デュアル CapSense® シグマデルタデータシート CSD2x V 2.40

Copyright © 2011-2012 Cypress Semiconductor Corporation. All Rights Reserved.

リソース	PSoC® ブロック				API メモリ (バイト)		ピン (センサとは別)
	CapSense®	I²C/SPI	タイマ	コンパレータ	フラッシュ	RAM	
CY8C21x45, CY8C22x45							
シングルチャネル、IDAC 付き *	1	-	-	1	1006	29	1
オートキャリブレーション有効	1	-	-	1	1198	35	1
ダブルチャネル、IDAC 付き *	2	-	-	2	1390	30	2
オートキャリブレーション有効	2	-	-	2	1576	36	2
シングルチャネル、Rb 抵抗付き *	1	-	-	1	1000	28	2
ダブルチャネル、Rb 抵抗付き *	2	-	-	2	1400	30	4
CapSense ボタン追加	-	-	-	-	8	12	1
5 つのセンサ素子を使った静電容量式スライダを使用する場合のコード容量と RAM 使用エリアの増分。	-	-	-	-	613	90	5
追加のスライダ素子	-	-	-	-	8	12	1
スライダ ダイプレックスを使用する場合の、コード容量と RAM 使用エリアの増分 (センサ 5 個)。	-	-	-	-	10	-	-

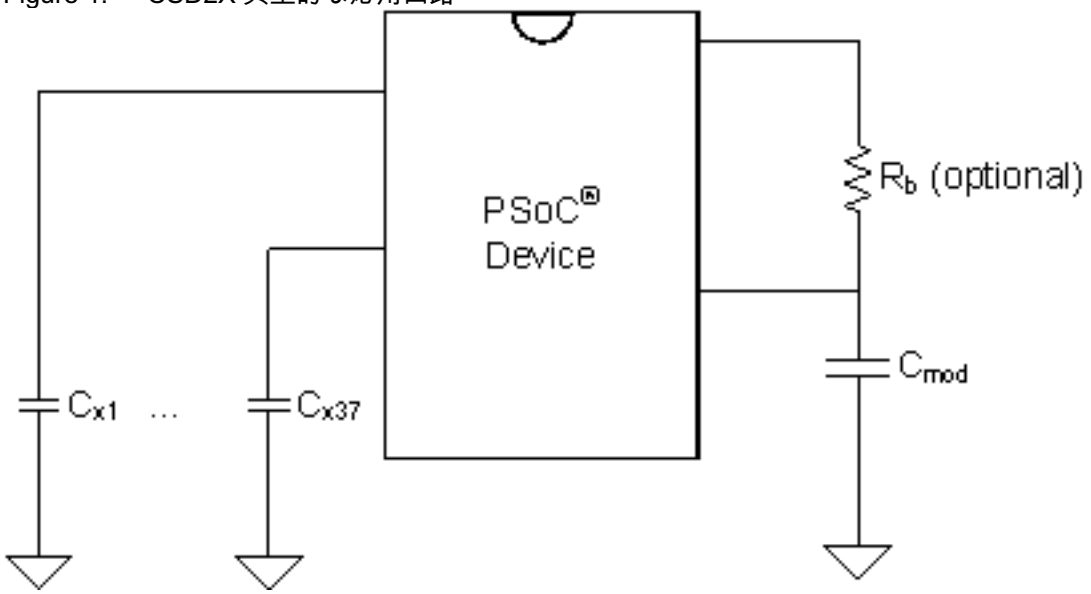
* 1 つボタンの構成を想定して計算

特長および概要

- 1 ~ 37 の静電容量式センサをスキャン
- ガラスのオーバーレイが最大 15 mm まで検知可能
- ワイヤベースセンサでは 20 cm の近接検知
- AC 電源ノイズ、EMC ノイズ、電源電圧変化に対する高耐性
- 異なる独立· スライド式静電容量式センサの組み合わせをサポート
- ダイプレックスを使用してスライドセンサの物理的分解能を倍加
- 補間法を使用してスライドセンサの分解能を向上
- 2 つのスライドセンサを使ったタッチパッド サポート
- 高抵抗伝導性材料を使った検知サポート (ITO フィルムなど)
- 水膜や水滴がある場合にも信頼できる動作をするシールド電極サポート
- CSD2x ウィザードを使用した誘導センサとピンの割り当て
- 温度、湿度、静電気放電 (ESD) に対応する、実績のある基底値更新アルゴリズム
- 調整が簡単な操作パラメータ
- Raw データ値のモニタリングとリアルタイムのパラメータ最適化のための PC GUI アプリケーション サポート

CSD2x (デルタ シグマ変調器を使用した静電容量式検知) は、スイッチトキャパシタ電流をデジタル値に変換するデルタシグマ変調器を使用した、スイッチトキャパシタ手法による静電容量式検知を行います。CSD2x ユーザ モジュールは、シングルチャネルの CapSense スキャニングとデュアルチャネルの CapSense スキャニングをサポートします。

Figure 1. CSD2X 典型的な応用回路



クイック スタート

1. 専用ピンを必要とするユーザモジュールを選択・配置します (例 : I2C と LCD)。ポートとピンを適宜割り当てます。
2. CSD2x ユーザ モジュールを選択・配置します。
3. Workspace Explorer で CSD2x ユーザ モジュールを右クリックし、CSD2x ウィザードを開きます (ウィザードについては後述します)。
4. センサ数、スライダ数、または回転式スライダ数を設定します。
5. 各センサの設定を指定します。
6. ピンとグローバルパラメータを設定します。パラメータの説明を全部読み、要件とガイドラインに従ってください。
7. アプリケーションを生成し、Application Editor を開きます。
8. 個別のセンサ、スライド式センサ、またはタッチパッドを実装するために必要なサンプルコードを使用します。
9. I2C-USB ブリッジをターゲットボードに接続し、信号を観察します。
10. CSD2x パラメータを変更して設定を最適化し、アプリケーションを再構築します。
11. PSoC デバイスをプログラムし、モジュール動作を確認します。AN2403 の Signal-to-Noise Ratio Requirements for CapSense Applications で説明してある 5:1 SNR の要件を実現するために CSD2x パラメータを調整してください。

問題がある場合は、「付録」の「トラブルシューティング」を参照してください。

機能説明

静電容量式センサは、次の物理、電気、ソフトウェア コンポーネントから成ります。

- 物理的コンポーネント : 物理的なセンサそのもの、通常、PCB 上で PSoC に接続された導電性のパターンとそれを覆う絶縁性のカバー、薄膜やディスプレイ上の透明なオーバーレイ。
- 電気的コンポーネント : センサのキャパシタンスをデジタル形式に変換する部分。変換システムは、検知を担うスイッチト キャパシタ、デルタシグマ変調器、変調器の出力ビット ストリームを読み取り可能なデジタル形式に変換する、カウンタベースのデジタル フィルタから構成されます。
- ソフトウェア :
  - 検出および補正ソフトウェア アルゴリズムが、カウント値をセンサの検出結果に変換します。
  - 連続的・組み上げセンサタイプの場合 (スライダやタッチパッドなど)、提供されている API の保管技術により、センサの物理的ピッチよりも高い分解能の位置を提供します。例えば、10 個のセンサを用いて音量スライダを構築し、一定のファームウェアを用いて音量レベル数を 100 まで拡大することができます。また、同じ API を使うと、途中から連結しあう 2 つの静電容量式センサを利用して、その間にある伝導性物体 (指など) の位置を特定することもできます。

静電容量を測定する方法は多数ありますが、このユーザ モジュールで使用されている方法は、スイッチト キャパシタとシグマデルタ変調器を組み合わせるものです。

センサのアレイとは、個別のセンサ、スライダ式センサ、スライダ式センサを直交に組み合わせたタッチパッドが含まれます。高レベルの判定理論が、温度、湿度、電源電圧の変化などの環境要因に追従して補正します。別個のシールド電極を使って、センサアレイをシールドして浮遊静電容量を低減します。これにより、水膜や水滴がある場合でも動作の信頼性が確保できます。

高レベルのソフトウェア機能によってスライダダイプレックスをサポートするため、1 つの電気センサを 2 つの物理的位置に使用して高分解能を得ることができます。また、物理的なセンサの位置の間で判定されたセンサ位置をさらに補間する機能もあります。

初めて CSD2x ユーザ モジュールを使用する際には、次の文書を読むことをお勧めします。

「CY8C22X45CY8C21345 PSoC プログラマブル システムオンチップの技術レファレンス マニュアル」のセクション : CapSense システム

次のアプリケーションノートは、CSD2x ユーザ モジュールのマニュアルを読んだあとに読むよう推奨されています。アプリケーション ノートは、サイプレス セミコンダクタのウェブサイト (www.cypress.com) からご覧いただけます。

- CapSense のベスト プラクティス – AN2394
- CapSense アプリケーションでの信号対雑音比要件 – AN2403
- CapSense アプリケーションをデバッグするチャート作成ツール – AN2397
- PSoC CapSense アプリケーションの EMC 設計における考慮点 – AN2318
- 容量検知アプリケーションの消費電力とスリープ機能 – AN2360
- PSoC CapSense のレイアウト ガイドライン – AN2292
- ユニバーサル非同期トランスミッタのソフトウェア実装 – AN2399
- 耐水静電容量検知 – AN2398

静電容量測定の操作

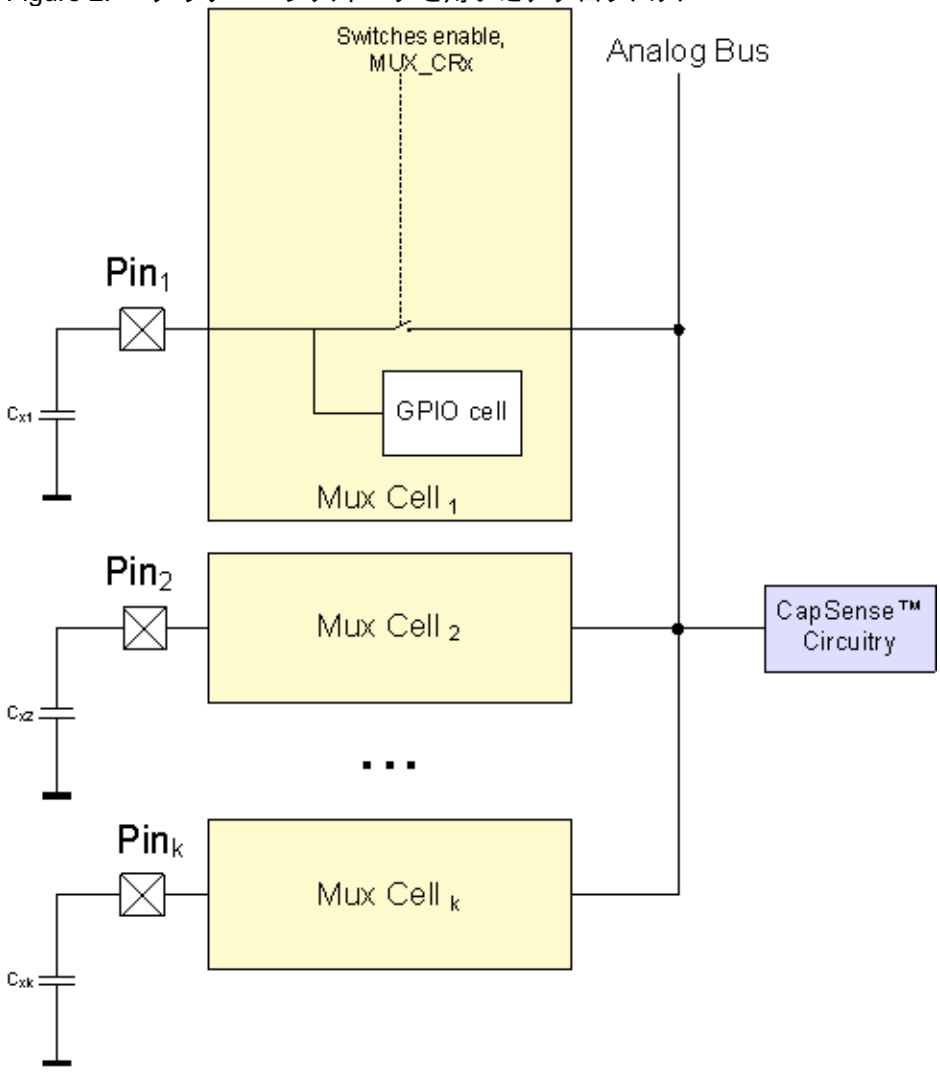
判定理論はファームウェアで実行されます。ファームウェアは、静電容量を分析し、環境要因による静電容量の変化をトラッキングし、判定理論を実行することにより、ボタンのタッチを検知し、スライダの位置を計算します。

センサのアレイをスキャンする

CY8C22x45 デバイスファミリーはアナログバスの中に構成され、どの PSoC ピンも静電容量式センサへの接続が可能です。CSD2x ユーザ モジュールは、内部プリチャージスイッチを使用して、クロック信号フェーズ Ph_1 で動作中のセンサを充電し、アナログバスをフェーズ Ph_2 でセンサと接続します。Sigma-Delta (シグマデルタ) 変調器の変調コンデンサとコンパレータの入力は、アナログバスに恒久的に接続されています。

ファームウェアは、MUX_CRx レジスタで該当するビットを設定して、センサのスキャンを実行します。

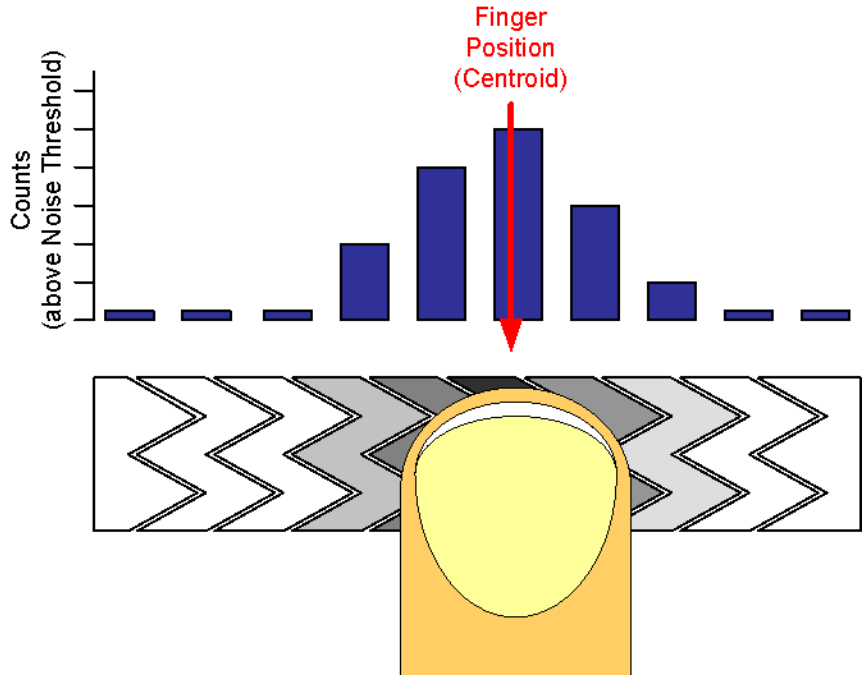
Figure 2.　プリチャージスイッチを用いたアナログバス



スライダ

スライダは、段階的調整を必要とする制御に使用します。例としては、照明管理（調光装置）、音量管理、グラフィックイコライザ、速度管理などが挙げられます。これらのセンサは機械的に互いに隣接しており、1 つのセンサの作動は、物理的に近接するセンサの部分的な作動につながります。スライダの実際の位置は、作動したセンサ セットのセントロイド位置を計算することによって判断できます。スライダは、CSD2x ウィザードで、各スライダグループがそれぞれ特定の順序を持つように設定されます。センサ スライダの実質的な下限は 5 で、上限は、選択した PSoC デバイスで利用できるセンサ数になります。

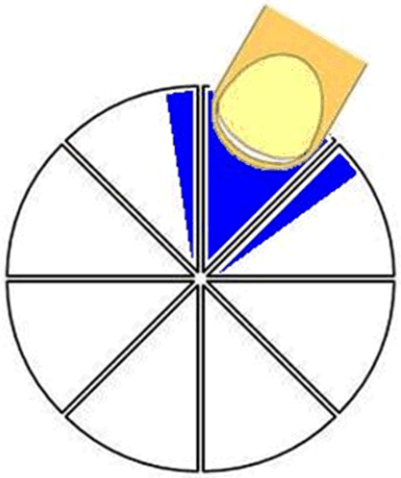
Figure 3.　物理的センサ位置の順序



スライダの半分で強い信号を近接検知すると、残り半分に同レベルの信号を生じますが、結果は分散してしまいます。検知アルゴリズムは、強い近隣信号セットを検索して、スライダ位置を特定します。

ラジアル スライダ

Figure 4.　指がラジアル スライダに触れる



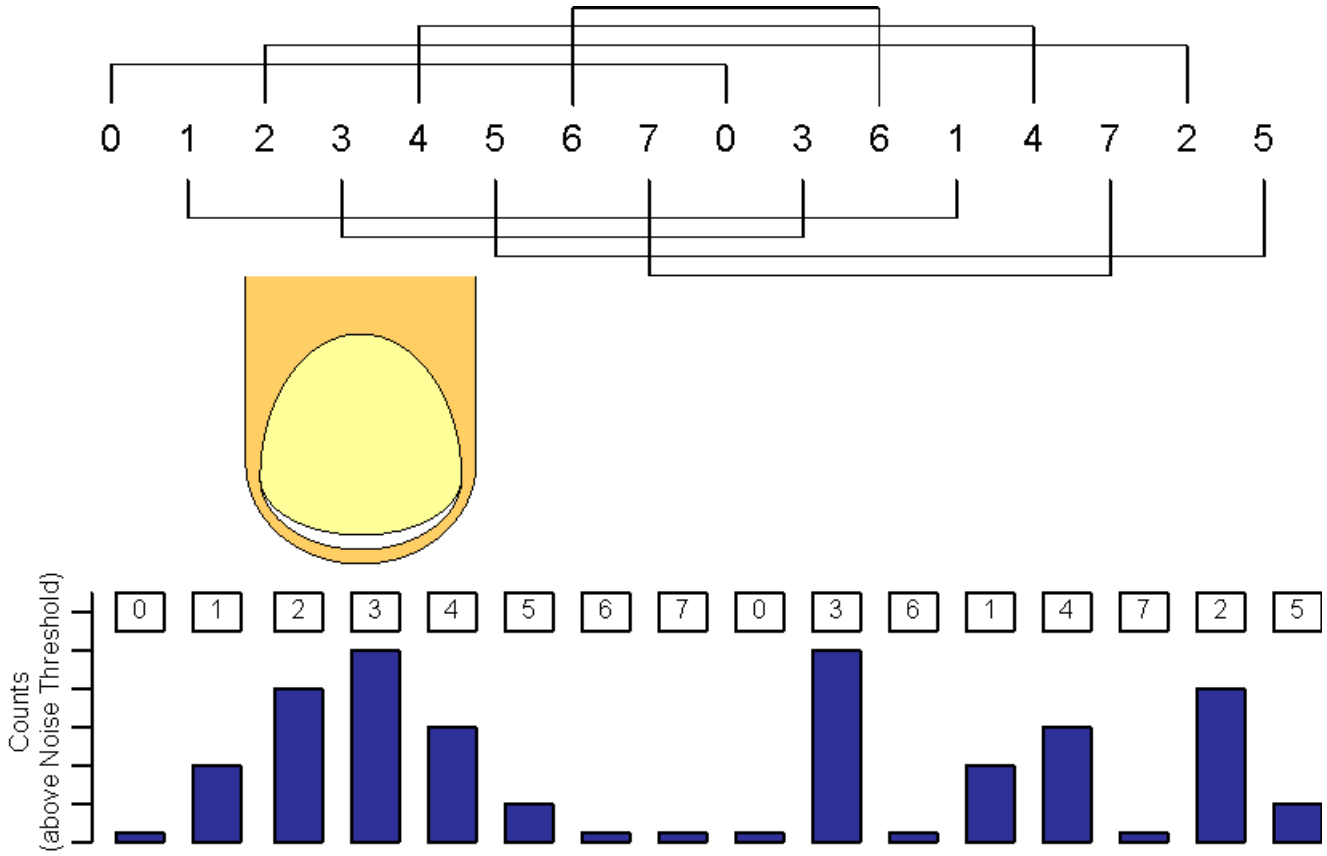
CSD2X ユーザモジュールでは、リニアおよびラジアルという 2 種類のスライダを利用できます。ラジアル スライダは、リニア スライダと似ています。リニア スライダには始点と終点がありますが、ラジアル スライダにはありません。スライダに触れると、セントロイド計算アルゴリズムが、スイッチのセンサ数を現在のスイッチの左右で考慮します。ラジアル スライダにダイプレックスは対応していません

CSD2X ユーザモジュールには、ラジアル スライダをサポートする 2 つの API 関数があります。最初の関数 CSD2X_wGetRadiaPos() はセントロイド位置を返し、2 つ目の関数 CSD2X_wGetRadialInc() は分解能単位で指のシフトを返します。指が時計回り方向に移動すると、正のオフセットとなります。

ダイプレックス

スライダセンサの各センサは、PSoC の各ピンに 2 つづつ接続されます。スライダセンサの最初の半分 (数字が小さい) は、CSD2x ウイザードでデザイナが割り当てたピンを各センサに連続的にマッピングします。残りの半分 (数字が大きい) のスライダセンサは、ウィザードのアルゴリズムによって自動的にマッピングされ、取り込みファイルに一覧表示されます。この順序は、半分内における近隣センサ起動が別の半分の近隣センサ起動を引き起こさないように設定されます。この順序の決定と、PCB 基板へのマッピングは慎重に行ってください。

物理センサ位置の後の半分に対して順序を決める方法は数多くあります。最も簡単な方法は、上半分のセンサのうち、偶数センサ全部を先に決定し、次に奇数センサ全部を決定するやり方です。他の方法では、別のインデックス値を使ってセンサを配置します。。このユーザ モジュールで選択された方法は 3 によるインデックス化です。



スライダ中のセンサ静電容量は均衡がとれていなければなりません。センサや PCB レイアウトによって、一部のセンサペアではセンサ配線が長くなる可能性があります。ダイプレックスセンサ インデックス表は、ダイプレックスを選択すると CSD2x ウィザードによって自動的に作成されます。以下の表に、異なるスライダ セグメント カウントのダイプレックス シーケンスを示します。

Table 1. 異なるスライダ セグメント カウントのダイプレックス シーケンス

スライダ セグメントの総カウント	セグメント シーケンス
10	0,1,2,3,4,0,3,1,4,2
12	0,1,2,3,4,5,0,3,1,4,2,5
14	0,1,2,3,4,5,6,0,3,6,1,4,2,5
16	0,1,2,3,4,5,6,7,0,3,6,1,4,7,2,5
18	0,1,2,3,4,5,6,7,8,0,3,6,1,4,7,2,5,8
20	0,1,2,3,4,5,6,7,8,9,0,3,6,9,1,4,7,2,5,8
22	0,1,2,3,4,5,6,7,8,9,10,0,3,6,9,1,4,7,10,2,5,8
24	0,1,2,3,4,5,6,7,8,9,10,11,0,3,6,9,1,4,7,10,2,5,8,11
26	0,1,2,3,4,5,6,7,8,9,10,11,12,0,3,6,9,12,1,4,7,10,2,5,8,11

スライダ セグメントの総カウント	セグメント シーケンス
28	0,1,2,3,4,5,6,7,8,9,10,11,12,13,0,3,6,9,12,1,4,7,10,13,2,5,8,11
30	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,0,3,6,9,12,1,4,7,10,13,2,5,8,11,14
32	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0,3,6,9,12,15,1,4,7,10,13,2,5,8,11,14
34	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,0,3,6,9,12,15,1,4,7,10,13,16,2,5,8,11,14
36	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,0,3,6,9,12,15,1,4,7,10,13,16,2,5,8,11,14,17
38	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,0,3,6,9,12,15,18,1,4,7,10,13,16,2,5,8,11,14,17
40	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,0,3,6,9,12,15,18,1,4,7,10,13,16,19,2,5,8,11,14,17
42	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,0,3,6,9,12,15,18,1,4,7,10,13,16,19,2,5,8,11,14,17,20
44	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,2,5,8,11,14,17,20
46	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20
48	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20,23
50	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20,23
52	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23
54	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23,26
56	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,0,3,6,9,12,15,18,21,24,27,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23,26

補間とスケーリング

多くの場合、スライド式センサとタッチパッド用のアプリケーションでは、個々のセンサのネイティブピッチよりも高い分解能を得られるように指 (またはその他の静電容量性物体) の位置を特定する必要があります。スライド式センサやタッチパッドで指が触れる工リアは、しばしば 1 個のセンサより大きくなっています。

セントロイド法を使用した補間位置の計算では、まずアレイをスキャンして、センサの位置が有効であることを確認します。ここでは、近隣センサ信号のある番号がノイズ閾値を超えていることが要件となります。最も強い信号が見つかったら、その信号と、ノイズ閾値より大きい近隣信号を使用してセントロイドを算出します。セントロイドは (通常)、2 つから 8 つのセンサを使用して、次の数式で計算されます。

Equation 1

$$N_{Cent} = \frac{n_{i-1}(i-1) + n_i i + n_{i+1}(i+1)}{n_{i-1} + n_i + n_{i+1}}$$

通常、計算結果は整数ではありません。たとえば 12 個のセンサに対して 0 ~ 100 という範囲である場合、セントロイドを特定の分解能の形で得るには、計算されたスカラー量をセントロイドに掛けます。1 つの計算で補間とスケーリングを組み合わせ、その結果を直接、希望のスケールで報告する方が効率的です。これは高レベル API で行う処理です。

スライダセンサ の数と分解能は、CSD2x ウィザードで設定します。スケーリング値は、ウィザードで計算され、整数ではない値として保存されます。

セントロイドの分解能の乗数は、3 バイトに含まれ、それぞれのビット定義は以下になります。

分解能乗数 MSB								
ビット	7	6	5	4	3	2	1	0
乗数	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
分解能乗数 ISB								
乗数	128	64	32	18	16	8	4	2
分解能乗数 LSB								
乗数	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256

分解能はこの数式を用いて算出されます。

分解能 = (センサ数 – 1) x 乗数

セントロイドは 24-bit 符号無し整数で保持され、その分解能はセンサ数と乗数の関数です。

フィードバックコンポーネント選択のガイドライン

CSD2x ユーザ モジュールには、外付け変調コンデンサが必要で、オプションのシールド電極もサポートします。このセクションでは、外付けコンポーネントを選択する方法を説明します。

変調コンデンサ

ユーザ モジュールには、外付け変調コンデンサ C_{mod} とモジュレータ フィードバック 抵抗 R_b が必要です。コンデンサは、P0[7] ポート ピンと Vss グラウンドに接続できます。フィードバック 抵抗 R_b は、ポートピン P0[5] とコンデンサピンに接続することができます。ピンは、ユーザ モジュールのパラメータ設定により選択されます。変調器コンポーネントの接続用に選択されているピンは、これ以外の目的で使用しないでください。

変調コンデンサ用に推奨されている値は、4.7 ~ 47 nF です。最適な静電容量は、最大 S/N 比 を得るための実験を行うことで決定することができます。PRS16 および PRS8 構成ではほとんどの場合、5.6 nF ~ 10 nF の値では良い結果が得られます。プリスケーラ付きの構成を選択した場合、統合コンデンサの推奨値は 22 nF ~ 47 nF です。フィードバック 抵抗を選択した後で、最良の S/N 比を得るためには、いくつかのコンデンサ値で実験してみるとよいでしょう。セラミックのコンデンサを使用するよう強く推奨します。温度静電容量係数は重要ではありません。抵抗値は、総センサ静電容量 C_s によって異なります。抵抗値は、次の条件で選択します。

- 異なるセンサのタッチの Raw カウントをモニターする。
- 選択されたスキャン分解能でフルスケール読み値の約70%の読み値を提供する抵抗値を選択する。抵抗値が減ると、Raw カウント値は増えます。

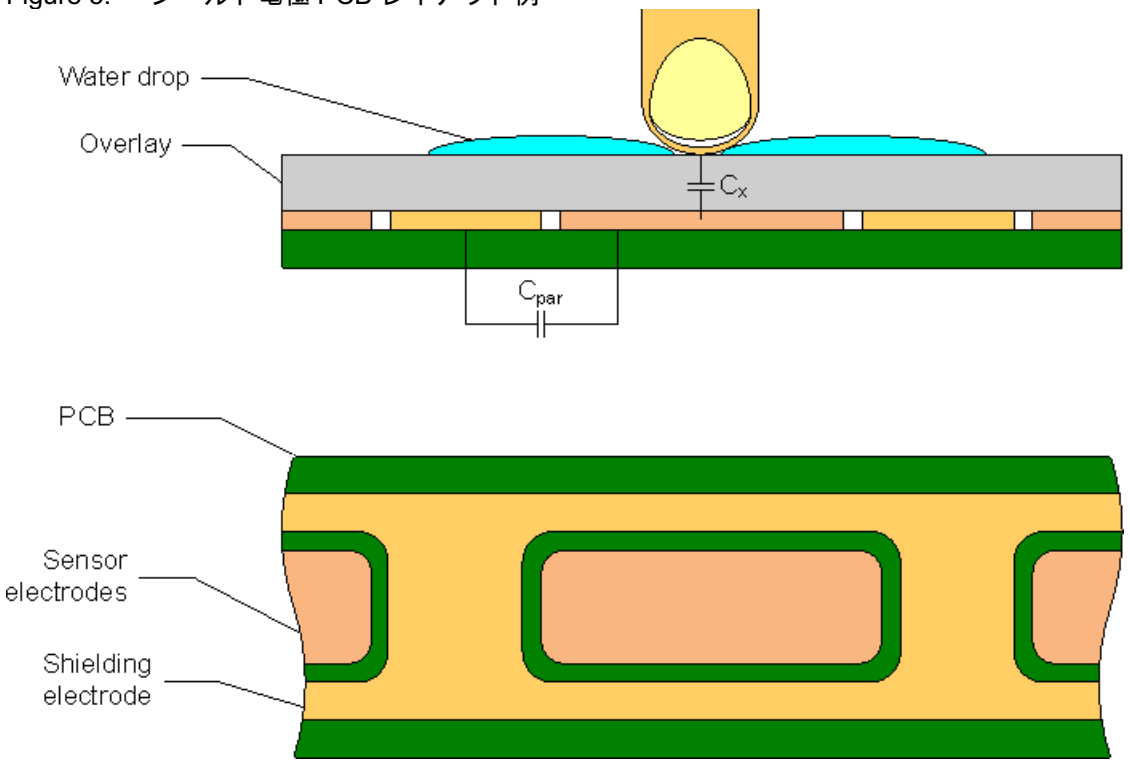
一般的な値は 500 Ω ~ 10 kΩの間で、センサキャパシタンスによって異なります。CY3280-22x45 評価ボードを使用している場合は、2 kΩから開始できます。

シールド電極

一部のアプリケーションでは、水膜や水滴がある場合でも動作の信頼性が要求されます。白物 (家電品) 、車載アプリケーション、様々な産業用アプリケーションなどでは、水、氷、湿度変化があっても誤反応をしない静電容量式センサが必要です。この場合、別個のシールド電極を使用することができます。この電極は検知電極の背部または外部に装備します。水膜がデバイスの遮断オーバーレイの表面にある場合、シールドと検知電極のカップリングが増えます。シールド電極は、寄生静電容量の影響を低減し、検知静電容量の変化の処理をするダイナミックレンジを広げます。

一部のアプリケーションでは、電極間のカップリングを増やして、検知電極の静電容量測定値のタッチ変化の逆を発生させるため、シールド電極信号と検知電極に対するその相対的位置を適切に選ぶことは有効です。これにより、高レベルソフトウェアの API 作業が簡単になります。CSD2x ユーザ モジュールは、シールド電極の個々の出力に対応します。

Figure 5.　シールド電極 PCB レイアウト例



前の図は、ボタンのシールド電極のレイアウト構成例を示しています。シールド電極は、LCD ドライブ電極のノイズの影響を阻止し、同時に浮遊静電容量を低減するため、透明な ITO タッチパッドデバイスでは特に有用です。

この例では、ボタンはシールド電極平面で覆われています。代替案として、ボタンの下のプレーンなど、PCB の反対側のレイヤに置くことも可能です。この場合、全面ベタに対して約 30 ~ 40% で、ハッチパターンを使用することが推奨されます。ここでは、グランド プレーンを追加する必要はありません。

水滴がシールドと検知電極の間にある場合、C_{par} が増え、変調器の電流が低減することがあります。実際のテストでは、変調器の基準電圧は API によって増加でき、水滴による生カウントはゼロに近いか、わずかにマイナスとなっています。これは、適切な変調器基準値を選択することによって達成できます。

このユーザ モジュールでは、プリチャージ可能なクロックで使用される同じ信号が、シールド電極にも提供されています。シールド電極は、ルーティング可能ないずれの PSoC ピンにも接続できます。駆動モードを Strong Slow に設定し、アース ノイズと放射妨害波を軽減します。立ち上がり、下り制限抵抗も、PSoC デバイスとシールド電極の間に接続できます。行 LUT 関数では、A を選択します。

コンパレータのリファレンス源

コンパレータのリファレンスは、コンパレータの基準電圧を形成するために使用されます。基準電圧値は、検出感度を決定します。

ユーザ モジュールは、IDAC コンフィギュレーションと Rb コンフィギュレーションでは、異なる基準形成原則を使用します。

Rb コンフィギュレーションの場合、ユーザ モジュールは、リファレンス源として以下のように複数の選択肢をサポートします。

- バンドギャップリファレンス
- 特殊な PWM 信号の駆動によるアナログ変調器
- 外付け抵抗分圧器

下表は、Rb コンフィギュレーションの場合のリファレンス選択のオプションをまとめたものです。

タイプ	外付けコンポーネント	UM の選択肢	使用時
バンドギャップリファレンス	なし	VBG	大部分のアプリケーションに推奨。このオプションから試してください。
アナログ変調器	なし	ASExx	読み値は電源電圧に比例。電源がうまく調節されている場合にのみ使用します。
外付け抵抗分圧器	2	AnalogColumn 入力選択	読み値はそれほど電源へ依存していません。推奨 R1 = 10k; R2 = 3.6k

IDAC コンフィギュレーションの場合、ユーザ モジュールは次のリファレンス源をサポートします。

- バンドギャップリファレンス
- V_{DD} リファレンス
- 外付け抵抗分圧器

下表は、IDAC コンフィギュレーションの場合のリファレンス選択のオプションをまとめたものです。

タイプ	外付けコンポーネント	UM の選択肢	使用時
バンドギャップリファレンス	なし	VBG	大部分のアプリケーションに推奨。このオプションから試してください。
電源リファレンス	なし	VDD	読み値は電源電圧に比例。電源がうまく調節されている場合にのみ使用します。

ほとんどの場合は、バンドギャップリファレンスまたはアナログ変調器のみを使用します。外付け抵抗分圧器は、特殊なケースに役立ちます。

クロック源

クロック源は、検知キャパシタ上でスイッチの制御に使用されます。ユーザ モジュールは、プリチャージスイッチのクロック源として、次の 2 つの選択オプションをサポートしています。

- 16-bit 擬似ランダム系列発生器（PRS）
- IMO プリスケーラ

必要な構成は、該当するユーザ モジュールのパラメータによって選択できます。

PRS 源は、スペクトル拡散を実現し、外部ノイズ源に対する良好な排除性を確実にします。さらに、拡散スペクトルクロックを使用した設計では、電磁放射レベルが低くなります。PRS クロック源は、アプリケーションが EMC/EMI テスト合格を目的としている場合、または厳しい環境で信頼性の高い動作が要求される場合に推奨されます。

この表では 2 つのクロック源を比較しています。

クロック源	動作周波数	EMC ノイズイミュニティ
PRS	スペクトラム拡散、平均は F_{IMO}/4/ プリスケーラ、ピークは F_{IMO}/4/ プリスケーラ。	高。高感度ポイントは、PRS シーケンスの繰返し期間の倍数で、PRS 基本周波数は F_{IMO}/ プリスケーラ。
IMO プリスケーラ	固定周波数 F_{IMO}/ プリスケーラ	中。高感度ポイントが多い。

配置

ユーザ モジュールのブロックは、ユーザ モジュールがインスタンス化されると自動的に配置されます。他の配置はシングルチャネル構成でのみ利用できます。CSD2X ユーザ モジュールは、CapSense ブロックと 1 つのコンパレータブロックを消費します。

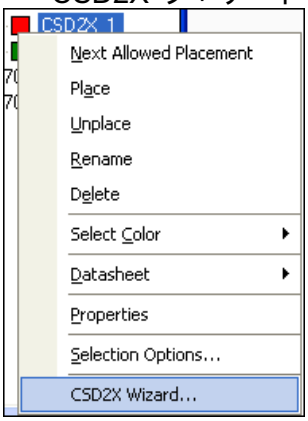
LCD や I2CHW などの特定のピンを必要とするユーザ モジュールは、CSD2X ユーザ モジュールのピン接続を確立するために、CSD2X ウィザードを開始する前に配置しなければなりません。

静電容量式センサの接続を配置する場合、P1[0] と P1[1] は避けてください。これらのピンは、そのデバイスのプログラミングに使用され、センサの検出感度とノイズに影響を与える過度のルーティング静電容量を持つ可能性があります。

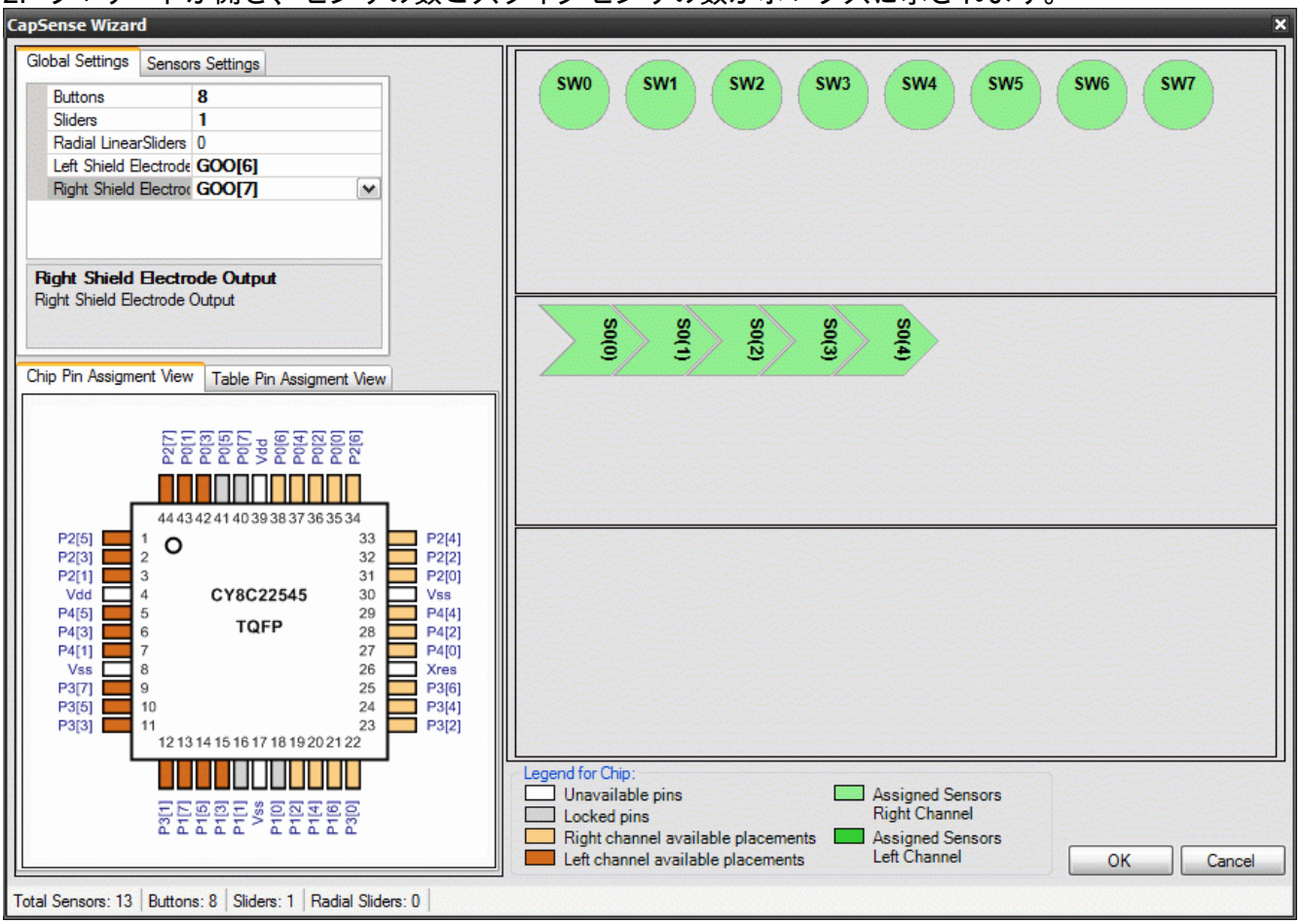
ウィザード

CSD2x ウィザードを使って、CapSense ボタン、スライダ、近接検知センサのピン配列を設定します。ドラッグ アンド ドロップ インタフェースを使って、構成を選択し、ボタンとセグメントを割り当てます。

1. ウィザードにアクセスするには、Workspace Explorer のユーザーモジュールの右側をクリックして、CSD2X ウィザードを選択してください。



2. ウィザードが開き、センサの数とスライダセンサの数がボックスに示されます。



ウィザードのピン凡例

白 – このピンは CapSense の入力に使用できません。

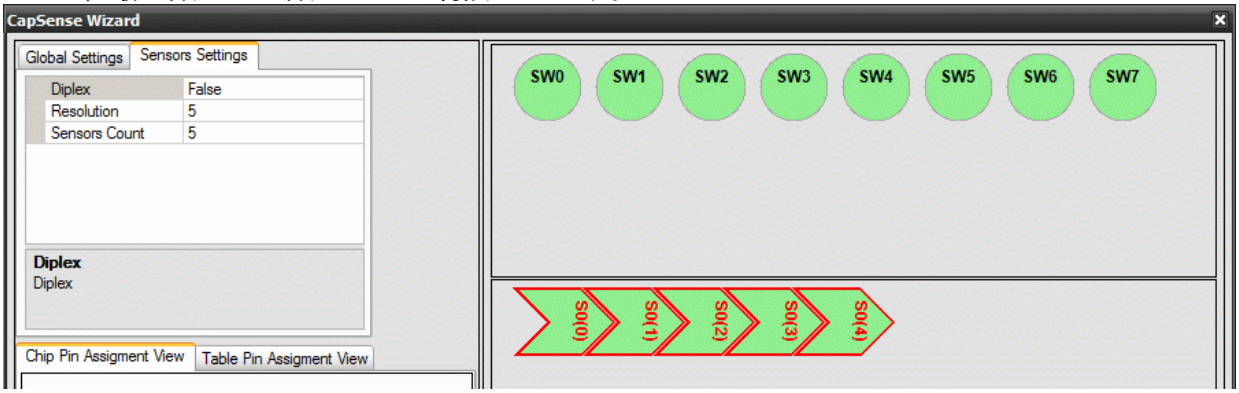
グレー – このピンはロックされています。これには 2 つの原因が考えられます。その 1 つ目は、LCD や I^2C などの別のユーザ モジュールが、そのピンを使用している場合です。2 つ目は、ピンの名前がデフォルトから変わった場合です。ピン名をデフォルトに戻すには、

Pinout ビューでそのピンの表示を広げ、Select メニューで Default を選択します。これで、ピンはウィザードで割り当てられるようになりました。

オレンジ – このピンは割り当て可能です。

グリーン – このピンは CapSense 入力に割り当てられています。

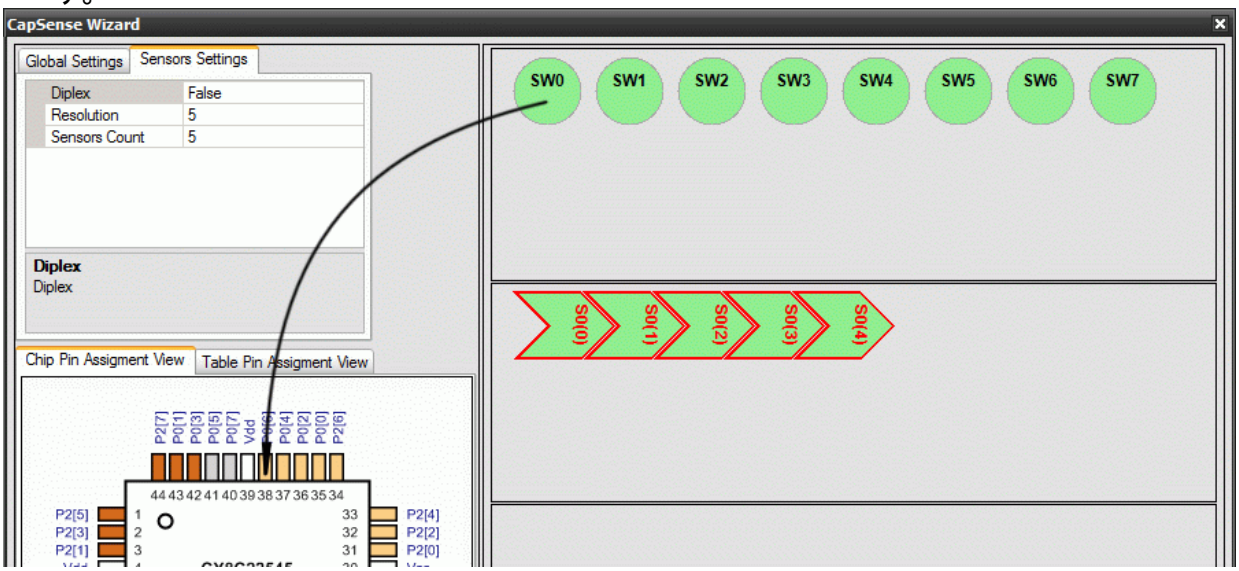
3. ウィザードの右側には、スイッチ、スライダ、回転式スライダを設定するための 3 つのエリアがあります。スイッチ数、スライダ数、回転式スライダ数をそれぞれのボックスで指定し、[Enter] キーを押します。今指定した構成に従って、表示が更新されます。
4. [センサ設定] タブを選択し、スライダのパラメータを設定します。スライダと回転式スライダ (スライダと回転式スライダのセンサ数を含む) は、まとめて設定できます。X-Y タッチパッドには 2 つのスライダが必要ですが、選択されるのは 1 つです。スライダセンサ中の現実的な最低センサ数は 5 で、最大数はピン数によって制限されます。



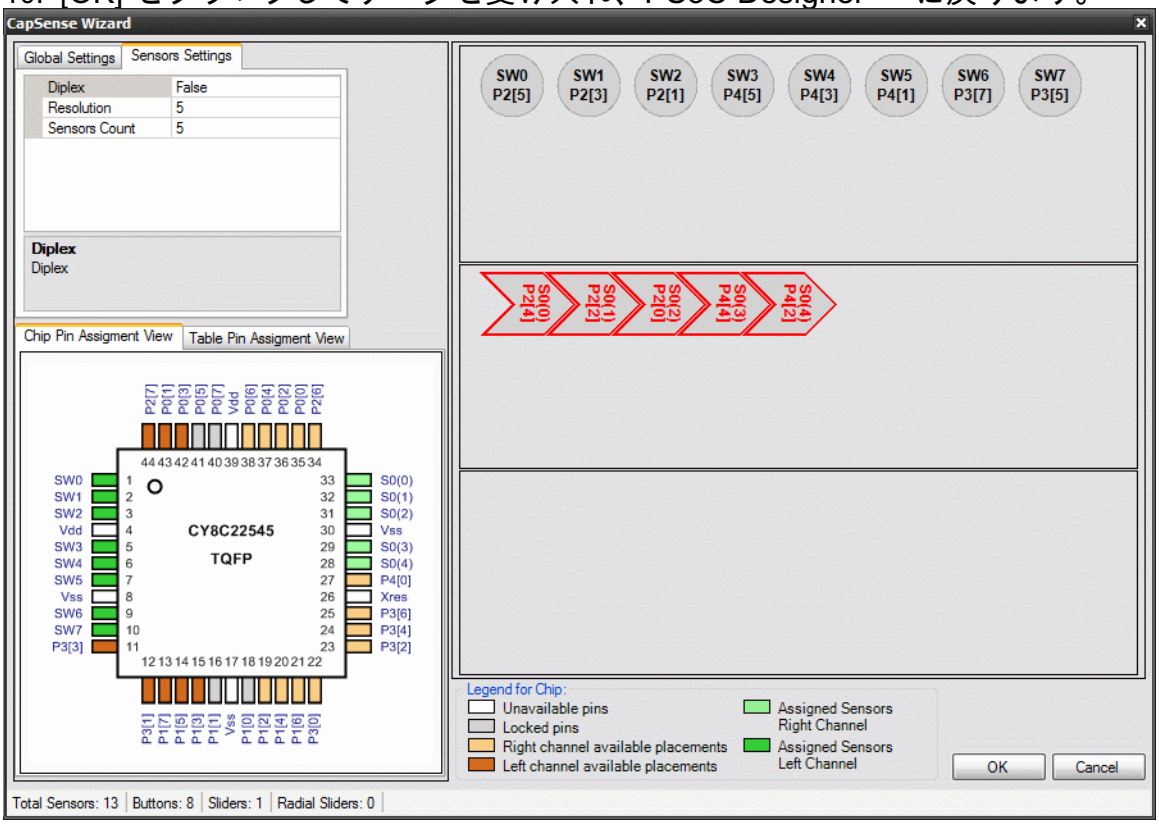
5. スライダの分解能は、スライダが持つ位置の数です。1 つ以上のスライダセグメントで読み値が出るタッチは、この分解能に補間されます。分解能を 100 に設定すると、スライダ上の指の位置は 0 と 99 の間のどこかであると報告されます。最低値は 5 です。最大値は、ダイプレックススライダの場合、(センサに使用されるピン数 – 1) x 2^8 – 1 or (2 x センサに使用されるピン数 – 1) x 2^8 -1 です。
6. 必要に応じて、ダイプレックスを選択します。これにより、センサ用に選択されたピンを、基板上で 2 倍の数のセンサ位置にマッピングできます。上図では、ダイプレックスセンサの上半分だけが示されています。下半分は、前述の「ダイプレックス」の項での説明の通り、自動的にマッピングされます。「ダイプレックス」の項で、ピン説明のダイプレックス表を参照してください。

7. ピンにスイッチをマッピングするには、スイッチを左クリックして空いているピンまでドラッグします。この操作は、[Chip Pin Assignment View](チップピン割り当てビュー) や [Table Pin Assignment View](表のピン割り当てビュー) で行うと便利です。ポート ピンを選択するとグレイになり、

使用不可となります。ポート ピンからセンサをドラッグして外すと、センサ割り当てを変更できます。



8. 他のセンサでも同じ操作を繰り返します。
9. スライダセグメントのポート ピンへのマッピングは、個々のセンサの場合と同様です。
10. [OK] をクリックしてデータを受け入れ、PSoC Designer ™ に戻ります。



これでセンサの配置が完了しました。ユーザ モジュールのパラメータを選択し、アプリケーションを生成します。必要に応じて、サンプルプロジェクトを使用することもできます。

ピン割り当てを変更するには、割り当てられているピンにカーソルを合わせてクリックし、それをピン割り当てボックスの外までドラッグ アンド ドロップします。これでこのピンは割り当てから外され、他に割り当てることが可能になります。

ウィザードのスライダ設定

ダイプレックス

スライダ専用。1 つのピンを使って、2 つの電気センサで分解能の向上をモニタすることができます。詳細はダイプレックスの項を参照してください。

分解能

スライダと回転式スライダでは、この値は CSD2x_wGetCentroidPos API によって返される値の範囲を設定します。スライダセンサが 1 つでもアクティブの場合、関数はゼロから CSD ウィザードで設定された分解能値までを返します。CapSense アルゴリズムは、隣接センサからの読み値に基づき、この分解能に合わせてセントロイド位置を補間します。

センサ数

この値は、各スライダや回転式スライダのセンサ数を設定します。

ウィザードで作る表

ウィザードを完了したら、[Generate Application] (アプリケーションの生成) をクリックします。入力したセンサ数、ピン割り当て、ダイプレックスに基づいて、一連の表が生成されます。これらの表は CSD2X_Table.asm に保存されています。

センサ表

センサ表は各センサに対して 2 バイトのエントリから構成されています。第 1 バイトはポート番号で、第 2 バイトはそのビットのビットマスクです (ビット番号ではありません)。左右のチャネル用に表は 2 つあります。表には、すべての独立したセンサ、次に各センサが順番に列挙されています。次に、6 つのセンサを含む表の例を示します。

```
CSD2X_Sensor_Table_Right:
_CSD2X_Sensor_Table_Right:
      dw    0x0140  //  Port 1 Bit 6
      dw    0x0301  //  Port 3 Bit 0
      dw    0x0304  //  Port 3 Bit 2
CSD2X_Sensor_Table_Left:
_CSD2X_Sensor_Table_Left:
      dw    0x0308  //  Port 3 Bit 3
      dw    0x0302  //  Port 3 Bit 1
      dw    0x0108  //  Port 1 Bit 3
```

この表は CSD2X_wGetPortPin() ルーチンで使用されます。

グループ表

グループ表は、ボタンセンサやスライダのグループを定義します。各スライダにつきエントリが 1 つ、フリーボタンセンサにもエントリが 1 つあります。最初のエントリは必ずフリーセンサです。各エントリは 6 バイトです。第 1 バイトはセンサ表のインデックスです。第 2 バイトはグループ内のセンサ数です。第 3 バイトは、スライダがダイプレックスであるかどうかを示します (4 はダイプレックス、0 は非ダイプレックス)。第 4、第 5、第 6 バイトは固定ポイント乗数で、スライダの計算されたセントロイドが乗算されて、CSD2X ウィザードで望ましい分解能が達成されます。

```
CSD2X_Group_Table:
```

```
_CSD2X_Group_Table:
; Group Table:
; Origin Count Diplex DivBtwSw(wholeMSB, wholeLSB, fractByte)
db 0x0,     0x3,     0x00,      0x00,    0x00,    0x00 ; Buttons
db 0x3,     0x8,     0x4,       0x0,     0x0,     0x44 ; Slider 1
```

ダイプレックス表

ダイプレックス表のスキャンオーダデータは、スライダでダイプレックスされている場合に、グループを対象として作成されます。これ以外の場合は、ラベルが作成されますが、データは記録されません。この表は、各スライダのセンサマッピングと、各スライダによるそれぞれの表のリファレンスという 2 つの部分から構成されています。ここでは、5 つのセンサ スライダを使った典型的な例を示します。

```
DiplexTable_0:
; This group is not a diplexed slider
DiplexTable_1:
db 0,1,2,3,4,5,6,7,0,3,6,1,4,7,2,5 // 8 switch slider
CSD2X_Diplex_Table:
_CSD2X_Diplex_Table:
db >DiplexTable_0, <DiplexTable_0
db >DiplexTable_1, <DiplexTable_1
```

順番表

順番表は、各センサについて 1 バイトで構成されます。このバイトは、チャネル依存のない行カウント結果アレイ中の左センサの順番です。この表はウィザードで作成され、センサの順番を示します。

```
 CSD2X_1_Order_Table_Left:
_CSD2X_1_Order_Table_Left:
DB 0x01 // Position 1
CSD2X_1_Order_Table_Right:
_CSD2X_1_Order_Table_Right:
DB 0x00,0x02 // Position 0 and 2
```

IDAC 表

IDAC 表は、AutoCalibration が有効なとき、ICAD 構成を持つ CSD2X 用に生成されます。この表は、実行時に変更できる各センサの校正済み電流値を示します。これは複雑なプロジェクトで役立ちます。

ダブルチャネル構成の場合：

```
 CSD2X_1_baDACCodeBaselineL:
_CSD2X_1_baDACCodeBaselineL:
BLK CSD2X_1_TotalLeftSensorCount
CSD2X_1_baDACCodeBaselineR:
_CSD2X_1_baDACCodeBaselineR:
BLK CSD2X_1_TotalRightSensorCount
```

シングルチャネル構成の場合：

```
 CSD2X_1_baDACCodeBaselineL:
_CSD2X_1_baDACCodeBaselineL:
BLK CSD2X_1_TotalSensorCount
CSD2X_1_baDACCodeBaselineR:
_CSD2X_1_baDACCodeBaselineR:
BLK CSD2X_1_TotalSensorCount
```

ソフトウェア割り込み互換性

警告： この操作の間、CSD2x ユーザ モジュールは、INT_CLR2 レジスタの内容を理論値「1」で上書きします。これは、ソフトウェア割り込み機能は、CSD2x ユーザ モジュールとともには使用できないことを意味します。INT_MSK3 レジスタの ENSWINT ビットフィールドの値を変更しないでください。これを変更するとプロジェクト操作が妨害されます。

パラメータおよびリソース

自動キャリブレーション

自動キャリブレーション API 関数を有効· 無効にします。

自動キャリブレーションは、IDAC 構成だけに含まれており、自動的に可能な IDAC 値を選択し、分解能範囲の半分で Raw カウントを取得します。これにより、CapSense アルゴリズムの全体的な感度は下がりますが、プロセス調整の開始時に読み取り可能な範囲で Raw カウントを手早く取得できます。自動キャリブレーションは、ROM と RAM を消費し、開始時間が増加します。キャリブレーション終了後の Raw カウント値が分解能範囲の半分より少ない場合、IDAC 範囲を増やすか、プリチャージ周波数を減らす必要があります。自動キャリブレーションを行うと、機能レベルが下限ぎりぎりの構成を改良できます。このパラメータが有効な場合、ユーザ モジュール起動中で CSD2X_Start API 終了時に、CSD2X_Calibrate API 関数が自動的に呼び出されます。

ノイズ閾値

この閾値を上回るカウント値が観測された場合、基底値は更新しません。スライダセンサでは、この閾値を下回るカウント値が観測された場合、セントロイドの計算に加えられません。可能な値は 5 から 255 です。

シールド電極出力

このパラメータは、オプションのシールド電極をサポートするアルゴリズムを有効·無効にします。有効な場合は、シールド電極は CSD2x ウィザードで、左チャネル用は P1[6] または P3[6]、右チャネル用は P1[7] または P3[7] に接続されます。ピン数の少ないデバイスでは、P3[6] と P3[7] が利用できない場合もあります。シングルチャネルでは、この接続は右左の CapSense チャネルのうちどちらが使用されるかによって異なります。

選択されたピンの駆動モードは「Strong」にセットします。

BaselineUpdate 閾値

新しい Raw カウント値が現在の基底値を上回っており、その差がノイズ閾値を下回る場合 (センサのオートリセットパラメータは無効にセットされている)、現在の基底値と Raw カウントの差はバケツに集められたような状態になります。バケツが満杯になると、基底値はある値分増分し、バケツは空になります。このパラメータは、基底値が増分するためにバケツが達成しなければならない閾値を設定します。可能な値は 0 から 255 です。パラメータ値が大きいときは、基底値の更新速度を遅くします。より頻繁な基底値の更新が必要なときは、このパラメータを小さくします。

センサの Autoreset

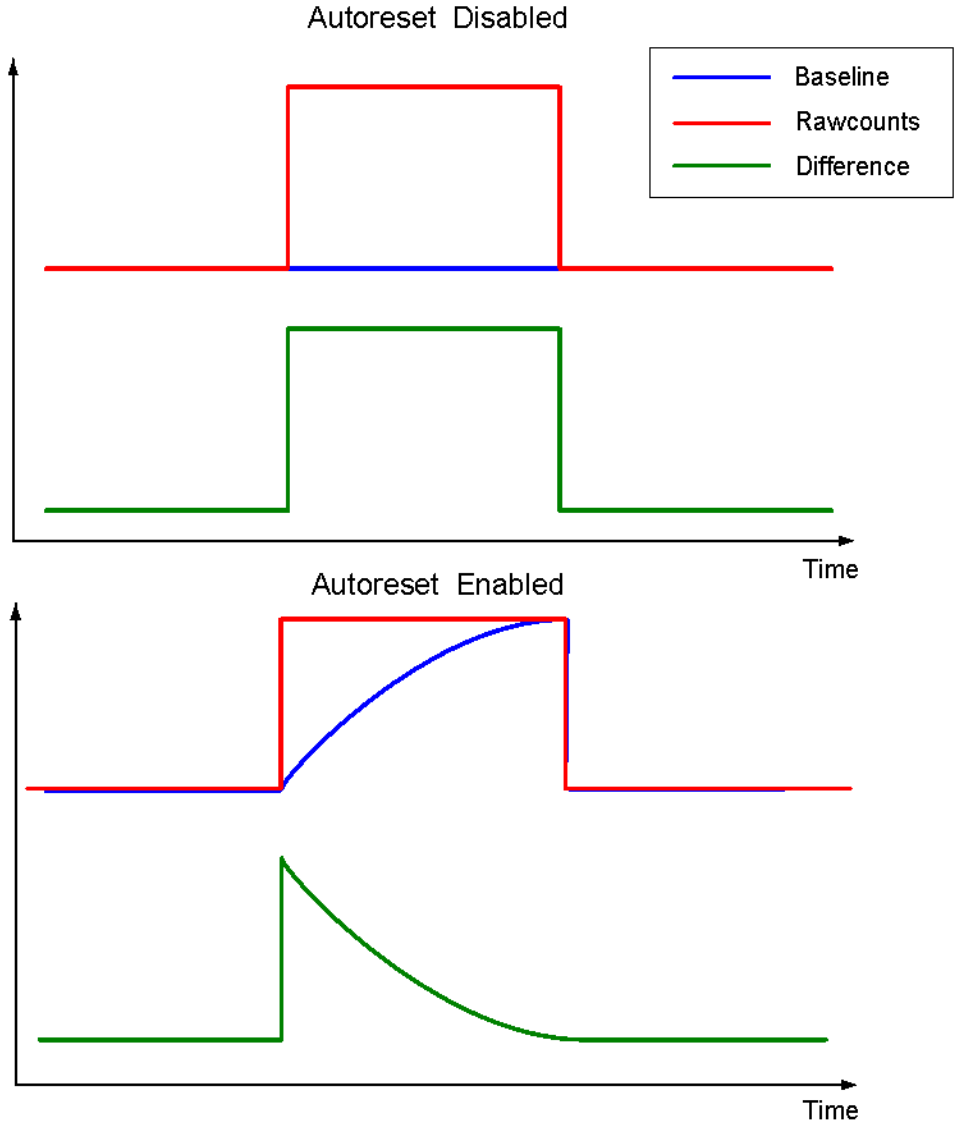
このパラメータは、基底値が常時更新されるか、信号差がノイズ閾値より低い場合のみ更新されるかを指定します。[Enabled] (有効) にセットされている場合、基底値は常時更新されます。この設定は、センサの最大時間を制限します (標準的な値は 5 ~ 10 秒) が、何もセンサに触れずに生力ウントが突然上がった際に、センサがマニュアル的にオンになるのを妨ぐことができます。この突然の上昇の原因には、大幅な電源電圧の変化、高工ネルギー RF ノイズ源、非常に速い温度変化があります。

パラメータが [Disabled] (無効) にセットされている場合、Raw 値と基底値の差がノイズ閾値パラメータを下回る場合にのみ、基準値は更新されます。何もセンサに触れずに Raw カウントが突然

上がった際に、センサがマニュアル的にオンになるという問題がない限り、このパラメータは [Disabled] にしておきます。

以下の図は、このパラメータが Baseline 値更新に与える影響について示しています。

Figure 6. センサ自動リセットパラメータ



ヒステリシス

ヒステリシスパラメータは、センサが現在動作中であるか否かに応じて、指閾値に数値を足したり、そこから引いたりします。センサが動作中でない場合、差の数は指閾値 + ヒステリシスを上回る必要があります。センサが動作中の場合、差の数は指閾値 - ヒステリシスを下回る必要があります。これは、デバウンス性と粘着性を指検知アルゴリズムに加えるために使用されます。ヒステリシスを伴う閾値は、bIsSensorActive() または bIsAnySensorActive() が呼び出されたときに評価されます。センサの状態は、戻り値 bIsSensorActive() または baSnsOnMask[] アレイを使用してモニターされます。可能な値は 0 から 255 ですが、指閾値パラメータ設定よりも低くなければなりません。

適切な高レベル判断理論パラメータを選択すると、環境要因 (温度や湿度の変化など) を効果的に補正し、ノイズ信号 (ESD、電源スパイク) を抑圧し、様々な条件化で信頼できるタッチ検知を実施できます。

Debounce (デバウンス)

デバウンス パラメータは、センサの動作遷移のためのデバウンスカウンタを設定します。センサが非作動中から動作中へ遷移するためには、指定されたサンプル数に対して、差 (Difference) 値が指閾値 + ヒステリシスを上回る状態を続けなければなりません。デバウンス数は、API 関数の bIsSensorActive または bIsAnySensorActive で増分されます。

可能な値は 1 ~ 255 です。1 をセットするとデバウンスは起こりません。

指の閾値

この閾値は、各ボタンセンサの状態を判断するために使用します。1 つでもセンサがアクティブな場合、bIsAnySensorActive() 関数は 1 を返します。すべてのセンサがオフの場合、bIsAnySensorActive() 関数は 0 を返します。

指検知閾値は、すべてのセンサとスライダに適用されます。個々のセンサ (スライダグループに属さないセンサ) では、これらの閾値は変数で、baBtnFThreshold[] アレイに提供されます。CSD2X_SetDefaultFingerThresholds() 関数を使用すると、閾値をユーザ モジュールのパラメータで設定されているデフォルト値に設定できます。個々のセンサの感度を調整するには、各センサの baBtnFThreshold[] 値を変更します。(このバイトアレイのサイズは、個々の実施センサ数と同等です。)

可能な値の範囲は 1 ~ 255 です。

スキャン速度

このパラメータは、センサのスキャン速度に影響を及ぼします。選択肢は次のとおりです。超 高速、高速、通常、低速。スキャン速度が遅いと、次のような利点があります。

- より改善された S/N 比
- 電源と温度変化に対するより良いイミュニティ

スキャン速度は、次のようにスキャン速度分周器に影響を与えます。

スキャン速度	分周器
超高速	1
高速	2
通常	4
低速	8

スキャン分解能

このパラメータはスキャン分解能をビット数で判断します。センサは 9 ~ 16 ビットの分解能でスキャンできます。ビット数が N の場合、スキャン分解能の最大 Raw カウントは 2^N-1 です。

分解能を高くすると、検出感度とタッチ検知の S/N 比が高くなります。近接検知用には高分解能を使用します。16-bit 分解能、低速スキャンモード、20 cm のワイヤによって、20 cm 以上離れた手を検知できます。

Table 2.　シングルチャネル構成の場合：24 MHz CPU 周波数 動作における、スキャン時間 (単位：µs) 対スキャン速度と分解能、プリスケーラ = SYSCLK/1 を持つ構成

分解能 (単位：ビット)	スキャン速度			
	超高速	高速	通常	低速
9	51.49	72.39	114.8	199.8
10	72.43	114.9	200.1	370
11	115.4	200.3	370	710
12	200.3	370	710.1	1391
13	370.1	710.3	1392	2752
14	710.8	1390	2752	5474
15	1390	2752	5472	10910
16	2752	5472	10910	21800

Table 3.　デュアルチャネル構成の場合：24 MHz CPU 周波数 動作における、スキャン時間 (単位：µs) 対スキャン速度と分解能、プリスケーラ = SYSCLK/1 を持つ構成

分解能 (単位：ビット)	スキャン速度			
	超高速	高速	通常	低速
9	83.4	104	146.5	231.7
10	104	146.5	231.7	402
11	146.5	231.6	402	741
12	231.6	402.2	741	1422
13	402	741.2	1422	2784
14	741	1422	2784	5504
15	1422	2784	5508	10950
16	2784	5504	10950	21840

Note

1. スキャン時間は、2 つのセンサスキャンの間の間隔を測定したものです。この時間には、センサのセットアップ時間、変調器安定化遅延、サンプル変換間隔、データ前処理時間が含まれています。
2. 任意の分解能のスキャン時間と速度設定は、プレスケーラ値の増加に比例して増加します。たとえば、プレスケーラを SYSCLK/1 から SYSCLK/2 に変更すると、2 つの要素によりこのテーブルのスキャン時間が増加します。

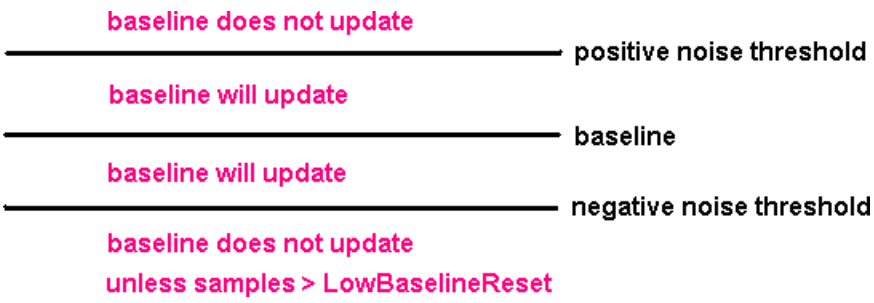
LowBaselineReset

LowBaselineReset パラメータは、NegativeNoiseThreshold パラメータとともに作動します。指定されたサンプル数で、サンプルカウント値が (Baseline 値 - NegativeNoiseTreshold) 以下の場合、Baseline 値は新しい Raw カウント値に設定されます。これは基本的に、異常に低い Raw カウントが認識されたときに、基底値をリセットする必要があるときに使われるもので、通常、指などが置

かれたまま起動された時等の異常な状態をリセットする為に使用されます。。使用可能な値の範囲は 0...255 です。

NegativeNoiseThreshold

NegativeNoiseThreshold パラメータは、マイナスの差カウント閾値を追加します。現在の Raw カウントが基底値を下回り、これらの差がこの閾値を上回る場合、基底値は更新されません。しかし、LowBaselineReset パラメータで設定されたサンプル数の間、現在の Raw カウントが低い状態で続く場合（差は閾値より大きい）、基底値はリセットされます。使用可能な値の範囲は 0...255 です。



変調器コンデンサピン

変調器コンデンサピンは、シングルチャネル構成でのみ使用します。このパラメータは、外付け変調器コンデンサ（C_{mod}）を接続するピンをセットします。利用可能なピンは P0[5] と P0[7] とです。ダブルチャネル構成では両方のピンを使用します。

リファレンス源

このパラメータは、コンパレータのリファレンスソースを設定します。選択肢は次の通りです。バンドギャップ（Vbg）、アナログ変調器（ASExx）、外付け電圧（AnalogColumn_InputSelect_x). IDAC 構成の場合、選択肢は Vbg または Vdd です。詳細は、「コンパレータリファレンス源」のセクションを参照してください。

リファレンス値

IDAC 構成と Rb 構成の両方で、値 0 は GND に近い基準電圧に相当します。値 31 は V に近い基準電圧に対応します。これにより、基準電圧に 5-bit 分解能をもたらします。

Rb 構成では、基準値が外部ピンまたは Vbg から来ている場合、リファレンス値パラメータは無効となります。この場合、リファレンス値の設定は無視できます。また Rb 構成では、基準電圧の実際の分解能は 4-bits です。ただし、ここで使用される値は 5-bit スケール（すなわち 0 ~ 31）のままです。

高いリファレンス値を使用すると、基準電圧が高くなります。基準電圧が高くなると、センサの Raw カウントが増加します。同様に、リファレンス値· 基準電圧が低いと、センサの行カウントが減少します。

プリチャージ源

このパラメータは、プリチャージスイッチのクロック源を選択します。可能なオプションは PRS とタイマです。ほとんどの場合は、PRS 源を使用すると、より良い EMI イミュニティと低放射を実現できます。

プリスケーラ

このパラメータは、プリスケーラ比をセットし、プリチャージスイッチ出力周波数を特定します。このパラメータは、任意の分解能とスキャン速度設定に対する、PRS 出力周波数とセンサスキャン時間にも影響します。

可能な値は以下のとおりです。

- SYSCLK/1
- SYSCLK/2
- SYSCLK/4
- SYSCLK/8
- SYSCLK/16
- SYSCLK/32
- SYSCLK/128
- SYSCLK/256

Feedback Resistor Pin (フィードバック 抵抗 ピン)

このパラメータは、Rb 構成でのみ利用できます。このパラメータは、外付けフィードバック 抵抗 (R_b) を接続するピンを設定します。Rb コンフィギュレーションを伴うデュアルチャネルは左右のチャネルそれぞれに、フィードバック抵抗ピン L とフィードバック抵抗ピン R プロパティを持ちます。利用できるピン：左チャネルでは P1[4]、P3[4]、GOO[4] 右チャネルでは P1[5]、P3[5]、GOO[5] です。デバイスパッケージによっては使用できないピンもあります。ヒント：これらのピンの一部が、センサ接続の割り当てなど、その他の目的で使用される場合は、ユーザモジュールパラメータ リストには表示されません。今後の CSD ユーザ モジュールのバージョンでは、フィードバック抵抗の接続に使用できるピンが追加される可能性があります。これにより、P3 ポートを持たないパッケージで第 2 の I^2C ポートを使用できるようになります。プログラミングの問題を避けるために、P1[5] または P3[1] を使用してください。

IDAC 範囲

このパラメータは、iDAC 構成でのみ利用できます。iDAC 電流マルチプレクサを設定します。デュアルチャネル構成では、この設定の結果は異なります。次にデュアルチャネル構成の結果を示します。

2 チャネル構成では、左のチャネルに大きなキャパシタンスを持つセンサを接続します。

IDAC 値

静電容量測定範囲は、このパラメータによって異なります。値が高い場合は範囲が広くなります。IDAC 値を調整して、フルレンジの約 50-70% の Raw カウントを得られるようにします。このパラメータは、CSD2X_SetLeftDACValue または CSD2X_SetRightDACValueAPI 関数を用いて実行時に変更できます。可能な値は 1 ～ 255 です。

補正 IDAC 値

補正 IDAC は、センサの初期キャパシタンスを補正することを目的としています。このパラメータを調整すると検出感度と S/N 比を改善できますが、あまり高い値に設定すると CSD2X の操作が妨害されます。調整プロセスでは、この値を 0 から開始して最大の S/N 比と検出感度が得られるまで上げていきます。このパラメータは、シングルチャネル IDAC 構成でのみ利用できます。

可能な値は 0 から 255 です。

アプリケーション プログラミング インタフェース

アプリケーション プログラミング インタフェース (API) 関数は、高レベルでモジュールを扱うことができるように、ユーザ モジュールの一部として提供されています。このセクションでは、各機能に対するインタフェースを「include」ファイルによって提供される関連定数とともに示します。

ユーザ モジュールを配置するたびに、インスタンス名が指定されます。デフォルトでは、PSoC Designer プロジェクトでは、CSD2X_1 をこのユーザ モジュールの最初のインスタンスとして指定します。これは識別子の構文ルールに従った一意の値に変更できます。割り当てたインスタンス名が、全てのグローバル関数名、変数、および定数記号の接頭語になります。次の説明では、簡単にするために、インスタンス名は省略されて単に「CSD2X」となっています。

注 ** ここでは、すべてのユーザ モジュール API と同様に、API 関数を呼び出すことで A と X レジスタの値は、変更される可能性があります。関数を呼び出す場合、呼出し後に A と X の値が必要になるならば、必ず呼び出し前に A と X の値を保存してください。PSoC Designer のバージョン 1.0 以降では、効率を高めるために、この「registers are volatile (レジスタの揮発性)」ポリシーが選択され、実施されています。C コンパイラは自動的にこの条件を処理します。アセンブリ言語のプログラマは、コードでこのポリシーを考慮する必要があります。ユーザ モジュール API 関数の中には、A と X を変更しないものもありますが、今後も変更されないという保証はありません。

大容量メモリデバイスでは、CUR_PP、IDX_PP、MVR_PP 及び MVW_PP レジスタの値を保護するのは呼び出し元の責任です。今すぐ修正されないレジスタもありますが、今後のリリースにこのケースが留まるという保証はありません。

エントリポイントは、CSD2X を初期化し、サンプリングを開始し、CSD2X を終了するためのものです。すべての場合では、モジュールのインスタンス名が CSD2X 次の入力内容に記載されているプリフィックスに置き換えられます。間違ったインスタンス名の使用は、構文エラーの一般的な原因です。

API の機能は、様々なグローバルアレイを使用します。これらのアレイをマニュアルで変更しないでください。ただし、デバックの目的でこれらの値を調べることが可能です。たとえば、チャート作成ツールを使用して、アレイの内容を表示することは可能です。次にいくつかのグローバルアレイを挙げます。

- CSD2X_waSnsBaseline[]
- CSD2X_waSnsResult[]
- CSD2X_waSnsDiff[]
- CSD2X_baSnsOnMask[]

CSD2X_waSnsBaseline[] – 各センサの基底値データを含む整数アレイです。アレイのサイズはセンサ数と同等です。CSD2X_waSnsBaseline[] アレイは以下の関数によって更新されます。

- CSD2X_UpdateAllBaselines();
- CSD2X_UpdateSensorBaseline();
- CSD2X_InitializeBaselines().

CSD2X_waSnsResult[] – 各センサの Raw データの INT 型アレイです。アレイのサイズはセンサ数と同等です。CSD2X_waSnsResult[] データは次の関数によって更新されます。

- CSD2X_ScanSensor();
- CSD2X_ScanAllSensors();

CSD2X_waSnsDiff [] – 各センサの Raw データと基底値データの差を含む整数アレイです。アレイのサイズはセンサ数と同等です。このデータは CSD_UpdateSensorBaseline(). によって更新されます。

CSD2X_baSnsOnMask[] – センサのオン・ オフの状態 (ボタンまたはスライダ) を維持するバイトアレイです。CSD2X_baSnsOnMask[0] は、センサ 0 ~ 7 のマスクされたビットを含んでいます (センサ 0 はビット 0、センサ 1 はビット 1)。CSD2X_baSnsOnMask[1] はセンサ 8 ~ 15 のマスクされたビットを

含んでいます。必要に応じて、同様の方法で、より多くのセンサにも対応できます。このバイトアレイには、全ての配置されているセンサの要素が含まれます。ボタンがオンの場合 1 つのビットの値は 1 で、オフの場合その値は 0 です。CSD2X_baSnsOnMask[] データは、CSD2X_bIsSensorActive(BYTE bSensor) 関数または CSD2X_bIsAnySensorActive() ルーチンによって更新されます。

CSD2X_Start

説明：

レジスタを初期化し、ユーザーのモジュールを起動します。他のユーザ モジュール関数を呼び出す前に、この関数を呼び出さなければなりません。AutoCalibration が有効なとき、他のユーザ モジュールパラメータが適用された後、CSD2X_CalibrateAPI はこの関数内で自動的に呼び出されます。

C プロトタイプ

```
void  CSD2X_Start()
```

アセンブラ

```
lcall  CSD2X_Start
```

パラメータ：

なし

戻り値：

なし

副作用：

**

CSD2X_Stop

説明：

センサスキャンを停止し、内部割り込みを無効にし、CSD2X_ClearSensors() を呼び出してすべてのセンサをリセットしてオフ状態にします。

C プロトタイプ

```
void  CSD2X_Stop()
```

アセンブラ

```
lcall  CSD2X_Stop
```

パラメータ：

なし

戻り値：

なし

副作用：

**

CSD2X_SetScanMode

説明：

この関数は、後続のすべてのスキャンで、ユーザ モジュール パラメータに設定されているスキャン速度とスキャン分解能を上書きします。分解能は 9 ~ 16 の間の整数です。

C プロトタイプ

シングルチャネル構成の場合：

```
 void  CSD2X_SetScanMode(BYTE bSpeed, BYTE bResolution);
```

ダブルチャネル構成の場合：

```
 void  CSD2X_SetLeftScanMode(BYTE bSpeed, BYTE bResolution);
void  CSD2X_SetRightScanMode(BYTE bSpeed, BYTE bResolution);
```

アセンブラ

シングルチャネル構成の場合：

```
 mov  A,  bSpeed
mov  X,  bResolution
lcall  CSD2X_SetScanMode
```

ダブルチャネル構成の場合：

```
 mov  A,  bSpeedL
mov  X,  bResolutionL
lcall  CSD2X_SetLeftScanMode
mov  A,  bSpeedR
mov  X,  bResolutionR
lcall  CSD2X_SetRightScanMode
```

パラメータ：

A => スキャン速度

X => スキャン分解能。9 ~ 16 の間の整数。

次の定数は bSpeed パラメータ用です。

定数	値
CSD2X_ULTRAFAST_SPEED	0x00
CSD2X_FAST_SPEED	0x01
CSD2X_NORMAL_SPEED	0x02
CSD2X_SLOW_SPEED	0x03

戻り値：

なし

副作用

**

CSD2X_SetPrescaler

説明：

この関数は、後続のすべてのスキャンで、ユーザ モジュール パラメータに設定されているプリスケーラ値を上書きします。

C プロトタイプ

シングルチャネル構成の場合：

```
void  CSD2X_SetPrescaler(BYTE bPrescaler);
```

ダブルチャネル構成の場合：

```
void  CSD2X_SetLeftPrescaler(BYTE bPrescaler);
void  CSD2X_SetRightPrescaler(BYTE bPrescaler);
```

アセンブラ

シングルチャネル構成の場合：

```
 mov  A,  bPrescaler
lcall  CSD2X_SetPrescaler
```

ダブルチャネル構成の場合：

```
 mov  A,  bPrescalerL
lcall  CSD2X_SetLeftPrescaler
mov  A,  bPrescalerR
lcall  CSD2X_SetRightPrescaler
```

パラメータ：

A => プリスケーラ値。

次の定数は bPrescaler パラメータ用です。

定数	値
CSD2X_PRESCALER_SYSCLK1	0x30
CSD2X_PRESCALER_SYSCLK2	0x20
CSD2X_PRESCALER_SYSCLK4	0x10
CSD2X_PRESCALER_SYSCLK8	0x00
CSD2X_PRESCALER_SYSCLK16	0x40
CSD2X_PRESCALER_SYSCLK32	0x50
CSD2X_PRESCALER_SYSCLK128	0x60
CSD2X_PRESCALER_SYSCLK256	0x70

戻り値：

なし

副作用

**

CSD2X_ScanSensor

説明：

選択されたセンサをスキャンします。各センサは、センサアレイ内で独自の番号を持っています。シングルチャネル構成では、この番号は CSD2X ウィザードによって順番に割り当てられます。Sw0 は センサ 0、Sw1 はセンサ 1 などのように割り当てられます。

2 チャネル構成では、センサ番号は 0 から最大チャネルセンサ番号までの間の値です。例えば、左側に 2 つのセンサがある場合、これらはそれぞれ 0 と 1 となります。右側にも 2 つのセンサがある場合、これらもそれぞれ 0 と 1 となります。0xFF の値がセンサ番号としてこの関数に入力されると、そのチャネルではセンサはスキャンされなくなります。

C プロトタイプ

```
void  CSD2X_ScanSensor(BYTE bSensor);
```

ダブルチャネル構成の場合：

```
 void  CSD2X_ScanSensor(BYTE bSensorLeft, byte bSensorRight);
```

アセンブラ

```
 mov  A,  bSensor
lcall  CSD2X_ScanSensor
```

ダブルチャネル構成の場合：

```
 mov  A,  bSensorLeft
mov  X,  bSensorRight
lcall  CSD2X_ScanSensor
```

パラメータ：

A => センサ番号

ダブルチャネル構成の場合：

A => 左のセンサ番号

X => 右のセンサ番号

戻り値：

なし

副作用

**

CSD2X_ScanAllSensors

説明：

各センサインデックスについて CSD2X_ScanSensor() を呼び出すことで、構成済み全センサをスキャンします。

C プロトタイプ

```
void  CSD2X_ScanAllSensors();
```

アセンブラ

```
lcall  CSD2X_ScanAllSensors
```

パラメータ：

なし

戻り値：

なし

副作用

**

CSD2X_UpdateSensorBaseline

説明：

この経時的カウント値は、センサ別に独立して計算され、センサの基底 (Baseline) 値と呼ばれます。基底 (Baseline) 値はバケツメソッドを用いて更新されます。

バケツメソッドは、次のアルゴリズムを使用します。

1. CSD2X_UpdateSensorBaseline() が呼び出されるたびに、生 (Raw) カウント値を前回の基底 (Ｂａｓｅｌｉｎｅ) 値から引いて差 (Difference) 値を計算します。この差 (Difference) 値は CSD2X_waSnsDiff[] アレイに保存され、表示されます。
2. センサ Autoreset が無効な場合、CSD2X_UpdateSensorBaseline() が呼び出されるたびに、差 (Difference) 値をノイズ閾値と比較します。差 (Difference) 値がノイズ閾値より小さい場合、仮想バケツに入れられます。差 (Difference) 値がノイズ閾値より大きい場合、バケツは更新されません。センサ Autoreset が有効な場合、ノイズ閾値パラメータに関わらず、差 (Difference) 値は仮想バケツに集積されます。
3. 仮想バケツで集積された差 (Difference) 値のカウントが BaselineUpdateThreshold 値に達すると、基底値 (Baseline) は 1 増分され、バケツは 0 にリセットされます。
4. 差 (Difference) 値がノイズ閾値より小さい場合、waSnsDiff[] アレイに保持されている値が 0 にリセットされます。従って、このアレイには 0 より大きく NoiseThreshold より小さい値を持つ成分は含まれません。

C プロトタイプ

```
void  CSD2X_UpdateSensorBaseline(BYTE bSensor)
```

アセンブラ

```
 mov  A,  bSensor
lcall  CSD2X_UpdateSensorBaseline
```

パラメータ：

A => センサ番号

戻り値：

なし

副作用：

**

CSD2X_UpdateAllBaselines

説明：

CSD2X_bUpdateSensorBaseline() 関数を使用して、すべてのセンサの基底値を更新します。

C プロトタイプ

```
void  CSD2X_UpdateAllBaselines()
```

アセンブラ

```
lcall  CSD2X_UpdateAllBaselines
```

パラメータ：

なし

戻り値：

なし

副作用：

**

CSD2X_bIsSensorActive

説明：

センサの差（Difference）値カウントアレイを確認し、指閾値と比較します。ここではヒステリシスを考慮します。ヒステリシス値は、センサが現在オンであるか否かに基づいて、指閾値を足したり引いたりします。アクティブ時の場合、閾値は下げられます。アクティブ時でない場合、閾値は上げられます。この関数は、CSD2X_baSnsOnMask[] アレイでセンサのビットを更新します。

C プロトタイプ

```
BYTE  CSD2X_bIsSensorActive(BYTE bSensor)
```

アセンブラ

```
mov  A,  bSensor
lcall  CSD2X_bIsSensorActive
```

パラメータ：

bSensor A => センサ番号

戻り値：

戻り値は、アクティブの場合 1 で、アクティブでない場合は 0 です。

A => 1 – 選択されたセンサがアクティブ。0 – 選択されたセンサがアクティブではない。

副作用：

**

CSD2X_bIsAnySensorActive

説明：

全センサの Ｄｉｆｆｅｒｅｎｃｅ カウントアレイを確認し、指閾値と比較します。各センサについて CSD2X_bIsSensorActive() を呼び出し、CSD2X_baSnsOnMask[] アレイが関数呼び出し後に最新の状態であるようにします。

C プロトタイプ

```
BYTE  CSD2X_bIsAnySensorActive()
```

アセンブラ

```
lcall  CSD2X_bIsAnySensorActive
```

パラメータ：

なし

戻り値：

戻り値は、アクティブの場合 1 で、アクティブでない場合は 0 です。

A => 1 – 1 つ以上のセンサがアクティブ。0 – アクティブのセンサなし。

副作用：

**

CSD2X_wGetCentroidPos

説明：

差 (Difference) 値アレイを確認し、セントロイドを探します。それが存在する場合は、オフセットと長さが一時変数に保存され、セントロイド位置が CSD2X ウィザードで指定された分解能に計算されます。この関数は、スライダが CSD2X ウィザードで指定されている場合のみ利用できます。

C プロトタイプ

```
WORD  CSD2X_wGetCentroidPos(BYTE bSnsGroup)
```

アセンブラ

```
 mov  A,  bSnsGroup
lcall  CSD2X_wGetCentroidPos
```

パラメータ：

bSnsGroup A => グループ番号

このパラメータは、スライダとして使用されるセンサグループへのレファレンスです。グループ 0 はボタン用です。スライダはグループ 1 以降に含まれています。

戻り値：

スライダの位置値は LSB は A に、MSB は X にあります。

副作用：

このルーチンは、ノイズ閾値を引くことによって差 (Difference) 値カウントを調整します。マイナスの差 (Difference) 値となることを避けるために、各スキャン後一度だけ呼び出します。差 (Difference) 値カウント信号をモニターするアプリケーションの場合、Ｄｉｆｆｅｒｅｎｃｅ カウントデータ転送後にこのルーチンを呼び出します。

スライダセンサが一つでも動作中の場合、関数はゼロから CSD2X ウィザードで設定された分解能値を返します。アクティブのセンサがない場合、関数は –1 (FFFFh) を返します。セントロイド / ダイプレックスアルゴリズムの実行中にエラーが発生した場合、関数は –1 (FFFFh) を返します。必要に応じて、CSD2X_blsSensorActive() ルーチンを使用してタッチされたスライダセグメントを判定できます。

注意事項：スライダセグメント上のノイズカウントがノイズ閾値より大きい場合、このサブルーチンは偽のセントロイド結果を示すことがあります。ノイズが偽のセントロイド結果を生じないように、ノイズ閾値は慎重に設定します (ノイズレベルを超える高さ)。

CSD2X_wGetRadialPos

説明：

差 (Difference) アレイを確認し、セントロイドを探します。それが存在する場合、セントロイド位置を CSD2X ウィザードに指定された分解能に計算します。この関数は、CSD2X ウィザードで指定されているラジアルスライダの場合のみ利用できます。

C プロトタイプ

```
WORD  CSD2X_wGetRadialPos(BYTE bSnsGroup)
```

アセンブラ

```
 mov  A,  bSnsGroup
lcall  CSD2X_wGetRadialPos
```

パラメータ：

bSnsGroup A => スライダ番号

このパラメータは、使用中のラジアル スライダの数です。この番号は、ラジアルスライダ値の左にある CSD2X ユーザモジュールウィザードを通して得ることができます (例えば s2 の場合、ラジアルスライダ番号は 2)。

戻り値：

ラジアル スライダの位置の値は LSB は A、MSB は X です。

副作用：

マイナスの差 (Difference) 値と基底 (Baseline) 値の更新を避けるために、各スキャン後一度だけ呼び出します。差 (Difference) カウント信号をモニターするアプリケーションの場合、差 (Difference) カウントデータ転送後にこのルーチンを呼び出します。

スライダセンサが一つでも動作中の場合、関数はゼロから CSD2X ウィザードで設定された分解能値を返します。アクティブのセンサがない場合、関数は –1 (FFFFh) を返します。

注 スライダセグメント上のノイズカウントがノイズ閾値より大きい場合、このサブルーチンは正しくないセントロイド結果を示すことがあります。ノイズが正しくないセントロイド結果を生じないように、ノイズ閾値は慎重に設定します (ノイズレベルを超える高さ)。

CSD2X_wGetRadialInc

説明：

実際の指シフト値を返します。これは現在と前回の指位置の差です。この関数は CSD2X_wGetRadialPos() とともに機能し、後者が生成したデータを利用します (データは内部変数に保存されます)。

C プロトタイプ

```
WORD  CSD2X_wGetRadialInc(BYTE bSnsGroup)
```

アセンブラ

```
 mov  A,  bSnsGroup
lcall  CSD2X_wGetRadialInc
```

パラメータ：

bSnsGroup A => スライダ番号

このパラメータは、使用中のラジアル スライダの数です。この番号は、ラジアルスライダ値の左にある CSD2X ユーザモジュールウィザードを通して得ることができます (例えば s2 の場合、ラジアルスライダ番号は 2)。

戻り値：

指シフト値。時計回りはプラス、反時計周りはマイナス。LSB は A、MSB は X。

指シフト値は現在と前回の指位置の差です。前回スキャンの間にタッチがなかった場合 (前回 CSD2X_wGetRadialPos() が 1 (FFFFh) を返した)、または今回タッチがない場合 (今回 CSD2X_wGetRadialPos() が -1 (FFFFh) を返した)。

副作用：

このルーチンは CSD2X_wGetRadialPos() API の後に呼び出されなければなりません。これは、CSD2X_wGetRadialPos(). によってセットされる内部データ CSD2X_waSliderPrevPos と CSD2X_waSliderCurrPos を使用するためです。

CSD2X_InitializeSensorBaseline

説明：

選択されたセンサをスキャンして、初期値を伴う CSD2X_waSnsBaseline[bSensor] アレイを読み込みます。Raw カウント値は、選択されたセンサの Baseline 値の配列エレメントにコピーされます。この関数は、個々のセンサの基底値をリセットするために使用されます。

C プロトタイプ

```
 void  CSD2X_InitializeSensorBaseline(BYTE bSensor)
```

アセンブラ

```
 mov  A,  bSensor
lcall  CSD2X_InitializeSensorBaseline
```

パラメータ：

A => センサ番号

戻り値：

なし

副作用：

**

CSD2X_InitializeBaselines

説明：

それぞれのセンサをスキャンして、CSD2X_waSnsBaseline[] アレイに初期値をロードします。Raw カウント値は、それぞれのセンサの基底値のアレイにコピーされます。

C プロトタイプ
```
void  CSD2X_InitializeBaselines()
```
アセンブラ
```
lcall  CSD2X_InitializeBaselines
```
パラメータ：

なし

戻り値：

なし

副作用：

**

CSD2X_SetDefaultFingerThresholds

説明：

FingerThreshold パラメータ値を伴う CSD2X_baBtnFThreshold[] アレイを読み込みます。CSD2X_baBtnFThreshold[] アレイがカスタム値とともにマニュアルで読み込まれていない場合、この関数はスキャン前に呼び出さなければなりません。

C プロトタイプ
```
void  CSD2X_SetDefaultFingerThresholds()
```
アセンブラ
```
lcall  CSD2X_SetDefaultFingerThresholds
```
パラメータ：

なし

戻り値：

なし

副作用：

**

CSD2X_SetLeftDACValue

説明：

この関数はユーザ モジュールのパラメータ設定の左の IDAC 値を上書きします。異なる iDAC 設定でスキャンする必要のあるセンサがある場合に、これを使用します。この関数は、CSD2X_ScanSensor(). と共に使用でき、Rb 構成では利用できません。

C プロトタイプ
```
void  CSD2X_SetLeftDACValue(BYTE bIdacValue);
```
アセンブラ
```
mov  A,  bIdacValue
lcall  CSD2X_SetLeftDACValue
```
パラメータ：

bIdacValue - iDAC 値をセットします。許可されている値は 1..255 です。

戻り値：

なし

副作用：

**

CSD2X_SetRightDACValue

説明：

この関数はユーザ モジュールのパラメータ設定の右の IDAC 値を上書きします。異なる IDAC 設定でスキャンする必要のあるセンサがある場合に、これを使用します。この関数は、CSD2X_ScanSensor(). と共に使用でき、Rb 構成では利用できません。

C プロトタイプ

```
void  CSD2X_SetRightDACValue(BYTE bIdacValue);
```

アセンブラ

```
mov  A,  bIdacValue
lcall  CSD2X_SetRightDACValue
```

パラメータ：

bIdacValue - iDAC 値をセットします。許可されている値は 1..255 です。

戻り値：

なし

副作用：

**

CSD2X_SetIdacValue

説明：

この関数は、IDAC と補正 IDAC 値用のユーザ モジュール設定を上書きします。異なる IDAC 設定でスキャンする必要のあるセンサがある場合に、これを使用します。この関数は、CSD2X_ScanSensor(). とともに使用でき、シングルチャネル IDAC 構成でのみ利用できます。

C プロトタイプ

```
void  CSD2X_SetIdacValue(BYTE bIDACVal, BYTE bCompIDACVal);
```

アセンブラ

```
mov  A,  bIDACVal
mov  X,  bCompIDACVal
lcall  CSD2X_SetIdacValue
```

パラメータ：

bIDACVal - IDAC 値をセットします。許可されている値は 1..255 です。

bCompIDACVal - 補正 IDAC 値をセットします。許可されている値は 0..255 です。

戻り値：

なし

副作用：

**

CSD2X_SetRefValue

説明：

この関数はユーザ モジュールのパラメータ設定のリファレンス値を上書きします。異なるリファレンス設定でスキャンする必要のあるセンサがある場合に、これを使用します。この関数は、CSD2X_ScanSensor(). とともに使用でき、Rb 構成では何の効果もありません。基準値には Vbg を設定するか、外部ピンからのものを採用します。その理由は、Vbg リファレンス電圧や外部ピンからのリファレンス電圧は実行時に調整できないからです。

C プロトタイプ

シングルチャネル構成の場合：

```
 void  CSD2X_SetRefValue(BYTE bRefValue);
```

ダブルチャネル構成の場合：

```
 void  CSD2X_SetLeftRefValue(BYTE bLeftRefValue);
void  CSD2X_SetRightRefValue(BYTE bRightRefValue);
```

アセンブラ

シングルチャネル構成の場合：

```
 mov  A,  bRefValue
lcall  CSD2X_SetRefValue
```

ダブルチャネル構成の場合：

```
 mov  A,  bLeftRefValue
lcall  CSD2X_SetLeftRefValue
mov  A,  bRightRefValue
lcall  CSD2X_SetRightRefValue
```

パラメータ：

bRefValue - 基準値をセットします。許可されている値は 0..31 です。

戻り値：

なし

副作用：

**

CSD2X_SetRefSource

説明：

この関数はユーザ モジュールのパラメータ設定のリファレンス源を上書きします。異なるリファレンス設定でスキャンする必要のあるセンサがある場合に、これを使用します。この関数は CSD2X_ScanSensor(). とともに使用できます。

C プロトタイプ

IDAC 構成を伴うすべてのシングルチャネルとダブルチャネルの場合：

```
 void  CSD2X_SetRefSource(BYTE bRefSource);
```

Rb 構成を伴うダブルチャネル構成の場合：

```
 void  CSD2X_SetLeftRefSource(BYTE bLeftRefSource);
void  CSD2X_SetRightRefSource(BYTE bRightRefSource);
```

アセンブラ

IDAC 構成を伴うすべてのシングルチャネルとダブルチャネルの場合：

```
 mov  A,  bRefSource
lcall  CSD2X_SetRefSource
```

Rb 構成を伴うダブルチャネル構成の場合：

```
 mov  A,  bLeftRefSource
lcall  CSD2X_SetLeftRefSource
mov  A,  bRightRefSource
lcall  CSD2X_SetRightRefSource
```

パラメータ：

bRefSource - リファレンスソース定数。以下のテーブルに、許可されている値が一覧表示されています。

定数（Rb 構成用）	値
CSD2X_REFERENCE_ACOLUMN_MUX	0x01
CSD2X_REFERENCE_VBG	0x03
CSD2X_REFERENCE_ASE	0x04

定数（IDAC 構成用）	値
CSD2X_REFERENCE_VDD	0x00
CSD2X_REFERENCE_2VBG	0x10

戻り値：

なし

副作用：

**

CSD2X_SetIdacRange

説明：

この関数はユーザ モジュールのパラメータ設定の DAC 範囲を上書きします。異なる範囲設定でスキャンする必要のあるセンサがある場合に、これを使用します。この関数は、CSD2X_ScanSensor(). と共に使用でき、Rb 構成では利用できません。

C プロトタイプ

```
void  CSD2X_SetIdacRange(const bRange);
```

アセンブラ

```
mov  A,  bRange
lcall  CSD2X_SetIdacRange
```

パラメータ：

bRange - 基準値をセットします。許可されている値は次に挙げる定数です。

設定	値	効果
CSD2X_IDAC_RANGE_1X	0x00	最大 IDAC 電流値は 19.92 µA です。
CSD2X_IDAC_RANGE_4X	0x01	最大 IDAC 電流値は 91.03 µA です。
CSD2X_IDAC_RANGE_16X	0x08	最大 IDAC 電流値は 318.75 µA です。
CSD2X_IDAC_RANGE_32X	0x09	最大 IDAC 電流値は 637.50 µA です。

戻り値：

なし

副作用：

**

CSD2X_Calibrate

説明：

この関数は CDS2X 自動キャリブレーションを実行します。AutoCalibration が有効なとき、CSD2X_Start() 関数のユーザ モジュール起動の際に、この関数は自動的に呼び出されます。この関数は Rb 構成では利用できません。

C プロトタイプ

```
void  CSD2X_Calibrate(void);
```

アセンブラ

```
lcall  CSD2X_Calibrate
```

パラメータ：

なし

戻り値：

なし

副作用：

**

CSD2X_ClearSensors

説明：

各センサに対して CSD2X_wGetPortPin() と CSD2X_DisableSensor() を続けて呼び出し、すべてのセンサをノンサンプリング状態にクリアします。

C プロトタイプ

```
void  CSD2X_ClearSensors()
```

アセンブラ

```
lcall  CSD2X_ClearSensors
```

パラメータ：

なし

戻り値：

なし

副作用：

**

CSD2X_wReadSensor

説明：

A (LSB) と X (MSB) で主要な Raw スキャン値を返します。

C プロトタイプ

```
WORD  CSD2X_wReadSensor(BYTE bSensor)
```

アセンブラ

```
mov  A,  bSensor
lcall  CSD2X_wReadSensor
```

パラメータ：

A => センサ番号

戻り値：

センサのスキャンです。A では LSB、X では MSB。

副作用：

**

CSD2X_wGetPortPin

説明：

特定のセンサのポート番号とピンマスクを返します。渡されたパラメータは、CSD2X_Sensor_Table[]. からのデータをインデックスし、選択します。戻り値は、CSD2X_EnableSensor()、CSD2X_DisableSensor(). へと渡すことができます。この関数はシングルチャネル構成でのみ利用できます。

C プロトタイプ

```
WORD  CSD2X_wGetPortPin(BYTE bSensor)
```

アセンブラ

```
mov  A,  bSensor
lcall  CSD2X_wGetPortPin
```

パラメータ：

bSensorNumber – 範囲は 0 ~ (n – 1) です。ここで n は、CSD2X ウィザードにセットされたセンサ数とスライダに含まれているセンサ数の合計です。センサ番号は、選択されたアクティブなセンサのポートとビットマスクを判定するために、CSD2X_wGetPortPin() によって使用されます。

戻り値：

A => センサのビットマップ

X => ポート番号

副作用：

**

CSD2X_wGetPortPinLeft

説明：

左チャネルに接続されているセンサのポート番号とピンマスクを返します。渡されたパラメータは、CSD2X_Sensor_Table_Left[]. からのデータをインデックスし、選択します。戻り値は、CSD2X_EnableSensor()、CSD2X_DisableSensor(). へと渡すことができます。この関数はダブルチャネル構成でのみ利用できます。

C プロトタイプ

```
WORD  CSD2X_wGetPortPinLeft(BYTE bSensor)
```

アセンブラ

```
mov  A,  bSensor
lcall  CSD2X_wGetPortPinLeft
```

パラメータ：

bSensor – センサ番号、範囲は 0 から (n – 1) 。この場合、'n' は左チャネルの CSD2X ウィザードに設定された合計センサ数 (左チャネルに接続されたスライダセグメント数も含む)。センサ番号は、選択されたアクティブなセンサのポートとビットマスクを判定するために、CSD2X_wGetPortPinLeft() によって使用されます。

戻り値：

A => センサのビットマップ

X => ポート番号

副作用：

**

CSD2X_wGetPortPinRight

説明：

右チャネルに接続されているセンサのポート番号とピンマスクを返します。渡されたパラメータは、CSD2X_Sensor_Table_Right[]. からのデータをインデックスし、選択します。戻り値は、CSD2X_EnableSensor()、CSD2X_DisableSensor(). へと渡すことができます。この関数はダブルチャネル構成でのみ利用できます。

C プロトタイプ

```
WORD  CSD2X_wGetPortPinRight(BYTE bSensor)
```

アセンブラ

```
mov  A,  bSensor
lcall  CSD2X_wGetPortPinRight
```

パラメータ：

bSensor – センサー番号、範囲は 0 から (n – 1) 。この場合、'n' は右チャネルの CSD2X ウィザードに設定された合計センサー数 (右チャネルに接続されたスライダセグメント数も含む)。センサ番

号は、選択されたアクティブなセンサのポートとビットマスクを判定するために、CSD2X_wGetPortPinRight() によって使用されます。

戻り値：

A => センサのビットマップ

X => ポート番号

副作用：

**

CSD2X_EnableSensor

説明：

次の測定サイクルで測定するために選択されたセンサを構成します。ポートとセンサは、CSD2X_wGetPortPin() 関数を使用して選択できます。このとき、ポート番号とセンサビットマスクはそれぞれ X と A に読み込まれています。選択されたポートとピンを Analog High Z (アナログ高インピーダンス) モードにし、正しい Analog Mux Bus (アナログ多重化バス) 入力を可能にするために、駆動モードを調整します。これによりコンパレータ機能も有効になります。

C プロトタイプ

```
void  CSD2X_EnableSensor(BYTE bMask, BYTE bPort)
```

アセンブラ

```
mov  X,  bPort
mov  A,  bMask
lcall  CSD2X_EnableSensor
```

パラメータ：

A => センサのビットマップ

X => ポート番号

戻り値：

なし

副作用：

**

CSD2X_DisableSensor

説明：

CSD2X_wGetPortPin() 関数により選択されたセンサを無効にします。駆動モードは、Strong (001) に変更されます。これにより、センサが効果的にグランド接続されます。ポートピンから Analog-MuxBus までの接続はオフになります。この関数パラメータは、CSD2X_wGetPortPin() 関数により返されます。

C プロトタイプ

```
void  CSD2X_DisableSensor(BYTE bMask, BYTE bPort)
```

アセンブラ

```
mov  X,  bPort
mov  A,  bMask
lcall  CSD2X_DisableSensor
```

パラメータ：

A => センサのビットマップ

X => ポート番号

戻り値：

なし

副作用：

**

ファームウェア ソースコードの例

例 1。このコードは、ユーザ モジュールを開始し、センサを連続的にスキャンします。通信セクションは、PC チャーティングツールに値を転送するために使用できます。

```
//-------------------------------------------------------------------------
// Sample C code for the CSD2X module
// Scanning all sensors continuously
//-------------------------------------------------------------------------
#include <m8c.h> // part specific constants and macros
#include "PSoCAPI.h" // PSoC API definitions for all User Modules
void main(void)
{
M8C_EnableGInt;
CSD2X_Start();
CSD2X_InitializeBaselines() ; //scan all sensors first time, init baseline
CSD2X_SetDefaultFingerThresholds() ;
//
// Loop Forever
//
while (1) {
CSD2X_ScanAllSensors(); //scan all sensors in array (buttons and sliders)
CSD2X_UpdateAllBaselines(); //Update all baseline levels;
//detect if any sensor is pressed
if(CSD2X_bIsAnySensorActive()){
// Add user code here to proceed the sensor touching
}
// now we are ready to send all status variables to chart program
// communication here
//
// OUTPUT CSD2X_waSnsResult[x] <- Raw Counts
// OUTPUT CSD2X_waSnsDiff[x] <- Difference
// OUTPUT CSD2X_waSnsBaseline[x] <- Baseline
// OUTPUT CSD2X_baSnsOnMask[x] <- Sensor On/Off
}
}
```

例 2。次のコードは、複数のセンサを並列に接続し、同時に CSD2X_ScanSensor() 関数を呼び出すことによってそれらをスキャンする能力を示しています。このサンプルは、接触があったセンサを区別することなくセンサのタッチを検知する必要がある場合に有用です。これは、デバイスウェークアップ検知と、バッテリの エネルギーを節約するためのスキャン時間の短縮に利用できます。ウェークアップタッチを検知した後は、各センサを従来型の個別スキャンに戻せます。

```
 //--------------------------------------------------------------------------
// Sample C code for the CSD2X module
// Scan several sensors in parallel
//--------------------------------------------------------------------------
#include <m8c.h> // part specific constants and macros
#include "PSoCAPI.h" // PSoC API definitions for all User Modules
void main(void)
{
M8C_EnableGInt;
CSD2X_Start();
CSD2X_SetDefaultFingerThresholds();
// Enable the sensor connected to P1[4]
CSD2X_EnableSensor(0x10, 1);
// Enable the sensor connected to P1[6]
CSD2X_EnableSensor(0x40, 1);
// Enable the sensor connected to P3[0]
CSD2X_EnableSensor(0x01, 3);
// Initialize baseline for sensor number "3"
CSD2X_InitializeSensorBaseline(3);
while (1) {
// Scan continuously sensor number "3" which is connected
//in parallel to the enabled above sensors
CSD2X_ScanSensor(3);
// CSD2X_ScanSensor(0xFF, 3); // for Double channel configuration
CSD2X_UpdateSensorBaseline(3);
if(CSD2X_bIsSensorActive(3)){
// Add user code here to proceed the buttons pressing
}
}
}
```

例 3。次の例は、各センサについて異なる指閾値レベルを設定する能力を示しています。異なるセンサが異なる場所に配置されており、一部のセンサが他のセンサより検出感度が高い場合に有効です。

```
 //--------------------------------------------------------------------------
// Sample C code for the CSD2X module
// Set individual finger threshold parameter for each sensor
//--------------------------------------------------------------------------
#include <m8c.h> // part specific constants and macros
#include "PSoCAPI.h" // PSoC API definitions for all User Modules
void main(void)
{
M8C_EnableGInt;
CSD2X_Start();
CSD2X_InitializeBaselines();
// set finger threshold for sensor "0"
CSD2X_baBtnFThreshold[0] = 10;
```

```
// set finger threshold for sensor "1"
CSD2X_baBtnFThreshold[1] = 20;
// set finger threshold for sensor "2"
CSD2X_baBtnFThreshold[2] = 30;
// set finger threshold for sensor "3"
CSD2X_baBtnFThreshold[3] = 40;
// set finger threshold for sensor "4"
CSD2X_baBtnFThreshold[4] = 50;
// set finger threshold for sensor "5"
CSD2X_baBtnFThreshold[5] = 255;
// set finger threshold for sensor "6"
CSD2X_baBtnFThreshold[6] = 200;
while (1) {
// Scan continuously all sensors
CSD2X_ScanAllSensors();
CSD2X_UpdateAllBaselines();
//detect if any sensor is pressed
if(CSD2X_bIsAnySensorActive()){
// Add user code here to proceed the buttons pressing
}
}
}
```

コンフィグレーション レジスタ

Table 4. ブロック CapSense、レジスタ : CS_CR0

ビット	7	6	5	4	3	2	1	0
値	0	0	CSD2X_PRSCLK	0	1	0	0	EN

Table 5. ブロック CapSense、レジスタ : CS_CR1

ビット	7	6	5	4	3	2	1	0
値	1	スキャン速度		0	0	0	0	0

Power : 0x01 アナログブロックの電源をオンにします。0x00 アナログブロックの電源をオフにします。

Table 6. ブロック CapSense、レジスタ : CS_CR2

ビット	7	6	5	4	3	2	1	0
値	1	0	0	0	0	1	0	0

Table 7. ブロック CapSense、レジスタ : CS_CR3

モード / ビット	7	6	5	4	3	2	1	0
値	0	1	1	1	0	0	0	0

Table 8. ブロック CapSense、レジスタ : CS_CNTH

ビット	7	6	5	4	3	2	1	0
データ出力 MSB								

Table 9. ブロック CapSense、レジスタ : CS_CNTL

ビット	7	6	5	4	3	2	1	0
データ出力 LSB								

Table 10. ブロック CapSense、レジスタ : PRS_CR

モード / ビット	7	6	5	4	3	2	1	0
値	1	0	8/12 ビット	1	Prescaler (プリスケーラ)			

Table 11. ブロックタイマ、レジスタ : PT1_CFG

モード / ビット	7	6	5	4	3	2	1	0
値	0	0	0	0	0	0	1	Start (開始)

Table 12. ブロックタイマ、レジスタ : PT1_DATA0

モード / ビット	7	6	5	4	3	2	1	0
値	データ LSB							

Table 13. ブロックタイマ、レジスタ : PT1_DATA1

モード / ビット	7	6	5	4	3	2	1	0
値	データ MSB							

付属資料

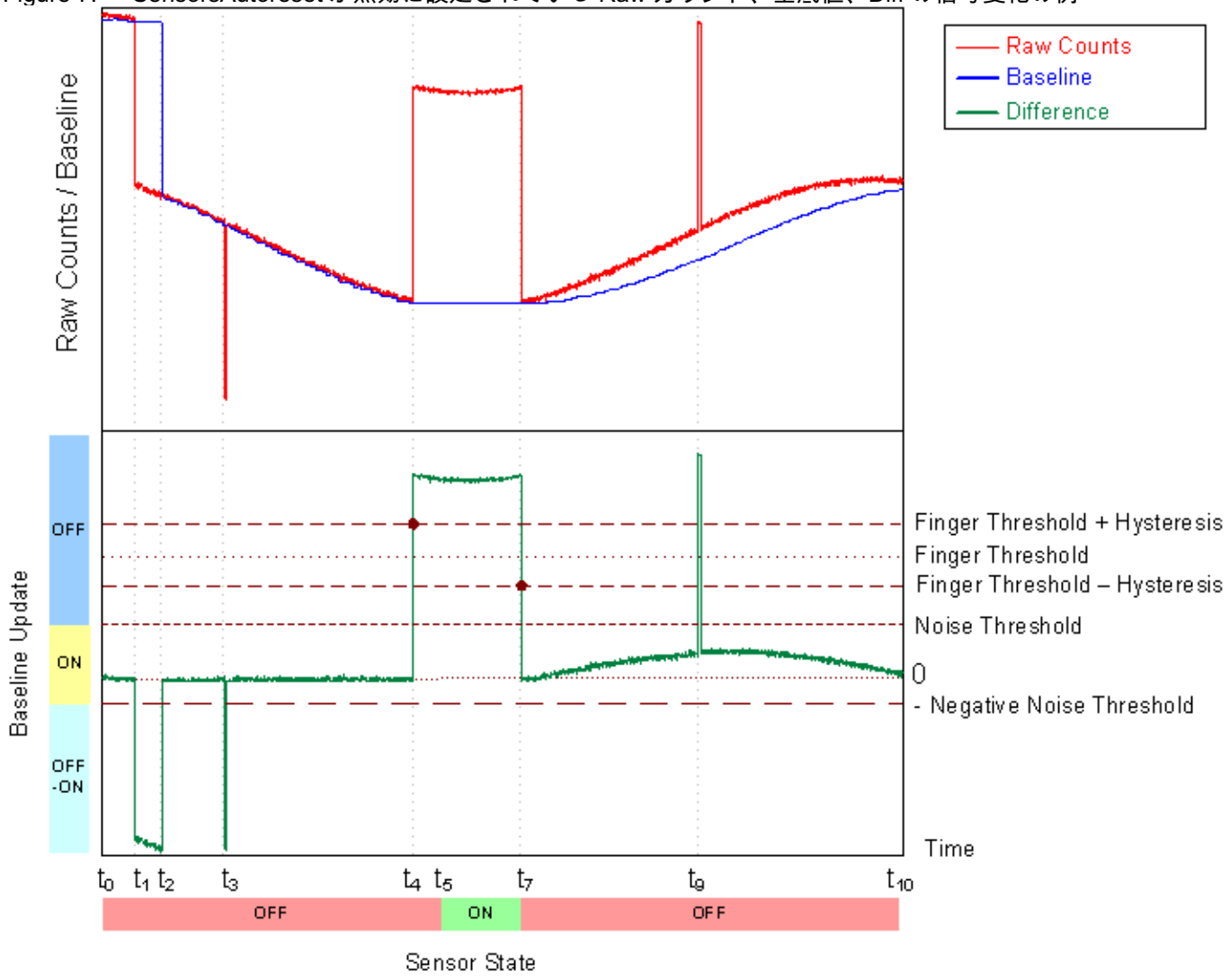
ここからのセクションには、ユーザ モジュールデータシートに通常含まれている以外の情報が記載されています。これらの詳細情報は、ユーザによる CapSense アプリケーションの設計を支援するために、サイプレスのエンジニアが開発したものです。情報の一部は、将来のアプリケーションノートに組み込まれる可能性もあります。

CSD2x パラメータのインタラクション

以下の図は基底値の更新と判定論理演算を示しており、最適なパフォーマンスを実現するための ユーザ モジュール パラメータの設定方法を理解するうえで役立ちます。最初の図は、Sensors Autoreset パラ

メータが [Disabled] (無効) に設定されている場合のシステム動作を示しています。2 番目の図は、Sensors Autoreset パラメータが [Enabled] (有効) に設定されている場合を示しています。指閾値、ノイズ閾値、ヒステリシス、ネガティブノイズ閾値が、差 (Difference) 値の信号とともに示されています (Raw カウント – Baseline)。データは、システム動作を実演する人工的なテストで、低速と高速の R a w カウント変化の際に収集されました。低速の変化は温度や湿度の変化などによって、また高速の変化はセンサタッチ、ESD イベント、または強い RF フィールドの影響などによって発生します。

Figure 7. SensorsAutoreset が無効に設定されている Raw カウント、基底値、Diff の信号変化の例



湿度や気温の変化により、基底値レベルに近い Raw カウントは t_0 で次第に速度を落とし始めます。2 つの連続した変換の間に起きる R a w カウントの変化は NegativeNoiseThreshold パラメータを上回らないため (絶対値)、Raw カウントの最小値をトラッキングし、Raw カウント信号のうち低い値を維持することにより、基底値は更新されます。

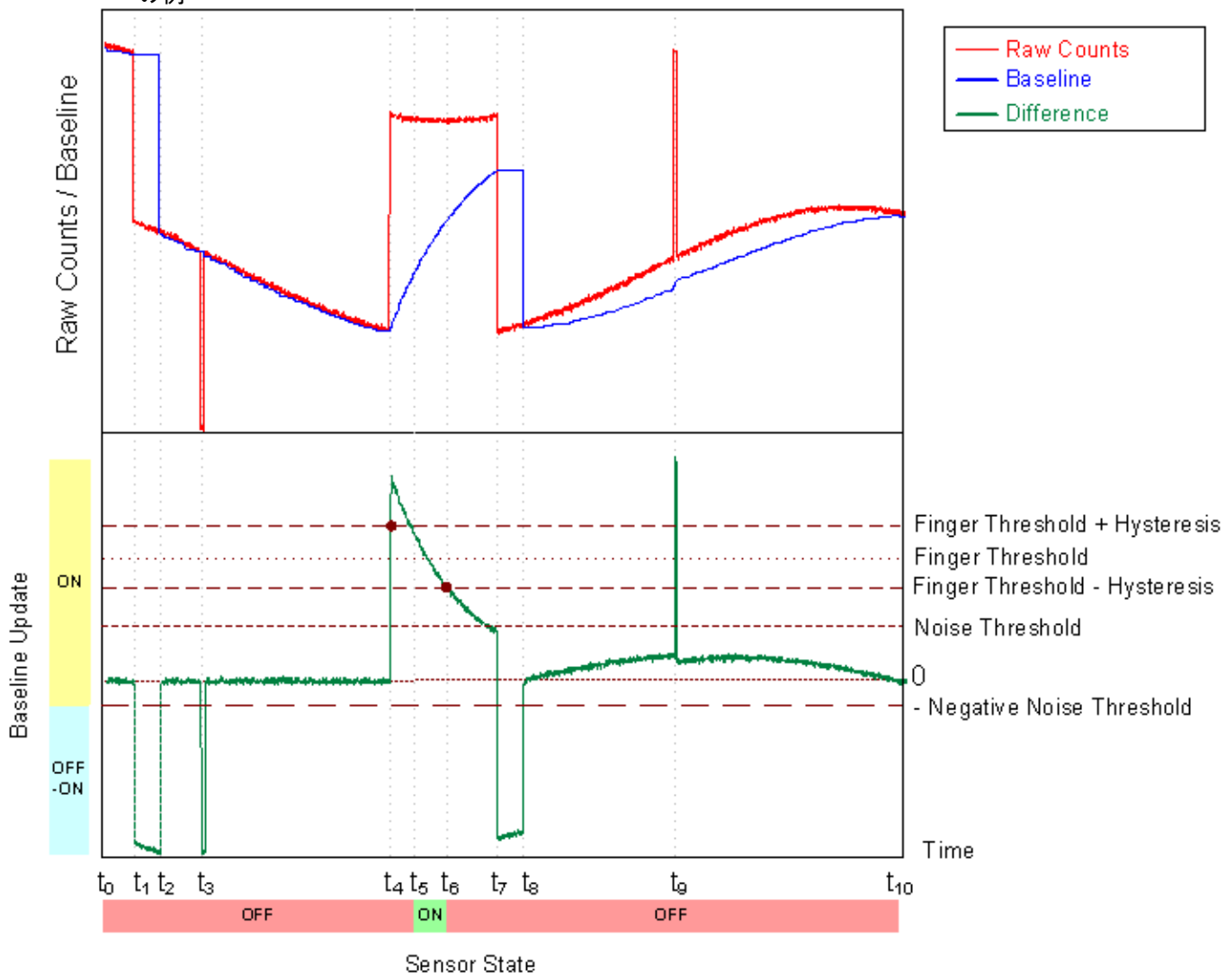
t_1 で Raw カウントは大幅に落下し、マイナスの差が NegativeNoiseThreshold を超えます。この状況は、指がセンサに接触しているときにデバイスの電源がオンになり、ある程度時間が経ってから指を離した場合に発生します。このとき、基底値更新のメカニズムが停止し、内部のタイムアウトカウンタが起動します。基底値は、LowBaselineReset のサンプル数の間、Difference 信号が NegativeNoiseThreshold を下回ったときにリセットされます。この状況は t_2 で起きます。

2 番目に大きなマイナスの差 (Difference) 値信号スパイクは、t_3 で発生します。このスパイクが発生する原因は ESD イベントなどです。サンプルカウント中のスパイクが LowBaselineReset パラメータを下回るため、基底値は待機状態となり、スパイクはフィルタにかけられます。これにより、正しくない基底値リセットと、その結果である正しくないタッチ検知を防ぐことができます。

センサは t_4 でタッチされています。差 (Difference) 値信号が FingerThreshold + ヒステリシス値を超えている場合、内部デバウンスカウンタが起動します。信号がこの値をデバウンスサンプル以上に超えると、センサの状態がオンにセットされます。これは t_5 で起きます。差 (Difference) 値信号が t_7 で FingerThreshold + ヒステリシスレベルを下回ると、センサの状態はすぐにオフに戻ります。t_9 の短く正側のスパイクは、デバウンスカウンタでフィルタされます。これは、サンプルユニットにおけるスパイク期間がデバウンス値を超えないためです。

生カウントは、t_7 と t_{10} の間、ゆっくりと上昇します。Difference 信号が NoiseThreshold を下回っている場合、バケツアルゴリズムを用いて基底値を更新します (SensorsAutoreset は Disabled (無効) に設定)。Difference 信号はドリフトレートに比例します。BaselineUpdate 閾値パラメータを用いて基底値更新の速度を制御することができます。パラメータ値を低く設定すると、基底値更新速度が速くなります。

Figure 8. SensorsAutoreset が有効に設定されている Raw カウント、Baseline 値、差 (Difference) 値信号変化の例



上の図のシステム動作は、前述のケースの動作に似ていますが、以下のような違いがあります。

- センサがタッチされている間は、動作中の基底値更新アルゴリズムによって、タッチ時間が短縮されます (t_6)。
- 指を離すと、タッチ検知を短時間阻止している LowBaselineReset サンプル (t_8) 後に、基底値がリセットされます。これは、もう一つのデバウンスメカニズムとして機能します。

ダブルチャネル スキャニング

ダブルチャネル スキャニングは、同期ブロック機能として行われます。

同期とは、左右のチャネルセンサが同時にスキャンされ、ペアのセンサのスキャンが完了してから、次のペアのスキャンを開始します。左右のセンサで分解能とスキャン時間が異なることもありえます。その場合、「センサスキャン」関数は遅いほうのセンサを待ち、もう一方のセンサには何もしません。左右のセンサアレイでセンサ数が異なる場合、「ScanAllSensors」関数は次の機能を果たします。

- ペアの中の可能なセンサをすべてスキャンする
- 片方のセンサ数が多い場合、そのチャネルの残りのセンサを 1 個ずつスキャンする

スキャンはアレイの最後から始まります。センサの位置は、左上から右下に向うセンサ配置に従って、GUI 中に割り当てられます。下記に、デュアルチャネルのスキャンができるだけ効率的に行われるようにするための注意点を挙げます。

- 各チャネルのセンサ数を同じにする
- スキャン時間が長いセンサ同士がペアになるようにセンサの配置を工夫する
- すべてのスライダセグメントを同じチャネルに配置する
- 大きなセンサを左チャネルに配置する
- 異なる基準値を使用する場合、基準値が高いセンサを初めに配置する

バージョン履歴

バージョン	編集者	説明
1.0	DHA	初期バージョン

バージョン	編集者	説明
2.1	DHA	1. 次のパラメータを CSD2x ウィザードから削除。 IDAC Value (IDAC 値)、Finger Threshold (指閾値)、Reference Value (基準値)、Scan Speed (スキャン速度)、Scanning Resolution (スキャン分解能) 2. ウィザードでのセンサ別スキャンパラメータを個別に設定するオプションを無効化。これにより、RAM と ROM のスペースが節約できます。 3. ユーザ モジュール パラメータウィンドウに、次の新パラメータを追加。 Finger Threshold (指閾値)、Reference Value (Left/Right) (左右の基準値)、IDAC Value (Compensation/Left/Right) (IDAC 値、補正・ 左右)、Scanning Speed (スキャン速度)、Resolution (分解能)。これらのパラメータは、起動時にすべてのセンサに適用されます。 4. 基準値 (左右) パラメータ分解能を 5 ビットに増大。 5. 他の CapSense ユーザ モジュールとの整合性を保つために、次の新しい API 関数を追加。 SetScanMode (左右)、SetPrescaler (左右)、SetRefSource (左右)、SetRefValue (左右)、SetDACValue (左右) 6. すべてのスキャンパラメータを RAM に保存。これにより、実行時に個々のセンサ用の異なるスキャンパラメータが設定できます。 7. 自動キャリブレーション (有効時) が起動時に自動的に作動。CSD2X_Calibrate API をマニュアルで呼び出す必要がなくなりました。 8. 更新された IDAC 値の入力範囲。 9. 基準値と Autocalibration パラメータの記述を更新。 10. PSoC Designer でのパラメータの順序を変更。 11. データシートセクションにリファレンスを追加。 12. SetRefSource API をデータシートに追加。 13. 次の API の記述を更新。 CSD2X_DisableSensor, CSD2X_Start, CSD2X_SetScanMode 14. MUM 構成の記述を変更。 15. 新しい部品番号へのサポートを追加。 16. LowBaselineReset、NegativeNoiseThreshold、Modular Capacitor Pin (モジュラコンデンサピン) の範囲を追加。 17. Finger Threshold (指閾値) パラメータの範囲の問題を修正。 18. SetLeftRefSource API と SetRightRefSource API 用のプロトタイプを追加。 19. その他の欠陥を修正し、ユーザ モジュールのソースコードをクリーンアップ。
2.20	DHA	割り当て無しセンサ用の DRC 警告メッセージを追加。

バージョン	編集者	説明
2.30	DHA	1. Start() API で RAM_EPILOGUE から RAM_USE_CLASS_3 に変更。 2. DiplexTables と Order_Table_Left/Right を AREA lit に移動。 3. ウィザードヘルプボタンとファイルを追加。
2.40	DHA	1. 基底値 の初期化を固定するために、CSD2X_InitializeSensorBaseline()API 関数実践で CSD2X_Order_Table_Left を CSD2X_Order_Table_Right に変更。 2. CSD2X_Start()API 関数で "call @INSTANCE_NAME`_ScanAllSensors" を追加。 3. CSD2x_baDACCodeBaselineL と CSD2x_baDACCodeBaselineR IDAC テーブルのエクスポートを追加。 4. CSD2X_wGetPortPinLeft() と CSD2X_wGetPortPinRight()API 関数の説明を更新。 5. 「フィードバック抵抗ピン」パラメータの説明で「GOO[4]」と「「GOO[5]」の新選択オプションを追加。 6. センサースキャン時間のプレスケーラ値への依存に関する情報を追加。 7. CSDx_CR1 レジスタで ACOL[1:0] ビットフィールド初期化値を 01b に設定。 8. SetIdacRange() API 機能で RAM_SETPAGE_CUR >`@INSTANCE_NAME`_bBitMask を追加。

Note　PSoC Designer 5.1 から、全ユーザ モジュール データシートに改訂履歴を追加しました。このセクションでは、ユーザー モジュールの過去のバージョンと現在のバージョンとの違いについて簡単な解説を掲載しています。

Copyright © 2011-2012 © Cypress Semiconductor Corporation. 本文書に記載される情報は、予告なく変更される場合があります。Cypress Semiconductor Corporation (サイプレス セミコンダクタ社) は、サイプレス 製品に組み込まれた回路以外のいかなる回路を使用することに対しても一切の責任を負いません。特許またはその他の権限下で、ライセンスを譲渡または暗示することもありません。サイプレス 製品は、サイプレスとの書面による合意に基づくものでない限り、医療、生命維持、救命、重要な管理、または安全の用途のために使用することを保証するものではなく、また使用することを意図したものでもありません。さらにサイプレスは、誤動作や故障によって使用者に重大な傷害をもたらすことが合理的に予想される、生命維持システムの重要なコンポーネントとしてサイプレス製品を使用することを許可していません。生命維持システムの用途にサイプレス 製品を供することは、製造者がそのような使用におけるあらゆるリスクを負うことを意味し、その結果サイプレスはあらゆる責任を免除されることを意味します。

PSoC® は Cypress Semiconductor Corp の登録商標であり、PSoC Designer ™ および Programmable System-on-Chip ™ 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。

全てのソース コード (ソフトウェアおよび / またはファームウェア) はサイプレス セミコンダクタ社 (以下「サイプレス」) が所有し、全世界の特許権保護 (米国およびその他の国)、米国の著作権法ならびに国際協定の条項により保護され、かつそれらに従います。サイプレスが本書面によりライセンシーに付与するライセンスは、個人的、非独占的かつ譲渡不能のライセンスであって、適用される契約で指定されたサイプレスの集積回路と併用されるライセンシーの製品のみをサポートするカスタム ソフトウェアおよび / またはカスタムファームウェアを作成する目的に限って、サイプレスのソース コードの派生著作物をコピー、使用、変更そして作成するためのライセンス、ならびにサイプレスのソース コードおよび派生著作物をコンパイルするためのライセンスです。上記で指定された場合を除き、サイプレスの書面による明示的な許可なくして本ソース コードを複製、変更、変換、コンパイル、または表示することは全て禁止されます。

免責条項： サイプレス は、明示的または黙示的を問わず、本資料に関するいかなる種類の保証も行いません。これには、商品性または特定目的への適合性の黙示的な保証か゛含まれますが、これに限定されません。サイプレスは、本文書に記載される資料に対して今後予告なく変更を加える権利を留保します。サイプレスは、本文書に記載されるいかなる製品または回路を適用または使用したことによって生ずるいかなる責任も負いません。サイプレスは、誤動作や故障によって使用者に重大な傷害をもたらすことが合理的に予想される生命維持システムの重要なコンポーネントとしてサイプレス製品を使用することを許可していません。生命維持システムの用途にサイプレス製品を供することは、製造者がそのような使用におけるあらゆるリスクを負うことを意味し、その結果サイプレスはあらゆる責任を免除されることを意味します。

ソフトウェアの使用は、適用されるサイプレス ソフトウェア ライセンス契約によって制限され、かつ制約される場合があります。