwr ファイルの設定に応じて主フラッシュまたは ECC フラッシュのいずれかに保存されます。

フラッシュの最高位の 64 バイトブロックは、2 つのプロジェクトの共同利用領域として使用されます。さまざまなパラメータがこのブロックに保存され、次が含まれます。

- ブートロード可能なプロジェクトの、フラッシュ内のエントリ(4 バイトのアドレス)
- ブートロード可能なプロジェクトが占有するフラッシュの量 (フラッシュ列数)
- フラッシュの、ブートロード可能なプロジェクトの部分のチェックサム(1 バイト)
- フラッシュの、ブートロード可能な部分のサイズ (4 バイト、バイト単位)

8051 詳細 (PSoC 3)

PSoC 3 において、唯一の「例外ベクタ」は、プロセッサリセット時に実行される、アドレス 0 にある 3 バイトの指示だけです。(インタラプトベクタはフラッシュ内にありません。インタラプトコントローラ (IC) により提供されます。) このため、リセット時には、8051 ブートローダコードは、単にフラッシュのアドレス 0 から実行を始めます。

ARM Cortex-M3 の詳細 (PSoC 5)

PSoC 5 においては、アドレス 0 に例外ベクタのテーブルが存在する必要があります。(このテーブルは、アドレス 0xE000ED08 にある、ベクタテーブル オフセットレジスタ(Vector Table Offset Register)によりポイントされてます。リセット時に、値が 0 に設定されます。) ブートローダのコードは、このテーブルの直後から始まります。

このテーブルは、ブートローダプロジェクトの初期スタックポインタ (SP) 値、およびブートローダプロジェクトのコードの開始アドレスを含みます。また、ブートローダが使用する、例外および割り込みのベクタも含みます。

ブートロード可能なプロジェクトは独自のベクタテーブルを保有します。これには、プロジェクトの開始 SP 値および最初の指示のアドレスが含まれます。転送が完了した後、ブートロード可能なプロジェクトへコントロールを渡す一環として、ベクタテーブル オフセットレジスタ(Vector Table Offset Register)内の値が、ブートロード可能なプロジェクトのテーブルのアドレスに変更されます。

ブートローダのパラメータ

ブートローダのパラメータにアクセスするには、DWR System Editor を開き、エディタの[ブートローダ]セクションを展開してください。

コマンド入力までウェイト(Wait for Command)

説明: リセット時に、ブートローダは、ブートロード可能なプロジェクトのフラッシュのチェックサムが有効であると検知した後に、ブートロード可能なプロジェクトのコードをすぐに実行せずに、転送開始コマンドが入力されるまでウェイトすることができます。

設定: ウェイトが行われるか、[Yes]または[No]を設定します。

デフォルト: あり

API もしくはドライバ ファームウェアにより変更可能か: 不可

他のパラメータとの関連性 [Yes]が選択された場合、コマンド入力までのウェイト時間(Wait for Command Time)パラメータが編集可能になります。[No]が選択された場合、このパラメータは灰色で表示され、無効になります。この場合、外部システムは一般的に転送を開始することができませんが、ブートロード可能なプロジェクトのコードは、Bootloader_Start() を呼び出すことで転送を開始できます。

コマンド入力までのウェイト時間(Wait for Command Time)

説明: リセット時に、ブートローダは、ブートロード可能なプロジェクトのフラッシュのチェックサムが有効であると検知した後に、ブートロード可能なプロジェクトのコードをすぐに実行せずに、転送開始コマンドが入力されるまでウェイトすることができます。このパラメータは、ウェイトのタイムアウト時間です。

設定: 1 – 255 (単位:10ミリ秒)。

デフォルト: 10 (100ミリ秒)。

API もしくはドライバ ファームウェアにより変更可能か: 不可

他のパラメータとの関連性 このパラメータは、コマンド入力までウェイト(Wait for Command)パラメータが有効のときのみ編集可能になります。それ以外の場合、このパラメータは灰色で表示され、無効になります。

IO コンポーネント (IO Component)

説明: これは、ブートローダが、コマンドを受け取り、応答を送信するために使用する、通信コンポーネントです。通信コンポーネントを1つのみ選択することが必要です。双方向通信コンポーネントのみが使用されています。例えば、UART は RX および TX の両者がイネーブルでなくてはならず、赤外線 (IrDA) コンポーネントは使用することができませんでした。双方向通信コンポーネントがブートローダプロジェクトの回路図に存在しない場合を検知するため、デザインルールチェック(DRC)が存在します。

設定: このプロパティは、ブートローダにサポートされる、使用可能なIO通信プロトコルのリストです。. いずれの場合においても、回路図の内容に関わらず、カスタムオプションがあり、ブートローダの関数を直接実装することができます。

デフォルト: 回路図に通信用コンポーネントがない場合、カスタムオプションが選択されます。これにより、通信をあらゆる方法で実装することができます。

API もしくはドライバ ファームウェアにより変更可能か: 不可

他のパラメータとの関連性 なし。

高速アプリケーション検証

説明: イネーブルの場合、ブートローダはアプリケーションコードのチェックサムを計算します。成功すると、将来のブート操作のためにこの情報を保存し、起動するたびにアプリケーションコードを検証する必要がなくなります。

設定: 検証が記憶されているか（はい、いいえ）。

デフォルト: 不可

API もしくはドライバ ファームウェアにより変更可能か: 不可

他のパラメータとの関連性 なし。

チェックサムのタイプ

説明: ホストとブートローダとの間でデータパケットを転送するとき使用するチェックサムのタイプについて、いくつかのオプションを提供します。

設定: Basic Sum（基本的な加算）はすべてのバイトを加算し、2 つのバックアップにします。CCITT アルゴリズムを使用した、16 ビット CRC。

デフォルト: Basic Sum（基本的な加算）。

API もしくはドライバ ファームウェアにより変更可能か: 不可

他のパラメータとの関連性 なし。

バージョン

説明: ブートローダのバージョンを表す 2 バイトの番号を提供します。

設定: 任意の 2 バイトの数字。

デフォルト: 0x0000

API もしくはドライバ ファームウェアにより変更可能か: 不可

他のパラメータとの関連性 なし。

Bootloadable Parameters（ブートロード可能なパラメータ）

バージョン

説明: ブートローダ可能なアプリケーションのバージョンを表す 2 バイトの番号を提供します。

設定: 任意の 2 バイトの数字

デフォルト: 0x0000

API もしくはドライバ ファームウェアにより変更可能か: 不可

他のパラメータとの関連性 なし。

Bootloadable ID（ブートロード可能な ID）

説明: ブートローダ可能なアプリケーションの ID を表す 2 バイトの番号を提供します。

設定: 任意の 2 バイトの数字。

デフォルト: 0x0000

API もしくはドライバ ファームウェアにより変更可能か: 不可

他のパラメータとの関連性 なし

Custom ID（カスタム ID）

説明: アプリケーションの任意の項目を表す、4 バイトのカスタム ID 番号を提供します。

設定: 任意の 4 バイトの数字。

デフォルト: 0x00000000

API もしくはドライバ ファームウェアにより変更可能か: 不可

他のパラメータとの関連性 なし。

ブートローダの API

ブートローダは、ブートロード可能なプロジェクトのコードから転送操作を開始するだけの、パブリックな API 呼び出しを提供します。呼び出された場合、ソフトウェアリセットが実行され、ブートローダが CPU を制御します。インタラプト ハンドラを含む、ブートロード可能なプロジェクトコードは実行されません。

転送操作が開始される際には、リソースおよび周辺機器は、必要に応じて再設定されます。他のすべてのリソースおよびペリフェラルはディスエーブルされます。

転送操作が終了すると、CPU はリセットされます。

void CyBtldr_Load(void)

説明: 転送操作を開始します。ブートローダプロジェクトの設定に従い、デバイスを再設定します。

パラメータ: void

返り値: なし。プロセッサは、転送操作が終了するとリセットされます。

副作用: なし

ブートローダの コマンド

ブートローダは、以下のコマンドをサポートします。コマンドバイトのいずれかからも開始されない、全ての受信バイトは廃棄され、応答は生成されません。全てのマルチバイトのフィールドは、LSB から出力されます。

注 ブートローダが任意のコマンドを実行するために必要な時間は、デバイスの構成に基づいています。タイミングに影響を及ぼすいくつかの要素を次に挙げます。

- パーツが実行されているクロックスピード
- プロジェクトを構築するために使用されているツールチェーン
- ビルド中に使用されている最適化設定
- バックグラウンドで実行されているインタラプトの数

Enter Bootloader（ブートローダに入る）

他の全てのコマンドは、このコマンドを受信するまで無視されます。

Input

- コマンドバイト: 0x38
- データバイト: なし

出力

- ステータスコードおよびエラーコード:
  - 成功(Success)
  - エラー コマンド(Error Command)
  - エラー データ(Error Data)
  - エラー 長さ(Error Length)

- エラー チェックサム(Error Checksum)

- データバイト:
  - 4 バイト - シリコン ID
  - 1 バイト - シリコンの版
  - 3 バイト - ブートローダのバージョン

Get Flash Size（フラッシュサイズの取得)

選択されたフラッシュアレイの利用可能な列のうち、最初と最後の列を応答します。

Input

- コマンドバイト: 0x32
- データバイト:
  - 1 バイト - フラッシュ アレイ ID

出力

- ステータスコードおよびエラーコード:
  - 成功(Success)
  - エラー コマンド(Error Command)
  - エラー データ(Error Data)
  - エラー 長さ(Error Length)
  - エラー チェックサム(Error Checksum)
- データバイト:
  - 2 バイト - 利用可能な最初の列
  - 2 バイト - 利用可能な最後の列

Program Row（列をプログラム)

デバイスにフラッシュデータを 1 列書き込みます。

Input

- コマンドバイト: 0x39
- データバイト:
  - 1 バイト - フラッシュ アレイ ID
  - 2 バイト - フラッシュの列の番号
  - n バイト - フラッシュの列に書き込むデータ

出力

- ステータスコードおよびエラーコード:
  - 成功(Success)
  - エラー コマンド(Error Command)
  - エラー データ(Error Data)

- エラー 長さ(Error Length)
- エラー チェックサム(Error Checksum)
- エラー フラッシュ列(Error Flash Row)
- エラー アクティブ (Error Active)

- データバイト: なし

Erase Row（列の削除）

提供されたフラッシュの列の内容を削除します。

Input

- コマンドバイト: 0x34
- データバイト:
  - 1 バイト - フラッシュ アレイ ID
  - 2 バイト - フラッシュの列の番号

出力

- ステータスコードおよびエラーコード:
  - 成功(Success)
  - エラー コマンド(Error Command)
  - エラー データ(Error Data)
  - エラー 長さ(Error Length)
  - エラー チェックサム(Error Checksum)
  - エラー フラッシュ列(Error Flash Row)
  - エラー アクティブ (Error Active)
- データバイト: なし

Verify Row（列の検証)

提供されたフラッシュの列の内容に対する、1 バイトのチェックサムを取得します

Input

- コマンドバイト: 0x3A
- データバイト:
  - 1 バイト - フラッシュ アレイ ID
  - 2 バイト - フラッシュの列の番号

出力

- ステータスコードおよびエラーコード:
  - 成功(Success)
  - エラー コマンド(Error Command)
  - エラー データ(Error Data)

- エラー 長さ(Error Length)
- エラー チェックサム(Error Checksum)

- データバイト:
  - 1 バイト - 列のチェックサム

Verify Checksum（チェックサムの検証）

フラッシュのチェックサムが、想定されているチェックサムの値と等しいか否かを示す、1 バイトの値を取得します。返り値 1 は、チェックサムが合致し、アプリケーションが有効とされていることを示します。返り値 0 は、チェックサムが合致せず、アプリケーションが無効であることを示します。これは、ブートローダがアプリケーション コードを実行する前に行うチェックと、同じものです。

Input

- コマンドバイト: 0x31
- データバイト: 該当なし

出力

- ステータスコードおよびエラーコード:
  - 成功(Success)
  - エラー コマンド(Error Command)
  - エラー データ(Error Data)
  - エラー 長さ(Error Length)
  - エラー チェックサム(Error Checksum)
- データバイト:
  - 1 バイト - アプリケーションのチェックサムが有効

Send Data(データの送信）

デバイスにデータのブロックを送信します。ブートローダが、データをどう処理するかを指示するコマンドを受信するまで、受信されたデータはバッファに蓄えられます。複数のデータの送信 (Send Data)コマンドが連続して実行された場合、新たなデータは以前のデータの後に追加されます。このコマンドは、一部のプロトコルでのバス欠乏を防ぐため、大きいデータの転送を、複数の小さいデータに分割するために用いられます。

Input

- コマンドバイト: 0x37
- データ バイト: n バイト – バッファに保存されるデータ

出力

- ステータスコードおよびエラーコード:
  - 成功(Success)
  - エラー コマンド(Error Command)
  - エラー データ(Error Data)
  - エラー 長さ(Error Length)

- エラー チェックサム(Error Checksum)

- データバイト: なし

Sync bootloader（ブートローダの同期）

ブートローダをリセットし、新しいコマンドを受け付けることができる、クリーンな状態にします。バッファ内のデータは、全て廃棄されます。これは、ホストとクライアント間の同期が失われた場合にのみ必要です。

Input

- コマンドバイト: 0x35
- データバイト: なし

出力

- NA – このパケットは受け取られませんでした。

Exit Bootloader（ブートローダの終了）

デバイスのソフトウェア リセットをトリガして、ブートローダを終了します。ソフトウェア リセットを実行する前に、ブートロード可能なアプリケーションが検証されます。アプリケーションが検証に合格すると、ソフトウェアのリセット後に、アプリケーションが実行されます。アプリケーションが検証に合格しない場合、ソフトウェアのリセット後に、ブートローダによって再度実行が開始されます。

Input

- コマンドバイト: 0x3B
- データバイト: なし

出力

- NA – このパケットは受け取られませんでした

Get Application Status（アプリケーション ステータスを取得する (デュアル アプリケーション ブートローダのみ)）

Input

- コマンドバイト: 0x33
- データバイト:
  - 1 バイト – アプリケーション #

出力

- ステータスコードおよびエラーコード:
  - 成功(Success)
  - エラー 長さ(Error Length)
  - エラー チェックサム(Error Checksum)
  - エラー データ(Error Data)
- データバイト:

- 1 バイト – アプリケーション # 有効
- 1 バイト – アプリケーション # アクティブ

Set Active Application（アクティブ アプリケーションを設定する (デュアル アプリケーション ブートローダのみ)）

Input

- コマンドバイト: 0x36
- データバイト:
  - 1 バイト – アプリケーション #

出力

- ステータスコードおよびエラーコード:
  - 成功(Success)
  - エラー アプリケーション
  - エラー 長さ(Error Length)
  - エラー データ(Error Data)
  - エラー チェックサム(Error Checksum)
- データバイト: なし

ブートローダのパケット

ブートローダに送信されるパケットは、以下の構造をとります:

- 1 バイト パケット開始 (0x01)
- 1 バイト コマンド
- 2 バイト データ長
- n バイト データ
- 2 バイト チェックサム
- 1 バイト パケット終了 (0x17)

ブートローダから出力されるパケットは、以下の構造をとります:

- 1 バイト パケット開始 (0x01)
- 1 バイト ステータスコードまたはエラーコード
- 2 バイト データ長
- n バイト データ
- 2 バイト チェックサム
- 1 バイト パケット終了 (0x17)

ブートローダの ステータスコードおよびエラーコード

ブートローダから出力される、ステータスコードおよびエラーコードは以下のとおりです。

ブートローダのアプリケーション

- Success – コマンドは正常に受信され、実行されました。値 = 0x00
- Error Length (CYRET_ERR_LENGTH) – 利用可能なデータの量が、予想範囲外です。値 = 0x03
- Error Data (CYRET_ERR_DATA) – データは適切な形式ではありません。値 = 0x04
- Error Command (CYRET_ERR_CMD) – コマンドが認識されませんでした。値 = 0x05
- Error Checksum (ERR_CHECKSUM) – チェックサムは期待される値と一致しません。値 = 0x08
- Error Flash Row (ERR_ROW) – フラッシュ列は有効ではありません。Value = 0x0A
- Error Unknown(未知のエラー)(ERR_UNK) – 未知のエラーが発生しました。Value = 0x0F
- Error Application (CYRET_ERR_APP) – アプリケーションは有効ではなく、アクティブに設定することができません。値 = 0x0C。(デュアル アプリケーション ブートローダのみ)
- Error Active (CYRET_ERR_ACTIVE) – アプリケーションは現在アクティブとしてマークされています。値 = 0x0D。(デュアル アプリケーション ブートローダのみ)

ブートローダ ホスト

- Error Device (CYRET_ERR_DEVICE) – 期待されるデバイスは検出されたデバイスと一致しません。値 = 0x06
- Error Version (CYRET_ERR_VERSION) – 検出されたブートローダ バージョンはサポートされていません。値 = 0x07

ブートローダのアプリケーションとコードデータのファイルフォーマット

ブートローダのアプリケーションとコードデータ(.cyacd)のファイルフォーマットは、デザインのブートロード可能な部分を保存するのに使用されます。このファイルには、ヘッダがあり、その後にフラッシュデータの行が続きます。ヘッダを除いた、.cyacd ファイルの各行は、それぞれフラッシュデータ 1 列分のデータです。データは、ビッグエンディアン形式の ASCII データとして保存されます。

ヘッダレコードの形式は以下のものです:

```
[4-byte SiliconID][1-byte SiliconRev][1-byte Checksum Type]
```

データレコードの形式は以下のものです:

```
[1 バイト ArrayID][2 バイト RowNumber][2 バイト DataLength][N バイト データ]
[1 バイト チェックサム]
```

ヘッダーのチェックサム タイプは、ブートローダ ホストとブートローダ自身との間で送信されたパケットに使用されるチェックサムのタイプを示します。データ レコードのチェックサムは基本的な加算であり、チェックサム自身を除く全てのバイトの和を取ったものの、2 の補数です。

ブートローダ ホストツール

PSoC Creator には、PSoC チップ上で実行されているブートローダのテストに使用する、ブートローダ ホストツール (*bootloader_host.exe*) が同梱されています。ブートローダ ホストツールは、ブートローダ自身と通信し、新しいブートロード可能なイメージを送信するアプリケーションです。付属のブートローダ ホスト ツールは、開発およびテスト用のツールとしてのみ使用します。

ソースコード

実行可能なホストに加えて、使用されるソースコードの大半が提供されています。このソースコードを再利用して、ブートローダ ホストアプリケーションを自作することが可能です。ソースコードは以下のディレクトリにあります:

<Install Dir>\cybootloaderutils

デフォルトでは、このディレクトリは:

C:\Program Files\Cypress\PSoC Creator\<Release Version>\PSoC Creator\cybootloaderutils

このソースコードは、4 つの異なるモジュールに分割されています。これらのモジュールは、ブートローダ ホストが必要とする、様々な機能をそれぞれ実装します。必要な制御のレベルに応じて、これらのモジュールの一部もしくは全てを使用して、カスタム ブートローダ ホスト アプリケーションを作成することができます。

cybtldr_command.c/h

このモジュールは、ブートローダに送信するパケットの作成、およびブートローダから受信したパケットを解析します。ブートローダが理解する各タイプのパケットを作成する関数が 1 つ、および ブートローダが送信できるパケットを解析する関数が 1 つ含まれています。

cybtldr_parse.c/h

このモジュールは、デバイスに送信する、ブートロード可能なイメージを含む、*.cyacd ファイルのパースを行います。ファイルへのアクセスを設定する、ヘッダを読み込む、列データを読み込む、およびファイルを閉じるための関数が含まれています。

cybtldr_api.c/h

提供された通信メカニズムを用いて、ブートローダにデータを 1 列ずつ送信するための、列レベルの API です。ブートロード操作の設定、列のプログラミング、列の消去、列の検証、およびブートロード操作の終了を行う関数が含まれています。

cybtldr_api2.c/h

ブートロードのプロセス全体を扱う、ハイレベルの API です。デバイスのプログラミング、デバイスの消去、デバイスの検証、および現在の操作の中止を行う関数が含まれています。

バージョン

以下は、ブートローダ ホストツールの、各バージョンが提供される機能です:

バージョン 1.0.0 (PSoC Creator 1.0, Beta5)

初期バージョン; 以下の用途向けの API が提供されます:

- *.cyacd ファイルのパース
- ブートローダに送信する、データパケットの作成

- 複数列にわたるデータの、プログラム、消去、および検証を行う関数
- ブートロード操作全体を行う関数

バージョン 1.1.0 (PSoC Creator 1.0, 製品版(Production))

ブートローダ ホスト バージョン 1.0.0 に含まれているすべての機能を提供します。さらに、ブートローダとの通信時にパケットの整合性を確保するための、単純和 (1.0.0 で使用) 、または新たな 16 ビット CCITT チェックサムの使用を、追加してサポートします。

バージョン 1.2.0 (PSoC Creator 2.0)

ブートローダ ホスト バージョン 1.0.0 に含まれているすべての機能を提供します。「標準ブートローダ」または「デュアル アプリケーション ブートローダ」のいずれかとの通信のサポートが追加されます。

12 システム関数

これらの関数は、すべてのアーキテクチャに適用されます。

汎用 API

uint8 CyEnterCriticalSection(void)

説明: CyEnterCriticalSectionは、インタラプトを無効化し、以前にインタラプトが有効化されていたか否かを示す値を返します(戻り値の値は、デバイスがPSoC 3の場合とPSoC 5の場合で異なります)。

注 CyEnterCriticalSection の実装では、割り込みが有効な状態のまま、IRQ イネーブル ビットを操作します。割り込みビットのテスト＆セット操作は、PSoC 3 および PSoC 5 のいずれに対しても、アトミックではありません。(訳注:割込みビットの操作中に割り込みが入る可能性があります) このため、プロセッサステートが壊れるのを防ぐため、全ての割り込みルーチンは、割り込みイネーブル ビットを割り込み前の状態に戻すように、設計する必要があります。

パラメータ: なし

返り値: uint8
PSoC 3 – ビットを2つ含む値を返します:
ビット0: CyEnterCriticalSection が呼び出される前に、割り込みが有効であった場合は、1。
ビット1: CyEnterCriticalSection が呼び出される前に 、IRQ 生成が無効であった場合は、1。
PSoC 5 – インタラプトが以前に有効だった場合は1、無効だった場合は0。

void CyExitCriticalSection(uint8 savedIntrStatus)

説明: CyExitCriticalSectionは、CyEnterCriticalSectionの呼び出し前にインタラプトが有効であった場合、再度有効にします。引数は、CyEnterCriticalSection の返り値の必要があります。

パラメータ: uint8 savedIntrStatus: CyEnterCriticalSection 関数により返された、保存された割り込みステータスです。

返り値: なし

void CYASSERT(uint32 expr)

説明: 式を評価し、偽の場合(0と評価された場合)プロセッサを停止するマクロです。このマクロは、NDEBUGが定義されていない場合に評価されます。NDEBUG が定義されている場合、このマクロに対するコードは生成されません。デフォルトでは、NDEBUG はリリース ビルド設定に対して定義され、デバッグ ビルド設定に対しては定義されません。

パラメータ: expr: ロジカルな式です。偽の場合、アサートします。

返り値: なし

void CyHalt(uint8 reason)

説明: CPUを停止します。

パラメータ: reason: デバッグするために渡す値。この値は CyHalt() が呼び出された理由を知るために役に立つことがあります。

返り値: なし

void CySoftwareReset(void)

説明: デバイスを強制的にソフトウェアリセットします。

パラメータ: なし

返り値: なし

CyDelay APIs

単純なソフトウェアベースのディレイループを実装する 4 つの CyDelay API があります。ループはバス クロック周波数を補償します。

CyDelay 関数は最小限の遅延を提供します。プロセッサに割り込みが入ると、ループの長さは、割り込みを実装するために必要なだけ延長されます。関数の入力と終了など、その他のオーバーヘッド要因も、この関数を実行するために費やされる合計時間に影響を与えることがあります。これは、名目上の遅延時間が短い場合、特に顕著です。

void CyDelay(uint32 milliseconds)

説明: 指定されたミリ秒の間、遅延します。デフォルトにより、遅延するサイクル数は、PSoC Creator に入力されたクロック設定により計算されます。クロックの設定がランタイムに変更された場合、新しいバスクロック周波数を求めるために、CyDelayFreq関数を使用します。CyDelay は複数のコンポーネントにより使用されるため、遅延のための周波数設定を更新することなくクロック周波数を変更することは、これらのコンポーネントに支障が発生する原因になることがあります。

パラメータ: milliseconds: 遅延するミリ秒数。

返り値: なし

副作用と制限: CyDelayは、命令キャッシュ(instruction cache)が有効なものとして実装されています。PSoC 5において命令キャッシュ(instruction cache)が無効の場合、CyDelayによる遅延時間は2倍になります。例えば、命令キャッシュ (instruction cache) が無効の場合、CyDelay(100) による遅延は 100 ミリ秒ではなく、約 200 ミリ秒になります。

void CyDelayUs(uint16 microseconds)

説明: 指定されたマイクロ秒の間、遅延します。デフォルトにより、遅延するサイクル数は、PSoC Creatorに入力されたクロック設定により計算されます。クロックの設定がランタイムに変更された場合、新しいバスクロック周波数を求めるために、CyDelayFreq関数を使用します。CyDelayUs は複数のコンポーネントにより使用されるため、遅延のための周波数設定を更新することなくクロック周波数を変更することは、これらのコンポーネントに支障が発生する原因になることがあります。

パラメータ: microseconds: 遅延するマイクロ秒数。

返り値: void

副作用と制限: CyDelayUSは、命令キャッシュ(instruction cache)が有効なものとして実装されています。PSoC 5において命令キャッシュ(instruction cache)が無効の場合、CyDelayUsによる遅延時間は2倍になります。例えば、命令キャッシュ (instruction cache) が無効の場合、CyDelayUs(100) による遅延は 100 マイクロ秒ではなく、約 200 マイクロ秒になります。

void CyDelayFreq(uint32 freq)

説明: CyDelayによる遅延を実装するために必要な、サイクル数を計算するためのバスクロック周波数を設定します。デフォルトでは、使用される周波数の値は、PSoC Creator によりビルド時に決定された値に基づきます。

パラメータ: freq: バス クロックの周波数 (Hz単位)。

0: デフォルトの値を使用してください。

0以外: 周波数の値を設定してください。

返り値: なし

void CyDelayCycles(uint32 cycles)

説明: ソフトウェアディレイループを使用して、指定されたサイクル数だけ遅延します。

パラメータ: cycles: 遅延するサイクル数。

返り値: なし

13 起動とリンク

システムの起動は、cy_boot コンポーネントが行います。以下の機能が実装されています。

- リセットベクタの提供
- 実行できるようにプロセッサをセットアップ
- 割り込みのセットアップ
- 8051 用の再入可能スタックも含む、スタックのセットアップ
- デバイスの設定
- 静的およびグローバル変数を、初期値で初期化する
- 残っている静的およびグローバル変数をクリアする
- ブートローダと統合する
- main() C エントリポイントを呼び出す

PSoC 3

起動は、Keil により提供されたテンプレートに基づいた、単一のアセンブリファイル (KeilStart.a51)により扱われます。特にリンクに関連付けられたファイルはありません。

PSoC 5

起動およびリンカ スクリプトは、サイプレスが独自に開発しましたが、現在当社がサポートしているツールチェーン ベンダが、当社独自の実装によって発生した多くの問題点を解消する、リンカ実装例および完全なライブラリを提供しています。

GCC の実装

標準 GCC ライブラリ(libcs3, libc, libcs3unhosted, libgcc, libcs3micro)をすべて使用してください。これらのライブラリはすべて、初期設定でリンクされています。

Realview の実装(MDK および RVDS に適用)

標準ライブラリ(C standardlib, C microlib, fplib, mathlib)をすべて使用してください。これらのライブラリはすべて、初期設定でリンクされています。

- RTOS、またはユーザーによる、代替ルーチンのサポート。ライブラリ ルーチンが「weak」と定義されているため、他の実装が提供されている場合、そちらによって置き換わります。

- 提供されたリンカおよびスキャッタファイルを、ユーザーのもので置き換えるメカニズムが提供されています。これは、ユーザーがプロジェクトにローカルにファイルを作成し、自動的に提供されるファイルの代わりに、このファイルをリンカ／スキャッタファイルとして指定するビルド設定を行えるようにすることで実装されます。
- 現在では、ヒープサイズおよびスタックサイズは固定された値（スタック 4K、ヒープ 1K）に指定されています。可能ならば、ヒープサイズおよびスタックサイズを指定する要求事項を、全て削除してください。不可能な場合、上記の値をデフォルトとして、オプションにより Design-Wide Resources（設計全般リソース）GUI で他の値を選択できるようにしてください。
- Generated Source(生成されたソース）ツリー内のコードは全て、ビルドプロセスの一環として一つのライブラリにコンパイルされます。このコンパイルされたライブラリは、最終的なリンクにおいて、ユーザーコードとリンクされます。

CMSIS サポート (PSoC 5)

Cortex マイクロコントローラ・ソフトウェア・インターフェイス規格(CMSIS)は、ARM が Cortex M シリーズのプロセッサとやりとりするのに使用する規格です。複数のサポートレベルが存在します。以下のサポートが提供されています。

- Core Peripheral Access Layer
  - core_cm3.c
  - core_cm3.h

これらのファイルは、修正することなく使用されます。同じファイルが、サポートされているプラットフォームすべてで動作します。

リセット ステータスの保存 (PSoC 3 と PSoC 5)

リセット ステータス レジスタの値が読み取られ、デバイスが起動するときは常にクリアされ、その値はグローバル SRAM 変数に保存されます。このレジスタは PSoC 3 では RESET_SR0 です。その変数は、レジスタのフィールドの定義とともに、提供されます。

uint8 CyResetStatus

名前	説明
CY_RESET_LVID	低電圧検出、デジタル
CY_RESET_LVIA	低電圧検出、アナログ
CY_RESET_HVIA	高電圧検出、アナログ
CY_RESET_WD	ウォッチドッグ リセット
CY_RESET_SW	ソフトウェア リセット
CY_RESET_GP0	汎用ビット 0
CY_RESET_GP1	汎用ビット 1

14 ウォッチドッグ タイマ

API

void CyWdtStart(uint8 ticks, uint8 lpMode)

説明: ウォッチドッグ タイマを有効にします。タイマは指定されたカウント間隔用に構成され、セントラル タイム ホイールがクリアされ、低電力モードの設定が構成され、ウォッチドッグ タイマがイネーブルになります。

ウォッチドッグは、一度有効になった後は無効化できません。ウォッチドッグは、セントラル タイム ホイール (CTW) が指定された期間に到達するたびにカウントします。ウォッチドッグが 3 まで数える前に、CyWdtClear() 関数を使用してウォッチドッグをクリアする必要があります。CTW はフリーランニングであるため、タイマ期間が 2 と 3 の間を経過してから、これを実行します。

パラメータ: ticks: 4つある、利用可能なタイマー周期の1つです

値	定義	時間
0	CYWDT_2_TICKS	2 CTW ティック
1	CYWDT_16_TICKS	16 CTW ティック
2	CYWDT_128_TICKS	128 CTW ティック
3	CYWDT_1024_TICKS	1024 CTW ティック

lpMode: 低電力モードの設定

値	定義	効果
0	CYWDT_LPMODE_NOCHANGE	変化なし。
1	CYWDT_LPMODE_MAXINTER	スリープおよび休止モードにおいて、最長のタイマーモードに切り替えます。
3	CYWDT_LPMODE_DISABLED	スリープおよび休止モードにおいて、WDT を無効にします。

返り値: なし

副作用と制限: すべての 'lpMode' パラメーターは PSoC 3 製造シリコンによりサポートされています。PSoC 5 と PSoC 3 ES2 は NOCHANGE のみサポートします。

ウォッチドッグ タイマーのハードウェア実装では、一度有効化された後は、タイマーの修正が一切不可能になります。また、一度有効化された後は、タイマーを無効化することもできません。これは、不正なコードによる変更から、ウォッチドッグ タイマーを保護するためです。このため、リセット後の CyWdrStart() への最初の呼び出しのみが、効果があります。

void CyWdtClear()

説明: ウォッチドッグ タイマーをクリア(フィード)します。

パラメータ: なし

返り値: なし

15 cy_boot コンポーネントにおける変更

バージョン 2.40

このセクションでは、cy_boot コンポーネント バージョン 2.40 での主要な変更点を列挙し、解説します。

バージョン 2.30 での変更点の解説	変更の理由 / 影響
CyPmSleep() と CyPmHibernate() API を更新。	電力モードの構成を改善するための変更が行われました。

バージョン 2.30

このセクションでは、cy_boot コンポーネント バージョン 2.30 での主要な変更点を列挙し、解説します。

バージョン 2.30 での変更点の解説	変更の理由 / 影響
初期設定で、CyIntEnable と CyIntDisable 関数が CYREENTRANT に変更。	多くのコンポーネントは CyIntEnable と CyIntDisable が再入可能になることが必要であり、これらのコンポーネントでそれが起きることは全くありません。つまり、呼び出さない関数に対して、cyre ファイルを追加する必要がなくなりました。
CyPmSleep() および CyPmAltActive() 関数の実装が、デバイス低電力モードの入力前に、32 KHz ECO、100 KHz と 1 KHz の ILO 電力モード構成を削除することによって、変更されました。	ユーザーは、スリープ モードおよびオルタネート アクティブ モード中のクロック電力モードの構成に、責任があるようになりました。 CyILO_SetPowerMode() と CyXTAL_32KHZ_SetPowerMode() はクロック電力モードを構成するために使用することができます。 クロック電力モード構成のユーザー責任に関する情報が、PM API セクションに追加されました。
CyPmSaveClocks() の実装は更新され、IMO クロック周波数を「Enable Fast IMO during startup（起動時に高速 IMO をイネーブル」がイネーブルのときは 48 MHz に、そしてその他の場合は 12 MHz に設定します。IMO 周波数は、低電力モード入力の直前に必ず 12 MHz に設定され、ウェイクアップ直後に復元されます。CyPmRestoreClocks() は IMO クロックの元の値を復元します。	IMO の値は FIMO と一致する必要があり、FIMO は常に 12 MHz です。

バージョン 2.30 での変更点の解説	変更の理由 / 影響
CyPmRestoreClocks() 関数の実装は更新され、復元 MHz ECO と PLL ディスエーブル ステートが削除されました。	CyPmRestoreClocks() 関数は CyPmSaveClocks() の後に呼び出されることが予想され、最後のものが必ず MHz ECO と PLL をディスエーブルにします。
グローバル割り込みは CyPmSleep()/CyPmHibernate() 入力の際にディスエーブルになり、関数から戻る前に復元されます。	デバイスの状態が復元される前にこれが発生しないように、割り込みはディスエーブルになっています。
CyPmSleep() および CyPmAltAct() 関数の PSoC 5 デバイスへの 実装は、すべてのパラメータを無視するように更新されました。PSoC 5 デバイスは、以下の 3 つのウェイクアップ ソースのいずれかからの割り込みによってウェイクアップされるまで、スリープ モードに入ります。セントラル タイム ホイール (CTW)、毎秒 1 回、またはポート割り込みコントローラ (PICU)。これらのウェイクアップ ソースは、割り込みを生成するようにあらかじめ構成されている必要があります。CTW はスリープ タイマ コンポーネントを使用して構成され、毎秒 1 回割り込みはリアルタイム クロック コンポーネントを使用して構成されます。	CyPmSleep() と CyPmAltAct() 関数は次のパラメーターと共にのみ使用する必要があります。CyPmSleep(PM_SLEEP_TIME_NONE, PM_SLEEP_SRC_NONE) と CyPmAltAct(PM_ALT_ACT_TIME_NONE, PM_ALT_ACT_SRC_NONE)。
コンテンツ全体で使用する、更新されたアーキテクチャ特有およびシリコン特有の #defines 。	
8051 デバイスでの、DMA 以外の構成の性能が向上しました。	これらの変更により、起動時間が減少し、コード メモリと内部データ メモリの消費がわずかに減少します。
CyPmRestoreClocks() の実装は PSoC5 シリコンに対して更新されました。メガヘルツ水晶振動子の安定に 130 ms が与えられています。ホールドオフ タイムアウト後、レディネスは検証されません。	これらの変更により、水晶振動子の起動時間が増加しますが、水晶振動子の使用準備が完了していることを確認します。
ウェイクアップ タイマとして使用されるタイマのソース クロックの電力モードが、PM API 関数から削除されました。	PM API 関数を呼び出す前に、ウェイクアップ タイマとして使用されるタイマのソース クロックの電力モードを、手動で構成する必要があります。
PSoC Creatorパワーマネージメントセクションが更新されました。	パワーマネージメント API の使用について、詳細情報が追加されました。

バージョン 2.21

このセクションでは、cy_boot コンポーネント バージョン 2.21 の主要な変更点を列挙し、解説します。

バージョン 2.21 の変更点の解説	変更の理由 / 影響
ブートローダ ホストからブートローダに転送されるデータのチェックサムを計算する方法を選択するための、新しいオプションを提供します。	IO 転送中のエラーを確認する、より堅牢な方法を提供します。
ユーザーが独自のカスタム ブートローダ 通信機能を定義できるようにする、包括的オプションを提供します。	追加通信プロトコル (SPI、UART、...) 用のサポートを追加 しやすくします。同じ設計で複数の通信コンポーネントを、同時にサポートする手段も提供します。
生じる可能性のある一部の問題を防ぐため、いくつかのパワーマネージメント機能を更新しました。	一部の括弧が欠落していたため、項目が間違った順序で評価されることがありました。

バージョン 2.21 の変更点の解説	変更の理由 / 影響
RESET_SR0 レジスタから情報を取得するために利用できる変数 CyResetStatus を追加しました。	RESET_SR0 レジスタ内に含まれているフィールドの多くは読み取ると消去されるため、これが提供されています。ブートローダは、動作の一部としてこのレジスタにアクセスする必要があるので、実際のアプリケーション コードが値にアクセスすることが防止されます。アプリケーションがそれでもすべての情報にアクセスできるように、変数が提供されています。
NVL 値が適切に初期化されたことを一部の PSoC3 デバイスが確認する、回避策を追加しました。	いくつかの PSoC 3 デバイスでは NVL 情報が適切に初期化されないことがあります。起動コードのいずれかを実行する前に NVL が適切にロードされることを確認する、回避策が提供されています。

バージョン 2.20

この節では、cy_boot コンポーネント バージョン 2.20 における主要な変更点を列挙し、解説します。

バージョン 2.20における変更点の解説	変更の理由 / 影響
CyDelayCycles関数を、命令キャッシュ (instruction cache)が有効時にも動作できるように更新しました。	元の CyDelayCycles関数は、命令キャッシュ(instruction cache) が無効な状態で動作するように設計してありました。PSoC 3 ES3 と製造シリコンに対しては、命令キャッシュがイネーブルされている場合、遅延の長さは適切ではありませんでした。
低電力モードの関数について、コード内のコメント、およびこのガイドの解説を更新しました。	CyPmSleep()、CyPmHibernate()、およびCyPmAltAct() 関数のコメントおよび解説で、PSoC 3 ES2 および PSoC 5 ES1 の欠陥に関するシリコン エラッタを参照するように明記しました。
CyBtldr_ValidateApp関数に関する、ブートローダの問題を修正しました。	PSoC 5のブートローダにおいて、アプリケーションコードが64 Kを越えた場合に、アプリケーションコードの検証ができなくなる問題を修正しました。
ブートローダの、トラフィックを受信するまでウェイトする問題に対処しました。	ブートローダが、アクティビティがあり、誰がが通信しようと試みている、と判断するために必要な要件を修正しました。 以前は、ブートローダが、選択された通信コンポーネント上からデータを少しでも受信した場合、いつまでもデータを受け取るためにウェイトしていました。この変更のため、ブートローダは Enter Bootloader (ブートローダ入力)コマンドを受信した場合、永遠にウェイトするようになります。
ダイ温度を 2 回読み取って、値が安定していることを確認するように、CySetTemp 関数を更新しました。	問題に対処するための変更が、行われました。影響はありません。

バージョン 2.10

この節では、cy_boot コンポーネント バージョン 2.10 における主要な変更点を列挙し、解説します。

バージョン 2.10における変更点の解説	変更の理由 / 影響
ブートローダのフラッシュ検証コードを更新しました	フラッシュがディスエーブルの場合、有効な画像に対しても故障が報告される原因となる、ブートローダの検証ルーチンの問題を修正しました。

バージョン 2.0

この節では、cy_boot コンポーネント バージョン 2.0 における主要な変更点を列挙し、解説します。

バージョン 2.0における変更点の解説	変更の理由 / 影響
Keil C51 キーワードが、マクロとして使用可能になりました	これらのマクロは、他のコンパイラとの互換性を保ちつつ、コードが変数およびポインタのメモリ空間を指定することが可能になりました。最もよく使用されるキーワードはCYCODE、CYXDATA、およびCYDATAであり、これらはC51のキーワードではそれぞれ code、xdata、および dataに対応します。他のコンパイラでは、これらのマクロは無視されます。
CyLib APIが条件的でなくなりました	CyLibからAPI関数をインクルードするために、CYLIB_POWER_MANAGEMENT といったマクロを定義する必要がなくなりました。
RealView リンカスクリプトにおいて、 .dma_init に関する別の節を追加しました	これは、DMAベースの設定が有効になっている場合に、RealViewでプロジェクトをビルドする際の潜在的なエラーおよび警告を防ぎためです。
SRAM割り込みベクタテーブルを完全に初期化します	以前のバージョンでは、テーブルの最初から32エントリのみ、初期化されます。
デフォルトの割り込みハンドラに対する、予約済みの割り込みベクタを初期化します	以前のバージョンでは、予約済みのベクタは0に初期化されます。これは、有効な割り込みサービスルーチンではありません。
PSoC 3における、CyIntSetVectorの戻り値の、誤ったエンディアンを修正しました	以前のバージョンでは、高位バイトと低位バイトが入れ替わっていました。
割り込みサービスルーチンの宣言において、割り込み属性を適用します	CY_ISR/CY_ISR_PROTO/cyisraddress により宣言された割り込みサービスルーチンに、割り込み属性が付加されます。これにより、コンパイラは、スタックアライメントを調整するコードを生成します。RealViewは、割り込み属性を型の一部としてチェックします。よって、割り込み属性を保有しない 割り込みサービスルーチンおよびベクタは、RealViewでコンパイルされません。このため、cy_isr_v1_20および以前のバージョンは、RealView使用時にはcy_boot_v1_50と互換性がありません。これは PSoC 5 のみに影響を及ぼします。インタラプト属性は PSoC 3バージョンに既に存在していました。
CY_GET_REGxx/CY_SET_REGxx を再入可能にし、パフォーマンスを向上しました	この機能を実装した、CyGetRegXX および CySetRegXX関数は、アセンブリルーチンに入れ替えられました。新しいルーチンは、インタラプト サービス ルーチンで安心して使用することができます。
CyDelayにおいて、PSoC 3 ES3をサポートします	CPUCLK_DIVを、ES3においてSFR からCLKDIST_MSTR1 へ移動しました。
limits.h と *ctype.h*を使用して *CyLib.h* におけるいくつかのマクロを置き換えてください	LONG_MIN、LONG_MAX、およびULONG_MAX マクロが、ツールチェーン固有の limits.h により提供されるようになりました。これは、コンパイラの limits.h と CyLib.h の違いにより生じる警告を防ぎます。
--gnuにおいて、RealViewモードをサポートするようになりました	RealView を --gnu オプションで使用する場合にエラーを起こす、いくつかのプリプロセッサの条件文を修正しました。
PSoC 5の起動コードを、標準ライブラリをより効率的に使用するように修正しました	PSoC 5の起動コードは、初期化時に標準ライブラリを使用するように更新されました。これは標準初期化を提供します。

バージョン 2.0における変更点の解説	変更の理由 / 影響
cydevice.hを cydevice_trm.hに置き換えました	cydevice.hは obsolete (古いため使用しないことを推奨)と指定されたため、PSoC Creatorとともに提供される、APIおよび生成されたコードは、これを使用すべきではありません。
CYCODE、CYDATA、CYXDATA、および CYFAR 定義は、cytypes.hに移されました	フラッシュコンポーネントなどの、全てのコンポーネントが、これらの定義を使用できるようになりました
スリープ後の、キャッシュフラッシュサイクルの不適切な挙動を修正しました。	CYREG_CACHE_CR はスリープ後 PSOC 3 で値を保持しません
雑多な Flash/EEPROM APIの変更を行いました。	アクティブ/スタンバイ 電源設定レジスタが、シリコンごとに定義されるようになりました。 アクティブ/スタンバイ 電源設定レジスタ定数が、シリコンごとに定義されるようになりました。 CyFlash_Start() およびCyFlash_Stop() APIを追加しました。 CyFlashEEActivePower()および CyFlashEEStandbyPower APIを削除しました。 CySetFlashEEBuffer()を、PSoC 3において、バッファが0でポインタであるか否かをテストしないように修正しました。 CyWriteRowConfig() API の引数の1つを rowDataに変更しました。 CyEEPROM_Start()、CyEEPROM_Stop()、 CyEEPROM_ReadReserve()、 CyEEPROM_ReadRelease()、および CyFlash_SetWaitCycles(uint8 freq) APIを追加しました。 電源モードレジスタ、および電源モードレジスタ定数が、シリコンごとに定義されるようになりました。 キャッシュ管理(Cache Control) レジスタが、シリコンごとに定義されるようになりました。
再入可能のサポートを追加しました。	cylib.c/h、cyflash.c/h、cydmac.c/h、および cyspc.c/h ファイル内の適切なAPIが、再入可能をサポートするように更新されました。
cymemset および cymemcpy をマクロに変換し、cymemmoveを削除しました。	ツールチェーンベンダにより提供される memset/memcpy/memmoveは、特定のターゲットプラットフォーム向けに設計されているため、一般的には汎用的な実装より早く動作します。ベンダにより提供される関数と、汎用的な実装を両方インクルードすることは、コード容量の無駄です。非レガシーコードは直接 memset と memcpy を使用する必要があります。

バージョン 2.0における変更点の解説	変更の理由 / 影響
多くの新しいクロックAPIを追加しました: CyPLL_OUT_Start_confirm CyMasterClock_SetSource CyMasterClock_ActivateIMOAndSet CyMasterClock_ActivatePLLAndSet CyIMO_ActivateAndWait CyIMO_SetFrequency CyIMO_EnableDisableDoubler CyIMO_EnableDoubler CyIMO_DisableDoubler CyIMO2X_SetSource CyPLL_P_SetValue CyPLL_Q_SetValue CyPLL_SetInput CyXTAL_SetGain CyXTAL_SetVref CyILO_1KHZ_Start CyILO_1KHZ_Stop CyILO_100KHZ_Start CyILO_100KHZ_Stop CyILO_33KHZ_Start CyILO_33KHZ_Stop CyILO_SelectClock CyUSB_Start CyUSB_Stop CyUSB_SetClockSource CyUSB_SetClockSource_IMO2X CyUSB_SetClockSource_IMO CyUSB_SetClockSource_PLL CyUSB_SetClockSource_DSI CyUSB_EnableDivider CyUSB_DisableDivider CyCLK_SetDividerValue CyCLK_SetBusDividerValue	追加の機能を提供するため。
char8 データ型は charとして定義されます。	将来の新しいコンパイラをサポートするため。
CySleep および CyHibernate APIを CyPmSleep および CyPmHibernateにそれぞれ改名し、新たに CyPmAltAct APIを追加しました。	cy_boot 2.0以前の低電力APIに欠陥がありました。回路図を更新し、cy_boot 2.0 を使用して、新しい API を使用するように下位電力部分を書き直します。