CapSense® Sigma-Delta（シグマデルタ）Data Sheet CSD V 1.20

Copyright © 2008-2010 Cypress Semiconductor Corporation. All Rights Reserved.

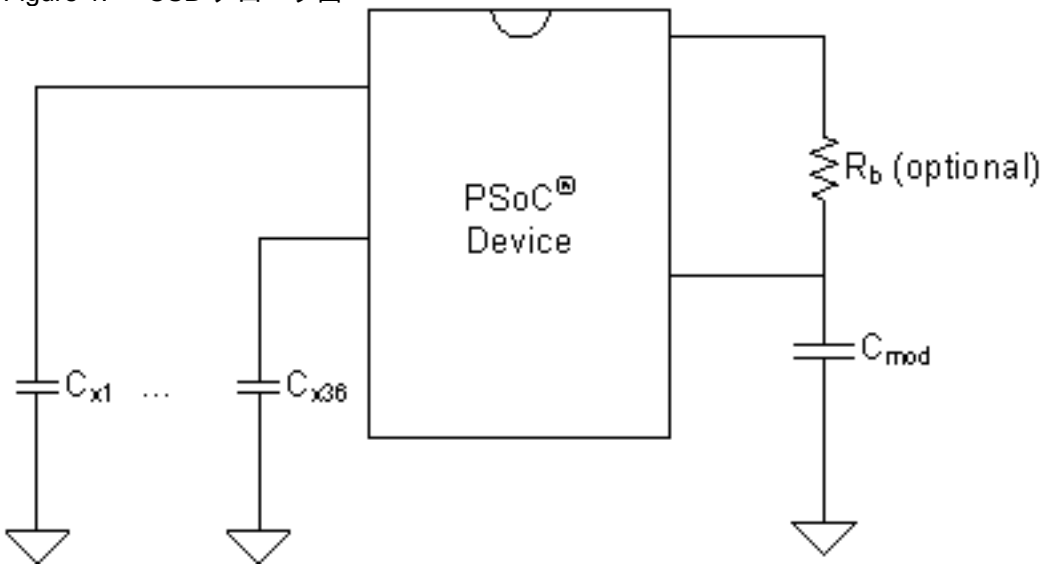
リソース	PSoC® ブロック				API メモリ（バイト）		センサー当たりのピン数
	CapSense®	I²C/SPI	タイマ	コンパレータ	Flash	RAM	
CY8C20x66, CY8C20x36, CY8C20336AN, CY8C20436AN, CY8C20636AN, CY8C20x46, CY8C20x96, CY7C645xx, CY7C643/4/5xx, CY7C60424, CY7C6053x, CYONS2110, CYONSFN2xxx							
	1	-	1	-	1540	35	1

特徴と概要

- 1 ~ 36 個の静電容量式センサをスキャン。
- ガラスオーバーレイで、最大 15mm まで対応
- ワイヤーセンサで、20cm の近接検知。
- AC 電源ノイズや EMC ノイズ、また電源電圧の変化によるノイズ等に対する高耐性。
- 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たスライドセンサの物理的分解能を 2 倍に。
- 補間法を使用してスライドセンサの分解能を高める。
- 2 つのスライドセンサー付きのタッチパッドサポート
- 高い伝導性材料を使ったセンシングをサポート（例：ITO フィルム）。
- 水膜や水滴に対して、シールド電極を使用し、信頼性の高い動作をサポート。
- CSD ウィザードを使用し、簡単にセンサーやピン配置が可能。
- 温度、湿度の変化や、静電気放電（ESD) 時にも基準値を自動アップデートする最新アルゴリズム。
- 調整が簡単な操作パラメータ。
- 生データをモニタリングしながら、リアルタイムでパラメタの最適化を可能にする PC GUI アプリケーションサポート。

CSD（デルタ シグマ変調器を使用した静電容量式検知）は、スイッチドキャパシタを使用し、シグマデルタ変調器を使用して電流値をデジタルコードにすることで、静電容量を検出します。

Figure 1. CSD ブロック図



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（適宜。例：I2C と LCD）。ポートとピンを適宜割り当てます。
2. CSD ユーザ モジュールを選択・配置します。
3. Workspace Explorer の CSD ユーザ モジュールを右クリックして、CSD ウィザードを開きます（ ）ウィザード。
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。すべてのパラメータ記述を読み、条件とガイドラインに準拠します。
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. CSD 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。CapSense アプリケーション用 *信号対ノイズ比条件で説明したように、5:1 の SNR 条件を達成するために、CSD パラメータを調整します* – AN2403.

[+] Feedback

機能説明

静電容量式センサは、物理的コンポーネント、電気的コンポーネント、およびソフトウェアで構成されています。

- 物理的コンポーネント
  - 物理センサ自体は、一般的に PSoC に接続されている PCB で構成されており、ディスプレイは隔絶カバー、軟質膜、または透明なオーバーレイで覆われています。
- 電気的コンポーネント
  - センサの静電容量をデジタル形式に変換する手段です。変換システムは、検知するスイッチド キャパシタ、Sigma-Delta（シグマデルタ）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によって構成されています。
- ソフトウェア
  - 検知と補償ソフトウェアのアルゴリズムが、カウント値をセンサ検知判定に変換します。
  - 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、付属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 を用いて、途中から連結しあう 2 つの静電容量式センサを用いて、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手法は数多くありますが、このユーザ モジュールは、スイッチングキャパシタとデルタシグマ変調方式を用いたものです。

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 CSD 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。

- PSoC CY8C20x66, CY8C20x66A, CY8C20x46/96, CY8C20x46A/96A, CY8C20x36, CY8C20x36A Technical Reference Manual 技術リファレンス マニュアル）セクション : CapSense システム

次のアプリケーションノートは、CSD 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 セミコンダクタ社のウェブサイト（www.cypress.com）から探すことができます。

- *CapSense ベストプラクティス* – AN2394
- *CapSense アプリケーション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 アプリケーション用の EMC 対策デザイン* – AN2318
- *容量検知アプリケーションの電力節約とスリープ機能* – AN2360
- *PSoC CapSense のレイアウトガイドライン* – AN2292
- *ユニバーサル非同時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

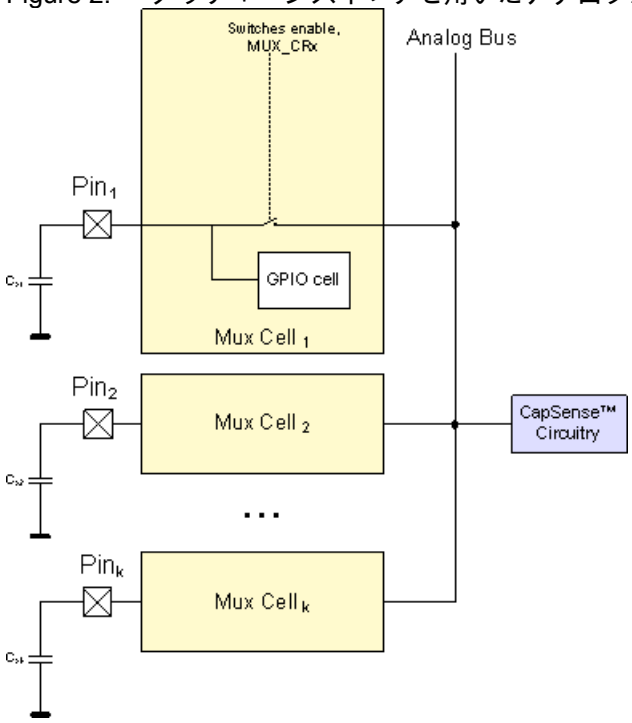
判定理論はファームウェアで実行されます。ファームウェアは、静電容量を分析し、環境要因による遅い静電容量の変化をトラッキングし、判定理論を実行することにより、ボタンのタッチを検知し、スライダの位置を計算します。

センサのアレイをスキャンする

CY8C20x66 デバイスファミリーは、内部アナログバスがあり、、どの PSoC ピンへの静電容量式センサの接続を可能にします。CSD 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。 f 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、アナログバスに永続的に接続されています。

ファームウェアは、MUX_CRx レジスタで対応ビットを設定することにより、連続でセンサスキャンを実行します。

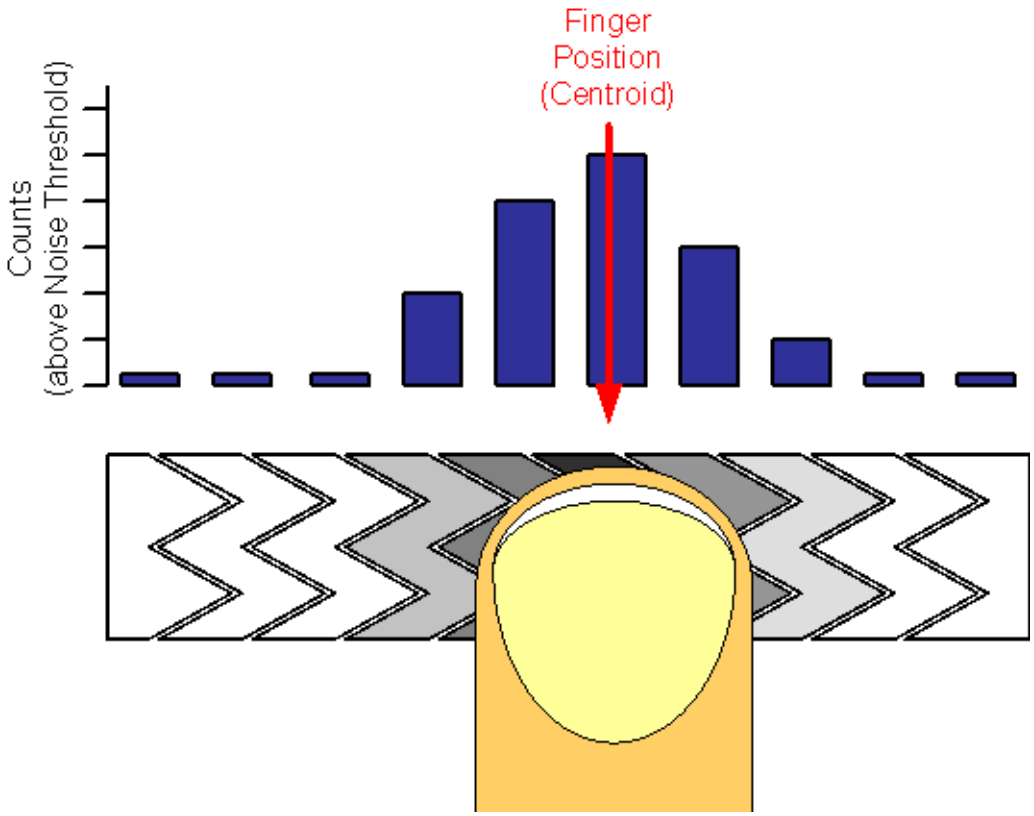
Figure 2.　プリチャージスイッチを用いたアナログバス



スライダ

スライダは、段階的な調整を必要とするアプリケーションに使用します。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、一つのセンサを起動すると、物理的に隣にあるセンサも部分的に起動されます。スライダの実際の位置は、反応したセンサセットの重心位置を計算することによって判断できます。スライダは、CSD 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。現実的なセンサスライダの最低数は 5 で、最大数は選択された PSoC デバイスで使用可能なセンサ位置数です。

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、上半分に同レベルの偽信号を生じますが、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特定し、スライダ位置の特定をします。

ダイプレックス

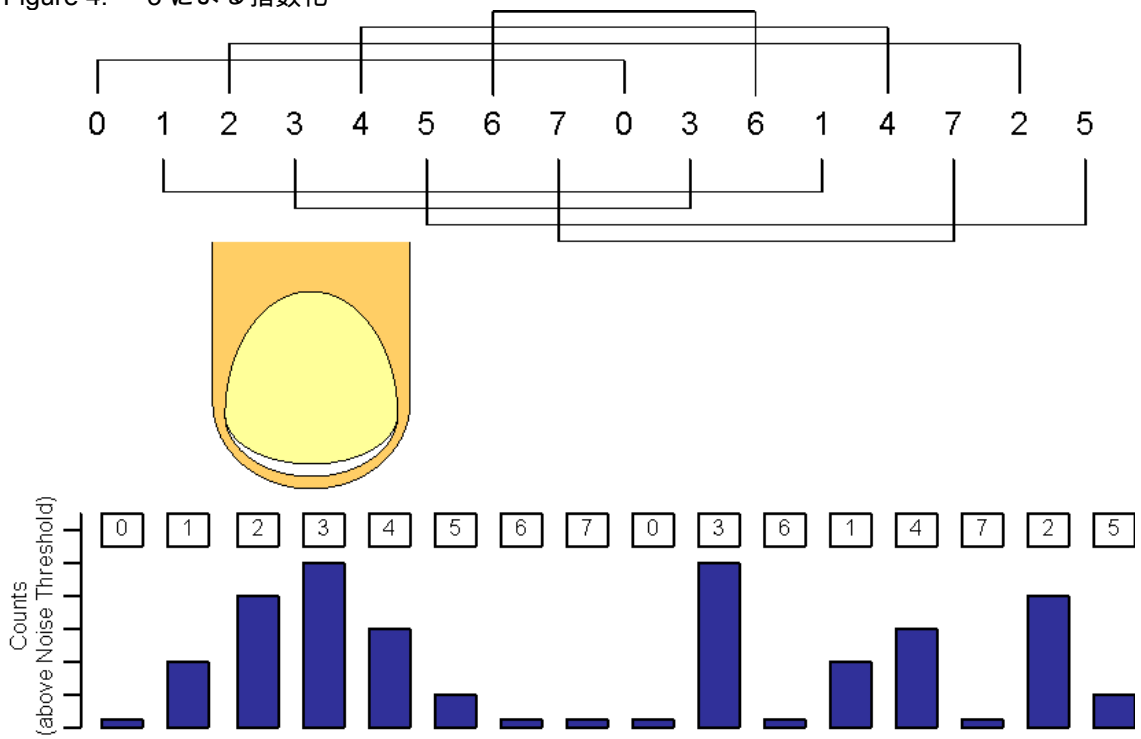
スライダセンサの各センサは、PSoC の各ピンに 2 つづつ接続されます。スライダセンサーの最初の半分（数字が小さい）は、CSD ウイザードでデザイナが割り当てたピンを各センサーに連続的にマッピングします。残りの半分（数字が大きい）のスライダーセンサーは、ウィザードのアルゴリズムによって自動的にマッピングされ、取り込みファイルに一覧表示されます。この順序は、半分における近隣センサ起動が別の半分の近隣センサ起動を引き起こさないように設定されました。この順序の決定と、PCB 基板へのマッピングは慎重に行ってください。

物理センサ位置の残りの半分で順序を設定する方法は幾つかあります。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

[+] Feedback

は、別の指数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 による指数化です。

Figure 4. 3 による指数化



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ー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指数表は、ダイプレックスを選択すると CSD ウィザードによって自動的に作成されます。次の表は、異なるスライダセグメント カウントに対応するダイプレックスシーケンスを一覧表示したものです。

Table 1. 異なるスライダセグメント カウントに対応するダイプレックスシーケンス

スライダセグメントの合計カウント	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
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

スライダセグメントの合計カウント	セグメントシーケンス
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エリアは、通常 1 個のセンサより大きくなっています。

重心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、いくつかの隣り合ったセンサ信号がノイズ閾値を超えていることが要件となります。最も強い信号を中心に、その信号と、ノイズ閾値を超えた隣り合ったセンサーの信号を使用して重心を算出します。重心は（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、計算結果は分数です。重心を特定の分解能の形で報告するには、例えば 12 個のセンサに対して 0 ~ 100 という範囲である場合、重心を計算されたスケーラで掛けます。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、CSD 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、分数として保存されます。

重心の分解能の乗数は、3 バイトに含まれ、そこには下記のビット定義も含まれます。

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

計算結果は、24-bit 符号無し整数であらわされ、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

ユーザ モジュールには、Vss グランド端子から P0[1] または P0[3] ポートピンに接続されている外付け変調コンデンサ C_{mod} が必要です。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。モジュレータ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。

モジュレータ用のコンデンサに推奨されている値は、4.7 ~ 47.0 nF です。最適なコンデンサ容量は、最大の SNR を得るための実験を行うことで選択できます。殆どの場合、5.6 ~ 10.0 nF の値では良い結果が得られます。最良の SNR を得るためには、幾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。

適切な iDAC 値は、総センサ静電容量 C_s によって異なります。この値は、次の条件で選択します。

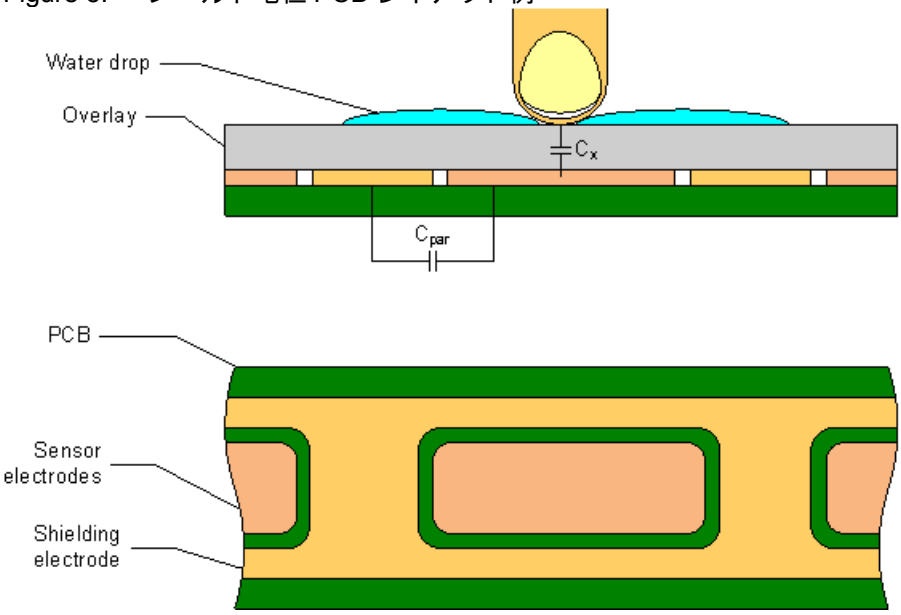
- 異なるセンサのタッチの Raw カウントをモニターする。
- 選択されたスキャン分解能でフルスケール読み値より約 30% 小さい最大読み値を提供する iDAC を選択する。iDAC 値が減ると、Raw カウントは増えます。

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。ホワイト ノイズ、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擬似反応を示さ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側にレイアウトします。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ことができるため、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選ぶことは有用です。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。CSD 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



は図 5、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、ボタン下の平面も含めて、シールド電極を相対する PDB レイヤに置くことも考えられます。この場合、充填率約 30 ~ 40% の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、さらなるグランドを使う必要はありません。

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、変調器のレファレンス電圧は 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、適切な変調器レファレンス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、専用の PSoC ピン（P0[7] または P1[2]）に接続されています。選択されたピンの駆動モードは「**Strong**」にセットします。また、スルー制限レジスタも、PSoC デバイスとシールド電極の間に接続できます。

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。

- 12-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イミュニティからの影響を低減します。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	拡散スペクトル。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。

DC および AC の電気的特性

Table 2.　ノイズ

パラメータ	最小値	標準値	最大値	単位	テスト条件 (Vdd = 3.3V, SysClk = CPU クロック = 24 MHz)
ノイズカウント、ピーク - ピーク	--	0.2	--	% (ノイズカウント) / (基準値カウント)	分解能 >= 14
ノイズカウント、ピーク - ピーク	--	0.5	--	% (ノイズカウント) / (基準値カウント)	分解能 = 12
ノイズカウント、ピーク - ピーク	--	1	--	% (ノイズカウント) / (基準値カウント)	分解能 = 10

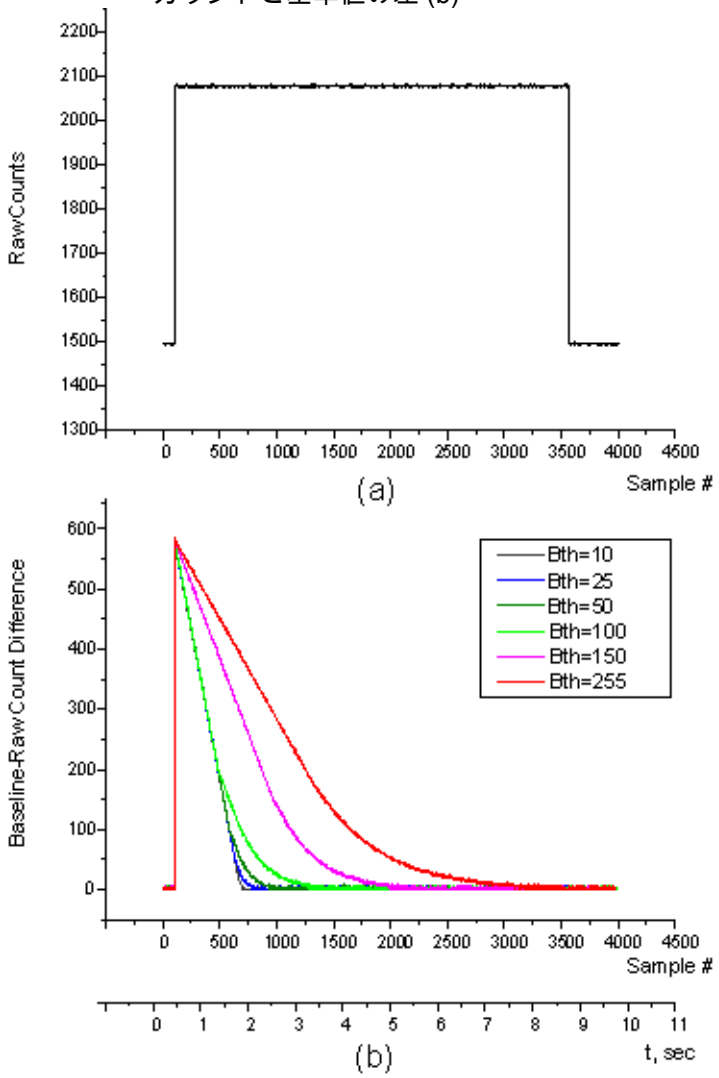
Table 3.　電源電圧

パラメータ	最小値	標準値	最大値	単位	テスト条件とコメント
値	1.7	5.0	5.25	V	

特性グラフ

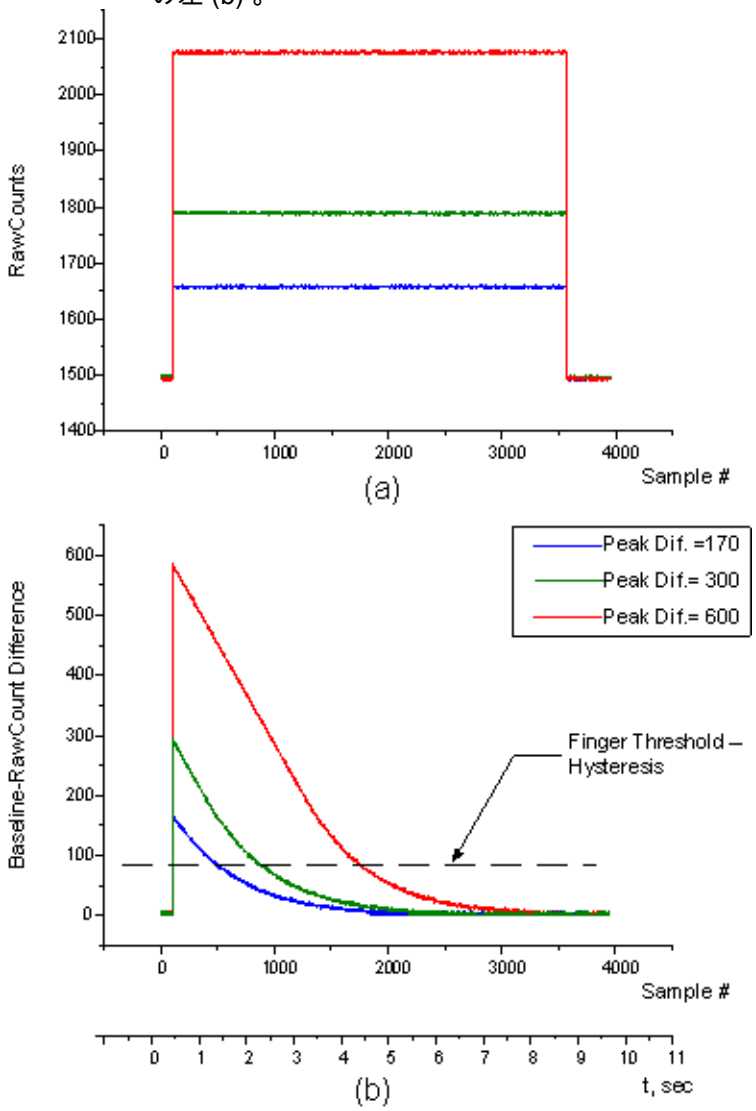
テスト条件 : 12-bit 分解能、SensorsAutoreset = 有効、センサアレイスキャンとデータ転送時間の合計は約 2.5 ms。

Figure 6. Raw カウントステップの変化 (a) と異なる BaselineUpdate 閾値 （ Bth ）パラメータ値 に対する Raw カウントと基準値の差 (b)



Note BaselineUpdate 閾値を上げると、差の減衰速度が下がり、最大ボタンタッチ検知時間が長くなります。

Figure 7. Raw カウントステップの変化 (a) と異なる生カウントステップ変化の値に対する生カウントと基準値の差 (b) 。



テスト条件 : 12-bit 分解能、SensorsAutoreset = 有効、センサアレイスキャンとデータ転送時間の合計は約 2.5 ms。パラメータの BaselineUpdate 閾値は 255 に設定。

Note Raw カウントステップ値が大きいと、下の FingerThreshold-Hysteresis 値の差が小さくなり時間が長くかかるため、センサのオートリセットの間隔が長くなります。

[+] Feedback

配置

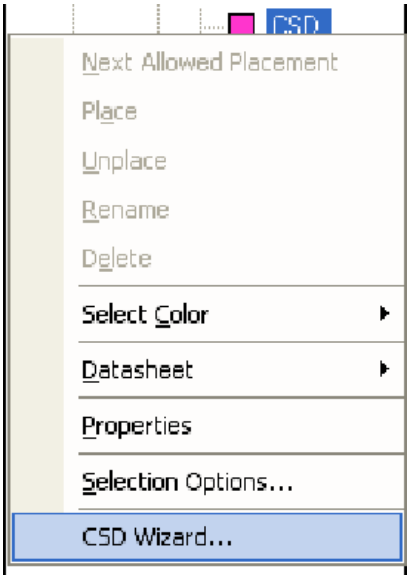
ユーザ モジュールのブロックは、ユーザ モジュールがインスタンス化されると自動的に配置されます。他の配置は利用できません。」CSD ユーザ モジュールは、CapSense ブロックと 1 つのタイマ（タイマ 1）を使用します。

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、CSD ユーザ モジュールのピン接続を確立するために、CSD ウィザードを開始する前に配置しなければなりません。

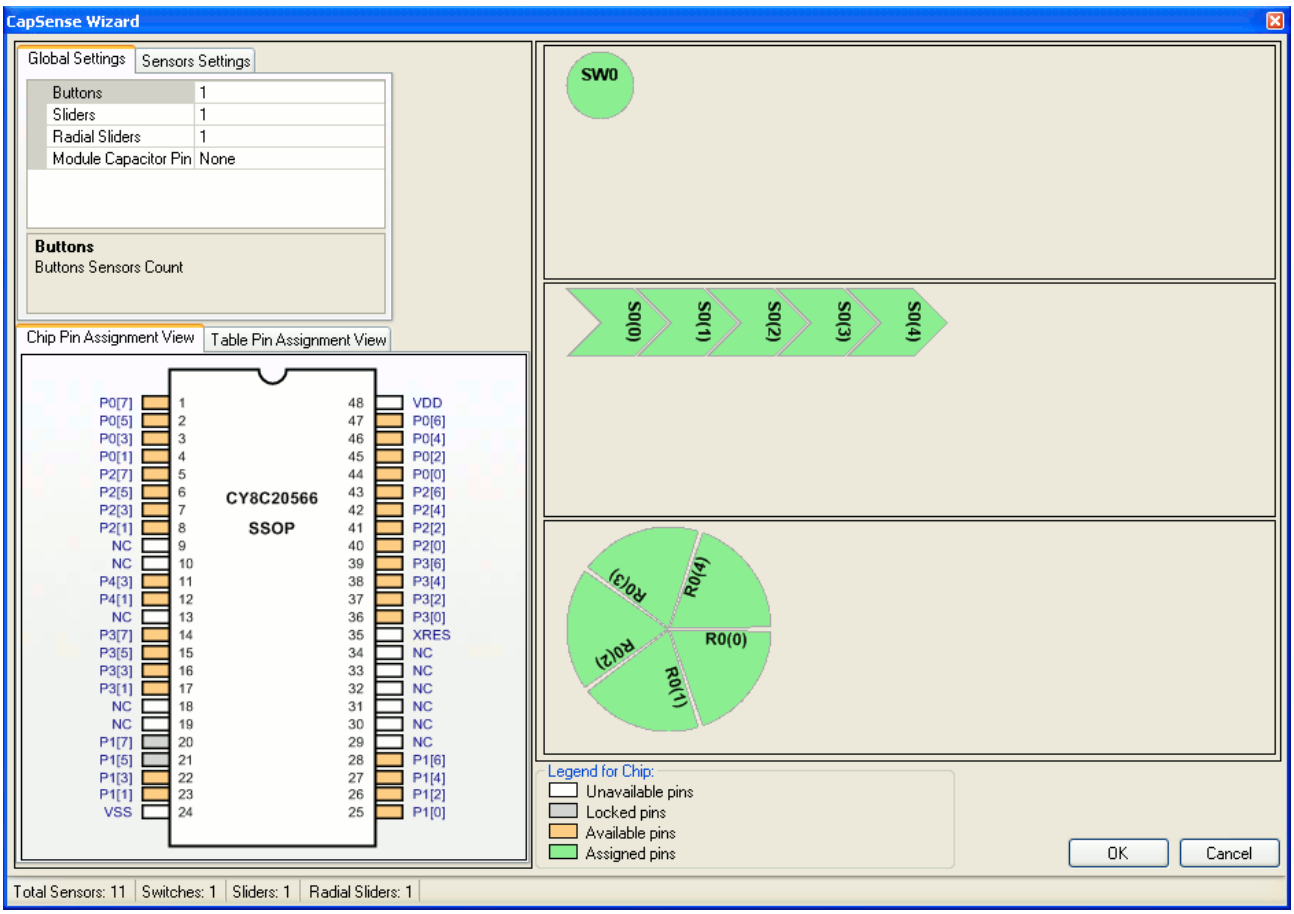
静電容量式センサの接続を配置する場合、P1[0] と P1[1] は避けてください。これらのピンは、その IC のプログラミングに使用され、センサの検出感度とノイズに影響を与える過度のルーティング静電容量を持っている可能性があります。

ウィザード

1. ウィザードにアクセスするには、デバイス エディタの相互接続ビューで CSD の任意のブロックを右クリックし、次に [CSD Wizard] を左クリックします。


CSD
Next Allowed Placement
Place
Unplace
Rename
Delete
Select Color
Datasheet
Properties
Selection Options...
CSD Wizard...

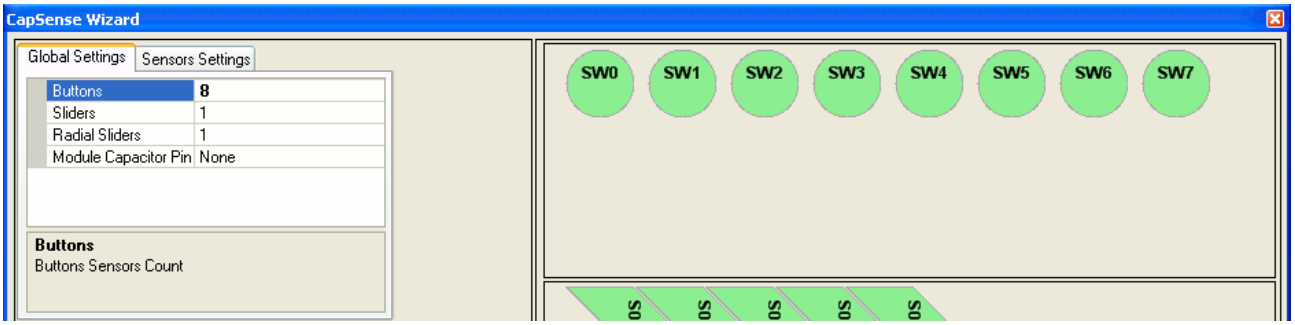
2. ウィザードが開き、ボタンセンサ数、スライダ数、ラジアルスライダ数がボックスに示されます。



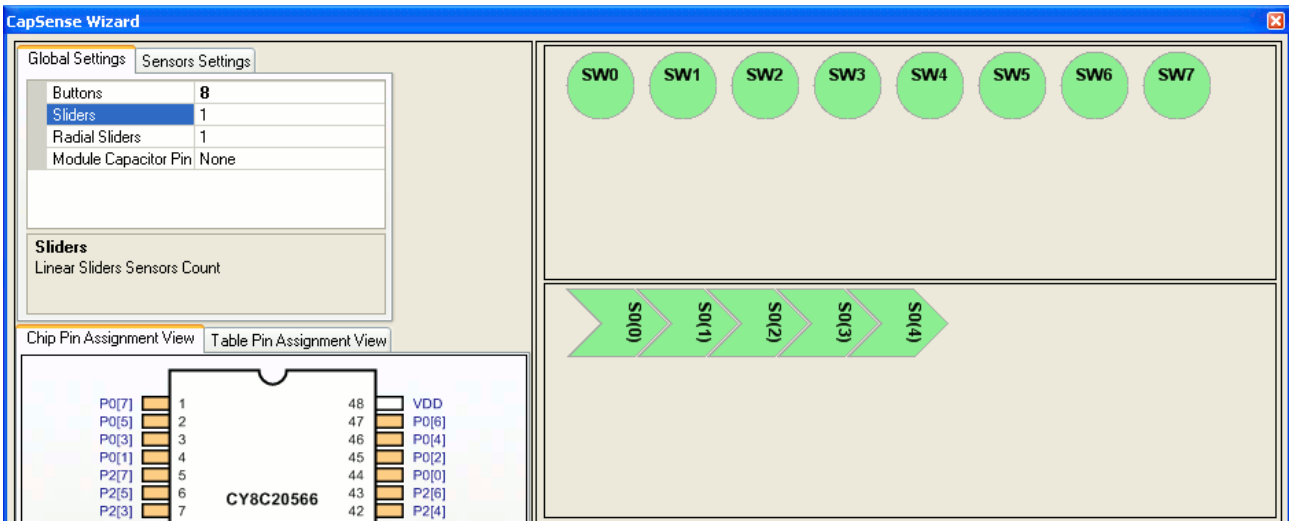
ウィザードのピン凡例

- 白 – このピンは CapSense の入力に使用できません。
- グレイ – 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、そのピンを占有している場合。2 つ目は、ピンの名前がデフォルトから変わった場合。ピン名をデフォルトに戻すには、Pinout ビューでそのピンを拡大し、**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に使用できます。
- グリーン – このピンは CapSense 入力に使用できます。独立センサ数を入力します。センサ数は、使用できるピン数に制限されています。[Enter] キーを押して、センサ数の新しい値を入力します。

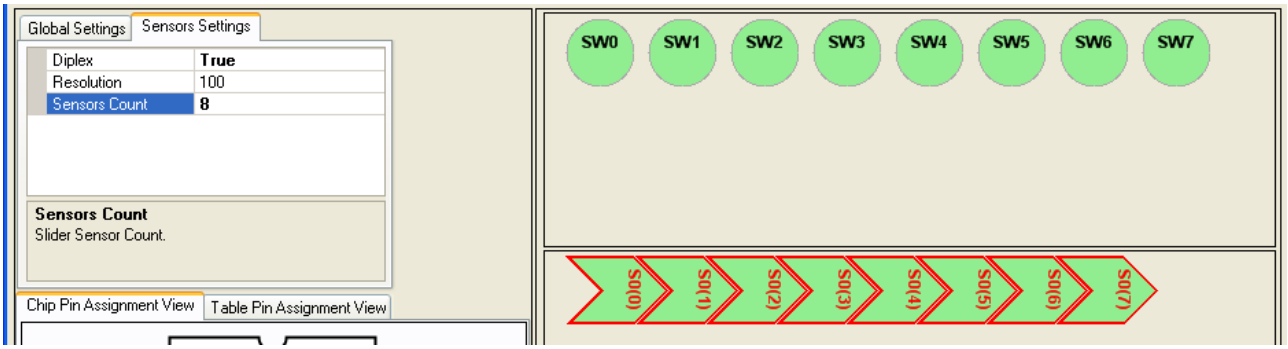
3. 設計に必要なボタンセンサ数を入力します。



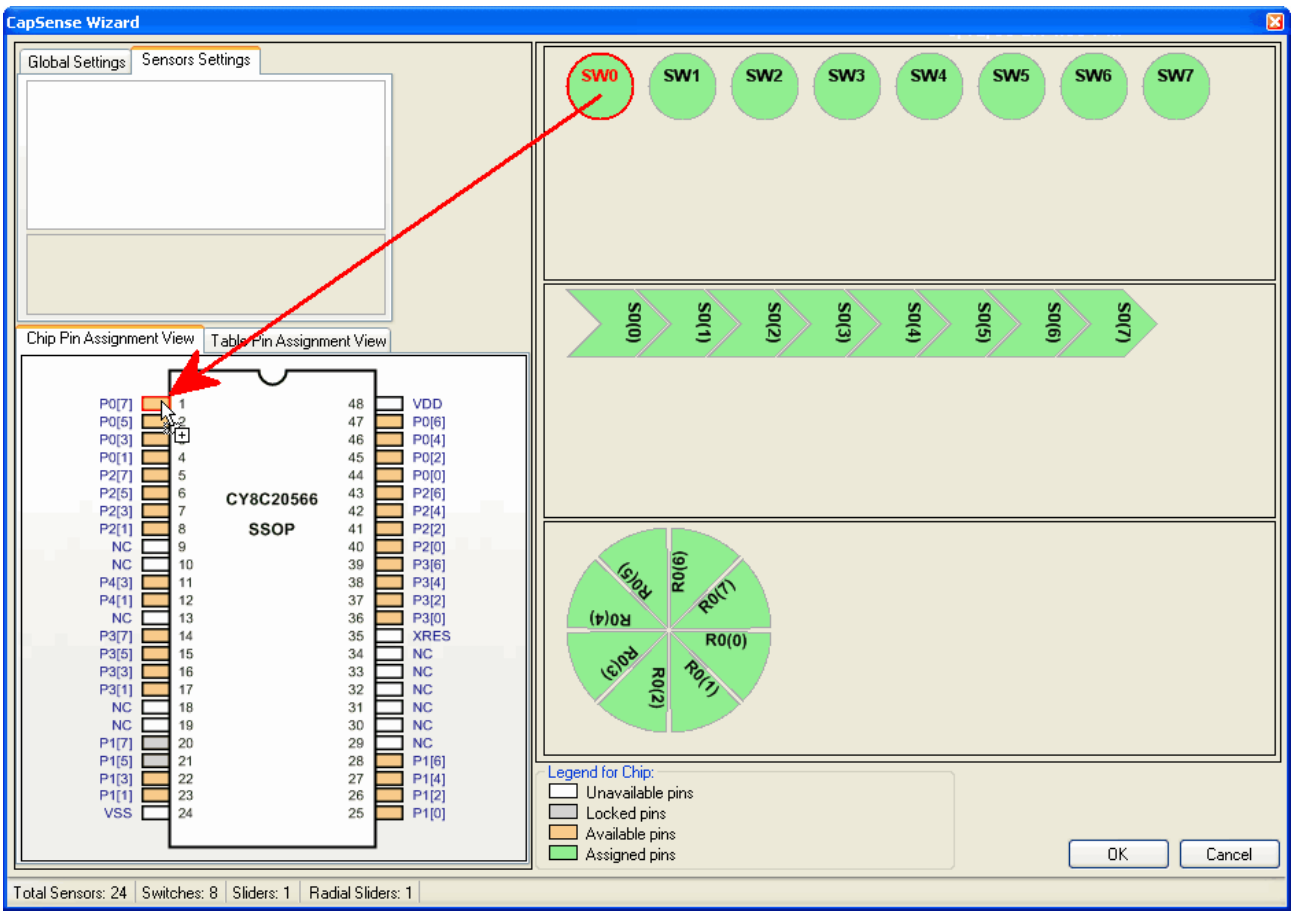
4. スライダ数を入力します。X-Y タッチパッドには 2 つのスライダが必要です。



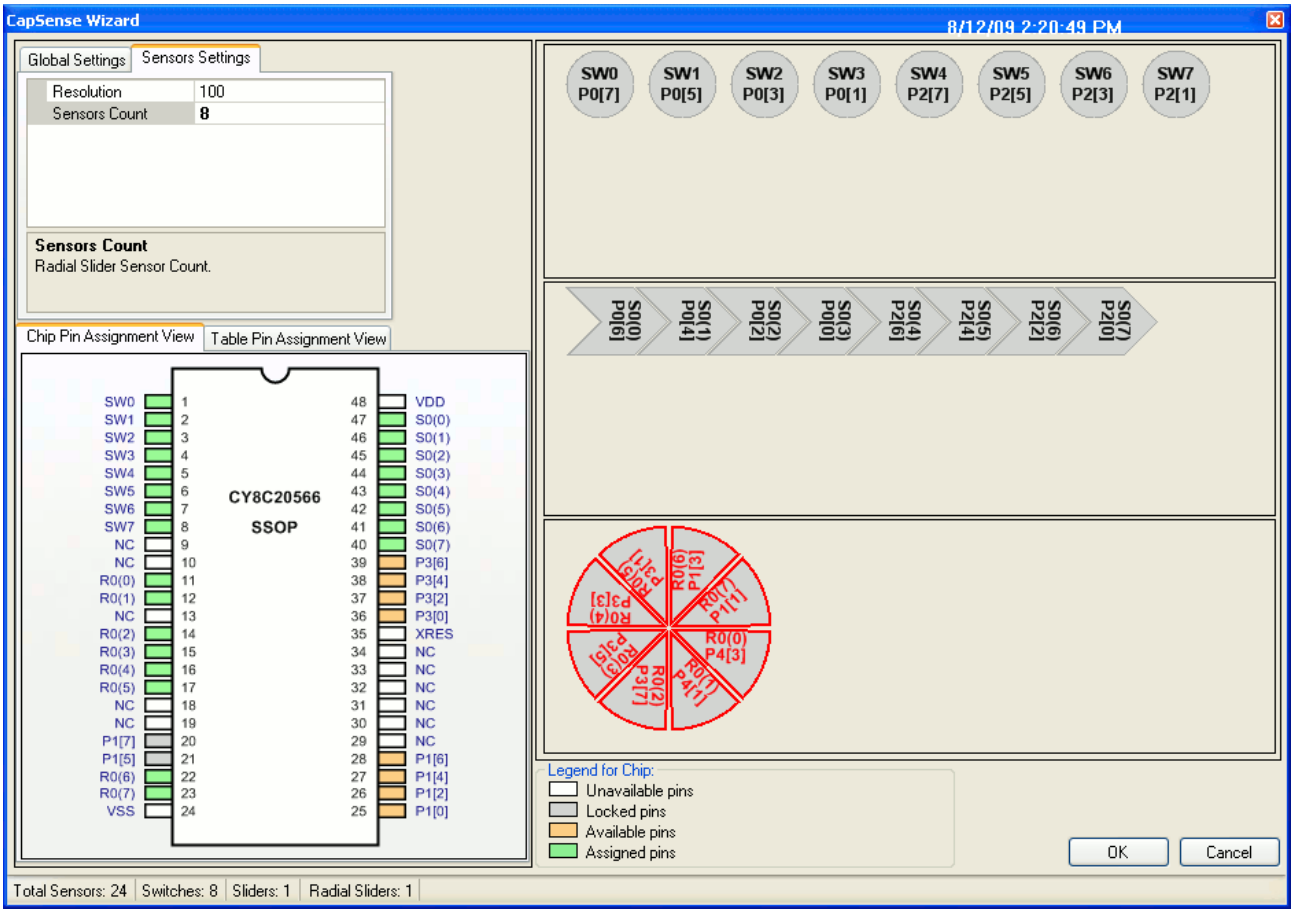
5. スライダをクリックすると、センサ設定が有効になります。センサ設定 **タブを選択** します。各スライダのセンサコンポーネント数を入力します。現実的なセンサスライダ中の最低センサ数は 5 で、最大数はピン数によって制限されます。
6. 適宜、ダイプレックスが選択できます。これにより、センサ用に選択されたピンを、基板上で 2 倍のセンサ位置にマッピングできます。上図では、ダイプレックスセンサの上半分だけが示されています。下半分は、前述の 「ダイプレックス」の項での説明の通り、自動的にマッピングされます。「ダイプレックス」の項で、ピン説明のダイプレックス表を参照してください。
7. 出力の分解能を入力します。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^{16} – 1 or (2 x センサに使用されるピン数 – 1) x 2^{16} -1 です。



8. ラジアルスライダ作成プロセスは、線形スライダ作成プロセスと同じですが、ラジアルスライダはダイプレックスができないところが異なります。
9. [Pin Assignment View] (ピン割り当てビュー) で、スイッチやセンサをピンにドラッグすると、スイッチやセンサをピンに割り当てることができます。または、[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、スイッチやセンサをピンにドラッグすることもできます。ポートピンは一旦選択するとグリーンになり、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。



10. 他のセンサでも同じ操作を繰り返します。物理ポートピンを個々のスライダセンサにマッピングすることは、個々のセンサの場合と同じです。[OK] をクリックして、データを受入れ、PSoC デザイナに戻ります。



これでセンサの配置が完了しました。デバイス エディタウィンドウを右クリックし、[Refresh]（リフレッシュ）を選択すると、ピン接続が更新されます。

ユーザ モジュールパラメータを選択し、アプリケーションを生成します。ここで適宜、サンプルプロジェクトを採用することもできます。

ピン割り当てを変更するには、カーソルを割り当てられているピンに当てクリックし、それをスイッチ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した。

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、分解能に基づいて、一連の表が生成されます。これらの表は CSD_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ついて 2 バイトのエントリから構成されています。1 バイト目はポート番号を示し、2 バイト目はビットのビットマスクです（実際のビット数ではありません）。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。

```
CSD_Sensor_Table:
_CSD_Sensor_Table:
      dw    0x0140  //  Port 1 Bit 6
      dw    0x0301  //  Port 3 Bit 0
      dw    0x0304  //  Port 3 Bit 2
      dw    0x0308  //  Port 3 Bit 3
      dw    0x0302  //  Port 3 Bit 1
      dw    0x0108  //  Port 1 Bit 3
```

この表は CSD_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一つあり、フリーボタンセンサにもエントリが一つあります。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か否かを示します（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重心が乗算されて、CSD ウィザードで望ましい分解能が達成されます。

```
CSD_Group_Table:
_CSD_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、および各スライダによるそれぞれの表のレファレンスという 2 つの部分から構成されています。下記に 8 つのセンサを使用したスライダの典型例を示します。

```
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


 CSD_Diplex_Table:
_CSD_Diplex_Table:
 db >DiplexTable_0, <DiplexTable_0
 db >DiplexTable_1, <DiplexTable_1
```

パラメータおよびリソース

自動キャリブレーション

このパラメータが有効になっていると、Idac 電流が設定され、センサの生カウント値が (2^N)-1 の 85% にセットされます。ここで N は分解能です。デバイス エディタで設定された Idac 値は上書きされます。

自動キャリブレーションパラメータが無効になっていると、Raw カウントは、デバイス エディタと Vref で設定される IDAC 範囲、IDAC 値、分解能、センサ静電容量、IMO 周波数に基づいて計算され、プリスケーラ、プリチャージ源のパラメータに基づいて計算されます。

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、プロセス調整が開始すると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。

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。SetDefaultFingerThresholds() 機能を使用すると、閾値をデバイス エディ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 5 から 255 です。

ノイズ閾値

個々のセンサにて、この閾値を上回るカウント値が観測された場合、ベースラインは更新しません。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、重心の計算に加えられません。可能な値は 5 から 255 です。

BaselineUpdate 閾値

新しい Raw カウント値が現在の基準値を上回っており、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）、現在の基準値と 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、ベースラインはある値分増分し、バケツは空になります。このパラメータは、ベースライン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、ベースラインの更新速度を遅くします。より頻繁なベースラインの更新が必要なときは、このパラメータを小さくします。

LowBaselineReset

LowBaselineReset パラメータは、NegativeNoiseThreshold パラメータと共に作動します。ある一定のサンプル数で、サンプルカウント値が基準値 - NegativeNoiseTreshold の場合、基準値は新しい 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、ベースライン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。。

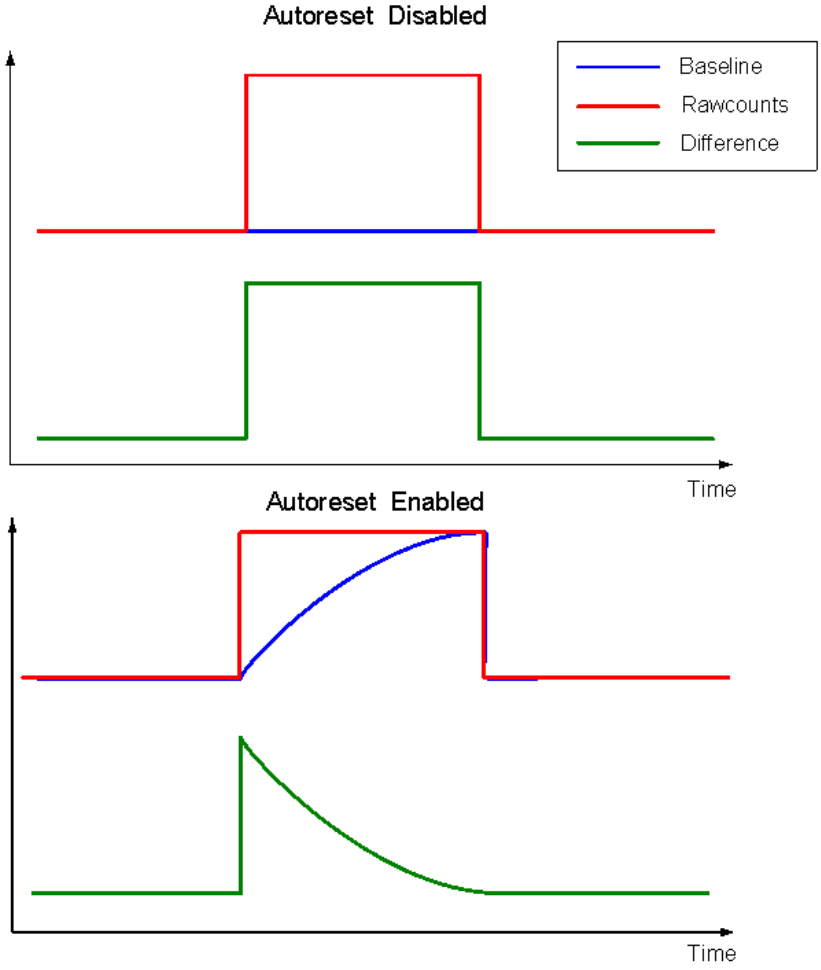
センサの自動リセット

このパラメータは、ベースライン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、ベースライン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 Raw カウントが突然上がった際に、センサが永続的にオンになるのを妨げます。この突然の上昇の原因には、大幅な電源電圧の変化、高エネルギー RF ノイズ源、非常に速い温度変化などが考えられます。

パラメータが [**Disabled**]（無効）にセットされている場合、Raw カウントとベースラインの差がノイズ閾値パラメータを下回る場合にのみ、ベースラインの値は更新されます。何もセンサに触れずに Raw カウントが突然上がった際に、センサが永続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。

は図 8、このパラメータのベースライン更新への影響について図示しています。

Figure 8.　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）を効果的に補償し、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。

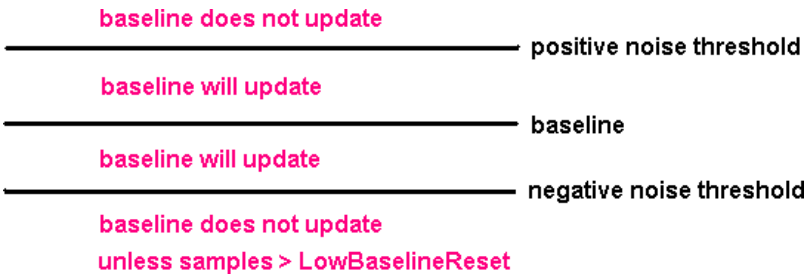
デバウンス

デバウンスパラメータは、デバウンス数をセンサの作動中遷移に追加します。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、差のカウント値が指閾値 + 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 から 255 です。1 をセットするとデバウンスは起こりません。

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 カウントがベースラインを下回り、これらの差がこの閾値を上回る場合、ベースラインは更新されません。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）、ベースラインはリセットされます。



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。

- 優れた SNR
- 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

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 SNR 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 4. 24 MHz IMO 動作におけるスキャン時間（単位：µs）、スキャン 速度と分解能の対照表

分解能（単位：ビット）	スキャン速度			
	超高速	高速	通常	低速
9	57	78	125	205
10	78	125	205	380
11	125	205	380	720
12	205	380	720	1400
13	380	720	1400	2800
14	720	1400	2800	5600
15	1400	2800	5600	11000
16	2800	5600	11000	22000

Note 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。

変調器キャパシタピン

このパラメータは、外付け変調器キャパシタ（C_{mod}）を接続するピンをセットします。利用可能なピン P0[1] と P0[3] から選択します。内部キャパシタが使用されている場合は、[None]（なし）を選択します。一般に、外付けキャパシタを利用するほうが、より良い SNR が得られます。

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、対応する API 関数を用いて行程時間で変更できます。可能な値は 1 から 255 です。

[+] Feedback

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 とプリスケーラです。殆どの場合は、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。

PRS 分解能

このパラメータは、PRS シーケンスの長さを変更します。可能な値は 8 から 12 ビットです。対応するシーケンスの長さは、入力クロック周期の 511 と 2047 です。8-bit の設定で 5:1 の SNR が十分得られない場合は、12-bit 設定を使用します。

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。PRS 出力周波数にも影響を与えます。

可能な値は以下のとおりです。

- 1
- 2
- 4
- 8
- 16
- 32
- 64
- 128
- 256

ShieldElectrodeOut

シールド電極信号源は、P0[7] または P1[2] に接続できます。

Idac 範囲

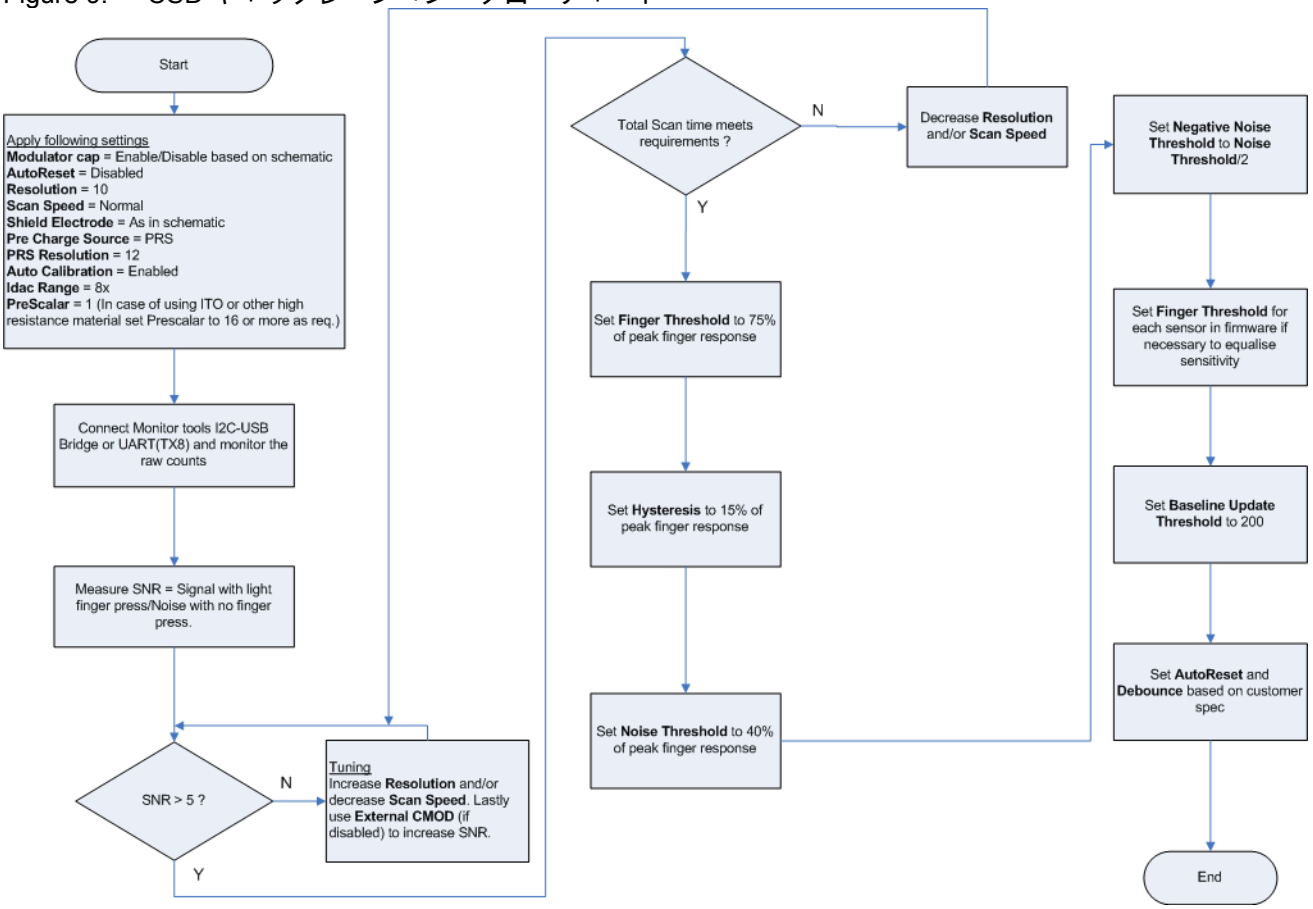
Idac 範囲パラメータは、IDAC 電流出力のスケールをセットします。このパラメータには、次の選択肢が含まれます。

ADC クロック	説明
1x	1X 範囲にスケール調整されている IDAC 出力
2x	2X 範囲にスケール調整されている IDAC 出力
4x	4X 範囲にスケール調整されている IDAC 出力
8x	8X 範囲にスケール調整されている IDAC 出力

CSD キャリブレーション

最適なパフォーマンスを実現するために、各 CSD パラメータは実際の CapSense ハードウェアとオーバーレイに合わせて調整されています。Figure 9 は、CSD のキャリブレーション方法を示しています。

Figure 9. CSD キャリブレーション フローチャート



1. CSD ユーザ モジュールのデフォルト設定から開始します。
2. I^2C-USB ブリッジまたは UART、および実際のハードウェアとオーバーレイを使用して、Raw カウント、ベースライン、センサごとの異なるカウントをキャプチャします。
3. **粗調整。**信号対雑音比（SNR）が 5 より大きいかどうかを確認します。SNR が 5 より小さい場合、推奨されている PCB ガイドラインに従い、外付け C_{mod} を使用して CSD 分解能を上げ、CSD のスキャン速度を下げることで、SNR を増加させます。PCB ガイドラインは、アプリケーション ノート CapSense ベストプラクティス AN2394. SNR とその測定方法の詳細については、アプリケーション ノート静電容量の感知 - CapSense アプリケーションにおける信号対雑音比要件 AN2403.
4. すべてのセンサの合計スキャン時間が要件を満たしているかどうかを確認します。満たしていない場合、分解能を下げたり、スキャン速度を上げたりします。これらのパラメータは SNR にも影響を及ぼすため、ステップ 3 を参照してください。いくつかのパスを使用して、最良の SNR と希望のスキャン時間が実現できる最適な分解能とスキャン速度パラメータを見つけます。
5. ボタンが有効な場合、差のカウントをキャプチャします。指閾値パラメータをピークの 75% にセットします。
6. ノイズ閾値をピーク値の 40% にセットします。
7. ヒステリシスをピーク値の 15% にセットします。

[+] Feedback

8. マイナスノイズ閾値をノイズ閾値の半分にセットします。
9. 適宜、個々のセンサの指閾値をセットします。これには、ファームウェアの CSD_baBtnFThreshold アレイに書き込みを行います。
10. ベースライン更新閾値を要件に応じてセットします。ベースライン更新の頻度は、各プロジェクトごとに指定します。ベースラインは、静電容量式センサへのノイズと温度の効果を低減するのに役立てるために、ゆっくりと動くレファレンスでなければなりません。
    - **ベースラインの更新が早い**：指をゆっくりとボタンへと移動すると、問題が生じる場合があります。これは、指の基準値外れ と呼ばれます。
    - **ベースラインの更新が遅い**：ボタンが温度変化に過敏になり、ボタンロック になる可能性があります。
11. 必要に応じて、自動リセットとデバウンスパラメータをセットします。詳細については、パラメータおよびリソース のセクションを参照してください。

アプリケーション プログラミング インタフェース (API)

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、include ファイルに記載されている関連定数とともに、各関数へのインターフェースについて定義します。

ユーザ モジュールを配置するたびに、インスタンス名が割り当てられます。デフォルトでは、PSoC Designer プロジェクトで、このユーザ モジュールの最初のインスタンスに CSD_1 を割り当てます。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 「CSD」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、A と X レジスタの値は API 関数を呼び出すことで変更できます。A と X の値が呼び出し後に必要な場合は、呼び出し元関数で A と X の値を保存してください。この 「レジスタは脆弱」というポリシーは、効率面の理由から採用され、PSoC デザイナのバージョン 1.0 から発効しています。C コンパイラは、自動的にこの条件で処理されています。またアセンブリ言語のプログラマは、コードがポリシーに準拠していることも保証しなければなりません。一部のユーザ モジュール API 関数では A と X は変更されないこともありますが、将来も変更されないという保証はありません。

エントリポイントは、CSD を初期化し、サンプリングを開始し、CSD を終了するためのものです。すべてのケースで、モジュールのインスタンス名は次のエントリポイントの CSD 接頭語と置き換えられます。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 関数は様々なグローバルアレイに使用されます。そのため、これらのアレイを変更してはなりません。ただし、デバックの目的でこれらの値を検査することは許可されています。例えば、チャーティングツールを使用して、アレイの内容を表示することは許されています。次にいくつかのグローバルアレイを挙げます。

- CSD_waSnsBaseline[]
- CSD_waSnsResult[]
- CSD_waSnsDiff[]
- CSD_baSnsOnMask[]

CSD_waSnsBaseline[] – 各センサのベースライン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。CSD_waSnsBaseline[] アレイは次の関数によって更新されます。

- CSD_UpdateAllBaselines();
- CSD_UpdateSensorBaseline();
- CSD_InitializeBaselines().

CSD_waSnsResult[] – 各センサの Raw 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。CSD_waSnsResult[] データは次の関数によって更新されます。

- CSD_ScanSensor();
- CSD_ScanAllSensors().

CSD_waSnsDiff [] – 各センサの Raw データとベースライン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。

CSD_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。CSD_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)。CSD_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、配置されているセンサをすべて含むために必要な工レメントがすべて含まれます。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。CSD_baSnsOnMask[] データは、CSD_bIsSensorActive(BYTE bSensnor) 関数または CSD_bIsAnySensorActive() ルーチンによって更新されます。

CSD_Start

説明：

レジスタを初期化し、ユーザ モジュールを開始します。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。

C プロトタイプ：

```
void  CSD_Start()
```

アセンブリ：

```
lcall  CSD_Start
```

パラメータ：

なし

戻り値：

なし

特殊作用：

**

CSD_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、CSD_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  CSD_Stop()
```

アセンブリ：

```
lcall  CSD_Stop
```

パラメータ：

なし

戻り値：

なし

特殊作用：

**

CSD_Resume

説明：

CSD_Stop の呼び出し後、ユーザ モジュール動作を再開します。

C プロトタイプ：

```
void  CSD_Resume()
```

アセンブリ：

```
lcall  CSD_Resume
```

パラメータ：

なし

戻り値：

なし

特殊作用：

**

CSD_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。この番号は CSD ウィザードによって順番に割り当てられるものです。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。

C プロトタイプ：

```
void CSD_ScanSensor(BYTE bSensor);
```

アセンブリ：

```
mov A,  bSensor
```

```
lcall CSD_ScanSensor
```

パラメータ：

A => センサ番号

戻り値：

なし

副作用

**

CSD_ScanAllSensors

説明：

各センサインデックスについて CSD_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 CSD_ScanAllSensors();
```

アセンブリ：

```
lcall CSD_ScanAllSensors
```

パラメータ：

なし

戻り値：

なし

副作用

**

CSD_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、センサのベースラインと呼ばれます。ベースライン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. CSD_UpdateSensorBaseline() が呼び出されるたびに、Raw カウント値を前回のベースラインから引いて差のカウントを計算します。この差は CSD_waSnsDiff[] アレイに保存され、表示されます。

2. センサ自動リセットが無効な場合、CSD_UpdateSensorBaseline() が呼び出されるたびに、Diff のカウントをノイズ閾値と比較します。Diff 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。Diff がノイズ閾値より大きい場合、バケツは更新されません。センサ自動リセットが有効な場合、ノイズ閾値パラメータに関わらず、差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のカウントが BaselineUpdateThreshold に達すると、基準値は 1 増分され、バケツは 0 にリセットされます。

4. 差のカウント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 CSD_UpdateSensorBaseline(BYTE bSensor)
```

アセンブリ：

```
mov   A,   bSensor
lcall  CSD_UpdateSensorBaseline
```

パラメータ：

A => センサ番号

戻り値：

なし

特殊作用：

**

CSD_UpdateAllBaselines

説明：

CSD_bUpdateSensorBaseline() 関数を使用して、すべてのセンサのベースラインを更新します。

C プロトタイプ：

```
void  CSD_UpdateAllBaselines()
```

アセンブリ：

```
lcall CSD_UpdateAllBaselines
```

パラメータ：

なし

戻り値：

なし

特殊作用：

**

CSD_bIsSensorActive

説明：

センサの差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、指閾値に足したり、指閾値から引いたりされます。動作中の場合、閾値は下げられます。動作中でない場合、閾値は上げられます。この関数は、CSD_baSnsOnMask[] アレイのセンサのビットを更新します。

C プロトタイプ：

```
BYTE  CSD_bIsSensorActive(BYTE bSensor)
```

アセンブリ：

```
mov  A,  bSensor
lcall CSD_bIsSensorActive
```

パラメータ：

bSensor A => センサ番号

戻り値：

戻り値は、動作中の場合 1 で、動作中でない場合は 0 です。

A => 1 – 選択されたセンサが動作中。0 – 選択されたセンサが動作中ではない。

特殊作用：

**

CSD_bIsAnySensorActive

説明：

全センサの Diff 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 CSD_bIsSensorActive() を呼び出し、CSD_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  CSD_bIsAnySensorActive()
```

アセンブリ：

```
lcall CSD_bIsAnySensorActive
```

パラメータ：

なし

戻り値：

戻り値は、動作中の場合 1 で、動作中でない場合は 0 です。

A => 1 – 1 つ以上のセンサが動作中。0 – 動作中のセンサはない。

特殊作用：

**

CSD_wGetCentroidPos

説明：

Diff アレイを確認し、重心を探します。1 つ存在する場合、オフセットと長さが一時変数に保存され、重心位置が CSD 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 CSD 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  CSD_wGetCentroidPos(BYTE bSnsGroup)
```

アセンブリ：

```
mov  A,  bSnsGroup
lcall CSD_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 から上に含まれています。

戻り値：

スライダの位置値は A では LSB、X では MSB です。

特殊作用：

このルーチンは、ノイズ閾値を引くことによって Diff カウントを調整します。マイナスの差の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。Diff のカウント信号をモニターするアプリケーションの場合、Diff の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 CSD ウィザードで設定された分解能値を返します。作動中のセンサがない場合、関数は –1 (FFFFh) を返します。重心 / 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、CSD_blsSensorActive() ルーチンを使用して接触されたスライダセグメントを判定できます。

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擬似重心結果を示すことがあります。ノイズが擬似重心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）。

CSD_wGetRadialPos

説明：

Diff アレイを確認し、重心を探します。1 つ存在する場合、重心位置を CSD 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、CSD 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  CSD_wGetRadialPos(BYTE bSnsGroup)
```

アセンブリ：

```
mov  A,  bSnsGroup
call CSD_wGetRadialPos
```

パラメータ：

bSnsGroup A => グループ番号

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 CSD UM 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値は A では LSB、X では MSB です。

特殊作用：

マイナスの Diff 値とベースライン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。Diff のカウント信号をモニターするアプリケーションの場合、Diff の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 CSD ウィザードで設定された分解能値を返します。作動中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擬似重心結果を示すことがあります。ノイズが擬似重心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）。

CSD_wGetRadialInc

説明：

実際の指シフトを返します。これは現在と前回の指位置の差です。この関数は CSD_wGetRadialPos() とともに機能し、後者から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  CSD_wGetRadialInc(BYTE bSnsGroup)
```

アセンブリ：

```
mov  A,  bSnsGroup
lcall CSD_wGetRadialInc
```

パラメータ：

bSnsGroup A => グループ番号

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 CSD UM 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。A では LSB、X では MSB。

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（前々回 CSD_wGetRadialPos() が 1 (FFFFh) を返した）、または今回タッチがない場合（今回 CSD_wGetRadialPos() が -1 (FFFFh) を返した）。

特殊作用：

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

CSD_InitializeSensorBaseline

説明：

選択されたセンサをスキャンして、初期値を伴う CSD_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ベースラインの配列エレメントにコピーされます。この関数は、個々のセンサのベースラインをリセットするために使用されます。

C プロトタイプ：

```
void CSD_InitializeSensorBaseline(BYTE bSensor)
```

アセンブリ：

```
mov A, bSensor
lcall CSD_InitializeSensorBaseline
```

パラメータ：

A => センサ番号

戻り値：

なし

特殊作用：

**

CSD_InitializeBaselines

説明：

各センサをスキャンして、初期値を伴う CSD_waSnsBaseline[] アレイを読み込みます。Raw カウント値は各センサのベースラインアレイにコピーされます。

C プロトタイプ：

```
void CSD_InitializeBaselines()
```

アセンブリ：

```
lcall CSD_InitializeBaselines
```

パラメータ：

なし

戻り値：

なし

特殊作用：

**

CSD_SetDefaultFingerThresholds

説明：

FingerThreshold パラメータ値を伴う CSD_baBtnFThreshold[] アレイを読み込みます。CSD_baBtnFThreshold[] アレイがカスタム値とともに手動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れなければなりません。

C プロトタイプ：

```
void CSD_SetDefaultFingerThresholds()
```

アセンブリ：

```
lcall CSD_SetDefaultFingerThresholds
```

パラメータ：

なし

戻り値：

なし

特殊作用：

**

CSD_SetScanMode

説明：

スキャン速度と分解能を設定します。この関数を実行時に呼び出し、スキャン速度と分解能を変更することができます。関数は、ユーザ モジュールのパラメータ設定を上書きします。異なるスキャン速度と分解能でスキャンする必要のあるセンサがある場合、この関数は効果的です。例としては、通常のボタンと近接検出器などが挙げられます。通常のボタンは 9-bit 分解能でスキャンできます。近接検出器では、長い範囲の検知を実行するために、16-bit の分解能と長いスキャン時間でスキャンの頻度を下げます。この関数は CSD_ScanSensor() 関数とともに使用できます。

C プロトタイプ：

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

アセンブリ：

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

パラメータ：

bSpeed

bResolution

戻り値：

なし

特殊作用：

**

CSD_SetSliderIdac

説明：

スライダエレメントの iDAC 電流値を各スキャングループの最大値にセットします。

C プロトタイプ：

```
void CSD_SetSliderIdac(void)
```

アセンブリ：

```
lcall  CSD_ SetSliderIdac
```

パラメータ：

なし

戻り値：

なし

特殊作用：

**

CSD_SetIdacValue

説明：

この関数はユーザ モジュールのパラメータ設定の 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 CSD_ScanSensor(). とともに使用できます。

C プロトタイプ：

```
void CSD_SetIdacValue(BYTE bRefValue);
```

アセンブリ：

```
mov     A, bIdacValue
lcall    CSD_SetIdacValue
```

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

特殊作用：

**

CSD_SetPrescaler

説明：

ユーザ モジュールのパラメータ設定のプリスケーラ値を上書きします。プリスケーラ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 CSD_ScanSensor(). とともに使用できます。

C プロトタイプ：

```
void CSD_SetPrescaler(BYTE bPrescaler);
```

アセンブリ：

```
mov     A, bPrescaler
lcall    CSD_SetPrescaler
```

パラメータ：

bPrescaler - プリスケーラ値をセットします。次の表に、許可されている値が一覧表示されています。

名前	値	プリスケーラ
CSD_PRESCALER_1	0x00	1
CSD_PRESCALER_2	0x01	2
CSD_PRESCALER_4	0x02	4
CSD_PRESCALER_8	0x03	8
CSD_PRESCALER_16	0x04	16

名前	値	プリスケーラ
CSD_PRESCALER_32	0x05	32
CSD_PRESCALER_64	0x06	64
CSD_PRESCALER_128	0x07	128
CSD_PRESCALER_256	0x08	256

戻り値：

なし

特殊作用：

**

CSD_CalibrateSensors

説明：

wLevel 値に近い Raw カウント値を得るために iDAC 電流を調整し、結果をグローバルアレイ CSD_baDAC[]. に保存します。

C プロトタイプ：

```
void CSD_CalibrateSensors(WORD wLevel)
```

アセンブリ：

```
lcall  CSD_CalibrateSensors
```

パラメータ：

wlevel - 対象となる Raw データ値。

戻り値：

なし

特殊作用：

**

CSD_ClearSensors

説明：

各センサに対して CSD_wGetPortPin() と CSD_DisableSensor() を連続的に呼び出すことにより、すべてのセンサを非サンプリング状態にクリアします。

C プロトタイプ：

```
void  CSD_ClearSensors()
```

アセンブリ：

```
lcall  CSD_ClearSensors
```

パラメータ：

なし

戻り値：

なし

特殊作用：

**

CSD_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  CSD_wReadSensor(BYTE bSensor)
```

アセンブリ：

```
mov A, bSensor
lcall CSD_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。

特殊作用：

**

CSD_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、CSD_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、CSD_EnableSensor()、CSD_DisableSensor(). へと渡すことができます。

C プロトタイプ：

```
WORD  CSD_wGetPortPin(BYTE bSensorNum)
```

アセンブリ：

```
mov  A,  bSensorNumber
lcall     CSD_wGetPortPin
```

パラメータ：

bSensorNumber – 範囲は 0 ~ (n – 1) です。ここで n は、CSD 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、選択された動作中のセンサに関連したポートとビットマスクを判定するために、CSD_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

特殊作用：

**

CSD_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、CSD_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 Z）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  CSD_EnableSensor(BYTE bMask, BYTE bPort)
```

アセンブリ：

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

特殊作用：

**

CSD_DisableSensor

説明：

CSD_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、ゼロにセットされます。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 AnalogMuxBus までの接続はオフになります。この関数パラメータは、CSD_wGetPortPin() 関数により返されます。

C プロトタイプ：

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

アセンブリ：

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

特殊作用：

**

ファームウェア ソースコート゛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 CSD 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;
  CSD_Start();
  CSD_InitializeBaselines() ; //scan all sensors first time, init baseline
  CSD_SetDefaultFingerThresholds() ;
  //
  // Loop Forever
  //
  while (1) {
    CSD_ScanAllSensors(); //scan all sensors in array (buttons and sliders)
    CSD_UpdateAllBaselines(); //Update all baseline levels;

 //detect if any sensor is pressed
 if(CSD_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 CSD_waSnsResult[x] <- Raw Counts
    // OUTPUT CSD_waSnsDiff[x] <- Difference
    // OUTPUT CSD_waSnsBaseline[x] <- Baseline
    // OUTPUT CSD_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 CSD_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 CSD 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;

     CSD_Start();
     CSD_SetDefaultFingerThresholds();

     // Enable the sensor connected to P1[4]
     CSD_EnableSensor(0x10, 1);
     // Enable the sensor connected to P1[6]
     CSD_EnableSensor(0x40, 1);
     // Enable the sensor connected to P3[0]
     CSD_EnableSensor(0x01, 3);

     // Initialize baseline for sensor number "3"
     CSD_InitializeSensorBaseline(3);

     while (1) {
          // Scan continuously sensor number "3" which is connected
          //in parallel to the enabled above sensors
          CSD_ScanSensor(3);
          CSD_UpdateSensorBaseline(3);
                if(CSD_bIsSensorActive(3)){
        // Add user code here to proceed the buttons pressing
                }
     }
}
```

例 3。次の例は、CSD_SetScanMode() 関数を使用して異なるスキャンパラメータを用いる異なるセンサをスキャンする能力を示しています。ボタンタッチ検知と近接検知を実行する必要がある場合に有用です。ボタンはスキャン時間を短縮するために低分解能でスキャンし、近接検知は最大感度を実現するために高分解能でスキャンします。近接検知スキャンの頻度を下げ、ボタンタッチが検知されない場合にのみ、このコードを採用することもできます。

```
//-------------------------------------------------------------------------
// Sample C code for the CSD module
// Scanning sensors with different scanning speed and resolution
//-------------------------------------------------------------------------

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

void main(void)
{

     M8C_EnableGInt;

     CSD_Start();
     CSD_SetDefaultFingerThresholds();

     // Set UltraFast, 9-bit resolution mode for baseline calculations
     CSD_SetScanMode(0, 9);

     // Initialize baselines for all of the sensors which operate in
     // Ultra Fast mode and 9-bit resolution
     CSD_InitializeSensorBaseline(0);
     CSD_InitializeSensorBaseline(1);
     CSD_InitializeSensorBaseline(2);

     // Set Slow, 14-bit resolution mode for baseline calculations
     CSD_SetScanMode(3, 14);
     // Initialize baselines for all of the sensors which operate in
     // Slow mode and 14-bit resolution
     CSD_InitializeSensorBaseline(3);

     while (1) {
          // Set UltraFast, 9-bit resolution mode for the following buttons
          CSD_SetScanMode(0, 9);
          // Scan sensor number "0"
          CSD_ScanSensor(0);
          // Scan sensor number "1"
          CSD_ScanSensor(1);
          // Scan sensor number "2"
          CSD_ScanSensor(2);

          // Set Slow, 14-bit resolution mode for the following sensor
          CSD_SetScanMode(3, 14);
          // Scan sensor number "3"
          CSD_ScanSensor(3);

          CSD_UpdateAllBaselines();
          //detect if any sensor is pressed
          if(CSD_bIsAnySensorActive()){
```

```
                // Add user code here to proceed the buttons pressing
            }
        }
}
```

例 4。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 CSD 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;

 CSD_Start();
 CSD_InitializeBaselines();

 // set finger threshold for sensor "0"
 CSD_baBtnFThreshold[0] = 10;
 // set finger threshold for sensor "1"
 CSD_baBtnFThreshold[1] = 20;
 // set finger threshold for sensor "2"
 CSD_baBtnFThreshold[2] = 30;
 // set finger threshold for sensor "3"
 CSD_baBtnFThreshold[3] = 40;
 // set finger threshold for sensor "4"
 CSD_baBtnFThreshold[4] = 50;
 // set finger threshold for sensor "5"
 CSD_baBtnFThreshold[5] = 255;
 // set finger threshold for sensor "6"
 CSD_baBtnFThreshold[6] = 200;

 while (1) {
  // Scan continuously all sensors
  CSD_ScanAllSensors();
  CSD_UpdateAllBaselines();
  //detect if any sensor is pressed
  if(CSD_bIsAnySensorActive()){
   // Add user code here to proceed the buttons pressing
  }
 }
}
```

設定レシ゛スタ

Table 5. 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	CSD_PRSCLK	0	1	0	0	EN

Table 6. 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 7. 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 8. 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 9. 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 10. 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 11. 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	プリスケーラ			

Table 12. 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	開始

Table 13. 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 14. 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							

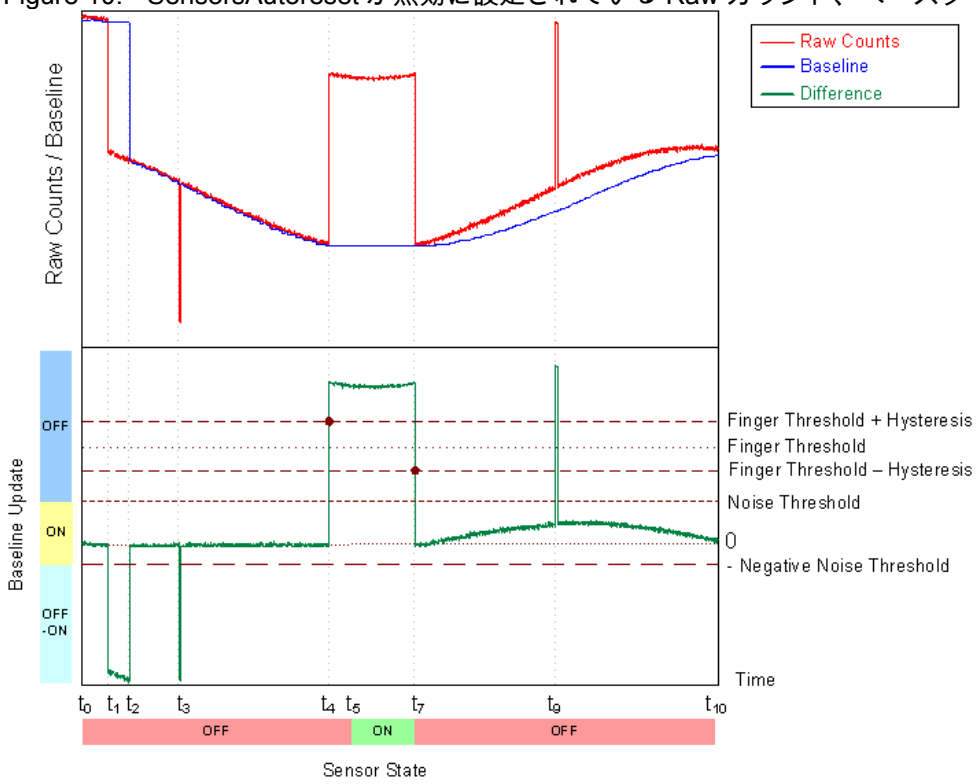
付録

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デザインを支援するために、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。

CSD パラメータのインタラクション

図 10 および図 11 は、ベースラインの更新と判定理論動作を示しており、最適なパフォーマンスを実現するための UM パラメータの設定方法を理解するうえで役立ちます。は図 10、センサオートリセットパラメータが [**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。は図 11、センサオートリセットパラメータが [**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、マイナスノイズ閾値が、Diff の信号とともに示されています（Raw カウント – ベースライン）。データは、システム動作を示す人工的なテストで、低速と高速の Raw 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、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 10. SensorsAutoreset が無効に設定されている Raw カウント、ベースライン、Diff の信号変化の例



湿度や気温の変化により、ベースラインレベルに近い Raw カウントは t_0 でゆっくりと値を落とし始めます。2 つの連続した変換の間に起きる Raw 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 カウントの最大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、ベースラインは更新されます。

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、ベースライン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。このベースラインは、LowBaselineReset のサンプルでいえば、Diff の信号が 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。

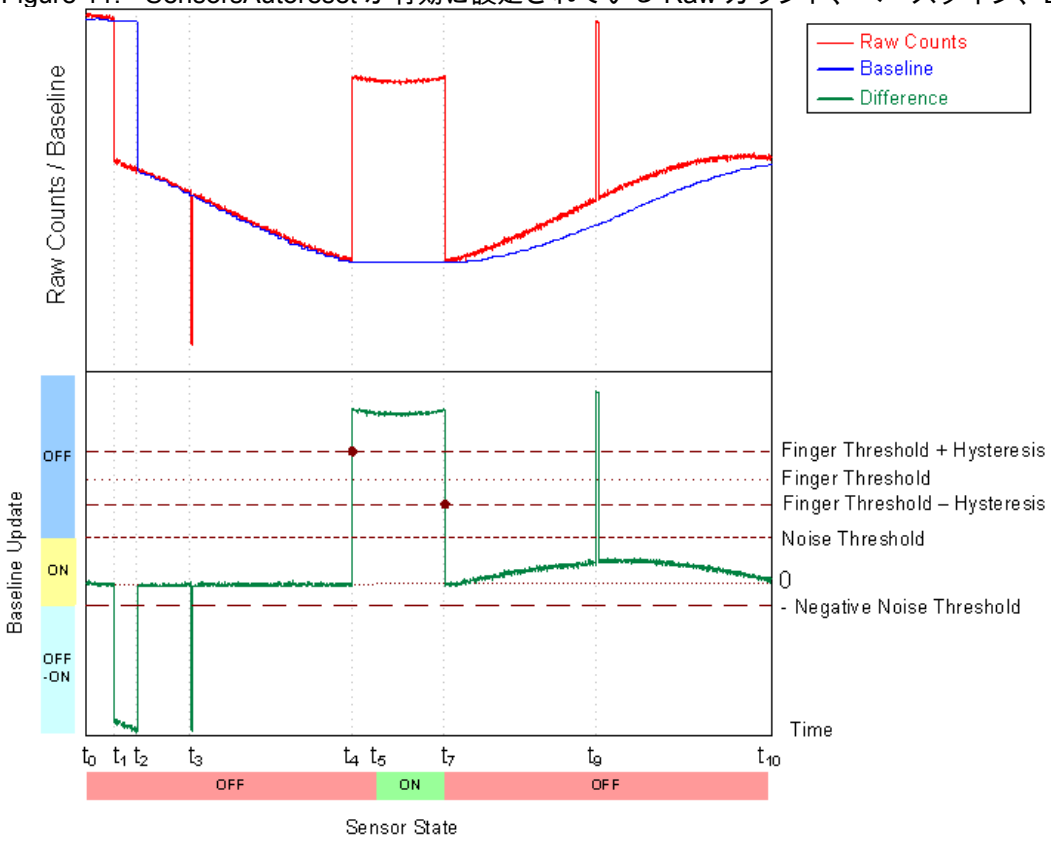
2 番目に大きなマイナスの Diff 信号スパイクは、t_3 で発生します。このスパイクが発生する原因は ESD イベントなどです。サンプルカウント中のスパイクが LowBaselineReset パラメータを下回るため、ベースラインは待機状態となり、スパイクはフィルタにかけられます。これにより、擬似基準値リセットと、その結果である擬似タッチ検知を防ぐことができます。

センサは t_4 でタッチされています。Diff 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。Diff 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 の短く上向きのスパイクは、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。

Raw カウントは、t_7 と t_{10} の間、ゆっくりと上昇します。Diff 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ベースライン値を更新します（SensorsAutoreset は Disabled に設定）。差の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ベースライン

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、ベースライン更新速度が速くなります。

Figure 11. SensorsAutoreset が有効に設定されている Raw カウント、ベースライン、Diff の信号変化の例



システム動作（ ）図 11 は、前述のケースの動作に似ていますが、下記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、動作中の基準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、タッチ検知を短い間ブロックする LowBaselineReset サンプル（t_8）後に、ベースライン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。

改訂履歴

バージョン	作成者	説明
1.1	DHA	1. このバージョンは、スライダをもう一つ追加できる容量があります。 2. オートキャリブレーションパラメータを追加しました。
1.20	DHA	CY8C20xx6 および CY8CTMA3xx デバイスファミリー用にラジアルスライダ機能を追加しました。

Note PSoC Designer 5.1 より、全ユーザ モジュールの データシートのバージョン履歴が追加されます このセクションは、現在および以前のユーザ モジュール バージョン間の差分の概要を説明するものです。

Copyright © 2008-2010 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer ™ and Programmable System-on-Chip ™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.