初版

CapSense® Sigma-Delta 数据手册 CSD V 1.90

资源	PSoC® 模块				API 存储器 （字节）		每个传感器所使用的引脚数
	CapSense®	I^2C/SPI	定时器	比较器	闪存	RAM	
CY8C20xx7/S、 CY8C20055							
用户模块	1	–	1	1	1146	40	1
滑条 API	–	–	–	–	2568	103	0
每个传感器	–	–	–	–	7	12	1

特性和概述

- 具有抗 GPIO 电流瞬变、 V_{dd} 波动、进入和退出睡眠以及 IDAC RTS 噪声的能力。
- 使用 sigma-delta 数据转换方法以执行 CapSense® 电容式感应。
- 通过可配置系统参数可以在各种应用中优化调试的性能。
- 支持高达 35 个电容式传感器或 6 个滑条。
- 能够检测低至 0.1 pF 的触摸；因此，在多达 15 mm 厚的玻璃或 5mm 厚的塑料的情况下，仍然可以检测到手指。
- 支持将电容式传感器配置为独立按键和 / 或作为相关阵列以形成滑条。
- 通过传感器复用技巧，滑条元件的有效数目能达到所使用的 I/O 引脚数的两倍。
- 支持通过插值将滑条的分辨率提高到物理间距以上。
- 利用屏蔽电极能够以较高的寄生电容和 / 或在有水膜的情况下进行可靠的操作。
- 通过 CSD 向导完成传感器和引脚分配。
- CY8C20055 系列不支持滑条。

采用 Sigma Delta 调制器 （CSD）用户模块的电容式感测是基于差分电容式感应方法的。该用户模块通过使用模拟复用器总线将电容式感应模拟电路连接到任何 PSoC 引脚。CSD 用户模块可将生效的传感器连接到模拟复用器总线，从而 CapSense 电路可以测量传感器电容，并将该电容值转换为数字代码。通过对 MUX_CRx 寄存器中的相应位依次进行设置，固件可以连续扫描各个传感器。

快速入门

1. 将需要专用引脚 （例如 I2C 和 LCD）的用户模块选择并放置。根据需要分配端口和引脚。
2. 选择并放置 CSD 用户模块。
3. 在工作区浏览器中右键单击 CSD 用户模块，以访问 CSD 向导 （该向导在本数据手册后续加以介绍）。
4. 对传感器、滑条或旋转滑条的所需数量进行设置。
5. 对于滑条，输入滑条特有的参数。
6. 将每个传感器分配到一个未使用的引脚。
7. 在 CSD 向导内，指定与外部调制电容相连的引脚。
8. 输入用于屏蔽传感器 （若适用于用户模块参数列表）的引脚名称 （请参考下面部分中的说明）。
9. 生成应用，并切换到应用编辑器。
10. 根据需要调整示例代码，以实现独立传感器、滑条传感器和 / 或触摸板。
11. 通过使用 PSoC Designer™ 所生成的 HEX 文件来对目标电路板上的 PSoC 进行编程。

简介

CapSense 是一种人机界面技术，通过使用由导电表面 （通常是蚀刻在 PCB 上的焊盘）构成的传感器检测人体的电容而实现。由于 CapSense 检测人体电容，所以它能够透过塑料或玻璃等绝缘外覆层来检测。这些外覆层通常构成了设备的外型。这些特性使 CapSense 成为按钮和电位器等机械输入器件的完美替代品。 CapSense 的主要优势包括：

- 更清洁，更美观的设计。
- 为更小终端产品提供更小的外形。
- 高级的用户接口特性，如 LED 效果和接近传感。
- 提供可靠性，因为各组件没有有限的生命周期。
- 提高了抵抗破坏的能力，因为表面没有机械破坏，
- 降低加工成本因为不需要机械加工设备的穿透或其他机械工作。

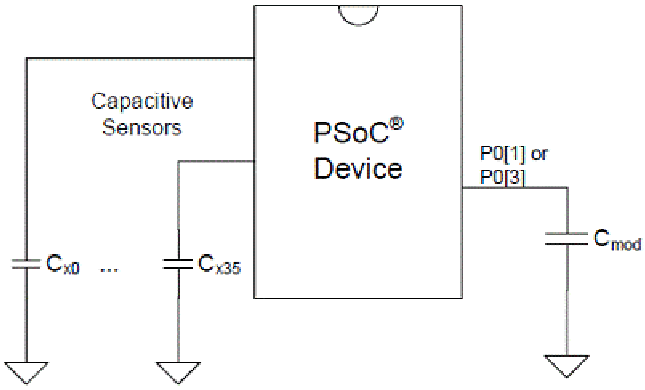
CSD 是赛普拉斯 CapSense 电容式感应系列中的一个方法。CSD 使用数字代码转换电路的 sigma-delta 电容，它的重要属性包括低 EMC 辐射和对 EMI 的出色抗噪性能。

CSD 用户模块包括 PCB 等级、 IC 等级和软件组件：

PCB 等级

图 1 显示的是 CSD 的原理图。物理传感器通常是安装在与 PSoC 相连的 PCB 上的导电纹路，并在其上面具有一个绝缘覆盖层。有关 PCB 等级 CapSense 实现的详细信息，请参考 CY8C20xx7/S CapSense 设计指南和 CapSense 入门。

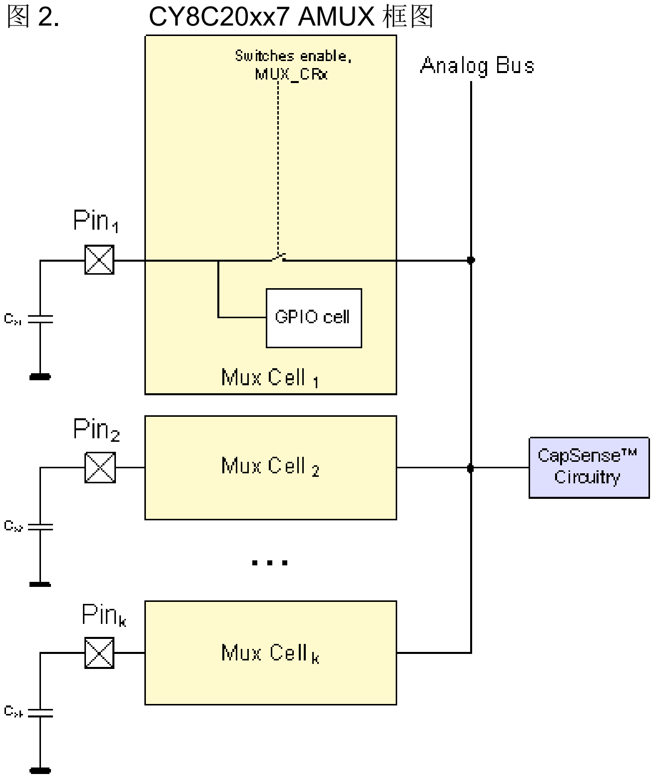
图 1. CSD 原理图



IC 等级

通过 CY8C20xx7 器件的模拟复用器总线可以将电容感应模拟电路连接到任何 PSoC 引脚。 CSD 用户模块可将活动传感器连接到模拟复用器总线，从而 CapSense 电路可以测量传感器电容，并将该电容值转换为数字代码。通过依次设置 MUX_CRx 寄存器中的相应位，固件可以连续地扫描各传感器，如图 2 所示。

图 2. CY8C20xx7 AMUX 框图



软件

CSD 软件组件的属性如下所示：

- 通过可调试系统的配置参数可以在各种应用中优化调试的性能。
- 运行时， API 函数对电容转换电路中的原始计数值进行分析，以确定传感器状态和补偿环境变化。
- 对于连续、复合传感器 （例如：滑条和触摸板），会提供 API 函数，以便插入一个分辨率高于传感器物理分辨率的位置。
- 高层级的软件功能可适应滑条复用法，从而使单个 I/O 引脚能够连接到两个物理传感器上，以便将针对给定数量的滑条元件消耗的 I/O 数减半。

建议读取

在使用 CSD 用户模块实施 CapSense 设计之前，建议阅读以下文档：

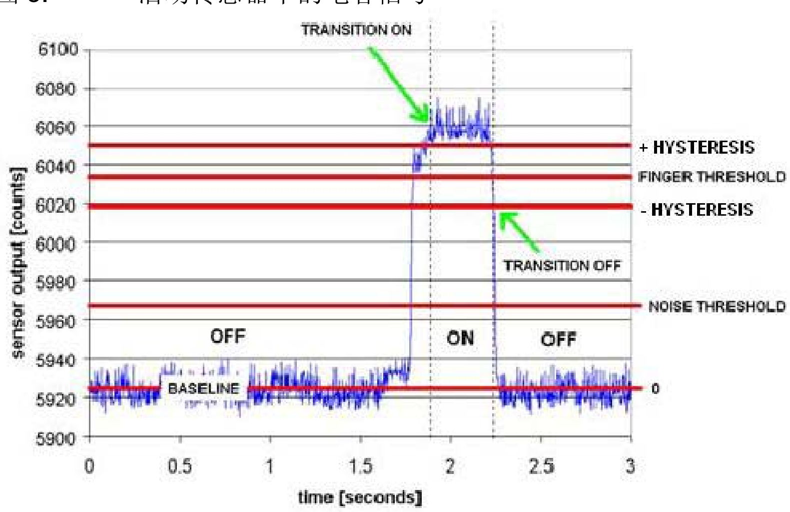
- Capsense 入门手册
- CY8C21x34/B CapSense 设计指南
- CY8C20x34 CapSense 设计指南
- CY8CMBR2044 CapSense 设计指南

电容式感应的实现

按键

CapSense 按键类似于机械式按键。这些按键可用于离散控制，如打开 / 关闭开关、功能键、菜单键等。API 函数对各传感器中的电容信号 （原始计数）进行监控并将这些值与可调试阈值参数进行比较。传感器被触摸时，它的电容信号将增加；如果该增量满足 CSD 决策逻辑，那么该传感器将被激活。图 3 显示的是活动传感器中的一个典型信号 （蓝线）。通过调试各阈值 （红线）可以得到所需要的性能。

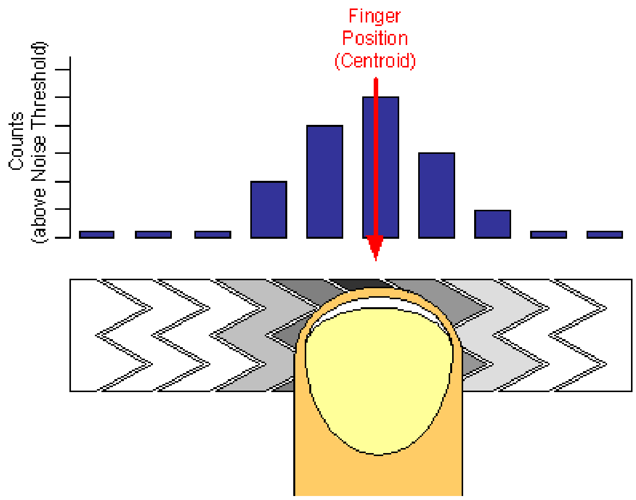
图 3. 活动传感器中的电容信号



滑条

CapSense 滑条类似于机械电位器。各滑条用于需要连续电平的控制，如照明调光器、音量控件、图形均衡器、速度控件，等等。通过使用一系列电容传感器可以实现 CapSense 滑条。当手指启动某一滑条时，一些相邻传感器将记录电容信号的增量，如图 4 所示。通过计算活动传感器组的中心位置，可以确定触摸的正确位置。滑条传感器中的传感器实际最小数量为五，最大值受限于 PSoC 器件上可用的 I/O 引脚数量。

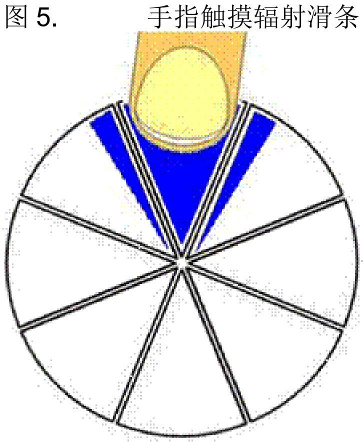
图 4.　　滑条上手指的插值中心位置



辐射滑条

CSD 支持两种滑条类型：线性和辐射。线性滑条有起始点和结束点，辐射滑条 （如图 5）却没有。在两种情况下，发生触摸时，坐标算法计算各个传感器的信号 （这些传感器与具有最大信号的传感器相邻），以计算触摸的正确位置。辐射状滑条未采用复用法。 CSD 用户模块包含两个支持辐射滑条的 API 函数。第一个函数 CSD_wGetRadiaPos() 返回中心位置，第二个函数 CSD_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， CSD_wGetRadialInc() 会返回一个正偏移。参考点 （0）位于第一个传感器的中心。线性滑条和辐射滑条都受限制，其限制为 （传感器所用的引脚数量 – 1）x 2^8 – 1，对于复用型滑条该值为 （2 x 传感器所用的引脚数量 – 1）x 2^8 – 1。

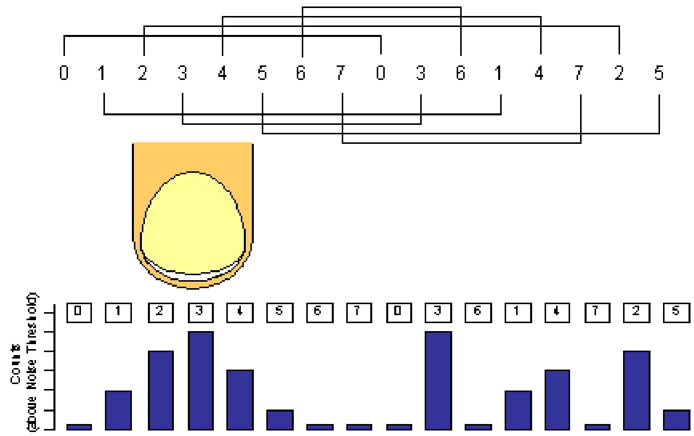
图 5.　　手指触摸辐射滑条



复用

使用复用法时，作为滑条元素的每个 PSoC 引脚都会映射到滑条传感器阵列中的两个物理位置上。物理位置的前半 （数字上较低的）部分根据 CSD 向导分配的端口引脚被映射。物理传感器位置的后 （或上）半部分将通过图 6 中所示的模式被自动映射。

图 6.　　由 CSD 进行的复用滑条阵列索引编制



越接近滑条下半部分的强信号，将导致上半部分产生相同程度的伪信号。然而，在上半部分中，这些结果被分散和断连。坐标算法搜索相邻最强的一组信号，以确定解析的滑条位置。映射上半传感器的格式确保该半部分中的有效信号格式不会导致剩下半部分的有效信号格式，如图 6 所示。

要小心确保传感器与印制电路板引脚的映射符合复用算法所用的 3 倍索引模式。复用滑条中传感器对的电容应该合理匹配 （不超过 10 pF）。当选择复用时，复用传感器索引表由 CSD 向导自动生成。表 1 显示的是复用成 28 个 PSoC I/O 引脚的 56 个滑条段的复用序列。

表 1.　　不同滑条段计数的复用序列

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

滑条段总计数	段序列
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

外部组件选择 （C_{mod}）

CSD 需要一个外部调制电容 C_{mod}，从 Vss 连接到两个专用 PSoC 引脚的其中一个：P0[1] 或 P0[3]。可以在 “Global Settings （全局设置）-> Modulator Capacitor Pin （调制电容引脚）” 下面的 CSD 向导进行

C_{mod} 引脚分配。所选引脚不得用于其他任何用途。外部调制电容的建议值为 2.2 nF。应该使用陶瓷电容。温度电容系数并不重要。强烈建议在射频干扰抑制的所有 CapSense 传感器走线上使用 560 Ω 串联电阻。该电阻必须尽可能接近 PSoC。

驱动屏蔽电极

通过一个驱动屏蔽电极可以降低传感器的寄生电容（C_p）。其目的在于当覆盖层上有水时将提高传感器灵敏度并防止错误传感器触摸。

屏蔽电极应该位于感应电极背面或其外侧，如图 7 所示。当水滴落到覆盖层上，而且没有驱动屏蔽电极时，各传感器和 PCB 的其他导体之间的电容耦合或寄生电容将增加。这样，将使传感器电容信号相应增加，可达到能够错误激活传感器的值。驱动屏蔽电极抵消寄生电容耦合，因此水滴对传感器电容信号没有产生影响，从而防止错误激活。

图 7.　　驱动屏蔽电极 PCB 布局

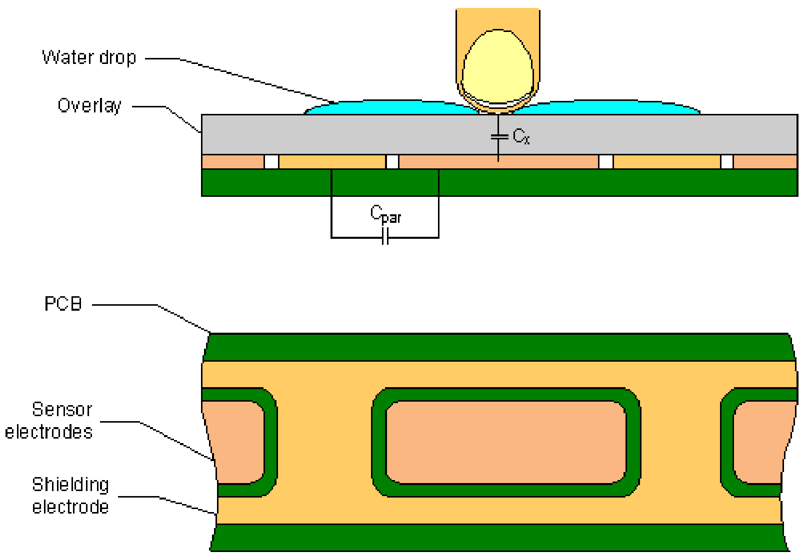
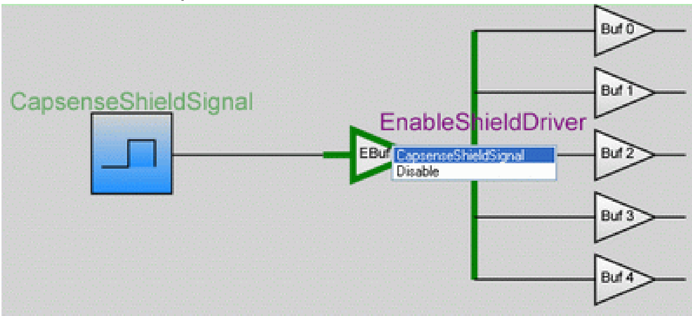


图 7 显示的是按键的驱动屏蔽电极。作为另一替代方法，屏蔽电极可以安装在相对的 PCB 层上，其中包括按键下面的平面。对于这种情况，建议使用填充模式，填充率约为 30 到 40%。不需要其他接地层。

屏蔽电极可以连接到任何专用的 PSoC 引脚：P2[4]、P2[2]、P0[2]、P0[0] 或 P1[2]。必须将选定引脚的驱动模式设置为 **High Z 模拟**。可在 PSoC 器件和屏蔽电极之间连接 560 欧姆上升限制电阻，以减少散发的电磁干扰。

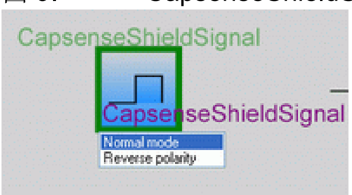
Capsense 屏蔽信号可以通过 5 个 ShieldBuffer （屏蔽缓冲区）路由到 P2[4]、 P2[2]、 P0[2]、 P0[0] 或 P1[2] 引脚，并由 PSoC Designer 芯片编辑器中的 EnableShieldDriver （使能屏蔽驱动）模块使能。

图 8. Capsense 屏蔽驱动



可以将屏蔽驱动配置为在两种模式下驱动屏蔽信号：正常模式和反向极性。当选择反向极性模式时，会反转屏蔽驱动时钟。

图 9. CapsenseShieldSignal 模块



电源要求

表 2. CSD 电源要求

参数	最小值	典型值	最大值	单位	测试条件和注释
V_{DD}	1.8[a]	–	5.50	V	如果 V_{DD} 下降率超过基本 V_{DD} 的 5%， V_{DD} 下降和恢复的比率不能超过 200 mV/s。

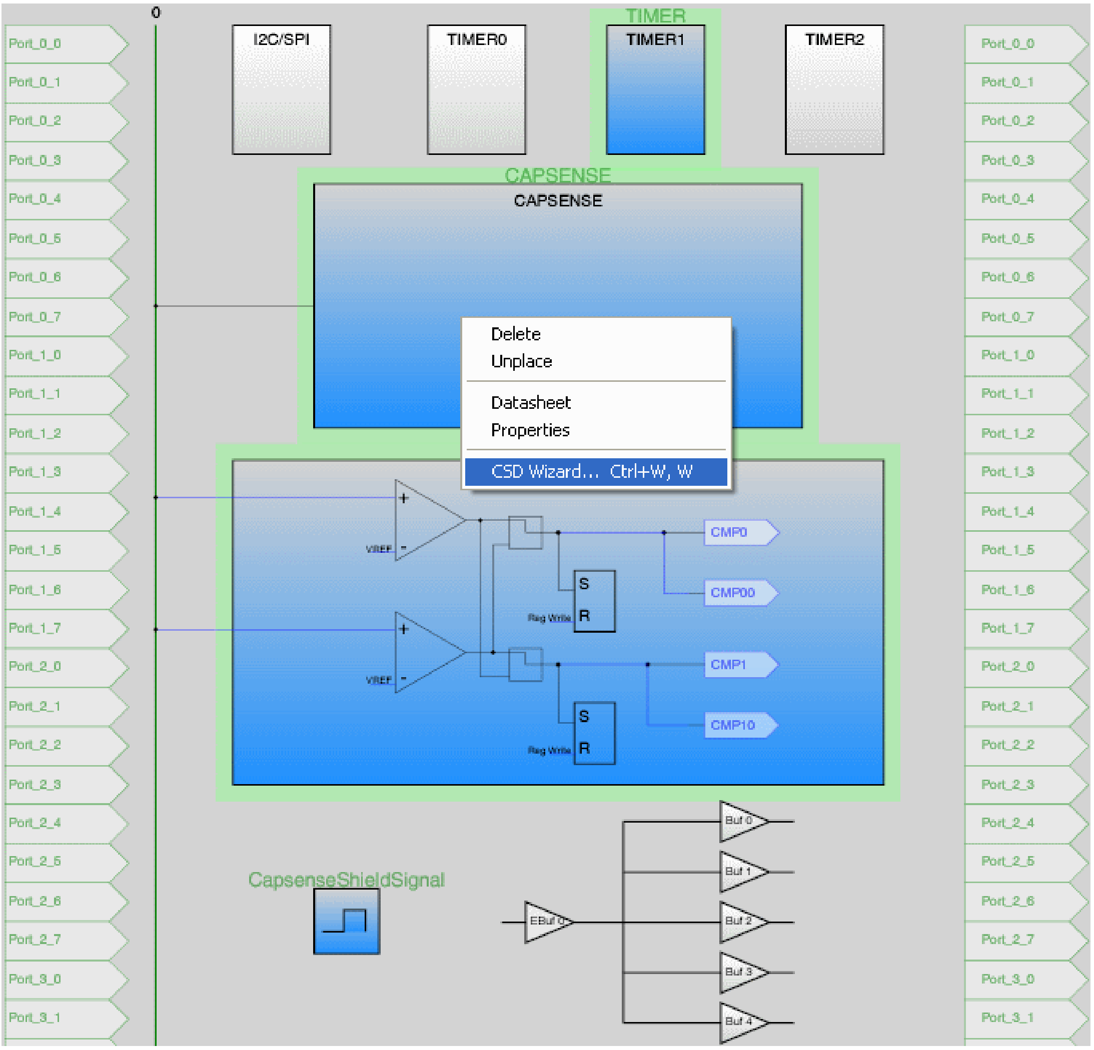
a. 最小绝对 VDD 规格为 1.8 V。 V_{DD} 下降到 1.8V 以下会引入过多噪声。

放置

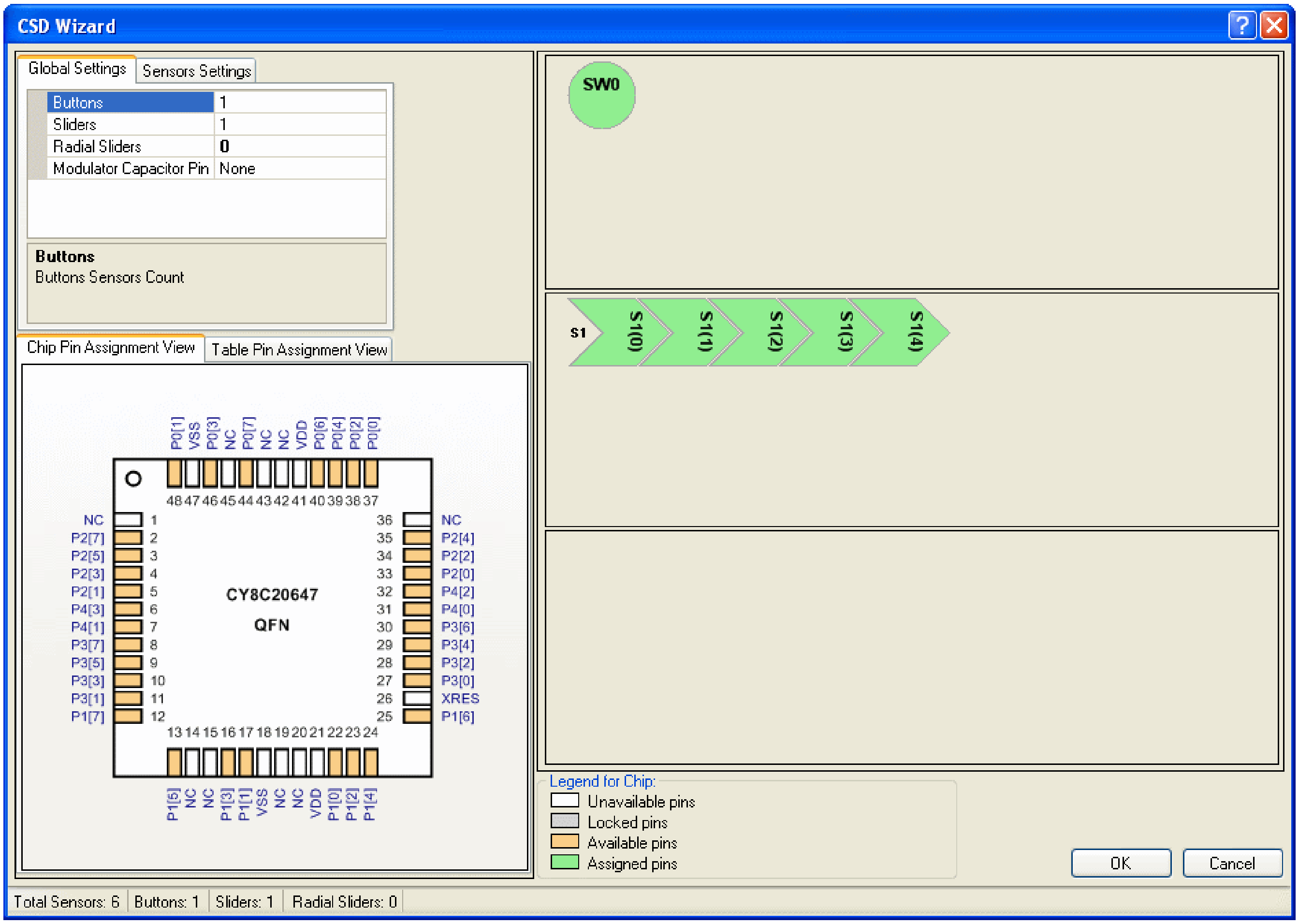
实例化用户模块时， CapSense 和 Timer1 模块将被分配到 CSD。备用放置不可用。必须在启动 CSD 向导之前，对需要专用引脚资源 （包括 LCD 和 I2CHW）的用户模块进行放置。这样，在传感器映射到 CSD 向导中的 I/O 引脚时，专用引脚将被保留而且不能作为传感器无意激活。在放置电容传感器连接时，请勿使用 P1[0] 和 P1[1]。这些引脚用来对部件进行编程，而且有可能存在过大的布线电容值，从而会影响传感器的灵敏度。

CSD 向导

1. 要访问向导，请在 “ 芯片编辑器互连视图 ” 中右键单击任意 CapSense 模块，然后通过鼠标左键单击选择 “CSD 向导 ” 。



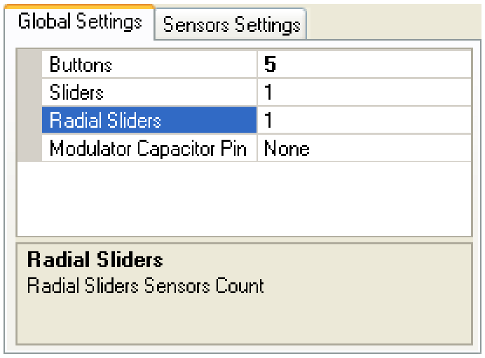
2. 向导打开后，会显示传感器和滑条传感器数量的数值输入框。



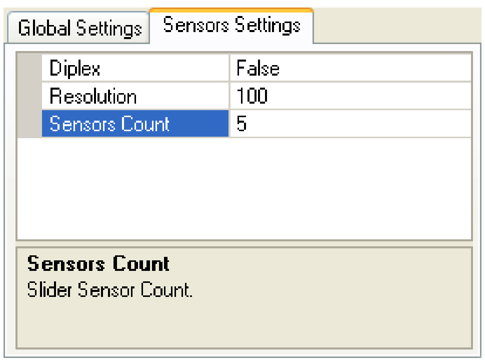
向导引脚图标

- 白色 — 该引脚不能作为 CapSense 输入使用。
- 灰色 — 引脚被锁定。此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 **Select** （选择）菜单中选择 **Default** （默认）。现在可以在向导中分配引脚了。
- 橙色 — 可以分配引脚。
- 绿色 — 引脚已分配为 CapSense 输入。

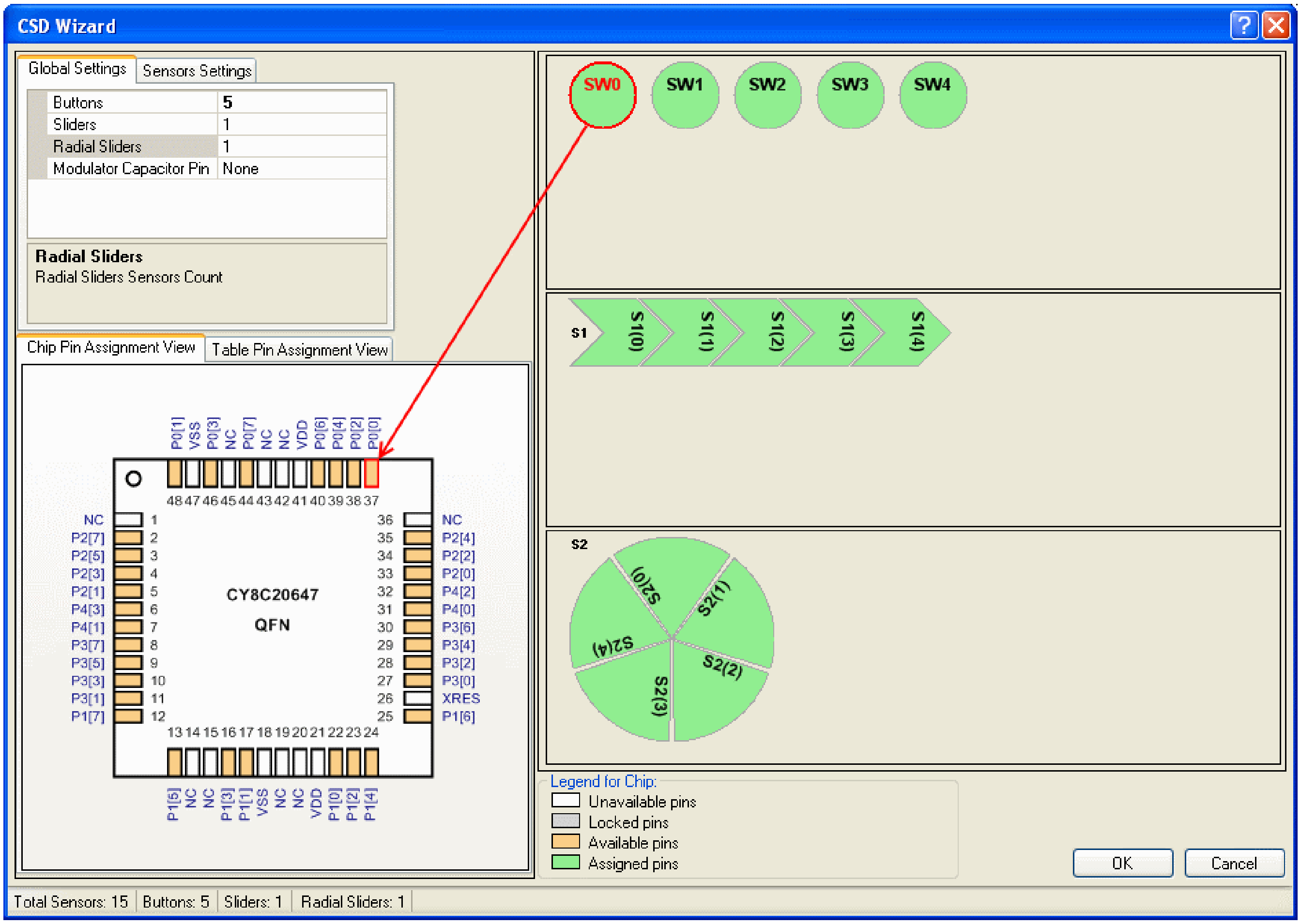
3. 键入独立按键、滑条和辐射滑条的数量。传感器 （按键加滑条元件）总数不得超过可用引脚数。输入数据后，按 [Enter] 键更新显示屏使其显示新值。



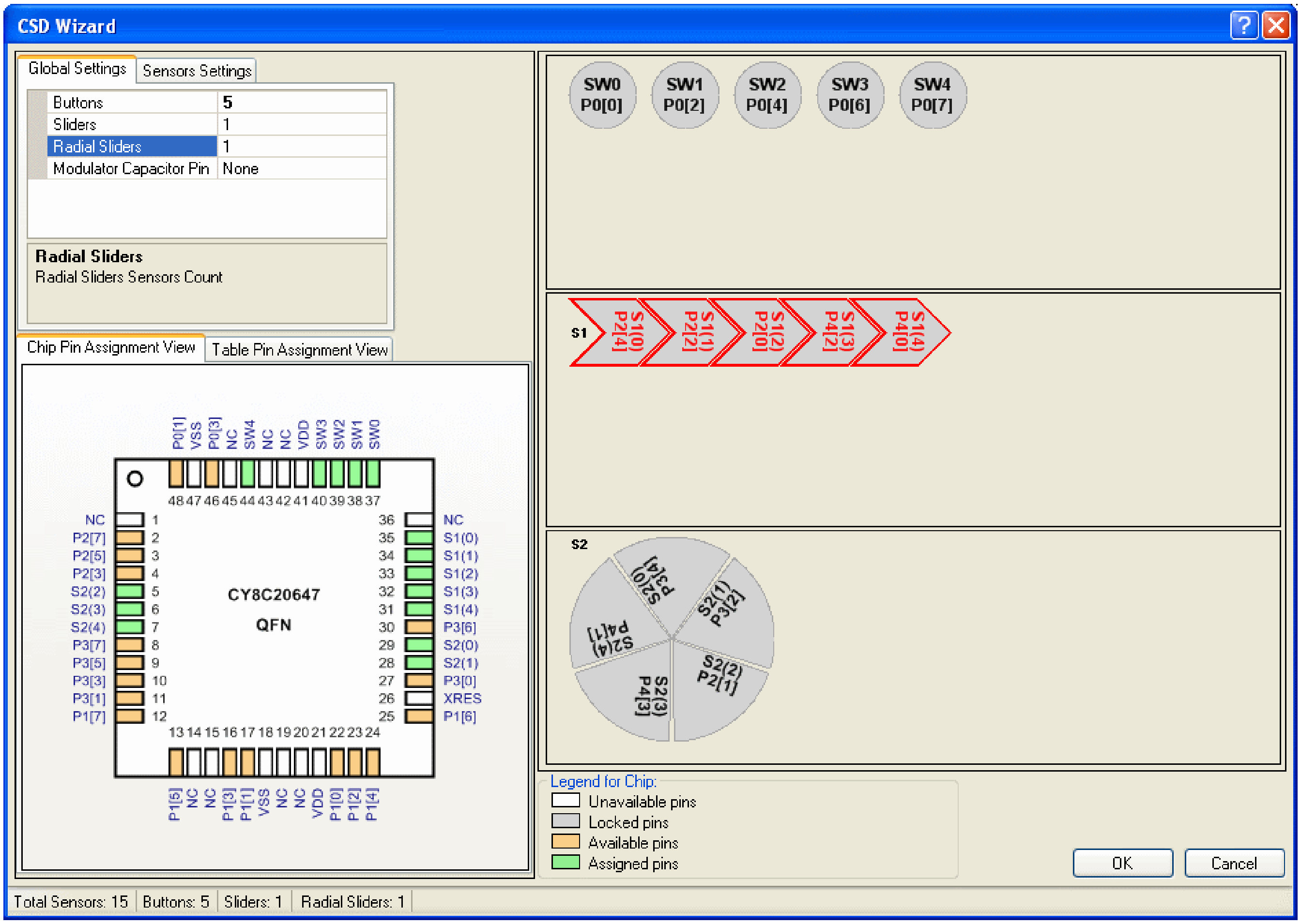
4. 选择 “ 传感器设置 ” 即可设置滑条和辐射滑条。要更改设置，请单击以激活其中一个滑条。键入每个滑条中传感器元件的数量。滑条传感器中的传感器实际最小数量为五，最大值受限于引脚数量。输入数据后，按 [Enter] 键更新显示屏。



5. 选择调制器电容 （C_{mod}）引脚。您可以选择 P0[1] 或 P0[3]。
6. 键入输出分辨率。最小值为 5。使用公式 （传感器的引脚数 – 1） x 2^8 – 1 可以计算最大值；或者对于复用型滑条，公式为 （2 x 传感器的引脚数 – 1） x 2^8 – 1。 CSD 尝试使用相邻段的相对强度将触摸结果插入到指定的分辨率中。软件在滑条报告的触摸结果是零和分辨率 –1 之间。
7. 如果需要，选择 “ 复用 ”。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。仅显示了复用传感器的前半部分；后半部分按前面 “ 复用 ” 章节所述自动映射。有关引脚连接的复用表，请参见 “ 复用 ” 一节。



8. 在引脚分配视图中将传感器拖到引脚上，这样就可以将传感器分配到各个引脚。您可以选择在 “ 芯片引脚分配视图 ” 或 “ 表引脚分配视图 ” 中将传感器拖放到引脚上。选择 I/O 引脚后，引脚变为绿色，不再可用。通过将端口引脚拖回未指定的表格，可改变传感器的分配情况。确保避免选择已经指定用于其他用户模块的引脚。
9. 对其他传感器重复以上操作。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。右键单击 “ 芯片编辑器 ” 窗口并选择刷新以更新引脚连接。



要更改引脚分配，请将光标放在已分配的引脚上，单击该引脚，然后将其拖放至开关框外面。此时引脚就会处于未分配状态，您可以重新分配。

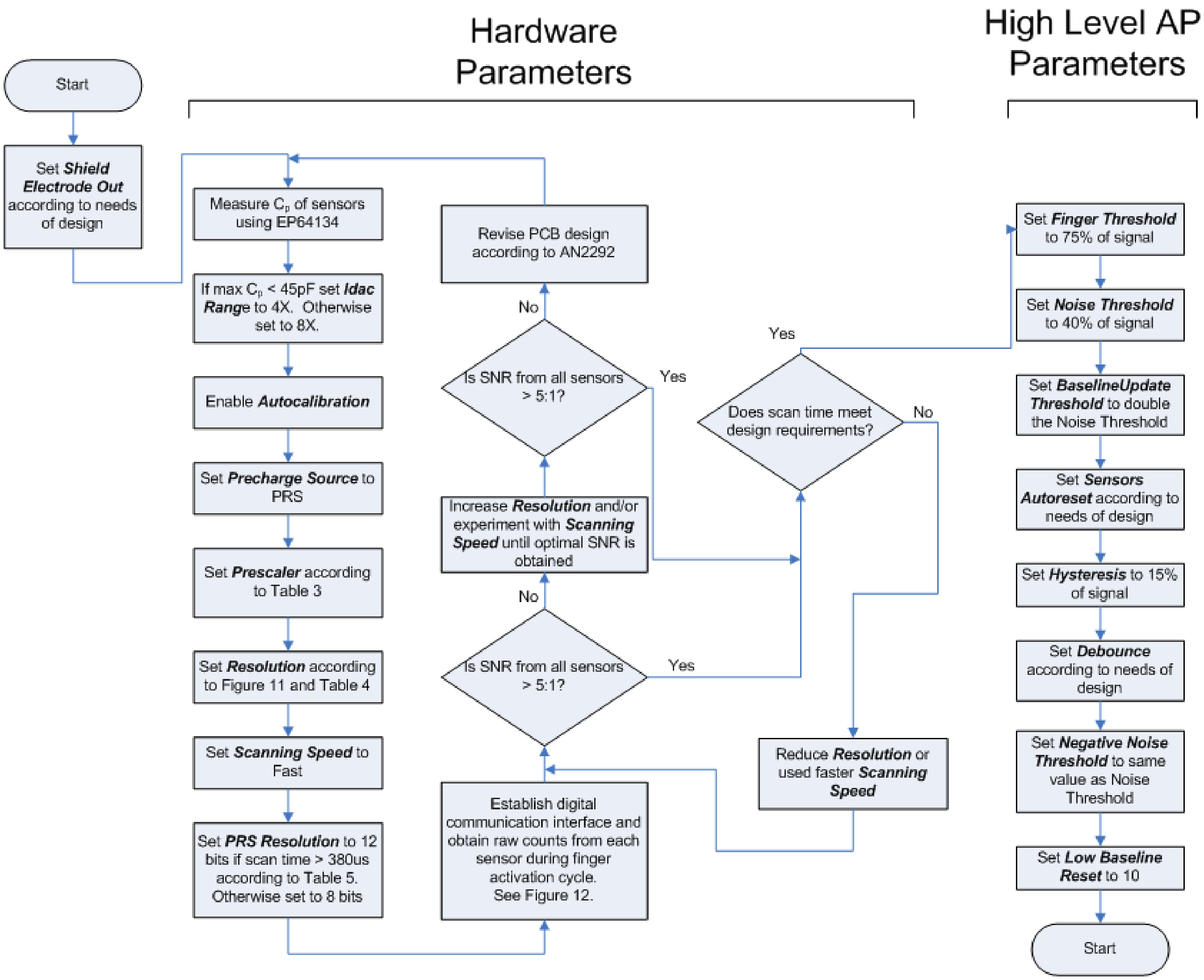
完成向导后，单击 “ 生成应用程序 ” 。根据您输入的传感器数量、引脚分配、复用和分辨率，会生成一系列表格。这些表格位于 CSD_Table.asm 中。

用户模块参数 — 调试指南

在 CSD 向导中完成配置和 I/O 引脚分配之后，必须设置模块参数。请注意，对于任何用户模块参数，必须重新生成对项目产生影响的更改。

图 10 展示的是 CSD 用户模块参数调试过程的流程图。 CSD 用户模块可分为两种广义类别：硬件参数和高级 API 参数。这两种类别的参数以不同的方式影响电容式感应系统的性能，因此，该部分分别介绍了各个参数。然而，每个传感器的灵敏度之间都有互补关系，由硬件参数的设置和若干高级 API 参数的设置决定。设计师必须确保当硬件参数改变时，相应的高级 API 参数也相应作出调整。调试 CSD 用户模块参数应始终从硬件参数开始。

图 10. CSD 用户模块参数调试流程图



硬件参数

硬件参数配置硬件，这些硬件中，使用 CSD 方法用来把每个传感器的物理电容值转换为数字代码。此部分描述这些参数，并为如何根据硬件特征和 / 或其他参数调试提供指导。

默认情况下，硬件参数是设计中应用到所有 CapSense 传感器的全局设置。在各设计中，如果传感器的寄生电容的总值 （C_p），和 / 或传感器的灵敏度在一个较大范围内浮动，那么全局硬件参数设置可能会不适合所有传感器。在这种情况下，通过使用 SetIdacValue()、 SetPrescaler() 和 SetScanMode() API 函数可以配置每个传感器的相应硬件参数。这时，这些函数应该与 CSD_ScanSensor() API 一起使用。

表 3 和表 4 提供了 C_p 为基准的几个关键硬件参数的推荐调试值。 C_p 的值是由 PSoC 的特征、 PCB 布局和产品组装板附近元件决定的。因上述原因， C_p 值必须在系统最终组装状态原位测量；即在与系统提供服务时一样，具有的同样的外围或覆盖层。测量 C_p 值的最佳方式是使用 www.cypress.com 网站上标题为 “使用 CY8C20xx6 CapSense 控制器测量传感器绝对电容 ” （EP64134）的示例项目。此项目使用 PSoC 测量系统中每个传感器的绝对电容值，会考虑所有影响 C_p 的因素。请参见示例项目相关的文件以查阅安装和使用步骤。

自动校准

该参数通过使能 iDAC 设置的逐次逼近来建立 85% 或 85% 以上的原始计数基准线。在 CY8C20xx7 CSD 设计中，应将自动校准始终设置为激活状态。自动校准算法能力能否成功设置 iDAC 取决于预分频器的合理设置和 C_{mod} 是否处于推荐大小的状态。默认设置为 Enable （使能）。

iDAC 值

当自动校准禁用后，此参数将决定当前 iDAC 输出。自动校准使能后，会使用推荐设定，该参数将被覆盖，并且失效。自动校准禁用后，提高该参数会降低原始计数基准线，反之亦然。默认值为 20。

iDAC 范围

通过该参数可以设置 iDAC 的输出范围。各选项为 4X 和 8X。对于少于 45 pF 的最大传感器 C_p 项目，可以使用 4X ；否则，则使用 8X。默认值为 4X。

预充源

此参数选择传感器开关时钟源。可用选项是预分频器，通过分频器或 PRS 使用 IMO，可使分割的 IMO 时钟通过任意发生器，提供扩频时钟。 PRS 提供高级降噪，释放更少的噪音，因而它也成为推荐预充源的默认设置。在有些情况下，预分频器预充源可提供更高的 SNR （信噪比）。然而，使用铜线路时，信噪比的提升通常比较细微，并且也不能明显看出是否益于前面提到的 PRS。默认值为 PRS。

预分频器

预分频器是应用于 IMO 以建立预充电时钟的分频器。合理调试 CSD 设计时，此为最重要的硬件用户模式参数。表 3 显示的是预分频器推荐设置。预分频器依据所选预充源、 IMO 和被扫描传感器的 C_p 值而定。表 3 显示了基于这些参数的预分频器推荐设置。默认值为 2。

表 3. 基于预充源、 IMO 和 C_p 的预分频器设置

Cp （pF）	预充源 = PRS			预充源 = 预分频器		
	预分频器 IMO=24MHz	预分频器 IMO=12MHz	预分频器 IMO=6MHz	预分频器 IMO=24MHz	预分频器 IMO=12MHz	预分频器 IMO=6MHz
<6	1	注解 1	注解 1	2	1	1
7 – 11	2	1	注解 1	4	2	1
12 – 15	2	1	注解 1	4	2	1
16 – 19	4	2	1	8	4	2
20 – 22	4	2	1	8	4	2
23 – 26	4	2	1	8	4	2
27 – 30	4	2	1	8	4	2
31 – 34	4	2	1	8	4	2
35 – 37	8	4	2	16	8	4
38 – 41	8	4	2	16	8	4
42 – 45	8	4	2	16	8	4
46 – 49	8	4	2	16	8	4
50 – 52	8	4	2	16	8	4
53 – 56	8	4	2	16	8	4
57 – 60	8	4	2	16	8	4

注意 1：不推荐使用这种预充源、预分频器和 C_p 的组合。

分辨率

可选范围为 9 到 16 位。提高分辨率会加强传感器的灵敏度，加大信噪比，并延长降噪的扫描时间。扫描分辨率为 n 时，最大原始计数值 （所有范围内）为 2^n-1。表 4 显示了基于 C_p 和手指电容值 C_f 的分辨率推荐设置。 C_f 是手指放置在传感器上时电容值的变化。 C_f 值取决于外覆层厚度、传感器大

小及传感器与其他大型导体的接近程度。图 11 显示了 C_f 值，它可作为外覆层厚度和圆形传感器直径的函数。默认值为 12。

图 11.　　基于外覆层厚度和圆形传感器直径的 （C_f）手指电容值

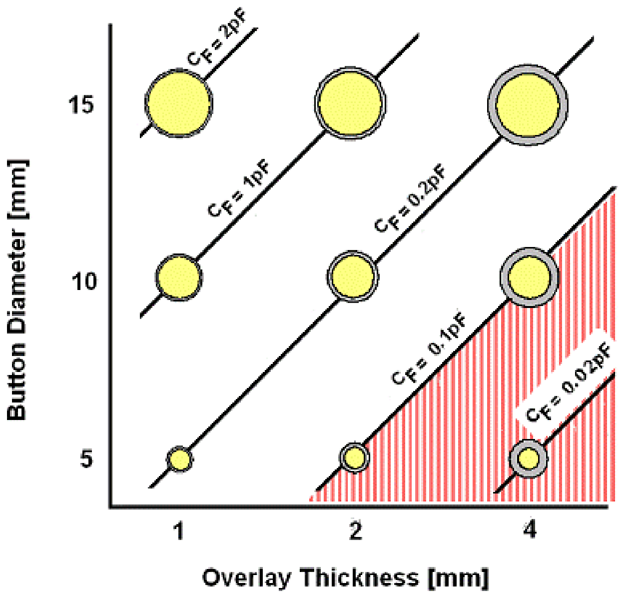


表 4.　　基于手指电容值和 Cp 的分辨率设置

Cp （pF）	C_f = 0.1pF	C_f = 0.2 pF	C_f = 0.4pF	C_f = 0.8pF
<6	12	11	10	9
7 - 12	13	12	11	10
13 - 24	14	13	12	11
25 - 48	15	14	13	12
>49	16	15	14	13

扫描速度

该参数控制各个扫描结果的 LSB 集成时间。扫描速度选项包括：超快、快速、正常和慢速。建议初始值选择 “ 快速 ” 。在某些情况下 （但非所有情况），较慢的扫描速度可以获得更高的信噪比，但扫描时间更长且功耗更大。表 5 显示了在不同分辨率和扫描速度下，单传感器的实际扫描时间 （单位为 ms）。默认设置为 Normal （正常）。

表 5. 单传感器在不同分辨率和扫描速度下的扫描时间 （ms）

分辨率 （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
13	380	720	1400	2800
14	720	1400	2800	5600
15	1400	2800	5600	11000
16	2800	5600	11000	22000

PRS 分辨率

此参数更改 PRS 序列长度。可能值为 8 和 12 位。相应的序列长度为 511 和 2047 输入时钟周期。如果需要极短的扫描时间，则必须使用 8 位 PRS 来避免过大噪声。扫描时间由分辨率 （不应该与 PRS 分辨率相互混淆）和扫描速度参数确定。如果扫描时间 ≤ 380 ms，则将 PRS 分辨率设置为 8 位 ；如果扫描时间 > 380 ms，则将 PRS 分辨率设置为 12 位。默认值为 8 位。

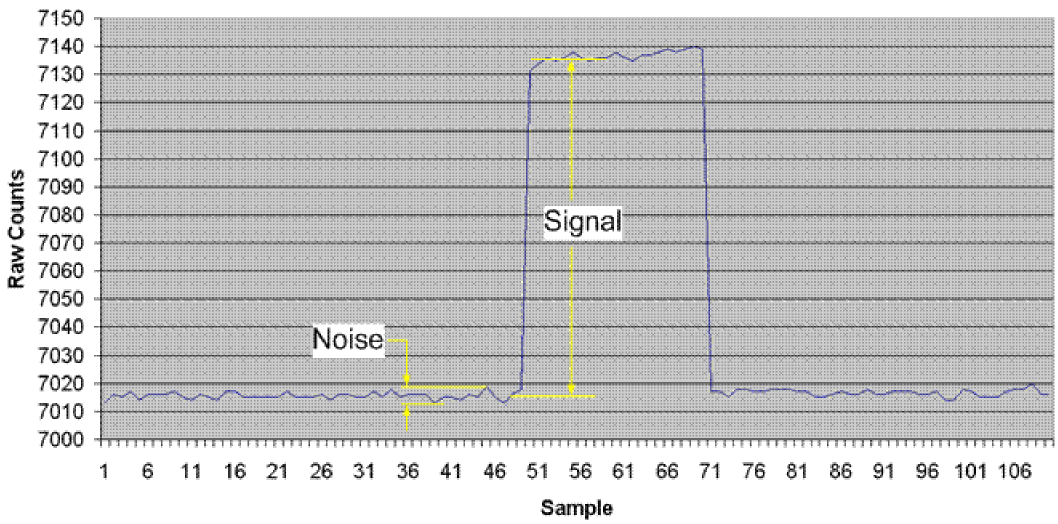
高层 API 参数

高层 API 参数决定高层固件算法的行为，此算法用于区分传感器的激活与噪声，并补偿由环境因素引起的信号漂移。为了确定适当的参数取值，须与系统建立起数字通信接口，以监控每个传感器手指激活事件期间的原始计数、基准线和差值。数据分别存储于如下三个数组：CSD_waSnsBaseline[]、CSD_waSnsResult[] 和 CSD_waSnsDiff[]。如该数据所示，高层 API 参数的设置主要基于环境噪声和手指信号强度。噪声与信号的强度取决于 EMI 环境、 PCB 布局、覆盖层厚度及其他系统特性。因此，该数据作为参数设置基础，须从系统的原始位置中获取，并且该系统须处于最终装配状态，并与未来实际应用时的 EMI 环境相同。

图 12 显示了传感器在一个手指激活周期中获取的典型原始计数值，即传感器被激活再取消激活。叠加在数据上的标签说明了如何根据原始数据计算噪声与信号。在适当情况下，下面的高层参数描述包括有关如何根据噪声和信号值进行设置每参数的信息。根据 CY8C20xx7/S CapSense 设计指南，为了 CapSence

系统操作的健壮性，信噪比 （SNR）应至少为 5:1。若信噪比低于 5:1，则需要根据 CapSense 入门调整硬件参数和 / 或更改 PCB 布局，以便将信噪比提高至 5:1。

图 12. 传感器在一个手指激活周期中的典型原始计数值



手指阈值

手指阈值确定原始计数的标称变化 （不包含迟滞），用于激活传感器。可能值的范围为从 5 到 255。默认情况下，通过 CSD_SetDefaultFingerThresholds() API 函数，全局手指阈值适用于所有传感器。对于独立按键 （滑条组中不包含），通过将所需要的值写入到 CSD_baBtnFThreshold[] 全局阵列中相应于传感器编号的索引位置内可以设置按键的手指阈值。为了设置该参数，必须分析每个传感器对手指激活的原始计数响应。当极限 （最小手指）触摸该传感器时，应该将每个传感器的手指阈值设置为原始计数信号的 75% （参见图 12）。对于作为滑条元素的传感器，手指阈值没有任何意义或功能。默认值为 60。

噪声阈值

如果原始计数的积极变化低于该等级，它将被累加，以用于更新原始计数基准线。值范围为 5 到 255。对于按键传感器，当传感器自动复位为禁用 （默认）时，超过该阈值的计数值将不更新基准线 ；对于滑条元素，坐标计算将不包含低于该阈值的计数值。噪声阈值是应用于所有传感器的全局参数。噪声阈值的良好起点是原始计数信号的 40% （请参考图 12）。默认值为 10。

基准线更新阈值

CSD 使用 “ 水桶 ” 方法来更新 CSD_UpdateSensorBaseline() API 函数中的基准线计数。在下列情况下，原始计数值和基准线计数值之间的半个差值被添加到 “ 水桶 ”：

1. 从扫描返回的原始计数值高于当前基准线 AND 值
2. 原始计数值和基准线之间的差值低于噪声阈值。

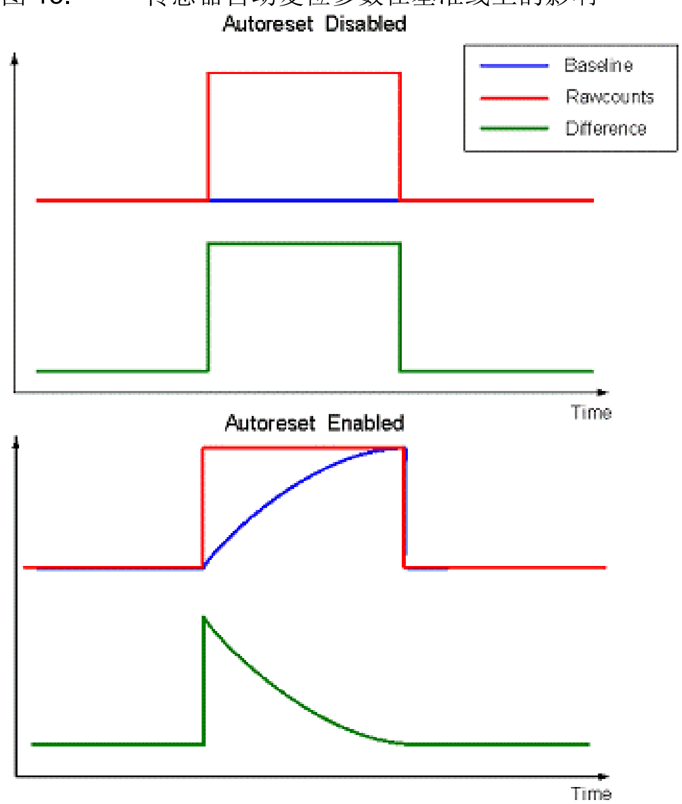
基准线更新阈值参数设置了基准线变化所需的 “ 水桶 ” 必须达到的值。该参数的初始值可以是噪声阈值的两倍。默认值为 100。

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。当此参数设置为“ 禁用 ”，则仅当原始计数与基准线的差值低于噪声阈值时，基准线才进行更新。图 13 说明了此参数对基准线更新的影响。当传感器自动复位为“Enabled”（使能）时，基准线始终被更新，无论噪声阈值如何。此设置限制传感器的最大激活时间（通常为 5 到 10s），但对防止传感器因非触摸引起的原始计数突然上升而被停滞带来了好处。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“Sensors Autoreset”（传感器自动复位）设置为禁用，则仅当原始计数与基准线之间的差值低于噪声阈值时，基准线才进行更新。应该将该参数设置为其默认的“Disabled”（禁用）状态。请参考该参数的其他说明附录。

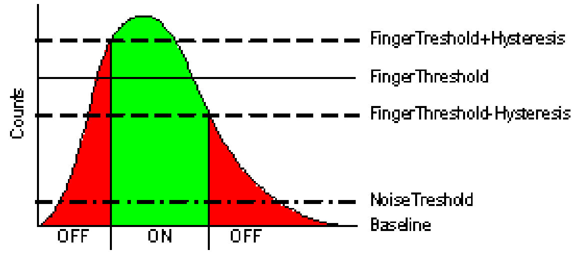
图 13.　　传感器自动复位参数在基准线上的影响



迟滞

为了提高按键传感器的激活状态识别并提供更稳定的操作，迟滞将传感激活状态依次设置为 OFF — ON — OFF，请参考图 14。计数值必须大于手指阈值 + 迟滞，以将状态从 OFF 修改为 ON。计数值必须小于手指阈值 - 迟滞，以将状态从 ON 修改为 OFF。迟滞的良好初始值是原始计数信号的 15%（参考图 11）。默认值为 10。

图 14.　　手指阈值和迟滞在按键传感器状态上的关系



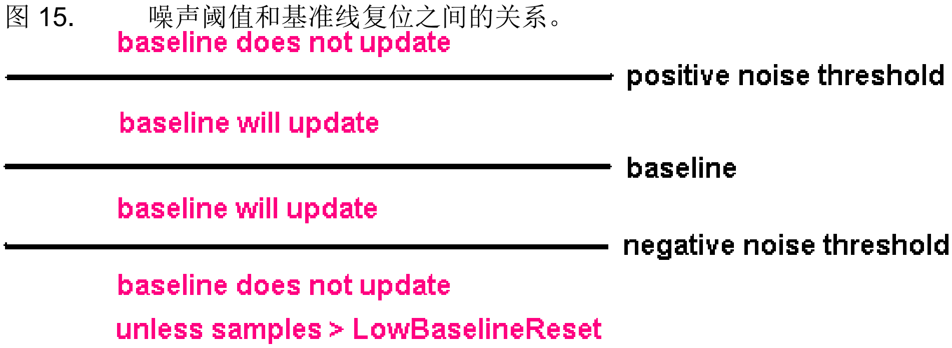
去抖动

该参数为传感器活动的瞬变增加了去抖动计数器。为了让传感器能够从未激活状态切换为激活状态，在该参数规定的样本数量内，差异计数值必须大于手指阈值与迟滞之和。去抖动计数器按 CSD_bIsSensorActive 或 CSD_bIsAnySensorActive API 函数递增。值范围为 1 到 255。如果设置为 ‘1’，则将没有去抖动但会提供最快的响应。默认设置为 3。

负噪声阈值

当原始计数低于基线的计数时，该参数被使用。它建立了一个相对于当前基准线的级别，基准线将复位，并基准线值将下降到原始计数值。如果原始计数低于该级别，基准线将不复位，除非达到低基准线复位参数的限制。在这种情况下，将复位基准线。图 15 显示的是噪声阈值和基准线复位之间的关系。负噪声阈值的良好起点是使用噪声阈值。默认值为 10。

图 15.　　噪声阈值和基准线复位之间的关系。



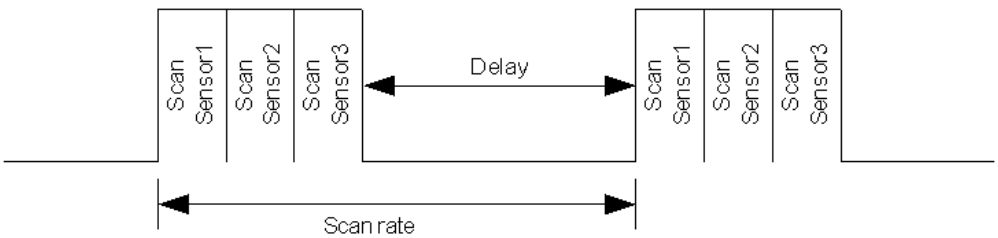
低基准线复位

该参数与负噪声阈值一起使用，以定义复位基准线到原始计数必须经过的采样次数，这种采样中的原始计数必须小于基准线值。对于由低基准线复位参数指定的采样数，如果原始计数值小于基准线值与负噪声阈值之差，则基准线值将下降到原始计数值。低基准线复位通常用来纠正启动时手指已经在传感器上面的情况。建议初始值选择 10；默认值为 50。

传感器扫描速率选择的指南

扫描速率是指传感器被扫描的速率。下图显示的是一个 3 按键设计的示例。设计中的所有传感器按顺序进行扫描，而且在各个扫描之间有一个延迟。

图 16.　　典型传感器扫描



为了确保基准线正常运行，推荐保持设计中的扫描速率至少为 15 ms。这意味着具有更少传感器的设计必须添加一个延迟，以使传感器扫描速率不低于 15 ms。具有更多传感器的设计不需要任何延迟，因为扫描所有传感器本身已消耗 15 ms 了。良好的设计会使 CapSense 控制器 （而不是使固件延迟子程序）进入睡眠模式，以节能。

应用编程接口

应用编程接口 （API）函数作为用户模块的一部分提供，使您能够更高级别处理该模块。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。

只能将此用户模式中的一个实例放置在项目中，并且它也应用于可加载的配置。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 将 CSD_1 分配到给定项目中此用户模块的第一个实例。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，则调用函数需要保留在调用前的 A 和 X 的值。此 “ 寄存器易失 ” 策略是针对提高效率的目的选择，并且从 PSoC Designer 的 1.0 版本起已强制使用。 C 编译器自动遵循该要求。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，但是无法保证它们在将来是否也被保留。

对于大型储存器模型器件，调用程序需要保留 CUR_PP、 IDX_PP、 MVR_PP 以及 MVW_PP 寄存器中的所有值。尽管一些寄存器现在不被修改，但是无法保证在将来的版本中也会如此。

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。不得手动更改这些阵列。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阵列的内容。下面显示的是几个全局阵列：

- 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。阵列大小与传感器数量相等。过下列函数更新 CSD_waSnsBaseline[] 阵列通：

- 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信号。阵列大小与传感器数量相等。通过下列函数更新 CSD_waSnsResult[] 数据：

- CSD_ScanSensor()；
- CSD_ScanAllSensors()。

CSD_waSnsDiff []：这是一个整数阵列，其中包含每个传感器中原始数据与基准数据之间的差值。阵列大小与传感器数量相等。

CSD_baSnsOnMask[]：这是一个字节阵列，用于保持传感器的开 / 关状态 （针对按键或滑条传感器）。CSD_baSnsOnMask[0] 包含传感器 0 到 7 的掩码位 （传感器 0 为位 0，传感器 1 为位 1）。CSD_baSnsOnMask[1] 包含传感器 8 到 15 的掩码位 （如果需要），以此类推。此字节阵列所包含的元素数量足以包含所有被放置的传感器。按键开启时位值为 1，关闭时位值为 0。 CSD_baSnsOnMask[] 数据由 CSD_bIsSensorActive(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。应当在调用任何其他用户模块函数之前先调用此函数。

C 原型：

```
void  CSD_Start(void)
```

汇编：

```
lcall CSD_Start
```

参数：

无

返回值：

无

其他影响：

**

CSD_Stop

说明：

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

C 原型：

```
void  CSD_Stop(void)
```

汇编：

```
lcall CSD_Stop
```

参数：

无

返回值：

无

其他影响：

**

CSD_Resume

说明：

调用 CSD_Stop 之后，恢复用户模块操作。

C 原型：

```
void  CSD_Resume(void)
```

汇编：

```
lcall CSD_Resume
```

参数：

无

返回值：

无

其他影响：

**

CSD_ScanSensor

说明：

扫描所选的传感器。每个传感器在传感器阵列中只有唯一一个编号。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

其他影响

**

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(void);
```

汇编：

```
lcall CSD_ScanAllSensors
```

参数：

无

返回值：

无

其他影响

**

CSD_UpdateSensorBaseline

说明：

针对每个传感器独立计算得出的历史计数值被称为这个传感器的基线。此基准线使用 “ 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线来计算差值计数。此差值存储在 CSD_waSnsDiff[] 阵列中，并是向您提供的。
2. 如果禁用了传感器自动复位 （Sensors Autoreset），则每次调用 CSD_UpdateSensorBaseline() 时，差值会与噪声阈值进行比较。如果差值低于噪声阈值，将被累加到虚拟的水桶中。如果差值高于噪声阈值，则不更新水桶。如果使能了传感器自动复位 （Sensors Autoreset），则无论噪声阈值参数如何，差值都会累加到虚拟的水桶中。
3. 虚拟水桶中的累计差值计数达到 BaselineUpdateThreshold 后，基准线会按 1 递增，且水桶将复位为 0。
4. 如果差值计数低于噪声阈值，则保留在 waSnsDiff[] 阵列中的值将复位为 0。因此，此阵列不包含数值大于 0 但低于噪声阈值的元素。

C 原型：

```
void  CSD_UpdateSensorBaseline(BYTE bSensorNum)
```

汇编：

```
mov   A, bSensorNum
lcall CSD_UpdateSensorBaseline
```

参数：

A => 传感器编号

返回值：

无

其他影响：

**

CSD_UpdateAllBaselines

说明：

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

C 原型：

```
void  CSD_UpdateAllBaselines(void)
```

汇编：

```
lcall CSD_UpdateAllBaselines
```

参数：

无

返回值：

无

其他影响：

**

CSD_bIsSensorActive

说明：

检查与手指阈值进行比较的给定传感器的差值计数阵列。将计算迟滞。根据传感器当前是否开启，对手指阈值增加或消减迟滞值。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 CSD_baSnsOnMask[] 阵列中传感器的位。

C 原型：

```
BYTE  CSD_bIsSensorActive(BYTE bSensorNum)
```

汇编：

```
mov   A, bSensorNum
lcall CSD_bIsSensorActive
```

参数：

bSensorNum A => 传感器编号

返回值：

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

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

其他影响：

**

CSD_bIsAnySensorActive

说明：

检查与手指阈值进行比较的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(void)
```

汇编：

```
lcall CSD_bIsAnySensorActive
```

参数：

无

返回值：

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

A => 1：表示一个或多个传感器处于活动状态， 0：表示没有任何传感器处于活动状态。

其他影响：

**

CSD_wGetCentroidPos

说明：

根据差值阵列计算坐标。如果存在，则偏移和长度存储为临时变量，并根据 CSD 向导中指定的分辨率计算坐标位置。仅在滑条是由 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。

其他影响：

此子程序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。如果应用监控差值信号，则在差值计数数据传输后调用此子程序。

如果某个滑条传感器处于活动状态，该函数会将数值从零恢复为 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，该函数将返回 –1 （FFFFh）。如果在执行坐标计算 / 复用算法时出现了错误，该函数将返回 –1 （FFFFh）。若需要，可以使用 CSD_bIsSensorActive() 子程序确定触摸了哪些滑条段。

注意：当滑条段的噪声计数大于噪声阈值时，此子例程可能会生成假的坐标结果。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坐标。

CSD_wGetRadialPos

说明：

根据差值阵列计算坐标。如果存在质位置，则根据 CSD 向导中指定的分辨率计算该坐标位置。此函数仅适用与 CSD 向导定义的径向滑条。

C 原型：

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

汇编：

```
mov   A, bSnsGroup
lcall CSD_wGetRadialPos
```

参数：

bSnsGroup A => 组编号

此参数是您使用的辐射状滑条的编号。可以通过 CSD 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：

辐射状滑条的位置值、 A 中的 LSB 和 X 中的 MSB。

其他影响：

此子程序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子程序。

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，则此子子程序可能生成假的坐标结果。设置噪声阈值时请务必小心 （应使之比噪声级别足够高），以便噪声不会产生假的坐标。

CSD_wGetRadialInc

说明：

返回实际手指移位情况，即手指的当前位置与先前位置之间的差值。此函数与 CSD_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  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 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。

手指移位值是手指的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() 将返回 -1 （FFFFh）；如果当前没有任何触摸，则此时 CSD_wGetRadialPos() 会返回 -1 （FFFFh）

其他影响：

仅在调用 CSD_wGetRadialPos() API 之后，才能调用此子程序。因为它使用由 CSD_wGetRadialPos() 设置的 CSD_waSliderPrevPos 和 CSD_waSliderCurrPos 内部数据。

CSD_InitializeSensorBaseline

说明：

通过扫描所选的传感器，加载含初始值的 CSD_waSnsBaseline[bSensorNum] 阵列元素。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  CSD_InitializeSensorBaseline(BYTE bSensorNum)
```

汇编：

```
mov   A, bSensorNum
lcall CSD_InitializeSensorBaseline
```

参数：

A => 传感器编号

返回值：

无

其他影响：

**

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线阵列中。

C 原型：

```
void  CSD_InitializeBaselines(void)
```

汇编：

```
lcall CSD_InitializeBaselines
```

参数：

无

返回值：

无

其他影响：

**

CSD_SetDefaultFingerThresholds

说明：

通过 FingerThreshold （手指阈值）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(void)
```

汇编：

```
lcall CSD_SetDefaultFingerThresholds
```

参数：

无

返回值：

无

其他影响：

**

CSD_SetScanMode

说明：

设置扫描速度和分辨率。可以在运行时调用此函数来更改扫描速度和分辨率。此函数会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：常用按键和接近检测器），此函数非常有效。可以用 9 位分辨率扫描常规按键。接近检测器经常可以采用比 16 位略低的分辨率进行扫描，对于大范围检测，扫描时间较长。此函数可与 CSD_ScanSensor() 函数结合使用。

C 原型：

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

汇编：

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

参数：

bSpeed：扫描速度

下面给出了 bSpeed 参数的常量：

常量	数值
CSD_ULTRA_FAST_SPEED	0x00
CSD_FAST_SPEED	0x01
CSD_NORMAL_SPEED	0x02
CSD_SLOW_SPEED	0x03

bResolution：扫描分辨率。将此值设置为所需的分辨率位数。此参数值不得小于 9 或大于 16。

提供 bResolution 参数的以下可能常量：

常量	数值
CSD_9_BIT_RESOLUTION	9
CSD_10_BIT_RESOLUTION	10
CSD_11_BIT_RESOLUTION	11
CSD_12_BIT_RESOLUTION	12
CSD_13_BIT_RESOLUTION	13
CSD_14_BIT_RESOLUTION	14
CSD_15_BIT_RESOLUTION	15
CSD_16_BIT_RESOLUTION	16

返回值：

无

其他影响：

**

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

其他影响：

**

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。此函数可与 CSD_ScanSensor() 函数结合使用。

C 原型：

```
void  CSD_SetIdacValue(BYTE bIdacValue)
```

汇编：

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

参数：

bIdacValue — 设置 iDAC 值。合适的值为 1..255。

返回值：

无

其他影响：

**

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。此函数可与 CSD_ScanSensor() 函数结合使用。

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。下表列出了合适的值：

名称	数值	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
CSD_PRESCALER_32	0x05	32
CSD_PRESCALER_64	0x06	64
CSD_PRESCALER_128	0x07	128
CSD_PRESCALER_256	0x08	256

返回值：

无

其他影响：

**

CSD_CalibrateSensors

说明：

调整 iDAC 电流以获取接近 wValue 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 wValue)
```

汇编：

```
lcall CSD_CalibrateSensors
```

参数：

wValue — 目标原始数据值。

返回值：

无

其他影响：

**

CSD_ClearSensors

说明：

通过按顺序为每个传感器调用 CSD_wGetPortPin() 和 CSD_DisableSensor()，将所有的传感器清除到非采样状态。

C 原型：

```
void  CSD_ClearSensors(void)
```

汇编：

```
lcall CSD_ClearSensors
```

参数：

无

返回值：

无

其他影响：

**

CSD_wReadSensor

说明：

返回 A 和 X 中的关键原始扫描值 （分别为 LSB 和 MSB）。

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。

其他影响：

**

CSD_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 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 bSensor)
```

汇编：

```
mov   A, bSensor
lcall CSD_wGetPortPin
```

参数：

bSensor — 范围为 0 到 （n – 1），其中 ‘n’ 是 CSD 向导中设置的传感器数量与滑条中包括的传感器数量之和。 CSD_wGetPortPin() 使用传感器编号来确定所选的活动传感器的端口和位掩码。

返回值：

A => 传感器位图

X => 端口编号

其他影响：

**

CSD_EnableSensor

说明：

配置所选的传感器，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，端口编号和传感器位掩码分别加载到 X 和 A 中。修改驱动模式，以便将所选的端口和引脚置于模拟高祖模式并使能正确的模拟复用器总线输入。这样同样也可以使能比较器的功能。

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 => 端口编号

返回值：

无

其他影响：

**

CSD_DisableSensor

说明：

禁用由 CSD_wGetPortPin() 函数选择的传感器。驱动模式更改为“Strong”（强驱动）（即 001）。这样可以将传感器有效地接地。端口引脚与“模拟复用器总线”（AnalogMuxBus）的连接关闭。函数参数由 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 => 端口编号

返回值：

无

其他影响：

**

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. 下面的代码演示了两个传感器在用户模块向导中配置时的一个传感器用途示例。

```
//--------------------------------------------------------------------------
// Sample C code for the CSD module
//--------------------------------------------------------------------------

#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(); // Start CSD UM
    CSD_SetDefaultFingerThresholds(); // Set default thresholds for buttons

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

    while (1)
    {
        // Scan continuously sensor number "3" which is connected
CSD_ScanSensor(3);
        CSD_UpdateSensorBaseline(3); // Update Baseline for sensor 3
        if(CSD_bIsSensorActive(3)) // check if sensor 3 is touched
        {
              // 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。当需要执行按键触摸检测和接近检测时非常有用。扫描按键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，只有当不进行按键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. 下面的示例演示了如何为每个传感器设置不同的手指阈值的级别。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

表 6. 模块 CapSense、寄存器：CS_CR0

位	7	6	5	4	3	2	1	0
数值	0	0	CSD_PRS CLK	0	1	0	0	EN

表 7. 模块 CapSense、寄存器：CS_CR1

位	7	6	5	4	3	2	1	0
数值	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。

表 8. 模块 CapSense、寄存器：CS_CR2

位	7	6	5	4	3	2	1	0
数值	1	0	0	0	0	1	0	0

表 9. 模块 CapSense、寄存器：CS_CR3

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

表 10. 模块 CapSense、寄存器：CS_CNTH

位	7	6	5	4	3	2	1	0
数据输出 MSB								

表 11. 模块 CapSense、寄存器：CS_CNTL

位	7	6	5	4	3	2	1	0
数据输出 LSB								

表 12. 模块 CapSense、寄存器：PRS_CR

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

表 13. 模块定时器、寄存器：PT1_CFG

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

表 14. 模块定时器、寄存器：PT1_DATA0

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

表 15. 模块定时器、寄存器：PT1_DATA0

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

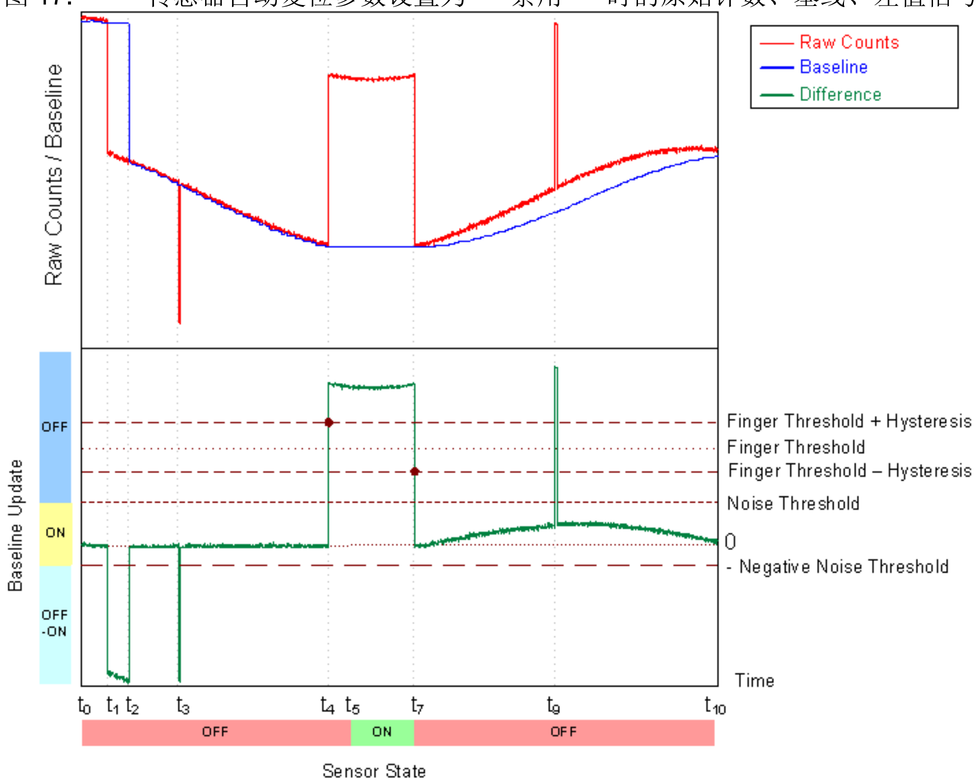
附录

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，帮助您成功设计 CapSense 应用。此信息的某些部分将来会移到应用笔记中。

CSD 参数的交互

图 16 和图 17 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。图 16 说明了当传感器自动复位参数设置为**禁用**时的系统操作。图 17 显示的是**使能**的传感器自动复位参数。图中还一同显示了手指阈值、噪声阈值、迟滞、负噪声阈值和差值信号 （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。

图 17. 传感器自动复位参数设置为 “ 禁用 ” 时的原始计数、基线、差值信号变化的示例



在 t_0 处，由于湿度或温度变化，接近于基线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 “ 负噪声阈值 ” （NegativeNoiseThreshold）参数 （绝对值），因此通过跟踪原始计数的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 （负噪声阈值）。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 LowBaselineReset （低基准线复位）样品的 NegativeNoiseThreshold （负噪声阈值）时，则基准线复位。这是在 t_2 处发生的。

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，内部去抖动计数器会激活。如果信号超过此值的量大于去抖动样品，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。由于采样单元中的尖峰脉冲持续时间不超过去抖动值， t_9 处的瞬时正值尖峰脉冲由去抖动计数器筛选。

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 （传感器自动复位设置为 “ 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，基准线更新速度越快。

图 18. 在传感器自动复位设置为 “ 使能 ” 的情况下原始计数、基准线、差信号的变化示例

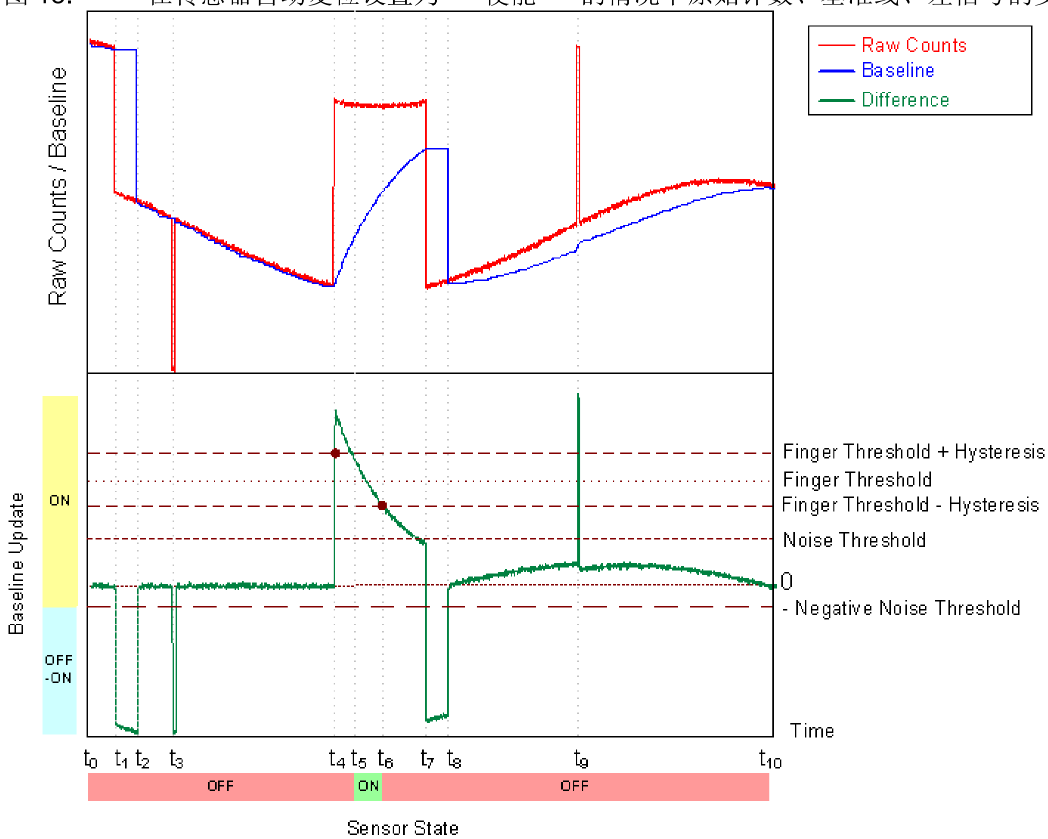


图 17 中的系统操作与上一实例中的操作类似，但有以下区别：

- 在 t_6 处触摸传感器时，由于活动基准线更新算法，触摸持续时间会减少。
- 手指移开后，基准线会在 LowBaselineReset （低基准线复位）采样 （t_8）后复位，这会短时阻止触摸检测。这可作为附加的去抖动机制。

版本历史记录

版本	创作者	说明
1.4	DHA	1. 最大分辨率为（传感器使用的引脚数量 – 1）x 2^8 – 1；对于复用滑条，此值为（2 x 传感器使用的引脚数量 – 1）x 2^8 – 1。消除了 0.5 的移位，并添加了对负值的补偿。 2. 修正了向导中的引脚列表。
1.50	DHA	增加了对 CY8C21x12 器件的支持。
1.50.b	DHA	1. 更改了滑条传感器的最大分辨率。 2. 在向导中添加了帮助文件。 3. 本用户模块数据手册中已更新了下面内容： a. 添加了 CY8C28xxx 的模拟总线的说明内容。 b. 更新了各图像。
1.60	DHA	1. 将‘DiplexTable’（复用表）从“AREA UserModules”传输到了“AREA lit”。 2. 将默认的“DiplexTable”参数值设置为 0x0112。 3. 添加了“DiplexUsed”参数，以提高代码的压缩能力。 4. 添加了“在启动 API 结束后调用 CSD_ScanAllSensors API”的内容。

版本	创作者	说明
1.70	DHA	1. 更新了区域声明以支持 Imagecraft 优化。 2. 为此用户模块数据手册中的分辨率参数添加了符号名。 3. 解决了与 CSD 和 DelSig 用户模块共存的问题。 4. 添加了预充电功能，这样能够将 Cmod 电容的电压预充电到参考电压。 5. 添加了对 CY8C21x34 器件上 Rb 引脚 P1[0]、 P1[4] 和 P3[0] 的支持。 6. 添加了设置错误反馈电阻参数时需要的设计准则检查。 7. 添加了对 CY8C24x94 上 Rb 引脚 P1[0]、 P3[0]、 P5[0]、 P1[4]、 P3[4] 和 P5[4] 的支持。 8. 添加了滑条和辐射滑条分辨率参数的上限值。 9. 在本用户模块数据手册中更新了下面部分： SetScanMode() API 函数的说明内容 电阻引脚部分的反馈 ISSP 引脚可能发生的冲突部分 Rb 引脚参考 10. 更新了用户模块向导中的滑条和辐射滑条分辨率范围内计算。 11. 更新了用户模块向导帮助。添加了一个滑条分辨率参数的最小 / 最大值说明。
1.70.b	DHA	1. 针对 PRS8 和带预分频器的配置，将频率峰值从 FIMO/4 更改为 FIMO/2。 2. 仅针对 CY8C20xx7/S 器件将 CSD_MODE 位的设置从 ScanSensor API 转移到 Start API。 3. 仅针对 CY8C20xx7/S 器件，在 CSD_Start API 中将校准分辨率从 9 位改为 12 位。
1.80	DHA	1. 将默认的 “Reference” （参考）值从 VBG 更改为 ASE11。 2. 更新了用户模块框图。 3. 本用户模块数据手册中更新了 RAM 和 ROM 的用途。 4. 消除了冗余的寄存器写入，并纠正了 PRS16 配置的屏蔽信号连接。 5. 添加了对 CYRF89x35 器件的支持。 6. 移除了 CY8C24x94 芯片架构的冗余比较器总线的使用情况。 7. 未放置 CSD 时，可以释放模拟复用器资源。

版本	创作者	说明
1.90	MYKZ	1. 添加了用户模块 API 的 Resume() 函数。 2. 纠正了有关保存滑条信息的问题。 3. 更新了基线算法，用于检查负差值计数。 4. 添加了当用户尝试编译项目前未先调用用户模块向导时的编译错误信息。 5. 更新了用户向导滑条的设置算法，以使用自由引脚。 6. 优化了启动用户模块的功能代码。 7. 删除了反馈电阻引脚的默认值。 8. 更新了 Precharge() 函数，以纠正 Cmod 与 GND 的连接。 9. 更新了 ScanSensor() 函数，以复位 PRS。

注意： PSoC Designer 版本 5.1 在所有用户模块数据手册中都介绍了 “ 版本历史记录 ”。本数据手册详细介绍了当前和先前用户模块版本之间的区别。