



PSoC[®] Creator[™]

コンポーネント作成者ガイド

文書番号 # 001-69351 版 *A

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

© Cypress Semiconductor Corporation, 2011-2014.

本文書に含まれる情報は、予告なく変更されることがあります。Cypress Semiconductor Corporation は、Cypress 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責任を負いません。かつ、Cypress Semiconductor Corporation、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、含意することはありません。Cypress 製品は、Cypress との明示的な書面による合意によらない限り、医療、生命維持、救命、重要な管理、または安全用途に使用することを保証するものではなく、使用することを意図したものでもありません。さらに Cypress は、誤動作や故障によって使用者に重大な傷害がもたらされることが合理的に予測される、生命維持システムの重要なコンポーネントとして Cypress の製品を使用する許可を与えるものでもありません。生命維持システム用途に Cypress 製品を供することは、製造者がそのように使用する上でのあらゆるリスクを負うことを意味し、その結果 Cypress はあらゆる責任を免除されることを意味します。

PSoC® および CapSense® は Cypress Semiconductor Corp の登録商標であり、PSoC Creator、Programmable System-on-Chip、および Warp は Cypress Semiconductor Corp. の商標です。本文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。

すべてのソースコード（ソフトウェアおよび／またはファームウェア）は 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って保護され、かつそれらに従います。Cypress が本書面によりライセンス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 Cypress の集積回路と併用されるライセンス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、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いて、本ソースコードをどのように複製、変更、変換、コンパイル、または表示することも、Cypress の明示的な書面による許可がない限り禁止されます。

免責条項：CYPRESS 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を含みますが、それらに限定されません。Cypress は、本文書に記載した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。Cypress は、本文書に記載した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。さらに Cypress は、誤動作や故障によって使用者に重大な傷害がもたらされることが合理的に予測される、生命維持システムの重要なコンポーネントとして Cypress の製品を使用する許可を与えるものでもありません。生命維持システム用途に Cypress 製品を供することは、製造者がそのように使用する上でのあらゆるリスクを負うことを意味し、その結果 Cypress 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 Cypress ソフトウェア ライセンス契約によって制限され、かつそれに従います。

目次



1. はじめに	9
1.1 コンポーネント作成プロセスの概要.....	9
1.2 慣例的な表記.....	10
1.3 参考文献.....	10
1.4 修正履歴	10
2. プロジェクトとコンポーネントの作成	11
2.1 Cypress によるコンポーネント要求項目.....	11
2.1.1 ファイル名	11
2.1.2 名づけ方法	11
2.1.3 ファイル名の長さ制限	11
2.1.4 コンポーネントのバージョン	12
2.2 ライブラリプロジェクトの作成.....	13
2.3 新しいコンポーネントアイテム（シンボル）の追加	14
2.3.1 空シンボルの作成	14
2.3.2 ウィザードを使用したシンボルの作成	17
3. シンボル情報の定義	19
3.1 パラメータの概要.....	19
3.1.1 フォーマルおよびローカルパラメータ	20
3.1.2 組み込みパラメータ	20
3.1.3 関数の表現方法	21
3.1.4 ユーザー定義型	22
3.2 シンボルパラメータの定義.....	22
3.3 パラメータの妥当性検証式の追加.....	24
3.4 ユーザー定義型の追加.....	25
3.5 文書のプロパティの指定.....	26
3.5.1 カタログの配置の定義	27
3.6 形のプロパティの定義.....	28
3.6.1 一般的な形のプロパティ	28
3.6.2 高度な形のプロパティ	28
4. 実装の追加	29
4.1 回路図を用いた実装.....	29
4.1.1 回路図の追加.....	29
4.1.2 回路図の完成.....	30
4.1.2.1 デザインワイドリソース (Design-Wide Resources) (DWR) の設定	30
4.2 回路図マクロの作成.....	30
4.2.1 回路図マクロの追加.....	30
4.2.2 マクロの定義.....	32
4.2.3 バージョン付け	32

4.2.4	コンポーネント アップデートツール	32
4.2.5	マクロファイルの名づけの慣例	33
4.2.5.1	同じ名前のマクロとシンボル	33
4.2.6	文書のプロパティ	33
4.2.6.1	コンポーネントのカatalogにおける配置	33
4.2.6.2	サマリーテキスト	33
4.2.6.3	隠れたプロパティ	33
4.2.7	マクロ データシート	33
4.2.8	マクロのポストプロセッシング	33
4.2.9	例	34
4.3	Verilog を用いた実装	34
4.3.1	Verilog のファイル要求項目	34
4.3.2	Verilog ファイルの追加	35
4.3.2.1	新規のファイル	35
4.3.2.2	既存のファイル	35
4.3.3	Verilog ファイルの完成	36
4.3.4	UDB アレイの変更	36
4.3.4.1	シリコンの版の定義	36
4.3.4.2	異なるシリコンの版におけるコンポーネントの設定	37
4.3.5	UDB 要素	38
4.3.5.1	クロック / イネーブルの仕様	38
4.3.5.2	データパス	39
4.3.5.3	コントロールレジスタ	45
4.3.5.4	ステータスレジスタ	46
4.3.5.5	Count7	47
4.3.6	固定ブロック	48
4.3.7	デザインワイド リソース	48
4.3.8	ロジックの代わりに Cypress が提供するプリミティブを使用すべき状況	48
4.3.9	コンポーネント作成における Warp の特徴	48
4.3.9.1	生成文	48
4.4	ソフトウェアによる実装	50
5.	ハードウェアのシミュレーション	51
5.1	ModelSim	51
5.2	VCS	52
5.2.1	ツール	52
5.2.2	テストベンチの定義	52
6.	API ファイルの追加	57
6.1	API の概要	57
6.1.1	API 生成	57
6.1.2	ファイルの名づけ	57
6.1.3	API テンプレートの拡張	57
6.1.3.1	パラメータ	57
6.1.3.2	ユーザー定義のタイプ	57
6.1.4	条件的 API 生成	58
6.1.5	Verilog ヒエラルキーの代替	59
6.1.6	マージ領域	59
6.1.7	API のケース	59
6.2	コンポーネントに API ファイルを追加する	60
6.3	.c ファイルの完成	60
6.4	.h ファイルの完成	60

7. コンポーネントのカスタマイズ	61
7.1 カスタマイザのソース	61
7.1.1 カスタマイザのソースの保護	61
7.1.2 開発フロー	61
7.1.3 ソースファイルの追加	62
7.1.4 Custom にサブディレクトリを追加する	62
7.1.5 リソースファイルの追加	63
7.1.6 クラス / カスタマイザの名づけ	63
7.1.7 アセンブリ レファレンスの指定	63
7.1.8 カスタマイザのキャッシュ	63
7.2 プリコンパイルされたコンポーネントカスタマイザ	64
7.3 使用のガイドライン	65
7.3.1 固有の名前空間の使用	65
7.3.2 固有の外部依存関係の使用	65
7.3.3 コードの共同使用のための共通コンポーネントの使用	65
7.4 カスタマイズ例	65
7.5 インターフェース	65
7.5.1 システムインターフェース	65
7.5.2 カスタマイズ用 インターフェース	67
7.5.3 カスタマイザのクロック クエリ	68
7.5.3.1 ICyTerminalQuery_v1	68
7.5.3.2 ICyClockDataProvider_v1	68
7.5.4 クロック API のサポート	68
8. チューニングのサポートの追加	69
8.1 チューニング フレームワーク	69
8.2 アーキテクチャ	70
8.3 API のチューニング	70
8.3.1 LaunchTuner API	70
8.3.2 Communications API (ICyTunerCommAPI_v1)	70
8.4 パラメータの受け渡し	71
8.5 コンポーネントチューナー DLL	71
8.6 通信のセットアップ	71
8.7 チューナーの開始	71
8.8 ファームウェアの「交通警察」	72
8.9 コンポーネントの修正	72
8.9.1 通信データ	73
8.10 簡単なチューナー	73
9. ブートローダー サポートの追加	75
9.1 ファームウェア	75
9.1.1 ガード	75
9.1.2 機能	75
9.1.2.1 void CyBtldrCommStart(void)	76
9.1.2.2 void CyBtldrCommStop(void)	76
9.1.2.3 void CyBtldrCommReset(void)	76
9.1.2.4 cystatus CyBtldrCommWrite(uint8 *data, uint16 size, uint16 *count, uint8 timeout)	77
9.1.2.5 cystatus CyBtldrCommRead(uint8 *data, uint16 size, uint16 *count, uint8 timeout)	77
9.1.3 カスタマイザ ブートローダー インターフェース	77

10. コンポーネントの完成	79
10.1 データシートの追加 / 作成	79
10.2 XML デバッグファイルの作成 / 追加	80
10.2.1 XML フォーマット	80
10.2.2 XML ファイルの例	81
10.3 XML 互換性ファイルの追加 / 作成	82
10.3.1 XML フォーマット	82
10.3.2 例: .cystate ファイル	83
10.4 プロジェクトのビルド	83
11. ベストプラクティス	85
11.1 クロック	85
11.1.1 UDB アーキテクチャーのクロックへの配慮	85
11.1.2 コンポーネントのクロックへの配慮	85
11.1.3 UDB からチップリソースへのクロックへの配慮	86
11.1.4 UDB から入出力へのクロックへの配慮	86
11.1.5 フリップフロップのメタスタビリティ	86
11.1.6 クロックによるドメイン境界の横断	87
11.1.7 長い組み合わせパスへの配慮	87
11.1.8 同期クロック対非同期クロック	87
11.1.9 cy_psoc3_udb_clock_enable プリミティブの活用	87
11.1.10 cy_psoc3_sync コンポーネントの活用	88
11.1.11 ルートクロック、グローバルクロック、および外部クロック	89
11.1.12 負のクロックエッジの隠れた危険性	89
11.1.13 一般的なクロックの規則	89
11.2 割り込み	89
11.2.1 ステータスレジスタ	90
11.2.2 内部割り込みの生成とマスクレジスタ	90
11.2.3 スリープ間の保持	91
11.2.4 FIFO ステータス	92
11.2.5 バッファ オーバーフロー	92
11.2.6 バッファアンダーフロー	92
11.3 DMA	93
11.3.1 データ移動用のレジスタ	94
11.3.2 ステータス用のレジスタ	94
11.3.3 スポーク幅	95
11.3.4 FIFO ダイナミック制御の解説	96
11.3.5 データパスの条件 / データ生成	96
11.3.6 UDB ローカルバス設定インタフェース	97
11.3.7 UDB ペア アドレス設定	97
11.3.7.1 ワーキングレジスタのアドレス空間	98
11.3.7.2 8 ビット ワーキングレジスタへのアクセス	99
11.3.7.3 16 ビット ワーキングレジスタのアドレス空間	99
11.3.7.4 16 ビット ワーキングレジスタのアドレス制限	100
11.3.8 DMA バスの使用	100
11.3.9 DMA チャネルのバーストタイム	100
11.3.10 コンポーネントの DMA 機能	101
11.4 低電力サポート	101
11.4.1 機能的な要求項目	101
11.4.2 デザインへの配慮	101
11.4.3 ファームウェア / アプリケーション プログラミング インターフェースの要求項目...	102

11.4.3.1	データ構成テンプレート	102
11.4.3.2	保存 / 復旧メソッド	102
11.4.3.3	有効化および停止関数への追加	103
11.5	コンポーネントの内包	103
11.5.1	階層デザイン	103
11.5.2	パラメータ化	107
11.5.3	コンポーネントのデザインへの配慮	107
11.5.3.1	リソース	107
11.5.3.2	電源管理	108
11.5.3.3	コンポーネントの開発	108
11.5.3.4	コンポーネントのテスト	109
11.6	Verilog	110
11.6.1	Warp: PSoC Creator 合成ツール	110
11.6.2	合成可能なコードのガイドライン	110
11.6.2.1	ブロッキング型アサイメントと非ブロッキング型アサイメント	110
11.6.2.2	ケース文	111
11.6.2.3	パラメータの取り扱い	112
11.6.2.4	ラッチ	114
11.6.2.5	リセットとセット	114
11.6.3	最適化	115
11.6.3.1	パフォーマンス最適化デザイン	115
11.6.3.2	サイズ最適化デザイン	115
11.6.4	リソースの選択	116
11.6.4.1	データパス	117
11.6.4.2	PLD ロジック	118
A.	式評価	119
A.1	評価のコンテキスト	119
A.2	データ型	119
A.2.1	ブーリアン型	119
A.2.2	エラー型	119
A.2.3	浮動小数点型	120
A.2.4	整数型	120
A.2.5	文字列型	120
A.3	データ型の変換	121
A.3.1	ブーリアン型	121
A.3.2	エラー型	121
A.3.3	浮動小数点型	121
A.3.4	整数型	121
A.3.5	文字列型	121
A.3.5.1	ブーリアン型のような文字列	122
A.3.5.2	浮動小数点型のような文字列	122
A.3.5.3	整数型のような文字列	122
A.3.5.4	その他の文字列	122
A.4	演算子	123
A.4.1	算術演算子 (+, -, *, /, %, 単項 +, 単項 -)	123
A.4.2	数値比較演算子 (==, !=, <, >, <=, >=)	123
A.4.3	文字列比較演算子 (eq, ne, lt, gt, le, ge)	123
A.4.4	文字列連結演算子 (.)	123
A.4.5	三項演算子 (?:)	124
A.4.6	型変換	124
A.5	文字列の補完	124

A.6	ユーザー定義のデータ型（列挙型）.....	124
B.	データパス設定ツール	125
B.1	一般的機能.....	125
B.2	フレームワーク.....	126
B.2.1	インターフェース.....	126
B.2.2	メニュー.....	127
B.2.2.1	ファイルメニュー.....	127
B.2.2.2	編集メニュー.....	127
B.2.2.3	閲覧メニュー.....	127
B.2.2.4	ヘルプメニュー.....	127
B.3	一般的なタスク.....	127
B.3.1	データパス設定ツールを起動する.....	127
B.3.2	Verilog ファイルを開く.....	128
B.3.3	ファイルを保存する.....	128
B.4	ビットフィールド型パラメータを使用する.....	129
B.4.1	列挙されたビットフィールドにパラメータを追加する.....	129
B.4.2	マスク ビットフィールドにパラメータを追加する.....	129
B.4.3	ビットフィールドの依存性.....	130
B.5	設定を行う.....	130
B.5.1	設定の名づけ.....	130
B.5.2	設定の編集.....	131
B.5.3	コピーとペーストの設定.....	131
B.5.4	設定のリセット.....	131
B.6	データパス インスタンスを使用する.....	131
B.6.1	新しいデータパス インスタンスを作成する.....	131
B.6.2	データパス インスタンスを削除する.....	132
B.6.3	初期レジスタ値を設定する.....	132

1. はじめに



このガイドには、**PSoC Creator** のコンポーネントを作成する方法と、作成に役に立つ情報が記載されています。このガイドは、他のユーザーが **PSoC Creator** を使用するための複雑なコンポーネントを作成しようとする、上級者ユーザー向けに書かれています。しかし、このガイドには、自分用のコンポーネントを作成したい初心者ユーザー向けの、基本的な原則も記載されています。この章の内容は以下のとおりです：

- コンポーネント作成プロセスの概要
- 慣例的な表記
- 参考文献
- 修正履歴

1.1 コンポーネント作成プロセスの概要

コンポーネントを作成するプロセスは、大きく分けて次のステップを含みます。詳細については、このガイドの様々な章を参照してください。

- ライブラリプロジェクトの作成 ([2 章](#))
- コンポーネント / シンボルの作成 ([2 章](#))
- シンボル情報の定義 ([3 章](#))
- 実装の作成 ([4 章](#))
- ハードウェアのシミュレーション ([5 章](#))
- API ファイルの作成 ([6 章](#))
- コンポーネントのカスタマイズ ([7 章](#))
- チューニング サポートの追加 (上級者向け) ([8 章](#))
- ブートローダー サポートの追加 (必要な場合) ([9 章](#))
- 説明書および他のファイル・文書の作成・追加 ([10 章](#))
- コンポーネントのビルドとテスト ([10 章](#))

注 これらの章は、論理的に関連する情報をそれぞれまとめています。このワークフローは、数あるコンポーネントを作成するワークフローうちの一つです。また、コンポーネントを作成する上での様々なベストプラクティスについては、[11 章](#)を参照してください。

1.2 慣例的な表記

以下の表には、このガイド全体を通して使われている、慣例的な表記がまとめられています。

表記法	使用法
Courier New	ファイル位置およびソースコードです。 C:\...cd\icc\, user entered text
イタリック	ファイル名および参考文献です。 <i>sourcefile.hex</i>
[かっこ付き、太字]	手順上、キーボードに打ち込むコマンドです。 [Enter] or [Ctrl] [C]
File > New Project	メニューパスを示します。 File > New Project > Clone
太字	手順中の、コマンド、メニューパス、選択肢、およびアイコン名です。 デバッガ 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 デバイスに関する注意、もしくは固有の特長です。

1.3 参考文献

このガイドは、PSoC Creator に関連するいくつかの文書のうちの一つです。必要ならば、以下の文書を参照してください。

- PSoC Creator オンラインヘルプ
- PSoC Creator Customization API Reference Guide
- PSoC Creator Tuner API Reference Guide
- PSoC Creator System Reference Guide
- PSoC 3/5 Technical Reference Manual (TRM)
- Warp Verilog Reference Guide

1.4 修正履歴

文書名 : PSoC Creator コンポーネント作成者ガイド		
文書番号 : 文書番号 # 001-69351		
修正	日付	修正内容
**	5/3/11	新しい文書。
*A	06/04/2014	No technical updates.

2. プロジェクトとコンポーネントの作成



本章では、PSoC Creator を使用してライブラリプロジェクトを作成し、シンボルを追加するための基本ステップを説明します。

2.1 Cypress によるコンポーネント要求項目

全てのコンポーネント開発は、PSoC Creator 内で行われますが、[データパス設定ツール \(125 ページ参照\)](#) のような追加ツールを使用する必要がある場合もあります。全てのコード、回路図エディタ、シンボル作成、インタフェース定義、ドキュメント作成などは、既定のコンポーネント内で行われます。

- Cypress が生成する全てのコンポーネントには、変更ログが必要となります。データシートと同様に、別個のコンポーネントアイテムとして、変更ログをコンポーネントに追加します。このログファイルは、簡単なテキストファイルにしてください。コンポーネントの各バージョンのデータシートには、**Change** セクションが含まれる必要があります。
- コンポーネントバージョンには、コンポーネント名を追加してください。表示名 (カタログ配置) 内にバージョン情報を入れないでください。ツールがこの情報を追加します。
- PSoC Creator インストールには、全てのコンポーネントバージョンが含まれます。従って、PSoC Creator において、既定コンポーネントの全てのバージョンが提供されていることを確認してください。

2.1.1 ファイル名

コンポーネントファイル名 (いくつものバージョン数を含む) は、C、UNIX、Verilog と互換性があり、大文字と小文字を区別するものとします。

PSoC Creator は、各コンポーネントで、"givenName" と "storageName" を維持します。givenName には大文字と小文字を両方使用でします。storageName は givenName を全て小文字にしたものです。PSoC Creator は、GUI 内で givenName を使用し、検索用に givenName を storageName にマッピングし、storageName を使用してディスクにファイルを保管します。ユーザーは、コンポーネントの givenName の大文字と小文字を変更することができます。

2.1.2 名づけ方法

コンポーネントを作成する場合、そのファイル名は、回路図マクロ、カスタマイザーソースファイル、API ソースファイルを除き、コンポーネントの全ての要素で同じ名前を使用します。従って、適切な名前を選択することが重要です。

2.1.3 ファイル名の長さ制限

コンポーネント名は、40 文字以下、リソースファイル名は、20 以下としてください。コンポーネントがエンドユーザーデザインで拡張される場合、長いファイル名を用いる場合、パス名の長さに問題が生じます。

2.1.4 コンポーネントのバージョン

Cypress は以下のコンポーネントバージョンポリシーに完全に従います。これは、内部仕様により詳細に文書化されています。以下の事項が、コンポーネントを開発する外部顧客に推奨されています。

PSoC Creator は、コンポーネントバージョンとパッチをサポートしています。コンポーネントのバージョン (メジャーおよびマイナー)、は内蔵されています。パッチはほとんど使用されません。使用されるとしても、実際のコンテンツを変更を伴わない場合です (たとえば、単なる文書の変更)。

- **メジャーバージョン**：互換性に影響する主要な変更 (例：API 変更、コンポーネントシンボルインタフェース変更、機能変更など)
- **マイナーバージョン**：API 変更なし互換性に影響しない、バグの修正または機能の追加
- **パッチ**：API 変更なし互換性に影響しない、文書または既存の機能に対するバグの修正パッチはコンポーネントの一部であるため、ユーザーがアップデートされたコンポーネントをインストールする場合に、自動的にこれらの変更が適用されることに留意してください。

バージョンナンバー "**_v<major_num>_<minor_num>**" がコンポーネント名に追加され、コンポーネント名の一部として扱われます。**<major_num>** および **<minor_num>** は、メジャーバージョンおよびマイナーバージョンを示す整数です。例えば、**CyCounter8_v1_20** は、**CyCounter8** コンポーネントのバージョン **1.20** となります。メジャーおよびマイナーナンバーは、それぞれ別個の整数であり、「実数」を構成するものではありません。例えば、**v1_1** と **v1_10** は同じではありません。また、**v1_2** は、**v1_10** より前のバージョンです。

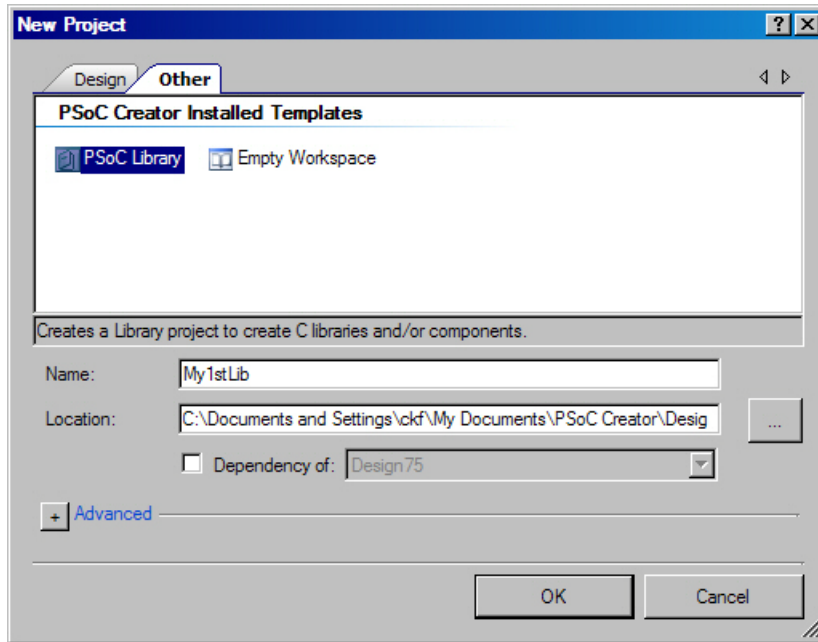
パッチレベルは、コンポーネントに属する (シンボルに付随する) ため、コンポーネント名に反映されません。コンポーネント名の一部にバージョンナンバーを組み込むことで、コンポーネントにメジャーな変更がある場合、既存の設計を保護します。メジャーおよびマイナーバージョン名は、コンポーネント作者の管理下にあることに留意してください。

2.2 ライブラリプロジェクトの作成

最初は、コンポーネントを含む PSoC Creator ライブラリプロジェクトが 1 つあります。コンポーネントの作者は、そのプロジェクト内で個別にコンポーネントを作成します。時間がたつにつれ、複数のコンポーネントが開発される場合、異なる種類のコンポーネント別に異なるライブラリプロジェクトを作成する必要がある可能性があります。

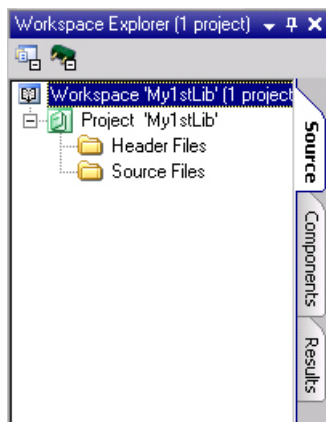
ライブラリプロジェクトを作成するには：

1. [File] [New> Project]> をクリックし  [New Project ダイアログ] を開きます。



2. [Other] タブ をクリックし、[PSoC Library] プロジェクトテンプレートを選択します。
3. プロジェクトの **Name** を入力し、必要に応じてリーダー (...) ボタンをクリックし、[場所] を指定し、プロジェクトを保存します。
4. **OK** をクリックします。

プロジェクトが、ソースタブ下の Workspace Explorer 内に表示されます。



2.3 新しいコンポーネントアイテム（シンボル）の追加

コンポーネント作成プロセスの最初のステップは、新規コンポーネントアイテムのいくつかのタイプのうち 1 つであるシンボル、回路図、Verilog fileなどをプロジェクトに追加することです。新しいコンポーネントアイテムを追加すると、間接的にコンポーネントが作成されます。このような、コンポーネントアイテムの追加に利用されるプロセスは同じですが、各コンポーネントアイテムを完成させるには、異なるステップが必要になります。

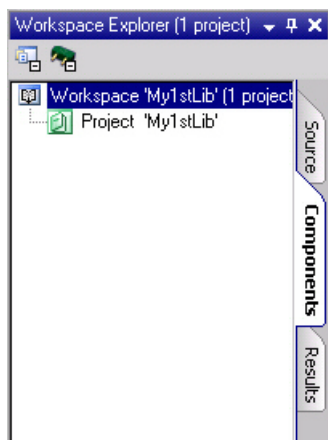
このセクションは、最初のコンポーネントアイテムとして、シンボルを作成することに焦点を当てています。シンボルを作成するには 2 つの方法があります：空のシンボルの使用とシンボルウィザードの使用です。

注：回路図から自動でシンボルを作成することもできますが、このセクションではそのプロセスは説明しません。代わりに、PSoC Creator ヘルプの使用説明を参照してください。

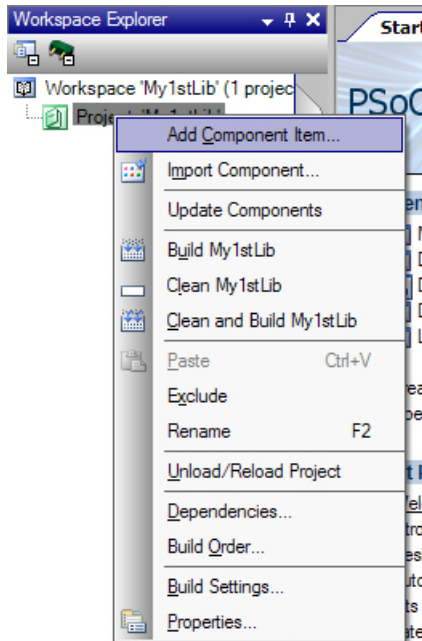
2.3.1 空シンボルの作成

このセクションでは、空シンボルを作成し、外形とターミナルを追加するプロセスを説明します。（シンボルを作成する別の方法については、[17 ページのウィザードを使用したシンボルの作成](#)を参照。）

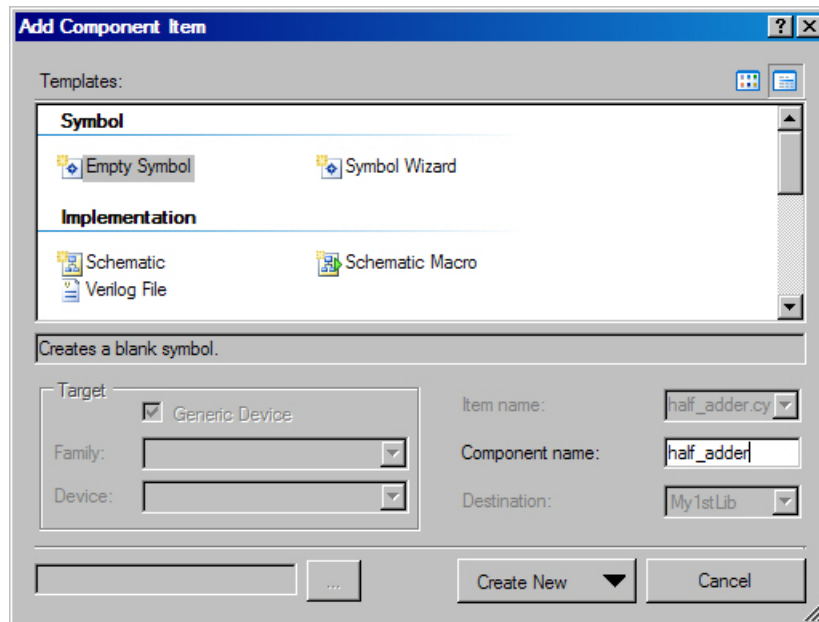
1. 必要に応じて、適切な PSoC Creator ライブラリプロジェクトを開きます。
2. Workspace Explorer の **[Components]** タブをクリックします。



- プロジェクトを右クリックし、**[Add Component Item]** を選択します。



[Add Component Item] ダイアログが表示されます。

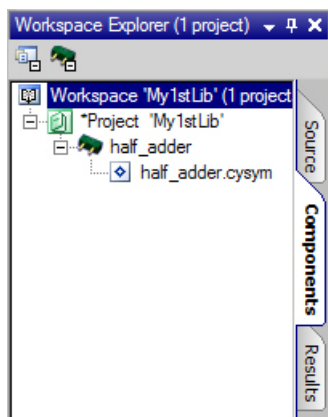


- Empty Symbol アイコンを選択します。
- バージョン情報を含む、コンポーネント名を入力します。
 注：シンボルと全てのコンポーネントの要素は、回路図マクロ、カスタマイザーソースファイル、API ソースファイルを除き、このコンポーネント名が適用されます。 [11 ページの名づけ方法](#)および [12 ページのコンポーネントのバージョン](#)を参照してください。
 また、コンポーネント内のシンボルを 1 つにすることも可能で、そのシンボルは常に標準的です。この場合、**Target** オプションは無効になります。

6. **[Create New]** をクリックすることで、PSoC Creator で新しいシンボルファイルを作成することができます。

注：プルダウンメニューから **[Add Existing]** を選択し、既存のシンボルファイルを選択して、コンポーネントに追加することもできます。

シンボルは、**Workspace Explorer** ツリー内に、唯一のノードとしてシンボルファイル (.cysym) をもつものとして表示されます。



Symbol Editor で、.cysym ファイルを開き、コンポーネントを表す画像を描いたり、インポートすることもできます。

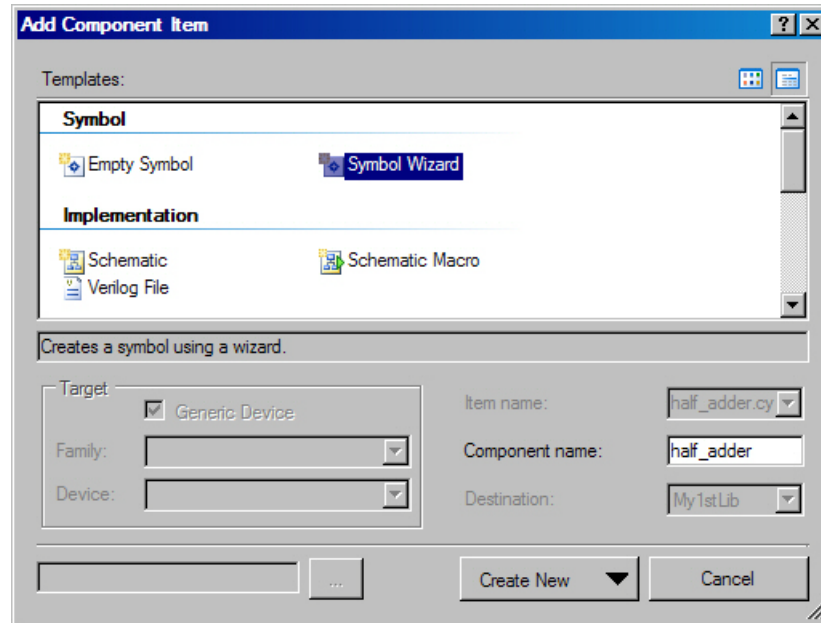
7. 基本的な形とターミナルを描き、**Symbol Editor** を使用して、シンボルを定義します。詳細については、**PSoC Creator** ヘルプを参照してください。

注：**Symbol Editor** キャンバスの中央のプラスサイン（または十字型）は、シンボルの原点になります。

8. **[File]>[Save All]** をクリックして、プロジェクトおよびシンボルファイルへの変更を保存します。

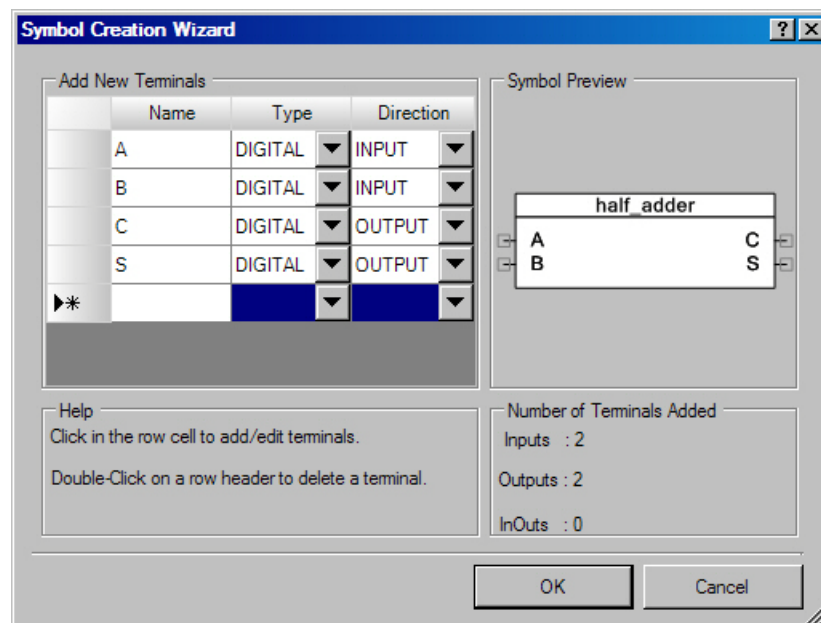
2.3.2 ウィザードを使用したシンボルの作成

[Add Component Item] ダイアログには、Empty Symbol (空シンボル) アイコンに加えて、Symbol Wizard (シンボルウィザード) アイコンが含まれています。このウィザードテンプレートを使用するメリットは、PSoC Creator が基本的シンボル図やターミナルを作成することです。



1. 14 ページの空シンボルの作成に記載されている、コンポーネントの作成の手順に従ってください。
2. ステップ 4 で Empty Symbol (空シンボル) アイコンを選択する代わりに、Symbol Wizard (シンボルウィザード) アイコンを選択します。

[Add Component Item] ダイアログで **OK** をクリックすると、Symbol Creation Wizard が表示されます。



3. **[Add New Terminals]** に、シンボルに加えたいターミナルの名前、タイプ、方向を入力します。
Symbol Preview セクションに、**Symbol Editor** キャンバスにどのようにシンボルが表示されるかのプレビューが示されます。
4. **[OK]** をクリックし、**Symbol Creation** ウィザードを閉じます。
空シンボルを作成するとき同様、新しいコンポーネントが **Workspace Explorer** ツリー内に、唯一のノードとしてシンボルファイル (**.cysym**) を持つものとして表示されます。ただし、**Symbol Editor** は、ウィザードが作成したシンボルが、十字型を中心として表示されます。
5. 必要に応じて、シンボル図を変更します。
6. **[File]>[Save All]** をクリックして、プロジェクトおよびシンボルファイルへの変更を保存します。

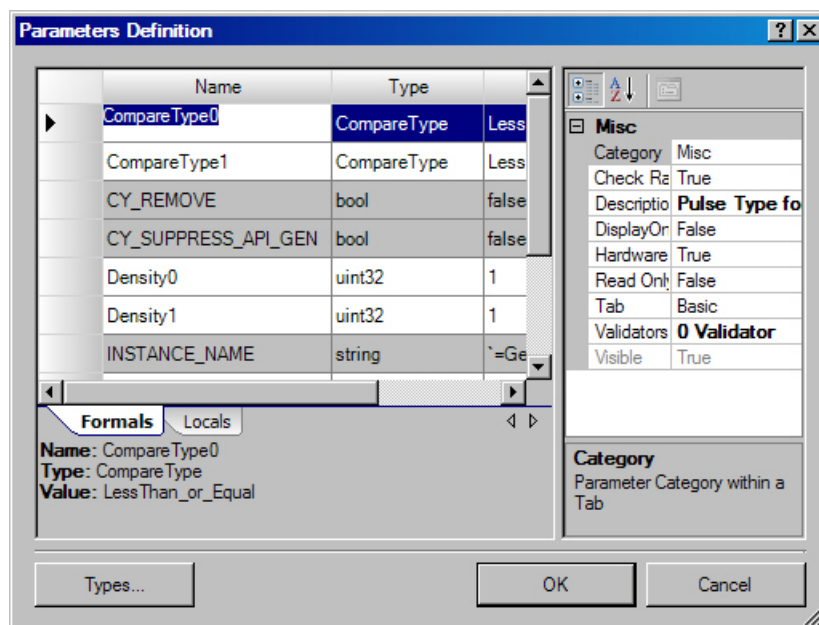
3. シンボル情報の定義



この章では、パラメータ、バリデータ、プロパティなどの、さまざまなシンボル情報を定義するプロセスを説明します。

3.1 パラメータの概要

PSoC Creator 内のパラメータは、コンポーネントの動作を定義する 1 つ以上の名前付きの式のセットです。パラメータ定義ダイアログを使用して、シンボルのパラメータを定義します。



パラメータは、名前 (**Name**)、タイプ (**Type**)、値 (**Value**) などのプロパティの関連リストで構成されています。

- 名前は、基本的に任意に設定できますが、接頭辞 "CY_" は、ビルトインパラメータとして予約されています。パラメータ名は、"CY_"、およびこれと大文字小文字が異なるもので、開始することはできません。
- タイプは、全てのデータタイプがリストアップされるプルダウンメニューに表示されます。データタイプの説明は、以下を参照してください：[119 ページの式評価](#)。
- 値は、インスタンス化されたコンポーネント用のデフォルト値です。
- 各パラメータのさまざまなプロパティは、パラメータがどのように表示され、および動作するかをコントロールします。このプロパティに関する詳細情報については、[22 ページのシンボルパラメータの定義](#)を参照してください。

3.1.1 フォーマルおよびローカルパラメータ

シンボルには、2 種類のパラメータ定義があります：フォーマルとローカルです。フォーマルパラメータは、シンボルの主インタフェースであり、コンポーネントのエンドユーザーは、フォーマルパラメータを表示および構成することのみができます。ローカルパラメータは、コンポーネント作者の利便性のために存在し、エンドユーザーは見ることも編集することもできません。

シンボル内のフォーマルパラメータには決められたデフォルト値がありますが、ローカルパラメータは、他のパラメータを参照する識別子の式を含むことができます。ローカルパラメータ式内の識別子は、フォーマルパラメータおよびローカルパラメータの組み合わせで構成されています。パラメータセットは、事実上ローカルパラメータ用の評価コンテキストです。

回路図ドキュメントには、パラメータセットが付随しています。回路図がエディタで開かれている間、シンボルパラメータセットの完全なコピーのパラメータセットが生成されています。このコピーは、ランタイムに作成され、シンボルと常に同期していることが保証されています。これらのパラメータは、回路図のシンボルを編集しない限り、PSoC Creator 内で直接見ることはできません。回路図にシンボルがない場合、パラメータセットはありません。

コンポーネントのインスタンスは、コンポーネントのシンボルからのフォーマルパラメータを所持します。このインスタンスは、回路図にドロップされた場合、フォーマルシンボルパラメータのコピーを作成します。ユーザーは、フォーマルパラメータの値を式へ変更することができます。インスタンス上のフォーマルパラメータは、ドロップされた回路図に存在するパラメータを全て参照することができます。インスタンスのフォーマルパラメータは、他のパラメータを参照することはできません。フォーマルパラメータは、コンポーネント開発者がデザインされた階層を通して、ユーザーが指定した構成値を伝達する手段です。

エンドユーザーは、ローカルパラメータ値を変更することはできません。インスタンスローカルパラメータは、インスタンスのコンテキスト内で評価されます。インスタンスの他のパラメータを参照することができますが、ドロップされた回路図からのパラメータを参照することはできません。評価コンテキストおよびその使用に関する詳細情報については、[119 ページの式評価](#)を参照してください。

インスタンスパラメータは、シンボルパラメータと、タイプに関わらず名前が対照します。メモリ内インスタンスには、アクティブパラメータセットと孤立パラメータがあります。アクティブセットは、シンボルパラメータセット内に含まれるパラメータです。孤立パラメータのセットは、インスタンス用に作成されたものの、シンボルが変更されているので、有効パラメータでなくなったパラメータのセットです。例えば、エンドユーザーはデータを損失することなく、ライブラリ検索パスを変更し、コンポーネントの別の実装またはバージョンへ切り替え、元の検索パスに戻ることができます。

孤立パラメータは、一時的なものです。孤立パラメータは、関連シンボルが更新され、存在する名前のフォーマルパラメータになった場合にアクティブとなります。エラボレートされた回路図パラメータは、親インスタンスのインスタンスパラメータの直接コピーです。

3.1.2 組み込みパラメータ

組み込みパラメータは、デフォルトで各シンボル用に定義されています。組み込みパラメータの定義は、パラメータ定義エディタから削除することはできません。組み込みパラメータの名前とタイプを変更することはできません。現在、PSoC Creator には、以下の組み込みパラメータが含まれています：

- **CY_COMPONENT_NAME** -- このパラメータには、コンポーネントのベースディレクトリ（シンボルが存在するディレクトリ）へのファイルシステムパスが含まれています。エラボレーション後のみに有効になります。エラボレーション前のこのパラメータ値の値は、"**__UNELABORATED__**" です。

- **CY_INSTANCE_SHORT_NAME** -- 何らかの理由で、コンポーネント作者が、エラボレートされたデザイン内でもインスタンス名の省略名を必要とする場合、このパラメータを通してアクセスすることができます。エンドユーザーがこれを見れる状態にすることは、絶対にあってはなりません。編集不可能です。値は、ユーザーが **INSTANCE_NAME** パラメータを修正する度に、自動的に更新されます。
- **CY_MAJOR_VERSION** -- このパラメータには、付属インスタンス用のメジャーバージョンナンバーが含まれています。
- **CY_MINOR_VERSION** -- このパラメータには、付属インスタンス用のマイナーバージョンナンバーが含まれています。
- **CY_REMOVE** -- このパラメータは、ネットリストからインスタンスを削除するよう伝えるエラボレータへのフラッグです。コンポーネント作者が、単一回路図内で、コンポーネントの 2 つの異なる実装間で動的に切り替えることを可能にする、**live mux** と協働して使用されます。デフォルト値は、**False** です (インスタンス内で維持)。
- **CY_SUPPRESS_API_GEN** -- このパラメータは、コンポーネントに **API** がある場合でも、特定のインスタンスに **API** を作成しないよう指示するために使用することができます。デフォルト値は **False** です。
- **INSTANCE_NAME** -- これは、特別なパラメータです。エディタ内には、インスタンス用の省略名があります。これは、ユーザーがインスタンスの名前を変えたい場合に編集するものです。エラボレートされたデザインでは、**full** または **hierarchical** インスタンス名へのアクセスを提供します。これは、**API** ファイルに不一致がないよう、**`\$INSTANCE\_NAME`** を通して 使用される名前です。つまり、この名前は、コンポーネント作者が唯一参照すべき名前で、適切な値をとります。

注： **INSTANCE_NAME** パラメータは、式エバリュエータと、**API** および **HDL** 生成時では、異なる値を返します。式エバリュエータでは、インスタンスのリーフ名を返し、**API** および **HDL** 生成時では、インスタンスのパス名を返します。

3.1.3 関数の表現方法

だれでも、式内で使用することで、以下の関数にアクセスすることができます。

- **bool IsAssignableName(string)** -- 既存インスタンス名にコンポーネントインスタンスに適切に適用できる場合、**True** を返します。これは、通常、コンポーネントインスタンスの名前を変更する際に、ユーザーインプットの検証に使用されます。
- **string GetShortInstanceName()** -- インスタンス名の省略名を返します。これは、回路図内にユーザーにより入力された名前です。
- **string GetHierInstanceName()** -- 階層インスタンス名を返します。これは、デザイン全てにおいて、インスタンスに与えられた独自の名前です。完全な階層インスタンス名は、回路図エディタ内で利用することはできません。この場合、**GetShortInstanceName()** と同じ値を返します。
- **bool IsValidCCppIdentifierName(string)** -- 既存文字列が、適切な **C** および **C++** 識別子名の場合、**True** を返し、それ以外の場合は **False** を返します。
- **bool** もしくは **error IsValidCCppIdentifierNameWithError(string)** -- 与えられた文字列が、適切な **C** および **C++** の識別子名である場合、ブーリアン型の **True** を返します。識別子が、適切な **C** または **C++** 識別子でない場合、エラー型のメッセージを返します。
- **string GetErrorText(error)** -- エラー型の値 に対応するエラーメッセージを返します。
- **bool IsError(anyType)** -- 引数タイプがエラー型の場合、**True** を返します。他の場合は、**False** を返します。

- `string GetMarketingNameWithVersion()` -- 完全なマーケティング名、および PSoC Creator の現在実行されているバージョンを返します。
- `int GetMajorVersion()` -- コンポーネントのメジャーバージョンナンバーを返します。
- `int GetMinorVersion()` -- コンポーネントのマイナーバージョンナンバーを返します。
- `string GetComponentName()` -- コンポーネント名を返します。

関数呼び出しを含む式を作成することができます。テキストラベル内で関数を使用するためには、例えば ``=IsError()` といった形式を使用します。このような関数のうち 1 つについて、パラメータ値を設定したい場合、単に値を関数に設定します。同様に、これらの関数は、バリデータ式内で使用することができます。[24 ページのパラメータの妥当性検証式の追加](#)も参照してください。

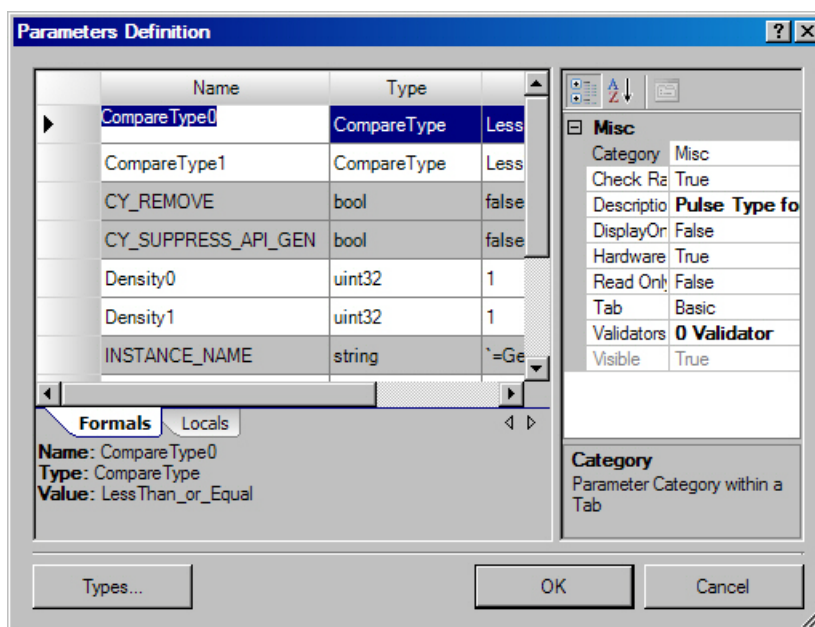
3.1.4 ユーザー定義型

ユーザー定義型 (または列挙タイプ) は、値が列挙からくるようなシンボルのパラメータを定義するために使用されます。ユーザー定義型は、継承することができます。例えば、Verilog 実装およびシンボルとなるカウンターの、UDB 実装を作成することができます。このシンボルは、修正された関数ブロック用の別のシンボルとともに、トップレベル回路図内に置かれます。全ての列挙タイプを再定義できますが、エラーの可能性が増えます。また、トップレベルコンポーネントは、低レベルコンポーネントから継承された列挙タイプを使用することができます。[22 ページのシンボルパラメータの定義](#)および [25 ページのユーザー定義型の追加](#)を参照してください。

3.2 シンボルパラメータの定義

シンボル用のパラメータを定義するには：

1. **Symbol Editor** キャンバスまたはシンボルファイルタブをクリックし、シンボルファイルをアクティブにしてください。
2. 右クリックし、**[Symbol Parameters...]**  を選択し、Parameter Definition ダイアログを開きます。



3. **[Fomals]** タブまたは **[Locals]** タブをクリックし、フォーマルまたはローカルとしてパラメータを定義します。詳細情報については、[20 ページのフォーマルおよびローカルパラメータ](#)を参照してください。

4. 各シンボルに対し、任意の数のパラメータを作成することができます。まず、ダイアログの左側にあるパラメータテーブル内における、名前、タイプ、およびパラメータを定義します。

注：デフォルトパラメータ値は、コンポーネントが最初にインスタンス化された時点の値のみを定義します。値は、その時点のスナップショットです。コンポーネントのデフォルト値が次に変更された場合、その値は既にインスタンス化されたコンポーネントに対して伝達されることはありません。

5. 各パラメータに対し、ダイアログの右側にて以下のプロパティを定義します。

- **Category** -- シンボルが **Parameter Editor** ダイアログ内で表示されるカテゴリ名です。これは、パラメータをグループ化する方法です。
- **Check Range** -- **True** の場合、各評価後に、パラメータの値が関連するタイプの範囲内にあることを確認します。範囲外の場合、**PSoC Creator** は式評価エラーを作成します。有効な範囲は以下です：

型	有効な範囲
列挙タイプ	列挙タイプを定義する整数値
符号付き 16 ビット整数型	-32768 .. x .. 32767
符号なし 16 ビット整数型	0 .. x .. 65535
符号付き 8 ビット整数型	-128 .. x .. 127
符号なし 8 ビット整数型	0 .. x .. 255
ブーリアン型	真および偽
他の全ての型	全ての値が有効です。

- **Description** -- **Parameter Editor** ダイアログ内で、エンドユーザーに対しこのパラメータについて表示される事項です。
 - **DisplayOnHover** -- **Schematic Editor** 内のインスタンス上をマウスが動いている間、パラメータ値を表示するか否かを指定します。
 - **Hardware** -- パラメータが作成された **Verilog** ネットリストに含まれるか否かを指定します。
 - **Read Only** -- エンドユーザーがこのパラメータを変更できるか否かを指定します。
 - **Tab** -- **Schematic Editor** の **Parameter Editor** において、このシンボルが表示されるタブ名。
 - **Validators** -- パラメータに対する 1 つ以上の検証式。このフィールドの使用方法については、[24 ページのパラメータの妥当性検証式の追加](#)を参照してください。**CyExpressions** に関する詳細情報は、付録 A を参照してください。
 - **Visible** -- パラメータ値が、**Parameter Editor** ダイアログに表示される用指定します。
6. パラメータの追加を終了するには、**OK** をクリックし、ダイアログを閉じます。

注：1 つ以上のパラメータ定義をコピーには、右クリックし、**[Copy Parameter Definitions] (Ctrl + C)** を選択します。右クリックして、**[Paste]**、**[Paste Parameter Definitions (Ctrl + V)]** を選択し、ペーストします。

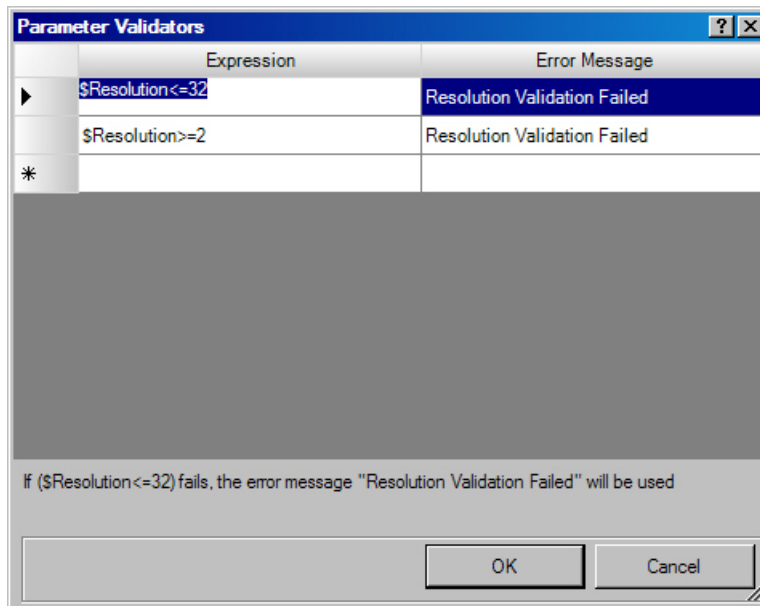
3.3 パラメータの妥当性検証式の追加

パラメータ検証式は、インスタンスのコンテキスト内で評価されます。検証式は、フォーマルおよびローカルパラメータを参照することができます。

パラメータの妥当性検証式を追加するには：

1. パラメータ定義ダイアログで、**[Validators]** フィールドをクリックし、**[省略 (...)]** ボタンをクリックします。

Parameter Validators ダイアログが表示されます。

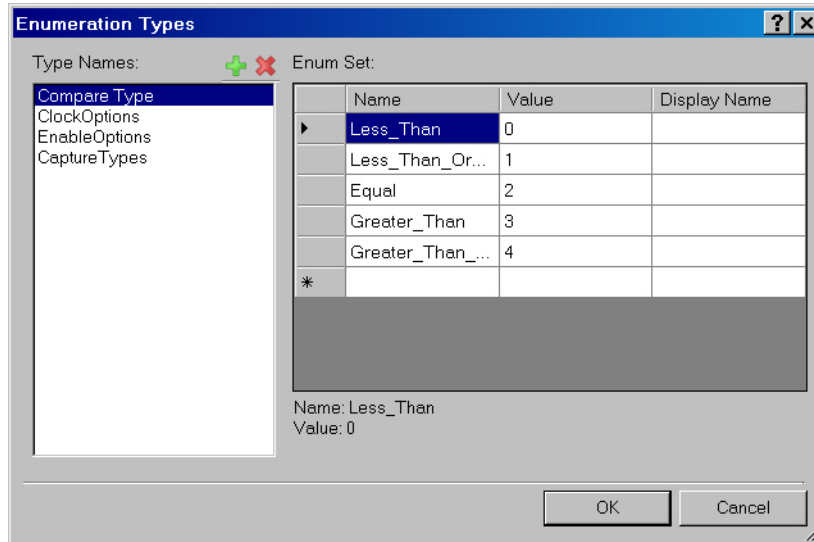


2. **[Expression]** フィールド内で、パラメータ検証式を打ち込みます。**\$** シンボルを使用することで、パラメータ名を参照することができます。例：
`$param1 > 1 && $param < 16`
`($param1 == 8) || ($param1 == 16) || ($param1 == 24)`
 式エバリュエータに関する詳細情報については、付録 A を参照してください。
3. **[Error Message]** フィールド内で、検証チェックで不適合だった場合のメッセージを打ち込みます。
4. 別のバリデータを追加するには、式フィールドの、>* マークのある空行をクリックし、必要に応じてフィールドに入力します。
5. 式を削除するには、削除する行の、最初の列の > シンボルをクリックし、**[削除]** キーを押します。
6. ダイアログを閉じるには、**OK** をクリックします。

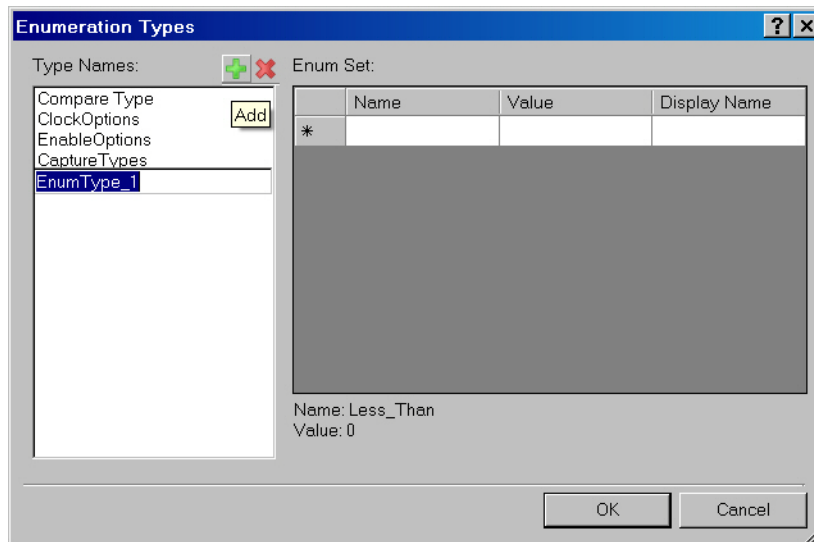
3.4 ユーザー定義型の追加

ユーザー定義型を作成するには：

1. Parameters Definition ダイアログの **[Types..]** ボタンをクリックし、Enumeration Types ダイアログを開きます。



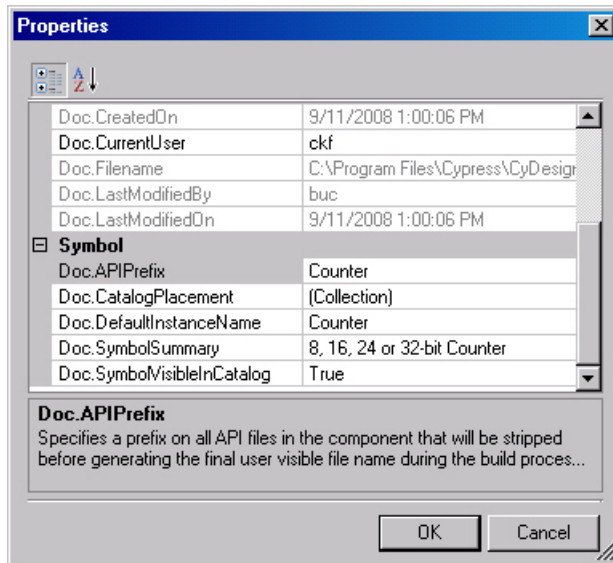
2. **Type Names** で、**Add** をクリックし、新しい列挙タイプを作成します。



3. EnumType_1 を使用したい名前に変更し、**[Enter]** をタイプします。
4. **[Enum Set]** で、**[Name]** 下の最初の行内をクリックし、最初の列挙タイプの名前入力して **[Value]** 下で値を入力するか、デフォルトを使用します。
5. もしくは、Parameters Definition ダイアログ内の **[Value]** プルダウンメニューに表示される **[Display Name]** 内に文字列を入力します。
 注：式として認識される可能性があるような、句読点の使い方をしないでください。例えば、"Clock + UpCnt & DwnCnt" のかわりに、"Clock with UpCnt & DwnCnt" を使用してください。
6. 必要なだけの列挙セットを入力し、**OK** をクリックして、Enumeration Types ダイアログを閉じます。

3.5 文書のプロパティの指定

各シンボルには、シンボルの様々な側面を定義する、プロパティのセットが含まれています。[プロパティ] ダイアログを開くには、シンボルキャンバスで右クリックし、**[Property]** を選択します。

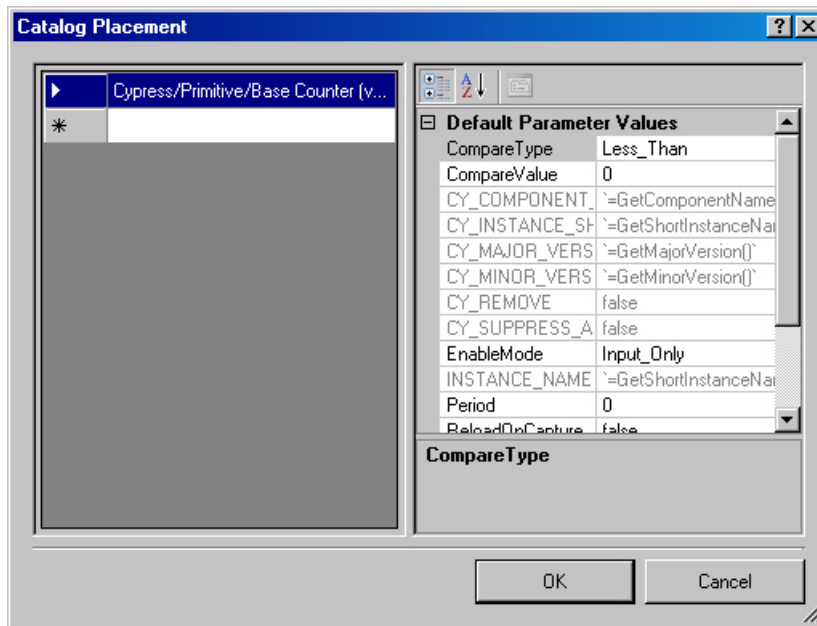


各コンポーネントについて、以下のドキュメントプロパティを指定することができます：

- **Doc.APIPrefix** -- コード生成中に、最終的にユーザーに見えるファイル名を作成する前に、コンポーネント内の全ての API ファイルから取り除く、接頭辞を指定します。例として、カウンターコンポーネント上に表示されるよう“Counter”を入力した場合、インスタンス名“Counter_1”を含むコンポーネント用の作成ヘッダーファイルは、*Counter_1.h* となります。このプロパティ内に何も入力しない場合、生成されるファイルは、*Counter_1_Counter.h* となります。
注：APIPrefix は、ディスクのプロジェクトの一部である API ファイルにのみ適用されます。API カスタマイザーが作成する API ファイルには適用されません。
- **Doc.CatalogPlacement** -- コンポーネントが、コンポーネントカタログ内でどのように表示されるかを定義します。27 ページの [カタログの配置の定義](#) を参照してください。
- **Doc.DefaultInstanceName** -- コンポーネントのデフォルトのインスタンス名を定義します。空欄にした場合、インスタンス名はデフォルトのコンポーネント名になります。
- **Doc.SymbolSummary** -- コンポーネントカタログ内で表示される、簡単な説明を入力するのに使用します。
- **Doc.SymbolVisibleInCatalog** -- コンポーネントがコンポーネントカタログ内で表示されるかどうかを、True か False で指定します。

3.5.1 カタログの配置の定義

カタログ配置ダイアログを使用して、さまざまなタブおよびツリー下のコンポーネントカタログ内で、シンボルがどのように表示されるかを定義することができます。



カタログ配置情報を定義しない場合、シンボルはデフォルトで、**[Default]>[Component]** 下に表示されます。

1. プロパティダイアログ内の **Doc.CatalogPlacement** フィールド内の省略 (...) ボタンをクリックし、ダイアログを開きます。
2. ダイアログの左側で、テーブルの最初の行に、以下のシンタックスで配置の定義を入力します：
t/x/x/x/x/d

t -- タブ名シンボルカタログ内で、タブはアルファベット順で表示され、大文字小文字の区別はありません。

x -- ツリー内のノード少なくとも 1 つのノードを必要とします。

d -- シンボルの表示名 (オプション) 表示名を指定しない場合、シンボル名が使用されますが、このシンタックスを使用しなければなりません :t/x/。

例えば、デフォルトコンポーネントカタログ内のコンポーネント **PGA**, において、タブ名 (t) は **Cypress**、最初のノード (x) は **Analog**、次のノード (x) は **Amplifiers**、表示名 (d) は **PGA** の場合、Cypress/Analog/Amplifiers/PGA となります。

3. ダイアログ右側のデフォルトパラメータ値で、必要に応じて、この特定のカタログ配置定義用のデフォルトパラメータ値を入力します。

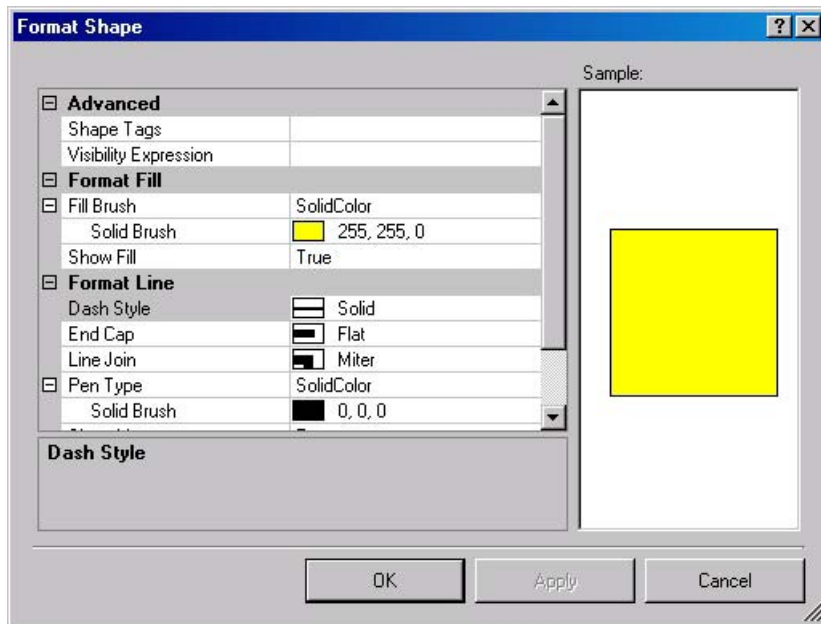
注：デフォルトパラメータ値は、コンポーネントが最初にインスタンス化された時点の値のみを定義します。値は、その時点のスナップショットです。コンポーネントのデフォルト値が次に変更された場合、その値は既にインスタンス化されたコンポーネントに対して伝達されることはありません。

4. 必要に応じて、複数のカタログ配置を定義するために、左側テーブル内の行を追加します。この場合、追加された全ての行につき、同じ、または異なるデフォルトパラメータ値を設定します。
5. **OK** をクリックし、ダイアログを閉じます。

3.6 形のプロパティの定義

Format Shape ダイアログは、形のプロパティを定義するために使用されます。利用可能なプロパティのタイプは、選択された形により異なります。例えば、テキストでは、フォント、色、サイズなどであり、線では、幅、ペンタイプ、キャップなどです。

このダイアログを開くには、1 つまたは複数の形を選択し、フォーマットシェイプ  ボタンをクリックします。



3.6.1 一般的な形のプロパティ

一般的な形のプロパティの大半は一目了然であり、他のワープロまたは作画プログラムのシェイプフォーマットと同様のものです。これらのプロパティのほとんどについては、プルダウンメニューから値を選択します。

3.6.2 高度な形のプロパティ

インスタンスターミナルなどのいくつかの形では、以下のような詳細プロパティがあります。

- **Default Expression** -- 値のデフォルト値を、`<サイズ>=<値>` で定義します。ここで、`<サイズ>` = ビット数、`<値>` = バイナリ値です。
- **Shape Tags** -- シェイプカスタマイズコードを使用するために、1 つ以上の形に関連する、1 つ以上の複数のタグを定義します。[61 ページのコンポーネントのカスタマイズ](#)を参照してください。
- **Visibility Expression** -- 選択した形が表示されるか否かを評価するための式を定義します。例えば、UART コンポーネントでは、このプロパティは、`rts_n` および `cts_n` ターミナル用の変数 `$FlowControl` で定義されます。`$FlowControl` がインスタンス内で真に設定された場合、これらのターミナルは回路図内に表示されます。偽の場合、表示されません。

4. 実装の追加



コンポーネントの実装には、デバイス、およびデバイスのファミリーにどうコンポーネントが実装されるかが、指定されています。コンポーネントには、1 つ以上の実装が必要です。コンポーネントを実装するには、いくつか方法があります：回路図、Verilog、そしてソフトウェアです。この章では、これらの方法を順に取り扱います。

4.1 回路図を用いた実装

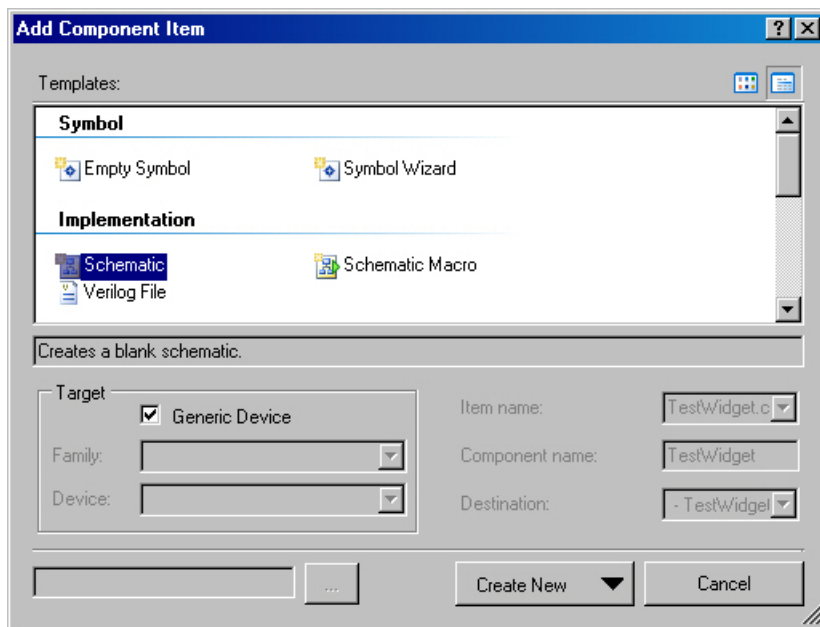
回路図は、コンポーネントの内部の接続を示す、グラフィックな図です。コンポーネントの作成を終了し、デザイン内でインスタンス化し、ビルドした時点で (79 ページのコンポーネントの完成参照)、Warp およびフィッタ等が使用するための情報を提供する、必要なファイルが生成されます。

4.1.1 回路図の追加

コンポーネントに回路図ファイルを追加するには：

1. コンポーネントを右クリックし、**Add Component Item** を選択します。

[Add Component Item] ダイアログが表示されます。



2. Schematic アイコンを選択します。

注 回路図はコンポーネント名を継承します。

3. **Target** 下で、ドロップダウンメニューからファミリーまたはデバイスを指定するか、**Generic Devices** を選択します。

回路図は特定のファミリーもしくはデバイスに対する実装であってもよいし、どのデバイスにも対応する実装であってもかまいません。このため、**Target** オプションを選択することができます。

4. **OK** をクリックします。

新しいコンポーネントが **Workspace Explorer** ツリー内の、選択されたデバイスオプションにとって適切なディレクトリに、回路図ファイル (.cysch) として表示されます。

.cysch ファイルは **Schematic Editor**(回路図エディタ) によって開くことができます。この時のコンポーネント接続性を示す図を書くことができます。

4.1.2 回路図の完成

Schematic Editor(回路図エディタ) の仕様の詳細については、**PSoC Creator** ヘルプの "Using Design Entry Tools > Schematic Editor(回路図エディタ)" を参照してください。

4.1.2.1 デザインワイド リソース (Design-Wide Resources) (DWR) の設定

DWR の設定は、DWR コンポーネントインスタンス (クロック、割り込み、DMA など) に付随します。

- これらは、自動的に生成された内部 ID に関連付けられます。
- また、階層インスタンス名にも関連付けられます。

DWR を用いてコンポーネントを作成する場合、問題なく名前を変更することができます。内部 ID は、関連付けの管理のために使用されます。

コンポーネントから DWR を削除した場合、システムは階層インスタンス名の使用に戻ります。新しい DWR を追加して、名前を変更した場合、エンドユーザーが DWR コンポーネントに関連付けた設定は、.cydwr ファイルから失われます。

4.2 回路図マクロの作成

回路図マクロは、既存のコンポーネント、ピン、およびクロックといった、複数の要素をもつコンポーネントの実装を可能にする、ミニ回路図です。コンポーネントは、複数のマクロを持つことができます。マクロは、インスタンス (マクロが定義されているインスタンスも含む)、ターミナル、およびワイヤを持つことができます。

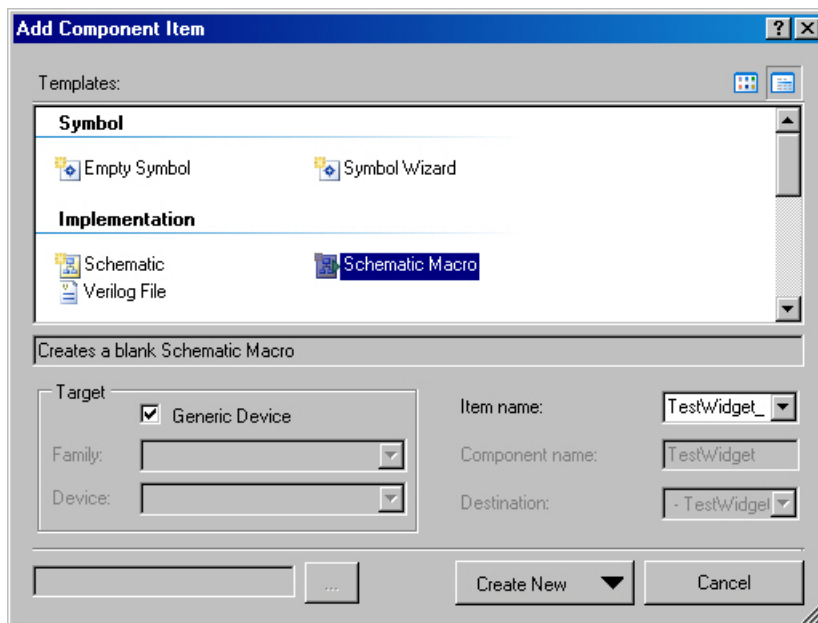
回路図マクロは一般的に、コンポーネントの使用を簡単にするために作成されます。回路図における一般的なコンポーネントの仕様法は、マクロとして作成および提供されます。エンドユーザーは、シンボルを直接使用する代わりにマクロを使用します。

4.2.1 回路図マクロの追加

回路図をコンポーネントに加えるには、他のコンポーネントと同様の方法で加えます。コンポーネントに回路図マクロファイルを追加するには：

1. コンポーネントを右クリックし、**Add Component Item** を選択します。

[Add Component Item] ダイアログが表示されます。



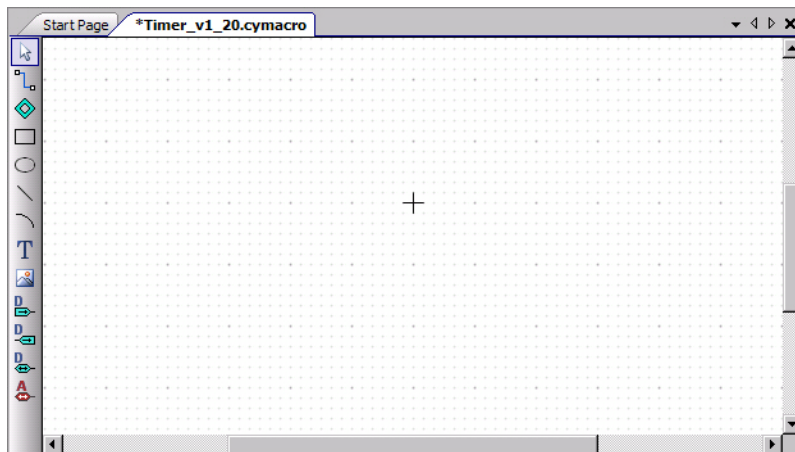
2. Schematic Macro アイコンを選択します。
3. **Target** 下で、ドロップダウンメニューからファミリーまたはデバイスを指定するか、**Generic Devices** を選択します。

回路図マクロは、コンポーネントに「所属」します。マクロは、デバイス、システム、もしくは汎用レベルで作成できます。同じレベルに複数のマクロが存在することが可能です。これは、コンポーネントの他の区分 (回路図、Verilog など) と異なります。マクロファイル名は、コンポーネント内でユニークなものにします。

4. **OK** をクリックします。

新しいコンポーネントは **Workspace Explorer** ツリー内の、選択されたデバイスオプションに適切なディレクトリに、回路図マクロファイル (.cymacro) として表示されます。

.cymacro ファイルは **Schematic Macro Editor (回路図マクロエディタ)** によって開くことができ、コンポーネントの追加、デフォルトクロック周波数などの既定の設定の定義、各コンポーネントのデフォルト設定の上書き、などを行えます。また、マクロを使用して、通信インターフェースまたはアナログコンポーネントに必要な、I/O ピン設定を事前に定義することができます。

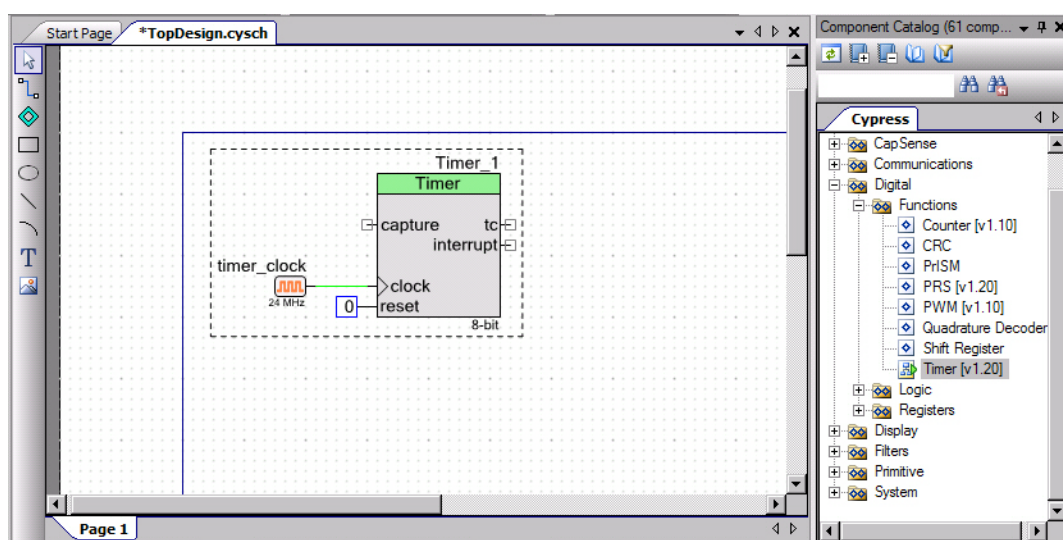


4.2.2 マクロの定義

以下は、Timer (タイマー) マクロの作成例です：

1. カタログから、Timer (タイマー) コンポーネントを取り出し、配置します。
必要ならば、Timer (タイマー) コンポーネントのパラメータ値を設定し、デフォルト設定を上書きします。使用可能なパラメータについては、コンポーネントデータシートを参照してください。
2. ライブラリから、クロックコンポーネントを取り出し、配置します。また、必要に応じて設定します。
3. クロック出力を、Timer (タイマー) コンポーネントの "clock" 入力ターミナルに接続します。
4. 回路図に "Logic Low (ロジック ロー)" コンポーネントを配置し、Timer (タイマー) コンポーネントの "reset" 入力ターミナルに接続します。
5. 回路図マクロ上で右クリックし、ドロップダウンメニューから**プロパティ**を選択します。
コンポーネントカタログの配置と概要を、求めるセッティングに変更します。
6. Timer (タイマー) シンボルファイルを開き、空いているスペースのどこかで右クリックし、ドロップダウンメニューから**プロパティ**を選択します。コンポーネントカタログから、コンポーネントを [隠す] 設定にします。

デザインプロジェクトにおいて、以下が見られます：



7. これをデザインに配置した後、ロジックローの消去、およびクロックまたはタイマー設定の変更などのテストを行ってください。

4.2.3 バージョン付け

前述のように、マクロはコンポーネントに所属するため、コンポーネントのバージョン付けに対応して、默示的にバージョン付けされています。マクロ名には、マクロおよびコンポーネントのバージョン名を含めません。

4.2.4 コンポーネント アップデートツール

コンポーネント アップデートツールは、回路図上のコンポーネントのインスタンスのアップデートに使用されます。マクロが回路図上に配置されたとき、配置された要素の「マクロらしさ」が消えます。マクロの各パーツは、個別の回路図要素になります。回路図上ではマクロの「インスタンス」が存在しないため、コンポーネント アップデートツールは何もアップデートできません。

しかし、回路図マクロのみは、回路図として定義されます。この回路図は、コンポーネント アップデートツールを用いてアップロード可能な、他のコンポーネントのインスタンスを含むことがあります。マクロ定義回路図はコンポーネント アップデートツールにとって可視であり、全ての適切なアップデートが可能です。

4.2.5 マクロファイルの名づけの慣例

マクロファイルの拡張子は、".cymacro" です。デフォルトのファイル名は、< コンポーネント名 >_<[0-9][0-9]>.cymacro です。

4.2.5.1 同じ名前のマクロとシンボル

同じ名前のマクロとシンボルを作成することは、エンドユーザーの混乱を招くため、推奨されません。

4.2.6 文書のプロパティ

回路図マクロは、シンボルと同じ特性を多く持ちます。詳細情報については、[26 ページの文書のプロパティの指定](#)を参照してください。

4.2.6.1 コンポーネントのカatalogにおける配置

マクロのカatalog配置プロパティは、配置に関連するパラメータがないことを除けば、シンボルのCatalog配置と似ています。Catalog配置エディタを用いて、マクロをコンポーネントCatalogの複数個所に配置することができます。

回路図マクロは、シンボルと同じように、コンポーネントCatalogにリストされます。マクロ表示名が他のコンポーネントでも存在した場合、マクロは括弧つきで表示され、同じ表示名のマクロのインスタンスが 2 つ以上存在することを示します。

4.2.6.2 サマリーテキスト

PSoC Creator では、シンボルは、コンポーネントCatalogで選択された場合にコンポーネント プレビューエリアに表示される、関連する「サマリーテキスト」を持つことができます。マクロについても、この機能がサポートされます。回路図マクロエディタにサマリーテキスト フィールドが存在し、マクロがコンポーネントCatalogで選択された場合、テキストはプレビューエリアに表示されます (シンボルと同じ取り扱い)。

4.2.6.3 隠れたプロパティ

マクロは、シンボルと同じ方法で隠れたプロパティをサポートします。隠れたプロパティは、マクロエディタで編集可能です。

4.2.7 マクロ データシート

回路図マクロは、独自のデータシートを持ちません。代わりに、特定のコンポーネントに定義される全てのマクロは、そのコンポーネントのデータシートの、別の節に文書化されます。コンポーネントCatalogで選択された場合、マクロのプレビューエリアに、コンポーネントのデータシートへのリンクが表示されます。

4.2.8 マクロのポストプロセッシング

マクロが回路図にドロップされた場合、インスタンス、ターミナル、およびワイヤの名前が再計算されます。ポストプロセッシングの終了時に、インスタンス、ターミナル、およびワイヤの名前は、回路図の既存の名前と重複してはいけません。

マクロの各インスタンス、ターミナル、およびワイヤに対し、PSoC Creator は、モデルに追加する前に、重複しない名前を割り当てます。各インスタンス、ターミナル、およびワイヤは、その名前の HDL 名が回路図にすでに使用されていないことを確認してから、回路図に追加されます。アイテムのベース名の終わりに番号が付けられ、この番号は独自の HDL 名が得られるまで増やされます。

4.2.9 例

インスタンス 1 つ ("i2c_master" という名前)、ターミナル 3 つ (それぞれ "sclk"、"sdata"、および "clk" という名前)、およびワイヤ 3 つ (ワイヤ自体の名前がないものの、それぞれの実質的な名前は sclk、sdata、および clk) のマクロがあるとします。回路図が空であり、マクロが回路図にドロップされた場合、インスタンス、ターミナル、およびワイヤ名は、以下になります：

- インスタンス : i2c_master
- ターミナル : sclk、sdata、および clk
- ワイヤ : sclk、sdata、および clk

マクロが再度ドロップされた場合、名前は以下になります：

- インスタンス : i2c_master_1
- ターミナル : sclk_1、sdata_1、および clk_1
- ワイヤ : sclk_1、sdata_1、および clk_1

これらの名前が既に使用されている場合、この接尾辞の数字は有効な名前が得られるまでインクリメントされます。

4.3 Verilog を用いた実装

Verilog を用いて、コンポーネントの機能を記載することができます。PSoC Creator を使用して、コンポーネントに Verilog ファイルを追加し、書き込みおよび編集することが可能です。Verilog を使用する場合、いくつかの要求項目があり、また PSoC Creator は Verilog 言語の合成可能部分のみをサポートすることに留意してください。Warp Verilog レファレンスガイドを参照してください。データパスインスタンスの設定を編集するには、データパス設定ツール (Datapath Configuration Tool) を使用する必要があるかもしれません。125 ページのデータパス設定ツールを参照してください。

注 Verilog に埋め込まれたプリミティブには、API が自動的に生成されません。適切な #define 値は生成されるものの、例えば Verilog に埋め込まれた cy_psoc3_control に対して、回路図にコントロールレジスタ (Control Register) が置かれたときに通常生成される API は、生成されません。

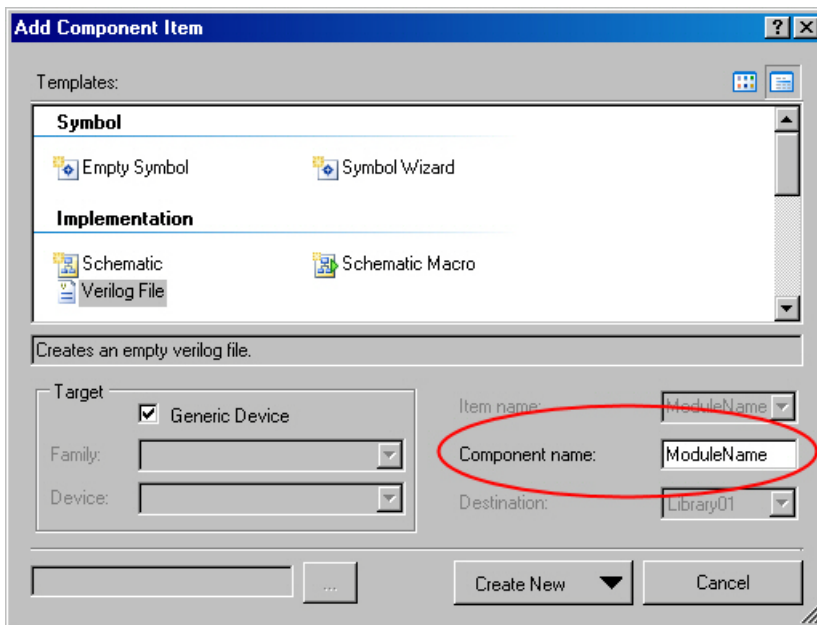
4.3.1 Verilog のファイル要求項目

PSoC Creator では、1 コンポーネントで使用できる Verilog ファイルは 1 個に限られており、PSoC Creator は Verilog 言語の合成可能な部分のみをサポートします。

PSoC Creator 内でシンボルとともに作成されたコンポーネントの名前は、Verilog ファイルで定義されたモジュール名と完全に合致する必要があります。PSoC Creator は、コンポーネントと同じ名前の Verilog ファイルを作成します。モジュール名と同じ、この名前を使用する必要があります。

たとえば、PSoC Creator にて作成された "ModuleName" コンポーネントは、デジタルコンポーネントの開発用に ModuleName.v ファイルを生成します。PSoC Creator で正しくビルドされるためには、このファイルのトップモジュールは "ModuleName"(大文字小文字を区別する) でなければなりません。

次の例では、コンポーネント名テキストボックスに入力されたテキスト ("ModuleName") は、Verilog ファイルにて使用されたモジュール名 ["module ModuleName(...)"] と一致する必要があります。



上の例のように、プロセスを始めるためにシンボルを作成する時点で、命名規則を考えておく必要があります。

Cypress は、コンポーネント Verilog ファイルで必要と思われる定数をすべて含むファイルを用意しています。これらの定数は、**cypress.v** に存在します。このファイルは、**PSoC Creator** 内の全ての Verilog コンパイルにおいて、サーチパスに含まれます。全ての Verilog ファイルは **cypress.v** ファイルを使用するため、以下の行がモジュールの宣言前に存在する必要があります。

```
`include "cypress.v"
```

4.3.2 Verilog ファイルの追加

4.3.2.1 新規のファイル

既存の Verilog ファイルが存在しない場合、**Verilog の生成**  ツールを使用できます。このツールは、どうシンボルを定義したかに基づいて、適切なファイル名、`'include "cypress.v"`、および基本的なモジュールの情報を含んだ、基本的な Verilog ファイルを作成します。

4.3.2.2 既存のファイル

既存の Verilog ファイルが存在する場合、これをコンポーネントアイテムとして追加できます。

1. コンポーネントを右クリックし、**Add Component Item** を選択します。

[Add Component Item] ダイアログが表示されます。

2. Verilog File アイコンを選択します。

注 Verilog ファイルは、コンポーネント名を継承します。[34 ページの Verilog のファイル要求項目](#)を参照してください。

3. 必要ならば、**Create New** ボタンを、**Add Existing** ヘトグルします。
 4. 省略ボタン [...] をクリックし、ファイルの位置に移動し、**Open** をクリックします。
 5. **Add Existing** をクリックします。
- この Verilog ファイルは、適切なファイル名をとり、**Workspace Explorer** に追加されます。

4.3.3 Verilog ファイルの完成

Verilog ファイルをダブルクリックして開き、必要に応じてファイルを完成させます。

注 Verilog 実装のモジュール名はコンポーネント名と合致する必要があります (大文字小文字を区別します)。

UDB ベースの PSoC デバイスについては、Verilog 実装は Verilog2005 デザインに対応し、アーキテクチャに存在する PLD にコンパイルします。なお、貴重な PLD リソースの消費を防ぐため、最適化された他のモジュールが存在します。これらのモジュールと使用法は、以下の数節にまとめてあります。

4.3.4 UDB アレイの変更

4.3.4.1 シリコンの版の定義

複数の UDB アレイの変更をサポートし、過去のシリコンの版との互換性を保つためには、コンポーネントがデバイスの系統とシリコンの版を特定する必要があるかもしれません。Verilog コンポーネントに、デバイスの系統とシリコンの版の情報を提供するため、ネットリストされたデザイン (*project.v* ファイル) に、以下の **define** がビルド時に生成されます。これらを使用することができます：

```
`define CYDEV_CHIP_FAMILY_UNKNOWN      0
`define CYDEV_CHIP_MEMBER_UNKNOWN      0
`define CYDEV_CHIP_FAMILY_PSOC3        1
`define CYDEV_CHIP_MEMBER_3A           1
`define CYDEV_CHIP_REVISION_3A_ES1     0
`define CYDEV_CHIP_REVISION_3A_ES2     1
`define CYDEV_CHIP_REVISION_3A_ES3     3
`define CYDEV_CHIP_FAMILY_PSOC5        2
`define CYDEV_CHIP_MEMBER_5A           2
`define CYDEV_CHIP_REVISION_5A_ES0     0
`define CYDEV_CHIP_REVISION_5A_ES1     1
`define CYDEV_CHIP_FAMILY_USED         1
`define CYDEV_CHIP_MEMBER_USED         1
`define CYDEV_CHIP_REVISION_USED       1
```

これらの **define** は、*cyfitter.h* の C **define** に合致するようアップデートされています。

これらの **define** に基づき、Verilog コンポーネントにおける 2 つのローカルパラメータ ([49 ページのローカルパラメータの使用と名前付きパラメータ参照](#)) を用いて、異なるデバイスの系統とシリコンの版において、コンポーネントの設定を正しく維持することができます

名前	式
PSoC3_ES2	`CYDEV_CHIP_FAMILY_USED == `CYDEV_CHIP_FAMILY_PSOC3 && `CYDEV_CHIP_REVISION_USED <= `CYDEV_CHIP_REVISION_3A_ES2
PSoC5_ES1	`CYDEV_CHIP_FAMILY_USED == `CYDEV_CHIP_FAMILY_PSOC5 && `CYDEV_CHIP_REVISION_USED <= `CYDEV_CHIP_REVISION_5A_ES1

以下の例は、"PSoC3_ES2" もしくはこれより古いバージョンと、ES2 以降の PSoC 3 を区別する方法です。同様に、"PSoC5_ES1" を用いて、PSoC 5 か、より古いシリコンかどうかを区別できます。

4.3.4.2 異なるシリコンの版におけるコンポーネントの設定

説明されたローカルパラメータは、Verilog 実装の生成されたブロック内で使用されます。以下は、**I2S_v1_50** コンポーネントにおける例です。コントロールレジスタブロックのインスタンス化における、異なるモードを表しています。(例えば、**ES2** ではダイレクトモード、**ES3** では同期モード):

```

/*****
* 異なるデバイスの系統とシリコンの版の定義
*****/

/* PSoC 3 ES2 もしくはそれ以前 */
localparam PSOC3_ES2 = (`CYDEV_CHIP_FAMILY_USED == `CYDEV_CHIP_FAMILY_PSOC3 &&
                        `CYDEV_CHIP_REVISION_USED <= `CYDEV_CHIP_REVISION_3A_ES2

/* PSoC 5 ES1 もしくはそれ以前 */
localparam PSOC5_ES1 = (`CYDEV_CHIP_FAMILY_USED == `CYDEV_CHIP_FAMILY_PSOC5 &&
                        `CYDEV_CHIP_REVISION_USED <= `CYDEV_CHIP_REVISION_5A_ES1);

/* PSoC 3 ES3 版に対する同期モードのサポートを追加 */
generate
if(PSOC3_ES2 || PSOC5_ES1)
begin: AsyncCtl
    cy_psoc3_control #(.cy_force_order(1)) ControlReg
    (
        /* 出力 [07:00] */ .control(ctrl1)
    );
end /* AsyncCtl */
else
begin: SyncCtl
    cy_psoc3_control #(.cy_force_order(1), .cy_ctrl_mode_1(8'h0),
                      .cy_ctrl_mode_0(8'h7)) ControlReg
    (
        /* 入力 */ .clock(clock),
        /* 出力 [07:00] */ .control(ctrl1)
    );
end /* SyncCtl */
endgenerate

```

上の例における 2 つの生成ブロックは、シリコンの版に応じて、**cyfitter.h** において異なるコントロールレジスタの定義を行います。定義は以下の通りです:

- **ES2:** \$INSTANCE_NAME_AsyncCtl_ControlReg__CONTROL_REG
- **ES3:** \$INSTANCE_NAME_SyncCtl_ControlReg__CONTROL_REG

これをコンポーネントの API(ヘッダファイル)で考慮するため、レジスタの定義に Verilog ファイル内で **#ifdef** ディレクティブを使用するのではなく、コンポーネントシンボル内に置換文字列パラメータを使うことが推奨されます。これにより、レジスタ定義の条件的コンパイルによる生成が可能になり、ヘッダファイル内の **#ifdef** の数が減るので読みやすくなります。上述の例では、追加のパラメータが 1 つ必要です:

名前	型	式
CtlModeReplacementString	文字列型	`=((GetDeviceFamily() eq "PSoC3" && GetSiliconRevision() <= 1) (GetDeviceFamily() eq "PSoC5" && GetSiliconRevision() <= 1)) ? "AsyncCtl" "SyncCtl"

このパラメータは評価コンテキストであり、**GetDeviceFamily()** および **GetSiliconRevision()** 関数により評価されます。これらの関数は、エラーレクション時に有効な値のみを返します。

- **GetSiliconRevision()** は、シリコンの版を整数として返します。不明の場合、-1 を返します。この返り値は、シリコンの版に対する API サポートのために提供される **cyfitter.h** のものと一致します。

- `GetDeviceFamily()` は、"PSoC3"、"PSoC5"、もしくは不明の場合に空文字列を返します。
この式を読みやすくするために、2つの列挙型が用意されています：

PSoC3SiliconRev	
PSoC3_ES1	0
PSoC3_ES2	1
PSoC3_ES3	2

PSoC5SiliconRev	
PSoC5_ES1	1
PSoC5_ES2	2

最後に、コンポーネント API(ヘッダ) のコントロールレジスタが、以下のように定義されます：

```
#define `$_INSTANCE_NAME`_CONTROL_REG
(* (reg8 *) `$_INSTANCE_NAME`_`$CtlModeReplacementString`_ControlReg__CONTROL_REG)
#define `$_INSTANCE_NAME`_CONTROL_PTR
(* (reg8 *) `$_INSTANCE_NAME`_`$CtlModeReplacementString`_ControlReg__CONTROL_REG)
```

4.3.5 UDB 要素

PSoC Universal Digital Block (ユニバーサル デジタルブロック) (UDB) は、デザイン作成時に Verilog がアクセス可能な、いくつかのプログラム可能なコンポーネントを含みます。UDB に含まれているアイテムは、データパス、ステータス `/statusi` レジスタ、コントロールレジスタ、および `count7` カウンタです。クロックを必要とする使い方をする場合、UDB コンポーネントは、特別な Verilog コンポーネントにより駆動できます。このコンポーネントは、使用者が、UDB のそのコンポーネントに強制するクロック挙動を、フィッタに通知させることができます。

注 このガイドの多くの参照、およびコンポーネント名は、"psoc3" を指定しています。この参照は、PSoC 3 および PSoC 5 を含む、全ての UDB ベースの PSoC に当てはまります。

4.3.5.1 クロック / イネーブルの仕様

クロックを使用する UDB の部分においては、必要とするクロックの種類 (同期対非同期)、およびクロックのイネーブル信号を指定することができます。フィッタは、この情報をもとにクロックを監視し、要求されたタイプをイネーブルし、UDB 内のクロックの配線および実装に必要な変更を加えることで、出力クロックに、要求される特性を満たすよう保証します。

cy_psoc3_udb_clock_enable_v1_0

このプリミティブを用いて、出力クロックの、駆動する UDB コンポーネントにより要求される、特性を指定することができます。要素の出力は、UDB コンポーネントのみ駆動できます。他のコンポーネントを駆動しようとする、フィッタで DRC エラーが発生します。要素にはパラメータが 1 つあります：

- `sync_mode` : ブーリアン型のパラメータであり、得られるクロックが同期(真)か非同期(偽)かを指定します。デフォルトは、同期です。

要素を、下記のようにインスタンス化します：

```
cy_psoc3_udb_clock_enable_v1_0 #(.sync_mode(`TRUE)) MyCompClockSpec (
    .enable(), /* インターコネクトからのイネーブル */
    .clock_in(), /* インターコネクトからのクロック */
    .clock_out() /* このコンポーネントの UDB 要素に使われるクロック */
);
```

4.3.5.2 データパス

以下の説明では、データパスに関するハイレベルの説明と、使用法について記載されています。データパスについての詳細については、*TRM* を参照してください。

UDB ベースの PSoC デバイスのデータパスは、非常に小さい 8 ビット幅のプロセッサで、「コントロールストア」に 8 個の状態が定義されているようなものです。また、操作全体を定義するのに役立つ、5 つの静的設定 (Static Configuration) レジスタが存在します。データパス内のコントロールストアと静的設定 (Static Configuration) レジスタは、[125 ページのデータパス設定ツール](#)を用いて設定できます。以下に記載されているように、事前に定義されたモジュールを使用して連続したデータパスを繋げることで、より大きいデータ幅にて操作することができます。データパスは、以下の区分に分けられます：

- **ALU** -- 算術ロジックユニット (ALU) は、8 ビットデータに対し、以下の操作を行えます。複数のデータパスを組み合わせると 16、24、もしくは 32 ビットにする場合、操作はデータ幅全体に対して行われます。
 - パススルー
 - インクリメント (INC)
 - デクリメント (DEC)
 - 加算 (ADD)
 - 減算 (SUB)
 - XOR
 - AND
 - OR
- **シフト** -- ALU の出力は、以下の操作を行うことができるシフト演算子に送られます。複数のデータパスを組み合わせると 16、24、もしくは 32 ビットにする場合、操作はデータ幅全体に対して行われます。
 - パススルー
 - 左にシフト
 - 右にシフト
 - ニブルのスワップ
- **マスク** -- シフト演算子の出力は、データパスの 8 ビットのいずれも任意でマスクすることができます。マスク演算子に送られます。
- **レジスタ** -- データパスは、ハードウェアと CPU に対し、以下のレジスタを提供します。静的設定 (Static Configuration) レジスタには、さまざまな設定オプションを定義することができます。
 - 2 つのアキュムレータ レジスタ : ACC0 および ACC1
 - 2 つのデータレジスタ : DATA0 および DATA1
 - 2 つの、深度 4 バイトの FIFO: 複数の操作モードに対応する、FIFO0 および FIFO1
- **比較演算子**
 - 0 の検出 : Z0 および Z1 は、ACC0 および ACC1 をそれぞれ 0 と比較し、バイナリ値の真偽を、必要に応じてハードウェアが使用できるようにインターコネクトロジックに出力します。
 - FF の検出 : FF0 および FF1 は、ACC0 および ACC1 をそれぞれ 0xFF と比較し、バイナリ値の真偽を、必要に応じてハードウェアが使用できるようにインターコネクトロジックに出力します。
 - 0 の比較 :

等価比較 -- (ACC0 & Cmask0) が DATA0 と等しいか比較し、バイナリ値の真偽を、必要に応じてハードウェアが使用できるようにインターコネクトロジックに出力します。

(Cmask0 は、静的設定 (Static Configuration) として設定可能です。)

未満比較 -- (ACC0 & Cmask0) が DATA0 未満か比較し、バイナリ値の真偽を、必要に応じてハードウェアが使用できるようにインターコネクトロジックに出力します。(Cmask0 は、静的設定 (Static Configuration) として設定可能です。)

□ 1 の比較:

等価比較 (ce1) -- Compare (ACC0 or ACC1) & Cmask1) が (DATA1 or ACC0) と等しいか比較し、バイナリ値の真偽を、必要に応じてハードウェアが使用できるようにインターコネクトロジックに出力します。(Cmask1 は、静的設定 (Static Configuration) として設定可能です。)

未満比較 (cl1) -- (ACC0 & Cmask0) が DATA0 未満か比較し、バイナリ値の真偽を、必要に応じてハードウェアが使用できるようにインターコネクトロジックに出力します。

(Cmask1 は、静的設定 (Static Configuration) として設定可能です。)

- オーバーフローの検出: msb がオーバーフローしたことを、バイナリ値の真偽を ov_msb 出力に駆動することで通知し、必要に応じてハードウェアが使用できるようにインターコネクトロジックに出力します。

データパスは、デザインされる殆ど全てのコンポーネントに頻繁に見られる、多くの異なる設定に対応できます。データパス内の機能で、Verilog により実装できるものは、PLD 内に納められます。ただし、データパスが固定ブロックであるのに対し、PLD はすぐに消費されます。データパス数と使用可能な PLD 数の間に、常にトレードオフがあります。どちらがより貴重かは、設計者が判断します。FIFO などの、いくつかの機能は、PLD 内に実装できません。

デザイン内でデータパスをインスタンス化するか、複数の連続したデータパスを使用する場合、Verilog モジュール内で以下のインスタンス化モジュールのいずれかを利用してください。

cy_psoc3_dp

これは、ベースとなるデータパス要素です。このベース要素の上には複数の他の要素が存在しています。この要素を選択する前にこれらの要素を使用してください。この要素で利用可能な全ての I/O は、インスタンス化したものの中に列記されています。しかし、UDB ベースの PSoC デバイスのアーキテクチャの制限により、これらの信号の多くは、デザインに接続できるワイヤ数が限られています。このため、単一のデータパスが必要な場合は、常に cy_psoc3_dp8 モジュールを使用することが推奨されます。

各データパスは、モジュール自身の名前付きパラメータとして、いくつかのパラメータを渡すことができます。全てのデータパスにおいて、Verilog ファイル内でパラメータを実装するために使用すべきメソッドはデータパス設定ツールです。このツールは Verilog ファイルを読み込み、全てのデータパスを表示し、コンパイラ (Warp) 用に正しい情報を Verilog ファイルに保存します。

パラメータは:

- cy_dpconfig: 「コントロールストア」および静的設定 (Static Configuration) を含むデータパスの設定です。デフォルト値は {128'h0, 32'hFFF0FFFF, 48'h0} です。
- d0_init: DATA0 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- d1_init: DATA1 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- a0_init: ACC0 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- a1_init: ACC1 レジスタの初期化時の値です。デフォルト値は 8'b0 です。

このデータパスを、以下の例のようにインスタンス化します：

```
cy_psoc3_dp DatapathName(
/* 入力          */ .clk(),           // クロック
/* 入力  [02:00]  */ .cs_addr(),       // コントロールストアの RAM アドレス
/* 入力          */ .route_si(),      // ルーティングによるシフト入力
/* 入力          */ .route_ci(),      // ルーティングによるキャリー入力
/* 入力          */ .f0_load(),       // FIFO 0 を開く
/* 入力          */ .f1_load(),       // FIFO 1 を開く
/* 入力          */ .d0_load(),       // データレジスタ 0 を開く
/* 入力          */ .d1_load(),       // データレジスタ 1 を開く
/* 出力          */ .ce0(),           // アキュムレータ 0 = データレジスタ 0
/* 出力          */ .cl0(),           // アキュムレータ 0 < データレジスタ 0
/* 出力          */ .z0(),            // アキュムレータ 0 = 0
/* 出力          */ .ff0(),           // アキュムレータ 0 = FF
/* 出力          */ .ce1(),           // アキュムレータ [0|1] = データレジスタ 1
/* 出力          */ .cl1(),           // アキュムレータ [0|1] < データレジスタ 1
/* 出力          */ .z1(),            // アキュムレータ 1 = 0
/* 出力          */ .ff1(),           // アキュムレータ 1 = FF
/* 出力          */ .ov_msb(),        // 操作オーバーフロー
/* 出力          */ .co_msb(),        // キャリー出力
/* 出力          */ .cmsb(),          // キャリー出力
/* 出力          */ .so(),            // シフト出力
/* 出力          */ .f0_bus_stat(),   // FIFO 0 ステータスを uP にする
/* 出力          */ .f0_blk_stat(),   // FIFO 0 ステータスを DP にする
/* 出力          */ .f1_bus_stat(),   // FIFO 1 ステータスを uP にする
/* 出力          */ .f1_blk_stat(),   // FIFO 1 ステータスを DP にする
/* 入力          */ .ci(),            // 前ステージからのキャリー入力
/* 出力          */ .co(),            // 次ステージへのキャリー出力
/* 入力          */ .sir(),           // 右側からのシフト入力
/* 出力          */ .sor(),           // 右側へのシフト出力
/* 入力          */ .sil(),           // 左側からのシフト入力
/* 出力          */ .sol(),           // 左側へのシフト出力
/* 入力          */ .msbi(),          // MSB チェーン入力
/* 出力          */ .msbo(),          // MSB チェーン出力
/* 入力  [01:00]  */ .cei(),           // 前ステージからの入力と等価比較
/* 出力  [01:00]  */ .ceo(),           // 後ステージへの出力と等価比較
/* 入力  [01:00]  */ .cli(),           // 前ステージからの入力と未満比較
/* 出力  [01:00]  */ .clo(),           // 後ステージへの出力と未満比較
/* 入力  [01:00]  */ .zi(),            // 前ステージからの入力に対し 0 検出
/* 出力  [01:00]  */ .zo(),            // 後ステージへの出力に対し 0 検出
/* 入力  [01:00]  */ .fi(),            // 前ステージからの入力に対し 0xFF 検出
/* 出力  [01:00]  */ .fo(),            // 後ステージへの出力に対し 0xFF 検出
/* 入力          */ .cfbi(),          // 前ステージからの CRC フィードバック入力
/* 出力          */ .cfbo(),          // 次ステージへの CRC フィードバック出力
/* 入力  [07:00]  */ .pi(),            // 並列データポート
/* 出力  [07:00]  */ .po(),            // 並列データポート
);
```

cy_psoc3_dp8

これは、単一の 8 ビット幅のデータパス要素です。

この要素は、各データパスにおいて、ベースのデータパス要素と同じパラメータを保有しています。全てのパラメータ末には、LSB データパスを示す場合、"_a" が連結されています。(以下のリストでは、LSB データパスにおいては X=a)。

- cy_dpconfig_X: 「コントロールストア」および静的設定 (Static Configuration) を含むデータパスの設定です。デフォルト値は {128'h0,32'hFFF0FFFF, 48'h0} です。
- d0_init_X: DATA0 レジスタの初期化時の値です。デフォルト値は 8'b0 です。

- **d1_init_X**: DATA1 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- **a0_init_X**: ACC0 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- **a1_init_X**: ACC1 レジスタの初期化時の値です。デフォルト値は 8'b0 です。

このデータパスを、以下の例のようにインスタンス化します：

```
cy_psoc3_dp8 DatapathName(
    /* 入力      */ /* .clk(),           // クロック
    /* 入力  [02:00] */ /* .cs_addr(),        // コントロールストアの RAM アドレス
    /* 入力      */ /* .route_si(),       // ルーティングによるシフト入力
    /* 入力      */ /* .route_ci(),       // ルーティングによるキャリー入力
    /* 入力      */ /* .f0_load(),        // FIFO 0を開く
    /* 入力      */ /* .f1_load(),        // FIFO 1を開く
    /* 入力      */ /* .d0_load(),        // データレジスタ 0を開く
    /* 入力      */ /* .d1_load(),        // データレジスタ 1を開く
    /* 出力      */ /* .ce0(),           // アキュムレータ 0 = データレジスタ 0
    /* 出力      */ /* .cl0(),           // アキュムレータ 0 < データレジスタ 0
    /* 出力      */ /* .z0(),           // アキュムレータ 0 = 0
    /* 出力      */ /* .ff0(),           // アキュムレータ 0 = FF
    /* 出力      */ /* .ce1(),           // アキュムレータ [0|1] = データレジスタ 1
    /* 出力      */ /* .cl1(),           // アキュムレータ [0|1] < データレジスタ 1
    /* 出力      */ /* .z1(),           // アキュムレータ 1 = 0
    /* 出力      */ /* .ff1(),           // アキュムレータ 1 = FF
    /* 出力      */ /* .ov_msb(),        // 操作オーバーフロー
    /* 出力      */ /* .co_msb(),        // キャリー出力
    /* 出力      */ /* .cmsb(),         // キャリー出力
    /* 出力      */ /* .so(),           // シフト出力
    /* 出力      */ /* .f0_bus_stat(),   // FIFO 0 ステータスを uP にする
    /* 出力      */ /* .f0_blk_stat(),   // FIFO 0 ステータスを DP にする
    /* 出力      */ /* .f1_bus_stat(),   // FIFO 1 ステータスを uP にする
    /* 出力      */ /* .f1_blk_stat(),   // FIFO 1 ステータスを DP にする
);
```

cy_psoc3_dp16

ここでは、2つの8ビット幅データベース要素が、2つの連続したデータパスとして用いられ、16ビット幅モジュールを構成しています。個別のデータパス間で、キャリー、シフト入力、シフト出力、およびフィードバック信号が直結しているため、ALU、シフト、およびマスク操作は、データの16ビット幅全てを対象として行われます。

この要素は、各データパスにおいて、ベースのデータパス要素と同じパラメータを保有しています。全てのパラメータ末には、LSB データパスを示す場合 "**a**" が、2つのデータパスのMSBを示す場合 "**b**" が連結されています。(以下のリストでは、LSB データパスにおいては X=a、MSB データパスにおいては X=b)

- **cy_dpconfig_X**: 「コントロールストア」および静的設定 (Static Configuration) を含むデータパスの設定です。デフォルト値は {128'h0,32'hFFF0FFFF, 48'h0} です。
- **d0_init_X**: DATA0 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- **d1_init_X**: DATA1 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- **a0_init_X**: ACC0 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- **a1_init_X**: ACC1 レジスタの初期化時の値です。デフォルト値は 8'b0 です。

このデータパスを、以下の例のようにインスタンス化します：

```
cy_psoc3_dp16 DatapathName(
/* 入力          */ /* .clk(),           // クロック
/* 入力      [02:00] */ /* .cs_addr(),        // コントロールストアの RAM アドレス
/* 入力          */ /* .route_si(),       // ルーティングによるシフト入力
/* 入力          */ /* .route_ci(),       // ルーティングによるキャリー入力
/* 入力          */ /* .f0_load(),        // FIFO 0 を開く
/* 入力          */ /* .f1_load(),        // FIFO 1 を開く
/* 入力          */ /* .d0_load(),        // データレジスタ 0 を開く
/* 入力          */ /* .d1_load(),        // データレジスタ 1 を開く
/* 出力      [01:00:00] */ /* .ce0(),           // アキュムレータ 0 = データレジスタ 0
/* 出力      [01:00:00] */ /* .cl0(),           // アキュムレータ 0 < データレジスタ 0
/* 出力      [01:00:00] */ /* .z0(),            // アキュムレータ 0 = 0
/* 出力      [01:00:00] */ /* .ff0(),           // アキュムレータ 0 = FF
/* 出力      [01:00:00] */ /* .ce1(),           // アキュムレータ [0|1] = データレジスタ 1
/* 出力      [01:00:00] */ /* .ce1(),           // アキュムレータ [0|1] < データレジスタ 1
/* 出力      [01:00:00] */ /* .z1(),            // アキュムレータ 1 = 0
/* 出力      [01:00:00] */ /* .ff1(),           // アキュムレータ 1 = FF
/* 出力      [01:00:00] */ /* .ov_msb(),        // 操作オーバーフロー
/* 出力      [01:00:00] */ /* .co_msb(),        // キャリー 出力
/* 出力      [01:00:00] */ /* .cmsb(),          // キャリー 出力
/* 出力      [01:00:00] */ /* .so(),            // シフト 出力
/* 出力      [01:00:00] */ /* .f0_bus_stat(),   // FIFO 0 ステータスを uP にする
/* 出力      [01:00:00] */ /* .f0_blk_stat(),   // FIFO 0 ステータスを DP にする
/* 出力      [01:00:00] */ /* .f1_bus_stat(),   // FIFO 1 ステータスを uP にする
/* 出力      [01:00:00] */ /* .f1_blk_stat(),   // FIFO 1 ステータスを DP にする
);
```

cy_psoc3_dp24

ここでは、3 つの 8 ビット幅データベース要素が、3 つの連続したデータパスとして用いられ、24 ビット幅モジュールを構成しています。個別のデータパス間で、キャリー、シフト入力、シフト出力、およびフィードバック信号が直結しているので、ALU、シフト、およびマスク操作は、データの 24 ビット幅全てを対象として行われます

この要素は、各データパスにおいて、ベースのデータパス要素と同じパラメータを保有しています。全てのパラメータ末には、LSB データパスを示す場合 "_a" が、中間のデータパスを示す場合 "_b" が、3 つのデータパスの MSB を示す場合 "_c" が連結されています。(以下のリストでは、LSB データパスにおいては X=a、中間のデータパスにおいては X=b、MSB データパスにおいては X=c)

- cy_dpconfig_X: 「コントロールストア」および静的設定 (Static Configuration) を含むデータパスの設定です。デフォルト値は {128'h0,32'hFF00FFFF, 48'h0} です。
- d0_init_X: DATA0 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- d1_init_X: DATA1 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- a0_init_X: ACC0 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- a1_init_X: ACC1 レジスタの初期化時の値です。デフォルト値は 8'b0 です。

このデータパスを、以下の例のようにインスタンス化します：

```
cy_psoc3_dp24 DatapathName(
    /* 入力          */ /* .clk(),           // クロック
    /* 入力    [02:00] */ /* .cs_addr(),        // コントロールストアの RAM アドレス
    /* 入力          */ /* .route_si(),       // ルーティングによるシフト入力
    /* 入力          */ /* .route_ci(),       // ルーティングによるキャリー入力
    /* 入力          */ /* .f0_load(),        // FIFO 0 を開く
    /* 入力          */ /* .f1_load(),        // FIFO 1 を開く
    /* 入力          */ /* .d0_load(),        // データレジスタ 0 を開く
    /* 入力          */ /* .d1_load(),        // データレジスタ 1 を開く
    /* 出力    [02:00:00] */ /* .ce0(),           // アキュムレータ 0 = データレジスタ 0
    /* 出力    [02:00:00] */ /* .cl0(),           // アキュムレータ 0 < データレジスタ 0
    /* 出力    [02:00:00] */ /* .z0(),            // アキュムレータ 0 = 0
    /* 出力    [02:00:00] */ /* .ff0(),           // アキュムレータ 0 = FF
    /* 出力    [02:00:00] */ /* .ce1(),           // アキュムレータ [0|1] = データレジスタ 1
    /* 出力    [02:00:00] */ /* .ce1(),           // アキュムレータ [0|1] < データレジスタ 1
    /* 出力    [02:00:00] */ /* .z1(),            // アキュムレータ 1 = 0
    /* 出力    [02:00:00] */ /* .ff1(),           // アキュムレータ 1 = FF
    /* 出力    [02:00:00] */ /* .ov_msb(),        // 操作オーバーフロー
    /* 出力    [02:00:00] */ /* .co_msb(),        // キャリー 出力
    /* 出力    [02:00:00] */ /* .cmsb(),          // キャリー 出力
    /* 出力    [02:00:00] */ /* .so(),            // シフト 出力
    /* 出力    [02:00:00] */ /* .f0_bus_stat(),   // FIFO 0 ステータスを uP にする
    /* 出力    [02:00:00] */ /* .f0_blk_stat(),   // FIFO 0 ステータスを DP にする
    /* 出力    [02:00:00] */ /* .f1_bus_stat(),   // FIFO 1 ステータスを uP にする
    /* 出力    [02:00:00] */ /* .f1_blk_stat(),   // FIFO 1 ステータスを DP にする
);
```

cy_psoc3_dp32

ここでは、4 つの 8 ビット幅データベース要素が、4 つの連続したデータパスとして用いられ、32 ビット幅モジュールを構成しています。個別のデータパス間で、キャリー、シフト入力、シフト出力、およびフィードバック信号が直結しているため、ALU、シフト、およびマスク操作は、データの 32 ビット幅全てを対象として行われます。

この要素は、各データパスにおいて、ベースのデータパス要素と同じパラメータを保有しています。全てのパラメータ末には、LSB データパスを示す場合 "_a" が、中の下のデータパスを示す場合 "_b" が、中の上のデータパスを示す場合 "_c" が、4 つのデータパスの MSB を示す場合 "_d" が連結されています。(以下のリストでは、LSB データパスにおいては X=a、中の下のデータパスにおいては X=b、中の上のデータパスにおいては X=c、MSB データパスにおいては X=d)

- **cy_dpconfig_X**：「コントロールストア」および静的設定 (Static Configuration) を含むデータパスの設定です。デフォルト値は {128'h0, 32'hFFF0FFFF, 48'h0} です。
- **d0_init_X**：DATA0 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- **d1_init_X**：DATA1 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- **a0_init_X**：ACC0 レジスタの初期化時の値です。デフォルト値は 8'b0 です。
- **a1_init_X**：ACC1 レジスタの初期化時の値です。デフォルト値は 8'b0 です。

このデータパスを、以下の例のようにインスタンス化します：

```
cy_psoc3_dp32 DatapathName (
/* 入力          */ /* .clk(),           // クロック
/* 入力    [02:00] */ /* .cs_addr(),        // コントロールストアの RAM アドレス
/* 入力          */ /* .route_si(),       // ルーティングによるシフト入力
/* 入力          */ /* .route_ci(),       // ルーティングによるキャリー入力
/* 入力          */ /* .f0_load(),        // FIFO 0 を開く
/* 入力          */ /* .f1_load(),        // FIFO 1 を開く
/* 入力          */ /* .d0_load(),        // データレジスタ 0 を開く
/* 入力          */ /* .d1_load(),        // データレジスタ 1 を開く
/* 出力    [03:00:00] */ /* .ce0(),           // アキュムレータ 0 = データレジスタ 0
/* 出力    [03:00]   */ /* .cl0(),           // アキュムレータ 0 < データレジスタ 0
/* 出力    [03:00:00] */ /* .z0(),            // アキュムレータ 0 = 0
/* 出力    [03:00:00] */ /* .ff0(),           // アキュムレータ 0 = FF
/* 出力    [03:00:00] */ /* .ce1(),           // アキュムレータ [0|1] = データレジスタ 1
/* 出力    [03:00]   */ /* .ce1(),           // アキュムレータ [0|1] < データレジスタ 1
/* 出力    [03:00:00] */ /* .z1(),            // アキュムレータ 1 = 0
/* 出力    [03:00:00] */ /* .ff1(),           // アキュムレータ 1 = FF
/* 出力    [03:00:00] */ /* .ov_msb(),        // 操作オーバーフロー
/* 出力    [03:00:00] */ /* .co_msb(),        // キャリー 出力
/* 出力    [03:00:00] */ /* .cmsb(),          // キャリー 出力
/* 出力    [03:00:00] */ /* .so(),            // シフト 出力
/* 出力    [03:00:00] */ /* .f0_bus_stat(),   // FIFO 0 ステータスを uP にする
/* 出力    [03:00:00] */ /* .f0_blk_stat(),   // FIFO 0 ステータスを DP にする
/* 出力    [03:00:00] */ /* .f1_bus_stat(),   // FIFO 1 ステータスを uP にする
/* 出力    [03:00:00] */ /* .f1_blk_stat(),   // FIFO 1 ステータスを DP にする
);
```

4.3.5.3 コントロールレジスタ

コントロールレジスタは、CPU により書き込み可能です。各 8 ビットは全て、PLD 操作およびデータパスの機能性を管理するために、インターコネクト配線から使用可能です。デザイン内に複数のコントロールレジスタを定義することができますが、それぞれ独立して動作します。

デザイン内のコントロールレジスタをインスタンス化するには、Verilog コード内で以下の要素のインスタンスを使用してください。

cy_psoc3_control

この 8 ビットコントロールレジスタには、以下のパラメータが用意されています：

- **cy_force_order:** コンパイラが使用するブーリアン型パラメータで、レジスタ内のビットの順序が重要でない場合、ルータの性能を向上させます。デフォルト値は偽です。一般的には順序が重要であるため、真に設定すべきです。
- **cy_init_value:** チップ設定時に、レジスタに入力される初期値です。

このコントロールレジスタを、以下の例のようにインスタンス化します：

```
cy_psoc3_control #(.cy_init_value (8'b00000000), .cy_force_order(`TRUE))
ControlRegName (
/* 出力 [07:00] */ /* .control()
);
```

4.3.5.4 ステータスレジスタ

CPU は、ステータスレジスタを読み込むことができます。各 8 ビットは全て、PLD 操作およびデータパスのステータスを提供するために、インターコネクト配線から使用可能です。8 ビットステータスレジスタには、もう 1 つの実装があります。この実装では、ステータスに 7 ビットを使用し、第 8 ビットが、他の 7 ビットのマスク出力を OR することにより得られる、割り込みソースとして使用されます。これは、以下の **cy_psoc3_statusi** モジュールに示されています。デザイン内に複数のステータスレジスタを定義することができますが、それぞれ独立して動作します。

デザイン内のステータスレジスタをインスタンス化するには、Verilog コード内で以下のモジュールのインスタンスを使用してください。

cy_psoc3_status

これは、8 ビットステータスレジスタです。この要素には、以下のパラメータが用意されています：これらは、以下のインスタンス例のように、名前付きパラメータとして渡されるべきです：

- **cy_force_order**: コンパイラが使用するブーリアン型パラメータで、レジスタ内のビットの順序が重要でない場合、ルータの性能を向上させます。デフォルト値は偽です。一般的には順序が重要であるため、真に設定すべきです。
- **cy_md_select**: レジスタの各ビットのモード定義です。ビットは、透明もしくは固定のいずれかのモードです。デフォルト値は、各ビットにおいて透明です。

このステータスレジスタを、以下の例のようにインスタンス化します：

```
cy_psoc3_status #(.cy_force_order(`TRUE), .cy_md_select(8'b00000000)) StatusRegName (
    /* 入力  [07:00:00] */ .status(), // ステータスビット
    /* 入力          */ .reset(),    // インターコネクトからリセット
    /* 入力          */ .clock()     // データの登録に用いられるクロック
);
```

cy_psoc3_statusi

このモジュールは、7 ビットステータスレジスタです。これらの 7 ビットに基づいた割り込み出力も付随しています。

statusi レジスタには、ハードウェアイネーブルとソフトウェアイネーブルがあります。割り込みを生成するには、両方のイネーブルが有効である必要があります。ソフトウェアイネーブルは、**Aux Control (副制御)** レジスタにあります。**Aux Control (副制御)** レジスタの複数のビットは異なるコンポーネント用であることもあるので、実装の割り込みを安全にするには、クリティカルリージョン API を用いて行うべきです。以下の例を参照してください。

/* コード例 */

このモジュールには、以下のパラメータが用意されていますが、以下のインスタンス例のように、名前付きパラメータとして渡されるべきです：

- **cy_force_order**: コンパイラが使用するブーリアン型パラメータで、レジスタ内のビットの順序が重要でない場合、ルータの性能を向上させます。デフォルト値は偽です。一般的には順序が重要であるため、真に設定すべきです。
- **cy_md_select**: レジスタの各ビットのモード定義です。ビットは、透明もしくは固定のいずれかのモードです。デフォルト値は、各ビットにおいて透明です。

- **cy_int_mask**: どのビットが、レジスタの 7 ビットの生成に含まれるかを選択する、マスクレジスタです。デフォルト値は 0 であり、7 ビット全てを、割り込み生成において無効化します。

このステータスレジスタを、以下の例のようにインスタンス化します：

```
cy_psoc3_statusi #(.cy_force_order(`TRUE), .cy_md_select(7'b00000000),
.cy_int_mask(7'b1111111)) StatusRegName (
    /* 入力 [06:00:00] */ .status(), // ステータスビット
    /* 入力 */ .reset(), // インターコネクトからリセット
    /* 入力 */ .clock() // データの登録に用いられるクロック
    /* 出力 */ .interrupt() // 割り込み信号 (Int Ctrl への配線)
);
```

4.3.5.5 Count7

単純な 7 ビットカウンタが、同じ機能を実装するためにわざわざ 8 ビットデータパスまたは複数の PLD を消費せずに済むように、提供されています。この要素は、アーキテクチャ内の、貴重な PLD およびデータパス以外のリソースを使用します。カウンタが 3 から 7 ビットの間の場合、このモジュールの使用により PLD およびデータパス リソースを節約できます。このカウンタの使用が便利な 1 つの例は、通信インタフェースのビットカウンタです。4 ビットカウンタが必要な場合、1 つのデータパス全て、または 4 マクロセル PLD 全てを使用することを回避できます。

count7 カウンタには、ハードウェアイネーブルとソフトウェアイネーブルがあります。count7 カウンタにカウントさせるには、両方のイネーブルが有効である必要があります。ソフトウェアイネーブルは、Aux Control (副制御) レジスタにあります。Aux Control (副制御) レジスタの複数のビットは異なるコンポーネント用であることもあるので、実装の割り込みを安全にするには、クリティカルリージョン API を用いて行うべきです。以下の例を参照してください。

```
/* コード例 */
```

cy_psoc3_count7

この要素には、以下のパラメータが用意されていますが、以下のインスタンス例のように、名前付きパラメータとして渡されるべきです：

- **cy_period**: 7 ビット周期の値です。デフォルト値は 7'b1111111 です。
- **cy_route_id**: ルートされてきた信号を、カウンタへの読み込み信号とすることをイネーブルするための、ブーリアン値です。偽ならば、カウンタは、ターミナルカウントにパラメータとして渡された周期の値を、再度読み込みます。真ならば、カウンタは、ターミナルカウントもしくは読み込み信号を読み込みます。デフォルトは偽です。
- **cy_route_en**: ルートされてきた信号を、カウンタへのイネーブルとすることをイネーブルするための、ブーリアン値です。偽ならば、カウンタは、常にイネーブルされます。真ならば、カウンタは、イネーブル出力がハイのときのみイネーブルされます。デフォルトは偽です。

このカウンタを、以下の例のようにインスタンス化します：

```
cy_psoc3_count7 #(.cy_period(7'b1111111), .cy_route_ld(`FALSE), .cy_route_en(`FALSE))
"Counter7Name"
/* 入力          */ .clock(),          // クロック
/* 入力          */ .reset(),          // リセット
/* 入力          */ .load(),           // cy_route_ld = TRUE ならば読み込み信号を使用
/* 入力          */ .enable(),         // cy_route_en = TRUE ならばイネーブル信号を使用
/* 出力 [6:0]    */ .count(),          // カウンタ値を出力
/* 出力          */ .tc()              // ターミナルカウントの出力);
```

4.3.6 固定ブロック

全ての固定ブロックに対し、**cy_registers** のパラメータが、ブロックの指定されたレジスタに対し、明示的に値を設定できるようにするため存在します。リストされた値は、**main()** 関数が呼び出される前に確立する部分をプログラムするために生成される、設定ビットストリームに含まれます。パラメータの値は、以下の形式をとる文字列です：

```
.cy_registers("reg_name=0x##[,reg_name=0x##]");
```

ここで、**reg_name** は **TRM** にリストされている、ブロック内のレジスタ名です (例えば、**DSM** の **CR1** レジスタ)。= の後の値は、レジスタに割り当てられる **16** 進数値です (頭に **0x** をつける必要はありませんが、この値は常に **16** 進数値として扱われます)。リストされたレジスタが、フィッタにおいても設定されている場合、パラメータ内にリストされた値が優先されます。

4.3.7 デザインワイド リソース

いくつかのコンポーネントは、デザインワイド リソースの一部として扱われ、**Verilog** ファイルには含まれません。例としては：

- 割り込みコントローラ
- DMA 要求

4.3.8 ロジックの代わりに Cypress が提供するプリミティブを使用すべき状況

- CPU からアクセスする必要のあるレジスタ、
- 大半の ALU 操作 (特に 4 ビットより幅が広い場合)
- シフト操作 (特に 4 ビットより幅が広い場合)
- マスク操作 (特に 4 ビットより幅が広い場合)
- カウンタ操作 (特に 4 ビットより幅が広い場合)

4.3.9 コンポーネント作成における Warp の特徴

PSoC Creator コンポーネントは、**Verilog** を使用して、システムのデジタルハードウェアを定義することができます。この **Verilog** は、**Verilog 2001** の規格をサポートします。

4.3.9.1 生成文

UDB ベースの **PSoC** デバイスでは、**Verilog** ファイルにおいてパラメータ値の取り扱いにおいて **`ifdef** 文を使用できないため、生成文を使用する機能は、コンポーネントの開発に際し重要になります。例えば、ユーザーが設定したパラメータが **8** ビット幅のデータバスを要求したい場合に、データバスを削除したいとします。これは、条件的生成文を用いて対応可能です。**Warp** は、**Verilog** コードを囲むことができる、条件のおよび **For** ループ生成文をサポートします。

注 生成文内でネットを定義する場合、スコープを理解しておくことが重要です。

注 生成文は、データパス、コントロールレジスタ、およびステータスレジスタから、(生成された *cyfitter.h* ファイル内の)API に渡される慣例的な名前に、レイヤーを付け加えます。"begin : GenerateSlice" 文が存在する生成文を使用した場合、コンポーネントは名前において 1 レベル押し下げられます。たとえば、"DatapathName" という名の、生成文内にないデータパスが、*cyfitter.h* で以下の変数を定義するものとします。

```
#define '$INSTANCE_NAME_DatapathName_u0_A0_REG 0
#define '$INSTANCE_NAME_DatapathName_u0_A1_REG 0
#define '$INSTANCE_NAME_DatapathName_u0_D0_REG 0
#define '$INSTANCE_NAME_DatapathName_u0_D1_REG 0
```

同じデータパスが "GenerateSlice" 生成文の中にあった場合は、同じ変数は *cyfitter.h* 内で以下のよう定義されます：

```
#define '$INSTANCE_NAME_GenerateSlice_DatapathName_u0_A0_REG 0
#define '$INSTANCE_NAME_GenerateSlice_DatapathName_u0_A1_REG 0
#define '$INSTANCE_NAME_GenerateSlice_DatapathName_u0_D0_REG 0
#define '$INSTANCE_NAME_GenerateSlice_DatapathName_u0_D1_REG 0
```

For ループ生成例

```
genvar i;
generate
for (i=0; i < 4; i=i+1) begin : GenerateSlice
    . . . i を使用する任意の Verilog コード
end
endgenerate
```

条件的生成例

```
generate
if(Condition==true) begin : GenerateSlice
    . . . 任意の Verilog コード
end
endgenerate
```

パラメータの使用法

パラメータは、パラメータアサイメントの右辺でのみ使用できます。これは、コンポーネントの開発、特にデータパス設定情報の割り当て時に重要です。例えば、SPI コンポーネントは、ビルドタイムにデータビット数をダイナミックに設定できる機能を要求します。データパスの設定の MSB 値ビットフィールドは、SPI 転送のデータ幅に応じた、異なる値を必要とします。ブロックの転送サイズを定義する、既存のパラメータ NumBits の値によって定まる、MSBVal というパラメータを定義することができます。手順は、データパス設定ツールに提供されています。データパスの設定においては、データパスに渡されるパラメータを手で編集してはなりません。しかし、他のパラメータ定義に対しては、行ってもかまいません。

ローカルパラメータの使用と名前付きパラメータ

パラメータは、PSoC Creator で設定可能なため、幅広いコンポーネントに使用されています。予期しない値で誤って上書きされないように、おパラメータの値を保護することは重要です。このため、Warp の PSoC Creator には、2 つの機能があります。ローカルパラメータのサポートと、名前付きパラメータの受け渡しです。

defparam を使用する場合、エラーがとてもしやすいため、Warp に名前付きパラメータのサポートが追加されました。これは、上記の節の標準モジュールで見られたように、頻繁に使用されます。名

前付きパラメータは、以下の例のように、インスタンス宣言時に、**#(...)** ブロック内で渡されます：

```
cy_psoc3_status #(.cy_force_order(`TRUE)) StatusReg (
    . . . ステータスレジスタのインスタンス化
```

パラメータをローカルパラメータを用いて保護することは可能ですし、できるだけ行うべきです。**SPI** についての例では、**MSBVal** はローカルパラメータとして設定すべきです。パラメータ **NumBits** により設定され、ユーザーがより高いレベルから設定できないからです。この例のコードは、以下のようになります：

```
module SPI(...)
parameter NumBits = 5'd8;
localparam MSBVal = (NumBits == 8 || NumBits == 16) ? 5'd8 :
                    (NumBits == 7 || NumBits == 15) ? 5'd7 :
                    (NumBits == 6 || NumBits == 14) ? 5'd6 :
                    (NumBits == 5 || NumBits == 13) ? 5'd5 :
                    (NumBits == 4 || NumBits == 12) ? 5'd4 :
                    (NumBits == 3 || NumBits == 11) ? 5'd3 :
                    (NumBits == 2 || NumBits == 10) ? 5'd2 :
                    (NumBits == 9) ? 5'd1;
endmodule
```

4.4 ソフトウェアによる実装

ソフトウェアによる実装では、単にハードウェアへの参照が省略されます。回路図および **Verilog** を使用せず、単にソフトウェアファイルを使用します。他の全ての手順は、他のコンポーネントの場合と同じものです。

ソフトウェアのみのコンポーネントの例の **1** つは、いくつかのコンポーネントのコードをリンクするインターフェースです。

5. ハードウェアのシミュレーション



この章では、ModelSim および VCS を用いた UDB ベースの PSoC コンポーネントの開発において、コンテンツをシミュレーションするためのツールおよびメソッドを解説します。

5.1 ModelSim

PSoC Creator 内の Warp ツールは、*vlg* サブディレクトリ中に、Cypress 固有の合成後モデル、デザインの合成シミュレーションモデル、およびテンプレートファイルを作成します。テンプレートファイルは、ターゲット Verilog コンパイラに、正しい順序で正しい Verilog ファイルのセットを渡すのに役立ちます。デザインを異なるシミュレーション環境下でシミュレートする手順が、以下に記載されています。

Verilog ソースデザインの合成は、Warp を使用することで可能です。Cypress は現時点で Verilog シミュレータを提供していませんが、サードパーティーのシミュレータを用いて合成前シミュレーションを行うために必要なシステムファイルをいくつか提供しています。

Warp は、Verilog デザインの合成をおこなうために、ソースファイルを 2 つ使用します。これらのファイルは、`$CYPRESS_DIR/lib/common` にあります。この章では、`$CYPRESS_DIR` は `$INSTALL_DIR/warp/` とします。

第一のファイル *lpm.v* は、ライブラリの LPM モデルが使用するパラメータについて、いくつかの値を定義するのに使用されます。第二のファイル *rtl.v* は、合成エンジンが対象デバイスの内部アーキテクチャにマッピングする際の様々なプリミティブを定義するのに使用されます。これらのモデルを使用して合成を行う際、ユーザーのソースデザインに次の行が含まれる必要があります：

```
include lpm.v
include rtl.v
```

前者は LPM モジュールを使用するため、後者はユーザーのデザインで RTL モジュールを使用するために必要です。合成前のシミュレーションを行うためには、次の行が含まれる必要があります。

合成プロセスで、これらのファイルへのパスは、Warp のコマンドラインにおいて `-i` スイッチで指定されます。(例： `-i$CYPRESS_DIR/lib/common`) Verilog シミュレータにおいては、`+incdir+switch`で行われます。(例： `+incdir+$CYPRESS_DIR/lib/common` or `+incdir+ $CYPRESS_DIR/lib/prim/presynth/vlg`) 後者の例は、Verilog のソースファイルのミラーが `$CYPRESS_DIR/lib/prim/presynth/vlg` にも置いてあり、Verilog の合成前ファイル構造と互換性があるため、有効です。

lpm.v および *rtl.v* ファイル以外に、合成前シミュレーションでは *lpmsim.v* ファイルも使用されます。このファイルは Warp の合成プロセスでは使用されません (合成エンジンが、既に定義され、実装されているモデルを使用するため)。しかし、モジュールのアーキテクチャを示すため、合成前シミュレーションでは必要となります。

これらのモジュール群を使用する ModelSim/Verilog シミュレータの一般的な呼び出しは、以下のようになります：

```
vlog +incdir+$CYPRESS_DIR/lib/common <testbench>.v <testfile>.v
```

必要なモジュールにアクセスするためには、ユーザーのソースコードにおいて、適切な ``include` 文が入っている必要があります。`.lpm.v` ファイルに入っている ``include` 文は、Warp により合成されないデザインに対し `lpm.v` をインクルードするものです。

5.2 VCS

5.2.1 ツール

UDB ベースの PSoC デバイスのデジタルコンテンツは Verilog により定義されます。また、大半のツールは、標準的な Verilog の開発およびシミュレーションツールです。この開発手法は、標準的な Verilog の手順に従いますが、UDB ベースの PSoC デバイスにおけるデータパスと UDB モデルに関して新たな見方をします。

Verilog ファイルは、任意のテキストエディタで作成できます。Verilog ファイル (コンポーネントおよびテストベンチは)、デザインチームが提供した UDB モデルとともにコンパイルされます。これらのモデルは、PSoC Creator とともに ``$CYPRESS_DIR/lib/sim/presynth/vlg'` にあります。

以下は、コンポーネント開発者が必要とするツールの短いリストです。

ツール	説明
VCS	コンパイラです。Verilog ファイルを全てシミュレーションファイルにコンパイルします。
SIMV	VCS により生成された実行可能ファイルです。この実行可能ファイルは、波形ビューアにデータファイルを提供するために実行される必要があります。
DVE	波形ビューアです。
Warp ツール チェーン (Tool Chain) (PSoC Creator)	PSoC デバイス用のコンパイラ (場所、ルート、およびその他のピース) です。
Cygwin	PC 上の Linux エミュレーション環境で、適切な Linux マシンに接続するのに使用します。VCS および Undertow を実行するのに必要な X-Windows 環境を提供します。
データパス (Datapath) 設定ツール	データパスコンポーネントの設定情報を定義するツール。設定 (DAT ファイル) がコンパイル時に設定に読み込まれます。

5.2.2 テストベンチの定義

VCS および undertow をシミュレーションに使用する際、テストベンチに必要なアイテムがあります。UDB モデルに、より多くのシミュレーション機能をもつ構造体がいくつかあります。これらは、UDB ベースの PSoC アーキテクチャ内における UDB を、より正確に表現します。

VCS を使用する際、テストベンチに以下のアイテムを含めてください：

- `.vcd` ファイルから可視なシグナルは、プロジェクト内で指定されている必要があります。下記のコードの一部分は、テストベンチファイルに含まれており、トップレベル (`um_testbench` モジュール) のシグナル全てを `.vcd` ファイル (`foo.vcd`) から可視にします。

```
initial begin
    $dumpfile("foo.vcd");
    $dumpvars(0,um_testbench);
end
```

- 全てのテストベンチ同様、シミュレーションがいつ終了するか指定しない限り、シミュレーションが永久に続く可能性があります。これを防ぐため、テストベンチファイルに以下を含まれます：

```
always begin
    #`RUN_LENGTH $finish;
end
```

より現実的なシミュレーション環境を提供するには、UDB ベースの PSoC アーキテクチャーについて、いくつかの点を理解する必要があります。UDB ベースの PSoC におけるデジタルな世界では、UDB は、メインプロセッサと、UDB 内のレジスタにおいて、レジスタ読み書きを行う、バスインターフェースを通じてやりとりします。シミュレーション環境は、シミュレーション用にこのバスインターフェースを用意しています。また、テストベンチ例および、レジスタ読み込み、書き込み、リセットの 3 タスクが提供されています。シミュレーションにおいて読み書きする前に、全てのレジスタのリセットが必要です。これらのタスクを呼び出すフォーマットは、以下の通りです：

```
bus_reset( `datapathName );
bus_write( `datapathName, REG_NAME, 8'hDATA );
```

これにより、テストベンチのフローを、読み込みおよび書き込みのシーケンスを実装する、ファームウェア同様に書くことが可能になります。以下は、UART コンポーネントに使用されるメソッドをもとにした、テストベンチの例です：

```
initial
begin
    clk100M = 1b0;
    reset = 1b0;  clkcnt = 0;
    force uart.tx_control[0] = 1b0;    //TX を無効化
    force uart.rx_control[0] = 1b0;    //RX を無効化
    #1 reset = 1b1;                    //100MHz クロックを使用し全てをリセット
    #197 reset = 1b0;                  // 全てリセットされたので操作を開始する
    bus_reset( TXShifter );
    bus_reset( TXBitClkGen );
    bus_reset( TXBitCounter );
    bus_reset( RXShifter );
    bus_reset( RXBitClkGen );
    bus_reset( RXBitCounter );

    clkcnt = 0;

    // 初期化：レジスタをテスト用に初期化。これには
    // 出力バッファに最初に送信したバイトを書き込むことも含む

    #(CLK1M_PERIOD*0.1);    // クロックの 1/10 サイクルを待機するのは、
                           // レジスタが、書き込みを完了するまで
                           // 平均 10 サイクルかかるシーケンスにより
                           // 書き込まれると単に仮定しているため
    force uart.tx_control[0] = 1b1;    // UART TX を RUN に設定（制御レジスタに書き込み！）
    force uart.rx_control[0] = 1b1;    // UART RX を RUN に設定（制御レジスタに書き込み！）
    bus_write(TXBitClkGen, UDB_D0_ADDR, 8h04);    //D0 を、ビットクロック生成時に
                                                // コンパレータとして使用

    #(CLK1M_PERIOD*0.1); bus_write(RXBitClkGen, UDB_D0_ADDR, 8h03);    // ビットクロック生成時に
                                                // コンパレータとして使用

    #(CLK1M_PERIOD*0.1); bus_write(TXBitCounter, UDB_D0_ADDR, 8h07);    // 送信されたビットを
                                                // 数えるのに使用

    #(CLK1M_PERIOD*0.1); bus_write(RXBitCounter, UDB_D0_ADDR, 8h08);    // 送信されたビットを
                                                // 数えるのに使用

    //START_TEST1:TX に数バイト送信して、個別のバイトの機能
    // およびデータレディ時の送信メカニズムをテスト
```

```
#(CLK1M_PERIOD*0.1); bus_write(TXShifter, UDB_F0_ADDR, DataToSend[0]); // 送信されたデータを保持
// これは最初に送信されるバイトであるはず

// 最初のバイトの送信にかかる時間として、最大 8 クロック周期の間待ち機
#(CLK1M_PERIOD*20*8);
#100 bus_write(TXShifter, UDB_F0_ADDR, DataToSend[1]); //FIFO_0 が送信されるデータを保持

// バイトの送信にかかる時間として、最大 20 クロック周期の間待ち機
// (各 bit_clock 周期は、Clk1M 周期の 1/8)
#(CLK1M_PERIOD*20*8);
#800 force uart.rx_control[0] = 1b0; //RX を無効化し、この機能をテスト
#100 bus_write(TXShifter, UDB_F0_ADDR, DataToSend[2]); //F0_0 が送信されるデータを保持

// バイトの送信にかかる時間として、最大 8 クロック周期の間待ち機
#(CLK1M_PERIOD*20*8);
#100 bus_write(TXShifter, UDB_F0_ADDR, DataToSend[3]); //F0_0 が送信されるデータを保持

// バイトの送信にかかる時間として、最大 8 クロック周期の間待ち機
#(CLK1M_PERIOD*20*8);
//END_TEST1: 個別バイトの送信テストを終了。

force uart.rx_control[0] = 1b1; //RX を再度有効にし、開始 / 停止機能を確認する。

// 以下の TX が、FIFO にラビッドファイア書き込みされる
// これにより、FIFO の機能を確認し、
// データが存在する限り連続書き込みの
// ロジックが正しいことを確認する

//START_TEST2:
bus_write(TXShifter, UDB_F0_ADDR, DataToSend[4]); //F0 が送信されるデータを保持
#(CLK1M_PERIOD*4*8); //FIFO に新しいデータを書き込むため、4 ビットクロック遅延する
bus_write(TXShifter, UDB_F0_ADDR, DataToSend[5]); //F0 が送信されるデータを保持
#(CLK1M_PERIOD*4*8); //FIFO に新しいデータを書き込むため、4 ビットクロック遅延する
bus_write(TXShifter, UDB_F0_ADDR, DataToSend[6]); //F0 が送信されるデータを保持
#(CLK1M_PERIOD*4*8); //FIFO に新しいデータを書き込むため、4 ビットクロック遅延する
bus_write(TXShifter, UDB_F0_ADDR, DataToSend[7]); //F0 が送信されるデータを保持
#(CLK1M_PERIOD*4*8); //FIFO に新しいデータを書き込むため、4 ビットクロック遅延する
bus_write(TXShifter, UDB_F0_ADDR, DataToSend[8]); //F0 が送信されるデータを保持
#(CLK1M_PERIOD*4*8); //FIFO に新しいデータを書き込むため、4 ビットクロック遅延する
bus_write(TXShifter, UDB_F0_ADDR, DataToSend[9]); //FIFO_0 が送信されるデータを保持
#(CLK1M_PERIOD*4*8); //FIFO に新しいデータを書き込むため、4 ビットクロック遅延する
//END_TEST2:
```

`bus_reset()`、`bus_read()`、および `bus_write()` タスクについて、以下の例から、テスト対象のデータパスをコードに置き換える修正が必要です。以下の図にタスクのフォーマットを示してありますが、下記の点に留意してください。

- `UART_dpTXShifter` を、実際のデータパスに対応する定数に置き換えてください
- `uart.dpTXShifter_` および `uart.dpTXShifter` を、実際のコンポーネント名とデータパスインスタント名に置き換えてください
- `clk100M` を、シミュレーションのシステムクロック名に置き換えてください
- 使用されるデータパスは、3 つのタスクそれぞれにつき、対応するケース文を必要とします。
- `UDB` レジスタの定義は以下の通りです：
// 以下のパラメータは、シミュレーションのバスインターフェースの定義なので、変更しないでください！

```
parameter UDB_A0_ADDR = 12'h000;
```

```
parameter UDB_A1_ADDR      = 12'h010;
parameter UDB_D0_ADDR      = 12'h020;
parameter UDB_D1_ADDR      = 12'h030;
parameter UDB_F0_ADDR      = 12'h040;
parameter UDB_F1_ADDR      = 12'h050;
parameter UDB_ST_ADDR      = 12'h060;
parameter UDB_CT_ADDR      = 12'h070;
parameter UDB_MK_ADDR      = 12'h080;
parameter UDB_AC_ADDR      = 12'h090;
```

- 各データパスのバスインターフェースのクロックは、1 データバスにつき 1 つの代入文を用いた、バスクロックにより駆動されます。

```
assign uart.dpTXShifter.U0.bus_clk = clk100M;

task bus_reset;
    input [3:0] dpaddr;
    begin
        case(dpaddr)
            UART_dpTXShifter:begin
                @(posedge clk100M)
                # 1;
                uart.dpTXShifter.U0.bi_cntl = 8b00000001; // リセット
                @(posedge clk100M)
                # 1;
                uart.dpTXShifter.U0.bi_cntl = 8b00000000; // リセット解除
            end
        endcase
    end
endtask

task bus_read;
    input [3:0] dpaddr;
    input [11:00] addr;
    begin
        case(dpaddr)
            UART_dpTXShifter:begin
                // FIFO が読み取り可能な状態になるまで待機 ....
                wait(uart.dpTXShifter.f0_not_empty == 1)
                // .... ストロブを完全な 1 サイクルの間アサートする ....
                @(posedge clk100M)
                # 1;
                // hsel_wrk_8、hwrite なし
                uart.dpTXShifter.U0.bi_cntl = 8b00000010; // リセット
                uart.dpTXShifter.U0.haddr = addr;
                @(posedge clk100M)
                # 1;
                // .... レディまで待機
                wait(uart.dpTXShifter.U0.hready == 1)
                uart.dpTXShifter.U0.bi_cntl = 8b00000000; // リセット解除
                @(posedge clk100M)
            end
        endcase
    end
endtask

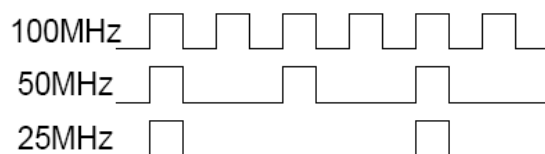
task bus_write;
```

```

input [3:0] dpaddr;
input [11:00] addr;
input [15:00] data;
begin
    case(dpaddr)
        UART_dpTXShifter:begin
            // FIFO が書き込み可能な状態になるまで待機 ....
            // .... ストローブを完全な 1 サイクルの間アサートする ....
            @(posedge clk100M)
            # 1;
            // hsel_wrk_8, hwrite
            uart.dpTXShifter.U0.bi_cntl = 8b00010010;
            uart.dpTXShifter.U0.haddr = addr;
            uart.dpTXShifter.U0.hwdata = data;
            @(posedge clk100M)
            # 1;
            // .... レディまで待機
            wait(uart.dpTXShifter.U0.hready == 1)
            uart.dpTXShifter.U0.bi_cntl = 8b00000000; // リセット解除
            @(posedge clk100M)
        end
    endcase
end
endtask

```

UDB ベースの PSoC デバイス用のもう一つの現実を模したシミュレーションモデルは、UDB が使用可能なクロックに関連があります。現時点でいくつかのクロックを選択できますが、クロックデバイダを使用するとき、重要な点が一つあります。UDB ベースの PSoC アーキテクチャ内のクロックデバイダは、50% デューティ比を提供しません。実際に提供される信号は、選択された分周時間 1 つにつき、ハイタイム 1 つです。たとえば、主発振回路が 100 MHz から、50 MHz および 25 MHz のクロック周波数を取り出すことを考えます：



以下の Verilog コードは、UDB ベースの PSoC におけるデューティ比 (CLK50M_COUNT = (Divider-1)、つまり、50 MHz の場合 1、25 MHz の場合 3) の、50 MHz 出力周波数を生成します。

```

always @(posedge clk100M) begin
    if(clkcnt == CLK50M_COUNT) begin
        clkcnt = 0;
        #(GCLK0_PERIOD*0.75)clk50M_en = 1;
    end
    else begin
        clkcnt = clkcnt+1;
        #(GCLK0_PERIOD*0.75)clk50M_en = 0;
    end
end

assign clk50M = clk100M & clk50M_en;

```

6. API ファイルの追加



この章は、コンポーネントにアプリケーション プログラミング インターフェース (API) およびコードを追加するプロセス、および API 生成、テンプレートの拡張といったその他の情報が記載されています。

6.1 API の概要

API ファイルは、コンポーネントに様々な関数およびパラメータを提供する、コンポーネントインタフェースを定義します。あなたはコンポーネント作成者として、エンドユーザー向けの特定のインスタンス用のコンポーネントにおける、API ファイルテンプレートを作成します。

6.1.1 API 生成

API 生成は、PSoC Creator がインスタンスに依るコードを作成するプロセスです。これには、ハードウェアアドレスを定義するのに用いる、シンボリック定数の生成も含まれます。ハードウェアアドレスは、プレースメント (ビルドプロセスの一環である、フィッタのステップの 1 つ) の結果により決まります。

6.1.2 ファイルの名づけ

API ディレクトリのファイルは全て、コンパイル様にインスタント_名前というファイルに拡張されたテンプレートです。たとえば、カウンターの API ファイル *Counter.c*、*Counter.h*、および *CounterINT.c* があったとします。

foo というトップレベルのカウンターコンポーネントが存在した場合、*foo_Counter.c*、*foo_Counter.h*、および *foo_CounterINT.c* の 3 つのファイルが生成されます。*bar_1_foo* という低レベルのカウンターインスタンスが存在した場合、*bar_1_foo_Counter.c*、*bar_1_foo_Counter.h*、および *bar_1_foo_CounterINT.c* の 3 つのファイルが生成されます。

6.1.3 API テンプレートの拡張

6.1.3.1 パラメータ

API テンプレートコードを作成する場合、連プレートの拡張に、以下のシンタックスのいずれかに従う、任意の組み込み、仮、もしくはローカルパラメータ (ユーザーが定義したパラメータも含む) を使用できます。

`\@< パラメータ >`

`\$< パラメータ >`

シンタックスはいずれも有効で、指定されたパラメータ値を提供するように拡張されます。たとえば：
`void `$_INSTANCE_NAME`_Start(void);`

foo_1 と名付けられたカウンター インスタンスの場合、このコードの一部分は以下のように拡張されます：

`void foo_1_counter_Start(void);`

6.1.3.2 ユーザー定義のタイプ

ユーザー定義のタイプに関するマッピングは、以下の構文により定義されます。

<code>\#DECLARE_ENUM</code> タイプ名	指定されたタイプ名に関するキー名を定義します。
<code>\#DECLARE_ENUM_ALL</code>	コンポーネントが使用する、全てのタイプ名に関するキー名を定義します。

注 タイプ名は、ローカルで (**PARITY**)、もしくはリモートで (**UART__PARITY**) 定義されることもあります。

これらのディレクティブにより、**#define** 宣言文がいくつか生成されます。インスタンスにとってローカルなタイプについては、拡張名はこの形式をとります：

< インスタンスへのパス >_< キー名 >

借用したタイプについては、拡張名はこの形式をとります：

< インスタンスへのパス >_< コンポーネント名 >__< キー名 >

なお、インスタンス名の後ろにはアンダーバーが 1 つ付きますが (他の API 同様)、コンポーネント名が用いられた場合、後ろにはアンダーバーが 2 つ付きますことに注意してください (他のシステム同様)。これにより、ローカルで定義されたキーの値は、レジスター名とコンフリクトを起こさないはずです。

例

コンポーネント :Fox
 タイプ :Color (RED=1, WHITE=2, BLUE=3)
 別名 :Rabbit__Species

コンポーネント :Rabbit
 タイプ :Species (JACK=1, COTTON=2, JACKALOPE=3, WHITE=4)

デザインの中の回路に、**bob** と呼ばれる、インスタンス **Fox** が一つあると仮定します。**bob** の API ファイルの

```
\#declare_enum Color`
\#declare_enum Rabbit_Species`
```

の行は、以下のように拡張されます：

```
#define path_to_bob_RED 1
#define path_to_bob_WHITE 2
#define path_to_bob_BLUE 3
#define path_to_bob_Rabbit__JACK 1
#define path_to_bob_Rabbit__COTTON 2
#define path_to_bob_Rabbit__JACKALOPE 3
#define path_to_bob_Rabbit__WHITE 4
```

bob の API ファイルの

```
\#DECLARE_ENUM_ALL`
```

の行は、以下のように拡張されます：

```
#define path_to_bob_RED 1
#define path_to_bob_WHITE 2
#define path_to_bob_BLUE 3
#define path_to_bob_Rabbit__JACK 1
#define path_to_bob_Rabbit__COTTON 2
#define path_to_bob_Rabbit__JACKALOPE 3
#define path_to_bob_Rabbit__WHITE 4
```

6.1.4 条件的 API 生成

コントロールレジスタ、ステータスレジスタ、**statusi** レジスタ、もしくは割り込みにおいては、デザインからこれらへのマッピングにより、これらのコンポーネントが接続性上使用されていないと判断された場合、マッピングプロセスにより、これらのコンポーネントはマッピングされたデザインから削除されます。コンポーネントに関するコードが不要になったことを API 生成システムに通知するため、

```
#define `$_INSTANCE_NAME`__REMOVED 1
```

のエントリが **cyfitter.h** に生成されます。これにより、コンポーネント **API** がデザインに条件的にコンパイルされます。さらに、除去されたコンポーネントの **API** を使用する全てのコードに対し、この **#define** を利用して、除去されたコンポーネントがアクセスされないように修正を加える必要があります。

6.1.5 Verilog ヒエラルキーの代替

Verilog を用いてデザインを実装する場合、コントロールレジスタ用の様々なフィッタ定数を使用することができます。PSoC Creator provides では、フィッタ定数を使用するために、相対パスを生成する代替メカニズム `[...]` があります。たとえば：

```
foo_1`[uart_x,ctrl]`ADDRESS
```

このコードの一部分は以下のように拡張されます：

```
foo_1_uart_x_ctrl_ADDRESS
```

6.1.6 マージ領域

マージ領域は、エンドユーザーが書いたコードが、コンポーネントおよびソースコードの将来のアップデート後においても保存される領域です。マージ領域はこのシンタックスで定義されます：

```
/* `#START <領域名>` */
/* `#END` */
```

エンドユーザーがこの領域に置いたものの全ては、ファイルの将来のアップデート後も保存されます。ファイルの次のバージョンが同じ名前前のマージ領域を持っていない場合、前のバージョンの領域全てがファイル末にコピーされ、コメントとして保存されます。

6.1.7 API のケース

コンポーネントに対する **API** と回路図を両方、もしくは片方だけ指定することが可能です。なお、**primitive** タイプのコンポーネントのみ、**API** も回路図も存在しないことが可能です。以下のリストに、**API** が存在する、もしくは存在しないケースを列記しています。

- 回路図が存在するものの、**API** が存在しません。

この場合、コンポーネントは、デザインでインスタントが使用された場合に平坦化される必要のある、その他のコンポーネントの階層的な組み合わせです。

ユーザーがコンポーネントプロジェクトの一般的な部分のデザインエントリ文書を編集した場合、回路図にファミリー固有のプリミティブを配置できないことに留意してください。同様に、ユーザーがファミリーのデザインエントリ文書を編集した場合、回路図に配置できるのは、そのファミリーと互換性のあるプリミティブだけです。

- **API** が存在するものの、回路図が存在しません。

この場合、コンポーネントは既存のコンポーネントを再利用せず、**API** を使用します。これは、他のコンポーネントを使用しない、ソフトウェアのみのコンポーネントで起きうる状況です。

- 回路図と **API** が両方提供されています。

これは、デザインに **API** および一つ以上のプリミティブまたは / および他のコンポーネントが含まれている、典型的なコンポーネントです。回路図に、それぞれに **API** を含む他のインスタンスのコンポーネントが含まれる場合、トップレベル デザインのコード生成プロセスは多少複雑です。たとえば、一般的な回路図の中で、インスタンス名 **u1** および **u2** である 2 つの **Count16** コンポーネントから、**Count32** コンポーネントを作成することを考えます。**Count32** コンポーネントのインスタンスがトップレベルデザインに存在する場合、コード生成メカニズムは、**API** 生成時にヒエラルキーを認識する必要があります。

この場合、**Count32** コンポーネントの作者は、**u1** の **API** をテンプレートのコードに ``$INSTANCE_NAME`[u1]`Init()` と使用する必要があります。

Count16 コンポーネントが、たとえば同様なヒエラルキーで **Count8** コンポーネントから構成されている場合、これらのインスタンス名も当然、階層的になります。

- 回路図と API が両方提供されていません。
Primitive タイプのコンポーネント以外では、無効な設定です。

6.2 コンポーネントに API ファイルを追加する

コンポーネントが必要とするだけの API ファイルを、PSoC Creator を用いて追加します：

1. コンポーネントを右クリックし **Add Component Item** を選択します。
[Add Component Item] ダイアログが表示されます。
2. 追加したいコンポーネントのアイコンを選択します。
3. [対象] オプションを選択します。
注：このコンポーネントが、一般的なデバイスなのか、特定のファミリーもしくはデバイスに固有のものかを選択することができます。
4. コンポーネント名を入力します。
5. 新規作成をクリックします。
コンポーネントが、ワークスペース エクスプローラ (Workspace Explorer) に表示されます。
対象オプションによっては、コンポーネントがサブディレクトリに置くことができます。
6. 他の API ファイルを追加するには、この手順を繰り返してください。

6.3 .c ファイルの完成

.c ファイルをダブルクリックして開き、必要ならば修正してください。このファイルは、ビルドの一部として生成される cyfitter.h ファイルを含みます。(79 ページの [コンポーネントの完成](#)を参照。)

6.4 .h ファイルの完成

.h ファイルには一般的に、コンポーネントの関数やパラメータのインターフェースが記載されています。これにより、コンポーネントに対する API 定義を提供します。

ヘッダが生成された場合、名づけに関するいくつかの慣習があります。ワーキングレジスタ (ステータス、制御、周期、マスク等) については、完全なインスタンス名 (ヒエラルキーを含む) に、以下の表に記す固有の文字列をつけたものを使用します：

ワーキングレジスタ	名前 (それぞれ、先頭に追加する文字列：<インスタンス名> _ <生成されるスライス名>)
ステータス	_<StatusRegComponentName>_STATUS_REG
ステータス、制御補助	_<StatusAuxCtrlRegComponentName>_STATUS_AUX_CTRL_REG
制御	_<ControlRegComponentName>_CONTROL_REG
制御、制御補助	_<ControlAuxCtrlRegComponentName>_CONTROL_AUX_CTRL_REG
マスク	_<MaskRegComponentName>_MASK_REG
Period (周期)	_<PeriodRegComponentName>_PERIOD_REG
アキュムレータ 0	_<A0RegComponentName>_A0_REG
アキュムレータ 1	_<A1RegComponentName>_A1_REG
データ 0	_<D0RegComponentName>_D0_REG
データ 1	_<D1RegComponentName>_D1_REG
FIFO 0	_<F0RegComponentName>_F0_REG
FIFO 1	_<F1statusRegComponentName>_F1_REG
データパス、制御補助	_<DPAuxCtlRegComponentName>_DP_AUX_CTL_REG

7. コンポーネントのカスタマイズ



コンポーネントのカスタマイズは、**PSoC Creator** 内でインスタンス化されたコンポーネントのデフォルト挙動に別の挙動を追加および置換するため、カスタムコード (**C#**) を追加する機構です。このコードは、カスタマイズ **EU** と呼ばれることがあり、以下を行うことができます：

- [設定] ダイアログのカスタマイズ
- パラメータ値に基づく、シンボルの形と表示法のカスタマイズ
- パラメータ値に基づく、シンボルターミナル名、カウント、および設定のカスタマイズ
- カスタム **Verilog** コードの生成
- カスタム **C** もしくはアセンブリコードの生成
- クロックシステムとのやりとり (クロックおよび **PWM** コンポーネントのみ)

この章は、コンポーネントのカスタマイズの例、カスタマイズ用ソースについての手順、およびコンポーネントのカスタマイズに関するインターフェースの列記および解説が記載されています。**C#** ソースコードファイル **icyinstancecustomizer.cs** は、カスタマイズ インターフェースに提供されるメソッドを定義する、パラメータ、および戻り値を提供します。**cydextensions** プロジェクトは、カスタマイズに必要なソースコードを含みます。このプロジェクトは **PSoC Creator** に同梱されており、説明文書は **PSoC Creator Customization API Reference Guide** 内にあります。**(customizer_api.chm** ファイル、この コンポーネント作成者ガイドと同じディレクトリにあります)。

7.1 カスタマイズのソース

PSoC Creator において、**C#** で開発されたカスタマイズが使用できます。以下の節にて、**C#** ソースコードの様々な側面が説明されています。

7.1.1 カスタマイズのソースの保護

ソースコードのカスタマイズは、外部アセンブリに依存する可能性があります。このため、カスタマイズをソースコードとして配布することを避けたい場合、外部アセンブリを呼び出し、外部アセンブリのメソッドを実行するだけの、ソースコードのスタブを用意することができます。

7.1.2 開発フロー

プロジェクトの **C#** ソースファイル、リソースファイル、およびアセンブリ レファレンスを用意する必要があります。**PSoC Creator** は、与えられたコードから、ランタイムに自動的にカスタマイズ用 **DLL** をビルドします。自動ビルドは、以下の状況で行われます：

1. それ自身、もしくは依存ファイルにカスタマイズのソースファイルを含むプロジェクトが開かれたものの、有効なカスタマイズ **DLL** が存在しない、もしくは以前作成した **DLL** が失効している場合。
2. それ自身、もしくは依存ファイルにカスタマイズのソースファイルを含むプロジェクトが作成されたものの、有効なカスタマイズ **DLL** が存在しない、もしくは以前作成した **DLL** が失効している場合。

3. コンポーネント作者が、PSoC Creator に新しいカスタマイズ DLL を作成するよう、明示的に要求した場合。

カスタマイズ DLL を「デバッグモード」と「リリースモード」のどちらで生成するかを指定できます。また、コンパイラにコマンドラインオプションを直接指定することが可能です。

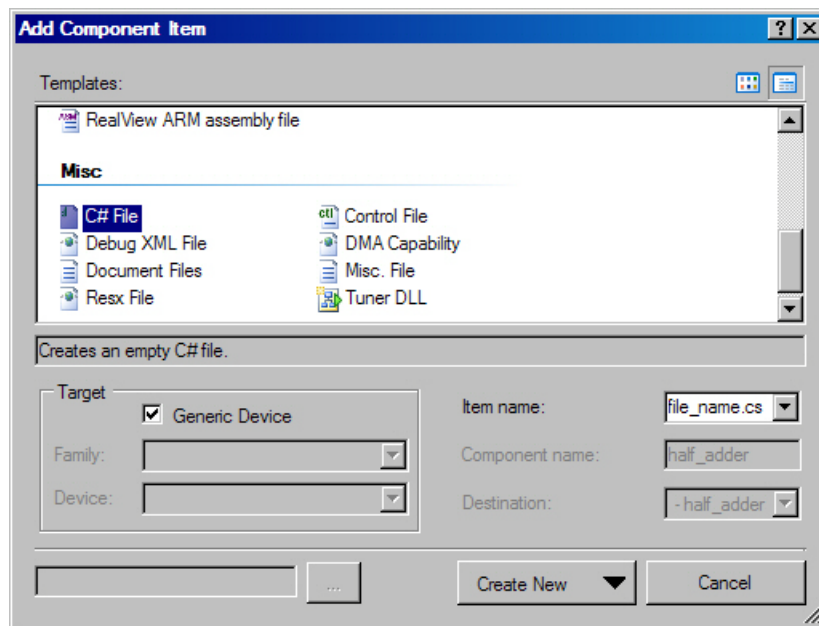
上記をビルド中に、エラーや警告が出た場合、エラーおよび警告は注意リスト (Notice List) ウィンドウに表示されます。DLL はオンザフライで生成されるので、ビルドされたカスタマイズ DLL をソースコントロール下に置く必要はありません。

7.1.3 ソースファイルの追加

ソースファイルをコンポーネントに追加する場合：

1. コンポーネントを右クリックし、**Add Component Item** を選択します。

[Add Component Item] ダイアログが表示されます。



2. [その他 (Misc)] 下の C# アイコンを選択します。
3. 対象下の一般的なデバイス (**Generic Device**) を選択します。
4. アイテム名 [Item Name] には、C# ファイルとして適切な名前を入力してください。
5. 新規作成をクリックします。

コンポーネントアイテムは、Workspace Explorer ツリーにおいて、Custom サブディレクトリ下に表示されます。

テキストエディタ (Text Editor) により .cs ファイルが開かれます。このとき、このファイルを編集することができます。

7.1.4 Custom にサブディレクトリを追加する

サブディレクトリを追加するには、Custom ディレクトリを右クリックし、[追加]>[新しいフォルダ]を選択します。

注：生成されるサブディレクトリは、全て物理的なものです。

7.1.5 リソースファイルの追加

カスタマイザに、.NET リソースファイル (.resx) を追加することができます。これらのファイルは、Custom ディレクトリ、もしくは生成したサブディレクトリに保存することができます。コンポーネント 1 つにつき、複数のリソースファイルを使用することができます。

リソースファイルを追加するには：

1. ディレクトリを右クリックし、**[Add]>[New Folder]** を選択します。
2. 新しいアイテムダイアログで、Resx ファイルアイコンを選択し、ファイル名を入力し、**OK** をクリックしてください。

7.1.6 クラス / カスタマイザの名づけ

必要なカスタマイザインターフェースを実装するのは、1 つのクラスのみです。このクラスは、"CyCustomizer" という名である必要があります。クラスが存在する名前空間は、カスタマイズされるコンポーネント名で終わる必要があります。例えば、"my_comp" という名のコンポーネントについて、"CyCustomizer" クラスが存在できる、有効な名前空間は：

```
Foo.Bar.my_comp  
my_comp  
Some.Company.Name.my_comp
```

[65 ページの使用のガイドライン](#)を参照してください。

7.1.7 アセンブリ レファレンスの指定

PSoC Creator 内部から、カスタマイザの追加外部 DLL へのレファレンスを指定することができます。以下のアセンブリは自動的に含まれます：

- "System.dll"
- "System.Data.dll"
- "System.Windows.Forms.dll"
- "System.Drawing.dll"
- "cydsextensions.dll"

.NET レファレンスおよびその他のユーザーレファレンスを指定するには、アセンブリが存在するディレクトリをブラウズし、指定してください。.NET レファレンスの場合、ここにブラウズします：

```
%windir%\Microsoft.NET\Framework\v2.0.50727
```

アセンブリ レファレンス ダイアログにて、相対パスを指定することも可能です。また、プロジェクトのトップレベルディレクトリ (.cydsn) に対し、アセンブリレファレンスを追加することもできます。相対パスを入力する必要があります。

7.1.8 カスタマイザのキャッシュ

カスタマイザキャッシュは、全てのカスタマイザの DLL が存在するディレクトリです。このディレクトリの所在は、DLL をコンパイルする実行ファイルに依存します。たとえば、カスタマイザ DLL が PSoC Creator 内からコンパイルされる場合、ディレクトリの位置は：

```
Documents and Settings/user_name/Local Settings/Application Data/  
Cypress Semiconductor/PSoC Creator/1.0/customizer_cache/
```

同様に、DLL がビルド (cydsfit) 内からコンパイルされる場合、ディレクトリの位置は：
Documents and Settings/user_name/Local Settings/Application Data/
Cypress Semiconductor/cydsfit/1.0/customizer_cache/

7.2 プリコンパイルされたコンポーネントカスタマイザ

プリコンパイルされたコンポーネントカスタマイザは上級者向けの機能であり、大半のコンポーネント作成者は使用しません。しかし、多くのコンポーネントカスタマイザを含む、とても大きい静的なライブラリ (たとえば PSoC Creator が提供するライブラリ) を扱う場合、この機能によりパフォーマンスを改善することができます。

仕様により、コンポーネントカスタマイザは必要な時に再ビルドされます。コンポーネントカスタマイザが最新のものかどうかを確認するには、構成ファイルをすべて確認する必要があるため、構成ファイルをディスクから読み込み確認するには、時間がかかることがあります。

読み込みと確認を省略し、使用するコンポーネントカスタマイザ アセンブリを定義するために、ライブラリ用にコンポーネントカスタマイザをあらかじめコンパイルしておくことが可能です。この場合、コンポーネントカスタマイザアセンブリがビルドされ、ライブラリのディレクトリ構造にコピーされ、ライブラリに関連づけられます。事前コンパイルされたコンポーネントカスタマイザは、_project_.dll と名付けられ、Workspace Explorer ではプロジェクトのトップレベルから見るができます。

注：事前コンパイルされたコンポーネントカスタマイザが存在する場合、ソースファイルは読み込まれません。ソースファイルが変更された場合、使用されるコンポーネントカスタマイザは默示的に最新のものではありません。さらに、事前コンパイルされたコンポーネントカスタマイザが自動的に開かれるため、プロジェクトの名前の変更、およびカスタマイザのアップデートができなくなります。このため、この機能は、パフォーマンスの都合上必要で、かつコンポーネントカスタマイザが変更されることがなさそうな場合のみ使用してください。

事前コンパイルされたコンポーネントカスタマイザは、常に使用されます。それに伴う混乱を防ぐため、プロジェクトに事前コンパイルされたコンポーネントカスタマイザが存在する場合は、PSoC Creator は新しいコンポーネントカスタマイザのビルドを拒否します。

ライブラリ用に事前コンパイルされたコンポーネントカスタマイザ アセンブリを作成するには、以下を行ってください。

1. 既存の事前コンパイルされたコンポーネントカスタマイザを削除してください (下の手順参照)。
2. PSoC Creator に高度なカスタマイザ (**Advanced Customizer**) ツールバーを追加してください。
3. **カスタマイザのビルドと関連付け** ボタンを押してください。

ライブラリから事前コンパイルされたコンポーネントカスタマイザ アセンブリを削除するには、以下を行ってください。

1. プロジェクトから、_project_.dll という名の、事前コンパイルされたコンポーネントカスタマイザ アセンブリを削除してください。ただし、ディスクからは削除できません (既に開かれているため)。
2. PSoC Creator を閉じ、プロジェクトフォルダ内の事前コンパイルされたコンポーネントカスタマイザ アセンブリを削除してください。

7.3 使用のガイドライン

7.3.1 固有の名前空間の使用

名前空間のリーフ名は、カスタマイザのソースが適用されるコンポーネントを示します。リソース (.resx) ファイルが使用された場合、コンパイルされたリソースを特定するため、同じファイル名を使用します。

コンポーネントの名前が変更された、もしくはインポートされた場合、カスタマイザソース内の名前空間の全てのインスタンスは置き換えられます。

予期しないソースコードの変更を避け、シェアードソース (同じコンポーネントが、同じライブラリの別のコンポーネントで定義された、共用ユーティリティルーチンを使用すること) を可能にするために、ライブラリの中のコンポーネントに共通の固有接頭辞をつけてください。たとえば：

Some.Company.Name.

7.3.2 固有の外部依存関係の使用

.NET は、外部アセンブリ依存関係を、ファイルの位置でなくアセンブリ名に基づいて解決します。このため、同じ名前の二つのアセンブリがあると、予期ないことが起きる可能性があります。外部依存関係による問題を防ぐため、外部にレファレンスされるアセンブリには、全て個別の名前をつけてください。経験的に、.NET はストロングネームを持つアセンブリをよく識別します。

7.3.3 コードの共同使用のための共通コンポーネントの使用

二つのカスタマイザ (A および B とする) がコードを共同使用する場合、共同使用するコードを含む、Common という新しいコンポーネントを作成してください。A もしくは B をインポートした場合、同時に Common もインポートする必要があります。

7.4 カスタマイズ例

以下のディレクトリ内のプロジェクト例を参照してください：

C:\Program Files\Cypress\PSoC Creator\1.0\PSoC Creator\examples\
customizers\SimpleDialogCustomizer.cydsn\

7.5 インターフェース

カスタマイズ用サポートは、2 種類のインターフェースを定義します：

- システムインターフェース (カスタマイズ用 インターフェースが使用する)
- カスタマイズ用 インターフェース (コンポーネント作成者が実装する)

インターフェース名の末尾には、バージョン番号が連結されます。これにより、既存のカスタマイザを破壊せずに、インターフェースをアップグレードできます。

以下の節に、様々なカスタマイゼーション インターフェースが列記され、まとめられています。詳細な説明については、PSoC Creator カスタマイズ用 API レファレンスガイド を参照してください。(customizer_api.chm ファイル、この コンポーネント作成者ガイドと同じディレクトリにあります)

7.5.1 システムインターフェース

この節では、カスタマイズ用インターフェースが使用するシステムインターフェースについて解説します。

PSoC Creator は、PSoC Creator 内部データ構造をアクセスするためのシステムインターフェースを実装しています。インターフェースを安定にし、かつ複雑性を下げるため、インターフェースにはネイティブな C# および .NET タイプが使用されます。これらのインターフェースは、パラメータ値を読み書きし、インスタンスの情報を探し、接続性の情報を調べるメソッドを提供します。

一般的に、システムインターフェースは 2 種類に分かれます: クエリ インターフェースと編集インターフェースです。クエリでは、PSoC Creator へのクエリが可能です。何も修正できません。編集では、カスタマイズ用インターフェースがコンポーネントの様々な箇所を更新および修正することができます。

以下の表は、システムインターフェースに関する簡単なまとめです。

表 7-1. システムインターフェースのまとめ

インターフェース	説明
ICyTerminalQuery_v1	インスタンスにおいて、ターミナルの情報、もしくはターミナルに接続されているアイテムを取得するのに使用します。
ICyTerminalEdit_v1	インスタンスにおいて、ターミナルを編集するのに使用します。これには関連するターミナルの形の変更も含みます。ターミナルの形は、形クエリおよび形編集インターフェースを使用しません。
ICySymbolShapeQuery_v1	インスタンスのマスターシンボルから、形情報をクエリするのに使用します。
ICySymbolShapeEdit_v1	インスタンスのマスターシンボルに基づいて、カスタマイズが基本的な形を修正するのに使用します。
ICyInstQuery_v1	インスタンスの情報について、読み込み専用アクセスを提供します。これには、インスタンスの一般的な情報、パラメータの情報、カスタム データアクセス、および追加のインターフェースを入手するためのフックが含まれます。
ICyInstEdit_v1	インスタンスに値を書き込むためのアクセスを提供します。インターフェースは、新規パラメータ値の書き込み、カスタムデータの設定、およびパラメータエディタの作成などのメソッドを含みます。パラメータの追加および削除はできません。パラメータの値の変更のみが可能です。
ICyInstValidate_v1	インスタンスのパラメータを検証し、エラーを設定するためのアクセスを提供します。パラメータの値の変更はできません。関連するエラーの設定のみが可能です。
ICyExpressMgr_v1	Express スタイルのコンポーネント用の、試作段階のサポートです。以後のリリースで変更される可能性があります。
ICyDesignQuery_v1	デザイン全体にわたる情報のクエリを可能にします。
ICyResourceTypes_v1	いくつかのデバイスがサポートする、特定のリソース名へのアクセスを提供します。これは、コンポーネントを潜在的な複数の実装から選択するために、ICyImplementationSelector_v1 インターフェースと連携して使用することができます。

7.5.2 カスタマイズ用 インターフェース

この節には、コンポーネント作成者により実装される、カスタマイズ用インターフェースについて記載してあります。

ユーザーが提供したカスタマイズのインスタンスが、呼び出されるたびに生成されます。このオブジェクトを使用するクライアントは、呼び出しから次の呼び出しへ情報を繰り越すことができません。データを連続して保持する必要がある場合、インスタンスの隠しパラメータとして保存することが可能です。

以下の表は、カスタマイズ用インターフェースに関する簡単なまとめです。

表 7-2. カスタマイズ用 インターフェースのまとめ

インターフェース	説明
ICyInstValidateHook_v1	このインタフェースの実装により、コードのインスタンスが変更されるたびに、インスタンスのパラメータ値を検証することができます。
ICyParamEditHook_v1	このインタフェースの実装により、インスタンスの "カスタマイズ ..." コンテキストメニューの実装を制御することができます。
ICyCustomData_v1	このインタフェースの実装により、コンポーネントにカスタム情報を保存することができます。カスタムデータは、他の全てのカスタマイズ用インターフェースにおいて、ICyInstEdit_v1 パラメータの ICyInstQuery_v1 を通して利用することができます。
ICyExpressComponent_v1	このインターフェースを使用すべきではありません。唯一の使用目的は、Express スタイルのコンポーネントの、試作段階のサポートです。以後のリリースで変更される可能性があります。
ICyShapeCustomize_v1	コードを通し、回路図上のインスタンスの「見た目」(含まれる形やターミナル)を変更することができます。
ICyVerilogCustomize_v1	コンポーネント インスタンスが呼び出した場合、実装されたこのインタフェースは、カスタム Verilog コードを生成します。
ICyAPICustomize_v1	インスタンスの API の生成の制御を必要とする、インスタンス カスタマイズにより実装されます。
ICyAPICustomize_v2	バージョン 1 からのアップデートです。バージョンが両方存在する場合、バージョン 2 が使用されます。バージョン 2 の CustomizeAPIs メソッドは、バージョン 1 に比べてシグネチャが若干異なります。
ICyToolTipCustomize_v1	ユーザーがコンポーネントインスタンス上をホバーしたときに表示される、ツールチップテキストをカスタマイズします。
ICyClockDataProvider_v1	クロックを生成もしくは操作できるコンポーネントが、クロック情報をシステム内の他のコンポーネントに提供できるようにします。
ICyDesignClient_v1	カスタマイズ開発者は、このインターフェース内のメソッドを用いて、このカスタマイズが、変化する情報に敏感で、データが変更されるたびに呼び出されることを表示することができます。
ICyImplementationSelector_v1	複数の潜在的な実装 (固定関数ブロックとソフト実装、など) が存在するかも知れないコンポーネントに使います。
ICyExprTypeConverter	CyExprEvalFunc に渡された配列引数を、.NET オブジェクトに変換します (bool 型、float 型、int 型、uint 型、short 型、ushort 型、byte 型、sbyte 型、string 型)。
ICyExprEval_v1	C# を用いて新しい式システム関数を作成するのに使用します。
ICyDRCPProvider_v1	このインタフェースの実装により、PSoC Creator 内の注意リストに、インターフェース固有の注意を追加することができます。

7.5.3 カスタマイザのクロック クエリ

PSoC Creator は、コンポーネント作成者が、クロックに関する情報を提供しクエリすることを可能にする、強固なメカニズムを提供します。これは、カスタマイズインターフェース群により提供されます。

7.5.3.1 ICyTerminalQuery_v1

GetClockData という、2 つのメソッドから構成されます。一つはパラメータを 2 個取り (ターミナル名とインデックス)、ターミナルを駆動するクロックの周波数を返します。残りの一つは、「インスタンスパス」、ターミナル名、およびインデックスをパラメータとして取り、インデックスと隠されたクロックの周波数を返します。

7.5.3.2 ICyClockDataProvider_v1

クロックを生成もしくは操作できるコンポーネントが、クロック情報をシステム内の他のコンポーネントに提供できるようにします。提供された情報は自動的に、GetClockData メソッドによりアクセス可能になります。

7.5.4 クロック API のサポート

いくつかのコンポーネント API (I2C など) は、正常に動作するために初期クロック構成にアクセスする必要があります。PSoC Creator は、以下の初期構成を、cyfitter.h 内の #define を通してアクセス可能にします。

```
#define BCLK__BUS_CLK__HZ      value
#define BCLK__BUS_CLK__KZ      value
#define BCLK__BUS_CLK__MHZ     value
```

8. チューニングのサポートの追加



チューニングのサポートは、ユーザーがデザイン中のコンポーネントのチューニングを必要とする、CapSense® などの限られたコンポーネントで使用されている高度な機能です。

チューニングにおいては、ファームウェアコンポーネントとホストアプリケーション間の通信が必要になります。ファームウェアが GUI にスキャンした値を送信し、GUI が更新されたチューニングパラメータをファームウェアに送信します。このプロセスは、デバイスが最良の結果をもたらすまで、繰り返し行われます。

PSoC Creator は、作成者がコンポーネントのチューニングに使用できる、フレキシブルな API を提供することでこの要求項目を満たします。

8.1 チューニング フレームワーク

チューニング フレームワークは、いかなるコンポーネントのチューニングもサポートできる、汎用的なフレームワークです。このフレームワークは、主に二つの API から構成されます。一つは、コンポーネントの作成者により実装される、コンポーネントのチューナーの送りだし地点です。もう一つは、PSoC Creator により実装され、チューナーが PSoC デバイスと通信するのに使う、通信メカニズムを提供します。

デバイスとの通信は、エンドユーザーのデザインの I²C もしくは EZ I²C が行います。

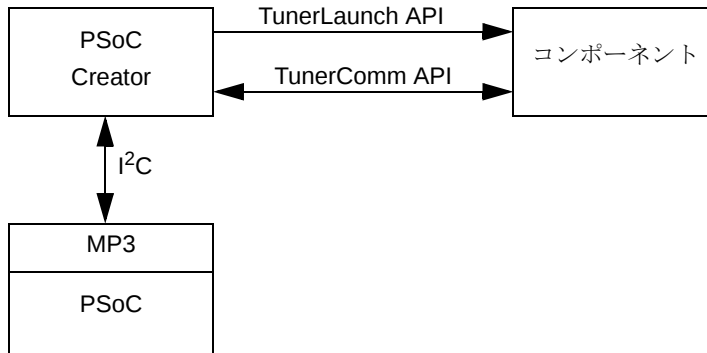
一般的なチューニングでは、GUI に様々な入力からスキャンした値を表示し、ユーザーが新たなチューニング パラメータを入力することを可能にします。パラメータの変更の効果は、変更次第リアルタイムで確認できます。

チューニング アプリケーションは PSoC Creator が実行されている PC で実行され、PSoC デバイスから送られる値を表示します。パラメータ値は、ユーザーの要求に応じて、PC からチップに書き込まれます。

チューニングフレームワークとチューニングするコンポーネントの双方がサポートする通信プロトコルを用いた、双方向通信チャネルを用意するのはユーザーの責任です。PSoC Creator がユーザーのデザインに合わせて通信チャネルを自動的に立ち上げることは不可能です。なぜなら、どのような通信が使えるのか、またユーザーがその他の用途で通信コンポーネントを使っているか、仮定するのは安全ではないからです。

8.2 アーキテクチャ

PSoC Creator チューニング フレームワークを以下の図に示します。PSoC Creator は、TunerLaunch API を用いてコンポーネントのチューニングを開始します。コンポーネントのチューナーは、TuningComm API を用いてデバイスに読み書きします。PSoC Creator は、I²C および MiniProg3 を通じて PSoC デバイスとやりとりします。



8.3 API のチューニング

PSoC Creator は、コンポーネントの作成者がチューニングのサポートを可能にするため、2 個の API を定義しています。LaunchTuner および TunerComm です。

以下の節は、様々なチューニング用インタフェースを列記し、まとめています。詳しい解説に関しては、*PSoC Creator チューニング API レファレンスガイド (PSoC Creator Tuning API Reference Guide)* (*tuner_api.chm* ファイル、この コンポーネント作成者ガイド (Component Author Guide) と同じディレクトリにあります) を参照してください。

8.3.1 LaunchTuner API

LaunchTuner API はコンポーネントに実装され、ユーザーがコンポーネントのインスタンスでチューナーの開始 (Launch Tuner) オプションを選択すると PSoC Creator に呼び出されます。LaunchTuner API を実装するコンポーネントは、全てチューニング可能なコンポーネントとみなされます。PSoC Creator は、チューナーに TunerComm API を実装する通信オブジェクトを受け渡します。チューナーは、このオブジェクトを通してデバイスに読み書きします。

PSoC Creator は、ユーザーがチューナーの実行中に他の作業ができるように (デバッグなど)、チューナーをモードレスウィンドウとして開始します。

8.3.2 Communications API (ICyTunerCommAPI_v1)

通信 API は、チューナー GUI が、使用されている通信メカニズムの詳細を知らずに、ファームウェアと通信できるような関数を定義します。チューナーを開始するアプリケーションは、使用する通信プロトコルに必要な関数を実装する必要があります。

TunerComm API は PSoC Creator に実装され、チューナーが、サポートされたチューニング通信チャンネルを通して PSoC デバイスと通信するのに用いられます。

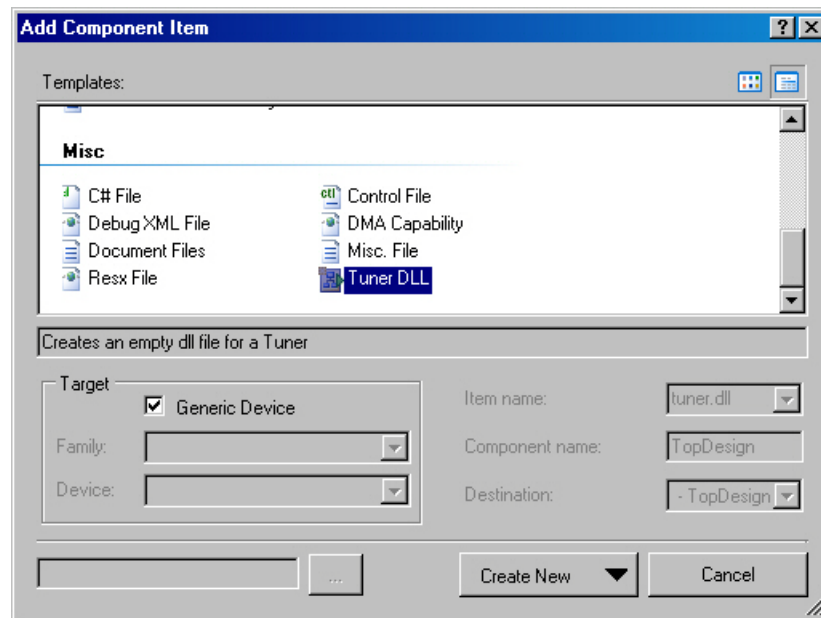
TunerComm API、チューナー GUI とチップのコンポーネントファームウェア間のデータ通信チャンネルを実装します。通信されているデータの中身は、PSoC Creator に不明瞭です。GUI とファームウェアが、やりとりするデータを同じように認識するように設定するのは、コンポーネントチューナーの役割です。

8.4 パラメータの受け渡し

PSoC Creator とチューナー間のパラメータの受け渡しは、名前 / 値ペアの簡単なラッパークラスである、CyTunerParams クラスを用いて行われます。

8.5 コンポーネントチューナー DLL

コンポーネントチューナーのコードは、コンポーネントに、チューナー DLL として含まれる必要があります。コンポーネントにチューナー DLL ファイルを追加するには、[コンポーネントの追加] ダイアログを使用してください。



PSoC Creator は、ソースベースのチューナーをサポートしません。このため、チューナーはツール外でコンパイルされ、DLL として提供されなければなりません。

8.6 通信のセットアップ

この章で既に記されているように、チューナとデバイス間の通信のセットアップは、コンポーネントのエンドユーザーが行います。コンポーネントチューナーは、通信セットアップの知識を一切持っていない。この抽象性をサポートするため、TunerComm API はセットアップメソッドを提供します。コンポーネントチューナーは、通信の設定を開始するため、このメソッドを呼び出してください。

PSoC Creator チューニング API レファレンスガイド (PSoC Creator Tuning API Reference Guide) (tuner_api.chm ファイル、この コンポーネント作成者ガイド (Component Author Guide) と同じディレクトリにあります) を参照してください。

8.7 チューナーの開始

PSoC Creator は、チューナーをコンポーネントインスタンスのために開始する、メニューアイテムを提供します。メニューアイテムは、特定のチューニング可能なコンポーネントインスタンスでのみ意味をもつので、コンポーネントインスタンスのコンテキストメニューにのみ現れます。

8.8 ファームウェアの「交通警察」

ユーザーのデザインには、ユーザーのファームウェアデザインを実装する *main.c* ファイルが含まれます。チューニングをサポートするには、ユーザーのメイン関数に、*I²C* とチューニング可能コンポーネント間のデータのやりとりを容易にする、「交通警察」コードを含む必要があります。以下は、*main.c* の疑似コード例です。

```
main() {

    struct tuning_data {
        // このデータは、PSoC Creator に不明瞭ですが、コンポーネントには既知です
    } tuning_data;

    I2C_Start();                // 通信コンポーネントの開始
    CapSense_Start();           // チューニング可能コンポーネントの開始

    I2C_SetBuffer(tuning_data); // バッファを管理するよう I2C を設定

    for (;;) {

        // 最後にプッシュされたデータが、PC により使用されている場合、
        // 再度バッファを埋める。

        if ( バッファ読み込み完了 ) {
            CapSense_FillBuffer(tuning_data);
        }

        // PC が新しいパラメータ値の書き込みを完了したならば
        // コンポーネントに受け渡し、処理を行う

        if ( バッファ書き込み完了 ) {
            CapSense_ProcessParameters(tuning_data);
        }
    }
}
```

このメイン関数は、チップとホスト PC 間の読み書きを同期する方法を必要とします。これは、読み込みおよび書き込み可能を指示する、バッファの 1 バイトのデータにより実装することが可能です。

8.9 コンポーネントの修正

チューニングをサポートするには、コンポーネントのファームウェアおよび API は、チューニングをサポートするように設計される必要があります。ユーザーは最終デザインにチューニングサポートのオーバーヘッドを含まないことを望むと思われるので、コンポーネント設定ダイアログに、コンポーネントをチューニングモードにするオプションを作ることが要求されます。コンポーネントがチューニングモードに入った後に、ユーザーのデザインが設計されてプログラムされる必要があります。

8.9.1 通信データ

チップがやりとりするデータは、スキャン値とチューニングパラメータ値のフィールドを含む、共通の **C** 構造体を通して通信されます。この構造体の正確な構造は、コンポーネントによりビルド時に決定されます。チューニングモードでビルドされたファームウェアは、この構造を認識しており、パラメータを正しく処理することができます。

8.10 簡単なチューナー

チューナーを実装するクラスは、チューナーの動作を完全に管理します。以下はチューナーのコード例です。(詳細は **API** の説明書を参照してください。)

```
public class MyTuner : CyTunerProviderBase
{
    MyForm m_form;    // カスタム GUI

    public MyTuner() // チューナーの構築子は、型のみを作成する
    {
        m_form = new MyForm();
    }

    public override void LaunchTuner(CyTunerParams params,
        ICyTunerComm comm, CyTunerGUIMode mode)
    {
        m_form.InitParams(params); // 型のセットアップ
        m_form.SetComm(comm);

        // このメソッドが存在する場合、LaunchTuner は Creator に戻ります。
        m_form.ShowDialog();
    }

    // Creator は、このメソッドを呼び出し、
    // 全てのパラメータの新しい値を取得します。
    public override CyTunerParams GetParameters()
    {
        return(m_form.GetParameters());
    }
}
```

このコードでは、チューナーはカスタム GUI (**MyForm**) を作成し、これのショーダイアログを呼び出します。チューナーは独自のスレッドで実行されているため、**PSoC Creator** をハングさせることなくブロッキングコールである **ShowDialog()** を呼び出すことができます。

PSoC Creator は **LaunchTuner** メソッドを呼び出し、復帰するまで待機します。メソッドコールが戻ってきた後、**PSoC Creator** は **GetParameters()** を呼び出して新しいパラメータ値を取得し、そしてコンポーネント インスタンスに保存します。

9. ブートローダー サポートの追加



この章には、コンポーネントにブートローダーのサポートを提供する方法が記載されています。**PSoC Creator** のブートローダーシステムについての詳細な情報については、**PSoC Creator** システム レファレンスガイドを参照してください。

コンポーネントにブートローダーのサポートを提供する場合、大きくみて 2 つの修正が必要です。

- ファームウェア
- カスタマイズ ブートローダー インターフェース

9.1 ファームウェア

コンポーネントがブートローダーをサポートする場合、[セクション 9.1.2](#) にて解説されている、5 つの関数を実装する必要があります。これらの関数は、ブートローダーが通信インターフェースをセットアップし、ホストとパケットをやり取りするのに使用します。

ソースコードおよび関数の実装の詳細については、[57 ページの API ファイルの追加](#)を参照してください。

9.1.1 ガード

デザイン内に複数の、ブートローダーをサポートしており、それぞれ必要なブートローダー関数を実装しているコンポーネントが存在しうるので、これら全てに対し、プリプロセッサが条件的にファイルを読み込むようにガードしてある必要があります。ガード例：

```
#if defined(CYDEV_BOOTLOADER_IO_COMP) &&
    (CYDEV_BOOTLOADER_IO_COMP == CyBtldr_`@INSTANCE_NAME`)

    // ブートローダーのコード

#endif
```

9.1.2 機能

ブートロードをサポートするためには、以下の関数を実装する必要があります：

- CyBtldrCommStart
- CyBtldrCommStop
- CyBtldrCommReset
- CyBtldrCommWrite
- CyBtldrCommRead

関数の定義については、以下の節に記載されています。

9.1.2.1 void CyBtldrCommStart(void)

説明： この関数は、選択された通信コンポーネントを起動します。多くの場合、単なる既存の関数の呼び出しです。`@INSTANCE\_NAME`_Start()

パラメータ： なし

戻り値： なし

副作用： 通信コンポーネントを起動し、PSoC がデータを読み書きできるようにする設定を全て行います。

9.1.2.2 void CyBtldrCommStop(void)

説明： この関数は、選択された通信コンポーネントを終了します。多くの場合、単なる既存の関数の呼び出しです。`@INSTANCE\_NAME`_Stop()

パラメータ： なし

戻り値： なし

副作用： 通信コンポーネントを終了し、通信コンポーネントを無効化するために必要な手順を行います。

9.1.2.3 void CyBtldrCommReset(void)

説明： 選択された通信コンポーネントに、失効データを消去するよう強制します。これは、コマンドが中断されたか壊れた場合に、コンポーネントの状態をクリアし、再起動するために使用します。

パラメータ： なし

戻り値： なし

副作用： 通信コンポーネントがキャッシュしたデータを消去し、コンポーネントが新たな読み込み、もしくは書き込みコマンドが実行できる状態に復帰させます。

9.1.2.4 `cystatus CyBtldrCommWrite(uint8 *data, uint16 size, uint16 *count, uint8 timeOut)`

説明： 提供されたバイト数が、入力データバッファから読み取られ、ホストデバイスに書きこまれるよう要求を出します。書き込みが終了すると、書き込まれたバイト数分カウントが更新されます。`timeOut` パラメータは、関数が実行される時間に上限を設けるために使用します。書き込みが早く完了した場合、出来るだけ早く成功コードを返すべきです。割り当てられた時間内に書き込みが成功しなかった場合、エラーを返すべきです。

パラメータ： `uint8 *data` 書き込むデータを含むバッファへのポインタ
`uint16 size` これから書き込みを行う、データバッファのバイト数
`uint16 *count` 通信コンポーネントが、実際に書き込みを行ったバイト数を書きこむアドレスへのポインタ
`uint8 timeOut` 通信コンポーネントが、通信がタイムアウトしたと判断する前に待つべき時間(10 ミリ秒単位)

戻り値： `CYRET_SUCCESS` 1 バイト以上の書き込みに成功した場合。`CYRET_TIMEOUT` ホストコントローラーが、`timeOut×10` ミリ秒経過しても書き込みに対し反応しなかった場合。

副作用： なし

9.1.2.5 `cystatus CyBtldrCommRead(uint8 *data, uint16 size, uint16 *count, uint8 timeOut)`

説明： 提供されたバイト数が、ホストデバイスから読み込まれ、データバッファに保存されることを要求します。書き込みが終了すると、書き込まれたバイト数分カウントが更新されます。`timeOut` パラメータは、関数が実行される時間に上限を設けるために使用します。読み込みが早く完了した場合、出来るだけ早く成功コードを返すべきです。割り当てられた時間内に読み込みが成功しなかった場合、エラーを返すべきです。

パラメータ： `uint8 *data` ホストコントローラーから読み込んだデータを保存するアドレスへのポインタ
`uint16 size` データバッファに保存するバイト数
`uint16 *count` 通信コンポーネントが、実際に読み込みを行ったバイト数を書きこむアドレスへのポインタ
`uint8 timeOut` 通信コンポーネントが、通信がタイムアウトしたと判断する前に待つべき時間(10 ミリ秒単位)

戻り値： `CYRET_SUCCESS` 1 バイト以上の読み込みに成功した場合。`.CYRET_TIMEOUT` ホストコントローラーが、`timeOut×10` ミリ秒経過しても読み込みに対し反応しなかった場合。

副作用： なし

9.1.3 カスタマイザ ブートローダー インターフェース

ブートローダーは、通信コンポーネントが、**PSoC** からの双方向の通信が可能になっていることを必要とします。この要求項目を満たすコンポーネント用に、カスタマイザにより実装され、**SoC Creator**に要求項目をサポートしていることを通知する、以下のようなインターフェースがあります：

■ ICyBootLoaderSupport

このインターフェースは、**PSoC Creator** カスタマイズ **API** レファレンスガイド (`customizer_api.chm` ファイル: このコンポーネント作成者ガイドと同じディレクトリにあります) の「共通インターフェース」の節に解説されています。ブートローダーが、現在のコンポーネントの設定がブートローダーと互換性が否かを判断するために用いる、手法が一つ記載されています。

```
public interface ICyBootLoaderSupport
{
    CyCustErr IsBootloaderReady(ICyInstQuery_v1 inst);
}
```

コンポーネントがブートローダーに対応している間、デザインに置かれたインターフェースは全て、**Design-Wide Resources System Editor** におけるブートローダー IO コンポーネントのオプションとして表示されます。カスタマイザの中でこの方法を実装する場合、現在のコンポーネントの設定が、双方向通信と互換性があることを確認することのみ必要となります。

カスタマイザについてのより詳細な情報は、[61 ページのコンポーネントのカスタマイズ](#)を参照してください。

10. コンポーネントの完成



この章では、コンポーネントの完成に関する、以下のような様々なステップを取り扱います：

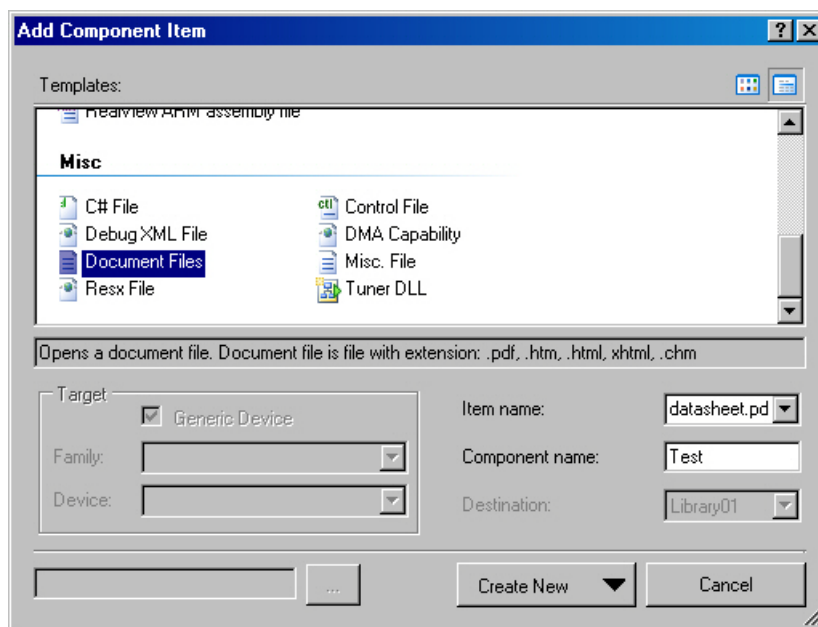
- データシートの追加 / 作成
- XML デバッグファイルの追加
- プロジェクトのビルド

10.1 データシートの追加 / 作成

コンポーネントのデータシートは、MS Word および FrameMaker などの、任意のワープロソフトを用いて作成することができます。その後、そのファイルを PDF、HTML、または XML 形式に変換し、コンポーネントに追加できます。主要要求項目として、コンポーネント カタログから閲覧可能にするためには、PDF ファイルがコンポーネントと同じ名前である必要があります。

1. コンポーネントを右クリックし、**Add Component Item** をクリックします。

[Add Component Item] ダイアログが表示されます。



2. **Document Files** アイコンをテンプレートの **Misc** カテゴリから選択し、**アイテム名**に追加する pdf 名を打ち込んでください。

注 このプロセスは、コンポーネントに追加したいその他のファイル全てに対し適応されます。異なるファイルタイプを追加するには、異なるテンプレートアイコンをクリックしてください。

3. コンポーネントに既存のファイルを追加するには、**Add Existing** を選択してください。PSoC Creator にダミーファイルを作成させるには、**新規作成**を選択してください。

アイテムはワークスペースエクスプローラー (Workspace Explorer) に表示され、PSoC Creator ではタブ付き文書として開かれます。

4. PSoC Creator 外でファイルを作成した場合、プロジェクトのファイルの上にコピーしてください。

10.2 XML デバッグファイルの作成 / 追加

PSoC Creator は、任意のコンポーネントにデバッグツール ウィンドウを生成するオプション機能を提供します。この機能を特定のコンポーネントで有効にするためには、コンポーネントに説明ファイルを XML 形式で追加します。この情報は、任意のメモリーのブロック、および / またはコンポーネントにとって重要なレジスタから構成されます。

1. コンポーネント を右クリックし、**Add Component Item** を選択します。

[Add Component Item] ダイアログが表示されます。

2. **Debug XML File** アイコンをテンプレートの **Misc** カテゴリから選択します。名前はコンポーネントと同じになります。
3. コンポーネントに既存のファイルを追加するには、**Add Existing** を選択してください。PSoC Creator にダミーファイルを作成させるには、**新規作成** を選択してください。

アイテムはワークスペースエクスプローラー (Workspace Explorer) に表示され、PSoC Creator Code Editor ではタブ付き文書として開かれます。

10.2.1 XML フォーマット

以下が XML デバッグファイルの主要な構成要素です。アドレスは、実際のアドレスか、*cydevice.h*、*cydevice_trm.h*、もしくは *cyfitter.h* の **#define** である必要があります XML 文書の中のアイテム名には、空白を含むことができません。

ブロック

関連するレジスタ / メモリの集合を表します。

- 名前 (name) -- ブロック名です。これが全てのサブアイテムに追加されます。(文字列型)
- 説明 (desc) -- ブロックの説明です。(文字列型)
- 可視 (visible) -- ブロック内容が表示されるか否かを指定します。(文字列型、ブーリアン型として評価)

メモリ

コンポーネントが使用する、連続したメモリブロックを表します。

- 名前 (name) -- メモリブロックの名前です。(文字列型)
- 説明 (desc) -- メモリブロックの説明です。(文字列型)
- アドレス (address) -- メモリの開始アドレスです。(文字列型、符号なし整数型として評価)
- サイズ (size) -- ブロックのバイト数です。(文字列型、符号なし整数型として評価)

レジスタ

コンポーネント内部のレジスタを表します。

- 名前 (name) -- レジスタの名前です。(文字列型)
- 説明 (desc) -- レジスタの説明です。(文字列型)
- アドレス (address) -- レジスタのアドレスです。(文字列型、符号なし整数型として評価)

- ビット幅 (bitWidth) -- レジスタのビット数です。(文字列型、符号なし整数型として評価)

フィールド

特定の意味を持つ、レジスタの一部分です。

- 名前 (name) -- フィールドの名前です。(文字列型)
- 説明 (desc) -- フィールドの説明です。(文字列型)
- 開始 (from) -- フィールドの高位ビットです。(文字列型、符号なし整数型として評価)
- 終了 (to) -- フィールドの低位ビットです。(文字列型、符号なし整数型として評価)
- アクセス (access) -- レジスタが使用できるアクセス方法です。(文字列型)

値

レジスタフィールドにとって意味のある、特定の値です。

- 名前 (name) -- 値の名前です。(文字列型)
- 説明 (desc) -- 値の説明です。(文字列型)
- 値 (value) -- 値です。(文字列型、符号なし整数型として評価)

10.2.2 XML ファイルの例

以下は XML ファイルの例です。全ての情報が、ブロック要素に追加される必要があります：

```
<?xml version="1.0" encoding="us-ascii"?>
<deviceData version="1"
  xmlns="http://cypress.com/xsd/cydevicedata"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://cypress.com/xsd/cydevicedata cydevicedata.xsd">

  <block name="\$INSTANCE_NAME" desc="">
    <block name="UDB" desc="" visible="!\$FixedFunction">
      <register name="CTL_REG"
        address="\$INSTANCE_NAME_UDB_sCTRLReg_CTL_REG"
        bitWidth="8" desc="">
        <field name="STAT" from="7" to="3" access="R" desc="">
          <value name="VALID" value="0" desc="" />
          <value name="INVALID" value="1" desc="" />
        </field>
        <field name="ACT" from="2" to="0" access="RW" desc="">
        </field>
      </register>
      <register name="COUNT_REG"
        address="\$INSTANCE_NAME_UDB_sCTRLReg_COUNT_REG"
        bitWidth="16" desc="">
      </register>
      <register name="MASK_REG"
        address="\$INSTANCE_NAME_UDB_sCTRLReg_MASK_REG"
        bitWidth="8" desc="">
      </register>
    </block>
    <block name="HW" desc="" visible="\$FixedFunction">
      <register name="CTL_REG"
        address="\$INSTANCE_NAME_HW_sCTRLReg_CTL_REG"
        bitWidth="16" desc="">
      </register>
    </block>
    <memory name="memory"
      address="\$INSTANCE_NAME_UDB_sCTRLReg_MASK_REG"
      size="256" desc="">
  </deviceData>
```

```
</memory>  
</block>  
</deviceData>
```

10.3 XML 互換性ファイルの追加 / 作成

PSoC Creator は、コンポーネントに互換性の情報を追加する、オプション機能を提供します。この機能により、コンポーネントが互換性のないシリコンのバージョンで使用された場合、DRC 警告、エラー、およびノートを生成することができます。cystate ファイルを含まない場合、DRC は生成されません。

1. コンポーネント を右クリックし、**Add Component Item>** を選択します。

[Add Component Item] ダイアログが表示されます。

2. その他のファイル アイコンを、テンプレートのその他カテゴリから選択してください。
3. 名前には、コンポーネント名に **cystate** 拡張子を追加したものを使用してください。(例えば、**cy_clock_v1_50.cystate**)
4. コンポーネントに既存のファイルを追加するには、**Add Existing** を選択してください。PSoC Creator にダミーファイルを作成させるには、新規作成を選択してください。

アイテムはワークスペースエクスプローラー (Workspace Explorer) に表示され、PSoC Creator Code Editor ではタブ付き文書として開かれます。

10.3.1 XML フォーマット

以下が **cystate** ファイルの主要な構成要素です。XML 文書の中のアイテム名には、空白を含むことができません。

ComponentStateMap

ComponentStateMap 要素は、グローバルデフォルトとして用いられる ComponentState 要素を含む必要がある、ルート要素です。

ComponentState

ComponentState 要素は、MessageType 要素を提供するために使われます。メッセージ文字列を提供することも可能です。

MessageType

MessageType 要素は、DRC のタイプを指定するのに使用されます。これは、次のいずれかの値をとります: None、Note、Warning、もしくは Error。

アーキテクチャ

Architecture 要素は、例えば PSoC3 もしくは PSoC5 といった、アーキテクチャを指定する必要がある場合に使用します。ComponentState 要素を提供し、またゼロ以上の Families 要素を含むことができます。

Families

Families 要素は、例えば 3A もしくは 5A といった、アーキテクチャの系統を指定するときに使用します。ComponentState 要素を提供し、またゼロ以上の SiliconRevision 要素を含むことができます。

SiliconRevision

SiliconRevision 要素は、例えば **ES1**、**ES2**、もしくは **ES3** といった、シリコンの版を指定するときに使用します。使用する場合、**ComponentState** を提供する必要があります。

10.3.2 例：.cystate ファイル

以下は **cystate** ファイルの例です。最低限必要な情報は、デフォルトレベルのルールです。

```
<?xml version="1.0" encoding="utf-8"?>
<ComponentStateMap Version="1">
  <!--Default component state-->
  <ComponentState>
    <MessageType>Warning</MessageType>
    <Message> コンポーネントは、このデバイスでテストされていません。
  </Message>
  </ComponentState>

  <Architecture Name="PSoC3">
    <Families Name="3A">
      <ComponentState>
        <MessageType None /MessageType><>
      </ComponentState>
    </Families>
  </Architecture>

  <Architecture Name="PSoC5">
    <Families Name="5A">
      <ComponentState>
        <MessageType None /MessageType><>
      </ComponentState>
      <SiliconRevision Name="ES1">
        <ComponentState>
          <MessageType Error /MessageType><>
        </ComponentState>
      </SiliconRevision>
    </Families>
  </Architecture>

</ComponentStateMap>
```

10.4 プロジェクトのビルド

コンポーネントに必要なコンポーネントを全て追加し完成した後、デザインをビルドおよびテストするには、コンポーネントをインスタント化する必要があります。

コンポーネントプロジェクトがビルドされる際、**Family** に固有な部分を含めて、全体がビルドされます。ビルド後、ビルドディレクトリには、コンポーネントの作成者がプロジェクトマネージャ経由で指定した、コンポーネント全てが含まれます。

11. ベストプラクティス



この章では、コンポーネントを作成する場合に考慮される、一般的なベストプラクティスが記載されます。

- クロック
- 割り込み
- DMA
- 低電力サポート
- コンポーネントの内包
- Verilog

11.1 クロック

PSoC デバイス内部でクロッキングリソースを使用する場合、いくつかの点に注意してください。他のデジタルシステムと同様に、クロックのアーキテクチャを完璧に理解し、正確な制限やデバイスキャラクタを念頭に置いて設計する必要があります。

PSoC デバイスで利用可能なデジタルリソースの多くは、デバイスの内部クロックを使用してクロックすることができます。ただし、これは常に正しいわけではありません。多くのリソースは、チップ内部で利用可能なクロックソース以外を活用します。PSoC デバイス内で使用するコンポーネントを設計する場合、これら両方の手法を考慮する必要があります。

11.1.1 UDB アーキテクチャーのクロックへの配慮

各 UDB 内部には、少なくとも 2 つのクロックドメインが存在します。最初のドメインはバスクロック (bus_clk) で、CPU が UDB 内部のレジスタにアクセスするために使用されます。このレジスタには、ステータス、コントロール、およびデータパス内部レジスタが含まれています。

クロックドメインの第二のクラスは、"user" クロックでコントロールされます。このクロックは、PSoC 内部クロック構造の外に、本体があることもあります。このクロックは、実装した機能のメインクロックです。

UDB のメタステーブル状態を回避するため、信号がクロックドメインバンダリーを越える場合、同期フリップフロップが必要となることがあります。UDB 内部リソースを達成するには、2 つの方法があります。最初の方法は、シンクロナイザーとして PLD マクロセルを使用する方法です。もう一つの方法は、ステータスレジスタを 4-bit シンクロナイザーとして割り当てる方法です。ステータスレジスタは、8 つの内部フリップフロップを、4 デュアルフリップフロップとして使用し、同期するように構成することができます。しかし、当然のことながら、この方法でステータスレジスタを使用する場合、レジスタは UDB 内部のリソースプールから削除されます。

11.1.2 コンポーネントのクロックへの配慮

設計時に、各機能にどのクロックを使用するかを考慮ときに、クロックの両方のエッジを使用しなくなる場合があります。これは、一部の状況では問題ないものの、非常に危険な場合もあります。例えば、使用するクロックソースが 50% デューティ比でない場合、セットアップおよび待機のタ

イミシングが崩れ、予測不可能または望まれない結果になることもあります。このような問題を防ぐため、全てのデザインは、**posedge (Verilog インスタンス)** クロックのみを使用し、コンポーネントのクロックピンはクロックの正センスに接続される必要があります。イベントがクロックのネガティブエッジで発生する必要がある場合、必要とするクロックの、2 倍の周波数のクロックを用意することで対応できます。

11.1.3 UDB からチップリソースへのクロックへの配慮

UDB アレイに入力または出力される信号は、必ずしもデバイスの I/O ピン用の信号ではありません。むしろ、これらの信号の多くは、固定された機能ブロック、DMA、または割り込みなどの PSoC の他のリソースに接続します。このようなブロックのいくつかには、再同期回路が組み込まれていますが、組み込まれていないものもあります。信号が、UDB アレイとこれらの要素のうち 1 つの間の境界を越える場合、問題が発生する可能性があります。このような信号は、必要なタイミングに適合するように解析されなければなりません。

クロックの、ドメイン境界を横断する信号のタイミング分析は、全ての状況で必要です。さらに、内部クロックに同期する信号に対しても、入力するリージョンにおける位相関係を検証する必要があります。ある量だけシフトしている同期信号は、目的の回路のタイミングの要件を満たしていない可能性があります。

11.1.4 UDB から入出力へのクロックへの配慮

UDB 内のグローバル、**bus_clk**、およびユーザークロック向けのクロック構造は、他の PSoC アーキテクチャで利用することができません。この制限のため、UDB が送受信するには、特別な配慮が必要です。

GPIO レジスタは、**bus_clk** のみにアクセスすることができ、機能が使用する他のあらゆる外部クロックと同期していません。この制限により、**bus_clk** によりクロックされない出力レジスタは、出力へ送信される前に、UDB リソースを使用しなければなりません。これにより、アウトパスへのクロックはとても長くなります。

UDB が使用できる任意の信号は、外部クロックとして利用可能です。ただし、これらの外部クロックは、クロックツリーを通して直接使用できません。クロックツリーに入力される前に、長いパスを通されます。これにより、クロック到着時間が長くなり、セットアップ時間が長くなり、I/O タイミング問題をより複雑にします。

11.1.5 フリップフロップのメタスタビリティ

デジタル回路図用のクロッキングについて考える際に、クロックアーキテクチャを理解することが必要です。最も重要な点は、フリップフロップのメタステーブル状態の定義と解説です。

メタステーブル状態とは、フリップフロップの出力が予測不可能、または不安定である (メタステーブル) 状態な期間です。ある程度の時間が経過した後、最終的に '1' または '0' のいずれかの安定状態に落ち着きます。落ち着くまでの時間は、最終結果を正しく評価するのに出力結果に依存する回路にとっては、長すぎる場合があります。

組み合わせ回路においては、この出力は駆動される回路内のグリッチの原因となります。シーケンス回路内では、このようなグリッチは、レジスタのデータ保管およびステートマシンの決定プロセスに影響を与える、危険な状態となります。このため、メタステーブル状態を回避する必要があります。

メタステーブル状態となるには、以下を含む、いくつかの条件があります：

- クロックによるドメイン境界を横断する信号
- 単一のクロックドメイン内のフリップフロップ間の長い組み合わせパス
- 単一のクロックドメイン内のフリップフロップ間のクロックスキュー

これらの (またはその他の) 状況うち 1 が成立するだけで、メタステーブル状態に陥る可能性があります。一般的に、フリップフロップのセットアップ時間 (Tsu) または待機時間 (Th) に違反した場合、メタステーブル状態に陥る可能性があります。

11.1.6 クロックによるドメイン境界の横断

PSoC 3/5 UDB 内部には、ストレージ要素がいくつかあります。これら全ては、CPU bus_clk を通じてアクセス可能です。回路用のプライマリクロック (ClkIn) である他のクロックソースを通して、アクセス可能です。このクロックソースは、いくつかの場所から選択することができます：

1) CPU bus_clk, 2) グローバルクロック (SrcClk)、または 3) 外部クロック (ExtClk)。

ClkIn が bus_clk である場合、単一のクロックドメインが保証されており、クロックによるドメイン境界の横断の心配は不要です。ただし、過剰スキューが可能、または長い組み合わせパスが存在する状況が起きる可能性があります。

11.1.7 長い組み合わせパスへの配慮

長い組み合わせパスがデザイン内で生成されると、次のストレージ要素用のセットアップ時間に違反する可能性があります。このような状況を回避するため、関連する遅延の合計は、クロックのサイクルタイムより短くなる必要があります。言い換えると、以下の方程式を満たす必要があります：

$$T_{co} + T_{comb} + T_{su} < T_{cycle} \quad \text{式 1}$$

Tco は、フリップフロップをドライブする時間以外 Tcomb は、介入する Tsu は、次のフリップフロップステージのセットアップ時間です。

このようなロングパスがコンポーネント内で完全に含まれる場合、より簡単に扱えます。従って、問題を回避するため、コンポーネントを出る信号は、フリップフロップにより駆動されるべきです。これが不可能か望ましくない場合、クロックからコンポーネントの出力までのタイミングを完全に理解し、伝える必要があります。

11.1.8 同期クロック対非同期クロック

クロックアーキテクチャには、複数のタイプのクロックが含まれています。このようなクロックのいくつかは、マスタシステムクロックに同期である可能性があります。また、いくつかは非同期である可能性があります。この文書において、MPU がコアクロックとして使用するクロック (またはいくつかの派生) を、マスタシステムクロックと定義します。これが、bus_clk と呼ばれるクロックです。

メタステーブル状態を回避するために、MPU へのインタフェースである信号は、bus_clk 信号と同期する必要があります。そのクロックの派生クロックである場合、設計者の課題としては、前述の長い組み合わせパスおよび問題のあるスキューがないことを保証する必要があります。

信号が、MPU へインタフェース接続される必要があるが、bus_clk と非同期のクロックでコントロールされる場合、そのインタフェースに接続される前に同期する必要があります。この同期に便利なプリミティブコンポーネントが、PSoC Creator により提供されています。以下の節で詳細が記載されています。

11.1.9 cy_psoc3_udb_clock_enable プリミティブの活用

クロッキングの複雑性をやわらげ、クロックコンディションを自動的に対処するため、Verilog プリミティブのインスタンス化によりアクセス可能な手段が、PSoC Creator 内にあります。そのプリミティブは、cy_psoc3_udb_clock_enable です。このプリミティブの使用により、同期および非同期のクロックの処理を補佐することができます。

cy_psoc3_udb_clock_enable には、有効 (enable) とクロック (clock_in) 用の入力、UDB コンポーネントを駆動するクロック出力 (clock_out)、およびクロック結果に対する同期動作 (sync_mode) を指定するパラメータがあります。

clock_in 信号は、グローバルクロックまたはローカルクロックのいずれかであり、**bus_clk** と同期でも非同期でも使用することができます。有効信号は、**bus_clk** と同期でも非同期でも使用することができます。これらの 2 つの信号はプリミティブへ接続され、ユーザーは同期モードもしくは非同期モードを選択します。これらが行われると、**PSoC Creator** のフィッタが、**UDB** 要素に求められるクロック動作に必要な実装を判断し、適切な信号マップされた **UDB** 結果に関連付けます。**UDB** へのクロック / 有効をマップする際に使用される規則セットは、以下の表に列記されます：

入力			UDB トランスレーション		
clock_in	en	同期モード	モード	有効化	Clock_out
Global Sync	同期	はい	レベル	en	ClkIn
Global Async	同期	はい	合成中にエラーが発生するため、許可されない		
Local Sync	同期	はい	エッジ	ClkIn & En	SrcClk(ClkIn)
Local Async	同期	はい	エッジ	Sync(ClkIn & En)	bus_clk
Global Sync	非同期	はい	レベル	Sync(En)	ClkIn
Global Async	非同期	はい	合成中にエラーが発生するため、許可されない		
Local Sync	非同期	はい	エッジ	Sync(ClkIn & En)	SrcClk(ClkIn)
Local Async	非同期	はい	エッジ	Sync(ClkIn & En)	bus_clk
Global Sync	同期	いいえ	レベル	en	ClkIn
Global Async	同期	いいえ	レベル	en	ClkIn
Local Sync	同期	いいえ	レベル	en	ExtClk(ClkIn)
Local Async	同期	いいえ	レベル	en	ExtClk(ClkIn)
Global Sync	非同期	いいえ	レベル	Sync(En)	ClkIn
Global Async	非同期	いいえ	レベル	Sync(En)	ClkIn
Local Sync	非同期	いいえ	レベル	Sync(En)	ExtClk(ClkIn)
Local Async	非同期	いいえ	レベル	Sync(En)	ExtClk(ClkIn)

上記の表内で、**Sync()** 関数は、**clock_out** のシグナルの同期にダブルフリップフロップが使用されていることを示しています。このようなダブルレジスタには、**UDB** マクロセルのものを使用します。

cy_psoc3_udb_clock_enable プリミティブの一般的なインスタンス化は、以下のように行われます：

```
cy_psoc3_udb_clock_enable_v1_0 #(.sync_mode ('TRUE')) My_Clock_Enable (
    .clock_in      (my_clock_input),
    .enable        (my_clock_enable),
    .clock_out     (my_clock_out));
```

11.1.10 cy_psoc3_sync コンポーネントの活用

もう 1 つの便利なツールは、**cy_psoc3_sync** プリミティブです。このプリミティブは、信号のクロックへの同期に使用する、ダブルフリップフロップです。このようなダブルフリップフロップは、**UDB** ステータスレジスタへのオプションとして生成されます。プリミティブの幅は 1 ビットをステータスレジスタを 1 つまるごと消費します。ただし、このようなプリミティブを 4 つインスタンス化しても、レジスタを 1 つのみ消費します。

cy_psoc3_sync プリミティブの一般的なインスタンス化は、以下のように行われます：

```
cy_psoc3_sync My_Sync (
    .clock      (my_clock_input),
```

```
.sc_in      (my_raw_signal_in),
.sc_out     (my_synced_signal_out));
```

11.1.11 ルートクロック、グローバルクロック、および外部クロック

UDB で利用可能なクロックには 3 つのタイプがあります。これらは：

- ユーザーが選択可能なクロックドライバからの出力である、8 つのグローバルクロック
- システム内で最も高い周波数のクロックである、bus_clk
- 外部シグナルから入力され、直接クロックされる機能 (SPI Slave など) へのクロックインプットとして使用される、外部クロック

11.1.12 負のクロックエッジの隠れた危険性

Verilog のデザインにおいて、negedge ステートメントを回避することになっています。この方針は、インスタンス化コンポーネントおよび回路図コンポーネントへ接続するクロックにも適用されます。一般的に、ポジティブエッジトリガーとネガティブエッジトリガーを混ぜて使用することで、信号がパスを通過する時間の長さを制限されます。仮に、フリップフロップがポジティブエッジでクロックされ、結果の出力が、何らかのロジックを通して、ネガティブエッジでクロックされる別のフリップフロップに送られる場合、Tcycle は、実質的に半分になります。クロックの両方のエッジでトリガーする必要がある場合、倍のレート of クロックを使用し、トリガにはポジティブエッジを交互に使用すべきです。

11.1.13 一般的なクロックの規則

- コンポーネント内にクロックを作成しないでください。クロック信号は、コンポーネントへの入力としてください。全てのクロックされたロジックは、その単一のクロック上に対しクロックしてください。例えば、ゲートクロックを実装する場合に、クロック信号を AND しないでください。
- グローバルクロック信号を用いて、組み合わせシグナルを作成しないでください。同様に、コンポーネントからグローバルクロックを送信しないでください。UDB 要素エレメントへのクロック出力は定義されていないので、グローバルクロックのタイミングは定義されていません。定義されているタイミング関係をもつクロック信号を生成するには、別のクロックでクロックしたフリップフロップの出力を使用しなければなりません。これにより、入力クロックの 1/2 倍、もしくはこれより低いレート of クロック出力を生成することができます。
- クロックのアウトタイムを最小限にするために、クロックされたエレメント (フリップフロップ) からのコンポーネントから、全ての出力を駆動してください。
- 非同期リセットと事前セットを行わないでください。タイミングの観点によると、これらは、リセット / 事前セット信号から存在するレジスタ出力まで、コンビネーショナルタイミングパスを引き起こします。
- データシグナルがクロックと非同期の場合、同期することが必要となります。
- グローバルクロック信号は、コンポーネントからの出力を用いてはなりません。
- クロックのネガティブエッジを使用しないでください。代わりにレートが倍のクロックを使用してください。

11.2 割り込み

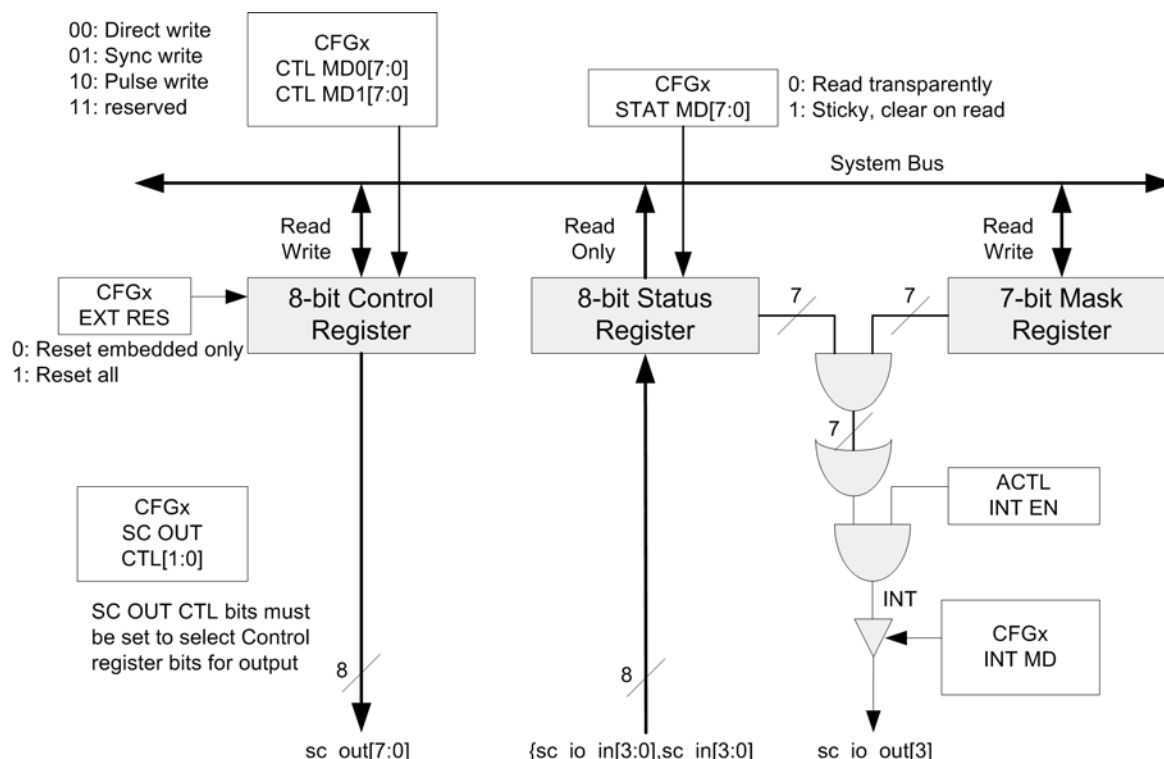
割り込みの主な用途は、CPU ファームウェアまたは DMA チャネルとのやりとりです。UDB アレイ配線内の全てのデータ信号は、割り込みまたは DMA リクエストの生成に使用することができます。ステータスレジスタと FIFO ステータスレジスタは、これらの割り込みを生成する主な手段です。CPU との頻繁な少量のデータのやりとりが必要な場合、割り込みを生成するためにステータスレジスタを使用することが合理的です。

割り込みをコンポーネントに埋め込むか、またはシンボルのターミナルとして露出させるかは、割り込みの処理がアプリケーション固有のものか否かによります。固有の場合、コンポーネントは必要な操作を実行し、必要に応じてアプリケーション処理用に結合バナーを追加します。

割り込みおよび **DMA** リクエストの生成は、デザイン固有のものです。例えば、**CPU** を対象とする割り込みは、固定的に (読み込み時にクリア) デザインされることがあります。しかし、**DMA** 用の同様のリクエストは、適切ではありません。操作記述のデザインは、リクエストをクリアする個別の記述を必要とする上、データを処理するために操作記述を組み込む必要があるためからです。

11.2.1 ステータスレジスタ

ステータスおよびコントロールモジュールのハイレベルビューを以下に示します。このブロックの主な目的は、**CPU** ファームウェアと内部 **UDB** 動作とのやりとりを調整することです。



ステータスレジスタは読み取り専用になっており、これを使って、内部配線から **UDB** 内部の状態を直接システムバスに読み出すことができます。これにより、ファームウェアは **UDB** 処理の状態を監視できます。このようなレジスタの各ビットには、配線マトリクスへプログラム可能な接続があります。

ステータス割り込みの例は、**PLD** またはデータパスブロックが、"compare true" 状況などの条件を生成しており、これがステータスレジスタによってキャプチャされ、次に **CPU** ファームウェアによって読みとられる (およびクリアされる) 場合です。

11.2.2 内部割り込みの生成とマスクレジスタ

ほとんどの機能で、割り込み生成はステータスビットの設定と関係があります。上記の図で示されるとおり、この機能は、ステータスのマスキング (マスクレジスタ) および **OR** リダクションとして、ステータスレジスタのロジックに組み込まれます。ステータスインプットの低位 7 ビットのみ、

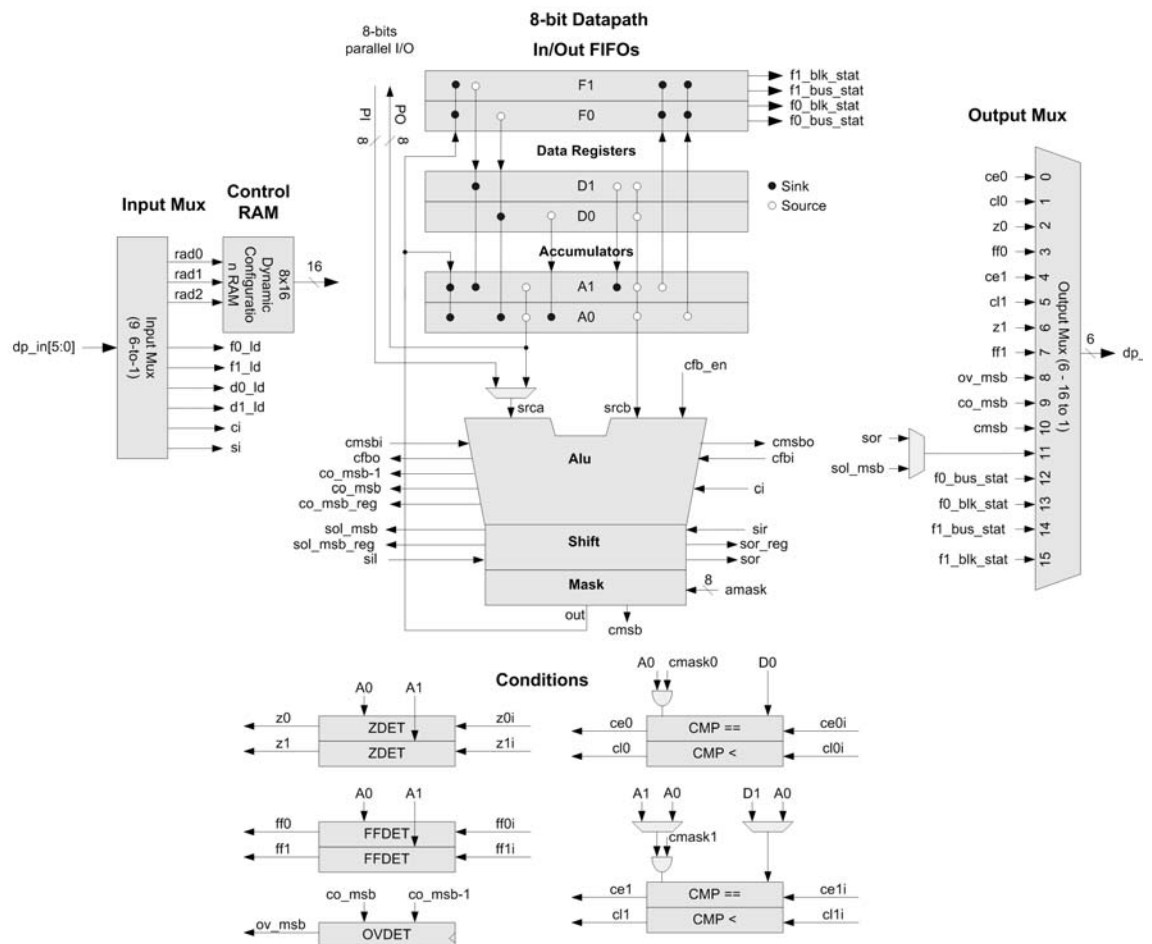
組み込まれた割り込み生成回路とともに使用することができます。デフォルトで、**sc_io** ピンは、出力モードになっており、割り込みは、割り込みコントローラへの接続のため、配線マトリクスヘドライブされることがあります。この構成内で、ステータスレジスタの **MSB** は、割り込みビットの状態として読み込まれます。

ステータスモードレジスタ (CFGx) は、ステータスレジスタの各ビットに対しモード選択を提供します。トランスペアレント読み込みは、ステータスレジスタの CPU 読み込みが、入力信号の配線の状態を返すモードです。固定モードは、読み込み時にクリアする、入力ステータスがサンプルされるモードであり、入力がハイになるとレジスタビットが設定され、入力の次の状態に関係なく設定が維持されます。レジスタビットは、CPU の次の読み込みによりクリアされます。このブロックで選択されたクロックにより、サンプルレートが決定されます。このレートは、ステータス入力信号の生成レート以上のレートであるはずで

11.2.3 スリープ間の保持

マスクレジスタは不揮発性で、スリープインターバルの間状態を維持します。ステータスレジスタは揮発性です。スリープ間に状態を失い、復帰時に **0x00** にリセットされます。

大容量のデータが流れる、または高速バーストが送信または受信される場合、**DMA** 操作が移動に最適な方法です。このような場合、データフローをコントロールする割り込みは、**FIFO** ステータスレジスタを用いて実行されるべきです。以下の図は、ハイレベルからデータパスモジュールを示します。このような **FIFO** は、シーケンサー、割り込みまたは **DMA** リクエストとやりとりするために配線されます。



11.2.4 FIFO ステータス

4 つの FIFO ステータス信号 (`f1_blk_stat`, `f1_bus_stat`, `f0_blk_stat`, および `f0_bus_stat`) は、各 FIFO において 2 つずつあります。これらは入力バッファ (システムバスが FIFO に書き込み、データバスが内部で FIFO へ読み込む)、または出力バッファ (データバスが内部で FIFO に書き込み、システムバスが FIFO から読み込む) として方向別に個別に構成することができます。"bus" ステータスは、デバイスシステムにとって意味があり、DMA リクエストとして DMA コントローラへ配信されるべきです。

"bus" ステータスは、主に DMA インタラクション時 (システムがバイトを読み込みまたは書き込みする時) のシステムバスコントロール用です。

トランザクション用にバッファを実装する場合、データ保存にどのくらいの RAM を使用するかを決定します。バッファのサイズが 4 バイト以下の場合、バッファは FIFO ハードウェアで実装されるべきです。

FIFO の受信および送信で提供されるバッファにより、ユーザーアプリケーションのロジックの処理順序が、バスにおけるデータ通信順となります。また、受信 FIFO は、インタフェースにおける、データのバースト的な挙動を吸収します。この FIFO は、特定バスの周波数から、ユーザーアプリケーションロジックの動作周波数を分離します。実行時に、バッファがオーバーフローもしくはアンダーフローしないか、両方確認してください。

11.2.5 バッファ オーバーフロー

受信 FIFO は、潜在的なオーバーフローを回避するため、特定のポートに対し、スケジュール済みもしくはスケジュール予定のペンディング状態のデータ送信を受け入れるスペースが必要です。理想的には、受信 FIFO 設計とステータスチェックメカニズムは、オーバーランによるデータ損失がないことを保証する必要があります。

オーバーフローの問題を解決するため、設計者は、FIFO ステータスの予測指示を採用せねばなりません。このため、いかなるポート FIFO も、データバス遅延 + ステータスバス遅延 + 最大バースト送信が最大限界値に達していない状態の間は、「問題なし」ステータスを指示します。これは、FIFO 用の最大値が、データバス遅延 + ステータスバス遅延 + 最大バースト と等しく設定される必要があることを示します。実効的に、ポート FIFO がバッファオーバーフローを回避するために、「問題なし」ステータスの間は、追加のスペースが必ず必要です。

11.2.6 バッファアンダーフロー

データ量が最小限界値以下になり、インターフェースの逆端からデータを受信していない場合、受信ポート FIFO はアンダーフローとなり、送信 FIFO がそのポート用へ送信するデータを保有していても空になります。これは、次の更新前に送信側が、事前に許可された割り当てを消費した場合に起きます (例えば受信側のステータスが欠乏または不足している場合)。アンダーフローを防ぐため、ステータス表示の限界最小値を十分高くし、アプリケーションロジックが、FIFO からポートデータを吸い取りすぎる前に、送信側が FIFO スペースが利用可能であるという通知に対応できるようにする必要があります。

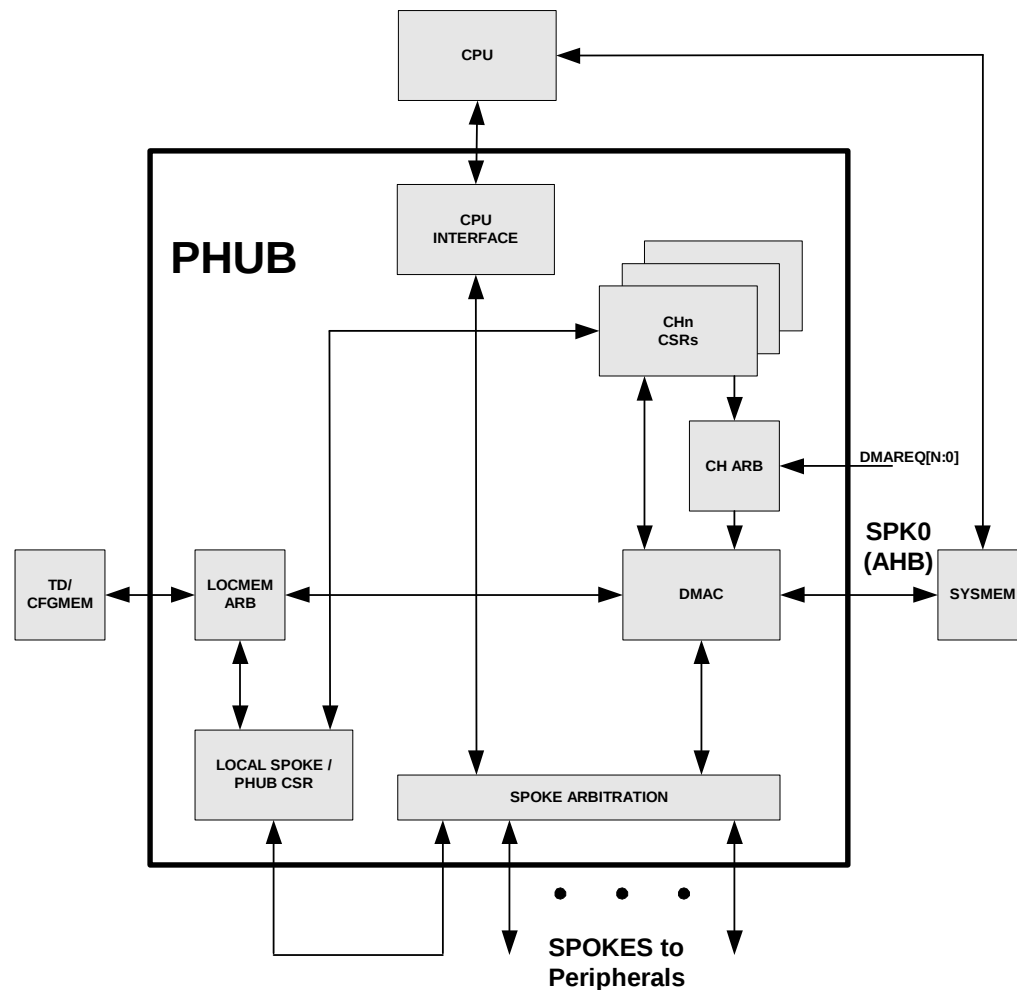
特定ポートに対し、データの受け取りに対し欠乏または不足を通知した時から、データを受け取るまでの時間は、合計バス遅延、すなわちステータス更新待ち遅延 + ステータスバス遅延 + データスケジューラ遅延の合計です。最初の 2 つの数は、トランスミッタに蓄積されるバースト情報を取得する際に必要な合計時間を反映します。最後の 2 つの数は、特定リンクに対し送信 FIFO から受信 FIFO までの、インタフェース上をデータが移動するのに必要な時間を定義します。

バッファアンダーフローは、アプリケーションロジックによるポート FIFO の最大読み込みレートにより異なります。アンダーフローを回避するため、ソフトウェアは各ポート FIFO に対し、十分な最小限界値を設定すべきです。

11.3 DMA

周辺デバイス HUB (PHUB) は、標準的なアドバンスマイクロコントローラバスアーキテクチャ (AMBA) ハイパフォーマンspb (AHB) を利用する、さまざまなオンチップシステム要素を接続する、PSoc 3 および PSoc 5 デバイス内で、プログラム可能及び構成可能なセントラルハブです。PHUB は基本的に、同時に AMBA-lite スタイルマスタリングを可能にする、マルチレイヤー AHB アーキテクチャを利用します。PHUB には、CPU へ負荷をかけずにシステムエレメント間でデータを通信するようプログラムできる、DMA コントローラ (DMAC) が含まれています。PHUB には、PHUB が下流スポークにアクセスする用に、DMAC と CPU 間で仲裁を行うロジックが含まれています。

以下の図は、CPU、PHUB、SYSMEM、TD/CFGMEM、および下流スポーク間の、一般的な接続を示しています。



レジスタマップに関する詳細は、*PSoC® 3, PSoC® 5 Architecture Technical Reference Manual (TRM)* 内に含まれています。ここでの確認ポイントは、DMA コントローラが CPU をオフロードし、これが個別バスマスタであり、DMAC が複数の DMA チャネル間で仲裁することです。DMA ハンドラーと関連 API については、概略が DMA コンポーネントデータシートに記載されています。

このセクションの主要ポイントは、DMA を使用するコンポーネントの構成方法、DMA を利用したデータ通信方法、送信信号を出す方法、および大容量データ、および単一バイトで構成されているものを含む小パケットに最適な送信方法、などを確認することです。構成の複雑性を軽減するため

に、エンドユーザーに **DMA** ウィザードが提供されています。コンポーネント開発に際し、その一部として、**DMA** に関するコンポーネントの機能は **XML** ファイルとして提供することができます。

11.3.1 データ移動用のレジスタ

システムバス接続は、全ての **UDB** に共通であり、**DMA** は、通常の操作と構成のために、**UDB** 内のレジスタと **RAM** にアクセスすることができます。

各データパスモジュールには、**6** つの **8** ビットレジスタがあります。全てのレジスタは、**CPU** および **DMA** で読み書き可能です。

各データパスには、インプットバッファおよびアウトプットバッファとして個別に方向別に構成できる、**2** つの **4** バイト **FIFO** が含まれています。このような **FIFO** は、シーケンサー、割り込みまたは **DMA** リクエストとやりとりするために配線されます。インプットバッファでは、システムバスは **FIFO** へ書き込み、データパス内部から **FIFO** を読み込みます。アウトプットバッファでは、データパス内部が **FIFO** へ書き込み、システムバスが **FIFO** を読み込みます。

少量の通信、特に単一のデータの packets が送信または受信される場合、また別の packets の送信前にデータに対する計算が必要な場合、アキュムレータを使用すべきです。

継続的なデータストリームを必要とする大規模データ通信には、**FIFO** が特に有効です。**FIFO** ステータスロジックと共に、継続的なデータストリームがデータの損失なく維持されます。

型	名前	説明
アキュムレータ	A0, A1	アキュムレータは、 ALU のソース、および ALU 出力の対象になりえます。また、関連するデータレジスタまたは FIFO からロードすることもできます。アキュムレータは、例えば、カウント、 CRC 、シフトなどの、機能の現在の値が含まれています。このようなレジスタは、揮発性であり、スリープ時に値を失い、復帰時に 0x00 にリセットされます。
データ	D0, D1	データレジスタには、 PWM 比較値、タイマー周期、 CRC 多項式などの、既定の関数用の定数データが含まれています。これらのレジスタは揮発性です。これらは、スリープイン間も値を維持します。
FIFO	F0, F1	バッファデータ用にソースと対象を提供する、 2 つの 4 バイト FIFO があります。 FIFO は、両方ともインプットバッファ、両方ともアウトプットバッファ、またはインプットバッファとアウトプットバッファ一つずつとして構成することができます。ステータス信号は、データバスアウトプットとして配線されることもでき、これらレジスタの読み込みと書き込みに関連します。使用例には、 SPI または UART 内のバッファされた TX および RX 、バッファされた PWM の比較、およびバッファされたタイマー周期データが含まれています。 FIFO は揮発性です。スリープ時に内容を失い、復帰時には内容は不明です。 FIFO ステータスロジックは、復帰時にリセットされます。

11.3.2 ステータス用のレジスタ

4 つの **FIFO** ステータス信号があり、**FIFO** 用に 各 **2** つあります : **fifo0_bus_stat**、**fifo0_blk_stat**、**fifo1_bus_stat**、および **fifo1_blk_stat** です。これらの信号の意味は、静的設定により定まる、各 **FIFO** の方向性に依存します。**"bus"** ステータスは、デバイスシステムにとって意味があり、通常は割り込みコントローラもしくは **DMA** コントローラに配線される、またはステータスレジスタを通してポーリングされます。**"blk"** ステータスは、内部 **UDB** オペレーションにとって意味があり、通常は、**PLD** マクロセルから構成されるステートマシンなどの、**UDB** コンポーネントブロックに配線されます。

データバス出力マルチプレクサーを通して配線される **UDB** へのドライブに駆動できる、各 **FIFO** ブロックから生成される **2** つのステータスビットがあります。**"bus"** ステータスは、主に **CPU/DMA** インタラクション時 (システムがバイトを読み込みまたは書き込みする時) のシステムバスコントロール用です。**"block"** ステータスは主に、内部 **UDB** ステートマシンに **FIFO** 状態を提供するためのロー

カルコントロールに用いられます。ステータスの意味は、構成方向 (Fx_INSEL[1:0]) と FIFO レベルビットにより決まります。

FIFO レベルビット (Fx_LVL) は、ワーキングレジスタスペース内の補助的コントロールレジスタに設定されます。以下の表に、オプションを示します。

Fx_INSEL [1:0]	Fx_LVL	信号	ステータス	説明
入力	0	fx_bus_stat	埋まっていない	このステータスは、FIFO 内に少なくとも 1 バイトの空間がある場合、アサートされます。このステータスは、システム割り込み、または FIFO へバイトをさらに書き込むための DMA リクエストをアサートするために使用されます。
入力	1	fx_bus_stat	半分以上空	このステータスは、FIFO 内に少なくとも 2 バイトの空間がある場合、アサートされます。
入力	N/A	fx_blk_stat	空	このステータスは、FIFO 内に残っているバイトがない場合、アサートされます。空でない場合、データパス機能がバイトを消費することがあります。空の場合、コントロールロジックがアイドルとなる、またはアンダーランステータスを生成する可能性があります。
出力	0	fx_bus_stat	空でない	このステータスは、FIFO から最低 1 バイトを読み込むことができる場合、アサートされます。このステータスは、システム割り込み、または FIFO からバイトをさらに読み込むための DMA リクエストをアサートするために使用されます。
出力	1	fx_bus_stat	半分以上埋まっている	このステータスは、FIFO から最低 2 バイトを読み込むことができる場合、アサートされます。
出力	N/A	fx_blk_stat	埋まっている	このステータスは、FIFO が埋まっている場合、アサートされます。埋まっていない場合、データパス機能は、FIFO にバイトを書き込むことができます。埋まっている場合、データパスがアイドルになるか、オーバーランステータスを生成します。

11.3.3 スポーク幅

DMA コントローラは、スポークのデータ幅と同じサイズのデータを、スポーク上で通信します。ただし、AHB 規則により、通信データは全て、通信サイズと同じサイズの、アドレス境界に合致することを要求されます。すなわち、32 ビットトランスファーの ADR[1:0] は、0b00 であり、16 ビットトランスファーの ADR[0] は、0 であることが要求されます。8 ビット通信の場合、アドレスは任意の値をとります。これは、バーストが、スポークのデータ幅と異なるアドレス境界から開始または終了する場合、次の通信で不規則な開始または終了をが起きるからです。

以下の表は、スポークおよびスポークの幅に関連する周辺デバイスを定義します。

PHUB スポーク	周辺デバイス	スポークデータ幅
0	SRAM	32
1	IO、PICU、EMIF	16
2	PHUB ローカル構成、パワー マネジャー、クロック、割り込みコントローラ、SWV、EEPROM、Flash プログラミングインタフェース	32
3	アナログインタフェース、デシメータ	16

PHUB スポーク	周辺デバイス	スポークデータ幅
4	USB、CAN、I2C、タイマー、カウンタ、PWM	16
5	DFB	32
6	UDB グループ 1	16
7	UDB グループ 2	16

このソーススポークおよび対象スポークは、異なるサイズであってもよいです。バーストエンジンは、2つのスポーク間のファンネルメカニズムとして、DMA コントローラ内の FIFO を使用します。

11.3.4 FIFO ダイナミック制御の解説

内部および外部アクセス間の設定は、データバス配線信号を通してダイナミックに切り替え可能です。データバス入力信号 `d0_load` および `d1_load` が、このコントロールに使用されます。

注: ダイナミック FIFO コントロールモードでは、`d0_load` および `d1_load` は、F0/F1 からの D0/D1 レジスタをロードするという、通常の使用法を利用することはできません。

与えられた使用シナリオでは、ダイナミックコントロール (`dx_load`) は、定数を含む。PLD ロジックまたは他の配線信号でコントロールされます。例えば、外部アクセス (`dx_load == 1`) を開始することで、CPU または DMA が、FIFO へ 1 バイト以上のデータを書き込むことができます。内部アクセス (`dx_load == 0`) へのトグル切り替えにより、データバスは、データの操作を実行することができます。次に、外部アクセスへトグル切り替えすることで、CPU または DMA は、計算結果を読みこむことができます。

11.3.5 データバスの条件 / データ生成

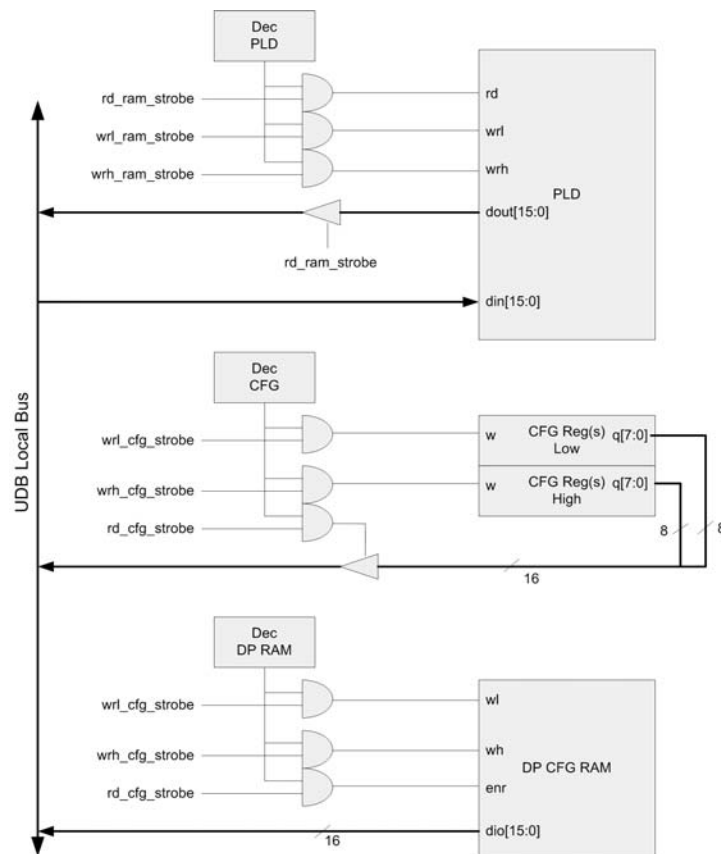
条件は、登録されたアキュムレータ値、ALU 出力、および FIFO ステータスから生成されます。このような条件は、他の UDB ブロック内による使用、割り込み、DMA リクエスト、グローバルおよび I/O ピンへの使用のために、UDB チャンネル配線へドライブされます。可能な 16 の条件を、以下の表に示します：

名前	条件	チェーン?	説明
ce0	等しいか比較	Y	A0 == D0
cl0	未満か比較	Y	A0、< D0
z0	ゼロの検出	Y	A0 == 00h
ff0	1 の検出	Y	A0 = FFh
ce1	等しいか比較	Y	A1 もしくは A0 == D1 もしくは A0 (ダイナミック選択)
cl1	未満か比較	Y	A1 もしくは A0 < D1 もしくは A0 (ダイナミック選択)
z1	ゼロの検出	Y	A1 == 00h
ff1	1 の検出	Y	A1 == FFh
ov_msb	オーバーフロー	N	Carry(msb) ^ Carry(msb-1)
co_msb	キャリーアウト	Y	MSB 定義ビットのキャリーアウト
cmsb	CRC MSB	Y	CRC/PRS 機能の MSB
so	シフトアウト	Y	シフト出力の選択

名前	条件	チェーン?	説明
f0_blk_stat	FIFO0 ブロックステータス	N	定義は FIFO 構成に依存する
f1_blk_stat	FIFO1 ブロックステータス	N	定義は FIFO 構成に依存する
f0_bus_stat	FIFO0 バスステータス	N	定義は FIFO 構成に依存する
f1_bus_stat	FIFO1 バスステータス	N	定義は FIFO 構成に依存する

11.3.6 UDB ローカルバス設定インタフェース

以下の図は、UDB バスインタフェースへの構成状態のインタフェースの構成を示しています。



インタフェースには 3 つの種類があります：PLD、構成ラッチ、および DP 構成 RAM です。全ての構成は、DMA をサポートするために 16 ビットとして、または 16 ビットプロセッサオペレーションとして書き込み可能です。また上位（奇数アドレス）と下位（偶数アドレス）バイトを個別に書き込み可能です。PLD には、RAM 読み書きタイミングを実装する独自の読み込み信号があります。CFG レジスタと DP CFG RAM は、同じ読み込みと書き込みコントロール信号を共有します。

11.3.7 UDB ペア アドレス設定

DMA を使用するデータ通信の方法は、ワーキングおよび設定レジスタの構成方法により異なります。UDB ペア内に、3 つの独立なあります。

- **8 ビットワーキングレジスタ** -- バスサイクルごとに8ビットのデータしかアクセスできないバスマスタは、このアドレススペースを使用して、**UDB ワーキングレジスタ**を読み書きすることができます。ブロックの通常操作中に **CPU ファームウェア**と **DMA** がやりとりする、レジスタがあります。
- **16 ビットワーキングレジスタ** -- **16 ビット**対応のバスマスタは、バスサイクルごとに**16 ビット**アクセスでき、本質的に**16 ビット**以上の機能のデータ通信を容易にします。このアドレススペースは**8 ビットモード**と異なるエリアへマップされるものの、同じ**8 ビット UDB ハードウェアレジスタ 2つ**がアクセスされます。
- **8 または 16 ビット設定レジスタ** -- これらのレジスタは、機能を実行するよう **UDB** を設定します。一旦設定されると、機能の操作中には静的な状態のままです。このようなレジスタは、スリープ中も状態を維持します。

11.3.7.1 ワーキングレジスタのアドレス空間

ワーキングレジスタは、ブロックの通常のオペレーション中にアクセスされ、アキュムレータ、データレジスタ、FIFO、ステータス、コントロールレジスタ、マスクレジスタ、補助コントロールレジスタを含みます。以下の図は、一つの **UDB** のレジスタ マップをです。

8-bit addresses		16-bit addresses
UDB Working Base +		
0xh	A0	0xh
1xh	A1	2xh
2xh	D0	4xh
3xh	D1	6xh
4xh	F0	8xh
5xh	F1	Axh
6xh	ST	Cxh
7xh	CTL/CNT	Exh
8xh	MSK/PER	10xh
9xh	ACTL	12xh
Axh	MC	14xh
Bxh		16xh

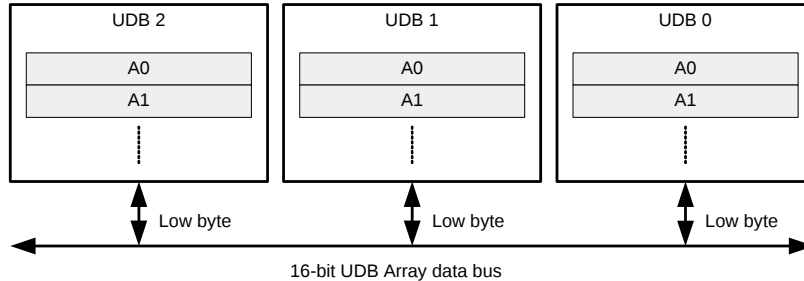
UDB は、**8 または 16 ビットオブジェクト**としてアクセスされ、これら各メソッドに異なるアドレス空間が対応します。

左側に **8 ビットアドレススキーム**が示され、レジスタ番号が上位ニブル内にあり、**UDB** 番号が下位ニブル内にあります。この枠組み内で、**16 UDB** のワーキングレジスタは、**8 ビットアドレス**を用いてアクセスされます。

右側に **16 ビットアドレス**があり、常に偶数 アライメントとして調整されています。**UDB** 番号は、偶数 アドレスアライメントのため、**4 ビット**ではなく**5 ビット**です。上位 **4 ビット**は、レジスタ番号です。合計 **9 アドレスビット**が、**16 ビットデータアクセスモード**内の **16 個の UDB** へのアクセスに必要があります。ワーキングレジスタは、**16 UDB** のバンクとして管理されます。

11.3.7.2 8 ビット ワーキングレジスタへのアクセス

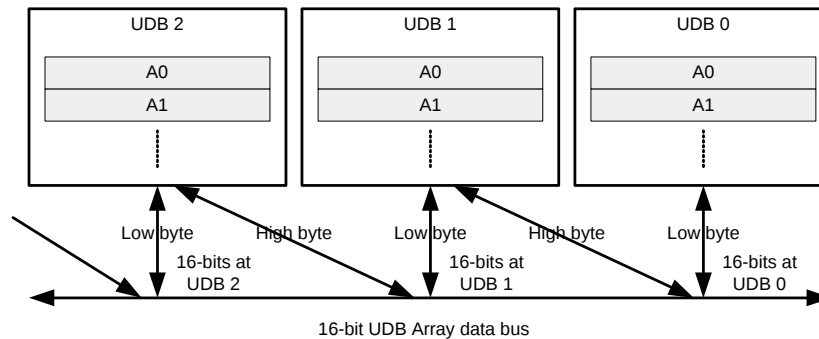
8 ビットレジスタアクセスモード内では、以下の図に示されるように、全ての UDB レジスタが、バイトアライメントされたアドレスにアクセスされます。UDB に書き込まれる全てのデータバイトは、16 ビット UDB バスの低いバイトにアライメントされます。



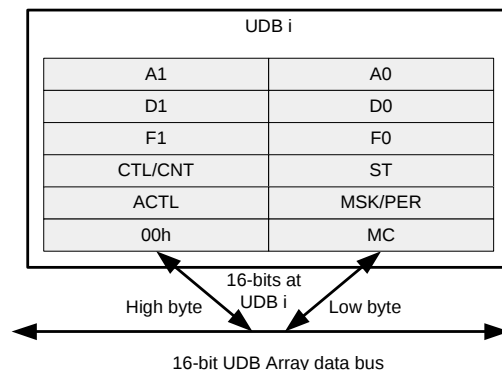
このモードでは1度に1バイトのみアクセスでき、PHUBは、有効な奇数（上位）または偶数（下位）バイトを、アライメントしプロセッサまたはDMAへ返します。

11.3.7.3 16 ビットワーキングレジスタのアドレス空間

16 ビットアドレススペースは、効率的な DMA アクセス（16 ビットデータ幅）用にデザインされています。16 レジスタアクセスには2つのモードがあります：「デフォルト」モードと「連結」モードです。以下の図で示されるように、デフォルトモードでは、UDB 'i' の特定のレジスタの低位バイト、および UDB 'i+1' の同じレジスタの上位バイトにアクセスします。これにより、16 ビット関数として構成される隣接 UDB（アドレス順序）の、16 ビットデータを効率的な処理ができます。



以下の図は連結モードを示します。単一 UDB のレジスタが連結され、16 ビットレジスタを構成します。このモード内では、16 ビット UDB アレイデータバスは、UDB 内のレジスタのペアへにアクセスできます。例として、A0 へのアクセスは、低位バイトに A0、高位バイトとして A1 を返します。



11.3.7.4 16 ビットワーキングレジスタのアドレス制限

16 ビットワーキングレジスタのアドレス空間において、DMA の使用に制限があります。このアドレス空間は、16 ビット UDB 機能への DMA および CPU アクセス用に最適化されています。機能が 16 ビットより大きい場合、非効率的になります。これは、アドレス設定が、以下の表に示されるようにオーバーラップしているからです：

アドレス	上位バイトの行き先	下位バイトの行き先
0	UDB1	UDB0
2	UDB2	UDB1
4	UDB3	UDB2

DMA が 16 ビットをアドレス 0 へ送信すると、下位および上位バイトはそれぞれ UDB0 および UDB 1 へ書き込まれます。では、アドレス 2 への、DMA による次の 16 ビットの送信では、送信の下位バイトが UDB の値を上書きします。このアドレス設定をサポートしつつもメモリバッファに冗長データを送信することを避けるためには、16 ビット以上の機能にはで 8 ビットワーキング空間内の 8 ビット DMA の使用が推奨されます。

11.3.8 DMA バスの使用

DMA コントローラには、パイプラインされることで並列に実行することができるため、デュアルコンテキストがあります。一般的に、スポークバスは、移行データに属する AHB バスサイクルを実質的に 100% の利用することができます。

チャンネル処理のオーバーヘッドは、一般的に、データバーストのバックグラウンド内に隠されます。チャンネルの仲裁、フェッチング、更新は、別のチャンネルのデータバーストのバックグラウンド内においても発生します。さらに、1 チャンネルのデータバーストは、スポーク衝突、ソースの衝突 (SRC)、DMA コントローラの対象 (DST) エンジンがない場合、別のチャンネルのデータバーストとオーバーラップすることがあります。

11.3.9 DMA チャンネルのバーストタイム

チャンネルバーストタイムは、SRC スポークへの最初のリクエストから、DST スポークの最終レディまでのクロック数として定義されます。理想的なバーストでは、初期コントロールサイクルの次に、データサイクルのバーストが起こります（次のコントロールサイクルが、並列にパイプラインされている状況）。このようにスポークの理想的なバーストでは、データ N 個を $N+1$ クロックサイクルで送信します (N = バーストの長さ / 周辺デバイスの幅)。

この数字に影響を与える、複数の要因があります：

- DMA コントローラコンテキストパイプの別のチャンネルコンテキストの存在、およびチャンネル内に残るバースト条件です。
- チャンネルのバースト条件：
 - スポークの使用において、CPU に対する競合
 - SRC/DST 周辺デバイスの準備
 - SRC/DST 周辺デバイスの幅
 - バーストの長さ
 - 不規則な開始と終了
 - インTRASポーク対インタースポーク DMA

イントラスポーク DMA では、全体の SRC バーストが、データが同じスポークに再度書き込まれる前に、DMA コントローラ FIFO 内に最初にバッファされることが必要です。この場合、長さが $N+1$ のバーストが 2 つ起こるため、理想的なイントラスポークバーストの長さは $2N+2$ です。

インタースポーク DMA では、SRC と DST バーストは、オーバーラップすることができます。データは、SRC スポークから読み込まれ、DMA コントローラ FIFO へ書き込まれる間に、DST エンジンに、DST スポークへ利用可能な FIFO データを書き込むことができます。このオーバーラップにより、インタースポーク DMA はより効率的です。最終的に、1 つのスポークから別のスポークへ 1 つのデータを移動するには、3 つのオーバーヘッドサイクルがあります。各スポークの初期コントロールサイクルプラス " 重複 " データサイクル (データの各部分を移動するために、各スポークにおいて 1 データサイクルが必要です)。このように、理想的なインタースポーク DMA バーストの一般的な長さは、データを N 個移動する場合、 $N+3$ です。

以下の表は、理想的なバースト例のサイクルタイムを示しています。

データのやりとり (スポークのサイズ)	イントラスポーク DMA バーストフェーズ (クロックサイクル)	インタースポーク DMA バーストフェーズ (クロックサイクル)
1	4	4
2	6	5
3	8	6
N	$2N+2$	$N+3$

11.3.10 コンポーネントの DMA 機能

PSoC Creator DMA ウィザードと互換性を持つコンポーネントでは、DMA 機能を含む XML ファイルが存在する必要があります。XML フォーマットには、インスタンス固有の情報を反映しなければなりません。これは、インスタンス用のパラメータセットに基づく設定を持てる、静的 XML を用いることで可能です。

11.4 低電力サポート

一般的に、低電力をサポートするコンポーネントは、低電力モードからの復帰時に失われる、揮発性レジスタおよびユーザーパラメータの内容を保持する API を提供します。レジスタおよびパラメータを保存する API は、低電力モードに入る前に呼び出されます。次に、低電力モードから復帰した後、レジスタおよびパラメータを戻すために API が呼び出されます。保存するレジスタは、デザイン内で使用されるレジスタの機能です。TRM が、どのレジスタが揮発性かを指定します。低電力モードに入るする場合、揮発性レジスタのみを保存する必要があります。

11.4.1 機能的な要求項目

コンポーネントに基づいた揮発性レジスタ値を保持するための、静的データ構成の提供。低電力モードの関数は、必要な場合のみ実装されます。これにより、全てのコンポーネントに対し一貫的なインタフェースを提供します。初期化、保存 / 復旧、スリープ / 復帰に必要なデータ構成と関数に対し、テンプレートが定義されています。これらの関数は全てのグローバルです。保存 / 復旧関数は、低電力コンテキスト外で使用することもできます。

11.4.2 デザインへの配慮

必要な場合のみ、低電力保持用の関数を定義します。これらの関数は、別個ファイル `$INSTANCE_NAME_PM.c` 内に置かれます。これにより、アプリケーションが低電力機能を使用

しない場合、静的データ構造を含む .o ファイルを、リンク時に削除することができます。さらに、`\$INSTANCE\_NAME`_Enable() および `\$INSTANCE\_NAME`_Stop() 関数が、代替アクティブレジスタの有効を有効 / 無効にします。これにより、代替アクティブテンプレートを自動的に有効および無効にするメカニズムを提供します。

11.4.3 ファームウェア / アプリケーション プログラミング インターフェースの要求項目

11.4.3.1 データ構成テンプレート

```
typedef struct _`$INSTANCE_NAME`_BACKUP_STRUCT
{
    /* コンポーネントブロックの 有効状態を保存 */
    uint8 enableState;

    /* コンポーネントの揮発性レジスタを保存 */

} `$_INSTANCE_NAME`_BACKUP_STRUCT;
```

11.4.3.2 保存 / 復旧メソッド

揮発性レジスタ値を静的データ構成に保存します。特定コンポーネントのレジスタ値のみを保存します。

```
`$_INSTANCE_NAME`_SaveConfig()
{
    /* バックアップデータ構成へ揮発性レジスタの値を保存 */
}
```

揮発性レジスタ値を静的データ構成から復旧します。特定コンポーネントのレジスタ値のみを復旧します。

```
`$_INSTANCE_NAME`_RestoreConfig()
{
    /* 揮発性レジスタ値をバックアップデータ構成から復旧します。 */
}
```

コンポーネントの有効状態を保存します。この状態を使用して、復帰時にコンポーネントを開始するかどうか判断します。コンポーネントを停止し、設定を保存します。

```
`$_INSTANCE_NAME`_Sleep()
{
    /* コンポーネントの有効状態 -- 有効化 / 無効化 */
    if (/* コンポーネントのブロックが有効 */)
    {
        backup.enableState = 1u;
    }
    else /* コンポーネントのブロックが無効 */
    {
        backup.enableState = 0u;
    }
    `$_INSTANCE_NAME`_Stop();
    `$_INSTANCE_NAME`_SaveConfig()
}
```

コンポーネント設定を復旧し、コンポーネントが有効となるかを判断します。

```
`$_INSTANCE_NAME`_Wakeup()
{
    `$_INSTANCE_NAME`_RestoreConfig()
    /* コンポーネントブロックの 有効状態を復旧 */
    if (0u != backup.enableState)
    {
```

```

/* コンポーネントのブロックが有効 */
`$INSTANCE_NAME`_Enable();
} /* コンポーネントのブロックが無効の場合、何もしない */
}

```

11.4.3.3 有効化および停止関数への追加

コンポーネントの別のアクティブレジスタ有効を有効化

```

`$INSTANCE_NAME`_Enable();
{
    /* 代替アクティブ中にブロックを有効化 */
}

```

コンポーネントの代替アクティブレジスタ有効を無効化

```

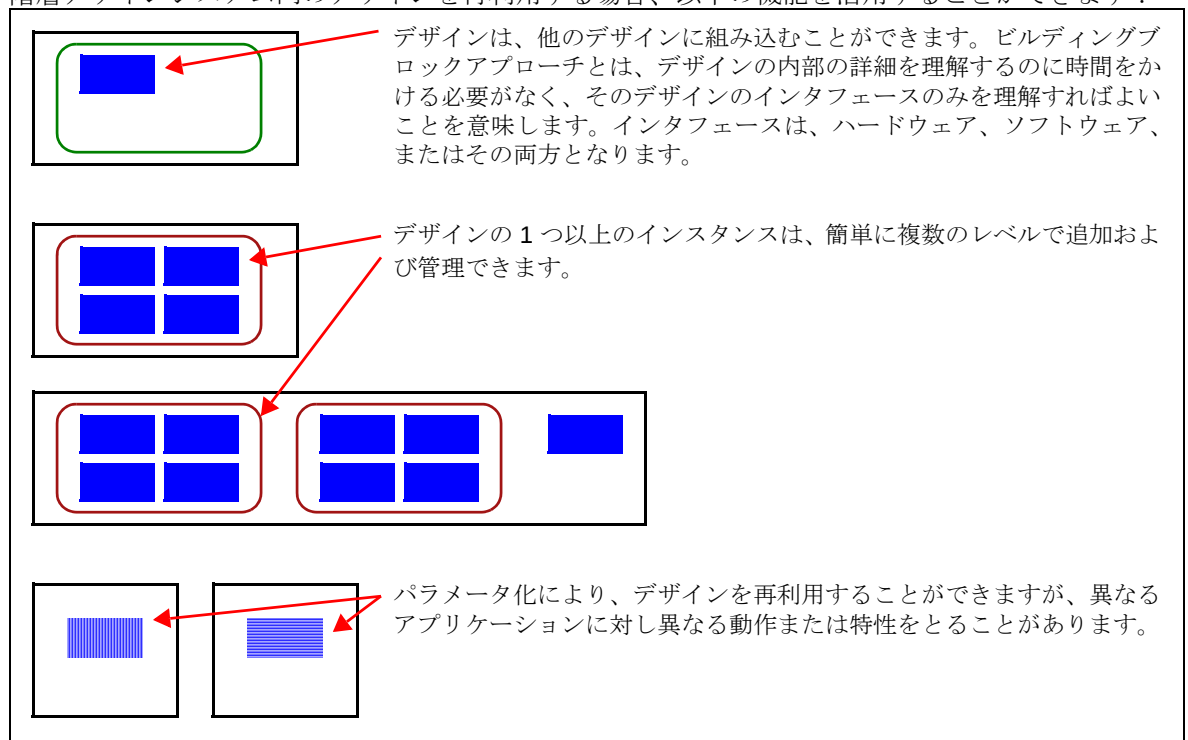
`$INSTANCE_NAME`_Stop();
{
    /* 代替アクティブ中にブロックを無効化 */
}

```

11.5 コンポーネントの内包

11.5.1 階層デザイン

階層デザインシステム内のデザインを再利用する場合、以下の機能を活用することができます：

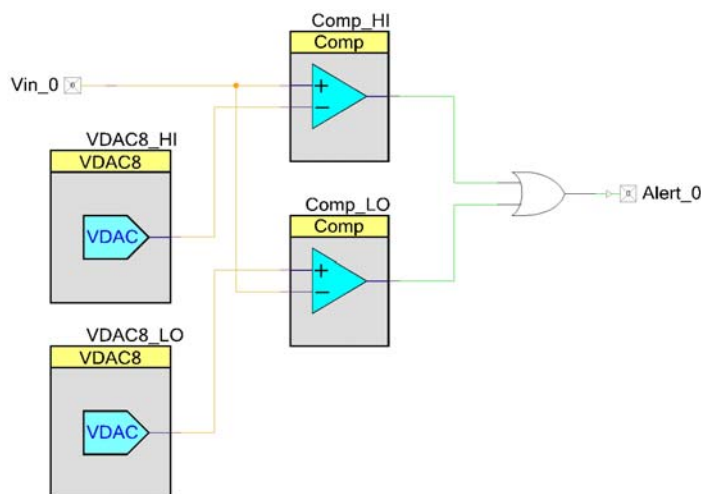


PSoC Creator にてデザインの再利用を簡単にするためには、PSoC Creator コンポーネントとして内包する必要があります。デザインは、以下の条件に 1 つ以上該当する場合、内包化および再利用を検討する価値があります。

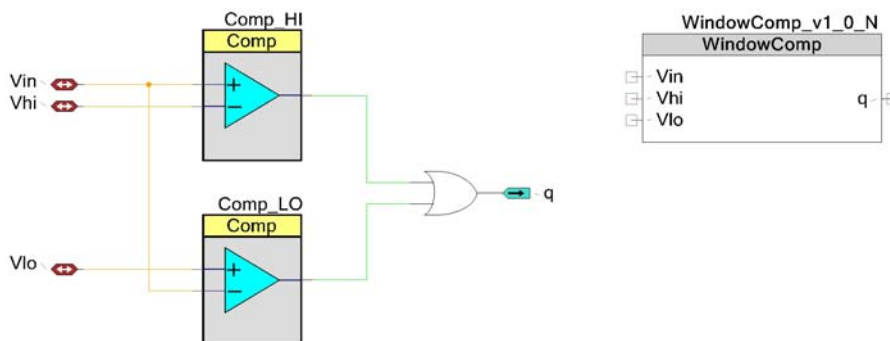
- 特定関数を実装一般的に、1つのことをうまくやることが重要です。
- ハードウェアターミナルまたは API コールの形式の、限られた比較的少ない入力および出力のみ持っている一般的に、少ない方がいいのですが、重要な機能性が損なわれるほど少なくては困ります。

以下のページに、いくつかの例を提供します：

コンポーネントとして IP を内包化する簡単な例の 1 つとして、デザイン内にウィンドウコンパレータが必要な場合どうするかを考えます。ウィンドウコンパレータは、入力電圧が 2 つの参照電圧間にある場合、アクティブです。PSoC Creator では、おそらくこのように設計します：

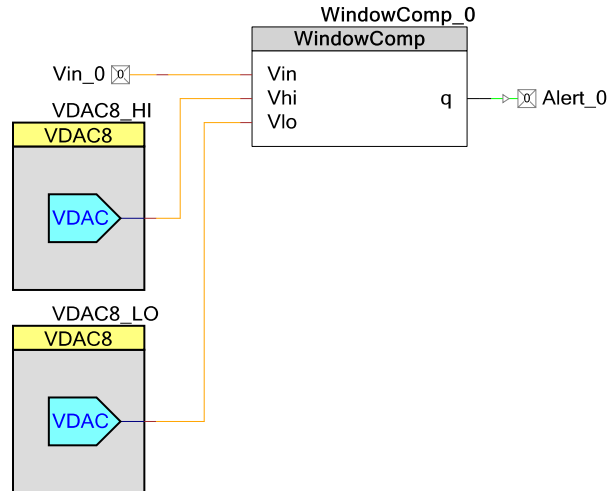


このデザインは、コンポーネントとして内包することが有効です。1つの特定の機能のみを実行します：ウィンドウコンパレータ。さらに、入力および出力の数が限られています。また、コンパレータを開始する、小さい API もあります。ゆえに、以下のようにデザインの重要な機能を、シンボル付きのコンポーネントとして内包できます：

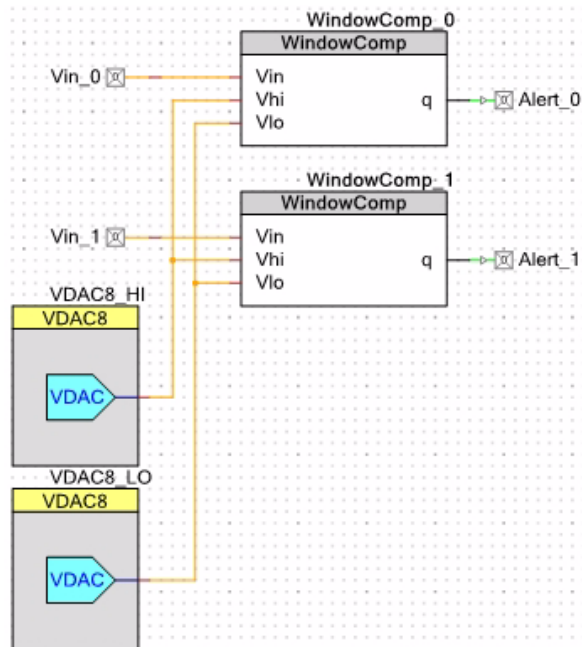


設計を内包する場合、コンポーネントから何を外すか決定することも重要です。上記の例では、VDAC がコンポーネントに含まれています。ただし、これは、1つの電圧と 2 つの参照電圧を単に比較するという、デザインの重要な機能ではありません。参照電圧は、VDAC または他のソースによって提供することもできます。この場合、VDAC を除外するのが適切です。

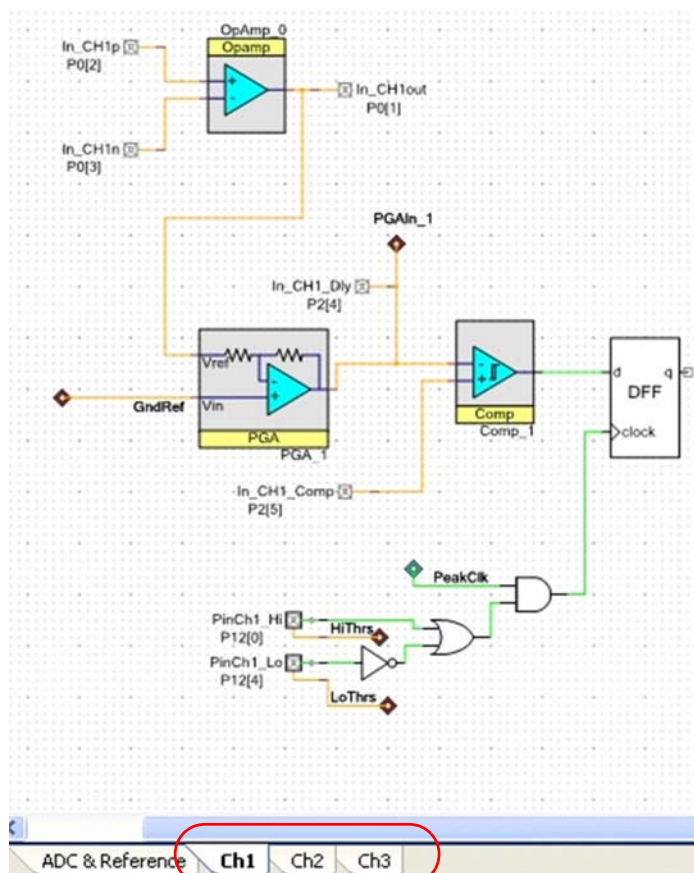
内包により、トップレベルデザインは簡単になります：



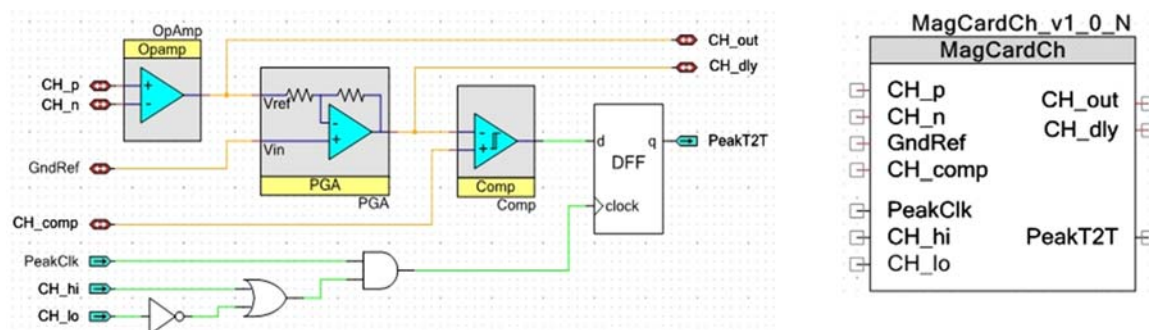
多くの場合、デザインをスケールすることが簡単です：



以下の磁気カードリーダーの例は、より複雑です。このデザイン内には、単一チャネルを読みこむための回路の複数のコピーとともに、セントラル ADC とリファレンスセクションがあります：



この場合の基本的なデザインの機能性は、チャンネルからのアナログ入力を、デジタルシステムが読み込み可能な形式へ変換することです。また、明確で機能が制限されている関数であるので、内包に適しています。



コンポーネントから除外する内容を確認する必要があります。オリジナルデザインは、異なる閾値の 2 つの SIO ピンを使用します。これらは、コンポーネント内に入れることもできますが、トップレベルデザインにピンを置くことが推奨されます。また、基本的なデザインは、デジタル信号 CH_hi および CH_lo が必要ですが、ソースは必要ではありません。

PSoC 3/5 のオペアンプは特定デバイスピンと密接に関連しているため、コンポーネントからオペアンプを外すことが適切です。

最後に、この設計は制限されたアナログリソースと配線を利用しているため、複数のインスタンスは小さな PSoC 3/5 デバイス内に適合しないことがあります。この事実、および電圧、温度、およびスピード制限は、ユーザーに伝える必要があります。たとえば、どの PeakClk の周波数を使うべきだろうか？オリジナルデザインのトップレベルでは知られているものの、コンポーネントを再利用するにはわからない可能性があります。

これにより、興味深い問題が提起されます：いつデザインを内包すべきでないか、です。デザインが以下の基準を 1 つ以上満たす場合、内包しないことが推奨されるかもしれません。内包化された場合、重要な問題点および制限は、ユーザーがコンポーネントを再利用しようと試みた後すぐに知ることができるよう通知すべきです。

- PSoC 3/5 内の重要なリソースを組み込んであるため、高レベルデザイン内でこれらを利用できません。具体例としては：
 - ADC_DelSig、CAN、USB、および 1²C などの、単一インスタンス、かつ固定機能ブロックの使用
 - UDB、コンパレータ、DAC、オペアンプ、タイマー、DMA チャネル、割り込み、ピン、クロックなど、豊富なリソースの多量の利用
 - アナログまたは DSI 配線、フラッシュ、SRAM、CPU サイクルなど、一般的でないリソースの多量の利用
- 特定の状況化でしか動作しない、たとえば特定 CPU または bus_clk スピード、Vdd レベル、または PSoC 5 ファミリーなどの特定パーツのみでしか動作しない場合です。
- IP の複数インスタンスが実装できません。

11.5.2 パラメータ化

PSoC Creator 内では、コンポーネントはパラメータ化することができます。つまり、コンポーネントのインスタンスの挙動は、ビルドタイムに設定されます（通常は、ダイアログボックスの使用します）。ハードウェアおよびソフトウェアの挙動の両方が、パラメータ化によって設定されます。パラメータ化は、以下の場合に使用されます：

- IP の異なるインスタンスが、わずかに異なる動作をするが、全体機能が同じ場合です。例として、ファンコントローラのアラート出力は、アクティブハイまたはアクティブローに設定することができます。
- 動作の相違点は、ランタイムに変更されることを期待されません。

パラメータ化が異なるインスタンスの機能性への大きな違いを起こす場合、複数コンポーネントとしてデザイン内に内包することを考慮すべきです。PSoC Creator は、複数コンポーネントが、単一のライブラリプロジェクト内でパッケージ化することが可能なので、コンポーネントの異なるバージョンを同梱することが可能です。例として、単一のファンコントローラのコンポーネントが、コントロールされるファンについてのパラメータを持つことができます。SPI および 12C などの、2 つの異なるインタフェースのあるファンコントローラは、"Fan Control" ライブラリプロジェクト内の 2 つのコンポーネントになっていることもできます。

11.5.3 コンポーネントのデザインへの配慮

11.5.3.1 リソース

コンポーネントは、ピンコンポーネントの組み込みを避けるべきです。代わりに、ターミナルを使用すべきです。なぜなら、ユーザーは、より高レベルでコンポーネントシンボルでピンを接続することができるからです。

クロックコンポーネントは、コンポーネント内で内包する場合としない場合があります。クロックを組み込まないメリットは、複数のコンポーネントを同じクロックに接続することができるので、クロックリソースを節約することができます。コンポーネントには、クロックを内部にいれるか否かの、パラメータを含めてもかまいません。

割り込みおよび **DMA** チャンネルは、コンポーネント内で内包する場合としない場合があります。コンポーネントは、**IRQ** または **DRQ** に接続される「データあり」シグナルを生成することができます。この場合、割り込みおよび **DMA** はコンポーネント内に組み込んではいけません。

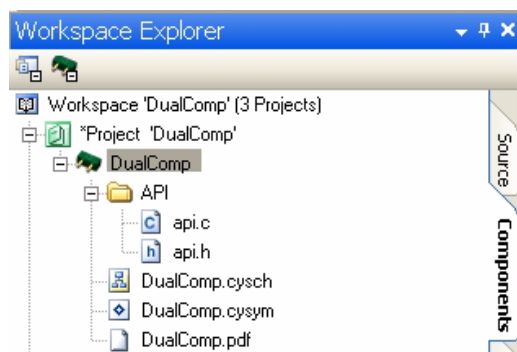
11.5.3.2 電源管理

コンポーネントは、スリープ、ハイバネート、復帰などの、電源管理をサポートするよう設計されるべきです。API には、適切な関数が含まれるべきです。Cypress 提供のコンポーネントは、電源管理 API を実装する、多くの例を提供しています。

11.5.3.3 コンポーネントの開発

PSoC Creator コンポーネントは、基本的に入れ物です。ターゲットデバイス内でビルドおよびインストールすることはできません。構築およびインストールされる、標準 **PSoC Creator** プロジェクトにリンクされるものです。

コンポーネントには、以下の異なるファイルタイプを含むことができます：シンボル (.cysym)、回路図 (.cysch)、Verilog (.v)、ファームウェアソースコード (.c, .h, .a51, etc.)、および文書 (.pdf, .txt)。コンポーネントは、他のコンポーネントも参照することができます。このため、階層デザイン設計が可能です。



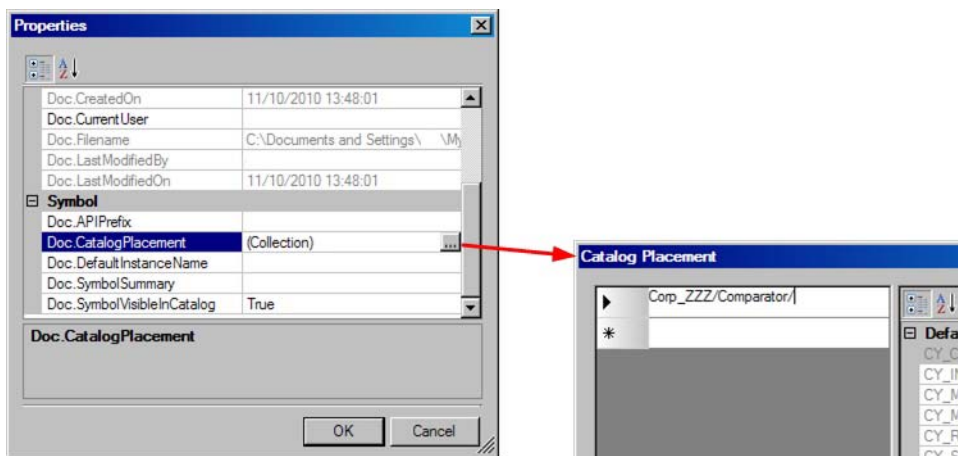
この仕様にに基づき開発されたコンポーネントは、最小限でもシンボルファイルおよびデータシートファイルを含みます。その他のファイルタイプは全て、コンポーネントの機能により必要かどうかが決まります。例として、ハードウェアのみのコンポーネントは、回路図または **Verilog** ファイルのみを持つことができます。また、ファームウェアのみのコンポーネントは、API 用の .h および .c ファイルのみを持つことができます。

参照コンポーネントのシンボル

シンボルは、常にアノテーションを含むインスタンス名 ``=$INSTANCE_NAME`` (バックワードシングルクォート) を持つべきです。追加のアノテーションが、ユーザーがコンポーネントの機能をすぐに理解できるように加えられます。

コンポーネントのカatalogにおける配置

シンボルプロパティ **Doc.CatalogPlacement** は、**PSoC Creator** コンポーネントカatalog内のどこで、ユーザーがコンポーネントを確認することができるかを管理します。特に複数のコンポーネントを作成する場合に、一貫したタブおよびツリーノードの命名法を使用すべきです。タブを少なくし、ツリーノードを多く使用することで、ほとんどのユーザーのスクリーン上のコンポーネントカatalog内でより見やすくなります。



詳細情報については、[27 ページの カatalogの配置の定義](#)を参照してください。

コンポーネントデータシート

コンポーネントを適切に文書化するため、データシートを用意すべきです。データシートには、上記の設計についての考慮が含むべきです。データシートを追加する方法については、[79 ページの データシートの追加 / 作成](#)を参照してください。

コンポーネントのバージョン付け

コンポーネント名には、バージョン情報を、以下をコンポーネント名に追加することで含むことができます：

"_v<major_num>_<minor_num>_<patch_level>"

<major_num> および <minor_num> は、メジャーおよびマイナーバージョン番号をそれぞれ指定する整数です。<patch_level> は、単一の英小文字です。コンポーネントにはバージョンをつけるべきです。バージョン付けについては、[12 ページの コンポーネントのバージョン](#)参照してください。

11.5.3.4 コンポーネントのテスト

再利用可能なデザインは、**PSoC Creator** ライブラリプロジェクト内で、コンポーネントとして内包されます。ただし、ライブラリプロジェクトもコンポーネントも、単体では実用的とはいえません。ライブラリプロジェクトは、ビルド、プログラム、およびテストすることができません。そこで、コンポーネントに対し、1つまたは複数の標準プロジェクトを、コンポーネントの再利用可能なデザインの機能をテストまたは実証する目的で開発します。

各パラメータが可能な限り多くの異なる設定をとる、参照コンポーネントを含むべきです。これを実行するためには、コンポーネントの複数のインスタンスを使用する、もしくは複数のテスト もしくはデモプロジェクトを生成する必要があるかもしれません。コンポーネントにおける **API** の全ての関数は、少なくとも 1 度呼び出してください。全てのマクロは、少なくとも 1 度呼び出してください。

可能な限り、テストプロジェクトは、さまざまな設定の PSoC 3 および PSoC 5 をサポートすべきです。例として、標準およびブートロード可能、デバッグおよびリリース、異なる PSoC 5 コンパイラ、異なるコンパイラ最適化設定が考えられます。

11.6 Verilog

多くのデジタルコンポーネントは、Verilog を利用してコンポーネントの実装を定義します。詳細情報については、[34 ページの Verilog を用いた実装](#)を参照してください。

この Verilog は、Verilog の合成可能サブセット内で書き込まれる必要があります。Verilog ベースのコンポーネントを作成する場合、Verilog サブセットに準拠することに加え、従う必要がある追加のガイドラインがあります。これらのガイドラインのほとんどは、合成ツール用のベストプラクティスとして受け入れられている慣習です。追加ガイドラインは、PSoC Creator コンポーネントの開発に関連する、特定状況下における推奨です。

11.6.1 Warp: PSoC Creator 合成ツール

設計内でデジタルロジックを処理する場合、PSoC Creator は、自動的に、PSoC Creator とインストールされる、Warp 合成ツールを実行します。これは、Cypress プログラム可能ロジックデバイスで以前使用された、Warp 合成ツールの PSoC 固有バージョンです。

PSoC Creator がサポートする Verilog の固有合成サブセットは、ワーブ Verilog 参照ガイド内で示されています。この合成可能サブセットは、他の Verilog 合成ツール内で実行されるサブセットと同じです。サポートされている Verilog 構造体の詳細説明については、ガイドを参照してください。

Verilog デザインの合成可能部分は、PLD ロジックとして合成されます。残りの UDB リソースは、Verilog デザインにインスタンス化されますが、合成プロセスにより仮定されません。

11.6.2 合成可能なコードのガイドライン

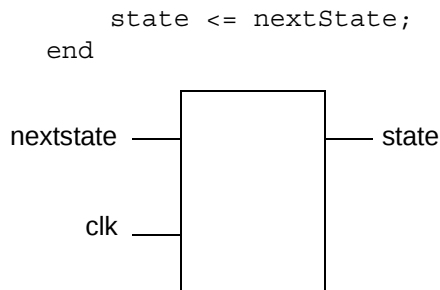
11.6.2.1 ブロッキング型アサイメントと非ブロッキング型アサイメント

Verilog 言語には、2 つのアサイメント文の形式があります。ブロッキング型アサイメント文は、次のステートメントへ進む前に、すぐに値を割り当てます。非ブロッキング型アサイメント文は、時間が経った後に値を割り当てます。これらの 2 つの割り当てタイプには、シミュレーションの観点から別の意味があります。これらの割り当てが合成の観点および最終的なハードウェアから考慮した場合、結果が異なることがあります。合成デザインのベストプラクティスは、シミュレーション結果と合成結果が適合するよう開発されました。

非ブロッキング型アサイメントを使用したシーケンシャルロジックの実装

この規則により、クロックされたロジックのシミュレーション結果は、ハードウェア実装と合致します。ハードウェア実装時に、その状態の前の新しい状態の全てのレジスタクロックが、次の状態の計算に使用されます。

```
always @(posedge clk)
begin
```



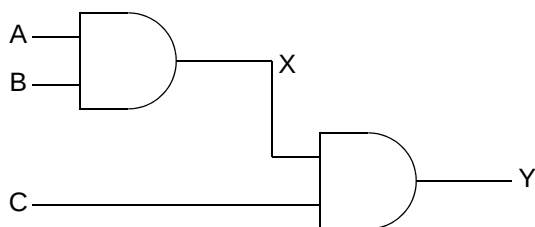
ブロッキング型アサインメントを使用した、「常に」ブロック内のシーケンシャルロジックの実装

この規則により、シミュレーションでは、「常に」ブロックが評価される直後に、値をシグナルへ割り当てます。これは、組み合わせロジックを通した遅延が無視される場合、ハードウェア実装と合致します。この場合、ロジックの 1 レベルの結果は、すぐにロジックの次のレベルの計算に使用されます。

```

always @(A or B or C)
begin
    X = A & B;
    Y = X & C;
end

```



同じ「常に」ブロック内で、ブロッキング型と非ブロッキング型アサインメントを混合しないでください。

この規則は、単一割り当ての実装以上を必要とする組み合わせロジックは、シーケンシャル「常に」ブロックと異なるブロックセパレート内で記述されるべきことを示します。この組み合わせロジックは、別の組み合わせブロック内、もしくは継続的な割り当て文内で実装できます。

複数の「常に」ブロックに対し、同じ変数を割り当てないでください。

PSoC Creator 内の合成エンジンは、この規則を行使し、この違反があった場合、エラーを生成します。

11.6.2.2 ケース文

Verilog 内には、3 つのケース文の形式があります：case、casex、および casez です。これらの文を用いてコーディングする場合、以下の規則に従ってください。

全てのケース文は、完全に定義してください。

これは、デフォルトケースを全てのケース文に置くことが、最適の実装法です。その場合、自動的にこの規則が満たされます。全てのケースが対象となるデフォルト文を含むことで、ケースアイテムが後にコードの開発時に変更されても、この規則が継続的に満たされることになります。ケースアイテムが削除される、もしくはケースの幅が変更される場合でも、デフォルトステートメントはすでに存在しています。

この規則は、特に、組み合わせ「常に」ブロック内で重要です。組み合わせ結果を割り当てるデフォルト条件を含めることで、設計内のラッチの合成を防ぎます。

casex の代わりに casez を使用します。

casex および casez 文は同じであり、同じ結果を合成します。相違点は、シミュレーション中に生成される結果です。ケースアイテムが「関係なし」ビットを指定する場合、casex 文は、"x" または "z" の入力値に適合しますが、casez 文では、"z" 値のみが、ケースアイテム内の「関係なし」ビットに適合します。casex 文では、シミュレーションは、値が初期化されていない、というデザインエラーを見逃すことがあります。PSoC デザインでは、デザインの合成可能部の内部に "z" 値がないため、casez 文では問題になりません。

casez 文内で、「関係なし」ビットの "?" を使用します。

この規則は、合成可能コードに影響しません。"?" が「関係なし」ビットであると、明確に示すための規則です。"?" が使用される別のメソッドでは、この意図がわかりません。

条件がオーバーラップする casez 文を使用しないでください。

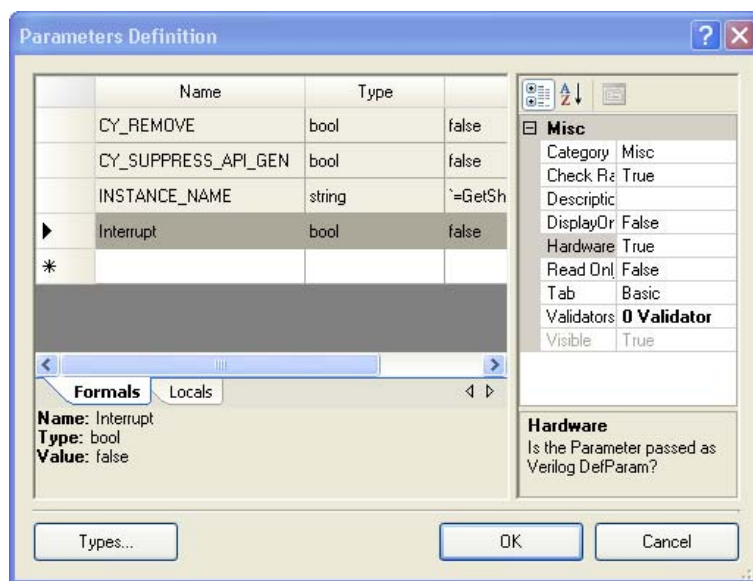
ケースアイテムにオーバーラップする casez 文は、プライオリティエンコーダを合成します。結果のロジックは、シミュレーション結果と同じになりますが、目的のロジックを読み取ることが困難になります。if-else if-else 文の使用により、プライオリティエンコーダの意図がはっきり伝わります。

11.6.2.3 パラメータの取り扱い

パラメータを使用することで、Verilog インスタンスは、コンポーネントインスタンスの指定要件に基づき合成することができます。

パラメータをコンポーネントに渡す

シンボルに設定されたコンポーネントのパラメータを、Verilog インスタンスに渡す場合、Misc 設定内の "Hardware" でパラメータを真に設定します。



Verilog モジュールへパラメータが渡されると、パラメータ設定は合成の実行前に適用されます。指定されたインスタンスは、パラメータ設定に基づき合成されます。この方法は、ビルドタイムで作成できる設定に最適ですが、ランタイムに決定する必要がある設定に使用することはできません。

ランタイムで変更できる設定は、データパスまたはコントロールレジスタから、ソフトウェアを通してコントロールする必要があります。どのパラメータが静的で、どれがダイナミック（ランタイムに変更可能）となるかの判断は、コンポーネントの複雑性およびリソース要件に影響します。

ハードウェアパラメータがある Verilog ベースのコンポーネントは、パラメータ文のあるパラメータ値にアクセスできます。このパラメータ文は、シンボルから自動的に生成される Verilog テンプレート内で自動的に生成されます。初期値は、常に、コンポーネントの指定インスタンスの値設定で上書きされます。

```
parameter Interrupt = 0;
```

ステートメントの生成

異なるパラメータの値が、大きく異なる Verilog コードを必要とすることが頻繁にあります。この機能性は、Verilog 生成文を使用して実装できます。例として、コンポーネントが、8 ビットと 16 ビットの両方を実装する場合、8 ビットまたは 16 ビットのデータパスコンポーネントのインスタンス化が必要になります。この機能性を実装する唯一の方法は、生成文の使用です。

```
parameter [7:0] Resolution = WIDTH_8_BIT;
```

```
generate
if (Resolution == 8) begin :dp8
    // 8 ビット用コード
end
else begin :dp16
    // 16 ビット用コード
end
endgenerate
```

パラメータをモジュールインスタンスに渡す

Verilog デザインは、設計内でインスタンス化される、他のモジュールにパラメータを渡す必要があります。パラメータは、多くの標準ハードウェアモジュールで必要とされます。

パラメータを Verilog 内で渡すするには、2 つの方法があります。オリジナルの方法は、defparam 文です。Verilog-2001 仕様から使用できるもう一つの方法では、名前付きパラメータを使用してパラメータを渡します。モジュールインスタンス内の "#" の後に、パラメータ名と値が含まれます。パラメータを、常に名前付きパラメータを使用して渡すことが推奨されます。

```
cy_psoc3_statusi#(.cy_force_order(1),
    .cy_md_select(7h07), .cy_int_mask(7h07))
stsreg(
    .clock(clock),
    .status(status),
    .interrupt(interrupt)
);
```

パラメータ化されたデータパスインスタンス

データパスインスタンスは、単一の、複雑に設定されているパラメータに基づき構成されます。複数バイトデータパスの構成、または複数のデータパス幅をサポートできる生成文を用いるコンポーネントのために、同じ設定値が、複数の設定パラメータに対し使用されることがあります。同

じ情報の複製は、管理するのが困難な上エラーが発生しやすいため、設定データが 1 箇所には存在しないように、パラメータ値を使用します。

```
parameter config0 = {  
    `CS_ALU_OP_PASS,  
    // 値の残りはここに表示しません。  
};  
  
cy_psoc3_dp8 #(.cy_dpconfig_a(config0)) dp0 (  
    // インスタンスの残りはここに表示しません。  
);
```

11.6.2.4 ラッチ

ラッチは、PSoC デザインにおいて回避すべきです。プログラム可能なロジックデバイスには、アーキテクチャにラッチが組み込まれていることがあります。これは、PSoC デバイスには当てはまりません。各マクロセルは、クロックエッジに対し組み合わせである、または登録されています。ラッチが PSoC 設計内で生成される場合、ラッチはクロス連結ロジックを使用して実装されます。ラッチの使用は、生成されるループ構成のため、タイミング分析の実施を制限します。実装に基づく組み合わせラッチは、フィードバックパスの長さがコントロールされないため、タイミングの問題を抱えることがあります。

ラッチが設計内に存在する場合、ラッチが目的通りに機能しないことが多々あります。ラッチは、出力に値が割り当てられていないブロックを通して少なくとも 1 つのパスがある、組み合わせ「常に」ブロックからの出力の合成中に示されます。これは、各 if-else 項内に出力への割り当てを行い、if-else チェーンの最後に else 項を含めることで回避されます。また、「常に」ブロックの再上部で、各出力用のデフォルト値を割り当てることで回避できます。

11.6.2.5 リセットとセット

リセットおよびセット信号は、以下の規則に従うべきです。この規則と説明は、リセットという単語を使用しますが、これは、リセットとセット機能の両方に適用されます。

可能ならば同期リセットを使用する

非同期リセットは、コントロール信号から、影響する全レジスタの出力への、組み合わせパスのように機能します。非同期リセットを使用する場合、ロジックを通して伝達されたリセット信号のタイミングが、タイミングの制限となります。同期リセットを使用する場合、唯一の追加で必要なタイミング分析は、クロックとリセット信号の関係のみです。

クロックが自由に動作している場合、同期リセットを使用する

非同期リセットが必要な唯一のケースは、クロックが自由に動作していない場合のみです。レジスタをクロックが実行している間にリセットする必要がある場合、非同期リセット信号が次に、必要になります。クロックが自由に動作している場合、次に、同期したリセットは、追加タイミングの問題がある非同期シグナルと、同じ結果になります。

非同期リセットとセットは、同じレジスタに対し使用することはできません。

PLD マクロセルのハードウェア実装は、非同期リセットとセットに同じ信号を使用します。この信号は、リセットに使用、セットに使用もしくは信号を使用しない、のうちのいずれかを選択する必要があります。非同期リセットとセットを両方使用することは、ハードウェア的に不可能です。

11.6.3 最適化

11.6.3.1 パフォーマンス最適化デザイン

一般的なデジタルコンポーネントのパフォーマンスは、2つのレジスタ間の最長組み合わせパスにより定まります。組み合わせパスの実行に要する時間は、利用可能なハードウェアに対するVerilog設計のマッピングにより決定されます。

コントロールストアアドレスの登録

データパスを使用するデザイン内の最長パスは、データパスアキュムレータもしくはマクロセルフリップフロップから、組み合わせロジック（ALU、条件生成、もしくはPLDロジック）を通して、最終的に、コントロールストアアドレスに到達するものです。パフォーマンスをよくするために、このロングパスは、2つの短いパスに分けられます。このパスにレジスタを挿入するための自然な場所は、しばしばコントロールストアアドレス入力前です。多くの場合、この入力は、マクロセル出力により駆動されます。全てのマクロセル出力にはオプションのレジスタがあるため、この出力を登録することにより、コンポーネントが消費するリソースは増加しません。この方法でのパイプライン化は、コンポーネントの操作を変更するため、このタイプの実装は、コンポーネントの初期アーキテクチャ定義の一部とすべきです。

条件付き出力の登録

他のマクロセル同様、PLD内にはオプションとして使用可能なフリップフロップがあり、データパスが生成する各条件は、組み合わせシグナル、またはレジスタ値として利用可能です。このようなレジスタは、リソース使用に影響することなく、デザインをパイプライン化するために使用できます。

出力の登録

コンポーネントの典型的な使用法によっては、コンポーネントの出力を登録することが有益になります。複数の出力があり、これらがピンへ送信される場合、出力の登録により、出力信号間のタイミング関係が、より予測可能となります。コンポーネントからの出力が、別のコンポーネントの供給に使用される場合、システムのパフォーマンスは、このコンポーネントからの出力タイミングと対象コンポーネントの入力タイミングにより異なります。出力が登録されている場合、そのパスの出力部分が最小化されます。

PLDパスの分割

各PLDには、12の入力と8のプロダクトタームがあります。出力が、単一のPLDが実装される入力数またはプロダクトターム数以上を必要とする場合、出力の式は、複数のPLDに分割されます。これにより、PLD伝達遅延、および必要な追加される各PLDの遅延が発生します。パフォーマンスを改善するため、これらのレベルは、いずれもパイプライン式に計算されます。レジスタが各マクロセル出力で利用可能なため、これはリソースの消費を増やすことなく実行できます。ただし、パイプライン化は、ロジックのタイミング関係を変更します。

特定の出力を計算する必要がある入力数を判断するために、if文、ケース文、および特定の出力用のアサイメント文内で使用される、全ての入力を数える必要があります。

11.6.3.2 サイズ最適化デザイン

通常、設計のプログラム可能なデジタル部分のサイズは、PLDリソース、もしくはデータパスリソースによって制限されます。データパスリソースと同等のロジックの機能性は、PLDアレイで利用可能なロジックよりかなり大きいです。データパス実装が可能な場合、これが最も効率的な実装方法であることが多いです。PLDのロジックを構築する場合、PSoC PLDの固有のアーキテクチャを考慮しなければなりません。

12 入力 PLD アーキテクチャを使用する

PSoC PLD には、4 つの出力と 12 の入力があります。デザインが同期されると、あらゆる出力に必要な入力数は 12 に制限されます。最終出力が 12 以上の入力が必要とする場合、機能は、それぞれ 12 以下の入力を必要とする、複数のロジックに分割されます。全てのロジックが 12 以下の入力を必要とする出力へ分割された後、このロジックは PLD にパックされます。理想的には、各 PLD は、4 つの出力の生成に使用されます。これは、組み合わせた時に 12 の入力しか必要なく、4 つの出力がある方程式が見つかる場合のみ可能となります。例として、単一の出力に全 12 入力が必要な場合、同じ PLD 内に置くことができる他の出力は、同じ入力を使用するもののみになります。

最初のステップは、コンポーネントをビルドし、レポートファイルにおける、パックされた PLD の平均統計を見ることです。

```

2629 -----
2630 PLD Packing Summary
2631 -----
2632 Packed 104 macrocells into 51 PLDs.
2633
2634 PLD Resource Type :      Average/LAB
2635 =====
2636 Inputs :                11.31
2637 Pterms :                 2.02
2638 Macrocells :             2.04

```

マクロセルの平均数が 4.0 より非常に低い場合、また入力数が 12.0 に近づいている場合、必要な入力数がデザイン内の制限要因となります。パッキングを改善するためには、必要な入力数を減らす必要があります。

一部のケースでは、入力グループを単一の入力に統合するよう、方程式を再構成することで可能になります。例として、複数ビットの比較が入力の場合、これは単一の結果の入力に置き換えられます。この値が、複数の出力を計算するために使用される場合（複数ビットレジスタ）、ロジックの特定部分の分割により、サイズが大きく低下することがあります。信号をマクロセルの出力になるよう強制するためには、**cy_buf** コンポーネントを使用します。**cy_buf** の出力は、常にマクロセル出力となります。

```

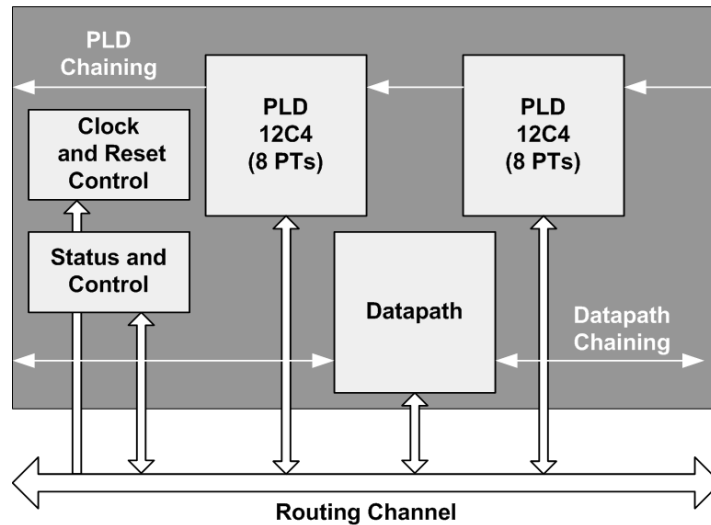
wire cmp = (count == 10'h3FB);
cy_buf cmpBuf (.x(cmp), .y(cmpOut));

```

11.6.4 リソースの選択

Verilog コンポーネントの実装は、PSoC UDB アーキテクチャ内で利用できる、さまざまなリソースを使用することがあります。どのリソースを選択するかは、必要な機能により異なります。全てのデジタルコンポーネントは、PLD ロジックを使用します。データパスインスタンスを使用するコ

コンポーネントもあります。全ての合成ロジックは、PLD 内に置かれます。データパス実装は、Verilog 設計内にデータパスインスタンスを含めることで行われる必要があります。



11.6.4.1 データパス

データパスリソースには、PLD リソースの数だけ方程式ロジックゲート能力があり、アプリケーションに適用できる場合、データパスリソースが最初の選択になります。

一般的なデータパスのアプリケーション

- FIFO が必要な操作 2 つの データパス FIFO は、デジタルシステム内の唯一のプログラム可能な FIFO パスです。いくつかのデザインでは、データパスは、PLD ベースのハードウェア設計に FIFO 機能性を追加するためだけに使用されます。
- ほとんどの計数関数（インクリメント、デクリメント、定数ステップ）レジスタは、計数、事前ロード、比較、捕獲に利用できます。
- 並列接続が CPU 側にあり、直列接続がハードウェア側にある場合の、並列から直列、および直列から並列への変換

データパスの制限

- データパスへの並列入力制限されます。これは、他のハードウェアが並列値を提供する必要がある場合、データパスの使用を制限します。並列ハードウェアローディングの代替手法が存在する可能性があります。十分なサイクルが利用可能な場合、値を並列的にシフトすることが可能です。CPU または DMA が値にアクセスできる場合、これらが値をデータパス内の FIFO またはレジスタに書き込むことができます。
- 並列出力は可能ですが、並列出力値は、常にデータパス ALU への左側の入力です。ALU への左側の入力は、A0 または A1 のいずれかであるため、出力は A0 または A1 に制限され、並列出力を使用中に実行できる ALU オペレーションに対し、この値を ALU 関数の左側の入力として使用するものに制限します。
- ALU 関数は一度に一つだけ実行できます。複数の操作が必要な場合、実効クロックの倍数を使用する、マルチサイクル実装が可能なかもしれません。
- 利用可能なダイナミック操作は 8 つあります。これは、十分なオペレーションのほうですが、複雑なマルチサイクル計算（シフト、加算 (add)、インクリメント）においては、制限になることがあります。

- データパスにより読み書き可能なのは、2 レジスタのみです。最大で 6 つのレジスタリソース (A0、A1、D0、D1、F0、F1) がありますが、2 つのレジスタのみ、書き込みおよび読み込みすることができます (A0、A1)。

11.6.4.2 PLD ロジック

PLD ロジックリソースは、最も柔軟性のあるデジタルハードウェアリソースです。

一般的な PLD アプリケーション

- ステートマシン
- 小カウンタ (<= 4 bits)、もしくは他のハードウェアに対する並列入力および出力が必要なカウンタ
- 汎用組み合わせロジック

PLD の制限

- PLD には、CPU から、および CPU への直接パスがありません。CPU のデータを取得するためには、コントロールレジスタが使用されます。CPU へデータを送信するには、ステータスレジスタが使用されます。
- レジスタビットの最大数は、UDB 数の 8 倍です (選択デバイスにより異なります)。コントロールおよびステータスレジスタは、このビット数を増加するために使用できますが、この両リソースとも、PLD により読み込みおよび書き込みの両方な可能なレジスタを提供しません。
- PLD につき 12 しかない入力ビットは、ワイド関数の効率およびパフォーマンスを制限します。例えば、wide mux 関数は、PLD にうまくマッピングできません。

A. 式評価



PSoC Creator の式評価は、デバッガ内以外の、PSoC Creator 内の式の評価に使用されます。これには、パラメータ値、文字列内の式、およびコード生成テンプレートが含まれます。式評価の言語は、Perl 5 に類似しています。大半の演算子を優先度を含めて借用しているものの、PSoC Creator のアプリケーションにより適合している、異なる型システムを採用しています。

A.1 評価のコンテキスト

評価のコンテキストには 2 つの基本的なものがあります：文書コンテキストと、インスタンスコンテキストです。文書コンテキストは、文書のフォーマルおよびローカルパラメータ、それと文書のプロパティから構成されます。インスタンスコンテキストは、インスタンスのフォーマルおよびローカルパラメータ、それと参照されたシンボルの、文書のプロパティから構成されます。

フォーマルおよびローカルパラメータの評価は、[20 ページのフォーマルおよびローカルパラメータ](#)にて解説されています。アノテーションに、式を埋め込むことができます。大半のアノテーションは、文書コンテキストで評価されます。インスタンスに関連付けられるアノテーションは、インスタンスコンテキストで評価されます。

A.2 データ型

式評価は、以下の第一級タイプを含みます：

- bool ブーリアン型 (真 / 偽)
- error エラー型
- float 浮動小数点型 (倍精度)
- int8 8 ビット符号付き整数
- uint8 8 ビット符号なし整数
- int16 16 ビット符号付き整数
- uint16 16 ビット符号なし整数
- int32 32 ビット符号付き整数
- uint32 32 ビット符号なし整数
- string 文字列型

A.2.1 ブーリアン型

有効な値は真と偽です。

A.2.2 エラー型

エラー型の値は、システムが自動的に生成する、もしくはエンドユーザーが生成することができます。これにより、評価式がカスタムエラーを生成する、標準的な方法が提供されます。

A.2.3 浮動小数点型

64 ビット倍精度の IEEE 形式の浮動小数点です。[+-] [0-9] [.0-9]*? [eE [+-]? [0-9]+]

有効な例 :1, 1., 1.0, -1e10, 1.1e-10

無効な例 :.2, e5

A.2.4 整数型

精度付きの、符号付き、および符号なし整数です。整数は、以下の 3 つの基底のいずれかを用いて表記できます：

- 16 進数 -- 0x で始まる
- 8 進数 -- 0 で始まる
- 10 進数 -- その他の数字列

符号なし整数リテラルは、接尾辞 u を用いて表記されます。

以下の表に、各型において有効な値を示します：

型	有効な値
INT8	-128 ～ 127
UINT8	0 ～ 255
INT16	-32,768 ～ 32,767
UINT16	0 ～ 65535
INT32	-2,147,483,648 ～ 2,147,483,647
UINT32	0 ～ 4294967295

A.2.5 文字列型

二重引用符 (") で囲まれる文字列です。内部では、UTF-16 符号化方式の、.NET 文字列として保存されます。現在では、パーサは ASCII 文字のみを認識します。文字列内でサポートされるエスケープシーケンスは、\\ および * のみです。

A.3 データ型の変換

この言語は Perl 5 に類似しているため、全てのデータ型は、一つの例外を除いて、他のすべての型に変換することができます。データ型変換の規則は、明瞭で予測可能です。

変換規則の例外は、エラー型です。全てのデータ型はエラー型に変換できますが、エラー型のみは他のいかなるデータ型に変換することができません。

A.3.1 ブーリアン型

変換先の型	規則
エラー型	一般的なエラーメッセージになります。
浮動小数点型	真は 1.0 になります。偽は 0.0 になります。
整数型	真は 1 になります。偽は 0 になります。
文字列型	真は "true" になります。偽は "false" になります。

A.3.2 エラー型

変換先の型	規則
ブーリアン型	無効
浮動小数点型	無効
整数型	無効
文字列型	無効

A.3.3 浮動小数点型

変換先の型	規則
ブーリアン型	浮動小数点の値が 0.0 の場合、偽です。それ以外の場合、真です。
エラー型	一般的なエラーメッセージになります。
整数型	小数点以下が切り捨てられ、残る整数部分になります。残る値が 32 ビット整数に変換できない場合、 0 に変換されます。
文字列型	浮動小数点を文字列表記に変換します。文字列の正確な表記は、自動的に定まります。

A.3.4 整数型

変換先の型	規則
ブーリアン型	整数の値が 0 の場合、偽です。それ以外の場合、真です。
エラー型	一般的なエラーメッセージになります。
浮動小数点型	最も値に近い、浮動小数点型の値になります。通常、単に ".0" が追加されます。
文字列型	文字列を、 10 進整数表記に変換します。

A.3.5 文字列型

文字列型は特殊なケースです。文字列型には **4** つのサブタイプがあります。これらは：

- ブーリアン型のような文字列：値が **"true"** もしくは **"false"** です。

- 浮動小数点型のような文字列：最初の空白以外の文字が、浮動小数点の値です。浮動小数点の部分に含まれない、以後の文字は無視されます。
- 整数型のような文字列：最初の空白以外の文字が、整数の値です。浮動小数点の部分に含まれない、以後の文字は無視されます。
- その他

A.3.5.1 ブーリアン型のような文字列

変換先の型	規則
ブーリアン型	値が "true" ならば真、"false" ならば偽です。
エラー型	文字列の内容をとる、エラーメッセージになります。
浮動小数点型	"true" ならば 1.0 になります。"false" ならば 0.0 になります。
整数型	"true" ならば 1 になります。"false" ならば 0 になります。

A.3.5.2 浮動小数点型のような文字列

変換先の型	規則
ブーリアン型	浮動小数点に含まれない部分が省略され、残る浮動小数点部分が本物の浮動小数点のように変換されます。
エラー型	文字列の内容をとる、エラーメッセージになります。
浮動小数点型	浮動小数点に含まれない部分が省略され、残る浮動小数点部分が本物の浮動小数点に変換されます。
整数型	浮動小数点に含まれない部分が省略され、残る浮動小数点部分が本物の浮動小数点のように変換されます。

A.3.5.3 整数型のような文字列

変換先の型	規則
ブーリアン型	浮動小数点に含まれない部分が省略され、残る浮動小数点部分が本物の整数のように変換されます。
エラー型	文字列の内容をとる、エラーメッセージになります。
浮動小数点型	浮動小数点に含まれない部分が省略され、残る浮動小数点部分が本物の整数に変換されます。
整数型	浮動小数点に含まれない部分が省略され、残る浮動小数点部分が本物の整数のように変換されます。

A.3.5.4 その他の文字列

変換先の型	規則
ブーリアン型	値が空文字列もしくは "0" の場合、偽です。それ以外の文字列は、真です。
エラー型	文字列の内容をとる、エラーメッセージになります。
浮動小数点型	0.0 になります。
整数型	0 になります。

A.4 演算子

式評価は、以下の演算子をサポートします。(優先順位が高いものから低いものへの順)

- 型変換
- ! 単項 + 単項 -
- * / %
- + - .
- < > <= >= lt gt le ge
- == != eq ne
- &&
- ||
- ?:

全ての演算子は、引数をよく定義された型に変換し、よく定義された型の結果を返します。

A.4.1 算術演算子 (+, -, *, /, %, 単項 +, 単項 -)

算術演算子は、以下の規則の順番に、引数を数値に変換します。

- いずれかの被演算子がエラー型の場合、演算の結果は最も左のエラーになります。
- いずれかの被演算子が浮動小数点型、浮動小数点型のような文字列の場合、両方の値が正しい型の浮動小数点に変換されます。
- それ以外の場合、両方の被演算子が整数に変換されます。

両方の被演算子が整数であり、少なくとも一つが符号なしの場合、演算は符号なしのものとして行われます。

算術演算子は常に、被演算子が上記の規則に従い変換された後、エラー値もしくは被演算子と同じ型の数値を返します。

A.4.2 数値比較演算子 (==, !=, <, >, <=, >=)

数値比較演算子は、算術演算子と完全に同じ規則に従って、引数を数値に変換します。数値比較演算子は、必ずブーリアン型の値を返します。

両方の被演算子が整数であり、少なくとも一つが符号なしの場合、演算は符号なしのものとして行われます。

A.4.3 文字列比較演算子 (eq, ne, lt, gt, le, ge)

文字列比較演算子は、いずれかの被演算子がエラー型の場合を除き、引数を文字列型に変換します。いずれかの被演算子がエラー型の場合、演算の結果は最も左のエラーになります。数値比較演算子は、必ずエラー値もしくは文字列型の値を返します。

A.4.4 文字列連結演算子 (.)

文字列連結演算子は、被演算子の変換においては、文字列比較演算子と同様にふるまいます。

注 文字列連結演算子の直前もしくは直後に数値が置かれた場合、浮動小数点として解釈されます (例: 1. = 浮動小数点; 1. = 文字列連結)。

A.4.5 三項演算子 (?:)

- ブール型をとるはずの、最初の被演算子がエラー型の場合、結果はエラー型です。
- ブール型の値が真の場合、結果の値および型は、? と : の間の式になります。
- ブール型の値が偽の場合、結果の値および型は、: の右側の式になります。

A.4.6 型変換

型変換 (小文字の `cast`) は、このシンタックスをとります : `cast(タイプ, 式)`

型変換は式を評価し、式の結果を与えられた型に変換します。

A. 5 文字列の補完

式評価は、文字列内の評価式を補完することができます。このシンタックスをとります :

``= expr``

``=` が、式の始点を示します。記号 `=` は、式の一部と解釈されません。次の ``` が式の終点を示すため、``=` ブロックを他の ``=` 内にネストすることはできません。複数の式を、ネストしていない限り同じ文字列に埋め込むことができます。これら全てが評価され、補完されます。埋め込まれた式は、一度のみ評価されます。評価された文字列が有効な式であった場合、これは再度評価されません。

始点の ``` から終点の ``` までの文字列は、``` から ``` までの式の結果に置き換えられます。

A. 6 ユーザー定義のデータ型 (列挙型)

式システムの観点によれば、ユーザー定義のデータ型は単なる整数です。ユーザー定義のデータ型は式に現れることが可能なものの、式内で評価される前に、対応する整数に変換されます。シンボルにユーザー定義の型を追加する方法については、[25 ページのユーザー定義型の追加](#)を参照してください。

B. データパス設定ツール



データパス設定ツールは、PSoC コンポーネントの Verilog 実装内の、データパスインスタンス設定を編集するのに用います。この付録には、ツールを使用して Verilog ファイルを修正するのに役立つ、手順と情報が記載されています。

Verilog を用いてコンポーネントを実装する際の情報については、[34 ページの Verilog を用いた実装](#)を参照してください。

B. 1 一般的機能

データパス設定ツールは、以下のときに利用してください：

- 既存の Verilog ファイルを読み込む
- 既存のデータパス設定を修正する
- 新規の設定を作成する
- 既存の設定を削除する

他の Verilog 実装に一切影響を与えずに、元の Verilog ファイルに変更を保存することができますこのツールは既存の Verilog ファイルに対し使用できますが、新たな Verilog 実装を作成することはできません。

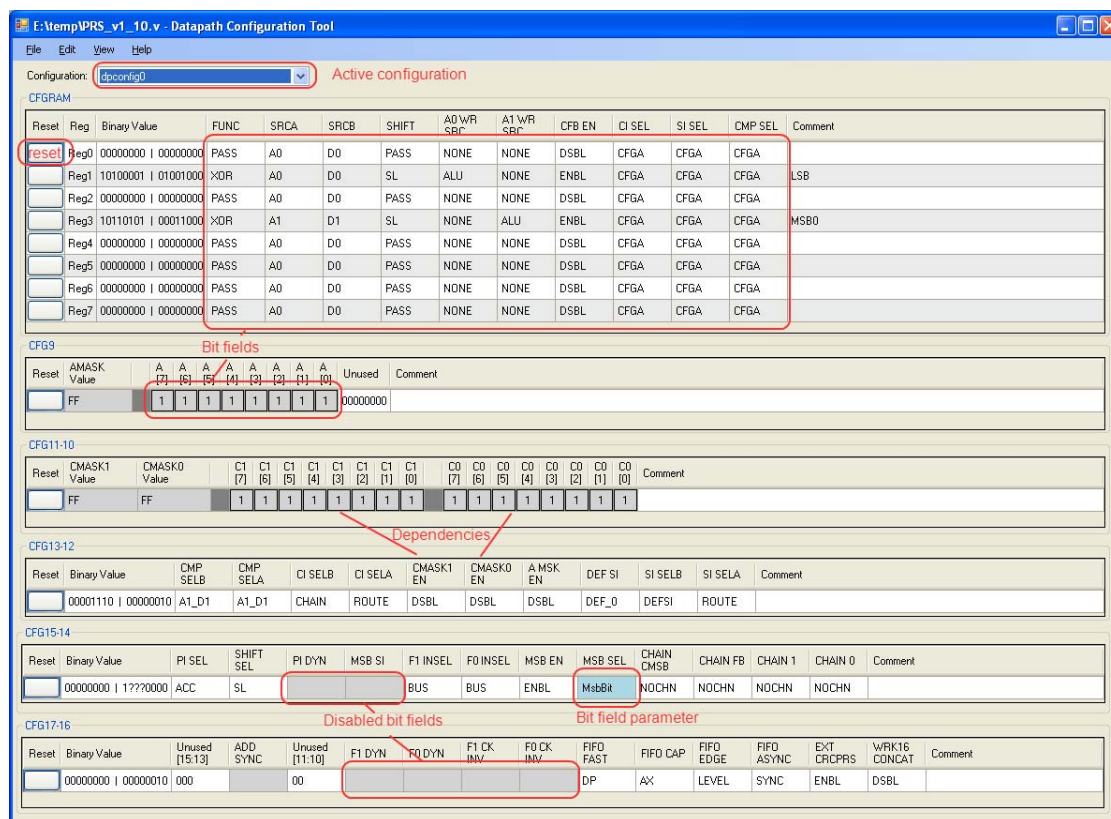
Verilog ファイルは開かれた後、存在するデータパス設定を探すためパースされます。これらは、パラメータとして、ローカルパラメータとして、またはデータパス内で直接名前付きパラメータとして、定義することができます。ツールは、発見できた設定を全て、アプリケーション上部にある設定ドロップダウンリストに表示します。

ビットフィールドの編集、コメントの編集、設定間もしくは設定内でのデータのコピーアンドペースト、データパス全体のコピーアンドペースト、および操作の作成または削除を行うことができます。

いくつかの操作は、データパスに対してのみ有効です（データパスの削除、初期レジスタ値の編集など）。アクティブデータパスは、選択された設定が所在するデータパスです。設定がパラメータとして定義された場合、アクティブデータパスは存在せず、全てのデータパスに関するオプションは無効になります。

B.2 フレームワーク

B.2.1 インターフェース



メインフォームは、以下の要素を含みます：

- **設定 ドロップダウンリスト** -- アクティブなVerilogファイルに存在した、全ての設定を含みます。現在選択されているものは、「アクティブ設定」もしくは「選択された設定」と呼ばれます。
- **設定レジスタ データグリッド(Data Grids)** -- レジスタ全体をデフォルト設定に戻す[レジスタリセット]ボタン、レジスタ用バイナリ値フィールド、ビットフィールドの設定コントロールおよびコメントフィールドを含みます。バイナリ値フィールドは読み取り専用です。ビットフィールドのコントロールは一般的に、既定の値のドロップダウンリストです。CFG9、CFG10、およびCFG11 レジスタはマスクレジスタであり、これらの値は個別のビットボタンをクリックすることでトグルすることができます。
- **ビットフィールド パラメータ** -- 各ビットフィールドは、定数値をパラメータ値で置き換えることができます。ビットフィールドがパラメータ値で定義されている場合、パラメータ名は青いバックグラウンドで表示されます。
- **未使用ビット** -- 未使用ビットは読み込み専用であり、常に値 0 で初期化されます。
- **無効ビットフィールド** -- アクティブ設定で使用されないビットです。ユーザーは、これらのビットのコンテキストメニューを使用して、有効化することができます。ビットは、特定のシリコンの版では無効である可能性があります。これらのビットフィールドは、灰色のバックグラウンドで、無効と表示され、フィールドに値が表示されません。常に値 0 で初期化されます。

B.2.2 メニュー

B.2.2.1 ファイル メニュー

メニューのアイテム	ショートカット	説明
開く	[Ctrl] + [O]	既存のファイルを開くためのダイアログを表示します。(.v Verilog ファイルのみフィルターして表示)
閉じる		アクティブファイルを閉じます。
最近使用したファイル		以前開いたファイルへのアクセスを提供します。
上書き保存	[Ctrl] + [S]	アクティブ文書を保存します。
名前を付けて保存		アクティブ文書を別のファイルに保存するためのダイアログを表示します。
閉じる		PSoC データパス設定ツールを閉じます。

B.2.2.2 編集 メニュー

メニューのアイテム	ショートカット	説明
データパスのコピー	[Ctrl] + [C]	選択されたデータパス設定をコピーします。
データパスの貼り付け	[Ctrl] + [V]	クリップボードのフルデータパス設定を、アクティブデータパスに貼り付けます。
ダイナミック設定の張り付け		クリップボードの、データパス設定の、フルダイナミック設定情報を、アクティブ設定に貼り付けます。ダイナミック設定の 8 レジスタ全てが張り付けられます。
データパスのリセット		現在のデータパス設定を、デフォルトのビットフィールド値にリセットします。
新しいデータパス		新しいデータパス インスタンスを追加します。
データパスの削除		データパス インスタンスを削除します。

B.2.2.3 閲覧 メニュー

メニューのアイテム	ショートカット	説明
初期レジスタ値		アクティブデータパスの初期レジスタ値を編集するダイアログを表示します。

B.2.2.4 ヘルプ メニュー

メニューのアイテム	ショートカット	説明
ドキュメント	[F1]	この文書を開きます。
バージョン情報		ビルド情報を提供する、バージョン情報を開きます。

B.3 一般的なタスク

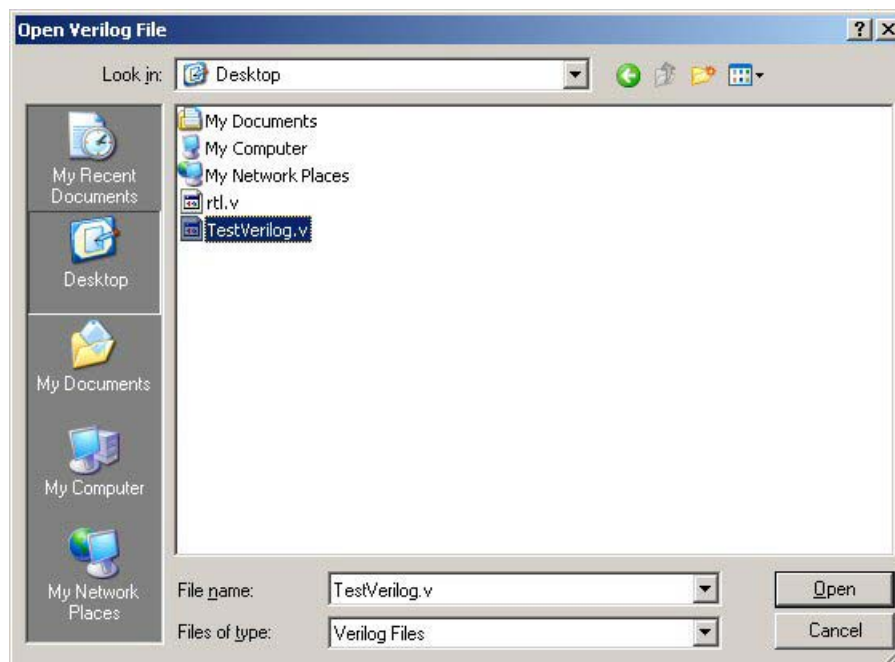
B.3.1 データパス設定ツールを起動する

ツールを起動するには、スタートメニューから、**Cypress > PSoC Creator 1.0 > Component Development Kit > Datapath Configuration Tool** を選択します。

アプリケーションの起動時には、データパス設定情報は保持されません。ツールを使用するためには、まず既存の **Verilog** ファイルを開く必要があります。

B.3.2 Verilog ファイルを開く

1. Verilog ファイルを開くには、**Open...** をファイルメニューから選択します。
[Open Verilog File] ダイアログが表示されます。



- 注：**最近使用したファイルをこのダイアログを使用せずに開くには、**最近使用したファイル** をファイルメニューから選択する方法があります。また、**Verilog** ファイルをアプリケーションのメインフォームにドラッグすることで開くこともできます。
2. アップデートするファイルを移動して選択し、**開く** をクリックします。
ダイアログが閉じ、ツールが **Verilog** から必要な情報を全て集め、ウィンドウの残りをアップデートします。

B.3.3 ファイルを保存する

アクティブファイルを保存するには、上書き**保存** をファイルメニューから選択します。

別の名前、もしくは別のディレクトリにあるファイルとして保存するには、**名前を付けて保存** をファイルメニューから選択します。

B.4 ビットフィールド型パラメータを使用する

ビットフィールドは、定義済みの決められたいくつかの定数フィールドに加えて、パラメータ値をサポートします。

B.4.1 列挙されたビットフィールドにパラメータを追加する

列挙されたビットフィールドは、**Add Parameter** コンテキストメニューアイテムを含みます。

Reset	Reg	Binary Value	FUNC	SRC A	SRC B	SHIFT	A0 W/R SRC	A1 W/R SRC	CFB EN	CI SEL	SI SEL	DMP SEL	Comment
☐	Reg0	000000?? 01000000	PASS	A0	D0	dynShiftDir	ALU	NONE	DSBL	CFGA	CFGA	CFGA	Shift
☐	Reg1	000000?? 11000000	PASS	A0	D0	dynShiftDir	F0	NONE	DSBL	CFGA	CFGA	CFGA	Shift and Load New FIFO Data
☐	Reg2	00000000 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA	
☐	Reg3	00000000 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA	
☐	Reg4	00000000 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA	
☐	Reg5	00000000 00000000	PASS	A0	D0	PASS			SBL	CFGA	CFGA	CFGA	
☐	Reg6	00000000 00000000	PASS	A0	D0	PASS			SBL	CFGA	CFGA	CFGA	
☐	Reg7	00000000 00000000	PASS	A0	D0	PASS			SBL	CFGA	CFGA	CFGA	

このコマンドは、ビットフィールドに対し、任意のパラメータ名を選択しとして追加することを可能にします。パラメータの追加後、ビットフィールドのドロップダウンメニューは選択肢として、定義済みの定数以外に、追加されたパラメータ値を含みます。パラメータ値が選択された場合、これは定義済みの値でないことを示すため、青色で表示されます。

パラメータ名を変更するには、ビットフィールドを右クリックし、コンテキストメニューから **Edit Parameter** を選んでください。

CFGRAM													
Reset	Reg	Binary Value	FUNC	SRCA	SRCB	SHIFT	A0 W/R SRC	A1 W/R SRC	CFB EN	CI SEL	SI SEL	CMP SEL	Comment
☐	Reg0	000000?? 01000000	PASS	A0	D0	dynShiftDir	ALU	NONE	DSBL	CFGA	CFGA	CFGA	Shift
☐	Reg1	000000?? 11000000	PASS	A0	D0	dynShiftDir	F0	NONE	DSBL	CFGA	CFGA	CFGA	Shift and Load New FIFO Data
☐	Reg2	00000000 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA	
☐	Reg3	00000000 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA	
☐	Reg4	000000?? 00000000	PASS	A0	D0	dynShift			DSBL	CFGA	CFGA	CFGA	
☐	Reg5	00000000 00000000	PASS	A0	D0	PASS			DSBL	CFGA	CFGA	CFGA	
☐	Reg6	00000000 00000000	PASS	A0	D0	PASS			DSBL	CFGA	CFGA	CFGA	
☐	Reg7	00000000 00000000	PASS	A0	D0	PASS			DSBL	CFGA	CFGA	CFGA	

B.4.2 マスク ビットフィールドにパラメータを追加する

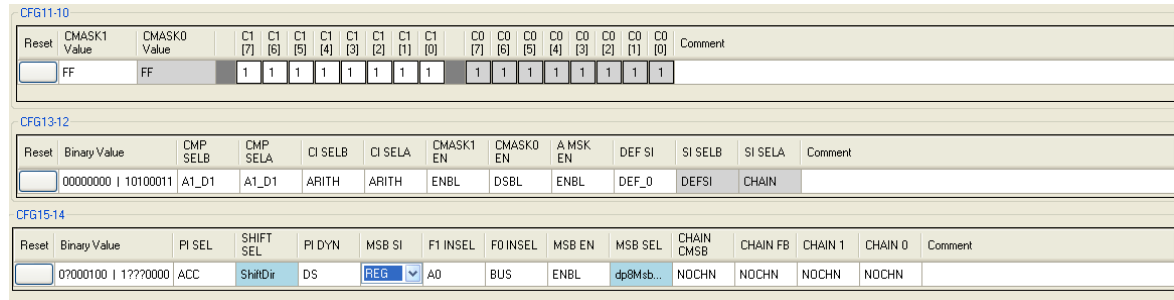
マスク ビットフィールドにパラメータを追加するには、**Add Parameter** をコンテキストメニューの値列から選択し、パラメータ名を入力します。パラメータの追加後、異なるビットは読み込み専用になります。

[illegible]

パラメータを削除するには、**Remove Parameter** をコンテキストメニューの値 列から選択します。

B.4.3 ビットフィールドの依存性

データパス設定には、他のビットに影響を与える、いくつかのビットがあります。これらの値によっては、他のビットが使用されなくなる可能性があります。ユーザーが見て判断できるようにするために、いくつかのビットフィールドは、他のフィールドの値によっては無効になります。



CFG11-10

Reset	CMASK1 Value	CMASK0 Value	C1 [7]	C1 [6]	C1 [5]	C1 [4]	C1 [3]	C1 [2]	C1 [1]	C1 [0]	C0 [7]	C0 [6]	C0 [5]	C0 [4]	C0 [3]	C0 [2]	C0 [1]	C0 [0]	Comment
	FF	FF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

CFG13-12

Reset	Binary Value	CMP SELB	CMP SELA	CI SELB	CI SELA	CMASK1 EN	CMASK0 EN	A MSK EN	DEF SI	SI SELB	SI SELA	Comment
	00000000 10100011	A1_D1	A1_D1	ARITH	ARITH	ENBL	DSBL	ENBL	DEF_0	DEFSI	CHAIN	

CFG15-14

Reset	Binary Value	PI SEL	SHIFT SEL	PI DYN	MSB SI	F1 INSEL	F0 INSEL	MSB EN	MSB SEL	CHAIN CMSB	CHAIN FB	CHAIN 1	CHAIN 0	Comment
	07000100 17770000	ACC	ShlDr	DS	REG	A0	BUS	ENBL	dpMSb...	NOCHN	NOCHN	NOCHN	NOCHN	

上のスクリーンショットでは、CMASK0、SI SELB、および SI SELA ビットが無効化されています。CMASK0 EN ビットが DSBL に設定され、MSB SI ビットが REG に設定されているからです。

B.5 設定を行う

データパスインスタンスの要素を設定するためには、2つの方法があります。

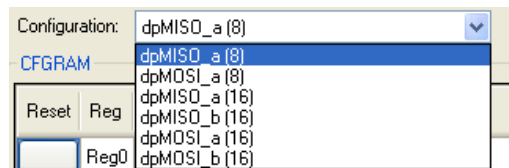
第一の方法では、データパスインスタンスに直接データパスの設定を入力します。複数バイトのデータパスインスタンスについては、インスタンス全体の各バイトにつき、異なる値の組み合わせが用いられます。これは、Verilog ファイル内で 1 度しか用いられない設定に対し、よく用いられます。

第二の方法では、データパスの設定をパラメータに割り当てます。データパスインスタンスは、このパラメータを使用します。この方法は、重複を防ぎコンポーネントの管理を簡単にするため、同じ設定が複数のインスタンスにおいて用いられるときに便利です。この方法は、Counter ライブラリコンポーネントにより使用されます。一つの Verilog ファイル内で両方の方法を併用することが可能です。

両方の方法とも PSoC データパス設定ツールによりサポートされています。全ての設定は、**Configuration** ドロップダウンリストに追加されます。

B.5.1 設定の名づけ

データパスインスタンスの場合、ラベルはデータパスインスタンス名です。dp8、dp16、dp24、および dp32 データパスインスタンスの場合、設定ラベルには、設定がインスタンスのどのバイトに効くかによって、"_a"、"_b"、"_c"、および "_d" のいずれかの接尾辞がつけます。また、データパスサイズ (8、16、24、または 32) が、ドロップダウンの設定名の終わりに追加されます。パラメータインスタンスの場合、ラベルはパラメータ名です。



Configuration: dpMISO_a (8)

CFGGRAM

Reset Reg

Reg0

B.5.2 設定の編集

設定を有効にするには、**Configuration** ドロップダウンリストにて選択してください。各レジスタのビットフィールドは、セルのドロップダウンリストの既定値のいずれかを選択するか、パラメータを追加することで変更できます。アクティブ設定に行われたすべての変更は、アプリケーションのオブジェクトストレージに自動的に保存されるため、ユーザーはデータを失うことなく別の設定に変更することができます。データは、メインメニューの [上書き保存] もしくは [名前を付けて保存] 操作を行わない限り、**Verilog** ファイルに保存されません。

B.5.3 コピーとペーストの設定

アクティブ設定のデータをクリップボードにコピーするには、**コピー**を**編集**メニューから選択してください。

設定データをクリップボードから貼り付けるには、**貼り付け**を**編集**メニューから選択してください。このコマンドは、アクティブ設定の全てのビットフィールドとコメントを、クリップボードの値で置き換えます。

ダイナミック レジスタデータのみを貼り付ける場合は、**ダイナミック設定の張り付け**を**編集**メニューから選択してください。

注：ダイナミック レジスタデータのみを貼り付ける場合、最初にコピーを行う時に**コピー**コマンドを使用してください。

異なるレジスタをコピーする場合、そのレジスタに所属する任意のビットフィールドのコンテキストメニューから**レジスタのコピー**を選択してください。

異なるレジスタを貼り付ける場合、そのレジスタに所属する任意のビットフィールドのコンテキストメニューから**レジスタの貼り付け**を選択してください。

B.5.4 設定のリセット

現在のデータパス設定をデフォルトのビットフィールド値にリセットするには、**Reset Datapath**を**Edit**メニューから選択してください。

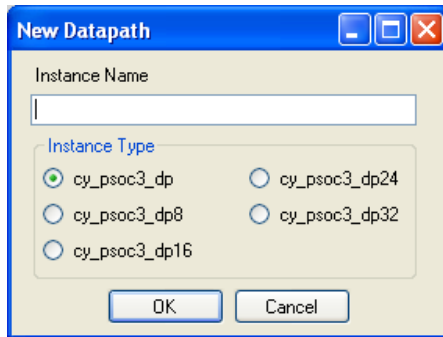
データパス設定の特定のレジスタをデフォルト値にリセットするには、メインフォームの該当する行のデータグリッドに置いて、**リセット**列のボタンを押してください。

マスクフィールドの **CFG9-11** を除くすべてのビットフィールドのデフォルト値は、**0** です。マスクフィールドの **CFG9-11** のデフォルト値は、**1** です。ダイナミックレジスタのコメント値のデフォルトは**アイドル (Idle)** であり、他のレジスタのデフォルトは空文字列です。

B.6 データパス インスタンスを使用する

B.6.1 新しいデータパス インスタンスを作成する

新しいデータパス インスタンスを作成するには、**New Datapath** を **Edit** メニューから選択してください。 **New Datapath** ダイアログが表示されます。



新しいデータパスの名前を **Instance Name** テキストボックスに入力し、データパス型を選択し、**OK** をクリックしてください。

新しいデータパスが、現在の **Verilog** ファイルの終わりに、**endmodule** の前に挿入されます。

新しいデータパスの設定は、全てのビットフィールドがデフォルト値で初期化され、全ての入力信号は **1'b0** に接続され、すべての出力信号は接続されていない状態になります。

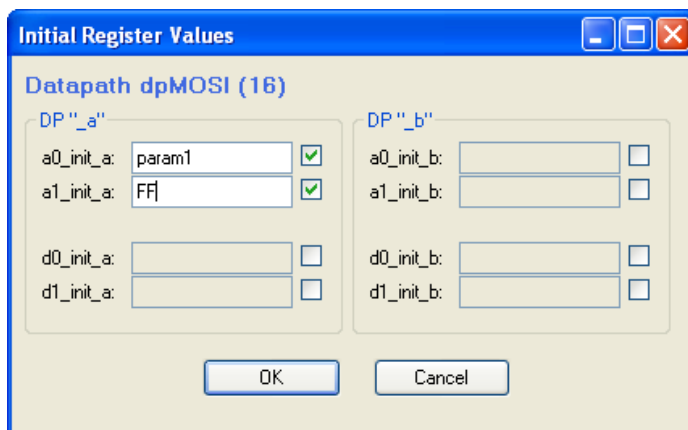
B.6.2 データパス インスタンスを削除する

アクティブデータパスを削除するには、**Delete Datapath** を **Edit** メニューから選択してください。

アクティブデータパスが存在しない場合、メニューにおいて、このオプションは無効です。データパスに複数の設定があった場合 (**dp16**、**dp24**、もしくは **dp32**)、全て削除されます。

B.6.3 初期レジスタ値を設定する

アクティブデータパスの **Initial Register Values** を設定するには、**Initial Register Values** を **View** メニューから選択してください。 **Initial Register Values** ダイアログが表示されます。



定義されるエントリの近くのボックスをチェックし、8 ビットの 16 進数値 (8'h?) もしくはパラメータ値を編集フィールドに入力してください。変更点を適用するには、**[OK]** をクリックしてください。初期値はデータパスに名前付きパラメータとして渡されるため、定義されていない値はデータパスに渡されず、内部でデフォルト値に設定されます。

アクティブデータパスが存在しない場合、メニューにおいて、このオプションは無効です。