SmartSense_EMCPlus 数据手册 SmartSense_EMCplus V 1.20

Copyright © 2012-2014 Cypress Semiconductor Corporation. All Rights Reserved.

资源	PSoC® 模块				API 存储器		外部 I/O
	CapSense®	I²C/SPI	定时器	比较器	闪存	RAM	
CY8C20xx7S, CY8C20055							
用户模块	1	–	1	1	3494	85	1
滑条 API	–	–	–	–	696	92	0
每个传感器	–	–	–	–	6	31	1

特性和概述

- 对 PSoC® 器件的 CY8C20xx7S 系列器件实现 CapSense 电容式感应功能
- 运行时，会根据传感器、 IC 和 PCB 的特性自动调校可配置的系统参数
- 支持高达 35 个电容式传感器和 6 个滑条
- 支持寄生电容在 5 pF 到 45 pF 的传感器
- 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触摸。
- 对交流电源噪声、其他 EMI 和供电电源的抗干扰能力较强。
- 支持将电容式传感器配置为独立按键和作为相关阵列以造成滑条。
- 提供了双工特性：可为每个专用的 I/O 引脚分配两个线性滑条元素
- 支持通过插值使滑条的分辨率高于物理间距。
- 利用屏蔽电极能够在存在较高的寄生电容和有水膜的情况下进行可靠的操作。
- 通过操作 SmartSense_EMC 向导，可以使能指导传感器和引脚分配。
- SmartSense 电磁兼容性 （SmartSense_EMCplus）对外部噪声的抗噪能力非常出色。
- CY8C20055 系列不支持滑条

注意： 该用户模块仅支持 C 语言工程，并不支持汇编语言 （ASM）工程。

快速入门

1. 选择并放置需要专用引脚 （例如 I^2C 和 LCD）的用户模块。根据需要分配端口和引脚。
2. 选择并放置 SmartSense_EMCplus 用户模块。
3. 在 Workspace Explorer （工作区浏览器）中，右击 SmartSense_EMCplus 用户模块，这样便能够访问 SmartSense_EMCplus 导向 （请参阅 SmartSense_EMCplus 导向部分）。
4. 设置传感器、滑条和旋转滑条的所需数量。
5. 对于滑条，输入特定于滑条的参数。
6. 将每个传感器分配到一个未用的引脚。
7. 指定与外部调制电容相连的引脚。
8. 在 Workspace Explorer 中，右击 SmartSense_EMCplus 用户模块，以访问属性列表。如果需要，请指定用于保护传感器的引脚 （请参阅 ‘ 参数和资源 ’ 部分）。
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

SmartSense_EMCplus 用户模块实现了 CapSense 电容感应。 CapSense 是一种用于检测人体电容的人机界面技术。它使用一个带有导电界面 （通常是蚀刻在 PCB 表面上的焊盘）的传感器进行检测。由于 CapSense 会检测人体电容，所以它能够检测诸如塑料或玻璃外覆层等绝缘层。这些外覆层通常构成器件的外壳。这些特性使 CapSense 成为按键和电位器等机械输入器件的完美替代选择。

- 设计平整光滑、简洁美观
- 更小外形非常适合装备在终端产品内。
- 新加的高级用户界面特性，如 LED 效果和接近传感。
- 提高各组件的可靠性，因为各组件不磨损或具有有限的生命周期。
- 提高了防泼溅性能，因为表面没有机械破坏，
- 降低加工成本因为不需要机械输入器件的穿透或其他机械工作。

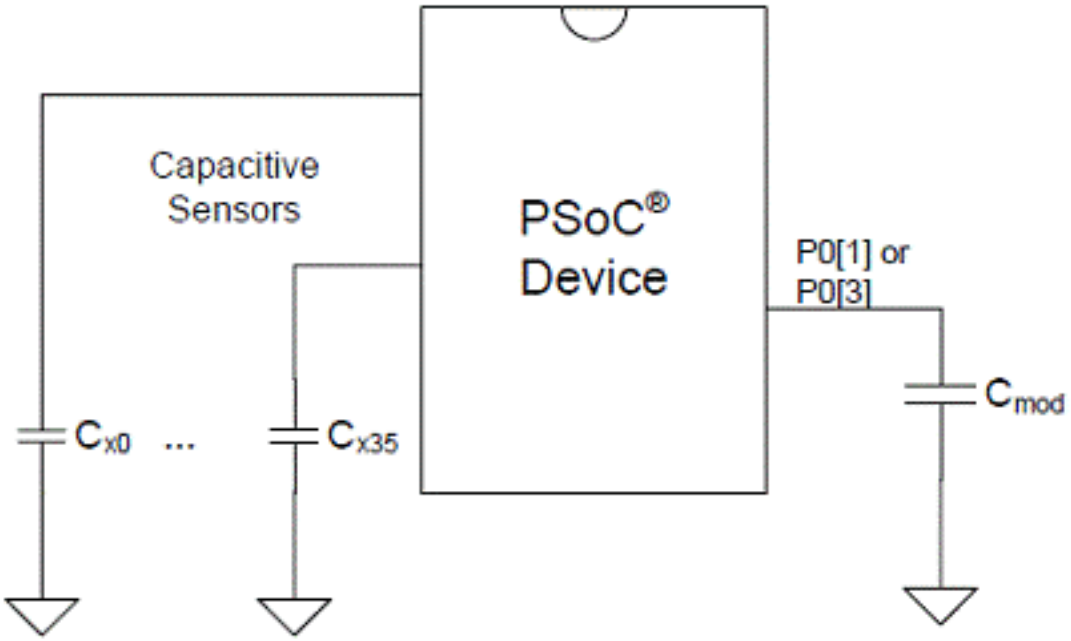
与所有其他 CapSense 解决方案相同， SmartSense_EMCplus 提供了卓越的抗传导和辐射噪声能力，但不同的是，它还提供了自动调校功能。运行时，自动调校会对 IC 和 PCB 特性及环境变化进行补偿，从而确保传感器能够正常可靠地工作。例如，样板和生产的 PCB 经常会表现出不同材料的属性，这样会影响传感器的寄生电容。若发生这类现象，需要重新调校 CapSense 系统参数。自动调校可对此类的变化进行补偿，因此不需要进行重新调校。此外， SmartSense_EMCplus 内的自动调校算法还会连续监控传感器数据，以便对环境条件变化进行补偿，如温度和环境噪声水平，从而能够确保传感器正常工作。

SmartSense_EMCplus 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 SmartSense_EMCplus 用户模块的原理图。物理传感器通常是一个 PCB 构成的传导模式，该 PCB 通过绝缘覆盖层与 PSoC I/O 引脚相连。

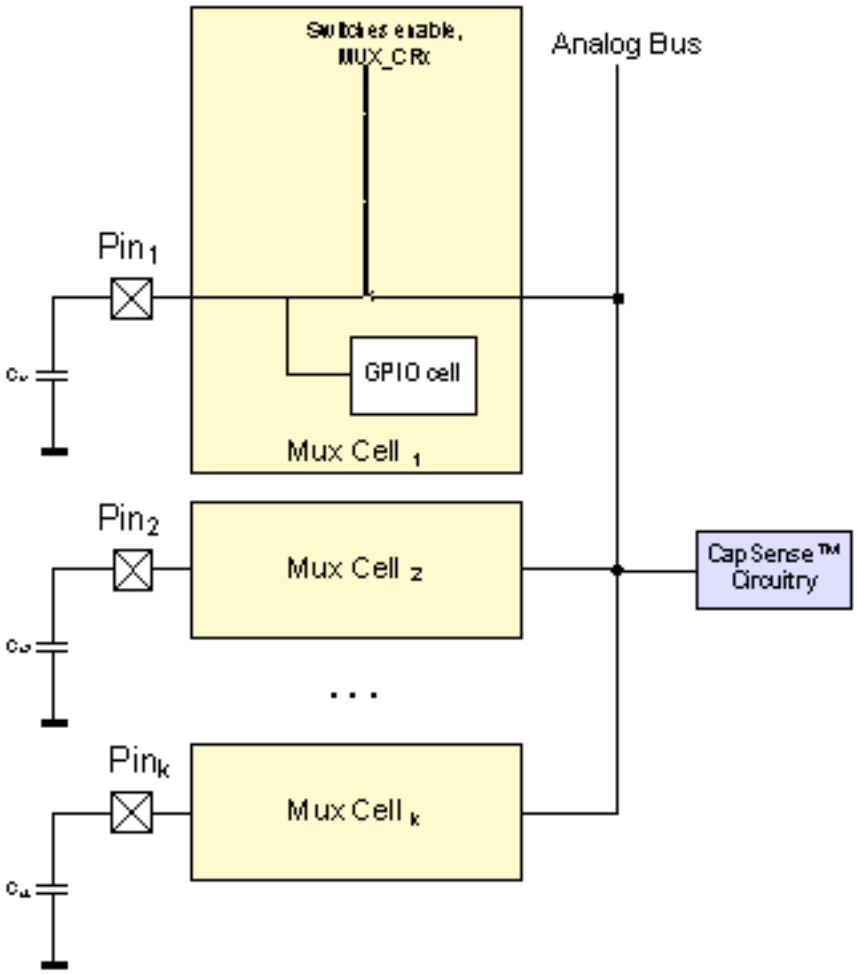
图 1. SmartSense_EMCplus 原理图



IC 等级

通过 CY8C20xx6L 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。SmartSense2X_EMC 用户模块可将活动传感器连接到 AMUX 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，固件可以连续扫描各个传感器。

图 2. CY8C20xx7S AMUX 框图



软件

SmartSense_EMCplus 软件组件具有以下各项特性：

- 在运行时自动调校算法会对模拟电容感应电路进行配置，以获得最佳性能。这些算法结合了物理传感器的特性、 IC 特性和用户模块的传感器灵敏度参数。
- 运行时， API 函数对电容转换电路中的原始计数值进行分析，以确定传感器状态并补偿环境变化。
- 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。
- 高级软件功能能够适配滑条双工法，因此一个 I/O 引脚能被路由到两个物理传感器。因此，特定数量的滑条元件所消耗的 I/O 数减去一半。

推荐阅读

赛普拉斯推荐在使用 SmartSense_EMCplus 用户模块实现 CapSense 设计前，应该先阅读下面文档。这些文档可在赛普拉斯网站 www.cypress.com 上获取：

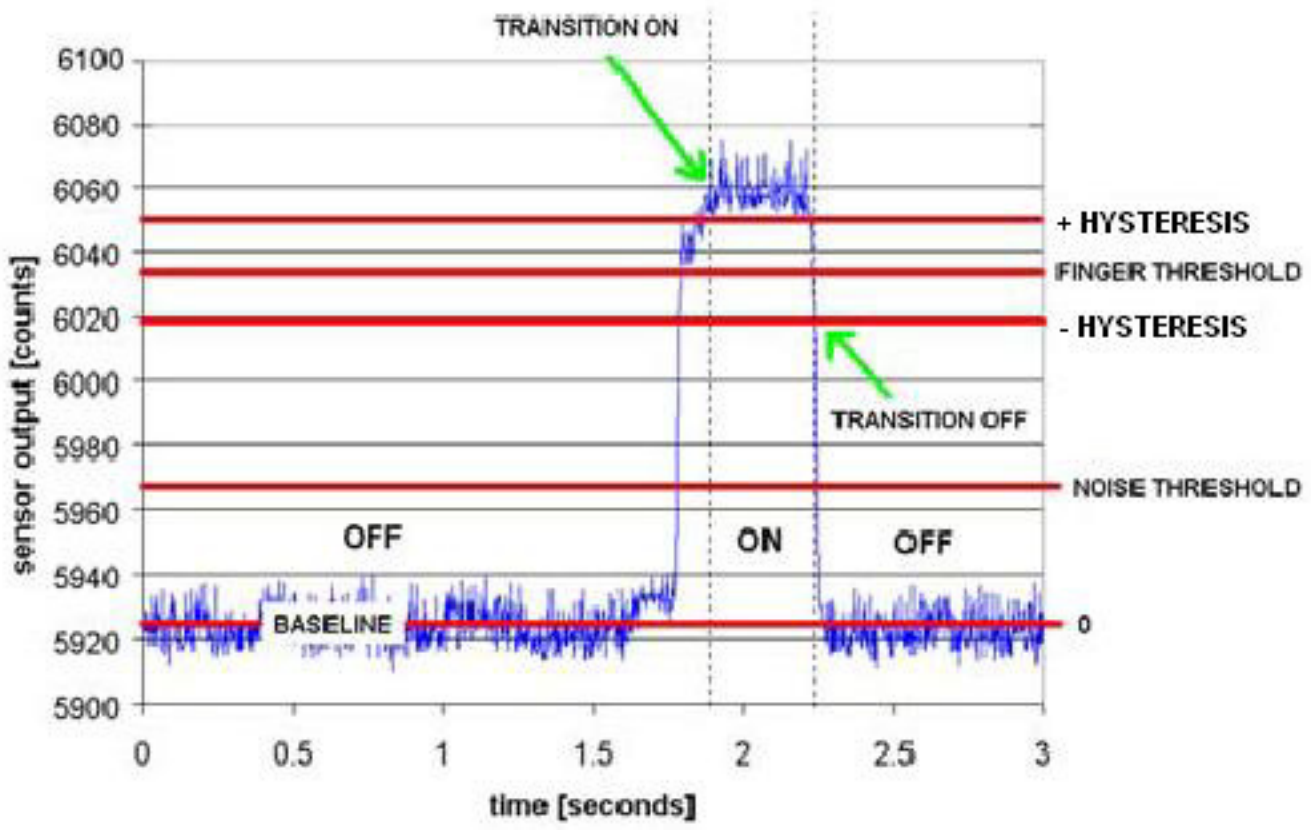
- *《系列 PSoC 混合信号阵列技术参考手册》中的相关章节：CapSense 系统*
- CY8C20xx7 器件数据手册
- Capsense 入门手册

电容式感应的实现

按键

类似于机械式按键， CapSense 按键可用于离散控制，如打开 / 关闭开关、功能键、菜单键等。API 函数监控着来自每个传感器的电容信号，并将这些值和阈值级别进行对比 （该阈值是通过 SmartSense_EMCplus 自动调校算法计算得到的）。当触摸到某个传感器时，它的电容信号会增高。如果 SmartSense_EMCplus 决策逻辑确定该增量足够大，那么将激活传感器。图 3 显示的是传感器活动时的一个典型信号 （蓝线）。 SmartSense_EMCplus 根据用户输入的自动设置阈值 （红线），以提供系统所需的操作。此外，可以访问手动模式，从而能够设置每个传感器的手指阈值。

图 3. 传感器活动时的电容信号

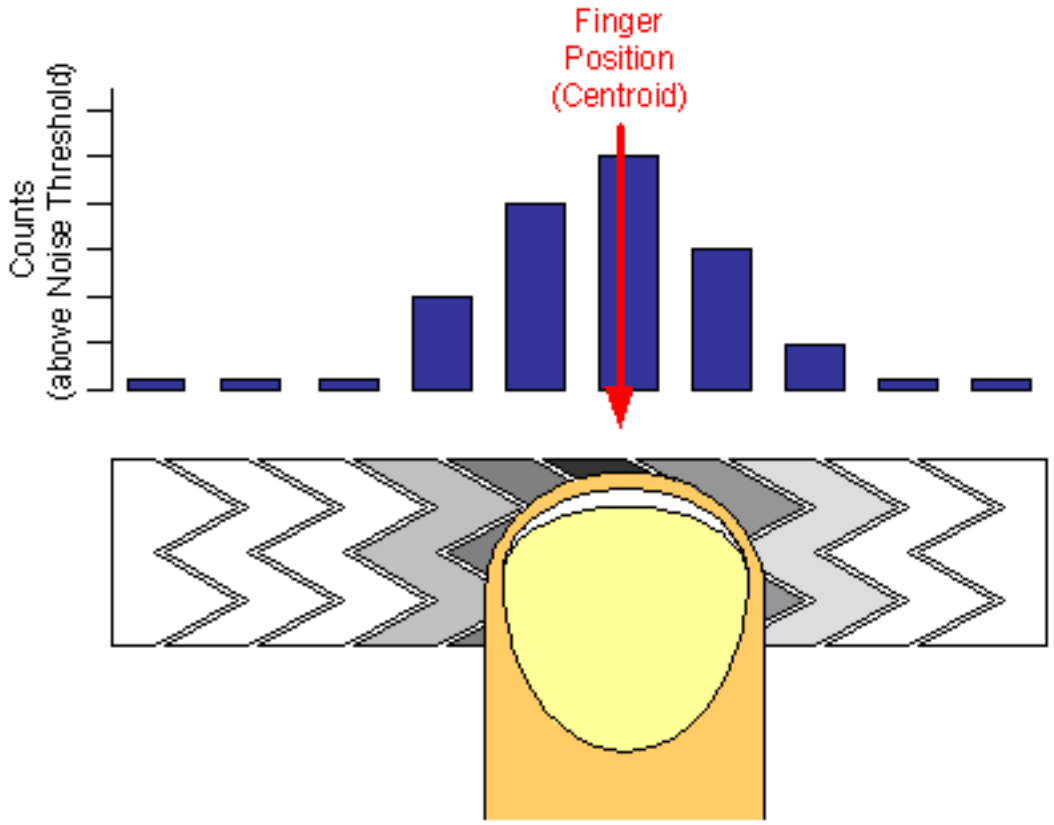


滑条

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。当手指启动滑条时，一些

相邻的传感器会寄存电容信号的增加值。通过计算活动传感器组的中心位置，可以确定触摸的实际位置。滑条中的传感器实际最小数量为五，最大值仅受限于 PSoC 器件上可用的 I/O 引脚数量。

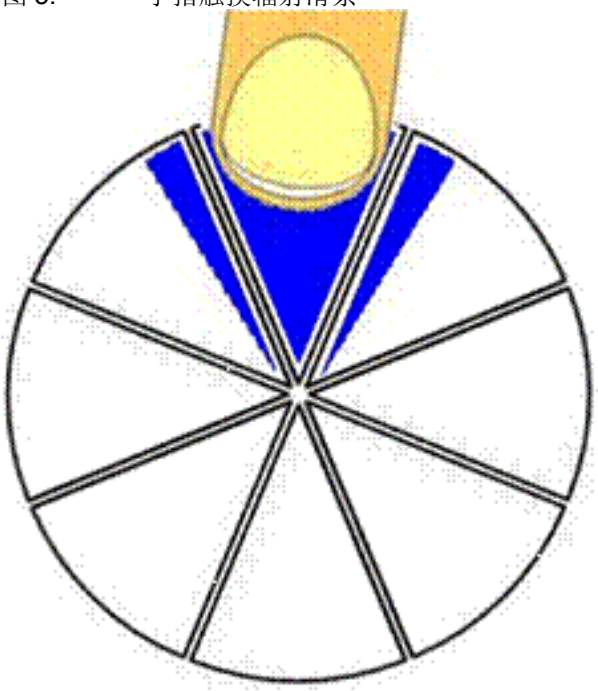
图 4. 滑条上手指的插值中心位置



Radial Slider （辐射滑条）

SmartSense_EMCplus 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 所示）却没有。在任何一种情况下，如果发生了触摸，中心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），以内插触摸的正确位置。辐射滑条未采用双工法。对于辐射滑条，SmartSense_EMCplus 用户模块带上两个 API 函数：第一个函数 SmartSense_EMCplus_wGetRadiaPos() 返回中心位置，第二个函数 SmartSense_EMCplus_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， SmartSense_EMCplus_wGetRadialInc() 会返回一个正偏移。参考点 （0）位于第一个传感器的中心。分辨率受限于 (传感器使用的引脚数量 – 1) x 2^8 – 1 ；对于双工滑条，此值为 (2 x 传感器使用的引脚数量 – 1) x 2^8 – 1。

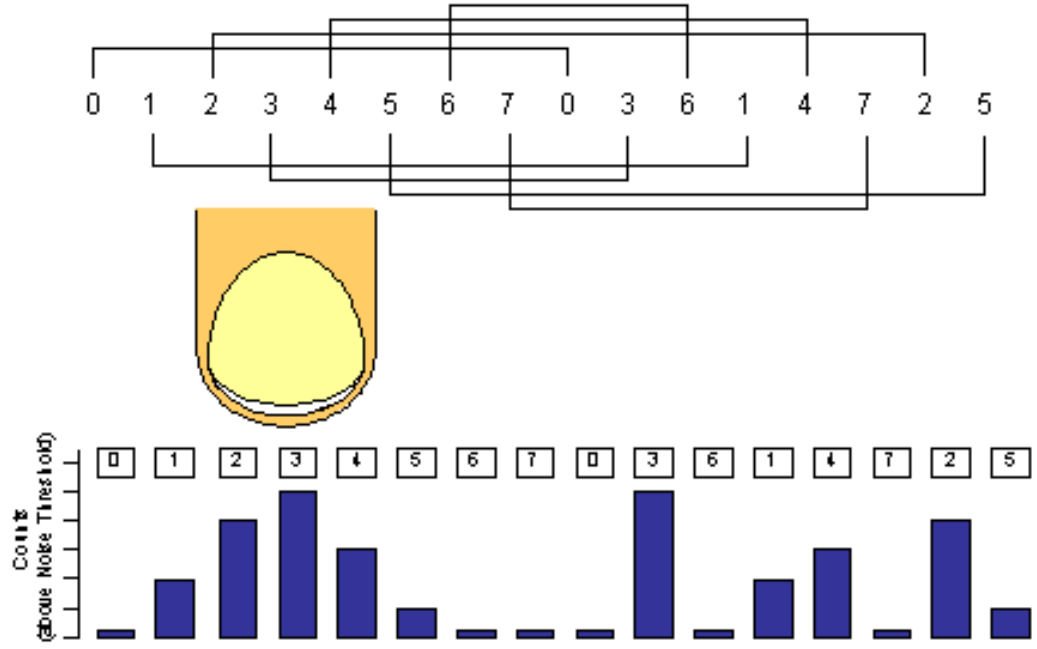
图 5. 手指触摸辐射滑条



双工

使用双工法时，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 （或下）（数字上较低的）部分根据 SmartSense_EMCplus 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 SmartSense_EMCplus 进行的双工滑条阵列索引编制



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。中心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 中所示。

必须确保传感器到印制电路板 （PCB）上各引脚的映射情况符合双工算法的 ‘ 按 3 编制索引 ’ 序列。双工滑条中传感器对的电容必须合理匹配 （不能超过 10 pF）。当您选择双工法时， SmartSense_EMCplus 向导会自动生成双工传感器索引表。表 1 显示的是双工成 28 个 PSoC I/O 引脚的 56 个滑条段的双工序列。

表 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

滑条段总计数	段序列
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

双工滑条的滑条段选型指南

选择滑条所需的段数主要取决于滑条的物理长度。然而，确定双工滑条的段数时必须特别小心。在双工滑条设计中，一个传感器将作为两个物理滑条段使用，以增加滑条的长度。手指触摸完全覆盖的段数必须小于从同一传感器派生出的两段之间的传感器数量。这样可确保双工滑条的正常工作。例如，在 10 段滑条（5 个传感器）的情况下，从传感器 3 中派生出的两个滑条段仅由两个传感器 （传感器 4 和 0）分隔。此时，手指触摸不得完全覆盖两个以上的传感器段，以确保滑条能正常工作。对于一个 12 段滑条，一个手指触摸不得覆盖 3 个段以上。同样，对于一个 18 段滑条，一个手指触摸不得完全覆盖 4 个段以上。

内插法和定标

在滑条应用中，确定带有比传感器的物理间距精度更高的手指位置。通过使用对传感器信号 （该传感器相邻有最大信号的传感器）的中心计算进行内插手指位置， SmartSense_EMCplus 可确定该位置。首先扫描阵列，以验证信号模型是否有效。合格要求提供一定数量的相邻传感器信号，且这些信号要高于噪声阈值。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中心。使用少至两个、多至（通常）八个传感器，通过公式 1 计算中心：

公式 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。为了指出在特定分辨率下的中心位置 （例如对于 12 个传感器为 0 到 100 的范围），要将中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定标的方法统一到一个计算中，按所需的量级直接报告结果。该过程可以在高级 API 内进行处理。

滑条传感器数量和分辨率在 SmartSense_EMCplus 向导中进行设置。比例值由向导计算出，并以分数值的形式进行存储。中心分辨率的乘数占用三个字节，相应的定义如表 2 所示：

表 2. 滑条的中心乘数位定义

位	7	6	5	4	3	2	1	0
分辨率乘数最高有效位								
乘数	2^{15}	21^{4}	21^{3}	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 = (传感器数量 - 1) x 乘数

中心以 24 位无符号的整数保存，其分辨率是传感器数量和乘数的函数。

外部组件选择 （C_{mod}）

SmartSense_EMCplus 需要一个外部调制电容 C_{mod}，该电容从 V_{SS} 连接到专用 PSoC 引脚 P0[1] 或 P0[3]。可以在 SmartSense_EMCplus 向导中的 **Global Settings > Modulator Capacitor Pin** 下进行 C_{mod} 引脚分配。所选的引脚不得用于其他任何用途。您必须使用陶瓷电容以及外部调制电容的建议值为 2.2 nF。温度电容系数并不重要。赛普拉斯推荐在所有 CapSense 传感器走线上安装大小为 560 Ω 的串联电阻，以抑制 RF 的干扰。必须将该电阻放置在离 PSoC 器件最近的位置。

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。这样的目的是为了提高传感器灵敏度，并在覆盖层上存在水滴时会防止误触发传感器。

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。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。这样能防止错误地激活。

图 7.　　驱动屏蔽电极 PCB 布局

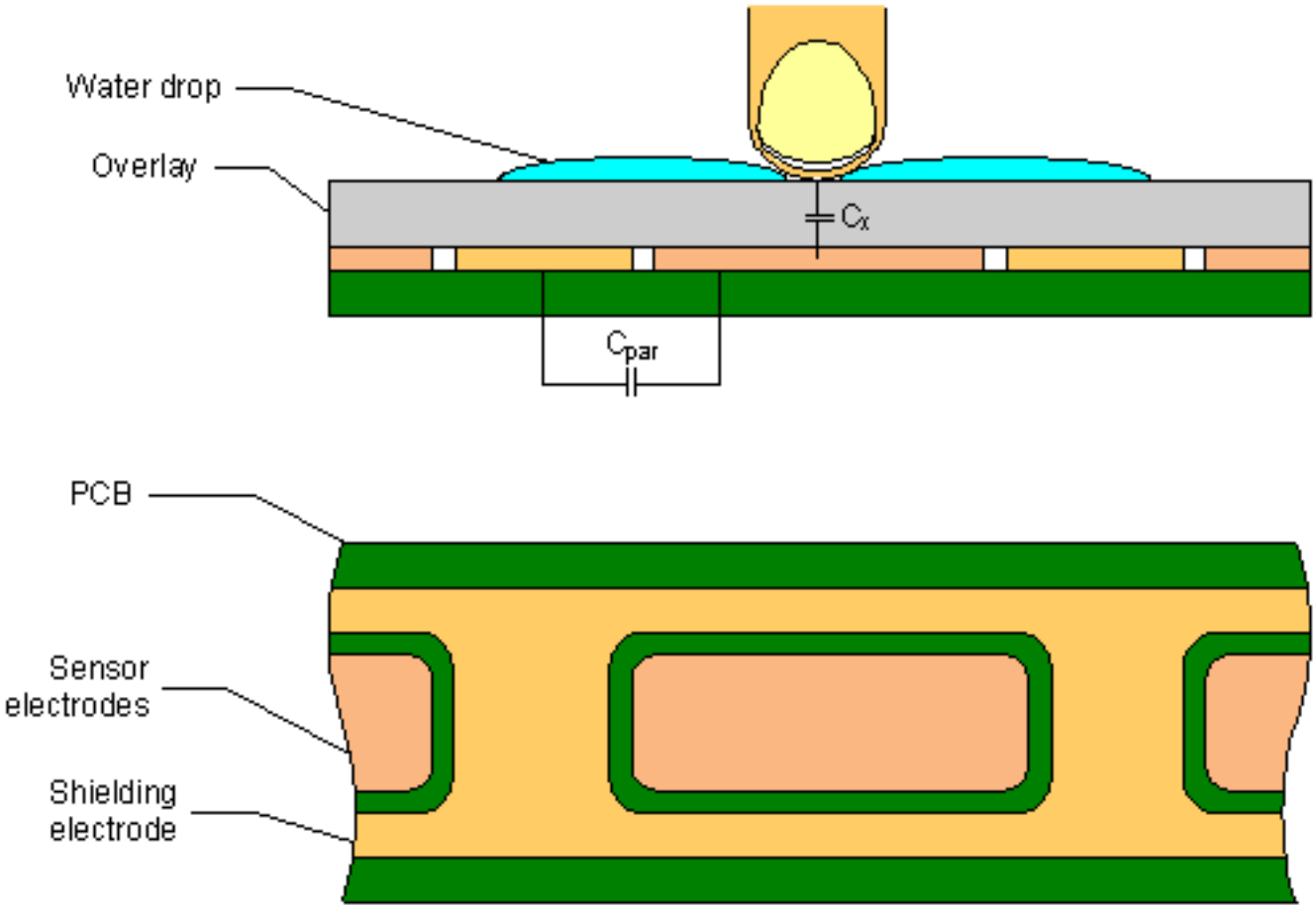
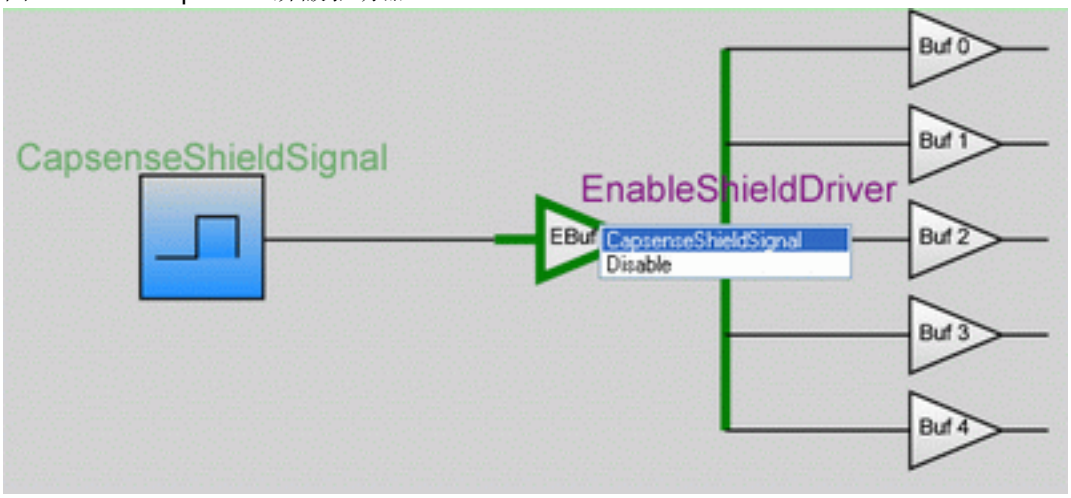


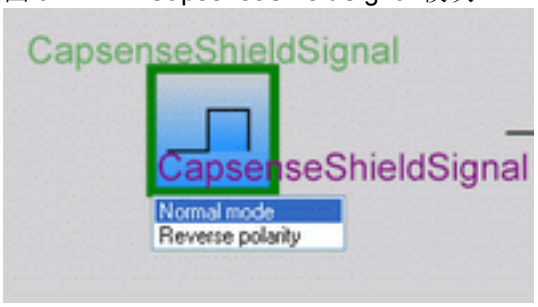
图 7 显示的是按键的驱动屏蔽电极。作为另一种替代方法，屏蔽电极可以安装在相对的 PCB 层上，其中包括按键下面的平面。对于这种情况，建议使用填充模式，填充率约为 30% 至 40%。不需要其他地层。屏蔽电极可以被连接到任何专用的 PSoC 引脚：P2[4]、P2[2]、P0[2]、P0[0] 或 P1[2]。必须将选定引脚的驱动模式设置为 High Z 模拟。为了减少散发的电磁干扰 （EMI），请在 PSoC 器件和屏蔽电极之间连接 560 Ω 斜率限流的电阻。 CapSense 屏蔽信号可以通过五个 ShieldBuffer （屏蔽缓冲区）路由到 P2[4]、P2[2]、 P0[2]、 P0[0] 或 P1[2] 引脚。PSoC Designer 芯片编辑器中的 EnableShieldDriver 模块使能此操作。

图 8. Capsense 屏蔽驱动器



屏蔽驱动器可在下面两种模式下驱动屏蔽信号：正常模式和反向极性模式。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 3. SmartSense_EMCplus 电源要求

参数	最小值	典型值	最大值	单位	测试条件和注意
V_{DD}	1.8	-	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。

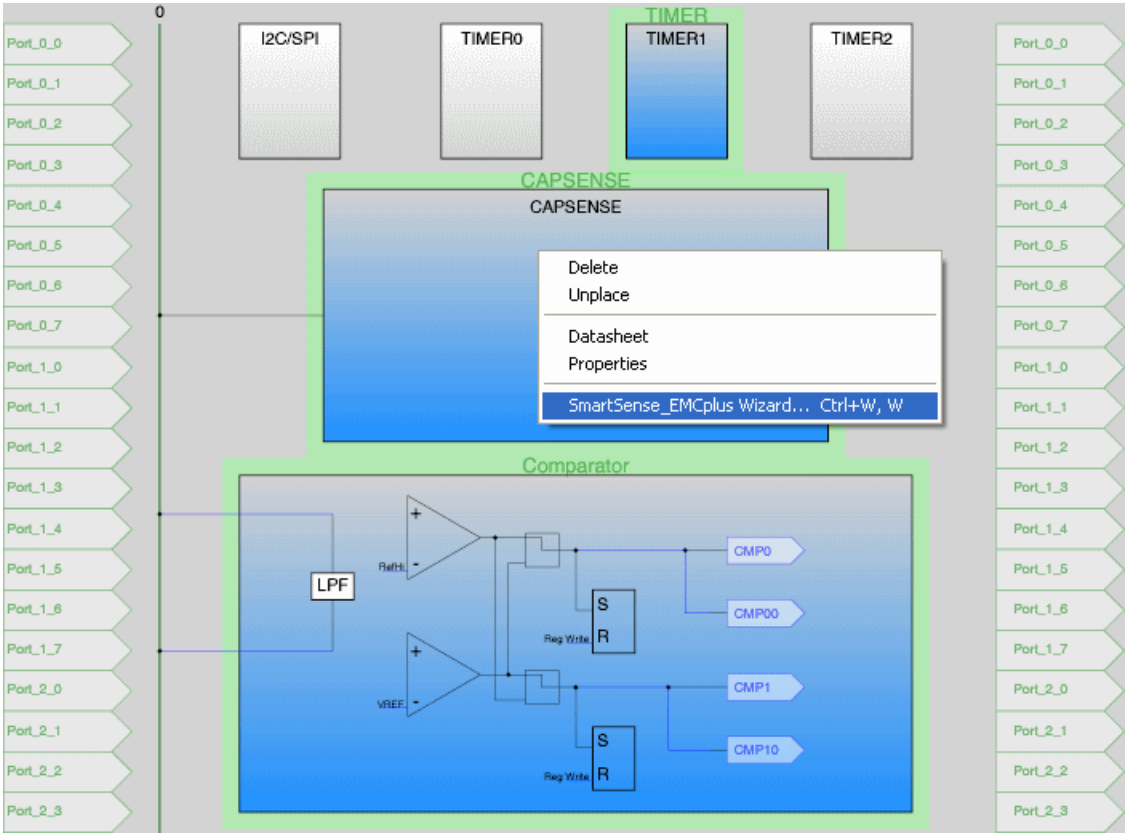
放置

当用户模块被实例化时，将 CapSense、比较器、 CapsenseShieldSignal 和定时器 1 模块分配到 SmartSense_EMCplus。备用放置不可用。在启动 SmartSense_EMCplus 向导之前必须对需要专用引脚资源 （包括 LCD 和 I2CHW）的用户模块进行放置。此外，使用 SmartSense_EMCplus 向导前进行配置 ShieldBuffer。这样，在传感器被映射到 SmartSense_EMCplus 中的 I/O 引脚时，可确保保留了专用引脚，并且这些引脚不会被意外分配给传感器。在放置电容传感器的连接时，请勿使用 P1[0] 和 P1[1]。这些引脚用于对器件进行编程，并且可能存在过大的走线电容值，这样会影响传感器的灵敏度。

SmartSense_EMCplus 向导

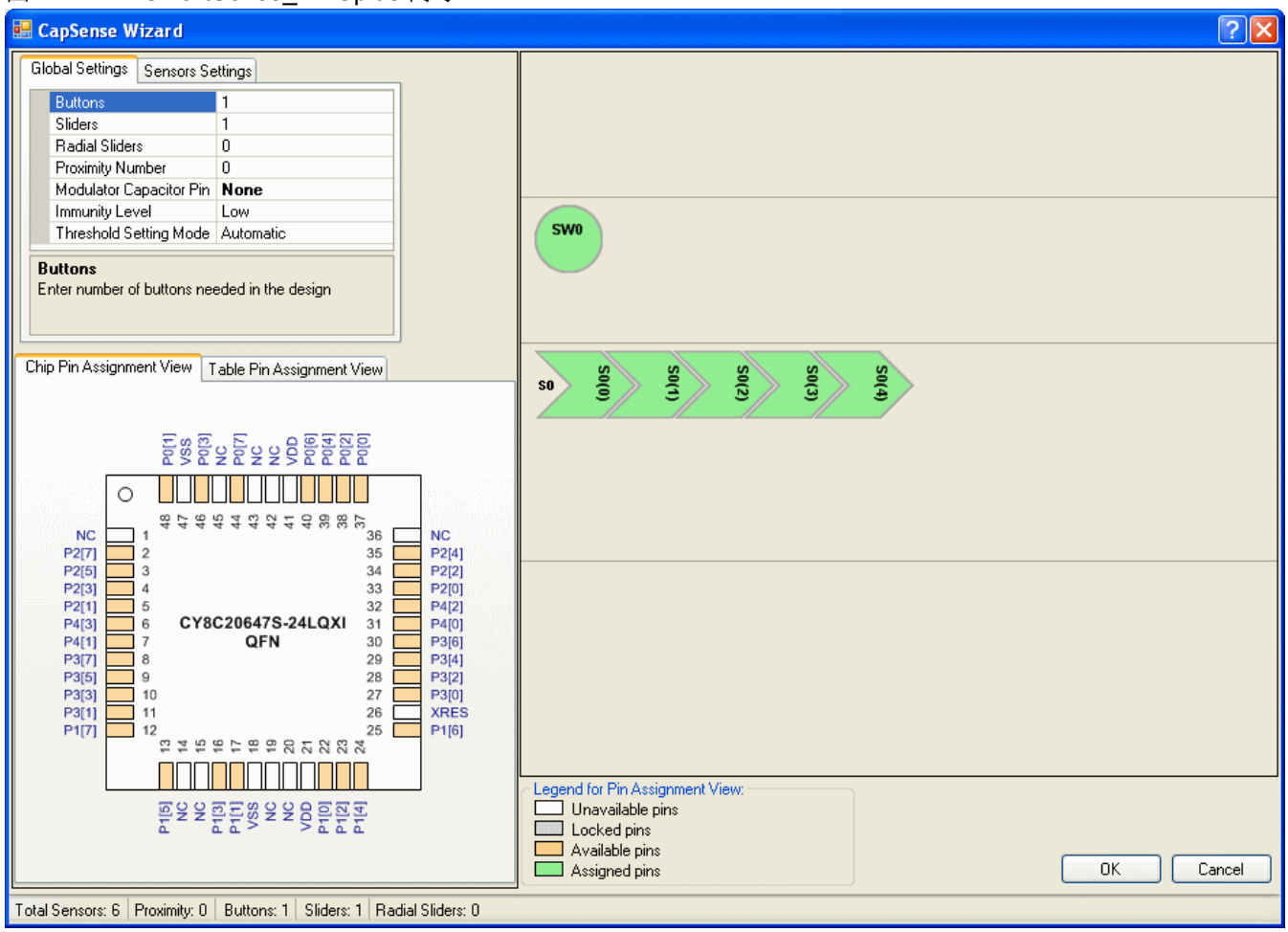
1. 如果要访问 SmartSense_EMCplus 向导，请右键点击在器件编辑器互连视图中 SmartSense_EMCplus 使用的任何模块，然后左键点击选择 SmartSense_EMCplus。

图 10. SmartSense_EMCplus 向导访问



2. 向导打开后，会显示传感器数量和滑条传感器数量的数值输入框。

图 11. SmartSense_EMCplus 向导



向导引脚图标

白色 — 引脚不能作为 CapSense 输入使用。

灰色 — 引脚被锁定。有两种可能原因：第一种是另一个用户模块 （如 LCD 或 I^2C）已占用了该引脚。第二种是引脚名称已更改，不再是默认值。要想恢复使用引脚的默认名称，请在引脚分布视图中展开该引脚，然后从 **Select** 菜单选择 **Default**。现在即可在向导中分配引脚。

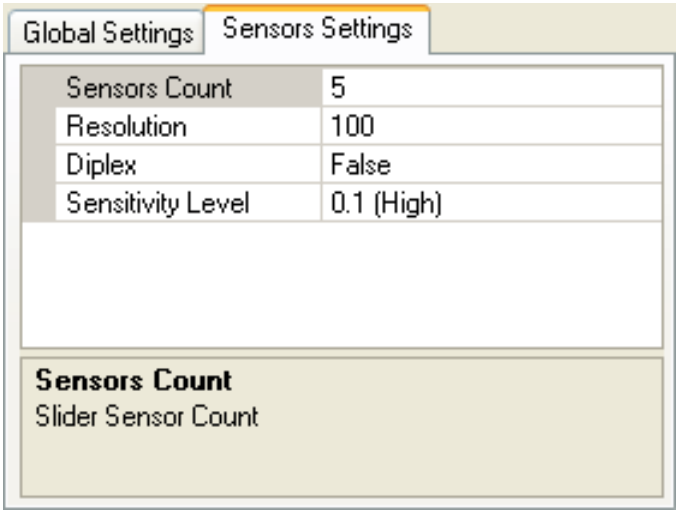
橙色 — 引脚可用于分配。

绿色 — 引脚已作为 CapSense 输入进行分配。

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

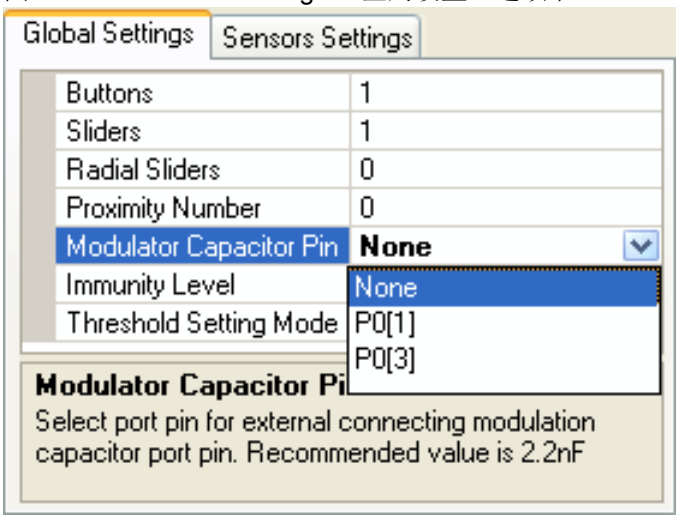
4. 选择 **Sensor Settings** （传感器设置）即可设置滑条和辐射滑条。要更改设置，请单击其中一个滑条以激活它。键入每个滑条中传感器元件的数量。滑条传感器中的传感器实际最小数量为五，最大值仅受限于引脚数量。输入数据后，按 [Enter] 键更新显示屏。

图 12. Sensor Settings （传感器设置）选项卡



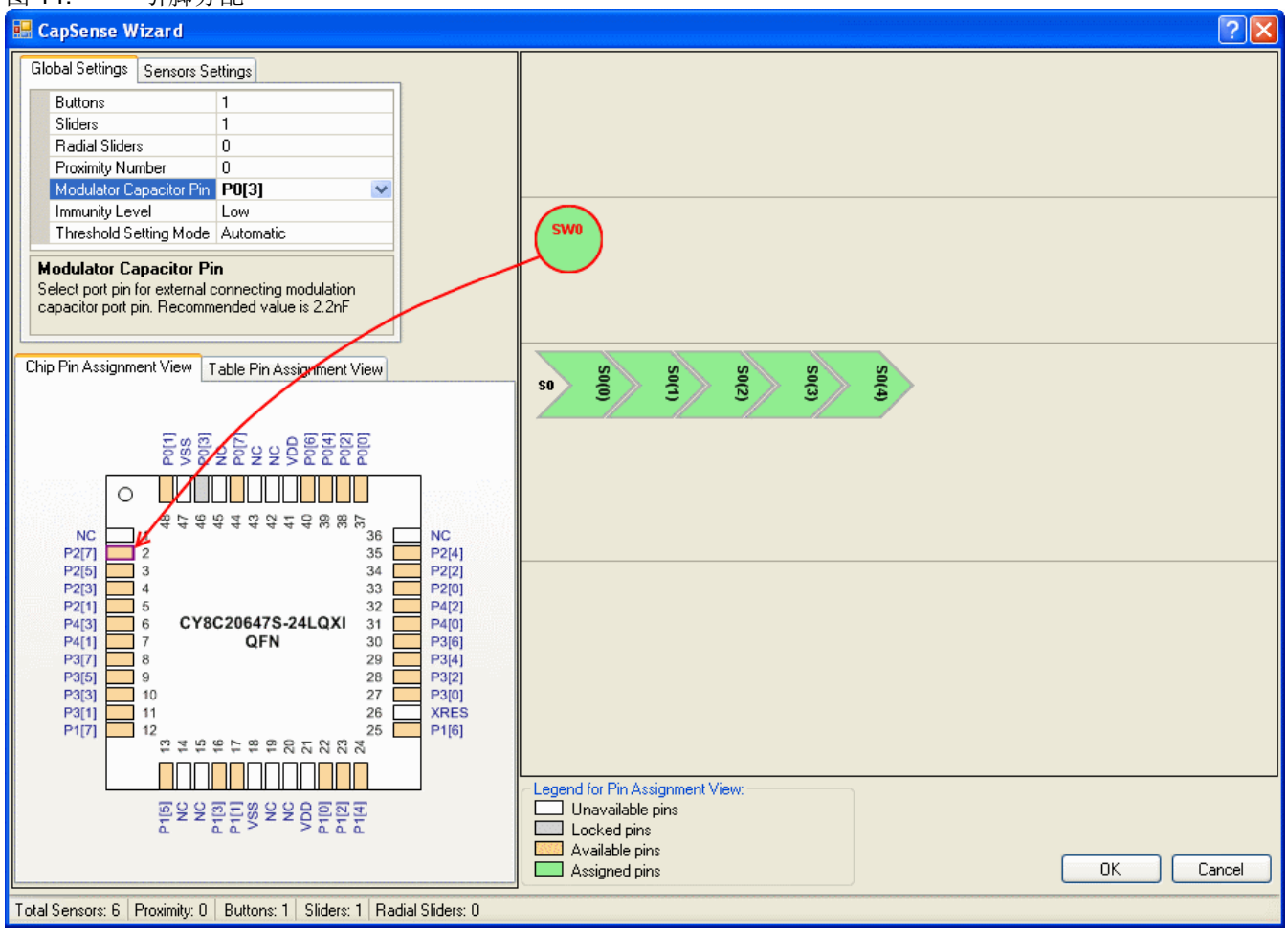
5. 选择调制器电容 （C_{mod}）引脚。选择 P0[1] 或 P0[3]。

图 13. Global Settings （全局设置）选项卡



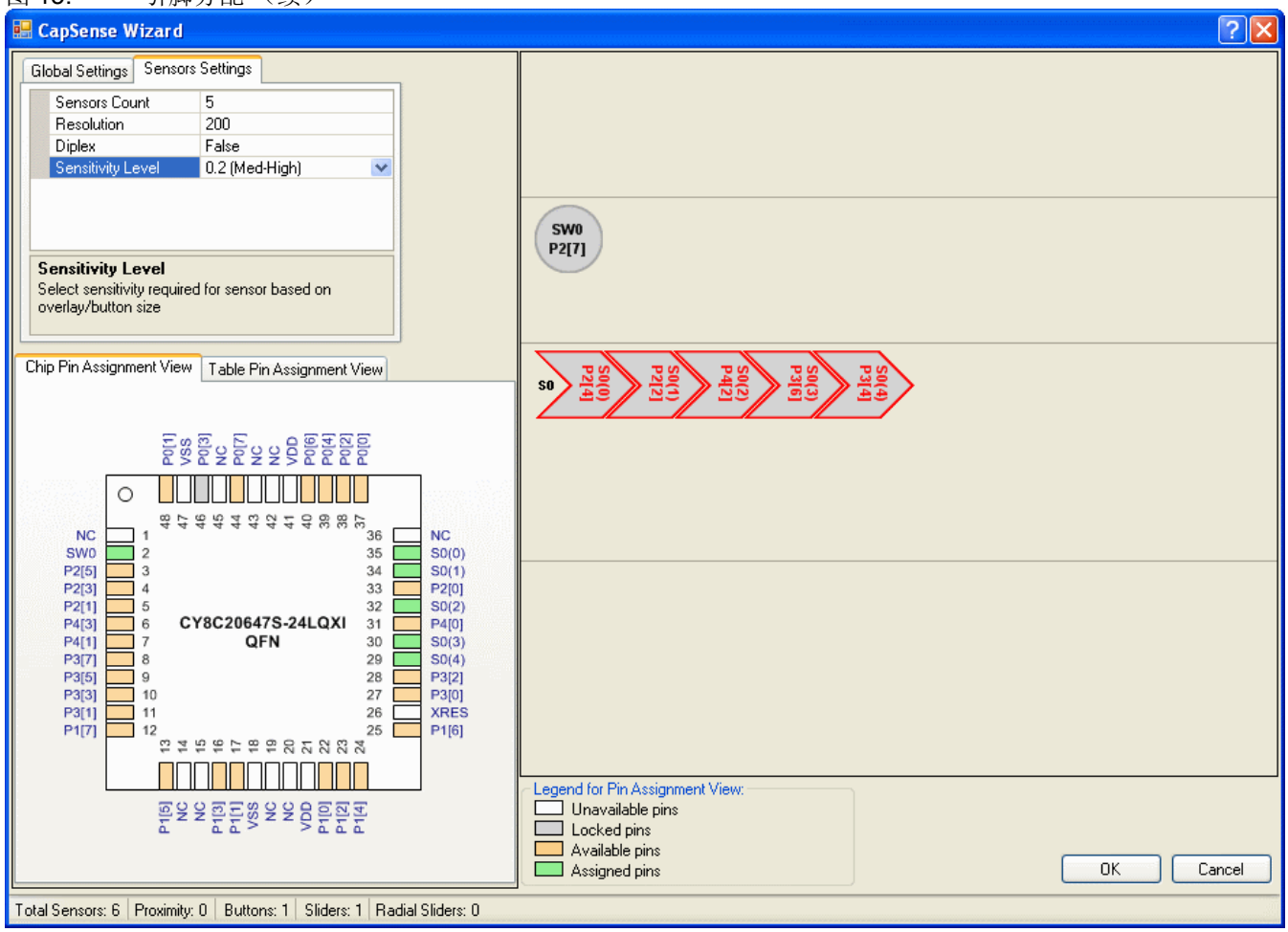
6. 键入输出分辨率。最小值为 5。SmartSense_EMCplus 尝试使用相邻段的相对强度将触摸结果内插指定的分辨率中。软件报告的滑条触摸结果在 0 ~ （分辨率 –1）之间。
7. 如果需要，请选择双工。这样会把为传感器选定的引脚数量映射为板上传感器位置数量的两倍数值。仅显示了双工传感器的前半部分；后半部分按前面 “ 双工 ” 一节所述的内容自动映射。有关引脚连接的双工表，请参见 “ 双工 ” 一节。
8. 在引脚分配视图中将每个传感器拖到引脚上，这样就可以将传感器分配到各个引脚。您可以选择在 “ 芯片引脚分配视图 ” 或 “ 表引脚分配视图 ” 中将传感器拖放到引脚上。如果您选择的是 I/O 引脚，它会变成绿色，这便表示它不再可用。通过将端口引脚拖回未提交的表格，可改变传感器的分配情况。避免选择已经指定给其他用户模块的引脚。

图 14.　　引脚分配



9. 对其他传感器重复相同的流程。单击 **OK** 按键以接收数据，然后返回到 PSoC Designer。已完成了传感器放置。右键单击 “Device Editor” 窗口并选择 **Refresh** 即可更新引脚连接。

图 15. 引脚分配 （续）



要想更改引脚分配，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。该引脚现在处于未分配状态，您可以重新分配它。

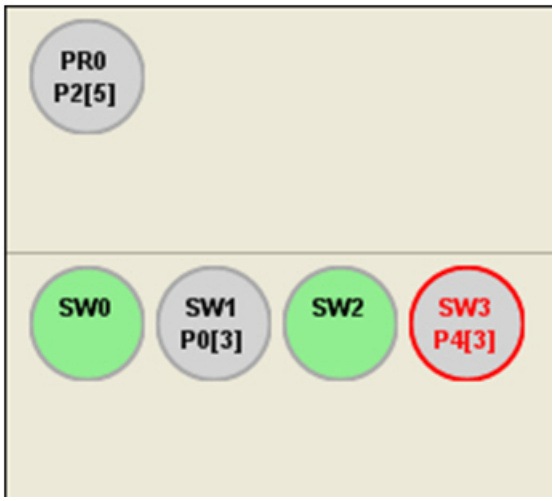
向导完成后，单击 **Generate Application** （生成应用）。根据您所输入的传感器数量、引脚分配、双工和分辨率，会生成一组表格。这些表位于 SmartSense_EMCplus_Table.asm 和 SmartSense_EMCplusHL.asm 中。

传感器演示

本章节通过图形方式表示在一个项目中表示所有可用的传感器类型。另外，本章节还描述了如何将传感器拖放到芯片或表格引脚分配视图内。所分配的传感器以灰色显示。本章节包括下面的三个部分：

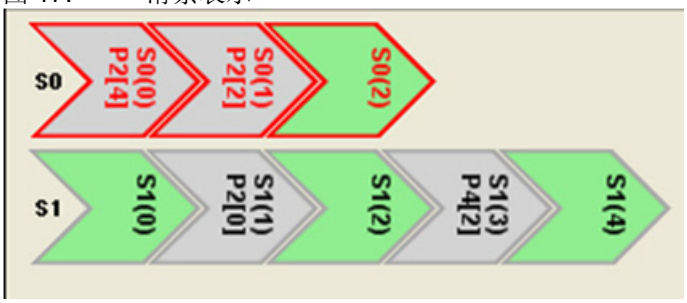
- 按键表示 — 按键传感器在向导中显示的情况如下图所示。每个按键传感器元件都有自己的标题。所有按键均可在芯片和表格引脚分配视图中实现拖放操作。如果已经分配了按键，则按键标题的下方会显示被分配的端口引脚编号。如果选择的是按键传感器，将按键 widget 的帧被设置为红色。

图 16. 按键表示



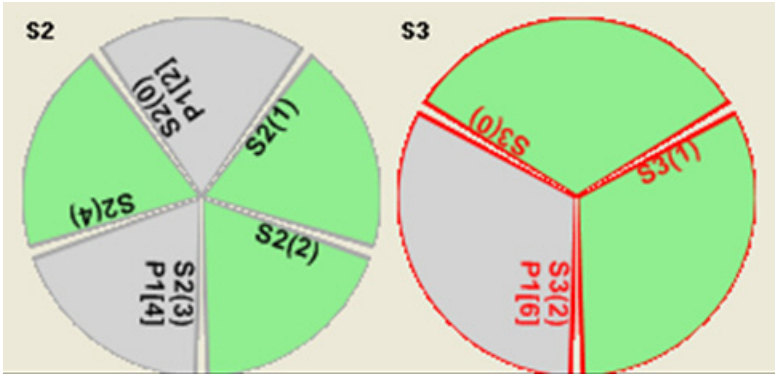
- 滑条表示 — 滑条被显示为一系列传感器段，如下图所示。在一个滑条中的每个段显示了特定传感器名称。滑条中的所有传感器均可在芯片和表格引脚分配视图中实现拖放操作。如果滑条段已被分配，则每个段会显示被分配的端口引脚编号。如果已选择了某个滑条，则滑条中所有传感器的帧都被设置为红色。

图 17. 滑条表示



- 辐射滑条表示 — 辐射滑条被显示为饼状段，如下图所示。在辐射滑条中的每一段都显示了传感器名称。辐射滑条中的所有传感器均可在芯片和表格引脚分配视图中实现拖放操作。如果辐射滑条段已被分配，每个段会显示分配的端口引脚编号。如果一个辐射滑条已被选择，辐射滑条中所有传感器的帧设置为红色。

图 18. 辐射滑条表示



状态栏

状态栏显示有关设计的通用信息，如下面所示：

- 传感器总数 — 显示设计中所使用的传感器总数量。
- 接近感应 — 显示设计中所使用的接近传感器的数量
- 按键 — 显示在设计中所使用的按键总数。
- 滑条 — 显示设计中所使用的辐射滑条总数。
- 辐射滑条 — 显示设计中所使用的辐射滑条总数。



向导按键

SmartSense_EMCplus 向导提供了具有预定义功能的按键。

1. “OK” — 该按键用于检查向导的参数是否正确，并是否已分配了所有传感器。如果正确，向导将保存参数并被关闭；否则，它将显示相应的警告信息，不会保存参数，并保持为打开状态。
2. “Cancel” — 该按键关闭向导而不会保存任何参数。
3. “Close” — 这是关闭窗口的标准按键，位于向导右上角的标题栏内。如果您点击关闭按键，则将关闭向导，而不会保存任何参数。
4. “Help” — 该按键可调用 Help 页面以提供有关如何使用 SmartSense_EMCplus 用户模块向导的参考信息。它简要地描述 SmartSense_EMCplus 用户模块向导的特性。通过点击标准窗口帮助按键，可打开 “Help” 页面。该按键标有一个问号，并位于向导右上角的标题栏内。

向导参数

在 Global Settings （全局设置）选项卡和 Sensor Settings （传感器设置）选项卡中介绍了向导参数。

Global Settings （全局设置）选项卡

Global Settings 选项卡包括以下各参数：Buttons （按键）、 Sliders （滑条）、 Radial Sliders （辐射滑条）、 Proximity （接近感应）、 Modulator Capacitor Pin （调制器电容引脚）、 Immunity Level （抗噪级别）和 Threshold Setting （阈值设置）。

Buttons （按键）

是指物理按键传感器的数量。

Sliders （滑条）

是指物理线性滑条传感器的数量。

Radial Sliders （辐射滑条）

是指物理辐射滑条传感器的数量。

Proximity （接近感应）

是指接近感应传感器的数量。

Modulator Capacitor Pin （调制器电容引脚）

该参数设置引脚，以将其连接至外部调制器电容 （C_{mod}）。可以选择可用的引脚 P0[1] 或 P0[3]。

Immunity Leve （抗噪级别）

该参数用于定义抗噪级别。可选的级别包括：Low （低）、 Medium （中）和 High （高）。选择 High （高）选项，可以改善高噪声环境下的性能。但该级别也会增大占用存储器。这样，支持传感器的最大数量会降低。高抗噪级别也会延长传感器扫描时间。同 Low 级别相比，将抗噪级别设为 Medium，会使扫描时间加倍；将抗噪级别设为 High，会使扫描时间增加 2 倍。如果您选择的是 Medium 级别，滑条会被禁用。

Threshold Setting Mode （阈值设置模式）

选择自动阈值设置或手动阈值设置。只有将抗噪级别设为 Low 时，才能选择 “ 阈值设置模式 ”。在所有其他抗噪模式下，阈值设置模式参数以灰色显示，并保持手动状态。

Sensor Settings （传感器设置）选项卡

Sensor Settings 选项卡包括以下各参数：Finger Threshold （手指阈值）、 Sensitivity Level （灵敏度）、 Diplex （双工）、 Resolution （分辨率）和 Sensors Count （传感器数量）。

Finger Threshold （手指阈值）

仅将阈值设置模式设为 Manual （手动）时，手指阈值参数才可用。该阈值用于确定每个传感器的状态。如果任何传感器处于活动状态，则 bIsAnySensorActive() 函数返回 ‘1’ 。如果所有传感器处于关闭状态，则 bIsAnySensorActive() 函数返回 ‘0’ 。可能值的范围为从 1 到 255。

应将该参数设为触摸传感器时所收到的信号的 80%。从每个传感器中收到的信号被存储在阵列变量 SmartSense_EMCplus_baSnsSignal[] 中。

Diplex （双工）

该选项可使能或禁用滑条双工特性。有关更详细的信息，请参见本用户模块数据手册中的 Diplexing （双工）一节。

Resolution （分辨率）

通过该选项，可将传感器分辨率的范围设置为 5 到 (传感器所用的引脚数量 – 1) × 2^8 – 1 ；对于双工滑条，该值范围为 5 到 (2 × 传感器所用的引脚数量 – 1) × 2^8 – 1。

Sensors Count （传感器数量）

这是滑条或辐射滑条中物理传感器的数量。

Sensitivity Leve （灵敏度级别）

灵敏度用于增加或减少传感器发出的信号强度。设置灵敏度值越低 （0.1 pF），传感器发出信号越强。为能够准确达到要求，设计的覆盖层越厚，所需的传感器信号也就越强。灵敏度选项有：“ 高 （0.1 pF）”、“ 中高 （0.2 pF）”、“ 中低 （0.3 pF）” 和 “ 低 （0.4 pF）”。

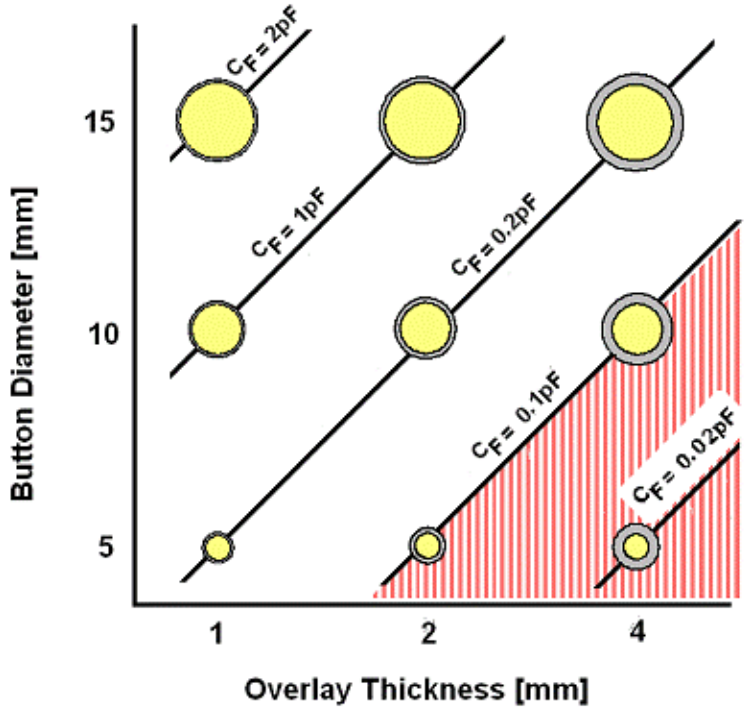
要产生更强的传感器信号 （高灵敏度）， SmartSense_EMCplus 须消耗更多的传感器扫描时间。这意味着：传感器灵敏度等级设置为 0.1 PF （“ 高 ”）与设置为 0.2 PF （“ 中高 ”）相比，前者消耗更多的扫描时间。

一个最佳调校方法是查找产生要求的 5:1 信噪比时传感器灵敏度的最高值。可以使用灵敏度最高值 （0.4 pF）开始调校，并按需减少值直至满足 5:1 信噪比。

图 16 显示了按键大小、覆盖层厚度 （丙烯酸塑料）和传感器响应 （CF）间的关系。该图可以作为设置传感器灵敏度的指南。始终应该将传感器灵敏度设置为图 16 所示的传感器响应或更低。这样可

确保能够稳定地运行。请注意，必须避免设置为红色阴影的区域的值，因为传感器响应低于 SmartSense_EMCplus 可检测到的最小值 0.1。

图 19. 按键大小、覆盖层厚度和传感器响应 （单位为 pF）之间的关系



示例代码部分中的示例 3 显示了如何将单独传感器的灵敏度设为另外一个数值而不是该参数指定的默认值。

向导生成的表

向导完成后，单击 “Generate Application” （生成应用）。根据您所输入的传感器数量、引脚分配、双工和分辨率，会生成一组表格。这些表位于 SmartSense_EMCplus_Table.asm 和 SmartSense_EMCplusHL.asm 中。

传感器表

传感器表中每个传感器条目包括两个字节。第一个字节是端口编号，第二个字节是位的掩码 （不是位编号）。下面的示例表显示了六个传感器：

```
SmartSense_EMCplus_Sensor_Table:
_SmartSense_EMCplus_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
```

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，独立按键传感器再对应一个条目。第一个条目通常是独立的按键传感器。每个条目为六字节。第一个字节表示传感器表中组开始的索引。第二个字节表示该组中传感器的数量。第三个字节表示是否对滑条采用了双工法（4 表示已采用， 0 表示未采用）。第四、五、六个字节是固定点乘数，将其与滑条中心相乘可以得出 SmartSense_EMCplus 向导中指定的分辨率。

```
SmartSense_EMCplus_Group_Table:
_SmartSense_EMCplus_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

双工表扫描顺序数据用于已使能双工的滑条组。否则，将创建标签而不放置任何数据。该表由两个部分组成：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。下面展示了一个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


 SmartSense_EMCplus_Diplex_Table:
_SmartSense_EMCplus_Diplex_Table:
 db >DiplexTable_0, <DiplexTable_0
 db >DiplexTable_1, <DiplexTable_1
```

手指阈值表

手指阈值表定义了每个传感器的标准手指阈值。下面是一个 1 按键和 5 传感器滑条的典型示例。

```
SmartSense_EMCplus_Finger_Threshold_Table:
_SmartSense_EMCplus_Finger_Threshold_Table:
   db 255   ; Buttons
   db 1,1,1,1,1  ; Sliders
```

灵敏度级别表

灵敏度级别表定义了每个传感器的灵敏度。下面是一个 1 按键和 5 传感器滑条的典型示例。

```
SmartSense_EMCplus_Sensitivity_Level_Table:
_SmartSense_EMCplus_Sensitivity_Level_Table:
   db 0   ; Buttons
   db 0,0,0,0,0   ; Sliders
```

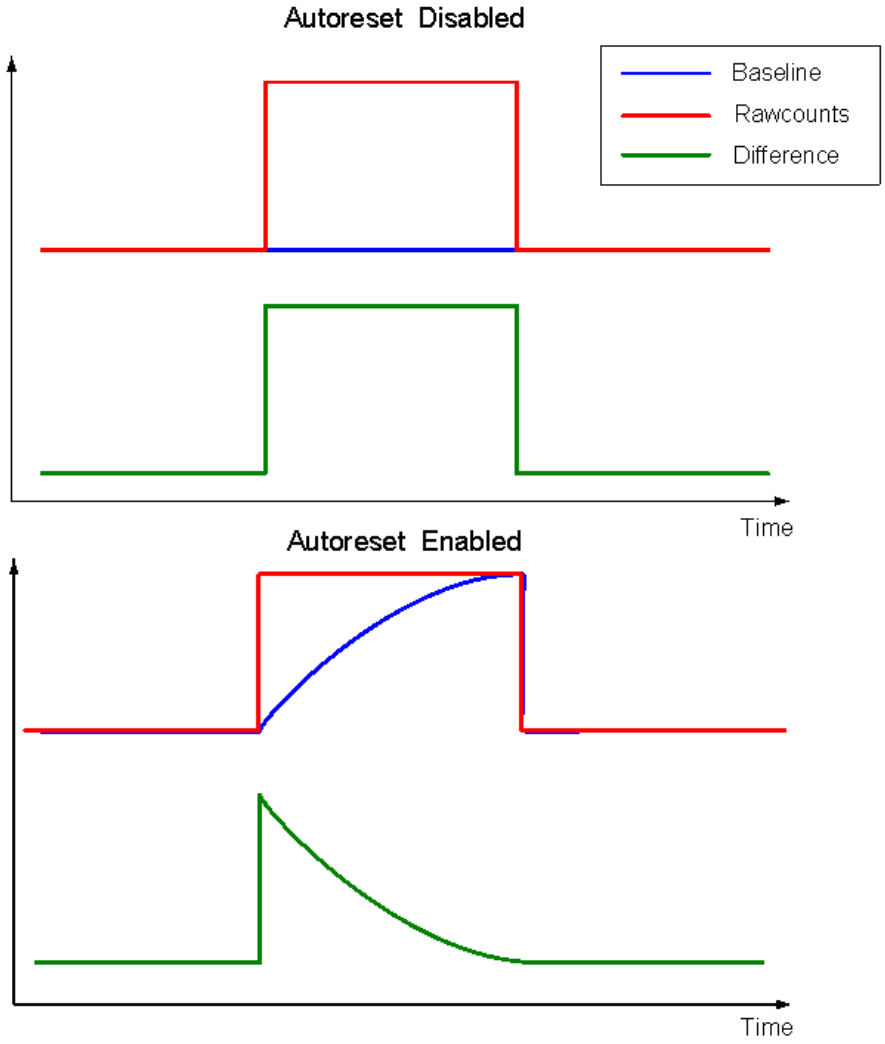
参数和资源

在 SmartSense_EMCplus 向导中完成配置和 I/O 引脚分配后，必须设置用户模块参数。请注意，更改任何用户模块参数生效时，都必须重新生成项目。

传感器自动复位

该参数确定了基准线被更新的时间：随时更新或只当信号差值低于噪声阈值时才会更新。如果将该参数默认值设置为 Disabled （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，才会更新基准线。图 17 说明了该参数对基准线更新的影响。当 Sensors Autoreset 设置为 Enabled （使能）时，无论噪声阈值如何，均始终更新基准线。这样会限制传感器的最大激活时间 （通常为 5 - 10 秒）。但对防止传感器因非触摸引起的原始计数突然上升而停滞带来了好处。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” 设置为 Disabled，则仅当原始计数与基准线之间的差值低于噪声阈值时，才进行更新基准线。应该将该参数设置为 “Disabled” 的默认状态。有关该参数的详细信息，请参考 “ 附录 ” 一节。

图 20. 传感器自动复位参数对基准线更新的影响



去抖动

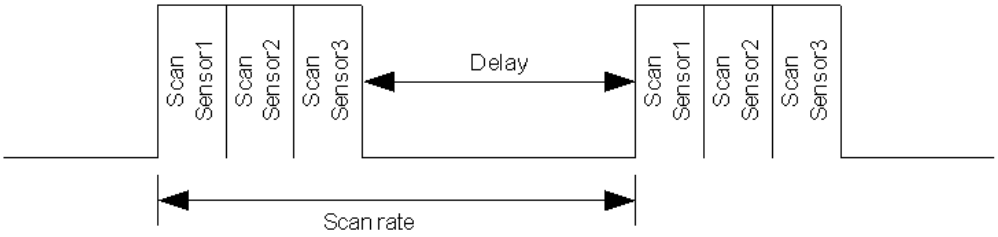
该参数为传感器活动的瞬变增加了去抖动计数器。为了让传感器能够从未激活状态切换为激活状态，在该参数指定的样本数量内，差异计数值必须大于手指阈值与迟滞之和。去抖动计数器由 bIsSensorActive 或 bIsAnySensorActive API 函数递增。

取值范围为 1 到 255。如果该参数设置为 ‘1’ ，则没有去抖动，但会提供最快的响应。默认设置为 3。

传感器扫描速率选型指南

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。按顺序进行扫描设计中的所有传感器，而在各次扫描间存在延迟。

图 21. 典型传感器扫描



为了确保基准线能正常运行，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。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，以创建一个低功耗设计。

应用编程接口 （API）

API 函数作为用户模块的一部分提供，允许您能够以更高级别处理该模块。本节指定每个函数的接口，以及 “ 包括 ” 文件所提供的相关常量。

只能将该用户模块的一个实例放置在项目中，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 向给定项目中此用户模块的第一个实例分配 SmartSense_EMCplus_1。可将该值更改为任意一个符合标识符语法规则的值。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 SmartSense_EMCplus。

**注意 **：在这种情况下，同所有用户模块的 API 一样， A 和 X 寄存器的值可以通过调用 API 函数来更改。如果调用后需要使用 A 和 X 的值，则调用函数将保留调用前 A 和 X 的值。选择这种 “ 寄存器易失 ” 策略是为了提高效率，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。汇编语言编程员也必须保证他们的代码遵守该策略。虽然一些用户模块 API 函数可以保持 A 和 X 不变，但是无法保证它们将来也会如此。

对于大型存储器模型器件，调用程序需要保留 CUR_PP、 IDX_PP、 MVR_PP 以及 MVW_PP 等寄存器中的所有值。尽管目前尚未修改某些寄存器，但无法保证在将来的版本中也会如此。

提供了进入点，用以初始化 SmartSense_EMCplus，启动其样本并停止 SmartSense_EMCplus。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 SmartSense_EMCplus 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。请不手动修改这些阵列。但是，您可以检查这些值以进行调校。例如，可以使用绘图工具显示阵列的内容。有多种全局阵列：

- SmartSense_EMCplus_waSnsBaseline[]
- SmartSense_EMCplus_waSnsResult[]
- SmartSense_EMCplus_waSnsDiff[]
- SmartSense_EMCplus_baSnsOnMask[]
- SmartSense_EMCplus_baSnsSignal[]
- SmartSense_EMCplus_baDAC[]
- SmartSense_EMCplus_baCompensationDAC[]

SmartSense_EMCplus_waSnsBaseline[]：这是一个包含每个传感器的基准线数据的整数阵列。阵列大小与传感器数量相等。通过下列函数更新 SmartSense_EMCplus_waSnsBaseline[] 阵列：

- SmartSense_EMCplus_UpdateAllBaselines()
- SmartSense_EMCplus_UpdateSensorBaseline()
- SmartSense_EMCplus_InitializeBaselines()

SmartSense_EMCplus_waSnsResult[]：这是一个包含每个传感器的原始信号的整数阵列。阵列大小与传感器数量相等。通过下列函数更新 SmartSense_EMCplus_waSnsResult[] 的数据：

- SmartSense_EMCplus_ScanSensor()
- SmartSense_EMCplus_ScanAllSensors().

SmartSense_EMCplus_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。通过下列函数更新 SmartSense_EMCplus_waSnsDiff[] 的数据：

- SmartSense_EMCplus_UpdateAllBaselines()
- SmartSense_EMCplus_UpdateSensorBaseline()

SmartSense_EMCplus_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）。 SmartSense_EMCplus_baSnsOnMask[0] 包含传感器 0 到 7 的掩码位 （传感器 0 的是位 0，传感器 1 的是位 1）。 SmartSense_EMCplus_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。通过下列函数更新 SmartSense_EMCplus_baSnsOnMask[] 数据：

- SmartSense_EMCplus_bIsSensorActive()
- SmartSense_EMCplus_bIsAnySensorActive()

SmartSense_EMCplus_baSnsSignal[]：这是一个字节阵列，该阵列包含每个传感器的标准手指触摸信号数据。该阵列值适用于监控手指触摸的信号，并在选择手动阈值设置模式时会设置手指阈值。阵列大小与传感器数量相等。使用 SmartSense_EMCplus_UpdateSensorSignal() 函数更新了 SmartSense_EMCplus_baSnsSignal 阵列。

SmartSense_EMCplus_baDAC[] ：这是一个包含了所有传感器的 IDAC1_CODE 寄存器值的字节阵列。扫描前，该阵列中的值被复制到 SmartSence_EMCplus_bIdacValue 变量 。

SmartSense_EMCplus_baCompensationDAC[] — 这是一个包含了所有传感器的 IDAC0_CODE 寄存器值的字节阵列。扫描前，将该阵列中的值复制到 SmartSense2X_EMC_bCompensationIdacValue 变量。

SmartSense_EMCplus_SnsErrorStatus[] ：这是一个字节阵列，该阵列为每个 CapSense 传感器保留了一位，用以表示 C_P 超出了设计极限。该阵列变量的大小等于传感器总数除以 8 字节 （与现有用户模块中的 _baSnsOnMask[] 阵列变量大小一样，用以指示传感器的打开 / 关闭状态）。如果有任何传感器的测量 C_P 值超出了设计的极限，将设置与那个传感器相应的位，以指示错误状态。

SmartSense_EMCplus_Start

说明：

初始化寄存器并启动用户模块。调用其他任何用户模块函数前必须调用该函数。

C 原型：

```
void  SmartSense_EMCplus_Start(void)
```

汇编：

```
lcall SmartSense_EMCplus_Start
```

参数：

无

返回值：

无

其他影响：

**

SmartSense_EMCplus_Stop

说明：

将 CapSense 模块恢复到其闲置的默认配置，出于其他目的释放 AMUX 总线，禁用内部中断，并调用 SmartSense_EMCplus_ClearSensors() 来使所有传感器重置为它们的未激活状态。

C 原型：

```
void  SmartSense_EMCplus_Stop(void)
```

汇编：

```
lcall SmartSense_EMCplus_Stop
```

参数：

无

返回值：

无

其他影响：

**

SmartSense_EMCplus_Resume

说明：

调用 SmartSense_EMCplus_Stop 后，恢复用户模块操作。

C 原型：

```
void  SmartSense_EMCplus_Resume(void)
```

汇编：

```
lcall SmartSense_EMCplus_Resume
```

参数：

无

返回值：

无

其他影响：

**

SmartSense_EMCplus_ScanSensor

说明：

扫描所选的传感器。每个传感器由传感器表中其位置单独标识。 SmartSense_EMCplus 向导分配该位置或传感器编号。

C 原型：

```
void  SmartSense_EMCplus_ScanSensor(BYTE bSensor)
```

汇编：

```
mov   A, bSensor
lcall SmartSense_EMCplus_ScanSensor
```

参数：

A => 传感器编号

返回值：

无

其他影响：

**

SmartSense_EMCplus_ScanAllSensors

说明：

通过调用每个传感器的 SmartSense_EMCplus_ScanSensor() 扫描所有已配置的传感器。

C 原型：

```
void  SmartSense_EMCplus_ScanAllSensors(void)
```

汇编：

```
lcall SmartSense_EMCplus_ScanAllSensors
```

参数：

无

返回值：

无

其他影响：

**

SmartSense_EMCplus_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准线。通过使用带有以下算法的水桶方法，对该基准线进行更新：

1. 每次调用 SmartSense_EMCplus_UpdateSensorBaseline() 时，通过将原始计数值减去以前的基准线值来计算差值计数。该差值存储在 SmartSense_EMCplus_waSnsDiff[] 阵列中。
2. 如果禁用了 Sensors Autoreset，每次调用 SmartSense_EMCplus_UpdateSensorBaseline() 时，将对差值和噪声阈值进行比较。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 Sensors Autoreset，则无论噪声阈值参数如何，差值都会累加到虚拟水桶中。

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 SmartSense_EMCplus_waSnsDiff[] 阵列中的值将复位为 0。因此，该阵列不包含数值大于 0 但低于噪声阈值的元素。

C 原型：

```
SmartSense_EMCplus_UpdateSensorBaseline(BYTE bSensorNum)
```

汇编：

```
mov   A, bSensor
lcall SmartSense_EMCplus_UpdateSensorBaseline
```

参数：

A => 传感器编号

返回值：

无

其他影响：

**

SmartSense_EMCplus_UpdateAllBaselines

说明：

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

C 原型：

```
void  SmartSense_EMCplus_UpdateAllBaselines(void)
```

汇编：

```
lcall SmartSense_EMCplus_UpdateAllBaselines
```

参数：

无

返回值：

无

其他影响：

**

SmartSense_EMCplus_bIsSensorActive

说明：

检查给定传感器与手指阈值间的差值计数阵列。迟滞也被计算在内。根据传感器当前是否开启，对手指阈值增加或减去迟滞值。如果传感器被激活，则阈值被降低；如果它处于无效状态，则阈值会升高。该函数还可更新 SmartSense_EMCplus_baSnsOnMask[] 阵列中传感器的位。

C 原型：

```
BYTE SmartSense_EMCplus_bIsSensorActive(BYTE bSensorNum)
```

汇编：

```
mov   A, bSensor
lcall SmartSense_EMCplus_bIsSensorActive
```

参数：

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：表示选定的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：

**

SmartSense_EMCplus_bIsAnySensorActive

说明：

检查所有传感器与手指阈值之间的差值计数阵列。针对每个传感器调用 SmartSense_EMCplus_bIsSensorActive()，以便在调用该函数后更新 SmartSense_EMCplus_baSnsOnMask[] 阵列。

C 原型：

```
BYTE  SmartSense_EMCplus_bIsAnySensorActive(void)
```

汇编：

```
lcall SmartSense_EMCplus_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：

**

SmartSense_EMCplus_wGetCentroidPos

说明：

检查中心位置的线性滑条。如果存在中心，则将偏移和长度都存储在临时变量中，并根据 SmartSense_EMCplus 向导中指定的分辨率计算中心位置。仅在滑条是由 SmartSense_EMCplus 向导定义时，此函数才可用。

C 原型：

```
WORD  SmartSense_EMCplus_wGetCentroidPos(BYTE bSnsGroup)
```

汇编：

```
mov   A, bSnsGroup
lcall SmartSense_EMCplus_wGetCentroidPos
```

参数：

bSnsGroup A => 组编号。

该参数表示滑条的组编号。组 0 始终是独立按键传感器。则滑条传感器包含在组 1 和更高的组中。

返回值：

滑条的位置， LSB 位于 A 中、 MSB 位于 X 中。

其他影响：

该子程序通过减去噪声阈值更改差值计数，并且在每次扫描结束后仅一次调用该子程序，以避免得到负差值。如果应用监控差值信号，在差值计数数据传输后调用该子程序。

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，则函数返回 -1 （FFFFh）。如果在执行中心 / 双工算法时出现错误，则该函数返回 -1 （FFFFh）。您可以使用 SmartSense_EMCplus_bIsSensorActive() 函数来确定触摸了哪些滑条段。

SmartSense_EMCplus_wGetRadialPos

说明：

检查中心位置的辐射滑条阵列。如果存在中心，根据 SmartSense_EMCplus 向导中指定的分辨率计算中心位置。

C 原型：

```
WORD  SmartSense_EMCplus_wGetRadialPos(BYTE bSnsGroup)
```

汇编：

```
mov   A, bSnsGroup
lcall SmartSense_EMCplus_wGetRadialPos
```

参数：

bSnsGroup A => 组编号

该参数表示辐射滑条的组编号。该编号可以通过辐射滑条表示法左侧上的 SmartSense_EMCplus 向导获取 （例如：“s2” 表示辐射滑条的组编号为 2）。

返回值：

辐射滑条的位置， LSB 位于 A 中和 MSB 位于 X 中。

其他影响：

该子程序通过减去噪声阈值来修改差值计数。每次扫描后必须只调用一次子程序，以避免得到负的差值及更新基准线。如果应用对差值信号进行监控，则在差值计数数据传输后会调用该子程序。如果某个滑条传感器处于活动状态，则该函数将返回一个属于零到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，则函数返回 -1 （FFFFh）。如果在执行中心算法时出现了错误，该函数将返回 -1 （FFFFh）。您可以使用 SmartSense_EMCplus_bIsSensorActive() 函数来确定触摸了哪些滑条段。

SmartSense_EMCplus_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。该函数与 SmartSense_EMCplus_wGetRadialPos() 同时使用。

C 原型：

```
WORD  SmartSense_EMCplus_wGetRadialInc(BYTE bSnsGroup)
```

汇编：

```
mov   A, bSnsGroup
lcall SmartSense_EMCplus_wGetRadialInc
```

参数：

bSnsGroup A => 组编号

该参数表示辐射滑条的组编号。该编号可以通过辐射滑条表示法左侧上的 SmartSense_EMCplus 向导获取 （例如：“s2” 表示辐射滑条的组编号为 2）。

返回值：

手指移位值 （顺时针为正，逆时针为负）， LSB 位于 A 中以及 MSB 位于 X 中。

手指移位值是手指的当前位置与先前位置之间的差值。

其他影响：

仅在调用 SmartSense_EMCplus_wGetRadialPos() 后才能调用该函数，因为后面要使用存储在全局变量中的内部数据。

SmartSense_EMCplus_InitializeSensorBaseline

说明：

通过扫描选定的传感器，将初始值加载到 SmartSense_EMCplus_waSnsBaseline[bSensor] 阵列元素中。原始计数值将被复制到所选传感器的基准线阵列元素中。该函数可用于复位单个传感器的基准线值。

C 原型：

```
void SmartSense_EMCplus_InitializeSensorBaseline(BYTE bSensorNum)
```

汇编：

```
mov   A, bSensor
lcall SmartSense_EMCplus_InitializeSensorBaseline
```

参数：

A => 传感器编号

返回值：

无

其他影响：

**

SmartSense_EMCplus_InitializeBaselines

说明：

通过扫描每个传感器，将其初始值加载到 SmartSense_EMCplus_waSnsBaseline[] 阵列中。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 SmartSense_EMCplus_InitializeBaselines(void)
```

汇编：

```
lcall SmartSense_EMCplus_InitializeBaselines
```

参数：

无

返回值：

无

其他影响：

**

SmartSense_EMCplus_ClearSensors

说明：

通过为每个传感器依次调用 SmartSense_EMCplus_wGetPortPin() 和 SmartSense_EMCplus_DisableSensor()，将所有的传感器清除为非采样状态。

C 原型：

```
void  SmartSense_EMCplus_ClearSensors(void)
```

汇编：

```
lcall SmartSense_EMCplus_ClearSensors
```

参数：

无

返回值：

无

其他影响：

**

SmartSense_EMCplus_wReadSensor

说明：

返回按键原始扫描值通过 A 和 X （分别为 LSB 和 MSB）。

C 原型：

```
WORD  SmartSense_EMCplus_wReadSensor(BYTE bSensor)
```

汇编：

```
mov   A, bSensor
lcall SmartSense_EMCplus_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：

**

SmartSense_EMCplus_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 SmartSense_EMCplus_Sensor_Table[] 中的数据进行索引和选择。可将返回值传递给 SmartSense_EMCplus_EnableSensor() 和 SmartSense_EMCplus_DisableSensor()。

C 原型：

```
WORD  SmartSense_EMCplus_wGetPortPin(BYTE bSensor)
```

汇编：

```
mov   A, bSensor
lcall SmartSense_EMCplus_wGetPortPin
```

参数：

bSensor — 范围为 0 到 （n – 1），其中 ‘n’ 是 SmartSense_EMCplus 向导中设置的传感器数量与滑条中存在的传感器数量之和。 SmartSense_EMCplus_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：

**

SmartSense_EMCplus_EnableSensor

说明：

配置所选的传感器，以便在下个测量周期中进行测量。通过 SmartSense_EMCplus_wGetPortPin() 函数可以选择端口和传感器，其中端口编号和传感器位掩码分别被加载到 X 和 A 中。驱动模式被修改，以便使所选的端口和引脚进入模拟高阻态模式并使能正确的模拟复用器总线输入。

C 原型：

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

汇编：

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

参数：

A => 传感器位图， X => 端口编号

返回值：

无

其他影响：

**

SmartSense_EMCplus_DisableSensor

说明：

禁用 SmartSense_EMCplus_wGetPortPin() 函数选定的传感器。驱动模式被更改为 Strong “ 强（001）”，便能够将传感器有效接地。端口引脚与模拟复用器总线的连接被断开。SmartSense_EMCplus_wGetPortPin() 函数返回函数参数。

C 原型：

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

汇编：

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

参数：

A => 传感器位图， X => 端口编号

返回值：

无

其他影响：

**

SmartSense_EMCplus_UpdateSensorSignal

说明：

使用标准差值更新 SmartSense_EMCplus_baSnsSignal[] 阵列。

C 原型：

```
BYTE SmartSense_EMCplus_UpdateSensorSignal(BYTE bSensorNum)
```

汇编：

```
mov   A, bSensor
lcall SmartSense_EMCplus_UpdateSensorSignal
```

参数：

A => 传感器编号

返回值：

A => 标准差值

其他影响：

**

SmartSense_EMCplus_GetSnsParasiticCapacitance

说明：

该 API 返回传感器的寄生电容，单位为 pF。

C 原型：

```
BYTE SmartSense_EMCplus_GetSnsParasiticCapacitance(BYTE bSensor)
```

参数：

bSensor A => 传感器编号

返回值：

A => 传感器的寄生电容，单位为 pF

其他影响：

**

示例固件源代码

示例 1. 该代码用于启动用户模块，并连续扫描传感器。可以使用通信部分将数值传输给 PC 绘图工具。

```
//-------------------------------------------------------------------------
// Sample C code for the SmartSense_EMCplus 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;
    SmartSense_EMCplus_Start();
SmartSense_EMCplus_InitializeBaselines(); // Scan all sensors first time, init baseline
    //
    // Loop Forever
//
while (1)
    {
        SmartSense_EMCplus_ScanAllSensors(); // Scan all sensors in array (buttons and
sliders)
        SmartSense_EMCplus_UpdateAllBaselines(); // Update all baseline levels;

// Detect if any sensor is pressed
        if(SmartSense_EMCplus_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 SmartSense_EMCplus_waSnsResult[x] <- Raw Counts
// OUTPUT SmartSense_EMCplus_waSnsDiff[x] <- Difference
// OUTPUT SmartSense_EMCplus_waSnsBaseline[x] <- Baseline
// OUTPUT SmartSense_EMCplus_baSnsOnMask[x] <- Sensor On/Off
}
}
```

示例 2. 该代码用于启动用户模块，并连续扫描传感器；但与示例 1 不同，通过传感器的环路是在用户代码中完成的。可以使用通信部分将数值传输给 PC 绘图工具。

```
//-------------------------------------------------------------------------
// Sample C code for the SmartSense_EMCplus 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)
{
    BYTE bIndex;
    M8C_EnableGInt;

    SmartSense_EMCplus_Start();
    SmartSense_EMCplus_InitializeBaselines(); // Scan all sensors first time, init
baseline
    while (1) // Loop forever
    {
        // Loop through all sensors
        for(bIndex=0; bIndex < SmartSense_EMCplus_TotalSensorCount; bIndex++)
        {
            SmartSense_EMCplus_ScanSensor(bIndex); // Scan Sensors
            SmartSense_EMCplus_UpdateSensorBaseline(bIndex); // Run baseline filter
            if(SmartSense_EMCplus_bIsSensorActive(bIndex))
            {
                // Add user code here to process the sensor touching
            }
            SmartSense_EMCplus_UpdateSensorSignal(bIndex);
}

        // Now we are ready to send all status variables to chart program
        // communication here
        //
        // OUTPUT SmartSense_EMCplus_waSnsResult[x] <- Raw Counts
        // OUTPUT SmartSense_EMCplus_waSnsDiff[x] <- Difference
        // OUTPUT SmartSense_EMCplus_waSnsBaseline[x] <- Baseline
        // OUTPUT SmartSense_EMCplus_baSnsOnMask[x] <- Sensor On/Off
        // OUTPUT SmartSense_EMCplus_baSnsSignal[x] <- Normalized diff count
    }
}
```

示例 3. 该代码除了将传感器 2 和 3 的灵敏度分别更改为 0.2 pF 和 0.4 pF 外，它与示例 1 中的代码完全相同。

```
//-----------------------------------------------------------------------
// Sample C code for the SmartSense_EMCplus module
// The Sensitivity Level parameter of Sensors 2 and 3 should be set
// to 0.2 (Med-High) and 0.4 (low) respectively for each sensor
// in the Sensor Settings Tab of SmartSense_EMCplus Wizard.
// 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)
{
```

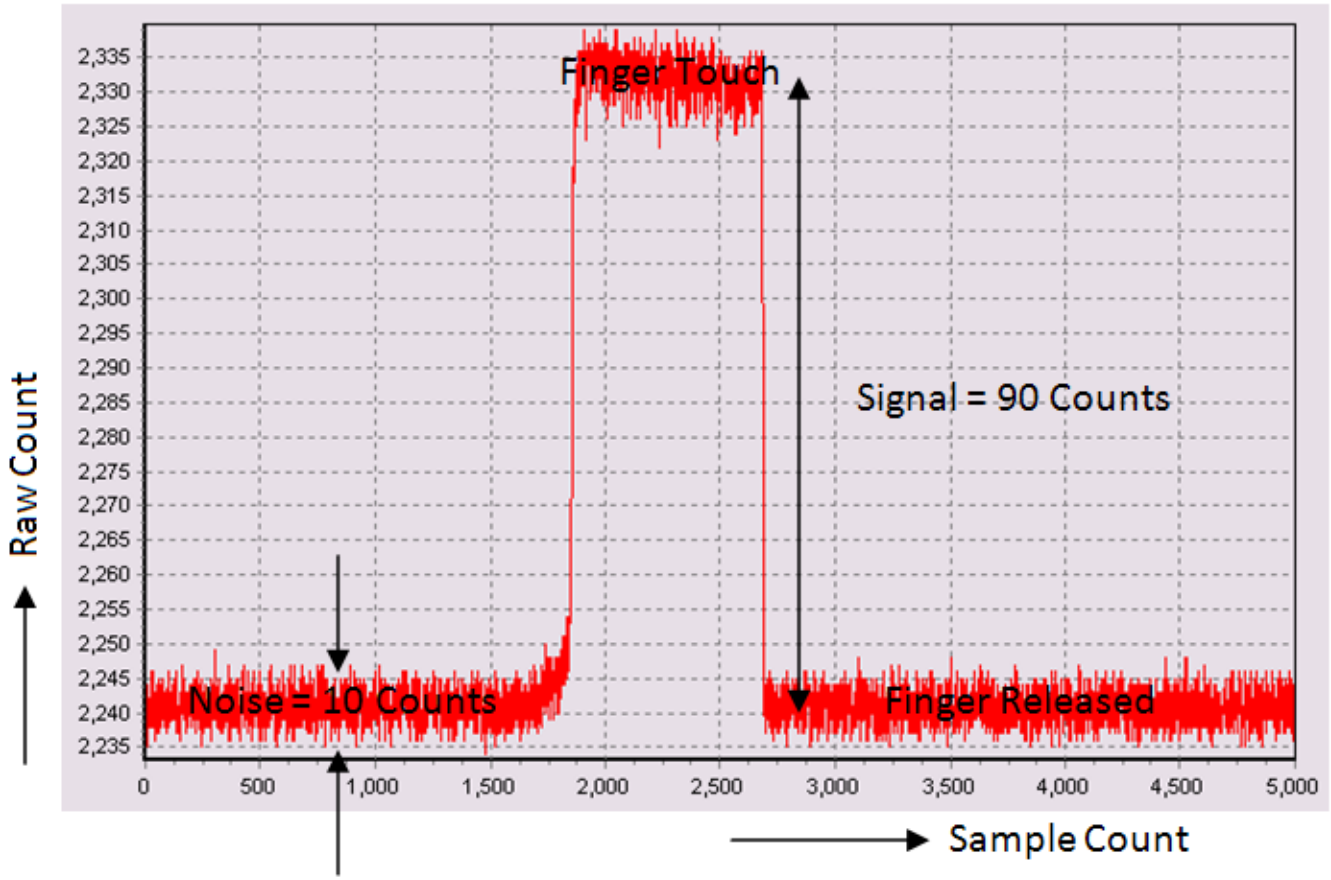
```
M8C_EnableGInt;
    SmartSense_EMCplus_Start();
SmartSense_EMCplus_InitializeBaselines(); // Scan all sensors first time, init baseline
    //
// Loop Forever
//
while (1)
    {
SmartSense_EMCplus_ScanAllSensors(); // Scan all sensors in array (buttons and sliders)
        SmartSense_EMCplus_UpdateAllBaselines(); // Update all baseline levels;

//detect if any sensor is pressed
        if(SmartSense_EMCplus_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 SmartSense_EMCplus_waSnsResult[x] <- Raw Counts
// OUTPUT SmartSense_EMCplus_waSnsDiff[x] <- Difference
// OUTPUT SmartSense_EMCplus_waSnsBaseline[x] <- Baseline
// OUTPUT SmartSense_EMCplus_baSnsOnMask[x] <- Sensor On/Off
}
}
```

如何设置灵敏度和手指阈值的最佳值

SmartSense_EMCplus 是先进的电磁器件设计，该设计基于 CSD 的 SmartSense 用户模块，且不需要繁琐的调校过程。然而，使用 SmartSense_EMCplus 用户模块时，有两个简单的步骤能确保设计的健壮性。

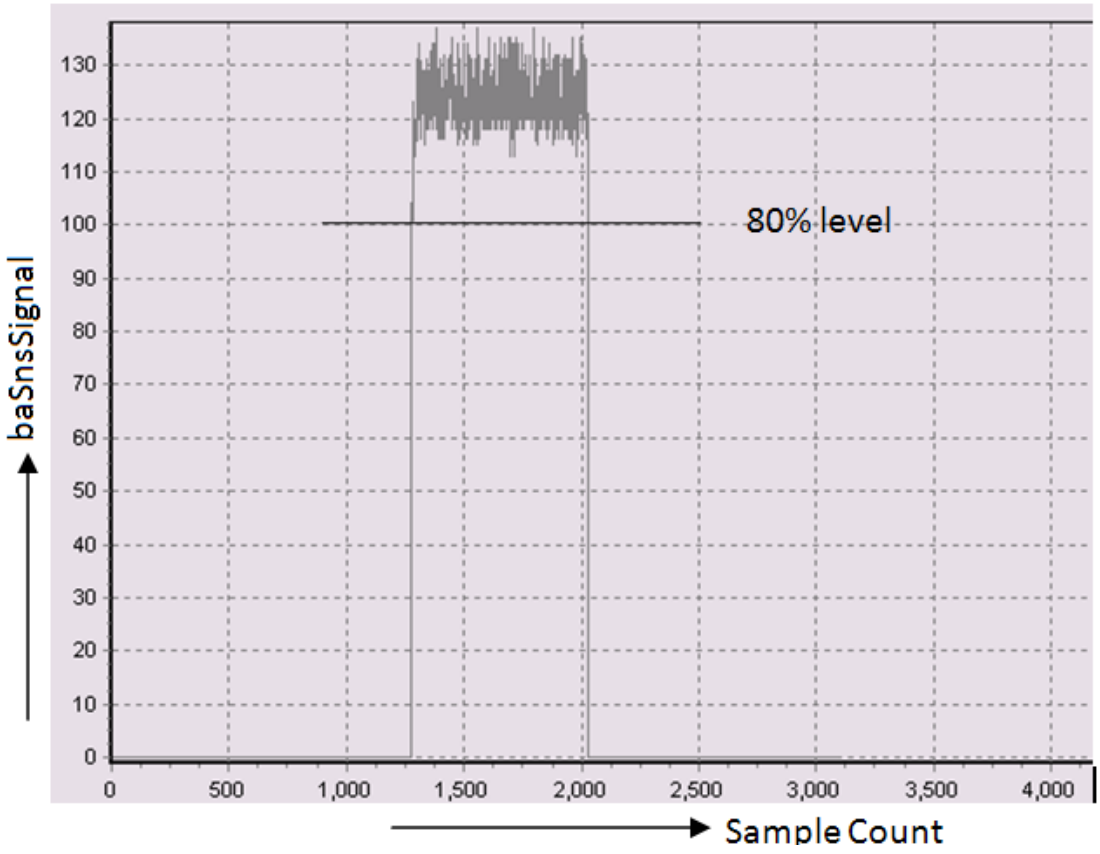
1. 为 CapSense 用户模块参数设置实时监控工具，以便使用本用户模块数据手册中说明的 “ 示例代码 2” 进行测量信噪比。在调校期间，可以观察到传感器原始计数（SmartSense_EMCplus_waSnsResult）、基准线 （SmartSense_EMCplus_waSnsBaseline）、差值计数 （SmartSense_EMCplus_waSnsDiff）、传感器标准化信号（SmartSense_EMCplus_baSnsSignal）和传感器手指阈值 （baBtnFThreshold）。请勿使用 LCD 或其他任何数字显示器来监测数据，因为这些工具运行较慢，无法观察到动态数据。推荐使用的数据监测工具是多图或 I2C USB 桥控制面板。
2. 将灵敏度级别设为 0.4 pF （低），监控传感器原始计数 （SmartSense_EMCplus_waSnsResult），并计算 SNR。图 18 显示了手指触摸时的典型原始计数图。根据 CapSense 最佳实践，强大设计的 SNR 应大于 5:1。如果测量得到的 SNR 小于 5:1，则会使灵敏度的值下降到下一可能的操作，直到获得的 SNR 为 5:1 或更高为止。

3. 手指触摸典型传感器的原始计数图



4. 如果您在设计中使用自动手指阈值，那么上一步 （第二步）完成调校过程。如果设计中采用了灵活的手指阈值，那么需要设置手指阈值，以便完成调校过程。为设置手指阈值，需监测传感器信号（SmartSense_EMCplus_baSnsSignal），并在触摸传感器时，设置手指阈值为传感器信号的 80%。图 19 显示了典型传感器信号和手指阈值。

图 22. 存在手指触摸的典型传感器信号



SmartSense_EMCplus 用户模块的特定指南

所有适用于 SmartSense 用户模块的指南均适用于 SmartSense_EMCplus 用户模块。有关 CapSense 设计和基于 SmartSense 设计的通用说明，请参阅 CapSense 入门指南。

本节介绍 SmartSense_EMCplus 用户模块的几个重要方面。

传感器扫描时间、响应时间和存储器占用

当传感器使用 SmartSense_EMCplus 用户模块时，传感器的扫描时间、响应时间以及 RAM 存储器占用均取决于用户模块中选择的抗噪级别。

- 抗噪级别设置为“中”时传感器扫描时间比设置为“低”时多一倍。抗噪级别设置为“高”时的传感器扫描时间比设置为“低”时多两倍。
- 随着扫描时间增加，传感器的响应时间也同比增加。抗噪级别设置为“中”时的传感器响应时间比设置为“低”时多一倍。同样，抗噪级别设置为“高”时的传感器响应时间比设置为“低”时多两倍。
- 为实现强大的电磁兼容算法，SmartSense_EMCplus 用户模块使用 RAM 存储器。因此，最高抗噪级别（“高”）需要的 RAM 内存大约比级别为“低”时大两倍。中等级别需要的 RAM 内存大约比低抗噪级别时的大一倍。

IMO 容差和时间临界任务

内部主振荡器 （IMO）的容差为 +5% 和 -20%

- 使用时间临界算法和逻辑时，须考虑 IMO 容差，以确保固件逻辑或算法避免破坏。
- 如果您的项目采用了中断，则当分析中断延迟和 ISR 执行时间时须考虑 IMO 的容差和其他因素。
- 每个基于 IMO 的时序分析（例如，IMO 定时器、固件中使用循环而导致的延迟、API 执行时间）必须考虑 IMO 容差，以确保能应用固件的健壮性。

I^2C 工作速度

I2C 接口工作频率最大为 SmartSense_EMCplus - enabled 器件用户模块实际工作频率的 80%。这是由于存在 IMO 的容差。

- 因此，在 I2C 用户模块中选择 400 kHz 的时钟频率时，I^2C 接口的最大工作频率为 320 kHz。同样，当 I2C 用户模块中选择 100 kHz 和 50 kHz 时钟模式时，最大工作频率为 80 kHz 和 40 kHz。
- 使用 I^2C 从接口时，主接口时钟需在前面提到的降频范围内运行。否则会导致数据损坏、I^2C 总线连接或与 I2C 用户模块动作不一致。
- 使用 I^2C 主接口模块只影响该接口的吞吐量。

配置寄存器

表 4. 模块 CapSense、寄存器：CS_CR0

位	7	6	5	4	3	2	1	0
数值	0	0	SmartSense_EMCplus_PRSCLK	0	1	0	0	EN

表 5. 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 （Power）：0x01 — 打开模拟模块的电源。 0x00 — 关闭模拟模块的电源。

表 6. 模块 CapSense、寄存器：CS_CR2

位	7	6	5	4	3	2	1	0
数值	0	0	0	0	0	1	0	0

表 7. 模块 CapSense、寄存器：CS_CR3

模式 / 位	7	6	5	4	3	2	1	0
数值	SHIELD_IO_EN[4]	0	0	0	SHILED_IO_EN[3:0]			

表 8. 模块 CapSense、寄存器：CS_CNTH

位	7	6	5	4	3	2	1	0
数据输出 MSB								

表 9. 模块 CapSense、寄存器：CS_CNTL

位	7	6	5	4	3	2	1	0
数据输出 LSB								

表 10. 模块 CapSense、寄存器：PRS_CR

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

表 11. 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 12. 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 13. 模块定时器、寄存器：PT1_DATA1

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

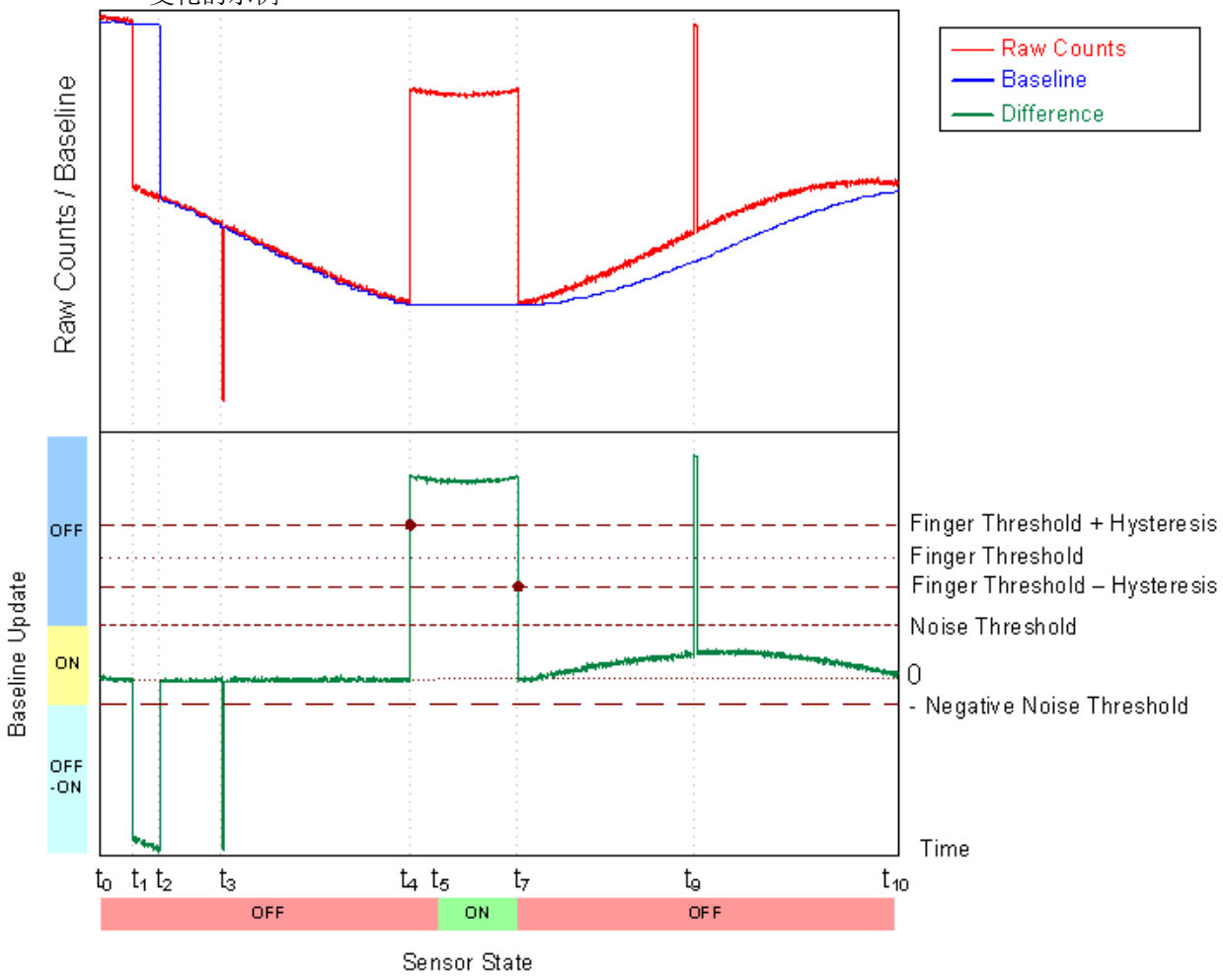
附录

下面各节介绍了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的详细信息，帮助您成功设计 CapSense 应用。该信息的某些部分将来会移到应用笔记中。

SmartSense_EMCplus 参数的交互

下图说明了基准线更新和决策逻辑操作。这有助于更好地了解如何设置用户模块参数以获得最佳性能。图 20 显示了当传感器自动复位参数设置为 Disabled （禁用）时的系统操作。图 21 显示的是被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 – 基准线）。数据通过一些人工测试收集，这些测试展示在低速和高速原始数据变化情况下的系统操作。低速变化可由温度或湿度变化引起，高速变化可由传感器触摸、 ESD 事件或强射频场的影响引起。

图 23. 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，由于湿度或温度变化，原始计数值会接近于基准线级别，并开始缓慢下降。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 ” 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值，从而可以保留等于原始计数信号较小值的值。

在 t1 处，原始计数值快速下降，负差值超过负噪声阈值。如果手指位于传感器上时为器件加电，经过一段时间手指被移开，则会发生这种情况。此时，基准线更新机制被冻结，而内部超时计数器则被激活。当差值信号低于 “ 低基准线复位 ” 样品的 “ 负噪声阈值 ” 时，会复位基准线。这是在 t_2 处发生的。

第二大负差信号尖峰位于 t_3 处。这一尖峰可能是由 ESD 事件等引起的。由于采样计数中的尖峰脉冲持续时间小于 “ 低基准线复位 ” 参数，因此保留基准线，并对尖峰脉冲进行滤波。这样可以防止假基准线复位及导致假触摸检测。

传感器是在 t_4 处被触摸的。当差值信号超过手指阈值和迟滞之和时，会激活内部防抖动计数器。如果信号超过此值的量大于去抖动样本数量，则传感器状态设置为开启。这是在 t5 处发生的。当差值信号在 t7 处下降到低于手指阈值和迟滞之差时，传感器将立即恢复为关闭状态。在 t_9 处短的正尖峰脉冲已被防抖动计数器滤波，这是因为样本单元中的尖峰脉冲持续时间不会超过抖动值。

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 （传感器自动复位设置为 “ 禁用 ”），并且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 BaselineUpdate 阈值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 24. 传感器自动复位参数设置为 “ 使能 ” 时的原始计数、基准线、差值信号的变化示例

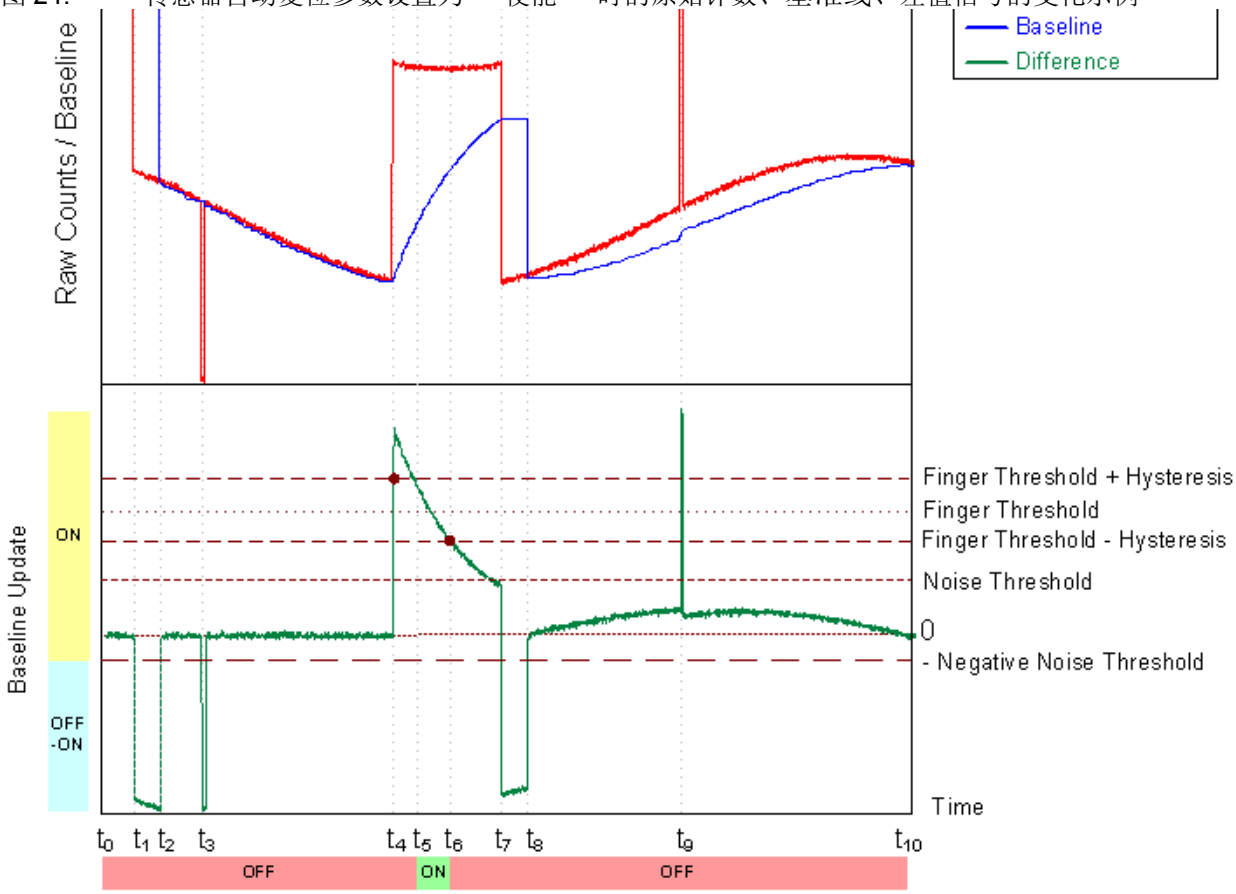


图 21 中的系统操作与上一个实例中的操作类似，但有以下区别：

- 在 t_6 处触摸传感器时，触摸持续时间因活动基准线更新算法而会减少。
- 抬起手指后，基准线会在低基准线复位采样 （t_8）后复位，这样会短期阻止触摸检测。这可作为附加的去抖动机制。

版本历史记录

版本	修订人	说明
1.00	DHA	初始版本。
1.10	DHA	1. 将灵敏度级别属性的默认值更新为 0.4。 2. CSD 模式被禁用，并且 PRS 时钟源位被复位在 SmartSense_EMCplus_Stop API 中。 3. 添加了对 CY8C200x5 器件的支持。
1.20	MYKZ	1. 纠正了有关保存滑条信息的问题。 2. 更新了基准线算法，用于检查负差值计数。 3. 将 GetSnsParasiticCapacitance 添加到用户模块 API。 4. 添加了编译时的错误信息，以便在不调用向导时可防止用户模块的使用。 5. 实现 InitializeSensorBaseline() 函数中的大型存储器模式。 6. 初始化了用户模块向导中 GlobalSettingsProxy 属性的默认值。

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。

Copyright © 2012-2014 赛普拉斯半导体公司。此处所包含的信息可能会随时更改，恕不另行通知。除赛普拉斯产品内嵌的电路外，赛普拉斯半导体公司不对任何其他电路的使用承担任何责任。也不会根据专利权或其他权利以明示或暗示的方式授予任何许可。除非与赛普拉斯签订明确的书面协议，否则赛普拉斯产品不保证能够用于或适用于医疗、生命支持、救生、关键控制或安全应用领域。此外，对于合理预计会发生运行异常和故障并对用户造成严重伤害的生命支持系统，赛普拉斯将不批准将其产品用作此类系统的关键组件。若将赛普拉斯产品用于生命支持系统，则表示制造商将承担因此类使用而招致的所有风险，并确保赛普拉斯免于因此而受到任何指控。

PSoC Designer™ 和 Programmable System-on-Chip™ 是赛普拉斯半导体公司的商标， PSoC® 是赛普拉斯半导体公司的注册商标。此处引用的所有其他商标或注册商标归其各自所有者所有。

所有源代码 （软件和 / 或固件）均归赛普拉斯半导体公司 （赛普拉斯）所有，并受全球专利法规 （美国和美国以外的专利法规）、美国版权法以及国际条约规定的保护和约束。赛普拉斯据此向获许可者授予适用于个人的、非独占性、不可转让的许可，用以复制、使用、修改、创建赛普拉斯源代码的派生作品、编译赛普拉斯源代码和派生作品，并且其目的只能是创建自定义软件和 / 或固件，以支持获许可者仅将其获得的产品依照适用协议规定的方式与赛普拉斯集成电路配合使用。除上述指定用途外，未经赛普拉斯的明确书面许可，不得对此类源代码进行任何复制、修改、转换、编译或演示。

免责声明：赛普拉斯不针对该材料提供任何类型的明示或暗示保证，包括 （但不仅限于）针对特定用途的适销性和适用性的暗示保证。赛普拉斯保留在不另行通知的情况下对此处所述材料进行更改的权利。赛普拉斯不对此处所述之任何产品或电路的应用或使用承担任何责任。对于合理预计可能发生运转异常和故障，并对用户造成严重伤害的生命支持系统，赛普拉斯不授权将其产品用作此类系统的关键组件。若将赛普拉斯产品用于生命支持系统，则表示制造商将承担因此类使用而导致的所有风险，并确保赛普拉斯免于因此而受到任何指控。

产品使用可能受适用于赛普拉斯软件许可协议的限制。