CapSense® Sigma-Delta 数据表 CSD V 1.20

Copyright © 2008-2014 Cypress Semiconductor Corporation. All Rights Reserved.

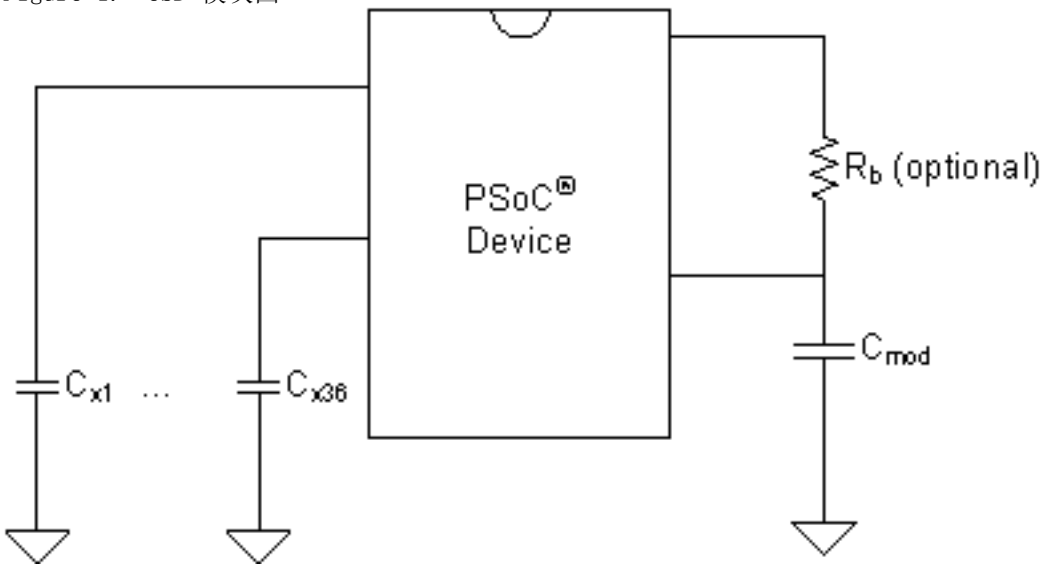
资源	PSoC® 模块				API 存储器 （字节）		每个传感器所使用的引脚数
	CapSense®	I²C/SPI	定时器	比较器	闪存	RAM	
CY8C20x66、CY8C20x36、CY8C20336AN、CY8C20436AN、CY8C20636AN、CY8C20x46、CY8C20x96、CY7C645xx、CY7C643/4/5xx、CY7C60424、CY7C6053x、CYONS2110、CYONSFN2xxx							
	1	-	1	-	1540	35	1

功能和概述

- 扫描 1 到 36 个电容式传感器。
- 感应能力最多可穿透 15 毫米的玻璃覆盖层。
- 使用线缆传感器时，检测距离接近 20 厘米。
- 对交流电源噪声、EMC 噪声和电源电压变化，具有极强的抗干扰能力。
- 支持独立传感器和滑条电容传感器的任意组合。
- 通过覆用法，使滑条传感器的物理分辨率增加一倍。
- 利用内插法，提高滑条传感器的分辨率。
- 带有两个滑条传感器的触摸板支架。
- 感应能力通过高阻抗性传导材质 （例如 ITO 薄膜）实现。
- 即使存在水膜或水滴的情况下，屏蔽电极仍可为可靠的运行提供保证。
- 通过 CSD 向导完成传感器和引脚分配。
- 使用集成基线更新算法处理温度、湿度和静电释放 (ESD) 事件。
- 可轻松调整的操作参数。
- PC GUI 应用程序支持实时原始数据监控和参数优化。

CSD （使用 Sigma-Delta 调制器的电容式感应技术）通过 sigma-delta 调制器使用开关电容技术将感应开关电容电流转换为数字代码，从而进行电容式感测。

Figure 1. CSD 模块图



快速启动

1. 如果使用，选择并放置需要专用引脚 （例如 I2C 和 LCD）的用户模块。 根据需要分配端口和引脚。
2. 选择并放置 CSD 用户模块。
3. 设置所需的传感器、滑条或旋转滑条的数量。
4. 设置每个传感器的传感器设置。
5. 设置引脚和全局参数。 阅读所有参数说明，遵守各种要求和相关指南。
6. 生成应用，并切换到应用编辑器。
7. 根据需要调整采样代码，以部署独立传感器、滑条传感器或触摸板。
8. 将 I2C-USB 桥连接到目标板，观察信号。
9. 更改 CSD 参数以优化设置和重建应用。
10. 对 PSoC 设备进行编程，验证模块运行。调整 CSD 参数以实现 “*CapSense 应用信噪比要求*” 中所述的 5:1 SNR 要求 - AN2403.

功能说明

电容式传感器包括物理、电气和软件组件：

- 物理组件
  - 即物理传感器本身，通常其传导模式基于与 PSoC 相连的 PCB，PSoC 是一种显示屏外覆有绝缘层、软膜或透明覆盖层的装置。
- 电气组件
  - 用于将传感器电容转换为数字格式。 转换系统包括感应开关电容、sigma-delta 调制器和基于计数器的数字滤波器，可将调制器输出的位流转换为可读取的数字格式。
- 软件
  - 检测和补偿软件算法可以将计数值转换为传感器检测结果。
  - 对于连续、附属型传感器 （例如：滑条和触摸板），会提供 API，以便插入一个分辨率高于传感器物理分辨率的位置。 例如，可以创建一个包含 10 个传感器的量级滑条，通过提供的固件可以将量级扩展到 100。 另外，通过相同的 API，可以使用两个电容式传感器，以凸凹咬合方式排列，用于确定其之间的传导体 （例如手指）的位置。

测量电容有许多方法，此用户模块中采用开关电容与 delta-sigma 调制器配合使用的方法。

传感器阵列包含独立传感器、滑条传感器以及触摸板，触摸板部署为一对互相垂直的滑条。 高级决策逻辑提供对环境因素的补偿，例如：温度、湿度和电源电压变化。 单独的屏蔽电极可用于为传感器阵列提供屏蔽，减少杂散电容，从而在水膜或水滴存在的情况下提供更可靠的运行。

高级软件功能可提供滑条双工法，以便在两个物理位置可以使用一个电气传感器，用以提高分辨率。 通过这些功能，还可以在物理传感器位置之间进一步插补解析传感器位置。

建议在首次使用 CSD 用户模块之前阅读以下文档。

PSoC CY8C20x66、CY8C20x66A、CY8C20x46/96、CY8C20x46A/96A、CY8C20x36、CY8C20x36A 技术参考手册 (TRM)

阅读 CSD 用户模块文档之后，建议阅读下列应用说明。 可以在赛普拉斯半导体公司网站上找到应用笔记，网址如下： www.cypress.com:

- *CapSense Best Practices*（CapSense 最佳应用） - AN2394
- *Signal-to-Noise Ratio Requirements for CapSense Applications*（CapSense 应用信噪比要求） - AN2403
- *Charting Tool to Debug CapSense Applications*（调试 CapSense 应用的制表工具） - AN2397
- *EMC Design Considerations for PSoC CapSense Applications*（PSoC CapSense 应用的 EMC 设计注意事项） - AN2318
- *Power Consumption and Sleep Considerations in Capacitive Sensing Applications*（电容式传感应用的功率消耗和睡眠注意事项） - AN2360
- *Layout Guidelines for PSoC CapSense*（PSoC CapSense 设计指南） - AN2292
- *Software Implementation of a Universal Asynchronous Transmitter*（通用异步发射器的软件实现） - AN2399
- *Waterproof Capacitance Sensing*（防水电容传感） - AN2398

电容测量操作

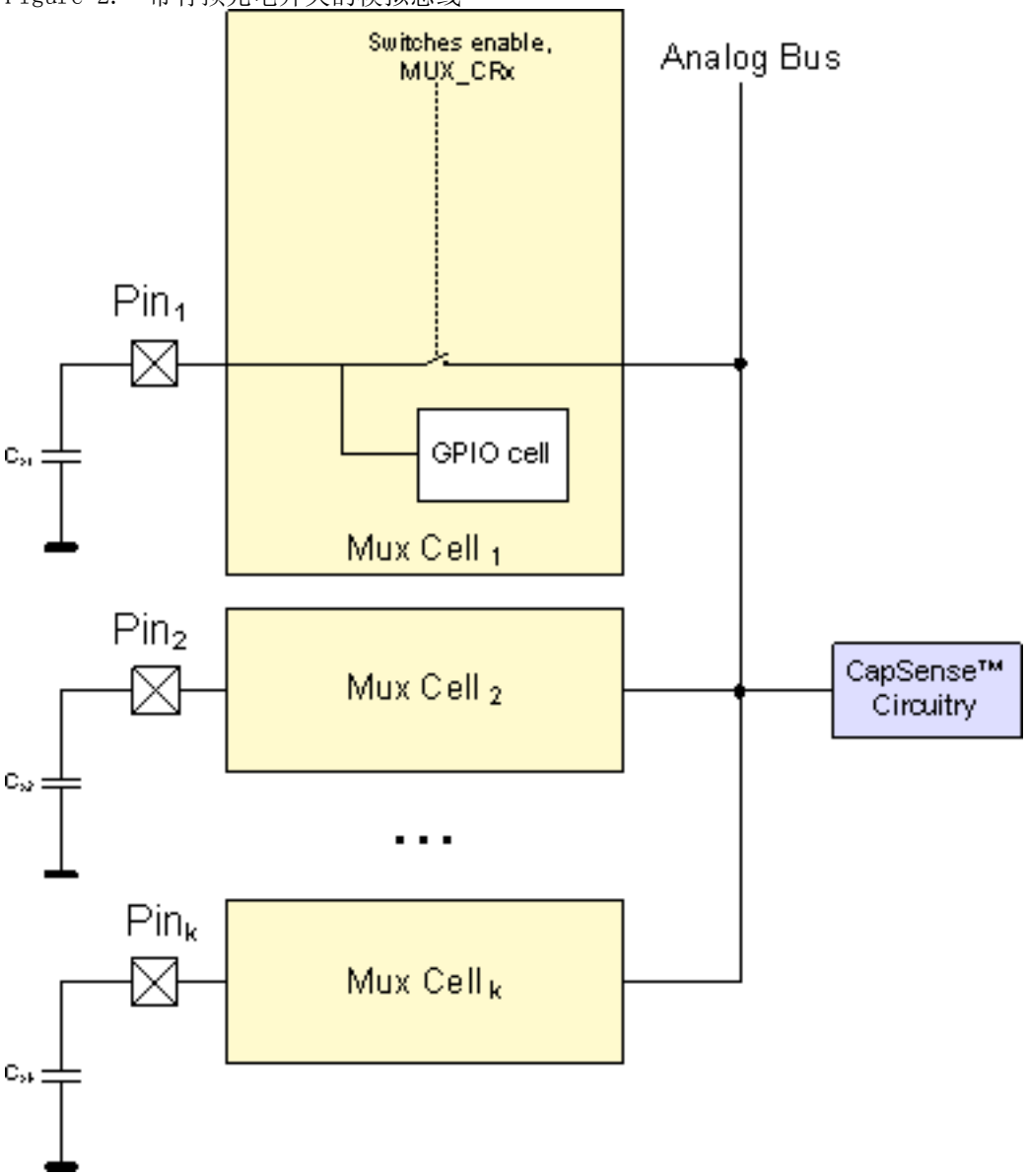
决策逻辑通过固件实现。 通过固件分析电容的测量，跟踪环境因素造成的缓慢电容变化，运行决策逻辑，以检测按钮触摸变化并计算滑条位置。

扫描传感器阵列

CY8C20x66 系列设备具有内置模拟总线， 可以使电容传感器连接到任意 PSoC 引脚。 CSD 用户模块使用内部预充电开关，在时钟信号 Ph_1 阶段为工作的传感器充电，在 Ph_2 阶段将模拟总线与传感器相连。 sigma-delta 调制器的调制电容和比较器的输入端与模拟总线始终相连。

固件通过在 MUX_CRx 寄存器中设置相应的位来执行传感器系列扫描。

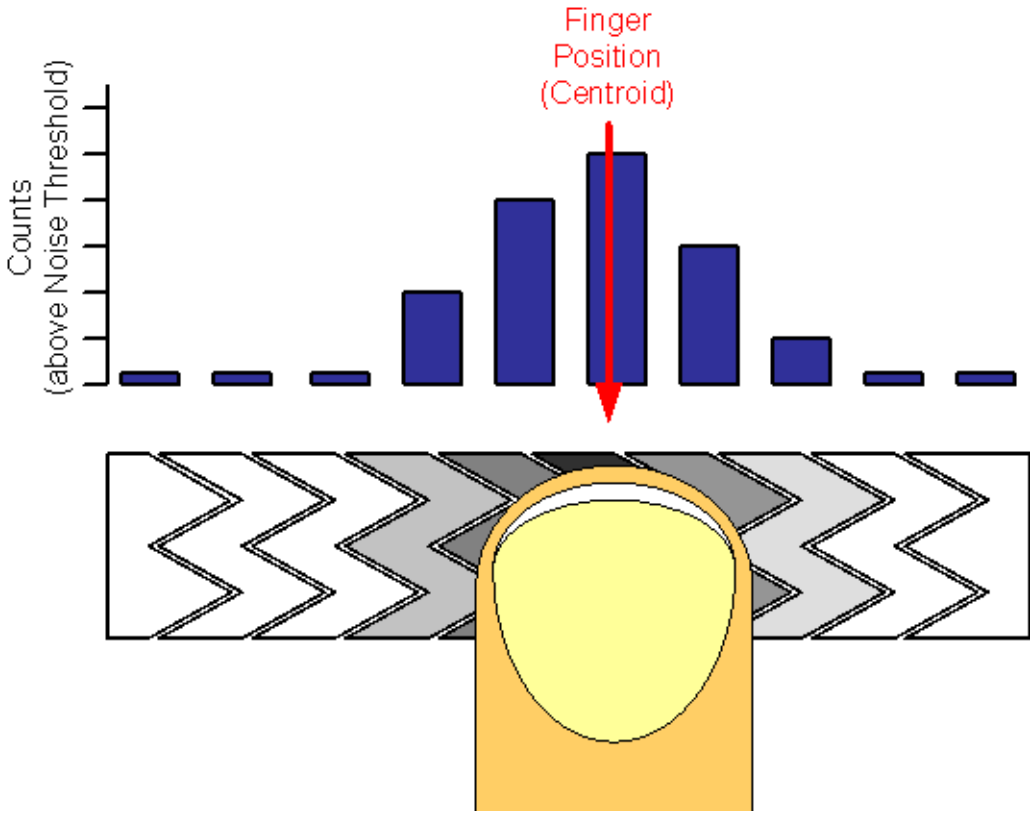
Figure 2. 带有预充电开关的模拟总线



滑条

滑条用于需要渐进调整的控件。 示例包括照明控件 （调光器）、音量控件、图示均衡器和速度控件。 这些传感器在布局上彼此相邻。 某个传感器的动作会导致邻近的其他传感器的部分动作。 通过计算已激活传感器组的质心位置，可以确定滑条的实际位置。 通过 CSD 向导可以调整滑条，方法是建立若干组，每组滑条有特定的次序。 传感器滑条数量的实际下限值是五，上限值仅取决于所选 PSoC 设备提供的传感器位置的数值。

Figure 3. 对物理传感器位置排序



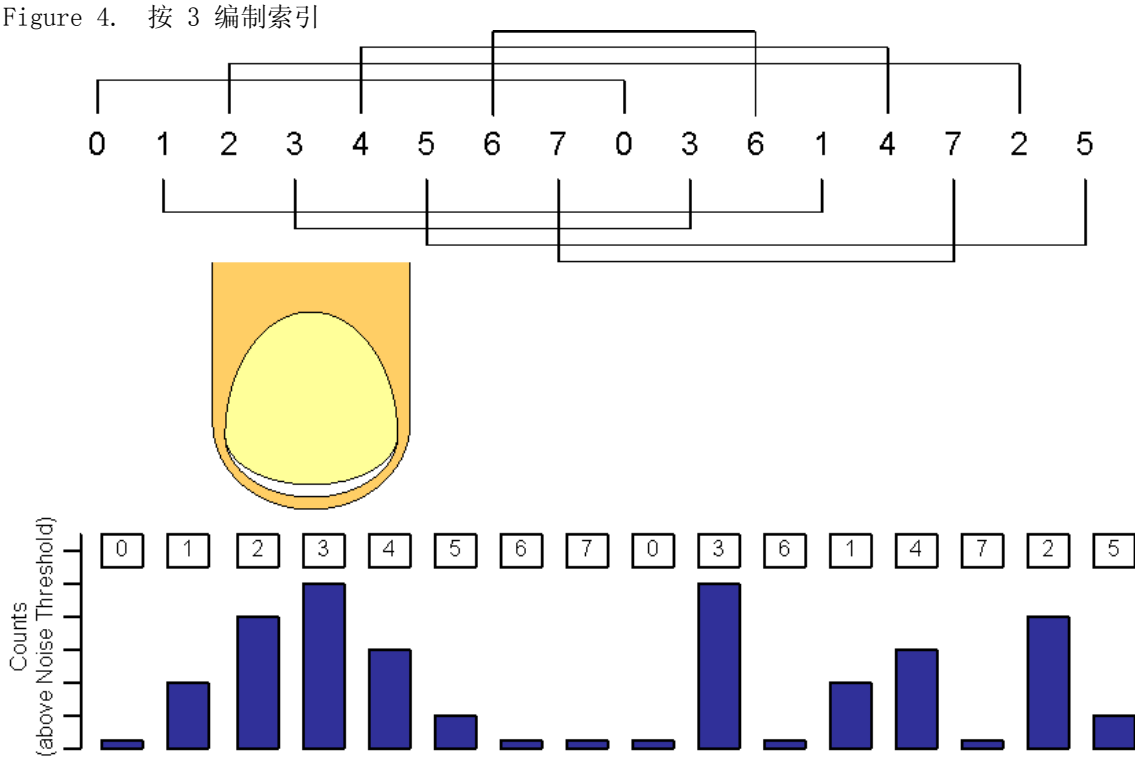
越接近滑条一半部分的强信号，将导致最上部分产生相同程度的伪信号，但最终是杂散信号。 感应算法搜索相邻最强的一组信号，以确定解析的滑条位置。

双工

滑条中的每个 PSoC 传感器连接都映射到滑条传感器阵列中的两个物理位置。 物理位置的第一部分 （较低数值部分）按顺序映射到基部分配的传感器上，端口引脚由设计人员使用 CSD 向导分配。 物理传感器位置的另一部分 （较高数值部分）由向导中的算法自动映射，并在包括文件中列出。 一旦创建好次序，某一部分中相邻的传感器动作则不会导致另一部分中相邻的传感器动作。 小心地确定此次序，将其映射到印刷电路板上。

有许多方法可对物理传感器位置的第二部分进行排序。 最简单的方法是对第一部分中的传感器编制索引，先对所有偶数传感器编制索引，然后是所有奇数传感器。 其他方法包括按相关值编制索引。 此用户模块选择的方法是按三编制索引。

Figure 4. 按 3 编制索引



应当使滑条中的传感器电容均衡。 根据传感器或 PCB 设计，某些传感器对可能需要更长的路由。 当选择双工时，双工传感器索引表由 CSD 向导自动生成。 下表列出了不同滑条段计数的双工序列：

Table 1. 不同滑条部分计数的双工序列

滑条段总计数	段序列
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
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

滑条段总计数	段序列
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

内插和测量

在滑动传感器和触摸板应用中，通常需要更精确地分辨手指 （或其他电容物体）的位置，而非单个传感器本身的分辨率。 手指接触滑动传感器或触摸板的面积通常大于任何一个传感器。

要使用质心计算内插位置，首先要扫描阵列，以确定所给的传感器位置是否有效。 要求提供一定数量的相邻传感器信号，且高于噪声阈值。 如果发现最强信号，将使用此信号和那些大于噪声阈值的连续信号计算质心。 使用少至两个、多至 （通常）八个传感器，利用下列公式计算质心：

Equation 1

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

计算出的值通常是分数。 要将质心报告给特定的分辨率 （例如：对于 12 个传感器，范围值 0 ～ 100），需要将质心数值乘以计算得出的量极。 另一种更有效的方法是将内插和按比例计算的方法统一到单一计算中，按所需的量级直接报告结果。 这是通过高级 API 实现的。

滑条传感器计数和分辨率在 CSD 向导中进行设置。 比例值通过向导计算，并存储为分数值。

质心分辨率的乘法器占用三个字节，相应位的定义如下：

分辨率乘法器 MSB								
位	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
乘法器	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
分辨率乘法器 ISB								
乘法器	128	64	32	18	16	8	4	2
分辨率乘法器 LSB								
乘法器	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256

使用以下公式确定分辨率：

分辨率 = (传感器数 - 1) x 乘法器

质心以 24-bit 无符号整数保存，其分辨率是传感器和乘法器数值的函数。

反馈组件选择指南

用户模块需要外部调制电容 C_{mod}，该电容从 Vss 地连接到 P0[1] 或 P0[3] 端口引脚。 通过用户模块参数设置可以选择引脚。 不要将选定用于调制器组件连接的引脚用于其他任何目的。

外部调制电容的建议值为 4.7 到 47.0 nF。 可以通过实验选择最佳的电容，以获得最大的信噪比。 在大多数情况下，5.6 到 10.0 nF 值具有很好的效果。 可以试验一些电容值以获取最佳信噪比。 应当使用陶瓷电容。 温度电容系数并不重要。

正确的 iDAC 值取决于传感器总电容 C_S。 选择该值的目的是：

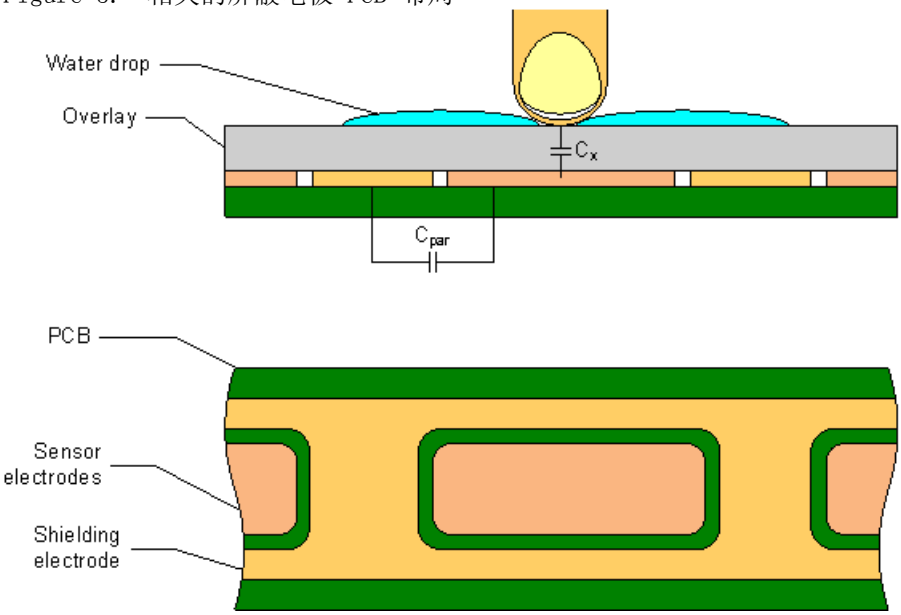
- 监控不同传感器触摸的原始计数。
- 在选定的扫描分辨率下，选择最大读数大约比全量程读数低 30% 的 iDAC 值。 当 iDAC 值下降时原始计数增大。

屏蔽电极

某些应用场合要求即使存在水膜或水滴，也能可靠地运行。 白色家电、汽车、各种工业领域和其他领域应用，都需要使用不会因为水、冰和湿度的变化而提供假触发信号的电容式传感器。 对于此种情况，可以使用单独的屏蔽电极。 此电极位于感应电极之后或其外侧。 如果设备绝缘覆盖层表面有水膜，则屏蔽和感应电极之间的耦合程度会加剧。 屏蔽电极有帮于降低寄生电容的影响，为处理传感电容的变化提供了更具动态性的数值范围。

在某些应用情况下，选择屏蔽电极信号以及屏蔽电极相对于传感电极的放置位置是非常有用的，这样可增加两种电极之间的耦合程度，使感应电极电容测量的触摸变化朝着相反方面改变。 这样可以简化高级软件 API 的工作。 CSD 用户模块支持屏蔽电极的单独输出。

Figure 5. 相关的屏蔽电极 PCB 布局



提图 5 供一种按钮屏蔽电极的可能布局配置。 屏蔽电极尤其适用于透明的 ITO 触摸板设备，在这种设备中，它不但可阻止 LCD 驱动电极的噪声影响，同时可减少杂散电容。

在此示例中，按钮覆盖有屏蔽电极板。 作为另一替代方法，屏蔽电极可以安装在相对的 PCB 层上，其中包括按钮下面的平板。 在这种情况下，建议使用开口方案，填充率大约为 30 到 40%。 在此种情况下，无需附加的接地层。

如果屏蔽电极与感应电极间存在水滴，C_{par} 将增加，调制器电流下降。 在实际测试中，通过 API 可以增大调制器参考电压，以便因水滴引起的原始计数增加值能够接近于零或略呈负值。 可以通过选择适当的调制器参考值来实现此目的。

在此用户模块中，用于预充电时钟的同一信号还提供给屏蔽电极。

屏蔽电极可以连接到专用 PSoC 引脚 （P0[7] 或 P1[2]）。 选定引脚的驱动模式应当设置为**强**。 另外，可以在 PSoC 设备与屏蔽电极之间连接上升限制电阻。

时钟源

时钟源用于控制感应电容上的开关。 用户模块支持将下列两个选择选项作为预充电开关的时钟源：

- 12-bit 伪随机序列发生器 (PRS)
- 直接 IMO 分频器

可以通过相应的用户模块参数选择所需的配置。

PRS 源提供扩频操作，确保很好地抵抗外部噪声源的噪声。 另外，带有扩频时钟的设计能够达到较低的电磁辐射级别。 如果应用旨在通过 EMC/EMI 测试或者必须在嘈杂环境下提供可靠操作，则建议使用 PRS 时钟源。

下表比较了这两种时钟源：

时钟源	操作频率	抗 EMC 噪声能力
PRS	扩频，平均值为 F_{IMO}/4/ 预分频器，峰值为 F_{IMO}/2/ 预分频器	高。 敏感点是 PRS 序列重复周期和 PRS 基本频率 F_{IMO}/ 预分频器的倍数。
直接 IMO 预分频器	固定频率 F_{IMO}/ 预分频器	中等。 有更多敏感点。

直流和交流电气特性

Table 2. 噪声

参数	最小值	典型值	最大值	单位	测试条件（Vdd = 3.3V，SysClk = CPU 时钟 = 24 MHz）
噪声计数，峰 - 峰	--	0.2	--	%（噪声计数）/（基线计数）	分辨率 >= 14
噪声计数，峰 - 峰	--	0.5	--	%（噪声计数）/（基线计数）	分辨率 = 12
噪声计数，峰 - 峰	--	1	--	%（噪声计数）/（基线计数）	分辨率 = 10

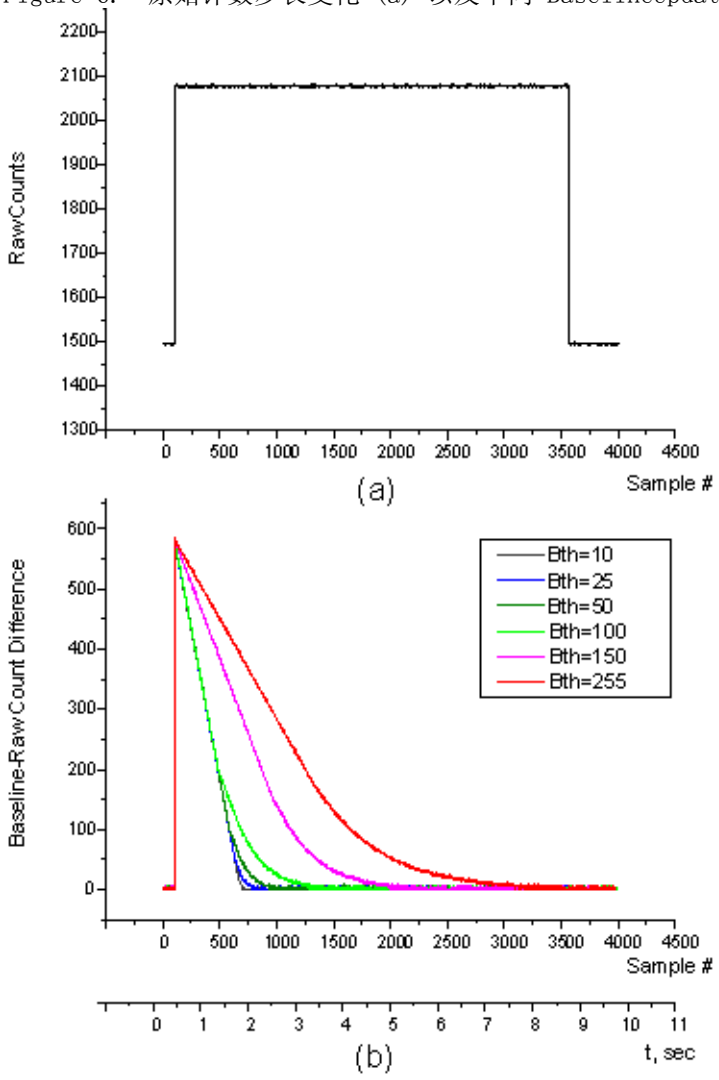
Table 3. 电源电压

参数	最小值	典型值	最大值	单位	测试条件和注释
值	1.7	5.0	5.25	V	

特征图

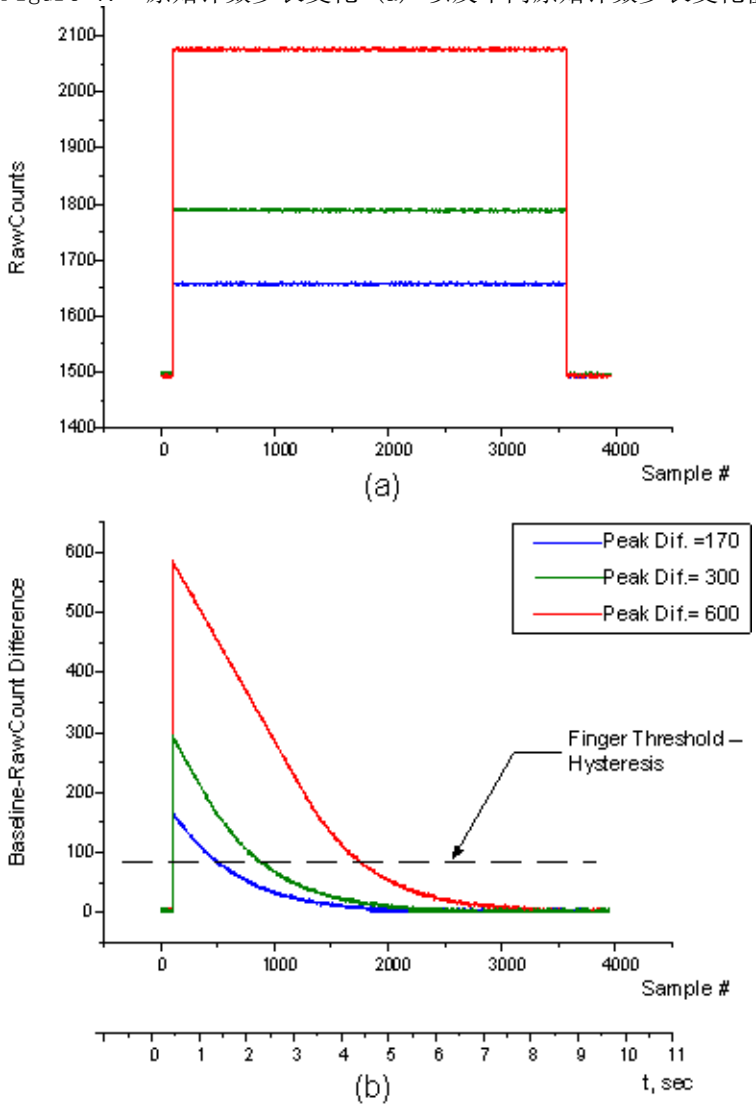
测试条件：12-bit 分辨率，SensorsAutoreset = 已启用，传感器阵列扫描和数据传输时间总计大约为 2.5 毫秒。

Figure 6. 原始计数步长变化 (a) 以及不同 BaselineUpdate 阈值 (Bth) 参数值的原始计数与基线之差 (b)



Note 增大 BaselineUpdate 阈值会降低差的减小程度，使最大按钮触摸检测时间加长。

Figure 7. 原始计数步长变化 (a) 以及不同原始计数步长变化值的原始计数与基线之差 (b)。



测试条件：12-bit 分辨率，SensorsAutoreset = 已启用，传感器阵列扫描和数据传输时间总计大约为 2.5 毫秒。 BaselineUpdate 阈值参数设置为 255。

Note 原始计数步长值越大，传感器自动复位启动所需的时间越长，从而需要更长时间才能将差降低到 FingerThreshold-Hysteresis 值之下。

放置

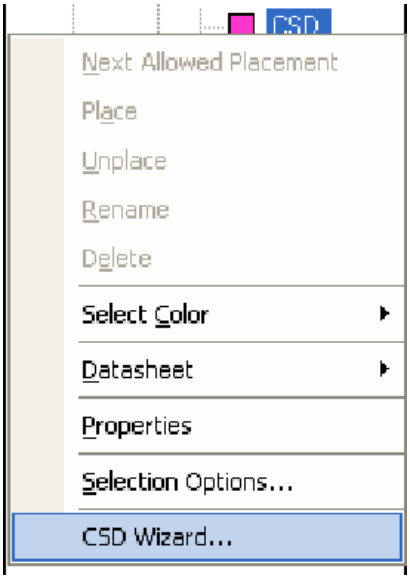
当实例化用户模块时，会自动放置适用于用户模块的模块，不提供其他放置方式。 CSD 用户模块使用 CapSense 模块和一个定时器 (Timer1)。

必须在启动 CSD 向导之前放置需要特定引脚资源 （包括 LCD 和 I2CHW）的用户模块，以建立 CSD 用户模块的引脚连接。

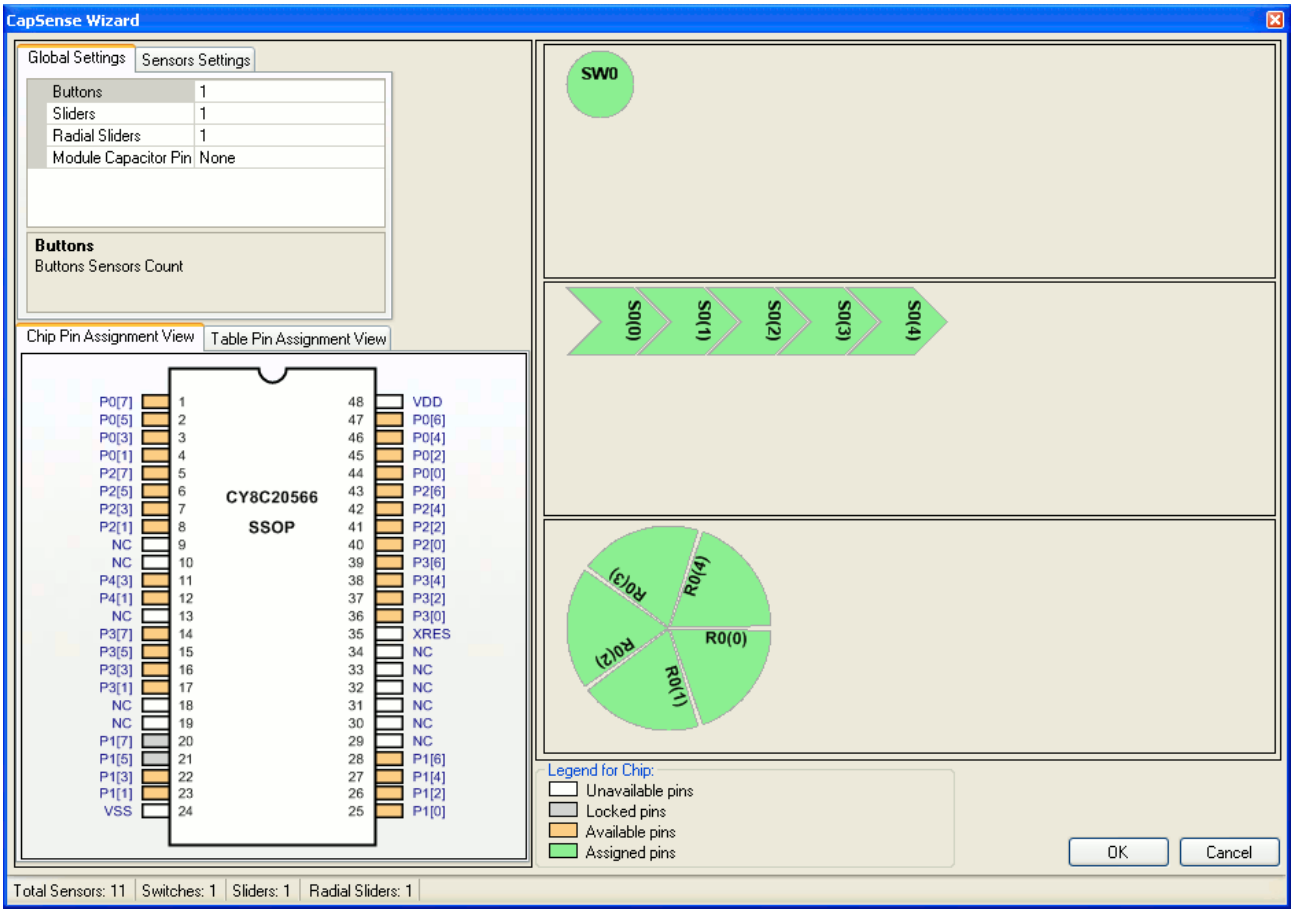
当放置电容式传感器连接时，避免使用 P1[0] 和 P1[1]。 这些引脚用来对部件进行编程，可能具有过多影响传感器灵敏度和噪声的路由电容。

“向导”

1. 要访问向导，请在设备编辑器互连视图中右键单击任意 CSD 块，然后通过鼠标左键单击选择 “CSD 向导”。



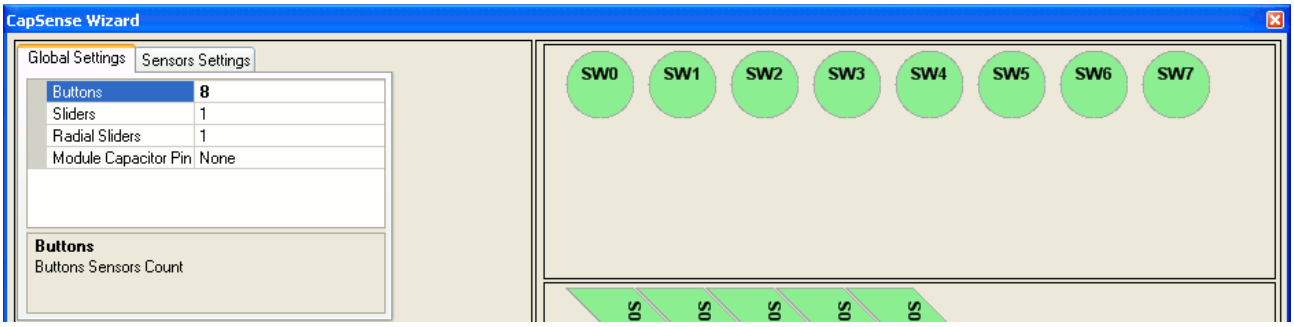
2. 向导打开，其中显示了按钮传感器数、滑条数和辐射状滑条数的数值输入框。



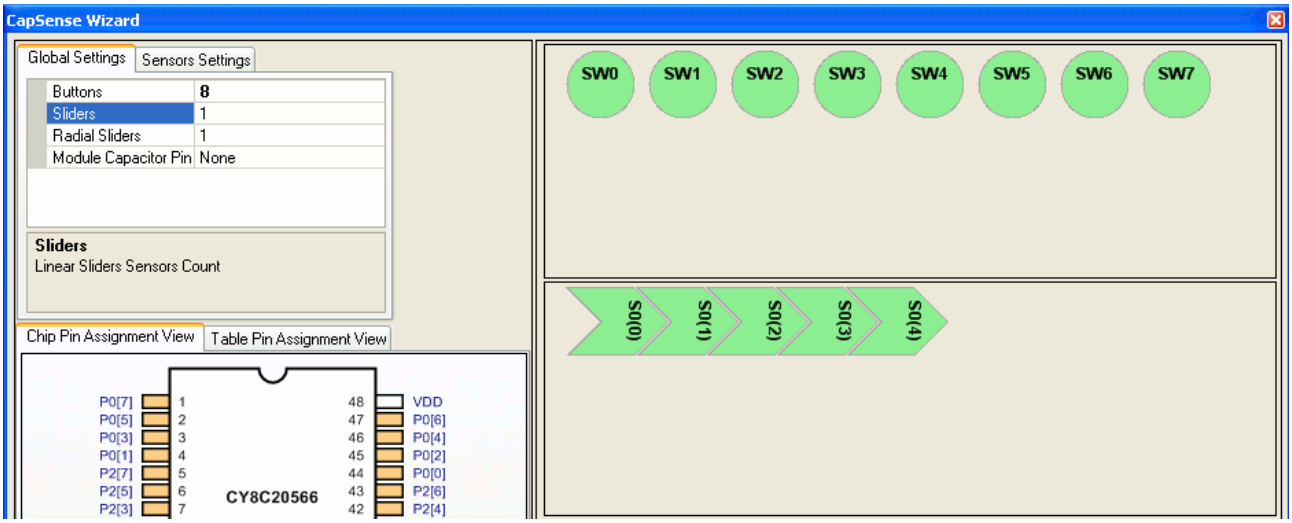
向导引脚图标

- 白色 – 引脚不能用作 CapSense 输入。
- 灰色 – 引脚处于锁定。这种情况有两种可能的原因。一种可能的原因是另一个用户模块（如 LCD 或 I^2C）已声明了该引脚。第二种可能是引脚名称已发生更改，不再是默认值。要将引脚名称恢复为其默认值，请在引脚输出视图中展开引脚，从**选择**菜单，选择**默认值**。现在可以在向导中分配引脚了。
- 橙色 – 引脚可用于分配。
- 绿色 – 引脚已分配为 CapSense 输入。请键入独立传感器数量。传感器数量限制为可用引脚的数量。按 [Enter] 键以输入传感器数的新值。

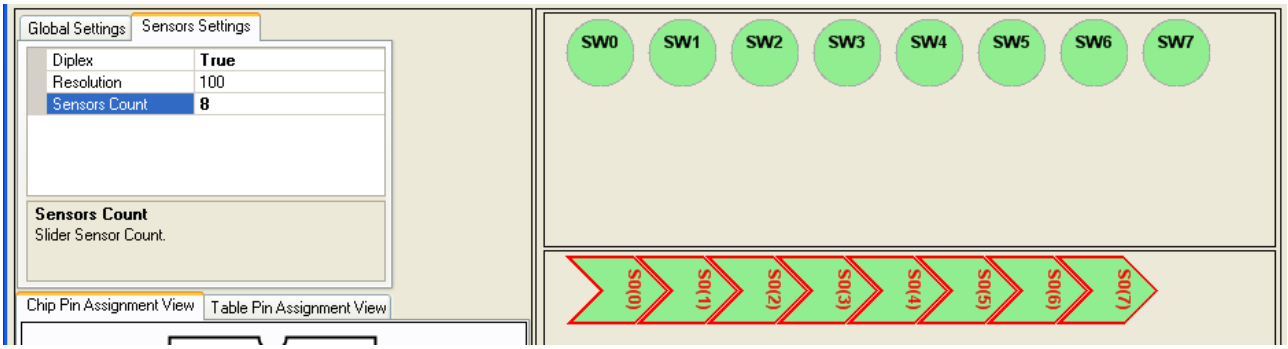
3. 输入设计所需的按钮传感器数。



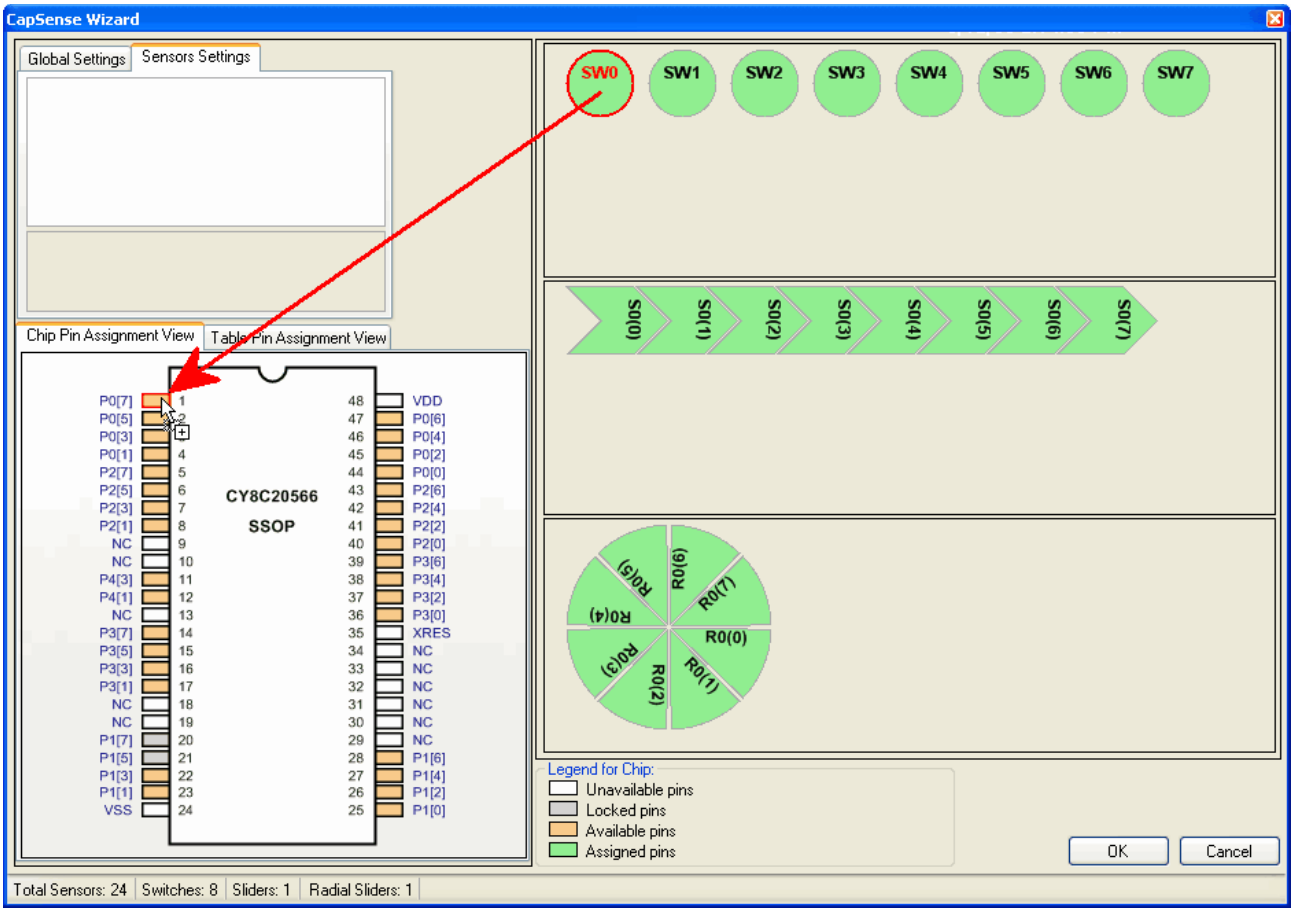
4. 键入滑条的数量。X-Y 触摸板需要两个滑条。



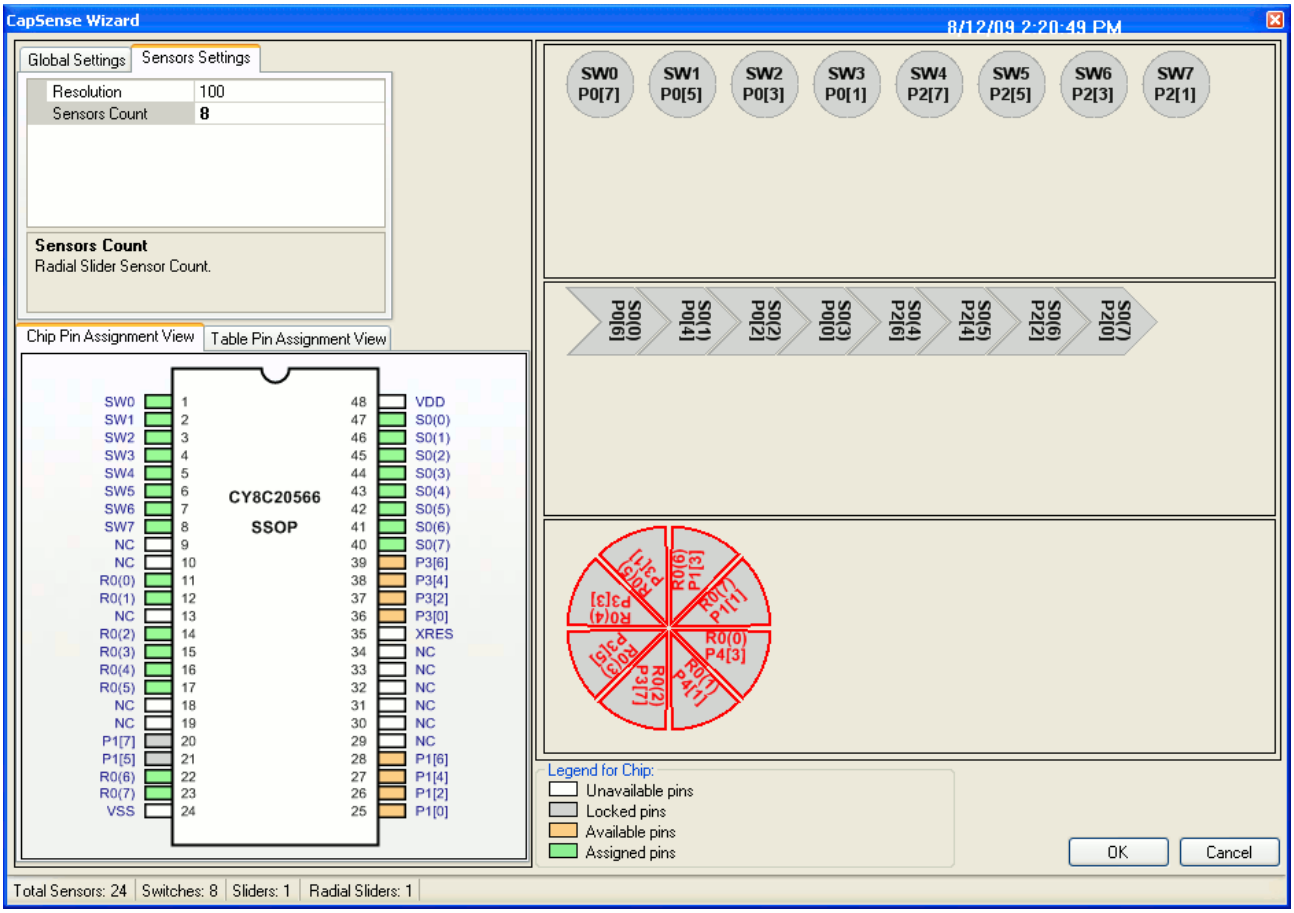
5. 单击滑条以启用传感器设置。选择 **传感器设置** 选项卡。键入每个滑条中的传感器元素数。滑条传感器中的传感器实际最小值为五，最大值受引脚计数限制。
6. 如果需要，选择 “ 双工 ”。这会将为传感器选定的引脚数值映射为板上传感器位置的两倍数值。仅显示了双工传感器的前一半部分；后一半部分按前面 “ 双工 ” 章节所述自动映射。有关引脚连接的双工表，请参见 “ 双工 ” 一节。
7. 键入输出分辨率。最小值为五。对于双工型滑条，最大值为（传感器的引脚数 – 1）x 2^{16} – 1 或（2 x 传感器的引脚数 – 1）x 2^{16} -1。



8. 创建辐射状滑条的过程与创建线性滑条的过程相同，但是不能对辐射状滑条进行双工处理。
9. 通过在引脚分配视图中将开关或传感器拖动到引脚，可以将开关或传感器分配给引脚。 可以选择在芯片引脚分配视图或表引脚分配视图中将开关或传感器拖动到引脚。 选择端口引脚后，引脚变为绿色，不再可用。 通过将传感器拖离端口引脚，更改传感器分配。



10. 对于其余独立传感器，重复操作。 将物理端口引脚映射到单个滑条传感器与映射到单个传感器相同。 单击**确定**接受数据，然后返回到 PSoC Designer。



传感器放置现在已完成。 在设备编辑器窗口中右键单击，选择**刷新**更新引脚连接。

设置用户模块参数，并生成应用。 如果需要，可以立即对示例项目进行调整。

如果想要更改引脚分配，请将光标放在分配的引脚上，单击引脚，拖放至开关框外侧。 该引脚分配取消，然后可以将其重新分配。

完成向导后，单击 “ 生成应用程序 ”。 根据传感器计数输入、引脚分配、双工和分辨率，生成一组表。 表位于 CSD_Table.asm 中

传感器表

在传感器表中，每个传感器对应一个 2-byte 条目。 第一个字节是端口号，第二个字节是位的掩码 （不是位编号）。 该表包含所有独立传感器，然后按次序列出每个传感器。 下面是一个包含六个传感器的表的示例：

```
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
```

该表由 CSD_wGetPortPin() 例程使用。

组表

组表对每一组按钮传感器或滑条进行定义。 每个滑条对应一个条目，另外还有一个条目用于备用按钮传感器。 第一个条目始终是备用传感器。 每个条目为六字节。 第一个字节是传感器表中组开始的索引。 第二个字节是该组中的传感器数量。 第三个字节指示滑条是否为双工 （4 为双工，0 为非双工）。 第四、五、六个字节是固定值乘数，将其与计算出的滑条质心相乘可以获得 CSD 向导中所需的分辨率。

```
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
```

双工表

当某个组是滑条且采用双工时，将为该组生成双工表扫描顺序数据。 否则，将创建标签而不放置数据。 该表由两个部分组成：针对每个滑条的传感器映射，以及每个单独滑条对其表格的引用。 八个传感器滑条的典型示例为：

```
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
```

参数和资源

Autocalibration

如果启用此参数，它设置 Idac 电流以使传感器的原始计数值设置为 (2^N)-1 的 85%，其中 N 是分辨率。设备编辑器中设置的 Idac 值被覆盖。

如果 AutoCalibration 参数设置为禁用，则根据设备编辑器中设置的 IDAC 范围、IDAC 值、分辨率、传感器电容和 IMO 频率、预分频器、预充电源参数和 Vref 计算原始计数。

Autocalibration 仅包括在 IDAC 配置中。 Autocalibration 自动选择可能的 IDAC 值以获取分辨率一半范围中的原始计数。 这会降低 CapSense 算法的整体灵敏度，但是它允许开始调整过程时快速获取可读范围中的原始计数。 Autocalibration 使用 ROM 和 RAM 资源，增加了启动时间。 如果校准后的原始计数值小于分辨率范围的一半，则应当增大 IDAC 范围或降低预充电频率。 Autocalibration 用于部分提高功能配置。

手指阈值

此阈值用于确定每个按钮传感器的状态。 如果任何传感器处于活动状态，则 bIsAnySensorActive() 函数返回 1。 如果所有传感器关闭，则 bIsAnySensorActive() 函数返回 0。

手指检测阈值适用于所有传感器和滑条。 对于单个传感器 （滑条组中不包含），这些阈值是变量，在 baBtnFThreshold[] 阵列中提供。 可以使用 SetDefaultFingerThresholds() 函数将阈值设置为设备编辑器中默认值设置。 要调整单个传感器的灵敏度，请更改每个传感器的 baBtnFThreshold[] 值。 （此字节阵列的大小等于部署的单个传感器的数量。）

可能值的范围为从 5 到 255。

噪声阈值

对于单个传感器，高于此阈值的计数值不会使基线更新。 对于滑条传感器，质心计算中不考虑低于此阈值的计数值。 可能值为 5 到 255。

基线更新阈值

如果新的原始计数值高于当前基准，差值低于噪声阈值 （传感器自动复位参数设置为 “ 禁用 ”），则当前基线与原始计数的差值累计到桶形变量（ 水桶 ）中。 当水桶充满时，基准按某个值递增，并清空桶形电极。 此参数用于设置为了使水桶必须达到基线而要增大的阈值。 可能值为 0 到 255。 参数值越大，基线更新速度越慢。 如果需要更频繁的基线更新，请减小此参数。

LowBaselineReset

LowBaselineReset 参数与 NegativeNoiseThreshold 参数协同工作。 对于指定数量的样品，如果样品计数值低于基线与 NegativeNoiseTreshold 的差值，则基线设置为新的原始计数值。 此参数实际上是对需要复位基准的异常低的样品数值进行计数。 它通常用于更正启动时手指位置的情况。

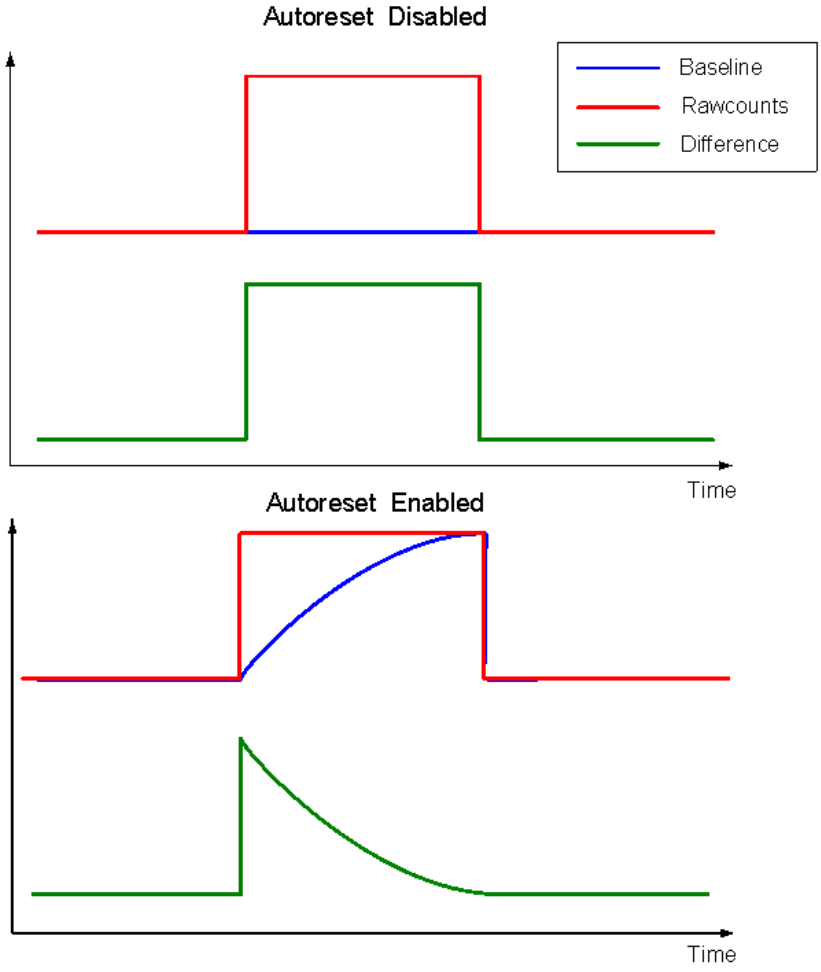
传感器自动复位

此参数确定基准是否随时更新，还是仅当信号差值低于噪声阈值时更新。 当设置为**启用**时，基准随时更新。 此设置限制传感器的最大持续时间 （典型值为 5 – 10s），但是当无任何物体触碰传感器而原始计数突然上升时，可以阻止传感器始终打开。 这一突然上升可能是由电源电压剧烈波动、高能射频噪声源或温度快速变化所导致。

当此参数设置为**禁用**，则仅当原始计数与基准的差低于噪声阈值参数时基准才进行更新。 除非是遇到在任何物体未触碰传感器而原始计数突然上升时传感器始终打开的问题，否则应将此参数保留为 “ 禁用 ”。

说图 8 明了此参数对基准更新的影响：

Figure 8. 传感器自动复位参数



迟滞

“ 迟滞 ” 参数根据传感器当前是处于活动还是非活动，来增大或减小手指阈值。 如果传感器处于非活动状态，则差值计数必须大于手指阈值与迟滞的和。 如果传感器处于活动状态，则差值计数必须低于手指阈值与迟滞的差。 此参数用于增加手指检测算法的防反跳和粘着性程度。 当调用 bIsSensorActive() 或 bIsAnySensorActive() 时，计算带有迟滞的阈值。 可以用 bIsSensorActive() 或 baSnsOnMask[] 阵列的返回值监控传感器状态。 可能的值为 0 到 255，但是必须小于 “ 手指阈值 ” 参数设置。

只有正确选择高级决策逻辑参数，才能高效补偿环境温度因数 （温度、湿度变化等），抑制噪声信号（ESD, 电源尖峰脉冲），并在各种情况下提供可靠触摸检测。

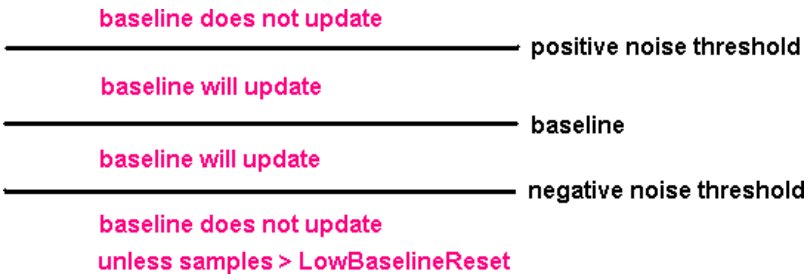
防反跳

“ 防反跳 ” 参数为传感器活动的瞬变增加了防反跳计数器。 为了让传感器从不活动转换到活动状态，对于指定的采样数，差计数值必须大于手指阈值与迟滞之和。 防反跳计数器按 bIsSensorActive 或 bIsAnySensorActive API 函数递增。

可能值为 1 到 255。 设置为 1 则不提供防反跳。

NegativeNoiseThreshold

NegativeNoiseThreshold 参数增加负的差值计数阈值。 如果当前原始计数低于基准且它们的差大于此阈值，则不更新基准。 但是，如果对于 LowBaselineReset 参数所指定的样品数量，当前原始计数处于较低状态 （差大于阈值），则对基准进行复位。



扫描速度

此参数影响传感器的扫描速度。 可用的选择有： **超快、快速、正常、慢速**。 较慢的扫描速度具有下列好处：

- 提高了信噪比
- 更好地应对电源和温度的变化

扫描速度在以下方面影响扫描速度除数：

扫描速度	除数
超快	1
快速	2
正常	4
慢速	8

分辨率

此参数确定扫描分辨率 （以 bit 为单位）。 可以用 9 到 16 位的分辨率来扫描传感器。 N 位的扫描分辨率最大原始计数为 2^N-1。

增大分辨率可提高触摸检测的灵敏度和信噪比。 对于接近检测，请使用高分辨率。 通过 16-bit 分辨率、慢速扫描模式和一根 20 cm 导线，可以在 20 cm 或更远距离检测到人手。

Table 4. 24 MHz IMO 操作的扫描时间 （以微秒为单位）、扫描速度和分辨率

分辨率 （以 bit 为单位）	扫描速度			
	超快	快速	正常	慢速
9	57	78	125	205
10	78	125	205	380
11	125	205	380	720
12	205	380	720	1400

分辨率 （以 bit 为单位）	扫描速度			
	超快	快速	正常	慢速
13	380	720	1400	2800
14	720	1400	2800	5600
15	1400	2800	5600	11000
16	2800	5600	11000	22000

Note 扫描时间是按两次传感器扫描的时间间隔测量的。 此时间包括传感器设置时间、调制器稳定延迟、采样转换间隔和数据预处理时间。

调制器电容引脚

此参数设置引脚以连接外部调制器电容 (C_{mod})。 从可用引脚 P0[1]、P0[3] 选择。如果使用内部电容，则选择 “ 无 ”。 通常，使用外部电容可获得更好的信噪比。

iDAC 值

电容测量范围取决于此参数。 值越高，范围越大。 调整 iDAC 值以获取大约整个范围的 50-70% 的原始计数。 可以在运行时使用相应的 API 函数更改此参数。 可能值为 1 到 255。

预充源

此参数选择预充电开关的时钟源。 允许的选项有 PRS 和预分频器。 大多数情况下，使用 PRS 源可获得较好的抗 EMI 能力和较低的辐射。

PRS 分辨率

此参数更改 PRS 序列长度。 可能值为 8 和 12 比特。 对应的序列长度为 511 和 2047 输入时钟周期。 如果 8-bit 设置未达到足够的 5:1 信噪比，则使用 12-bit 设置。

预分频器

此参数设置预分频器比例，并确定预充电开关输出频率。 此参数还影响 PRS 输出频率。

可能值有：

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

ShieldElectrodeOut

屏蔽电极信号源可以路由到 P0[7] 或 P1[2]。

Idac 范围

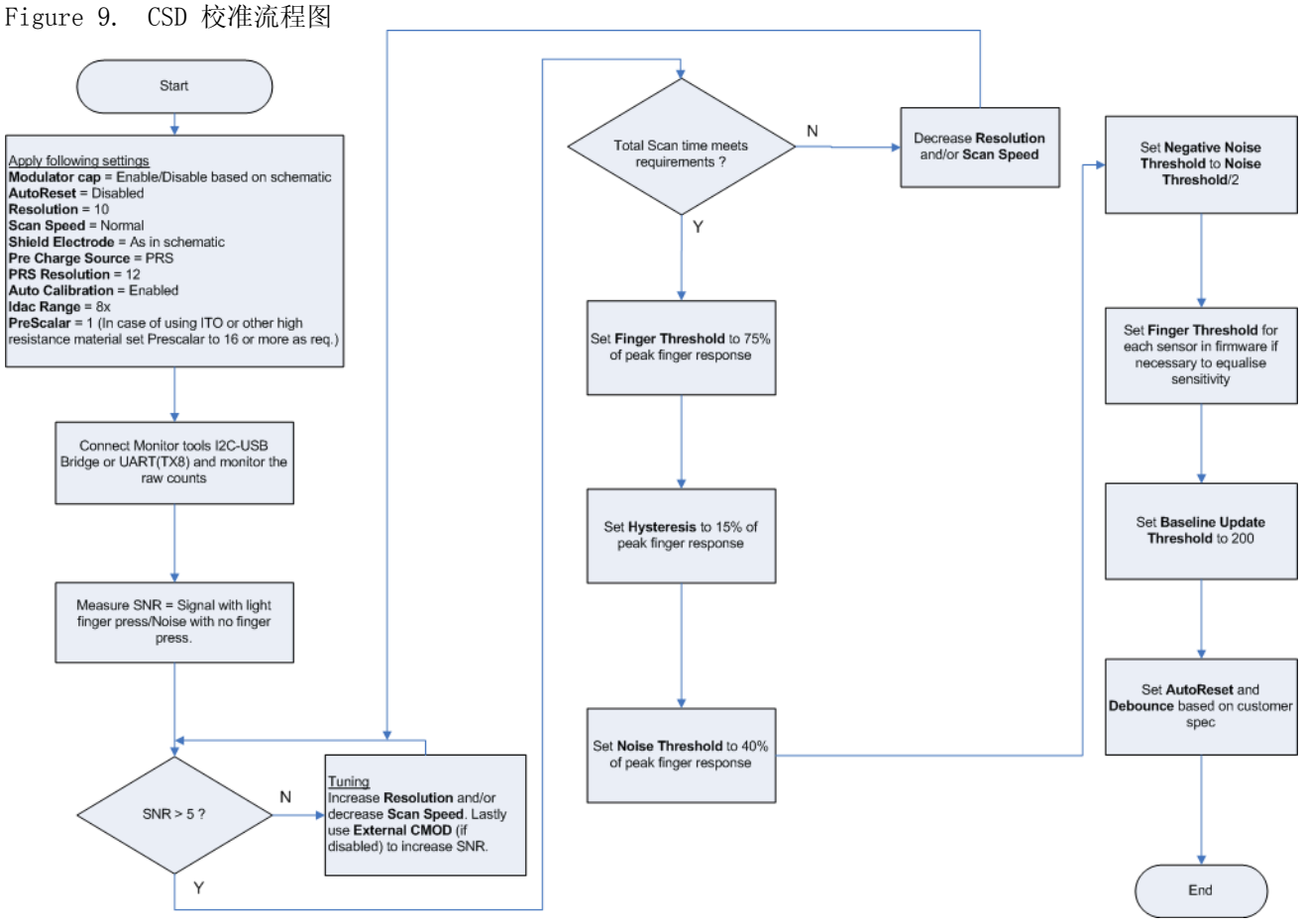
Idac 范围参数设置 IDAC 电流输出量程。此参数包含下列选择：

ADC 时钟	说明
1x	IDAC 输出量程为 1X 范围
2x	IDAC 输出量程为 2X 范围
4x	IDAC 输出范围为 4X 范围
8x	IDAC 输出范围为 8X 范围

CSD 校准

为了获得最佳性能，可以通过实际的 CapSense 硬件和外覆层调整 CSD 参数。 显图 9 示了如何校准 CSD：

Figure 9. CSD 校准流程图



1. 开始操作时采用 CSD 用户模块的默认设置。
2. 使用 I^2C-USB 电桥或 UART 以及实际的硬件和外覆层，捕获传感器的原始计数、基准和差计数。
3. **粗调。** 检查信噪比是否大于 5。 如果信噪比小于 5，则通过下列建议的 PCB 指南增加信噪比：增加 CSD 的分辨率，使用外部 C_{mod}，降低 CSD 的扫描速度。 有关 PCB 指南，请参考应用说明 “CapSense Best Practices” （CapSense 最佳做法） AN2394. 有关信噪比和信噪比测量方法的详细信息，请参考应用说明 “Capacitance Sensing – Signal-to-Noise Ratio Requirement for CapSense Applications” （有关 CapSense 应用的电容感应检测的信噪比要求）。 AN2403.

4. 检查所有传感器的总扫描时间是否符合要求。 如果不符合，请降低分辨率和 / 或增加扫描速度。 由于这些参数也影响信噪比，请返回到步骤 3。 通过几次尝试，找到可以生成最佳信噪比和所需的扫描时间的最佳分辨率和扫描速度参数。
5. 当激活按钮时捕获差值计数。 将手指阈值参数设置为峰值的 75%。
6. 将噪声阈值设置为峰值的 40%。
7. 将迟滞设置为峰值的 15%。
8. 将负噪声阈值设置为噪声阈值的一半。
9. 如果需要，设置单个传感器的手指阈值。 这可以通过写入到固件中的 CSD_baBtnFThreshold 阵列来实现。
10. 根据需要设置基准更新阈值。 更新基准的频率必须依据项目之间的关系决定。 基准应当是慢速移动参考，这有助于减少噪声和温度对电容式传感器的影响。
    - **快速更新基准速率：** 如果使手指缓慢移向按钮，则会出现问题。 这称为 “ 基准速率超过手指速率 ”。
    - **慢速更新基准速率：** 这样会使按钮易于受到温度波动的影响，可能导致 “ 按钮锁定 ”。
11. 根据需要设置 “ 自动复位 ” 和 “ 防反跳 ” 参数。 请参考 息参数和资源。

应用程序编程接口

应用程序编程接口 (API) 函数作为用户模块的一部分提供，从而能够采用更高级的方式处理模块。 本节指定每个函数的接口，以及引用文件所提供的相关常量。

每次放置用户模块时，都会为其分配一个实例名称。 默认情况下，PSoC Designer 会为指定项目中此用户模块的第一个实例分配 CSD_1。 可将该值更改为符合标识符语法规则的任意唯一值。 分配的实例名称成为每个全局函数名称、变量和常量符号的前缀。 在下面的说明中，为了简单起见，实例名称缩写为 CSD。

注意 ** 此种情况如同所有用户模块的 API，A 和 X 寄存器的值可以通过调用 API 函数来更改。 如果在调用后需要 A 和 X 的值，则调用函数负责在调用前保留 A 和 X 的值。 选择此 “ 寄存器易失 ” 策略旨在提高效率，自 PSoC Designer 1.0 版起已强制使用此策略。 C 编译器自动遵循此要求。 汇编语言编程人员也必须确保他们的代码符合此策略。 虽然一些用户模块 API 函数可以保留 A 和 X 不变，但是无法保证它们将来也会如此。

提供了进入点以初始化 CSD、启动其采样和停止 CSD。 在所有情况下，模块的实例名称会替换在下列进入点中显示的 CSD 前缀。 未能使用正确的名称是常见的语法错误原因。

API 函数使用不同的全局阵列。 不应手动更改这些阵列。 但是，可以出于调试目的检查这些值。 例如，可以使用绘图工具显示阵列的内容。 下面是一些全局阵列：

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

CSD_waSnsBaseline[] – 这是一个整数阵列，其中包含每个传感器的基准数据。 阵列大小等于传感器计数。 CSD_waSnsBaseline[] 阵列通过下列函数更新：

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

CSD_waSnsResult[] – 这是一个整数阵列，其中包含每个传感器的原始数据。 阵列大小等于传感器计数。 CSD_waSnsResult[] 数据通过下列函数更新：

- CSD_ScanSensor();
- CSD_ScanAllSensors().

CSD_waSnsDiff [] – 这是一个整数阵列，其中包含原始数据与每个传感器基准数据之间的差值。 阵列大小等于传感器计数。

CSD_baSnsOnMask[] – 这是一个保持传感器开或关状态的字节阵列 （对于按钮或滑条）。
CSD_baSnsOnMask[0] 包含传感器 0 到 7 的掩码位 （传感器 0 为 0 位，传感器 1 为 1 位）。
CSD_baSnsOnMask[1] 包含传感器 8 到 15 的掩码位 （如果需要），依此类推。 此字节阵列包含的元素数足以包含所有放置的传感器。 如果按钮开启，则位的值为 1；如果按钮关闭，则位的值为 0。
CSD_baSnsOnMask[] 数据由 CSD_blsSensorActive(BYTE bSensnor) 函数或 CSD_bIsAnySensorActive() 例程更新。

CSD_Start

说明：

初始化寄存器并启动用户模块。 此函数应当在调用任何其他用户模块函数之前调用。

C 原型：

```
void  CSD_Start()
```

汇编：

```
lcall  CSD_Start
```

参数：

无

返回值：

无

副作用：

**

CSD_Stop

说明：

停止传感器扫描仪，禁用内部中断，调用 CSD_ClearSensors() 以将所有传感器复位为非活动状态。

C 原型：

```
void  CSD_Stop()
```

汇编：

```
lcall  CSD_Stop
```

参数：

无

返回值：

无

副作用：

**

CSD_Resume

说明：

在 CSD_Stop 调用后恢复用户模块操作。

C 原型：

```
void  CSD_Resume()
```

汇编：

```
lcall  CSD_Resume
```

参数：

无

返回值：

无

副作用：

**

CSD_ScanSensor

说明：

扫描所选传感器。 每个传感器在传感器阵列中有唯一编号。 此编号由 CSD 向导按顺序分配。 Sw0 为传感器 0，Sw1 为传感器 1，依此类推。

C 原型：

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

汇编：

```
mov A,  bSensor
lcall CSD_ScanSensor
```

参数：

A => 传感器编号

返回值：

无

副作用

**

CSD_ScanAllSensors

说明：

通过调用每个传感器索引的 CSD_ScanSensor()，扫描所有已配置的传感器。

C 原型：

```
void CSD_ScanAllSensors();
```

汇编：

```
lcall CSD_ScanAllSensors
```

参数：

无

返回值：

无

副作用

**

CSD_UpdateSensorBaseline

说明：

单独针对每个传感器计算的历史计数值称为传感器的基准。 此基准使用 “ 水桶方法 ” 进行更新。

水桶方法使用下列算法：

1. 每次调用 CSD_UpdateSensorBaseline() 时，通过从原始计数值中减去以前的基准来计算差值计数。 此差值存储在 CSD_waSnsDiff[] 阵列中向您提供。

2. 如果禁用传感器自动复位，则每次调用 CSD_UpdateSensorBaseline() 时，差值计数与噪声阈值进行比较。 如果差值低于噪声阈值，将被累计到虚拟水桶中。 如果差值高于噪声阈值，则不更新水桶。 如果启用传感器自动复位，则无论噪声阈值参数如何，差值都将累计到虚拟水中。

3. 虚拟水桶中的累计差值计数达到 BaselineUpdateThreshold 后，基准按 1 递增，桶形电极复位为 0。

4. 如果差值计数低于噪声阈值，则保留在 waSnsDiff[] 阵列中的值复位为 0。 因此，此阵列不包含值大于 0 但低于噪声阈值的元素。

C 原型：

```
void CSD_UpdateSensorBaseline(BYTE bSensor)
```

汇编：

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

参数：

A => 传感器编号

返回值：

无

副作用：

**

CSD_UpdateAllBaselines

说明：

使用 CSD_bUpdateSensorBaseline() 函数更新所有传感器的基准。

C 原型：

```
void  CSD_UpdateAllBaselines()
```

汇编：

```
lcall CSD_UpdateAllBaselines
```

参数：

无

返回值：

无

副作用：

**

CSD_bIsSensorActive

说明：

与手指阈值进行比较，检查给定传感器的差值计数阵列。 将迟滞考虑在内。 根据传感器当前是否开启，对手指阈值加减迟滞值。 如果传感器处于活动状态，则降低该阈值。 如果传感器处于非活动状态，则提高该阈值。 此函数还更新传感器 CSD_baSnsOnMask[] 阵列中的位。

C 原型：

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

汇编：

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

参数：

bSensor A => 传感器编号

返回值：

如果传感器处于活动状态，则返回值为 1；如果传感器处于非活动状态，则返回值为 0。

A => 1 - 所选传感器处于活动状态，0 - 所选传感器处于非活动状态。

副作用：

**

CSD_bIsAnySensorActive

说明：

与手指阈值进行比较，检查所有传感器的差值计数阵列。 针对每个传感器调用 CSD_bIsSensorActive()，以便在调用此函数后 CSD_baSnsOnMask[] 阵列为最新。

C 原型：

```
BYTE  CSD_bIsAnySensorActive()
```

汇编：

```
lcall CSD_bIsAnySensorActive
```

参数：

无

返回值：

如果传感器处于活动状态，则返回值为 1；如果传感器处于非活动状态，则返回值为 0。

A => 1 - 一个或多个传感器处于活动状态，0 - 没有传感器处于活动状态。

副作用：

**

CSD_wGetCentroidPos

说明：

检查质心的差值阵列。 如果存在，则偏移和长度存储在临时变量中，根据 CSD 向导中指定的分辨率计算质心位置。 只有当滑条是由 CSD 向导定义时，此函数才可用。

C 原型：

```
WORD  CSD_wGetCentroidPos(BYTE bSnsGroup)
```

汇编：

```
mov  A,  bSnsGroup
lcall CSD_wGetCentroidPos
```

参数：

bSnsGroup A => 组编号

此参数可引用作为滑条的一组特定的传感器。 组 0 用于按钮。 滑条包含在组 1 和更高的组中。

返回值：

滑条的位置数值、A 中的 LSB 和 X 中的 MSB。

副作用：

此例程通过减去噪声阈值来修改差值计数。 此例程在每次扫描后只能调用一次，以避免得到负的差值。 如果应用程序监控差值计数信号，则在差值计数数据传输后调用此例程。

如果有滑条传感器处于活动状态，则该函数将值从零恢复为 CSD 向导中设置的分辨率值。 如果没有传感器处于活动状态，则该函数返回 -1 (FFFFh)。 如果在执行质心 / 双工算法时出现错误，则该函数返回 -1 (FFFFh)。 如果需要，可以使用 CSD_blsSensorActive() 例程确定触摸了哪些滑条段。

注意：如果滑条段的噪声计数大于噪声阈值，则此子程序可能生成假的质心结果。 设置噪声阈值时应小心 （显著大于噪声级别），以便噪声不会产生假的质心。

CSD_wGetRadialPos

说明：

检查质心的差值阵列。 如果存在，则根据 CSD 向导中指定的分辨率计算质心位置。 此函数仅用于 CSD 向导定义的辐射滑条。

C 原型：

```
WORD  CSD_wGetRadialPos(BYTE bSnsGroup)
```

汇编：

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

参数：

bSnsGroup A => 组编号

此参数是使用的辐射滑条的编号。 可以通过 CSD UM 向导从辐射滑条表示法的左侧获取其编号 （例如：对于 s2，辐射滑条编号为 2）。

返回值：

辐射滑条的位置数值、A 中的 LSB 和 X 中的 MSB。

副作用：

此例程在每次扫描后只能调用一次，以避免得到负的差值和基准更新。 如果应用程序监控差值计数信号，则在差值计数数据传输后调用此例程。

如果有滑条传感器处于活动状态，则该函数将值从零恢复为 CSD 向导中设置的分辨率值。 如果没有任何传感器处于活动状态，则该函数返回 -1 (FFFFh)。

注意：如果滑条段的噪声计数大于噪声阈值，则此子程序可能生成假的质心结果。 设置噪声阈值时应小心 （显著大于噪声级别），以便噪声不会产生假的质心。

CSD_wGetRadialInc

说明：

返回实际的手指移动情况、手指当前位置与先前位置的差值。 此函数与 CSD_wGetRadialPos() 成对使用，采用后者生成的数据 （数据保存在内部变量中）。

C 原型：

```
WORD  CSD_wGetRadialInc(BYTE bSnsGroup)
```

汇编：

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

参数：

bSnsGroup A => 组编号

此参数是使用的辐射滑条的编号。 可以通过 CSD UM 向导从辐射滑条表示法的左侧获取其编号 （例如：对于 s2，辐射滑条编号为 2）。

返回值：

手指移动数值 （顺时针为正，逆时针为负）、A 中的 LSB 和 X 中的 MSB。

手指移动数值是手指当前位置与先前位置之间的差值。 如果在先前的扫描期间未发生触摸 （倒数第二次 CSD_wGetRadialPos() 返回 -1 (FFFFh)）或者当前没有任何触摸（此时 CSD_wGetRadialPos() 返回 -1 (FFFFh)）

副作用：

只能在 CSD_wGetRadialPos() API 之后调用该例程。 因为它使用由 CSD_wGetRadialPos(). 设置的内部数据 CSD_waSliderPrevPos 和 CSD_waSliderCurrPos

CSD_InitializeSensorBaseline

说明：

通过扫描所选传感器，加载含有初始值的 CSD_waSnsBaseline[bSensor] 阵列元素。 原始计数值将复制到所选传感器的基准阵列元素中。 此函数可用于复位单个传感器的基准。

C 原型：

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

汇编：

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

参数：

A => 传感器编号

返回值：

无

副作用：

**

CSD_InitializeBaselines

说明：

通过扫描每个传感器，加载含有初始值的 CSD_waSnsBaseline[] 阵列。 原始计数值将复制到每个传感器的基准阵列中。

C 原型：

```
void CSD_InitializeBaselines()
```

汇编：

```
lcall CSD_InitializeBaselines
```

参数：

无

返回值：

无

副作用：

**

CSD_SetDefaultFingerThresholds

说明：

通过 FingerThreshold 参数值加载 CSD_baBtnFThreshold[] 阵列。 如果 CSD_baBtnFThreshold[] 阵列不是通过自定义值手动加载，则必须在扫描之前调用此函数。

C 原型：

```
void CSD_SetDefaultFingerThresholds()
```

汇编：

```
lcall CSD_SetDefaultFingerThresholds
```

参数：

无

返回值：

无

副作用：

**

CSD_SetScanMode

说明：

设置扫描速度和分辨率。 此函数可以在运行时调用来更改扫描速度和分辨率。 此函数覆盖用户模块参数设置。 当某些传感器需要用不同的扫描速度和分辨率进行扫描时 （例如：常用按钮和接近检测器），此函数非常有效。 可以用 9-bit 分辨率扫描常规按钮。 接近检测器经常可以采用比 16-bit 略低的分辨率进行扫描，对于大范围检测，扫描时间较长。 此函数可以与 CSD_ScanSensor() 函数一起使用。

C 原型：

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

汇编：

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

参数：

bSpeed

bResolution

返回值：

无

副作用：

**

CSD_SetSliderIdac

说明：

将滑条元素的 iDAC 电流设置为每个滑条组的最高值。

C 原型：

```
void CSD_SetSliderIdac(void)
```

汇编：

```
lcall  CSD_ SetSliderIdac
```

参数：

无

返回值：

无

副作用：

**

CSD_SetIdacValue

说明：

此函数覆盖用户模块参数设置 iDAC 值。 如果需要用其他 iDAC 设置扫描某些传感器，则使用此函数。 此函数可以与 CSD_ScanSensor(). 一起使用

C 原型：

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

汇编：

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

参数：

bIdacValue - 设置 iDAC 值。 接受的值为 1..255。

返回值：

无

副作用：

**

CSD_SetPrescaler

说明：

此函数覆盖用户模块参数设置预分频器值。 如果需要用预分频器设置扫描某些传感器，则使用此函数。 此函数可以与 CSD_ScanSensor(). 一起使用

C 原型：

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

汇编：

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

参数：

bPrescaler - 设置预分频器值。 下表列出了接受的值：

名称	值	预分频器
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

名称	值	预分频器
CSD_PRESCALER_32	0x05	32
CSD_PRESCALER_64	0x06	64
CSD_PRESCALER_128	0x07	128
CSD_PRESCALER_256	0x08	256

返回值：

无

副作用：

**

CSD_CalibrateSensors

说明：

调整 iDAC 电流以获取接近 wLevel 值的原始计数，并将结果存储在全局阵列 CSD_baDAC[]. 中

C 原型：

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

汇编：

```
lcall  CSD_CalibrateSensors
```

参数：

wlevel - 目标原始数据值。

返回值：

无

副作用：

**

CSD_ClearSensors

说明：

通过按顺序调用每个传感器的 CSD_wGetPortPin() 和 CSD_DisableSensor()，将所有传感器清除为非采样状态。

C 原型：

```
void  CSD_ClearSensors()
```

汇编：

```
lcall  CSD_ClearSensors
```

参数：

无

返回值：

无

副作用：

**

CSD_wReadSensor

说明：

在 A (LSB) 和 X (MSB) 中返回关键原始扫描值。

C 原型：

```
WORD  CSD_wReadSensor(BYTE bSensor)
```

汇编：

```
mov A, bSensor
lcall CSD_wReadSensor
```

参数：

A => 传感器编号

返回值：

传感器的扫描值、A 中的 LSB 和 X 中的 MSB

副作用：

**

CSD_wGetPortPin

说明：

返回给定传感器的端口号和引脚掩码。传递的参数对 CSD_Sensor_Table[]. 中的数据编制索引并进行选择。返回值可以传递给 CSD_EnableSensor()、CSD_DisableSensor().

C 原型：

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

汇编：

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

参数：

bSensorNumber - 范围为 0 到 (n - 1)，其中 n 是 CSD 向导中设置的传感器总数与滑条中包括的传感器数量之和。CSD_wGetPortPin() 使用传感器数量来确定所选活动传感器的端口和位掩码。

返回值：

A => 传感器位图

X => 端口号

副作用：

**

CSD_EnableSensor

说明：

配置所选传感器以便在下一测量周期中进行测量。 可以使用 CSD_wGetPortPin() 函数选择端口和传感器，端口号和传感器位掩码分别加载到 X 和 A 中。 修改驱动模式以便将所选端口和引脚置于模拟 High Z 模式并用正确的模拟复用器总线输入。 这还可以启用比较器功能。

C 原型：

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

汇编：

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

参数：

A => 传感器位图

X => 端口号

返回值：

无

副作用：

**

CSD_DisableSensor

说明：

禁用 CSD_wGetPortPin() 函数选择的传感器。 驱动模式更改为 “ 强 (001)” 并设置为零。 这可以将传感器有效接地。 端口引脚与 AnalogMuxBus 的连接关闭。 函数参数由 CSD_wGetPortPin() 函数返回。

C 原型：

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

汇编：

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

参数：

A => 传感器位图

X => 端口号

返回值：

无

副作用：

**

固件源代码示例

示例 1。 此代码启动用户模块并连续扫描传感器。 可以使用通信部分将值传递给 PC 绘图工具。

```
//-------------------------------------------------------------------------
// 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
 }
}
```

示例 2。下面的代码演示了如何能够并行连接多个传感器并通过调用 CSD_ScanSensor() 函数同时扫描它们。 当您需要在未区分已触摸哪些传感器的情况下检测传感器触摸时，此示例非常有用。 可以使用示例进行设备唤醒检测和最大程度地减少扫描时间以节省电池能量。 如果检测到唤醒触摸，则可以分别将每个传感器返回到常规扫描。

```
//-------------------------------------------------------------------------
// 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
                }
     }
}
```

示例 3。 下面的示例演示如何能够使用 CSD_SetScanMode() 函数，用不同的扫描参数扫描不同的传感器。当需要执行按钮触摸检测和接近检测时非常有用。 扫描按钮时采用低分辨率以减少扫描时间，扫描接近情况时采用较高分辨率以获取最大灵敏度。 您可以调整此代码，以便降低扫描接近情况的频率，只有当不进行按钮触摸检测时才扫描接近情况。

```
//-------------------------------------------------------------------------
// 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
```

```
            }
        }
}
```

示例 4。 下面的示例演示如何能够为每个传感器设置不同的手指阈值级别。 当多个传感器放置在不同位置上而其中一些传感器比其他更灵敏时，非常有用。

```
//-------------------------------------------------------------------------
// 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
  }
 }
}
```

配置寄存器

Table 5. 模块 CapSense、寄存器： CS_CR0

位	7	6	5	4	3	2	1	0
值	0	0	CSD_PRSCLK	0	1	0	0	EN

Table 6. 模块 CapSense、寄存器： CS_CR1

位	7	6	5	4	3	2	1	0
值	1	扫描速度		0	0	0	0	0

电源： 0x01 打开模拟模块的电源。 0x00 关闭模拟模块的电源。

Table 7. 模块 CapSense、寄存器： CS_CR2

位	7	6	5	4	3	2	1	0
值	1	0	0	0	0	1	0	0

Table 8. 模块 CapSense、寄存器： CS_CR3

模式 / 位	7	6	5	4	3	2	1	0
值	0	1	1	1	0	0	0	0

Table 9. 模块 CapSense、寄存器： CS_CNTH

位	7	6	5	4	3	2	1	0
数据输出 MSB								

Table 10. 模块 CapSense、寄存器： CS_CNTL

位	7	6	5	4	3	2	1	0
数据输出 LSB								

Table 11. 模块 CapSense、寄存器： PRS_CR

模式 / 位	7	6	5	4	3	2	1	0
值	1	0	8/12 位	1	预分频器			

Table 12. 模块定时器、寄存器： PT1_CFG

模式 / 位	7	6	5	4	3	2	1	0
值	0	0	0	0	0	0	1	启动

Table 13. 模块定时器、寄存器： PT1_DATA0

模式 / 位	7	6	5	4	3	2	1	0
值	数据 LSB							

Table 14. 模块定时器、寄存器： PT1_DATA1

模式 / 位	7	6	5	4	3	2	1	0
值	数据 MSB							

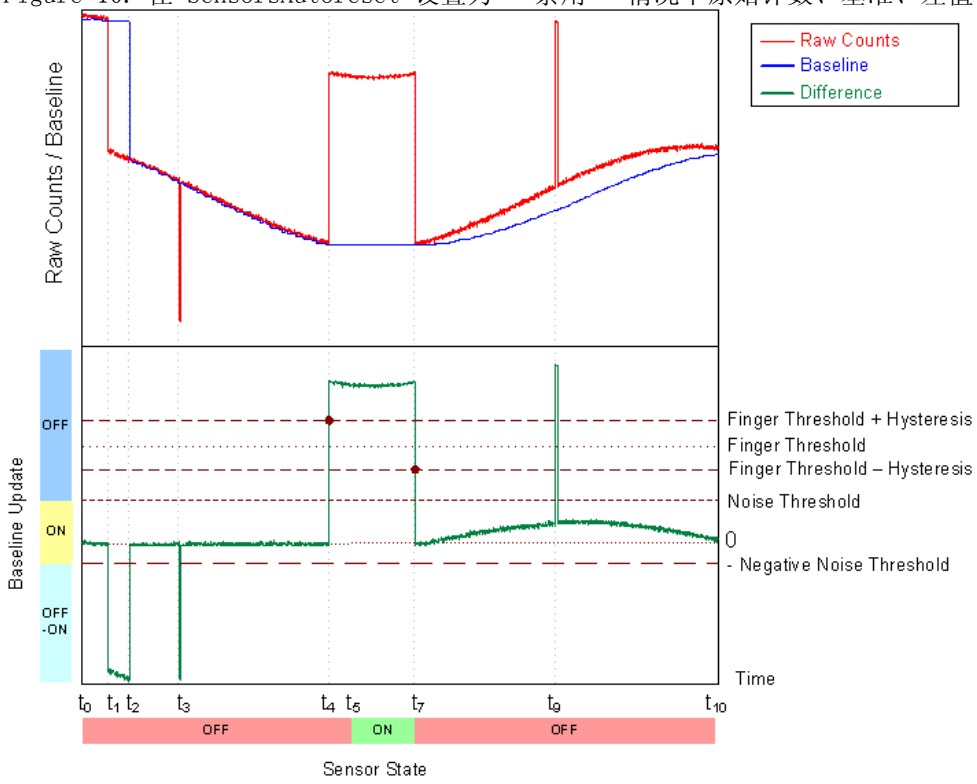
附录

下面各节将介绍用户模块数据表中通常包含内容之外的信息。 赛普拉斯工程师开发了详细信息来帮助您成功设计 CapSense 应用程序。 此信息的某些部分将来会移动到应用笔记中。

CSD 参数的交互

和图 10 说图 11 明了基准更新和决策逻辑操作，对于更好了解如何设置 UM 参数以获得最佳性能很有帮助。 说图 10 明了当传感器 Autoreset 参数设置为**禁用**时的系统操作。 说图 11 明了**启用**的 Autoreset 参数。 图中还一同显示了手指阈值、噪声阈值、迟滞和负噪声阈值与差值信号 （原始计数 - 基准）。 数据是在一些人工测试中收集的，这些测试展现了原始计数慢速和快速变化时的系统操作。慢速变化可能是温度或湿度变化所致，快速变化可能是由传感器触摸、ESD 事件或强射频场的影响触发的。

Figure 10. 在 SensorsAutoreset 设置为 “ 禁用 ” 情况下原始计数、基准、差值信号变化的示例



在 t_0 处，由于湿度或温度变化，接近于基准级别的原始计数开始缓慢下降。 由于两次连续转变之间的原始计数变化不超过 NegativeNoiseThreshold 参数 （绝对值），因此通过跟踪原始计数最小值来更新基准，保留原始计数信号的较小值。

在 t_1 处，原始记录快速下降，负差超过 NegativeNoiseThreshold。 如果手指位于传感器上时设备加电，过一段时间后手指移开，则会发生这种情况。 此时，基准更新机制冻结，激活内部超时计数器。 当差值信号低于 LowBaselineReset 示例的 NegativeNoiseThreshold 时，基准复位。 这是在 t_2 发生的。

第二大负差信号尖峰脉冲发生在 t_3 处；例如，可能已通过 ESD 事件触发此尖峰脉冲。 由于计数示例中的尖峰脉冲持续时间小于 LowBaselineReset 参数，因此保留基准，对尖峰脉冲进行滤波。 这可以阻止假基准复位和导致假触摸检测。

传感器是在 t_4 上被触摸的。 当差值信号超过 “ 手指阈值 + 迟滞 ” 值时，激活内部防反跳计数器。 如果信号超过此值的量大于防反跳示例，则传感器状态设置为启用。 这是在 t_5 发生的。 当差值信号在 t_7 下降到 “ 手指阈值 - 迟滞 ” 级别之下时，传感器立即恢复为关闭状态。 由于采样单元中的尖峰脉冲持续时间不超过防反跳值，t_9 处的瞬时正值尖峰脉冲由防反跳计数器筛选。

原始计数在 t_7 与 t_{10} 之间缓慢升高。 当差信号低于 NoiseThreshold（SensorsAutoreset 设置为 “ 禁用 ”），差信号与漂移速率成比例时，使用水桶算法来更新基准。 可以使用 BaselineUpdate 阈值参数来控制基准更新速度。 参数值越低，基准更新速度越快。

Figure 11. 在 SensorsAutoreset 设置为 “ 启用 ” 的情况下原始计数、基准、差信号的变化示例

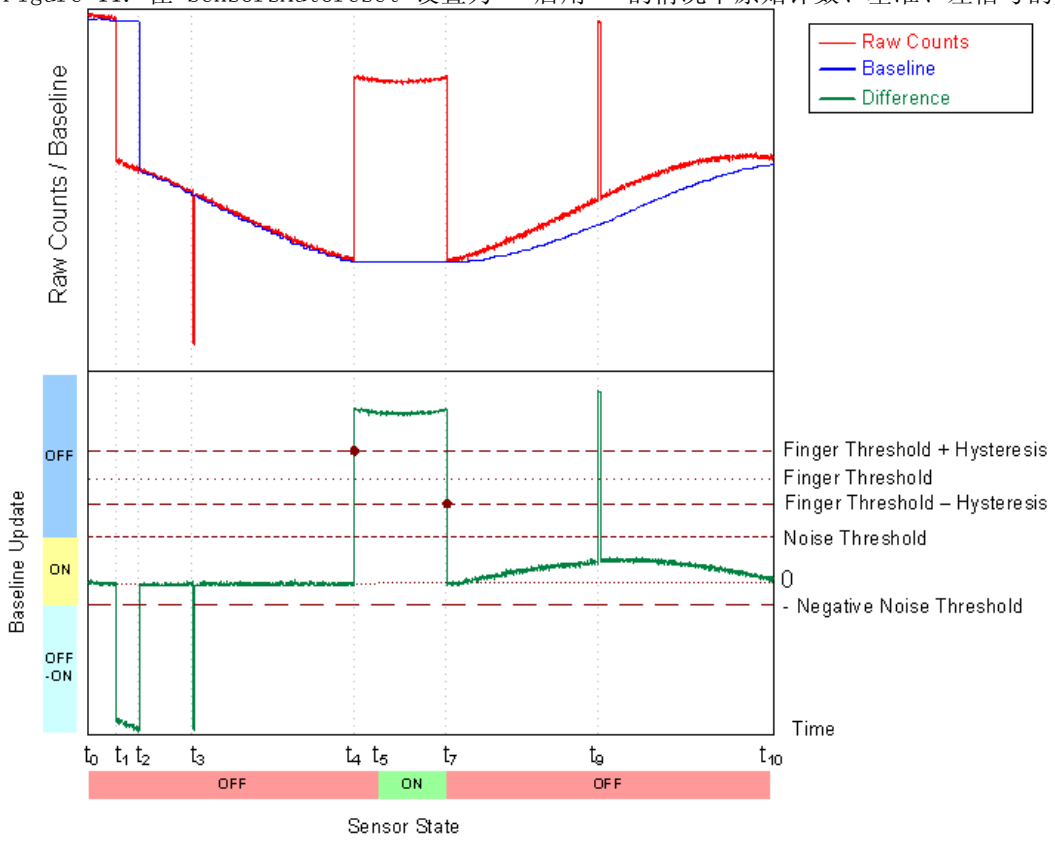


图 11 中的系统操作 类图 11 似于上例中的操作，但有下列不同之处：

- 在 t_6 处传感器被触摸，根据活动基准更新算法，触摸持续时间下降。
- 手指移开后，在 LowBaselineReset 样品（t_8）模块触摸检测很短时间后，基准复位。 这称为防反跳机制。

版本历史记录

版本	创作者	说明
1.1	DHA	1. 此版本具有添加附加滑条的功能。 2. 增加了 Autocalibration 参数。
1.20	DHA	为 CY8C20xx6 和 CY8CTMA3xx 设备系列增加了辐射状滑条功能。

Note PSoC Designer 5.1 在所有的用户模块数据表中提供版本历史记录。 本数据表详细介绍了当前和先前用户模块版本之间的区别。

Copyright © 2008-2014 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.