双 CapSense® Sigma-Delta 调制数据手册 CSD2x

Copyright © 2009-2012 Cypress Semiconductor Corporation. All Rights Reserved.

资源	PSoC® 模块				API 存储器 （字节）		引脚 （除传感器之外）
	CapSense®	I²C/SPI	定时器	比较器	闪存	RAM	
CY8C21x45, CY8C22x45							
单通道，带 IDAC*	1	–	–	1	1006	29	1
自动校准已启用	1	–	–	1	1198	35	1
双通道，带 IDAC*	2	–	–	2	1390	30	2
自动校准已启用	2	–	–	2	1576	36	2
单通道，带 Rb 电阻 *	1	–	–	1	1000	28	2
双通道，带 Rb 电阻 *	2	–	–	2	1400	30	4
每个额外的 CapSense 按钮	–	–	–	–	8	12	1
在使用包括五个电容式传感器的滑条的静态代码和 RAM 的 增加	–	–	–	–	613	90	5
每个额外的滑条元件	–	–	–	–	8	12	1
使用滑条复用设置时 （5 个传感器）的静态代码和 RAM 的增加量	–	–	–	–	10	–	–

* 根据 1 个电容按键的配置计算

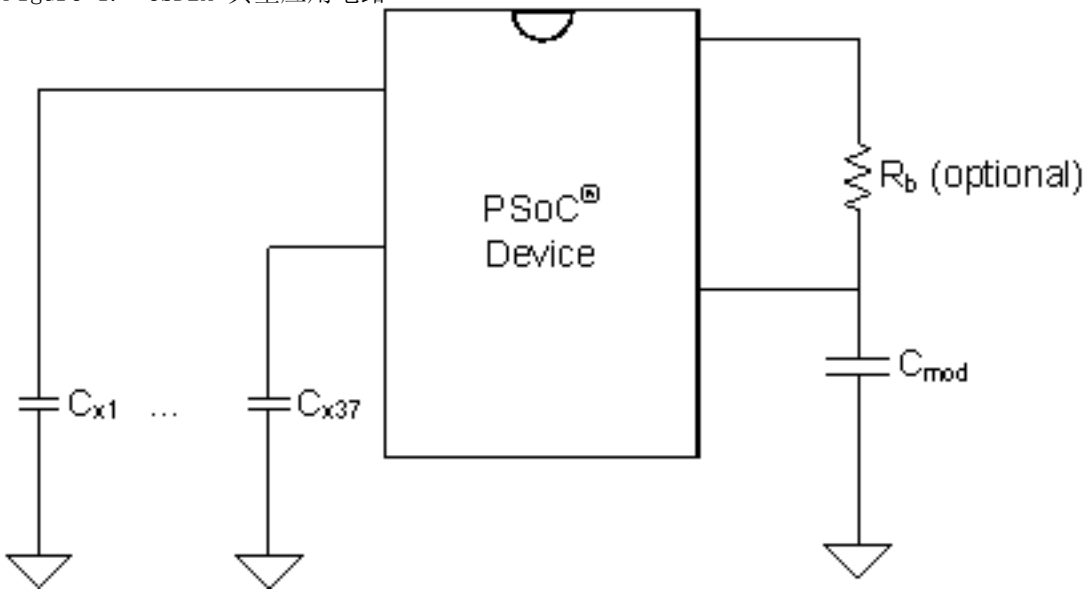
功能和概述

- 能够扫描 1 到 37 个电容传感器
- 可感应高达 15 mm 厚的玻璃覆层
- 使用导线式传感器时，接近检测可达 20 cm
- 对于交流电网噪声、电磁兼容性 （EMC）噪声以及供电电压变动的高抗干扰性
- 支持独立和滑条电容式传感器的不同组合
- 可使用双工法将滑条传感器的物理分辨率翻倍
- 可使用插值方法增加滑条传感器的分辨率
- 支持采用 2 个滑条的触摸板
- 通过高阻抗导电材料 （如 ITO 薄膜）的感应支持
- 具有屏蔽电极支持，可在存在水膜或水滴的情况下可靠工作
- 使用 CSD2x 向导进行指导配传感器和引脚的分配
- 集成了用于处理温度、湿度和静电放电 （ESD） 事件的基准线更新算法

- 具有易于调整的运行参数
- 具有 PC GUI 应用程序支持，用于实时进行原始数据监控和参数优化

CSD2x 用户模块 （使用 Sigma-Delta 调制器的电容式感应）使用开关电容技术实现电容感应，其中使用一个 Sigma-Delta 调制器将感应开关电容电流转换为数字代码。 CSD2x 用户模块可支持单通道 CapSense 扫描和双通道 CapSense 扫描。

Figure 1. CSD2X 典型应用电路



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. 选择和放置 CSD2x 用户模块。
3. 在工作区浏览器中右键单击 CSD2x 用户模块，以访问 CSD2x 向导 （稍后将在本数据手册中加以介绍）。
4. 设置所需的传感器、滑条或旋转滑条的数量。
5. 设置每个传感器的传感器设定。
6. 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。
7. 生成应用程序，并切换到应用程序编辑器。
8. 按照要求采用样本代码来实现独立传感器、滑条或触摸板。
9. 将 I2C-USB 桥接器连接到目标板，观察信号。
10. 更改 CSD2x 参数以优化您的设置并重建应用。
11. 为 PSoC 器件编程并验证模块操作。调整 CSD2x 参数以满足 5:1 的 SNR 要求，如 *CapSense 应用的信噪比要求* - 中所述 AN2403.

如果遇到任何问题，请参见*附录*中的*故障查找*部分。

功能描述

电容式传感器包括物理、电气和软件组件：

- **物理：** 物理传感器本身在典型情况下是一个连接到 PSoC 器件的在印刷电路板 （PCB）构建的导电图形，并配备了 1 个绝缘盖板、1 个柔性薄膜或 1 个覆盖在显示器上的透明覆层。。
- **电气：** 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。
  - 对于连续的相关传感器 （如滑条和触摸板），所提供的 API 以插值计算出一个优于传感器物理间距分辨率的位置。 例如，您可以创建一个包含 10 个传感器的音量滑条，并使用给定的固件将音量级别数扩展为 100。 另外，通过相同的 API，用户可以使用 2 个相互连接的电容传感器并计算出两者之间导电物体 （例如手指）的位置。

测量电容的方法有许多种，本用户模块中使用的方法是将开关电容与 Delta-Sigma 调制器相结合的方式。

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。 高级判断逻辑可为环境因素 （如温度、湿度和电源电压变动等）提供补偿。 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。 这些功能还可以在物理传感器位置之间对所求解的传感器位置进行进一步插值。

在首次使用 CSD2x 用户模块之前，建议您先阅读以下文档：

CY8C22X45 和 CY8C21345 PSoC 可编程片上系统技术参考手册，章节： CapSense 系统

建议在了解 CSD2x 用户模块文档之后进一步阅读以下应用笔记。 下列应用笔记参见赛普拉斯半导体公司网站 www.cypress.com：

- *CapSense Best Practices*（CapSense 最佳应用） - AN2394
- *Signal-to-Noise Ratio Requirements for CapSense Applications*（CapSense 应用的信噪比要求） - AN2403
- *Charting Tool to Debug CapSense Applications*（调试 CapSense 应用的图表工具） - 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*（通用异步发送的软件实现） - AN2399
- *Waterproof Capacitance Sensing*（防水电容传感） - AN2398

电容测量操作

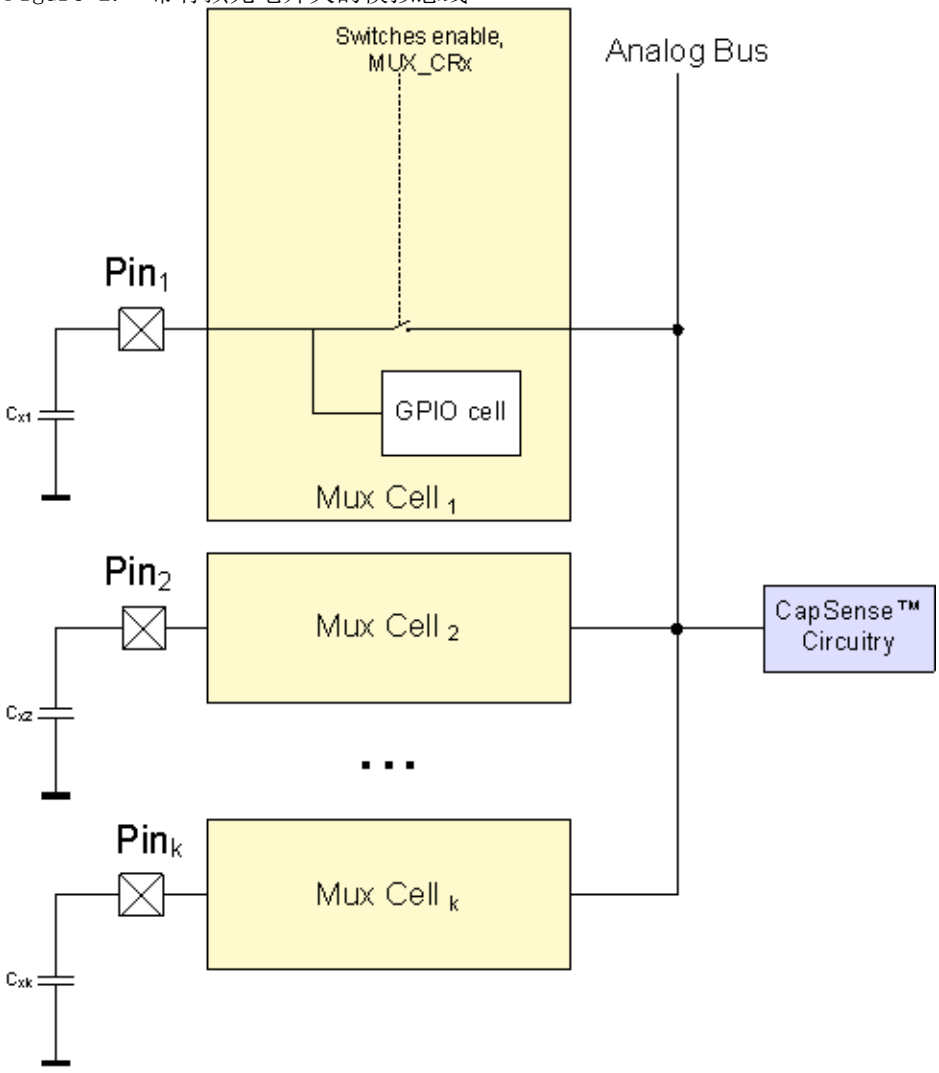
判断逻辑通过固件实现。 通过固件分析电容的测量，跟踪环境因素造成的缓慢电容变化，运行判断逻辑，以检测按键触摸变化并计算滑条位置。

扫描传感器阵列

CY8C22x45 系列器件具有内置模拟总线， 可以将电容式传感器连接到任意 PSoC 引脚。 CSD2x 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连续执行传感器扫描。

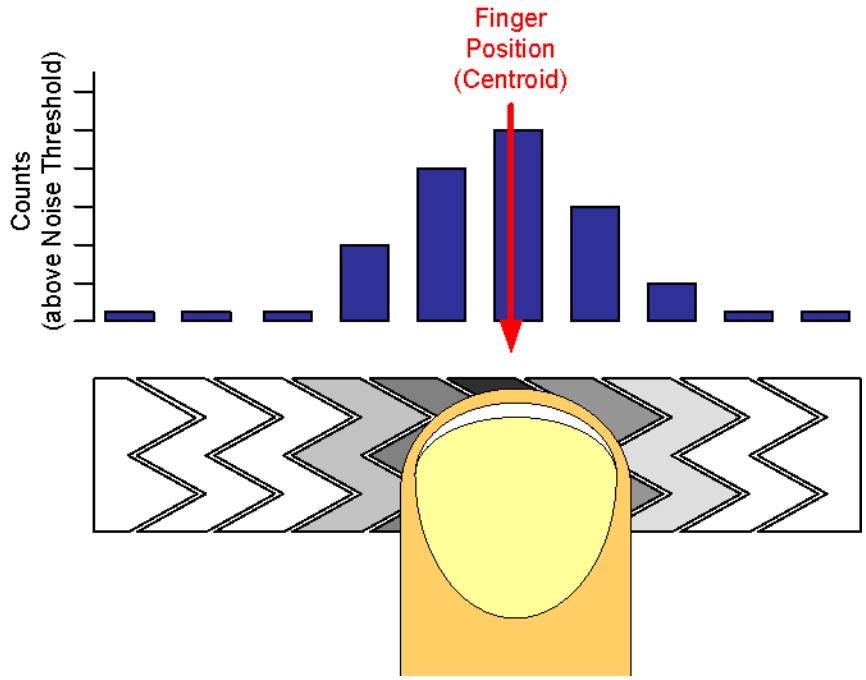
Figure 2. 带有预充电开关的模拟总线



滑条

滑条适用于需要渐进式调节的控制。 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。 通过计算活动传感器组的中心位置，可以确定滑条的实际位置。 滑条可在 CSD2x 向导中设置，即建立多个分组，每组滑条都有一个特定的顺序。 滑条传感器的实际下限值是五，上限值仅取决于所选 PSoC 器件提供的传感器的数值。

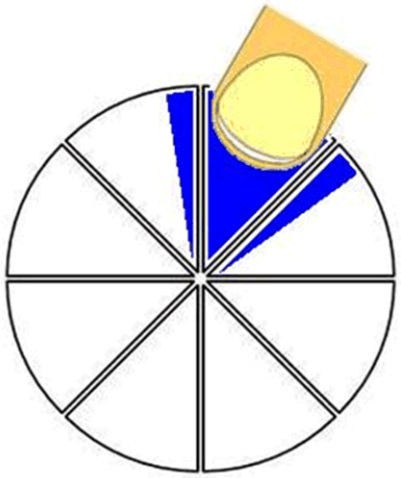
Figure 3. 对物理传感器位置排序



在滑条的一半内强烈信号的靠近会导致混叠到滑条上的最上部分传感器的电平相同，但每个传感器的电平值是离散的。 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。

辐射滑条

Figure 4. 手指触碰辐射状滑条



对于 CSD2X 用户模块，提供两种滑条类型：线性滑条和辐射滑条。 辐射状滑条与线性滑条类似。 线性滑条有起始点和结束点，辐射状滑条却没有。 发生碰触时，质心计算算法将考虑传感器开关切换到电流开关左右两侧的次数。 辐射状滑条未采用双工法。

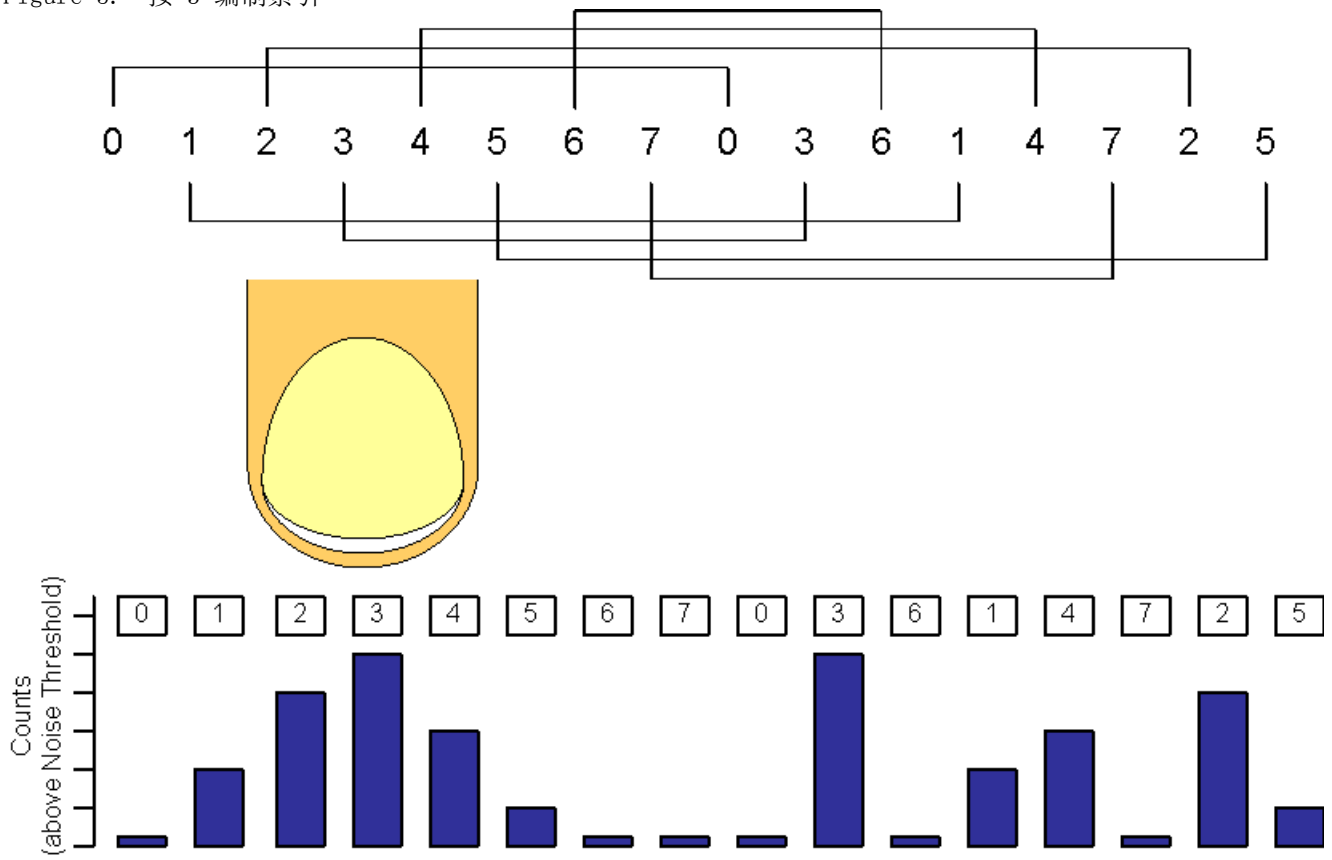
CSD2X 用户模块包含两个支持辐射滑条的 API 函数。 第一个函数 CSD2X_wGetRadiaPos() 返回质心位置，第二个函数 CSD2X_wGetRadialInc() 则返回以分辨率单位表示的手指移位。 当手指以顺时针方向移动时，会产生正的偏移。

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。 物理位置的前半 （数字上较低的）部分顺序映射到基本分配的传感器，并采用由设计者使用 CSD2x 向导分配的端口引脚。 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 包括的头文件中列出。 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。 请注意确定此次序，将其映射到印刷电路板上。

有多种方法可以确定物理传感器位置另一部分的次序。 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 5. 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 布局，某些传感器的走线可能会很长。 当您选择复用时，CSD2x 向导会自动生成复用传感器索引表。 下表说明了不同滑条段计数的双工序列。

Table 1. 不同滑条段计数的双工序列

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
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

滑条段总计数	段序列
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

插值和缩放

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。 手指接触滑条传感器或触摸板的面积通常大于任何单个的传感器。

要使用质心计算插值后的位置，首先要扫描阵列，以验证所给的传感器位置是否有效。 这要求一定数量的相邻传感器信号要大于于一个噪声阈值。 在找到最为强烈的信号后，此信号和那些大于噪声阈值的邻近信号均用于计算质心。 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

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

计算得出的值通常是分数。 为了能够采用某一特定分辨率来报告质心 （例如对于 12 个传感器为 0 到 100 的范围），要将质心值乘以计算得出的标量。 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，并在想要的标量中直接报告其结果 这是通过高级 API 实现的。

滑条传感器数量和分辨率在 CSD2x 向导中进行设置。 向导将计算出标量数值并以分数值的形式进行存储。

质心分辨率的乘数包含在三个字节中，且具有以下位定义：

分辨率乘数最高有效位								
位	7	6	5	4	3	2	1	0
乘数	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
分辨率乘数中等有效位								
乘数	128	64	32	18	16	8	4	2
分辨率乘数最低有效位								
乘数	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256

使用以下公式计算分辨率：

分辨率 = （传感器数 - 1） x 乘数

质心以 24-bit 无符号整数保存，其分辨率是传感器个数和乘数的函数。

反馈组件选择指南

CSD2x 用户模块需要外部调制电容，还支持可选屏蔽电极。 本节介绍如何选择外部组件。

调制电容

用户模块需要外部调制电容 C_{mod} 和调制器反馈电阻 R_b。 该电容可以连接到 P0[7] 端口引脚和 Vss 地。 反馈电阻 R_b 可以连接到端口引脚 P0[5] 和电容引脚。 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用途。

调制电容的建议值为 4.7 nF 到 47 nF。 可通过实验选择最佳电容值，以获得最大信噪比。 对于 PRS16 和 PRS8 配置，在大多数情况下，5.6 到 10 nF 值可以给出很好的结果。 如果选择了带有预分频器的配置，则积分电容的建议值为 22 到 47 nF。 选择反馈电阻后，可以试验几个电容值，以获得最佳的信噪比。 应使用陶瓷电容。 温度电容系数并不重要。 电阻值取决于总传感器电容 C_S。 应通过以下方法选择电阻值：

- 监控不同传感器触摸的原始计数。
- 选择一个电阻值，该值在选定扫描分辨率下提供的最大读数为全量程读数的 70%。 电阻值升高时，原始计数会增加。

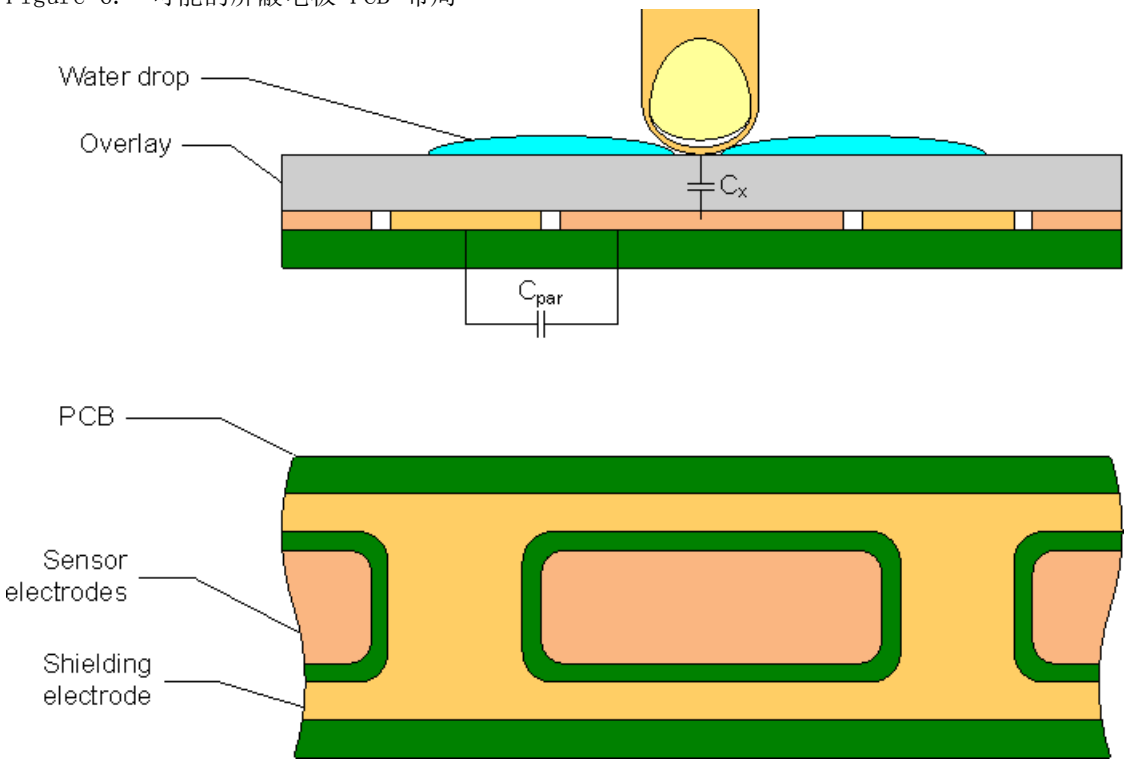
典型值为 500？到 10 k？，具体取决于传感器电容。 如果使用 CY3280-22x45 评估板，则最低值可以定为 2 k？。

屏蔽电极

有些应用场合要求在有水膜或水滴的情况下也能进行可靠的操作。 在白色家电、汽车应用场合、各种工业应用场合和其他场合，都需要即使有水、冰和湿度变化也不允许误触发的电容式传感器。 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。 如果器件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。 屏蔽电极有助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。

在某些应用场合，选择屏蔽电极信号及其相对于感测电极的位置十分有用，这样，增强两个电极之间的耦合就会造成感测电极电容测量的触摸变化减弱。 这样可以简化高级软件 API 的工作。 CSD2x 用户模块支持屏蔽电极的单独输出。

Figure 6. 可能的屏蔽电极 PCB 布局



上图展示了可能为按键屏蔽电极采取的一种布局配置。 屏蔽电极尤其适用于透明的 ITO 触摸板器件，在这种器件中，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。

在本例中，按键上覆盖有一层屏蔽电极面。 作为另一替代方法，屏蔽电极可以安装在 PCB 的另一面，其中包括按钮下面的平板。 对于这种情况，建议使用填充模式，填充率约为 30 ～ 40%。 这时无需额外的接地层。

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，因此水珠引发的原始计数增加量可能接近于零或出现较小的负值。 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它可以路由到的任何 PSoC 引脚。 将驱动模式设置为 “ 很慢 ” 可以降低接地噪声和辐射。 在 PSoC 器件与屏蔽电极之间，可以连接可选的斜率限流电阻。 对于行 LUT 函数，请选择 **A**。

比较器参考源

比较器参考源用于形成比较器参考电压。 该参考电压值决定了灵敏度。

对于 IDAC 和 Rb 配置，用户模块使用不同的参考形成原则。

对于 Rb 配置，用户模块支持参考源的下列多种选择：

- 带隙参考
- 模拟调制器，由特殊 PWM 信号驱动
- 外部电阻分压器

下表汇总了 Rb 配置的参考选择选项：

类型	外部组件	用户模块选择	使用场合
带隙参考	无	VBG	建议应用于大多数场合。 尝试从此选项开始测试。
模拟调制器	无	ASExx	读数与电源电压成比例。 仅当电源稳压良好时使用。
外部电阻分压器	2	模拟列 (AnalogColumn) 输入选择	读数与电源相关性较低。 建议 R1 = 10k; R2 = 3.6k

对于 IDAC 配置，用户模块支持下列参考源：

- 带隙参考
- V_{DD} 参考值
- 外部电阻分压器

此表汇总了 IDAC 配置时的参考选择选项：

类型	外部组件	用户模块选择	使用场合
带隙参考	无	VBG	建议应用于大多数场合。 尝试从此选项开始测试。
电源参考	无	VDD	读数与电源电压成比例。 仅当电源稳压良好时使用。

您可以仅使用带隙参考或模拟调制器。 外部电阻式分压 器适用于特殊场合。

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：

- 16-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，确保很好地抑制外部噪声源。 另外，带有扩频时钟的设计能够达到较低的电磁辐射级别。 如果您的应用程序目标是通过 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	工作频率	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。

放置

当实例化用户模块时，会自动放置用户模块，其他放置仅可用于单一通道配置。 CSD2X 用户模块使用 CapSense 模块和一个比较器模块。

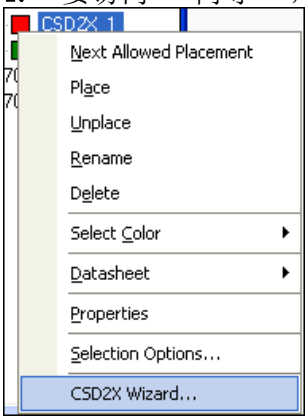
必须在用 CSD2X 向导建立用户模块的端口引脚连接之前，放置使用特定引脚资源 （包括 LCD 和 I2CHW）的用户模块。

在布置电容式传感器连接时，应避免使用 P1[0] 和 P1[1]。 这些引脚用来对部件进行编程，而且有可能存在过大的布线电容值，从而会影响传感器的灵敏度和噪声。

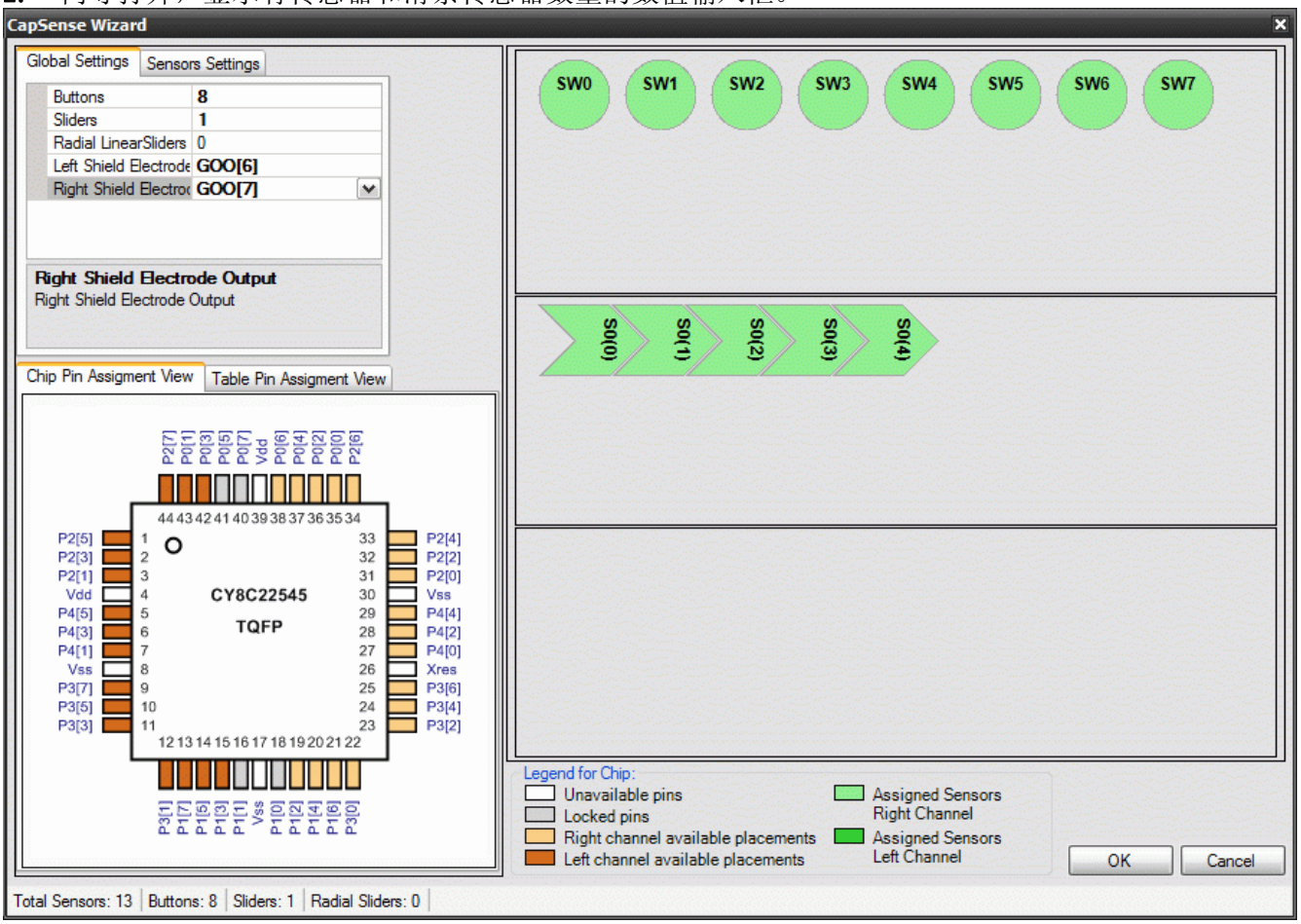
向导

CSD2x 向导用于设置 CapSense 按钮、滑条和接近传感器的引脚输出。 可以选择所需的配置，使用拖放界面分配按钮和区段。

1. 要访问 “ 向导 ”，请在工作区浏览器中右键单击用户模块，然后选择 “CSD2X 向导 ”。



2. 向导打开，显示有传感器和滑条传感器数量的数值输入框。



向导引脚图标

白色 – 该引脚不能用作 CapSense 输入。

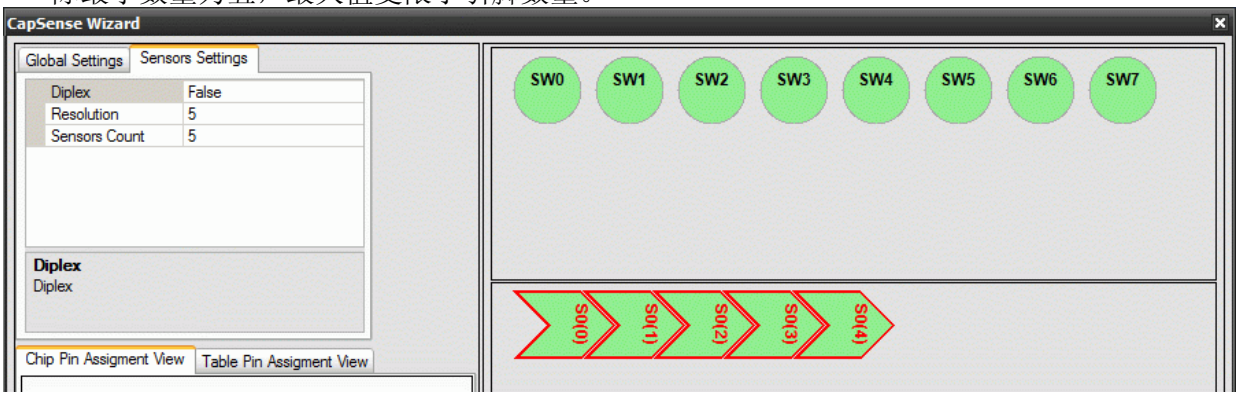
灰色 – 引脚处于锁定。 这种情况有两种可能的原因。 一种可能的原因是另一个用户模块（如 LCD 或 I^2C）已占用了该引脚。 第二种可能性是引脚已更改为使用非默认名称。要恢复使用引脚的默认名称，请在 “ 引脚分布 ” 视图中展开引脚，然后从**选择**菜单中选择**默认**。现在可以在向导中分配引脚了。

橙色 – 引脚可用于分配。

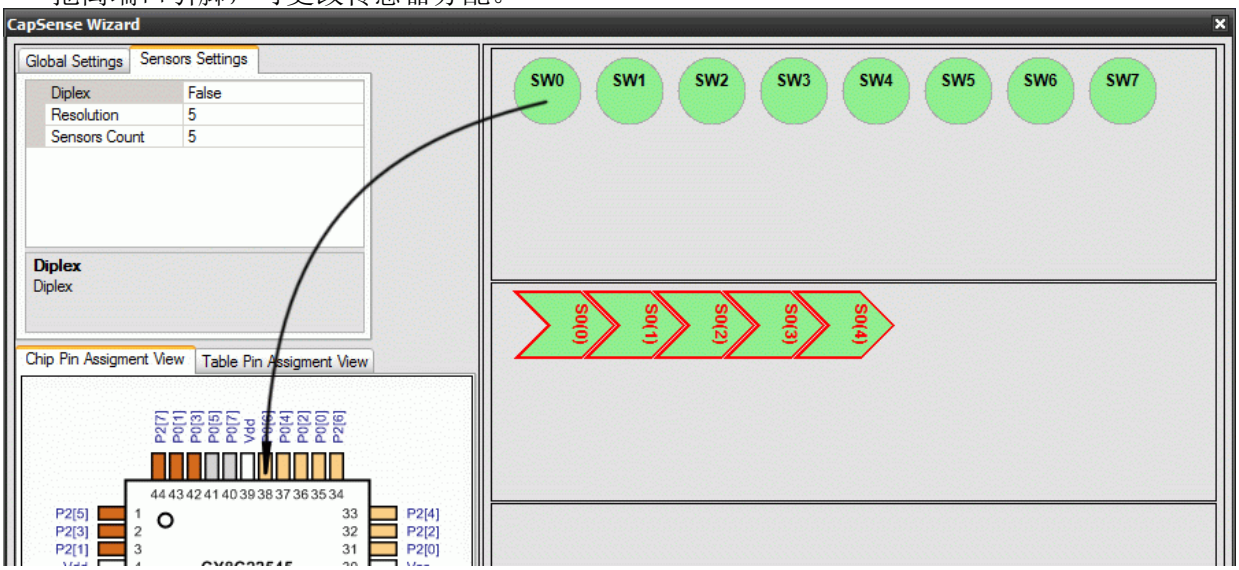
绿色 – 引脚已分配为 CapSense 输入。

3. 向导右侧区域包含三个区域，其中添加了开关、滑条和旋转滑条的表示。 在相应的框中键入所需的开关、滑条和旋转滑条数量，然后按 [Enter]。 显示内容随您选择的配置表示而更新。

4. 切换到 “ 传感器设置 ” 选项卡可设置滑条的参数。 滑条和旋转滑条的设置 （包括滑条或旋转滑条中的传感器数）是共同设置的。 X-Y 触摸板需要两个滑条，但是选择了一个。 滑条传感器中的传感器实际最小数量为五，最大值受限于引脚数量。

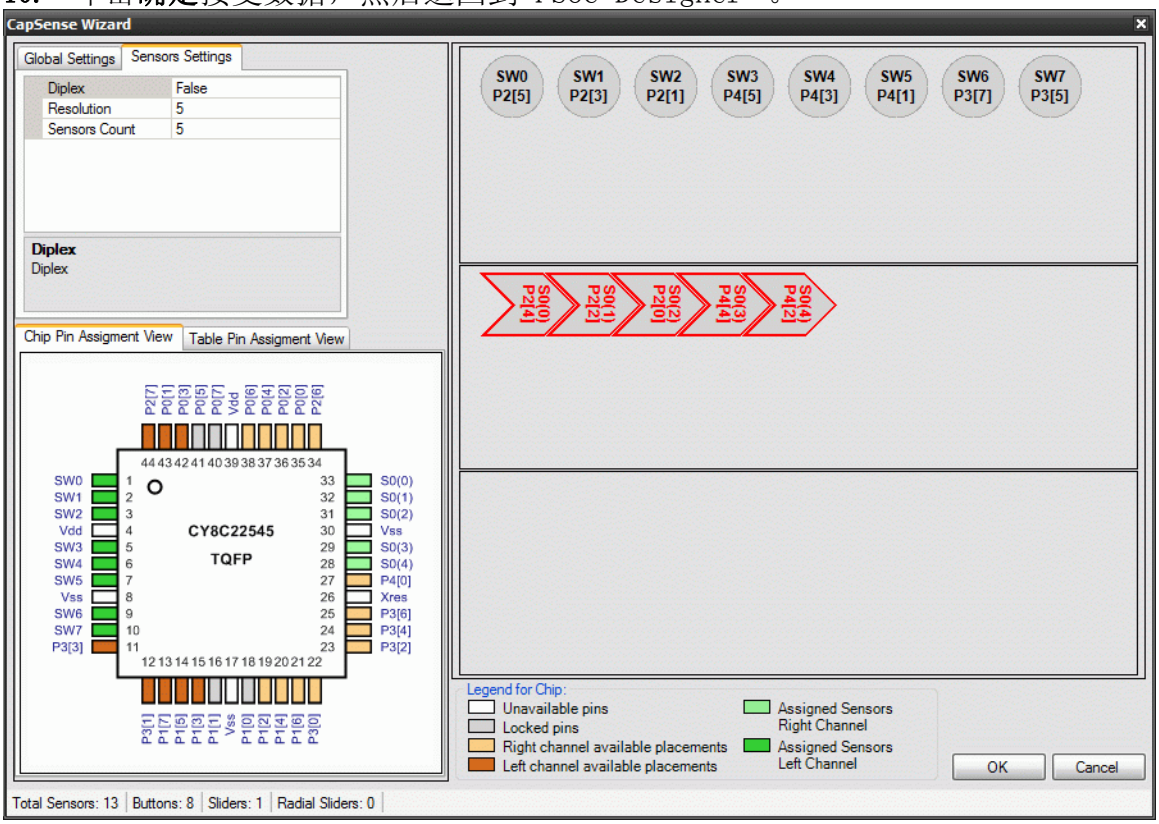


5. 滑条分辨率即是滑条具有的位置数。 在滑条的多个段上给出读数的触摸值会内插到此分辨率中。 如果将分辨率设置为 100，则手指在滑条上的位置报告为处于 0 与 99 之间的某个位置。 最小值为五。 对于双工型滑条，最大值为 （传感器的引脚数 - 1）x 2^8 - 1 或 （2 x 传感器的引脚数 - 1）x 2^8 - 1。
6. 如果需要，请选择双工。 这将把为传感器选定的引脚数值映射为板上传感器位置的两倍数值。 仅显示了双工传感器的前半部分；后半部分按前面 “ 双工 ” 章节所述自动映射。 有关引脚连接的双工表，请参见 “ 双工 ” 一节。
7. 要将开关映射到引脚，请单击开关，将其拖动到任意可用引脚。 可以使用 “ 芯片引脚分配 “ 视图或 “ 列表引脚分配 ” 视图来进行此操作。 选择端口引脚后，引脚变为灰色，不再可用。 通过将传感器拖离端口引脚，可更改传感器分配。



8. 对于其余独立传感器，重复操作即可。
9. 滑条段到端口引脚的映射与单个传感器相同。

10. 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。 设置用户模块参数，生成应用程序。 如果需要，可以立即对示例项目进行调整。

如果要更改引脚分配，请将光标放在已分配的引脚上，单击引脚，然后将其拖放到引脚分配框外。 该引脚分配取消，然后可以将其重新分配。

向导滑条设置

双工

仅适用于滑条。 可以使用一个引脚监控两个电子传感器以提高分辨率。 有关更多信息，请参阅双工章节。

分辨率

对于滑条和旋转滑条，该值设置 CSD2x_wGetCentroidPos API 所返回值的范围。 如果有滑条传感器处于活动状态，则该函数返回值是从零到 CSD 向导中设置的分辨率值。 CapSense 算法根据相邻传感器的读数，将质心位置内插到此分辨率。

传感器数量

该值设置每个滑条或旋转滑条中的传感器数。

向导生成的表

完成向导后，单击 “ 生成应用程序 ”。 根据传感器计数、引脚分配和双工的输入，将生成一组表。 这些表位于 CSD2X_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）。 有两个表，分别用于左右通道。 表格中列出了所有的独立传感器，然后是按顺序排列的各个传感器。 下面是一个包含六个传感器的表的示例：

```
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
```

该表由 CSD2X_wGetPortPin() 例程使用。

分组表

分组表定义了每个按键传感器组或滑条组。 每个滑条对应一个条目，自由按键传感器再对应一个条目。 第一个条目始终对应自由传感器。 每个条目为六个字节。 第一个字节表明传感器表中组开始的索引。 第二个字节是该组中的传感器总数。 第三个字节表示是否对滑条采用了双工法 （4 表示已采用双工法，0 表示未采用）。 第四、五、六个字节是固定点乘数，滑条计算出的质心乘以这些乘数就可以得到 CSD2X 向导中所需的分辨率。

```
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
```

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。 下面是八传感器滑条的典型示例。

```
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
```

顺序表

顺序表针对每个传感器包含一个字节。 该字节是不依赖于通道的原始计数结果阵列中的左传感器顺序。 该表由向导生成，反映了传感器顺序。

```
 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

启用 “ 自动校准 ” 时，会为具有 ICAD 配置的 CSD2X 生成 IDAC 表。 对于可在运行时更改的每个传感器，该表包含了已校准的电流值。 这在复杂项目中很有帮助。

对于双通道配置：

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

对于单通道配置：

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

软件中断兼容性

警告： CSD2x 用户模块在 INT_CLR2 寄存器运行期间用逻辑 “1” 值覆盖该寄存器的内容。 这意味着软件中断功能不能用于 CSD2x 用户模块。 不要更改 INT_MSK3 寄存器中的 ENSWINT 位字段的值 – 这会中断项目运行。

参数和资源

Autocalibration

启用或禁用自动校准 API 函数。

“ 自动校准 ”(Autocalibration) 仅包括在 IDAC 配置中。 “ 自动校准 ”(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，但是它允许开始调试过程时快速获取可读范围中的原始计数。 “ 自动校准 ”(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。 “ 自动校准 ”(Autocalibration) 用于部分提高功能配置。 如果启用此参数，在用户模块启动期间 CSD2X_Start API 结束时，会自动调用 CSD2X_Calibrate API 函数。

噪声阈值

如果差值计数值高于此阈值，则不更新基准线。 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。

屏蔽电极输出

此参数启用或禁用支持可选屏蔽电极的算法。 在 CSD2x 向导中，屏蔽电极 （如果启用）路由到左侧通道的 P1[6] 或 P3[6] 或右侧通道的 P1[7] 或 P3[7]。 在具有较少引脚的器件上，P3[6] 和 P3[7] 不可用。 对于单一通道，连接取决于使用的是右侧还是左侧 CapSense 通道。

选定引脚的驱动模式应当设置为**强**。

基准线更新阈值 (BaselineUpdate Threshold)

如果新的原始计数值高于当前基准线，差值低于噪声阈值 （“ 传感器自动复位 ”(Sensors Autoreset) 参数设置为 “ 禁用 ”），则当前基准线与原始计数的差值累计到水桶中。 当水桶充满时，基准线按某个值递增，并清空水桶。 此参数用于设置为了使基准线增加时 ″ 水桶 ″ 所必须达到的阈值。 可能值为 0 到 255。 参数值越大，基准线更新速度越慢。 如果需要进行更加频繁的基准线更新，请减小此参数。

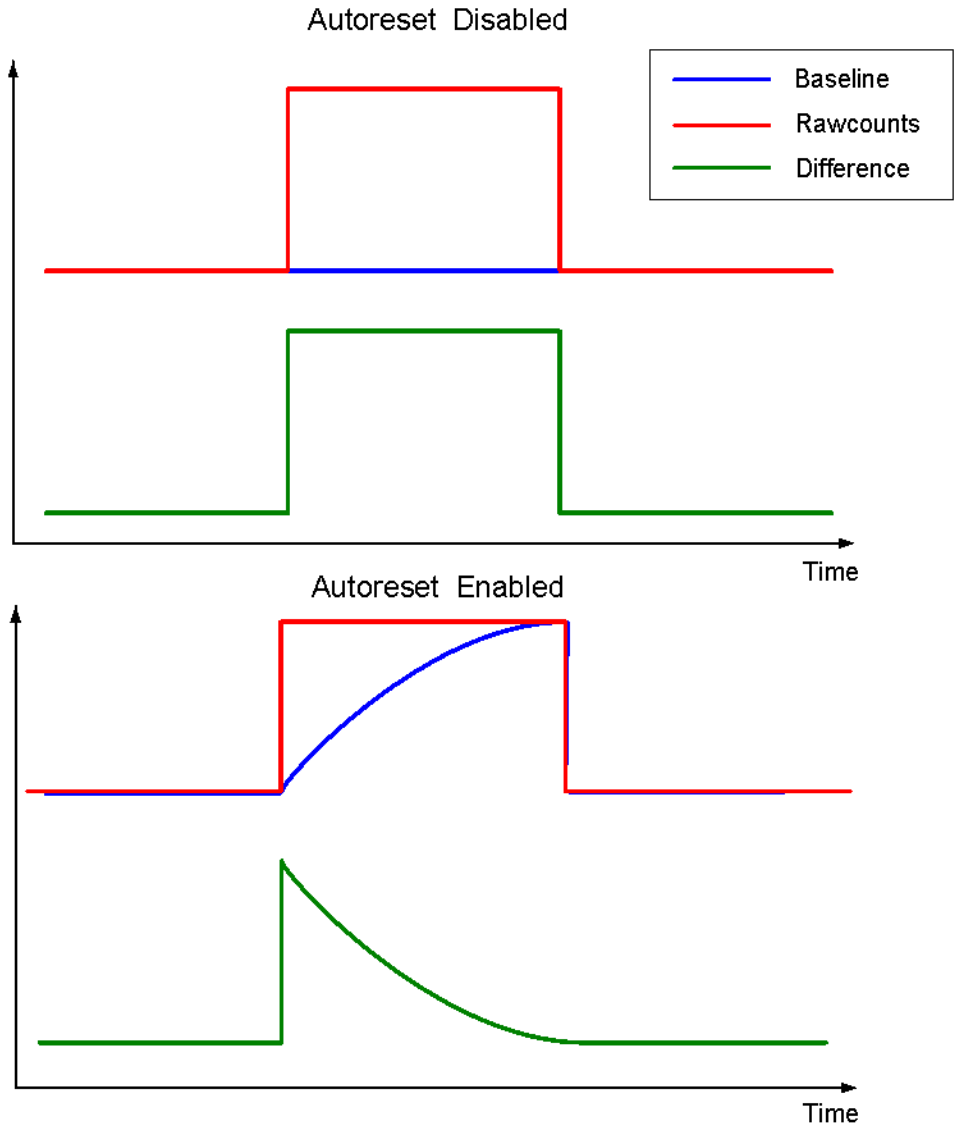
传感器自动复位

此参数确定基准线是否随时更新，还是仅当信号差值低于噪声阈值时更新。 当设置为**启用**时，基准线随时更新。 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触摸有效。 原始计数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线的差低于噪声阈值参数时基准线才进行更新。 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 ”。

下图说明了此参数对基准线更新的影响。

Figure 7. 传感器自动复位 (Sensor Autoreset) 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平稳性和牢固性。 当调用 bIsSensorActive() 或 bIsAnySensorActive() 时，带有迟滞的阈值参与计算 。 可以用 bIsSensorActive() 或 baSnsOnMask[] 数组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 ”(Finger Threshold) 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，对于指定的样品数量，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。

手指阈值

此阈值用于确定每个按键传感器的状态。 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 （不包含在滑条组中），这些阈值是变量，在 baBtnFThreshold[] 数组中提供。 可以使用 CSD2X_SetDefaultFingerThresholds() 函数将阈值设置为用户模块参数中设置的默认值。 要调整单个传感器的灵敏度，请更改每个传感器的 baBtnFThreshold[] 值。 （此字节数组的大小等于部署的各个传感器的数量。）

可能值的范围为从 1 到 255。

扫描速度 (Scanning Speed)

此参数影响传感器的扫描速度。 可选择的速度包括： **超快、快速、正常、慢速**。 较慢的扫描速度具有下列好处：

- 信噪比提高
- 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 (Scanning Speed)	除数
超快	1
快速	2
正常	4
慢速	8

扫描分辨率

此参数确定扫描分辨率 （以 “ 位 ” 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 2. **对于单通道配置：** 对于 24 MHz CPU 频率操作、带有预分频器的情况，扫描时间 （以 µs 为单位）与扫描速度和分辨率的关系 = SYSCLK/1

分辨率 （以位为单位）	扫描速度 (Scanning Speed)			
	超快	快速	正常	慢速
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

分辨率 （以位为单位）	扫描速度 (Scanning Speed)			
	超快	快速	正常	慢速
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. **双通道配置：** 对于 24 MHz CPU 频率操作、带有预分频器的情况，扫描时间 （以 µs 为单位）与扫描速度和分辨率的关系 = SYSCLK/1

分辨率 （以位为单位）	扫描速度 (Scanning Speed)			
	超快	快速	正常	慢速
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

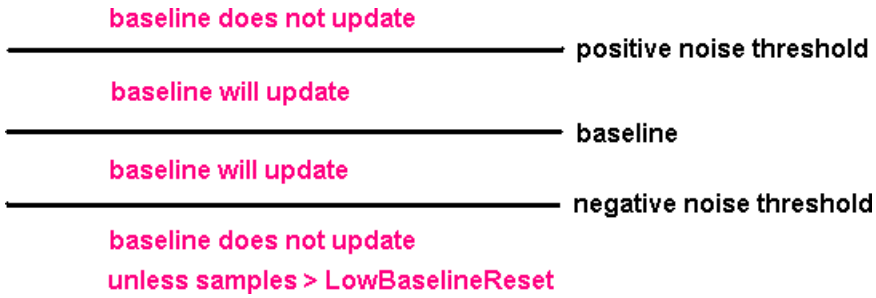
1. 扫描时间是按两次传感器扫描的时间间隔测量的。 此时间包括传感器设置时间、调制器稳定延迟、采样转换间隔和数据预处理时间。
2. 对于给定分辨率和扫描速度设置的情况，扫描时间的增加正比于预分频器的值的增加。 例如，将预分频器从 SYSCLK/1 更改为 SYSCLK/2，将使用 2 作为因子，增加此表中的扫描时间。

低基准线复位 (LowBaselineReset)

“ 低基准线复位 ”(LowBaselineReset) 参数与 “ 负噪声阈值 ”(NegativeNoiseThreshold) 参数协同工作。 如果采样到的计数值低于基准线与 “ 负噪声阈值 ”(NegativeNoiseThreshold) 之差，并且低于它的次数达到指定的采样次数，基准线会将设置为新的原始计数值。 此参数实际上是对重设基准线所需的异常低的采样数值的次数进行计数。 它通常用来纠正启动时手指已经在传感器上面的情况。 可能值的范围为 0-255。

负噪声阈值 (NegativeNoiseThreshold)

“ 负噪声阈值 ”(NegativeNoiseThreshold) 参数增加负的差值计数阈值。 如果当前的原始计数低于基准线且二者之差大于此阈值，则不更新基准线。 但是，如果对于当前原始计数处于较低状态 （差大于阈值）的次数等于低基准线复位 (LowBaselineReset) 参数所指定的采样次数，则对基准线进行复位。 可能值的范围为 0-255。



调制器电容引脚 (Modulator Capacitor Pin)

调制器电容引脚仅用于单一通道配置。 此参数设置引脚以连接外部调制器电容 (C_{mod})。 可用引脚为 P0[5] 和 P0[7]。对于双通道配置，同时使用两个引脚。

参考源

此参数设置比较器参考源。 可能的选择有带隙 (Vbg)、模拟调制器 (ASExx) 或外部电压 （来自 AnalogColumn_InputSelect_x). 对于 IDAC 配置，可能的选择有 Vbg 或 Vdd。 如需其他信息，请参阅 “ 比较器参考源 ” 一节。

参考值

对于 IDAC 和 Rb 配置，值 0 对应于参考电压接近 GND。 值 31 对应的参考电压接近 V。这提供参考电压一个 5-bit 分辨率。

在 Rb 配置中，如果参考来自外部引脚或 Vbg，则参考值参数没有任何效果。 在这种情况下，可以忽略参考值设置。 另外，在 Rb 配置中，参考电压的实际分辨率为 4-bits。 但是，所用的值仍然处于 5-bit 范围 （即，0 到 31）。

使用的参考值越高，提供的参考电压就越高。 较高的参考电压会导致传感器原始计数上升。 同样，较低的参考值 / 电压会导致传感器原始计数下降。

预充源

此参数选择预充电开关的时钟源。 允许的选项为 PRS 和定时器。 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。

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。 对于给定分辨率和扫描速度设置的情况，此参数也影响 PRS 输出频率和传感器扫描时间。

可能值有：

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

反馈电阻引脚 (Feedback Resistor Pin)

此参数仅可用于 Rb 配置。 此参数设置引脚以连接外部反馈电阻 (R_b)。 使用 Rb 配置的双通道在左右通道拥有相应的**左反馈电阻引脚**和**右反馈电阻引脚**属性。 从以下可用引脚中选择： P1[4]、P3[4]、GOO[4] 用于左通道，P1[5]、P3[5]、GOO[5] 用于右通道。 在某些器件封装上，某些引脚不可用。 提示：如果这些引脚中的一些引脚用于其他用途 （例如，分配用于传感器连接），它们在用户模块参数列表中不可选择。 CSD 用户模块的将来版本可允许使用附加引脚来连接反馈电阻。 此将允许在没有 P3 端口的封装中使用另一个 I^2C 端口。 使用引脚 P1[5] 或 P3[1] 是为了避免编程问题。

IDAC 范围

此参数仅可在 iDAC 配置中可用。 设置 iDAC 当前乘数。 该设置的结果不同于双通道配置。 下面显示了双通道配置结果：

在双通道配置中，将具有大电容的传感器连接到左通道。

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，以获取大约为整个范围的 50-70% 的原始计数。 可以在运行时使用 CSD2X_SetLeftDACValue 或 CSD2X_SetRightDACValue API 函数更改此参数。 可能值为 1 到 255。

补偿 IDAC 值

补偿 IDAC 旨在补偿传感器的初始电容。 虽然调整此参数会增大灵敏度和信噪比，但是设置太高的值会中断 CSD2X 操作。 在调整过程中，调整起点为 0，增加该值可达到最大信噪比和灵敏度。 此参数仅可用在单通道 IDAC 配置中。

可能值为 0 到 255。

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，使您能够以更高级别处理模块。 本节指定每个函数的接口，以及内置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 CSD2X_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。 为简便起见，在以下说明中将实例名称缩写为 CSD2X。

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。 选择这种 “ 易失性寄存器 ” 策略是为了提高效率，并且从 PSoC Designer 的 1.0 版本起已开始使用。 C 编译器自动遵循此要求。 汇编语言编程人员也必须确保其代码遵守该策略。 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。

对于大型储存器模块器件， 保存 CUR_PP、IDX_PP、MVR_PP 以及 MVW_PP 寄存器中的所有值也是调用程序的职责。 尽管部分寄存器现在可能不可修改，但是无法保证在将来的版本中也会如此。

提供了初始化 CSD2X、启动其采样和停止 CSD2X 的进入点。 在所有情况下，模块的实例名称会替换下列进入点中显示的 CSD2X 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数组。 不应手动更改这些数组。 不过，您可以出于调试目的对这些值进行检查。 例如，可以使用绘图工具显示数组中的内容。 以下是几个全局数组：

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

CSD2X_waSnsBaseline[] – 这是一个整数数组，其中包含每个传感器的基准数据。 数组大小与传感器数量相等。 CSD2X_waSnsBaseline[] 数组通过下列函数更新：

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

CSD2X_waSnsResult[] – 这是一个整数数组，其中包含每个传感器的原始数据。 数组大小与传感器数量相等。 CSD2X_waSnsResult[] 数据通过下列函数更新：

- CSD2X_ScanSensor();
- CSD2X_ScanAllSensors();

CSD2X_waSnsDiff [] – 这是一个整数数组，其中包含原始数据与每个传感器基准数据之间的差值。 数组大小与传感器数量相等。 该数据通过以下函数更新： CSD_UpdateSensorBaseline().

CSD2X_baSnsOnMask[] – 这是一个保持传感器开或关状态的字节数组 （对于按键或滑条）。 CSD2X_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）。 CSD2X_baSnsOnMask[1] 包含传感器 8 到 15 的掩码位 （如果需要），依此类推。 此字节数组包含的元素数足以包含所有放置的传感器。 按键开启时位值为 1，关闭时位值为 0。 CSD2X_baSnsOnMask[] 数据由 CSD2X_blsSensorActive(BYTE bSensor) 函数或 CSD2X_bIsAnySensorActive() 子程序更新。

CSD2X_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。 启用 “ 自动校准 ” 时，在应用所有其他用户模块参数之后，会在此函数中自动调用 CSD2X_Calibrate API。

C 原型：

```
void  CSD2X_Start()
```

汇编：

```
lcall  CSD2X_Start
```

参数：

无

返回值：

无

副作用：

**

CSD2X_Stop

说明：

停止传感器扫描，禁用内部中断，调用 CSD2X_ClearSensors() 以将所有传感器复位为非活动状态。

C 原型：

```
void  CSD2X_Stop()
```

汇编：

```
lcall  CSD2X_Stop
```

参数：

无

返回值：

无

副作用：

**

CSD2X_SetScanMode

说明：

此函数会覆盖所有后续扫描的用户模块参数中设置的扫描速度和扫描分辨率。 分辨率为 9 与 16 之间的整数值。

C 原型：

在单通道配置中：

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

在双通道配置中：

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

汇编：

在单通道配置中：

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

在双通道配置中：

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

参数：

A => 扫描速度

X => 扫描分辨率。 9 与 16 之间的整数值。

下面给出了 bSpeed 参数的常量：

常量	值
CSD2X_ULTRAFAST_SPEED	0x00
CSD2X_FAST_SPEED	0x01
CSD2X_NORMAL_SPEED	0x02
CSD2X_SLOW_SPEED	0x03

返回值：

无

副作用

**

CSD2X_SetPrescaler

说明：

此函数会覆盖所有后续扫描的用户模块参数中设置的预分频器值。

C 原型：

在单通道配置中：

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

在双通道配置中：

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

汇编：

在单通道配置中：

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

在双通道配置中：

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

参数：

A => 预分频器值。

下面给出了 bPrescaler 参数的常量：

常量	值
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

返回值：

无

副作用

**

CSD2X_ScanSensor

说明：

扫描选定的传感器。 每个传感器在传感器阵列中有唯一编号。 对于单通道配置，此编号由 CSD2X 向导按顺序分配。 Sw0 为传感器 0，Sw1 为传感器 1，依此类推。

对于双通道配置，传感器编号为 0 到最大通道传感器编号之间的一个值。 例如，如果左通道有两个传感器，则它们的值分别为 0 和 1。 如果右通道也有两个传感器，则它们的值也分别为 0 和 1。 如果 0xFF 值作为传感器编号传入此函数，则不扫描该通道的传感器。

C 原型：

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

在双通道配置中：

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

汇编：

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

在双通道配置中：

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

参数：

A => 传感器编号

在双通道配置中：

A => 左通道传感器编号

X => 右通道传感器编号

返回值：

无

副作用

**

CSD2X_ScanAllSensors

说明：

通过调用每个传感器索引的 CSD2X_ScanSensor()，扫描所有已配置的传感器。

C 原型：

```
void  CSD2X_ScanAllSensors();
```

汇编：

```
lcall  CSD2X_ScanAllSensors
```

参数：

无

返回值：

无

副作用

**

CSD2X_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线。 此基准线使用 “ 水桶方法 ” 进行更新。

“ 水桶方法 ” 使用以下算法。

1. 每次调用 CSD2X_UpdateSensorBaseline() 时，通过从原始计数值中减去以前的基准线来计算差值计数。 此差存储在 CSD2X_waSnsDiff[] 阵列中并提供给您。
2. 如果禁用传感器自动复位，则每次调用 CSD2X_UpdateSensorBaseline() 时，差值会与噪声阈值进行比较。 如果差值低于噪声阈值，将被累加到虚拟水桶中。 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，差值都将累加到虚拟水桶中。
3. 虚拟水桶中的累计差值计数达到 BaselineUpdateThreshold 后，基准按 1 递增，水桶复位为 0。
4. 如果差值低于噪声阈值，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  CSD2X_UpdateSensorBaseline(BYTE bSensor)
```

汇编：

```
mov  A,  bSensor
```

```
lcall  CSD2X_UpdateSensorBaseline
```

参数：

A => 传感器编号

返回值：

无

副作用：

**

CSD2X_UpdateAllBaselines

说明：

使用 CSD2X_bUpdateSensorBaseline() 函数更新所有传感器的基准线。

C 原型：

```
void  CSD2X_UpdateAllBaselines()
```

汇编：

```
lcall  CSD2X_UpdateAllBaselines
```

参数：

无

返回值：

无

副作用：

**

CSD2X_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。 此函数还可更新 CSD2X_baSnsOnMask[] 阵列中传感器的位。

C 原型：

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

汇编：

```
mov  A,  bSensor
lcall  CSD2X_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：

**

CSD2X_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 CSD2X_bIsSensorActive()，以便在调用此函数后 CSD2X_baSnsOnMask[] 阵列为最新。

C 原型：

```
BYTE  CSD2X_bIsAnySensorActive()
```

汇编：

```
lcall  CSD2X_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：

**

CSD2X_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，并根据 CSD2X 向导中指定的分辨率计算质心位置。 只有当滑条是由 CSD2X 向导定义时，此函数才可用。

C 原型：

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

汇编：

```
mov  A,  bSnsGroup
lcall  CSD2X_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 用于按键。 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。

如果有任何滑条传感器处于活动状态，则该函数返回从零到 CSD2X 向导中设置的分辨率值之间的值。 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 CSD2X_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，此例程可能会生成错误的质心结果。 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。

CSD2X_wGetRadialPos

说明：

检查质心的差值阵列。 如果存在，则根据 CSD2X 向导中指定的分辨率计算质心位置。 此函数仅可用于 CSD2X 向导定义的辐射状滑条。

C 原型：

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

汇编：

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

参数：

bSnsGroup A => 滑条编号

此参数是您使用的辐射状滑条的编号。 可以通过 CSD2X 用户模块向导从辐射滑条表示法的左侧获取其编号 （例如，s2：辐射滑条编号为 2）。

返回值：

辐射状滑条的位置值、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线更新。 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。

如果有任何滑条传感器处于活动状态，则该函数返回从零到 CSD2X 向导中设置的分辨率值之间的值。 如果没有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，则此子例程可能生成错误的质心结果。 设置噪声阈值时应小心 （显著大于噪声级别），以便噪声不会产生错误的质心。

CSD2X_wGetRadialInc

说明：

返回实际手指移位，即手指当前位置与先前位置之间的差值。 此函数与 CSD2X_wGetRadialPos() 配对使用，并采用后者生成的数据 （数据保存在内部变量中）。

C 原型：

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

汇编：

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

参数：

bSnsGroup A => 滑条编号

此参数是您使用的辐射状滑条的编号。 可以通过 CSD2X 用户模块向导从辐射滑条表示法的左侧获取其编号 （例如：s2 表示辐射滑条编号为 2）。

返回值：

手指移位值：顺时针为正、逆时针为负，A 中的 LSB 和 X 中的 MSB。

手指移位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 CSD2X_wGetRadialPos() 返回 -1 (FFFFh)）或者当前没有任何触摸 （此时 CSD2X_wGetRadialPos() 返回 -1 (FFFFh)）

副作用：

只能在 CSD2X_wGetRadialPos() API 之后调用该例程。 因为它使用由 CSD2X_wGetRadialPos(). 设置的内部数据 CSD2X_waSliderPrevPos 和 CSD2X_waSliderCurrPos

CSD2X_InitializeSensorBaseline

说明：

通过扫描选定的传感器为 CSD2X_waSnsBaseline[bSensor] 阵列加载初始值。 原始计数值将复制到所选传感器的基准线阵列元素中。 此函数可用于复位单个传感器的基准线。

C 原型：

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

汇编：

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

参数：

A => 传感器编号

返回值：

无

副作用：

**

CSD2X_InitializeBaselines

说明：

通过扫描每个传感器，加载含有初始值的 CSD2X_waSnsBaseline[] 阵列。 原始计数值将复制到每个传感器的基准线阵列中。

C 原型：

```
void  CSD2X_InitializeBaselines()
```

汇编：

```
lcall  CSD2X_InitializeBaselines
```

参数：

无

返回值：

无

副作用：

**

CSD2X_SetDefaultFingerThresholds

说明：

通过 “ 手指阈值 ”(FingerThreshold) 参数值加载 CSD2X_baBtnFThreshold[] 阵列。 如果 CSD2X_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  CSD2X_SetDefaultFingerThresholds()
```

汇编：

```
lcall  CSD2X_SetDefaultFingerThresholds
```

参数：

无

返回值：

无

副作用：

**

CSD2X_SetLeftDACValue

说明：

此函数将会覆盖用户模块参数设置中的左 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 CSD2X_ScanSensor(). 一起使用。此函数在 Rb 配置中不可用。

C 原型：

```
void  CSD2X_SetLeftDACValue(BYTE bIdacValue);
```

汇编：

```
mov  A,  bIdacValue
lcall  CSD2X_SetLeftDACValue
```

参数：

bIdacValue - 设置 iDAC 值。 接受的值为 1..255。

返回值：

无

副作用：

**

CSD2X_SetRightDACValue

说明：

此函数将会覆盖用户模块参数设置中的右 IDAC 值。 如果需要用不同的 iDAC 设置扫描某些传感器，则使用此函数。 此函数可以与 CSD2X_ScanSensor(). 一起使用。此函数在 Rb 配置中不可用。

C 原型：

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

汇编：

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

参数：

bIdacValue - 设置 iDAC 值。 接受的值为 1..255。

返回值：

无

副作用：

**

CSD2X_SetIdacValue

说明：

此函数将会覆盖用户模块设置中的 IDAC 值和 IDAC 补偿值。 如果需要用不同的 IDAC 设置扫描某些传感器，则使用此函数。 此函数可以与 CSD2X_ScanSensor(). 一起使用。此函数仅在单通道 IDAC 配置中可用。

C 原型：

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

汇编：

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

参数：

bIDACVal - 设置 IDAC 值。 接受的值为 1..255。

bCompIDACVal - 设置 IDAC 补偿值。 接受的值为 0 至 255。

返回值：

无

副作用：

**

CSD2X_SetRefValue

说明：

此函数将会覆盖用户模块参数设置中的参考值。 如果需要用不同的参考设置扫描某些传感器，则使用此函数。 此函数可以与 CSD2X_ScanSensor(). 一起使用。当使用 Rb 配置且参考值设置为 Vbg 或

外部引脚时中的某个值时，此函数不起作用。 这是因为在运行时不能调整 Vbg 参考电压或来自外部引脚的参考电压。

C 原型：

在单通道配置中：

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

在双通道配置中：

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

汇编：

在单通道配置中：

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

在双通道配置中：

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

参数：

bRefValue - 设置参考值。 接受的值为 0 至 31。

返回值：

无

副作用：

**

CSD2X_SetRefSource

说明：

此函数将会覆盖用户模块参数设置中的参考源。 如果需要用不同的参考设置扫描某些传感器，则使用此函数。 此函数可以与 CSD2X_ScanSensor(). 一起使用。

C 原型：

在所有进行了 IDAC 配置的单通道和双通道中：

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

在进行了 Rb 配置的双通道中：

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

汇编：

在所有进行了 IDAC 配置的单通道和双通道中：

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

在进行了 Rb 配置的双通道中：

```
 mov  A,  bLeftRefSource
```

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

参数：

bRefSource - 参考源常量。 下表中列出了接受的值：

常量 （对于 Rb 配置）	值
CSD2X_REFERENCE_ACOLUMN_MUX	0x01
CSD2X_REFERENCE_VBG	0x03
CSD2X_REFERENCE_ASE	0x04

常量 （对于 IDAC 配置）	值
CSD2X_REFERENCE_VDD	0x00
CSD2X_REFERENCE_2VBG	0x10

返回值：

无

副作用：

**

CSD2X_SetIdacRange

说明：

此函数将会覆盖用户模块参数设置中的 DAC 范围。 如果需要用不同的范围设置扫描某些传感器，则使用此函数。 此函数可以与 CSD2X_ScanSensor(). 一起使用。此函数在 Rb 配置中不可用。

C 原型：

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

汇编：

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

参数：

bRange - 设置参考值。 接受的值为下列常量之一。

设置	值	作用
CSD2X_IDAC_RANGE_1X	0x00	最大 IDAC 电流为 19.92 µA
CSD2X_IDAC_RANGE_4X	0x01	最大 IDAC 电流为 91.03 µA
CSD2X_IDAC_RANGE_16X	0x08	最大 IDAC 电流为 318.75 µA
CSD2X_IDAC_RANGE_32X	0x09	最大 IDAC 电流为 637.50 µA

返回值：

无

副作用：

**

CSD2X_Calibrate

说明：

此函数用于执行 CDS2X 自动校准。 启用 “ 自动校准 ” 之后，当用 CSD2X_Start() 函数启动用户模块时，系统将会自动调用此函数。 此函数在 Rb 配置中不可用。

C 原型：

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

汇编：

```
lcall  CSD2X_Calibrate
```

参数：

无

返回值：

无

副作用：

**

CSD2X_ClearSensors

说明：

通过针对每个传感器按顺序调用 CSD2X_wGetPortPin() 和 CSD2X_DisableSensor()，将所有传感器清除为非采样状态。

C 原型：

```
void  CSD2X_ClearSensors()
```

汇编：

```
lcall  CSD2X_ClearSensors
```

参数：

无

返回值：

无

副作用：

**

CSD2X_wReadSensor

说明：

返回 A (LSB) 和 X (MSB) 中的关键原始扫描值。

C 原型：

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

汇编：

```
mov  A,  bSensor
lcall  CSD2X_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：

**

CSD2X_wGetPortPin

说明：

返回指定传感器的端口号和引脚掩码。传递的参数对 CSD2X_Sensor_Table[]. 中的数据编制索引并进行选择。返回值可以传递给 CSD2X_EnableSensor()、CSD2X_DisableSensor().。此函数仅在单通道配置中可用。

C 原型：

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

汇编：

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

参数：

bSensor - 传感器数量，范围为 0 到 (n - 1)，其中 “n” 是 CSD2X 向导中设置的传感器数量与滑条中包含的传感器数量之和。CSD2X_wGetPortPin() 使用传感器编号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：

**

CSD2X_wGetPortPinLeft

说明：

返回连接到左通道的给定传感器的端口号和引脚掩码。 传递的参数对 CSD2X_Sensor_Table_Left[]. 中的数据编制索引并进行选择。返回值可以传递给 CSD2X_EnableSensor()、CSD2X_DisableSensor()..。此函数仅在双通道配置中可用。

C 原型：

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

汇编：

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

参数：

bSensor - 传感器数量，范围是 0 到 （n - 1），其中 “n” 是左通道 CSD2X 向导中设置的传感器总数（包括连接左通道的滑条段数量）。 CSD2X_wGetPortPinLeft() 使用传感器编号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：

**

CSD2X_wGetPortPinRight

说明：

返回连接到右通道的给定传感器的端口号和引脚掩码。传递的参数对 CSD2X_Sensor_Table_Right[]. 中的数据编制索引并进行选择。返回值可以传递给 CSD2X_EnableSensor()、CSD2X_DisableSensor()..。此函数仅在双通道配置中可用。

C 原型：

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

汇编：

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

参数：

bSensor - 传感器数量，范围为 0 到 （n- 1），其中 “n” 是右通道 CSD2X 向导中设置的传感器数量 （包括连接右通道的滑条段数量）。 CSD2X_wGetPortPinRight() 使用传感器编号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：

**

CSD2X_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 CSD2X_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 模式并启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  CSD2X_EnableSensor(BYTE bMask, BYTE bPort)
```

汇编：

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

参数：

A => 传感器位图

X => 端口号

返回值：

无

副作用：

**

CSD2X_DisableSensor

说明：

禁用 CSD2X_wGetPortPin() 函数选择的传感器。 驱动模式更改为 “ 强 (001)” 。 这可以将传感器有效接地。 端口引脚与 “ 模拟复用器总线 ”(AnalogMuxBus) 的连接关闭。 函数参数由 CSD2X_wGetPortPin() 函数返回。

C 原型：

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

汇编：

```
mov  X,  bPort
mov  A,  bMask
lcall  CSD2X_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 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
}
}
```

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 CSD2X_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。 可以使用示例进行器件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 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
}
}
}
```

示例 3。 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 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
}
}
```

}

配置寄存器

Table 4. 模块 CapSense、寄存器： CS_CR0

位	7	6	5	4	3	2	1	0
值	0	0	CSD2X_PRSCLK	0	1	0	0	EN

Table 5. 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 6. 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 7. 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 8. 模块 CapSense、寄存器： CS_CNTH

位	7	6	5	4	3	2	1	0
数据输出 MSB								

Table 9. 模块 CapSense、寄存器： CS_CNTL

位	7	6	5	4	3	2	1	0
数据输出 LSB								

Table 10. 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 11. 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 12. 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 13. 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							

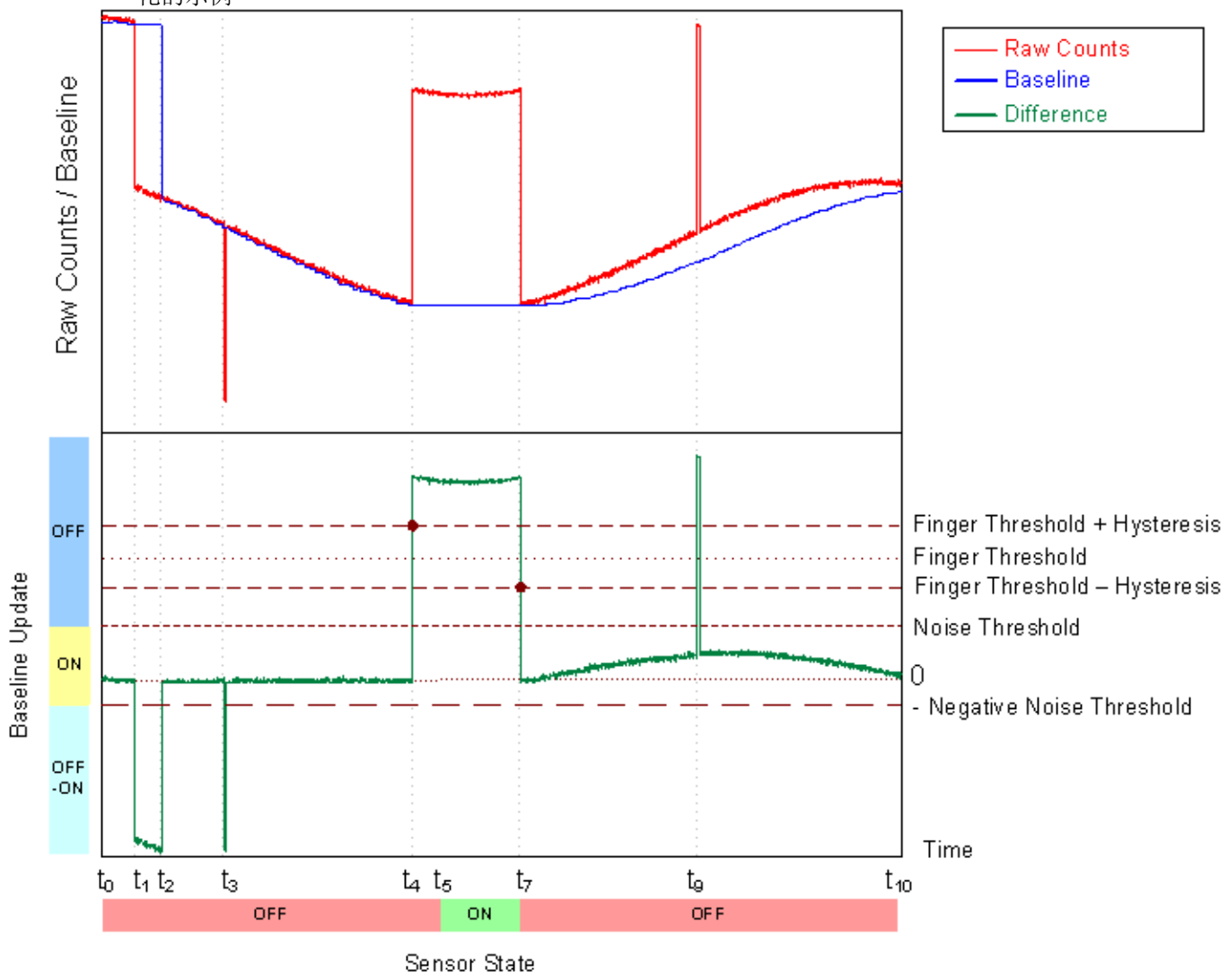
附录

以下部分介绍用户模块数据手册中通常没有包含的信息。 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到应用笔记中。

CSD2x 参数的交互

下图说明了基准线更新和决策逻辑操作，对于更好地了解如何设置用户参数以获得最佳性能很有帮助。 第一个图形说明了当 “ 传感器自动复位 ”(Sensors Autoreset) 参数设置为**禁用**时的系统操作。 第二个图形说明了 “ 传感器自动复位 ”(Sensors Autoreset) 参数设置为**启用**时的情况。 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 8. 在 “ 传感器自动复位 ”(SensorsAutoreset) 设置为 “ 禁用 ” 情况下原始计数、基准线、差值信号变化的示例



在 t_0 处，原始计数接近于基准水平，然后由于湿度或温度变化，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 “ 负噪声阈值 ”(NegativeNoiseThreshold) 参数 （绝对值），因此通过跟踪原始计数最小值来更新基准线，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器件加电，过一段时间后手指移开，则会发生这种情况。 此时，基准线更新机制冻结，内部超时计数器会激活。 当差值信号低于 “ 负噪声阈值 ”(NegativeNoiseThreshold) 的次数为 “ 低基准线复位 ”(LowBaselineReset) 采样值 时，基准线复位。 这是在 t_2 处发生的。

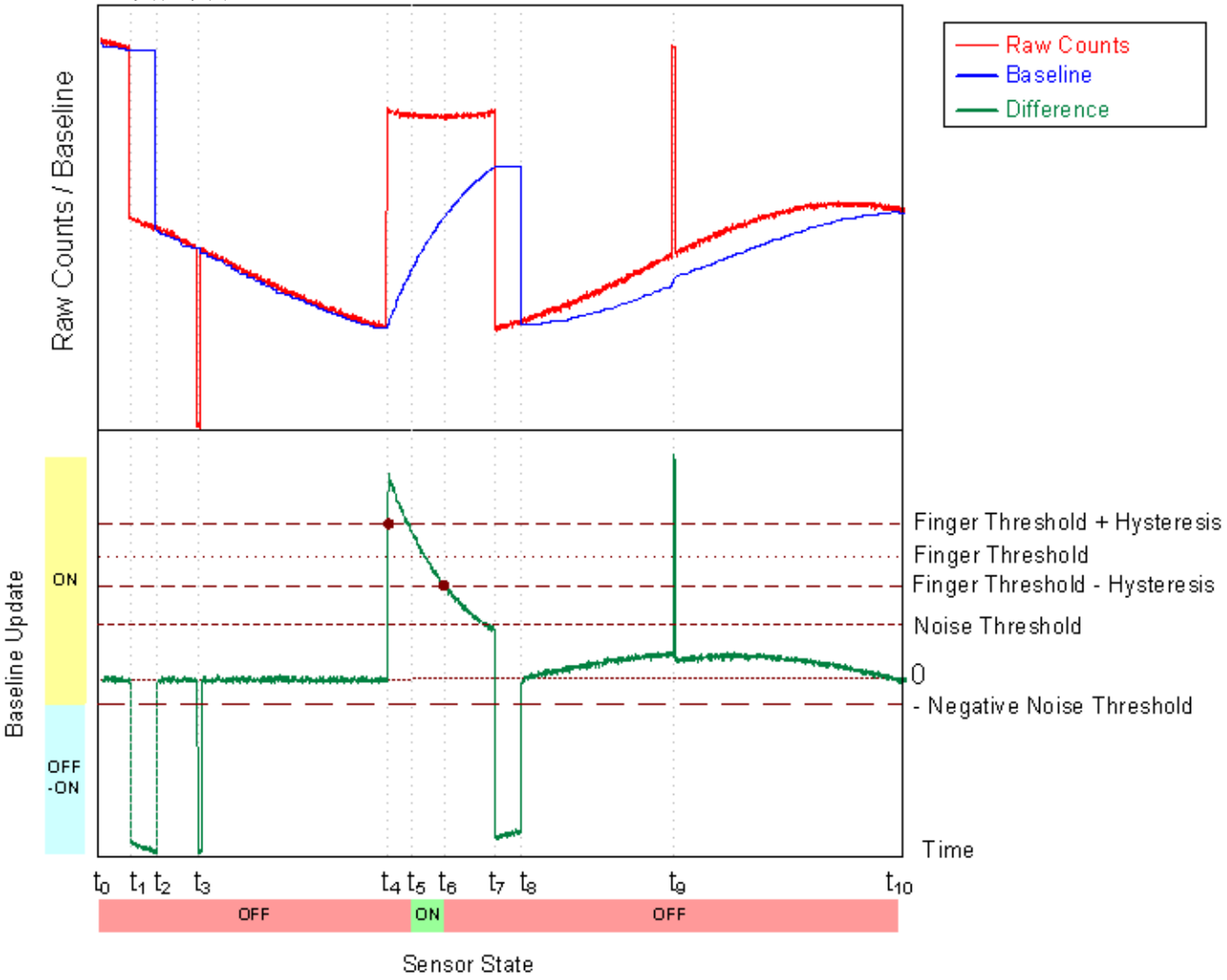
第二大负差值信号尖峰脉冲在 t_3 处发生，此尖峰脉冲可能由 ESD 事件触发。 由于采样计数中的尖峰脉冲持续时间小于 “ 低基准线复位 ”(LowBaselineReset) 参数，因此保留基准线，对尖峰脉冲进行滤波。 这可以阻止假基准线复位和导致假触摸检测。

传感器是在 t_4 处被触摸的。 当差值信号超过 “ 手指阈值 + 迟滞 ”(FingerThreshold + Hysteresis) 值时，内部防反跳计数器会激活。 如果信号超过此值的量大于防反跳采样数，则传感器状态设置为开启。 这是在 t_5 处发生的。 当差值信号在 t_7 处下降到 “ 手指阈值 - 迟滞 ”(FingerThreshold -

Hysteresis）水平之下时，传感器立即恢复为关闭状态。 t_9 处短的正尖峰脉冲已被防反跳计数器滤波，这是因为样品单元中的尖峰脉冲持续时间不会超过防反跳值。

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

Figure 9. 在 “ 传感器自动复位 ”(SensorsAutoreset) 设置为 “ 启用 ” 的情况下原始计数、基准线、差信号的变化示例



上图中的系统操作与上一实例中的操作类似，但有以下区别：

- 在 t_6 处触摸传感器时，触摸持续时间因为活动基准线值更新算法而下降。
- 手指移开后，基准线会在 LowBaselineReset （低基准线复位）采样 (t_8) 后复位，??会短时阻止触摸检测。 这可用作附加的防反跳机制。

双通道扫描

双通道扫描是以同步阻塞函数形式进行的。

同步意味着同时扫描左右通道传感器，只有在完成上一对传感器的扫描之后，才能开始扫描下一对传感器。 左右传感器可以具有不同的分辨率和扫描时间。 在此情况下，“Scan Sensor” 函数等待较慢的传感器，不对另一个传感器执行任何操作。 如果左右传感器阵列由不同数量的传感器组成，则 “ScanAllSensors” 函数：

- 成对扫描所有可能的传感器
- 如果一侧的传感器比另一侧多，则扫描该通道上其余的传感器时一次扫描一个

扫描从阵列末尾开始。 在 GUI 中根据从左上到右下的传感器放置顺序分配传感器位置。 要尽量确保您的双通道扫描高效进行，请执行下列操作：

- 在每个通道中放置相同数量的传感器
- 排列传感器，以便扫描时间较长的传感器成对放置
- 将所有滑条段放置在同一通道中
- 将大型传感器放置在左通道中。
- 如果使用不同的参考，则首先放置具有较高参考值的传感器

版本历史记录

版本	创作者	说明
1.0	DHA	初始版本

版本	创作者	说明
2.1	DHA	1. 从 CSD2x 向导中移除了下列参数： IDAC 值、手指阈值、参考值、扫描速度、扫描分辨率 2. 在向导中禁用了用于设置每个传感器扫描参数的选项。 这可以节省 RAM 和 ROM 空间。 3. 向用户模块参数窗口中添加了下列新参数： 手指阈值、参考值 （左 / 右）、IDAC 值 （补偿 / 左 / 右）、扫描速度、分辨率。 这些参数在启动时应用于所有传感器。 4. 参考值 （左 / 右）参数分辨率增加到 5 比特。 5. 增加了下列新 API 函数，以便与其他 CapSense 用户模块保持一致： SetScanMode （左 / 右）、SetPrescaler （左 / 右）、SetRefSource （左 / 右）、SetRefValue （左 / 右）、SetDACValue （左 / 右） 6. RAM 中存储了所有扫描参数。 这允许在运行时为单个传感器设置不同的扫描参数。 7. 现在，在启动时自动完成自动校准（如果启用）。 不需要手动调用 CSD2X_Calibrate API。 8. 更新了 IDAC 值输入范围。 9. 更新了参考值和自动校准参数的说明。 10. 更新了 PSoC Designer 中的参数顺序。 11. 添加了对数据表部分的引用。 12. 向数据表添加了 SetRefSource API。 13. 更新了下列 API 的说明： CSD2X_DisableSensor, CSD2X_Start, CSD2X_SetScanMode 14. 更改了 MUM 配置说明。 15. 添加了对新部件号的支持。 16. 添加了 LowBaselineReset、NegativeNoiseThreshold 和模块电容引脚参数的范围。 17. 解决了手指阈值参数的范围问题。 18. 添加了 SetLeftRefSource 和 SetRightRefSource API 的原型。 19. 解决了其他不足并清理了用户模块源代码。
2.20	DHA	添加了针对未分配传感器的 DRC 警告消息。

版本	创作者	说明
2.30	DHA	1. 在 Start() API 中将 RAM_EPILOGUE 更改为了 RAM_USE_CLASS_3。 2. 将 DiplexTables 和 Order_Table_Left/右表移入了 AREA lit。 3. 添加了向导帮助按钮和文件。
2.40	DHA	1. 在 CSD2X_InitializeSensorBaseline() API 函数实现中将 CSD2X_Order_Table_Left 更改为了 CSD2X_Order_Table_Right，以修复基线初始化。 2. 在 API 函数中增加了 CSD2X_Start() “call @INSTANCE_NAME`_ScanAllSensors”。 3. 添加了 CSD2x_baDACCodeBaselineL 和 CSD2x_baDACCodeBaselineR IDAC 表格导出。 4. 更新了 CSD2X_wGetPortPinLeft() 和 CSD2X_wGetPortPinRight() API 函数说明。 5. 在 “ 反馈电阻引脚 ” 参数说明中添加了新的 “GOO[4]” 和 “GOO[5]” 选项。 6. 添加了有关扫描时间对预分频器值依赖性的更多信息。 7. 将 CSDx_CR1 寄存器中的 ACOL[1:0] 位字段值设置为了 01b。 8. 在 SetIdacRange() API 函数中添加了 RAM_SETPAGE_CUR >`@INSTANCE_NAME`_bBitMask。

Note PSoC Designer 5.1 在所有用户模块数据手册中都引入了 “ 版本历史 ”。 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 © 2009-2012 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.