CapSense® 逐次逼近数据手册 CSA V 1.40

Copyright © 2006-2011 Cypress Semiconductor Corporation. All Rights Reserved.

资源	PSoC® 模块			API 内存 （字节）		引脚 （每个外部 I/O）
	I2C/SPI	CapSense	定时器	闪存	RAM	
CY8C20x34, CY8C20x24	-	1	-	1198	129	1

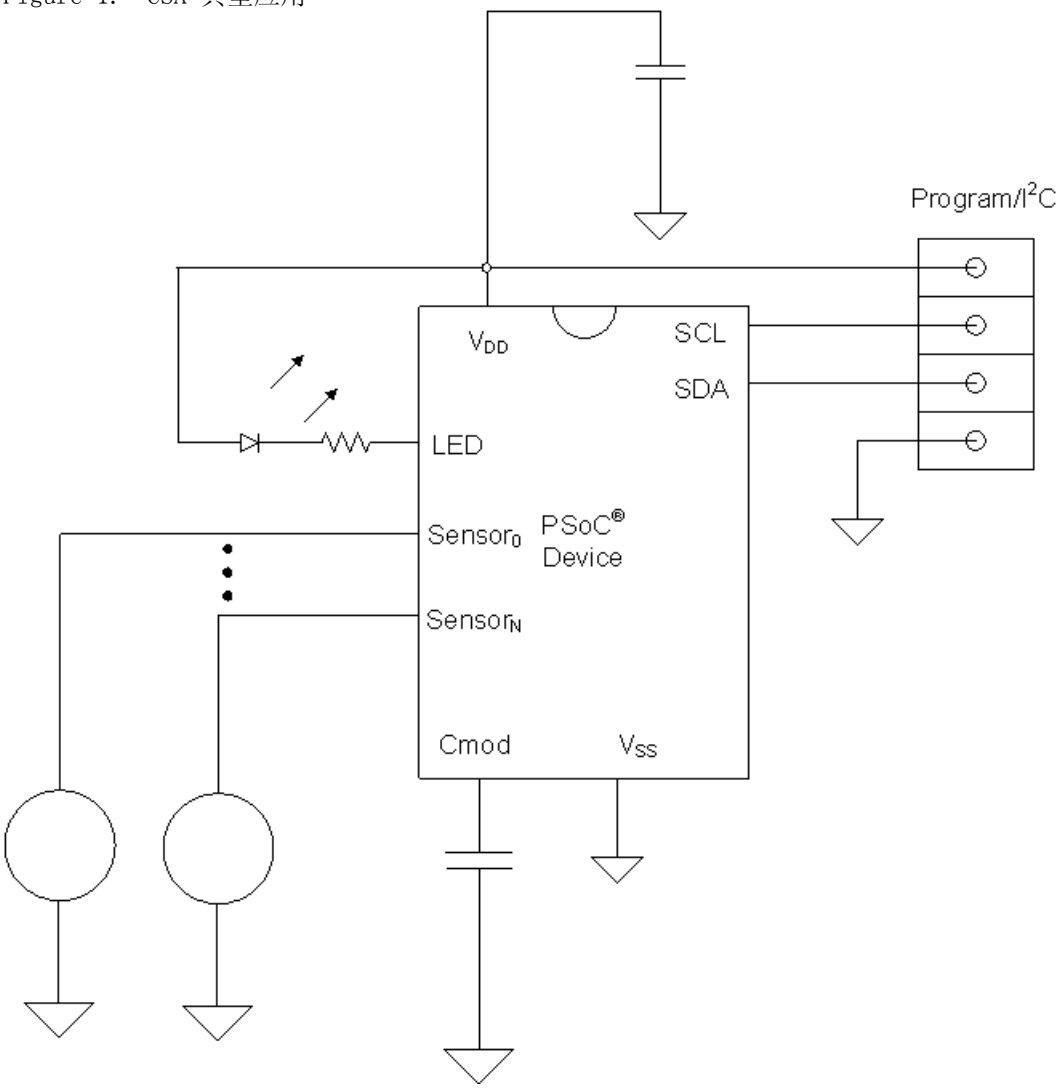
若要获取使用此用户模块的一个或多个完全配置的功能示例项目，请访问 www.cypress.com/psocexampleprojects.

功能和概述

- 扫描 1 到 28 个电容式传感器
- 扫描由 2 到 28 个元件构成的电容性滑条
- 通过双工法，使滑条的物理分辨率翻倍
- 滑条插值分辨率可达 65535 分之 1
- 采用多个滑条传感器构成触摸板
- 传感器灵敏度、检测阈值和采样率可调
- 使用 CSA 向导在指导下分配传感器 / 引脚
- 集成了能够处理温度变化的基准线更新算法
- 环境和物理传感器变动补偿

CapSense® 逐次逼近 (CSA) 用户模块使用开关式电容电路、一个模拟复用器、数字计数功能和用于补偿环境和物理传感器变动的高层软件例程，实现了一个电容性触摸传感器阵列。 传感器阵列可组合使用独立传感器、滑条传感器和利用一对正交滑条传感器实现的触摸板。 高级软件例程负责处理滑条双工。 滑条双工使得一个引脚能够测量两个不同物理位置处的两个电气传感器。 双工可增强滑条的分辨率，而无需另外增加一个 IO。

Figure 1. CSA 典型应用



快速启动

1. 使用时请选择并布置需要专用引脚 （如 I^2C 或 LCD）的用户模块，并分配端口和引脚。
2. 选择和布置 CSA 用户模块。
3. 右键单击 CSA 用户模块访问 CSA 向导。
4. 设置按键传感器个数、滑条配置以及引脚分配和关联项。
5. 设置引脚和全局用户模块参数。
6. 生成应用并切换到应用编辑器。
7. 改编样本代码以实现按键、滑条或触摸板。

[+] Feedback

功能说明

电容式传感器由物理、电气和软件组件构成。

CSA 用户模块所测量的每个传感器都是一个电容，其一端接地，另一端连接到一个 PSoC 引脚。 导体的存在会增加接地传感器的电容。 这种电容变化可控制传感器的激活。

直流和交流电气特性

除非另有说明，否则 Vdd = 2.7V 到 5.0V，C_{mod} = X7R 型电容，容差 ±20%。

Table 1. CSA 用户模块的电气规范

符号	说明	条件	最小值	典型值	最大值	单位
C_P	寄生电容[a]	请参见 C_P 范围相关注释[b]	5		50	pF
C_F	手指电容[c]		0.1			pF
N	输出计数器分辨率		16		16	位
S_{FINGER}	手指灵敏度[d]	C_P = 5 到 15 pF IDAC 设置 = 10 建立时间 = 255 C_{mod} = 3300 pF CSA 时钟 = 6 MHz IMO = 12 MHz CPU = 3 MHz	500			计数 /pF
		C_P = 9 到 27 pF IDAC 设置 = 10 建立时间 = 220 C_{mod} = 5600 pF CSA 时钟 = 6 MHz, IMO = 12 MHz CPU = 3 MHz	500			计数 /pF
		C_P=16 到 50 pF IDAC 设置 = 10 建立时间 = 220 C_{mod} = 0.01 uF CSA 时钟 = 6 MHz IMO = 12 MHz CPU = 3 MHz	500			计数 /pF

[+] Feedback

符号	说明	条件	最小值	典型值	最大值	单位
t_C	转换时间，单个传感器。	C_P = 5 到 15 pF IDAC 设置 = 10 建立时间 = 255 C_{mod} = 3300 pF CSA 时钟 = 6 MHz IMO = 12 MHz CPU = 3 MHz			1300	µs/ 传感器
		C_P = 9 到 27 pF IDAC 设置 = 10 建立时间 = 220 C_{mod} = 5600 pF CSA 时钟 = 6 MHz IMO = 12 MHz CPU = 3 MHz			1300	µs/ 传感器
		C_P = 16 到 50 pF IDAC 设置 = 10 建立时间 = 220 C_{mod} = 0.01 µF CSA 时钟 = 6 MHz IMO = 12 MHz CPU = 3 MHz			2700	µs/ 传感器
I_{DDCS}	平均供电电流	Vdd = 3.3V C_P = 5 到 15pF 4 按键扫描 100 ms 报告速率		35	50	µA
R_S	抗射频干扰的串联电阻[e]	长度超过 25 mm 的高导电性走线 （铜或银墨）	300		560	欧姆

a. C_P 包括 2-3 pF 的封装相关电容。

b. 手指灵敏度和转换时间按三个 C_P 范围之一指定：(5-15 pF)、(9-27 pF)、(16-50 pF)

c. 手指电容是由手指触摸传感器引起的 C_P 的增加。

d. 选择 IDAC 设置以使手指触摸产生至少 50 个计数的信号。

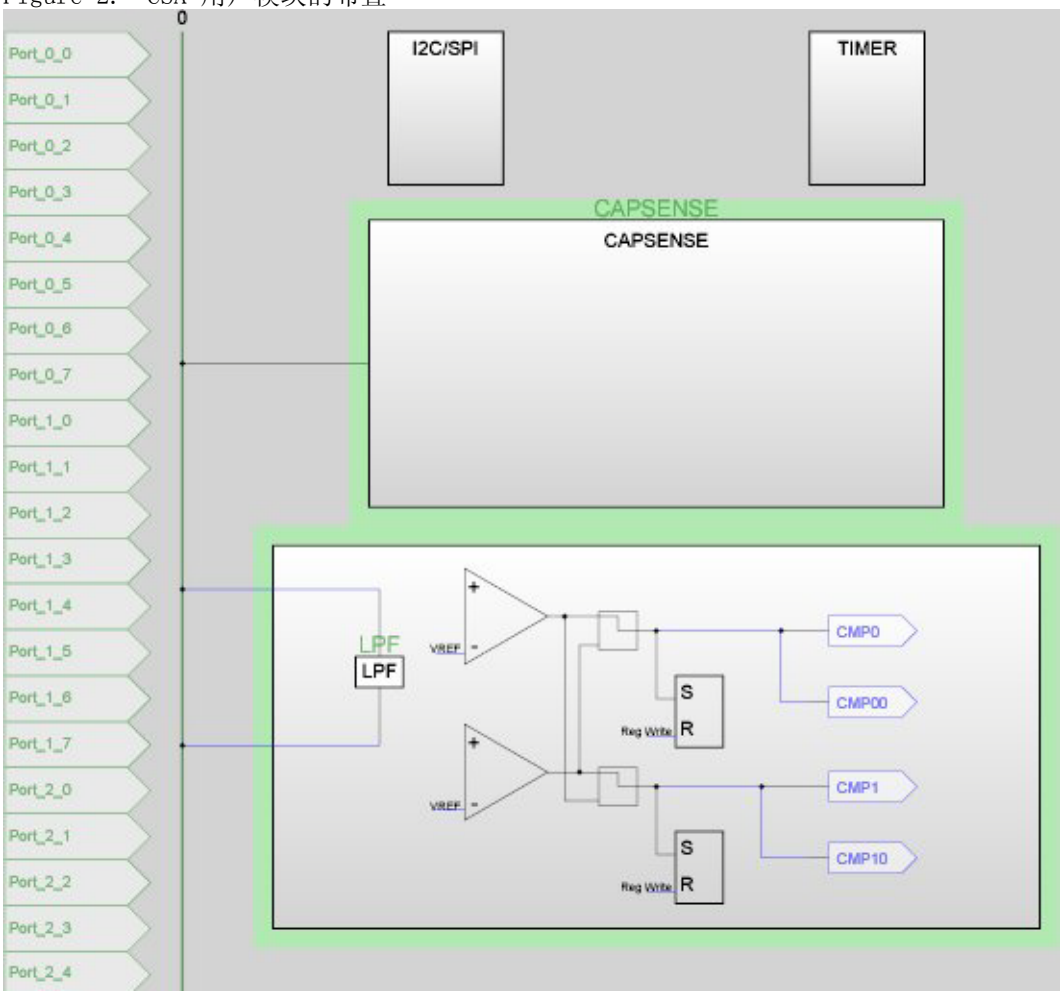
e. 较薄的低导电性 ITO 薄膜无需 R_S。 电阻放在 PSoC 引脚的 10 mm 范围内。

[+] Feedback

放置

用户模块所用的各模块将在用户模块实例化后自动放置，而没有提供备选放置方式。 使用特定引脚资源（包括 LCD 和 EZI2C）的用户模块必须在为 CSA 用户模块建立端口引脚连接之前进行放置。 打开向导时，这些选择会反映在向导中。

Figure 2. CSA 用户模块的布置



在布置电容式传感器连接时，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，从而降低传感器的性能。

[+] Feedback

应用信息

本小节将逐步说明如何使用 CSA 用户模块设计一个电容感应系统。 我们假定 PSoC 器件使用的电源为 Vdd = 2.7V 到 5.0V，且 C_{mod} 是一个 X7R 型电容。

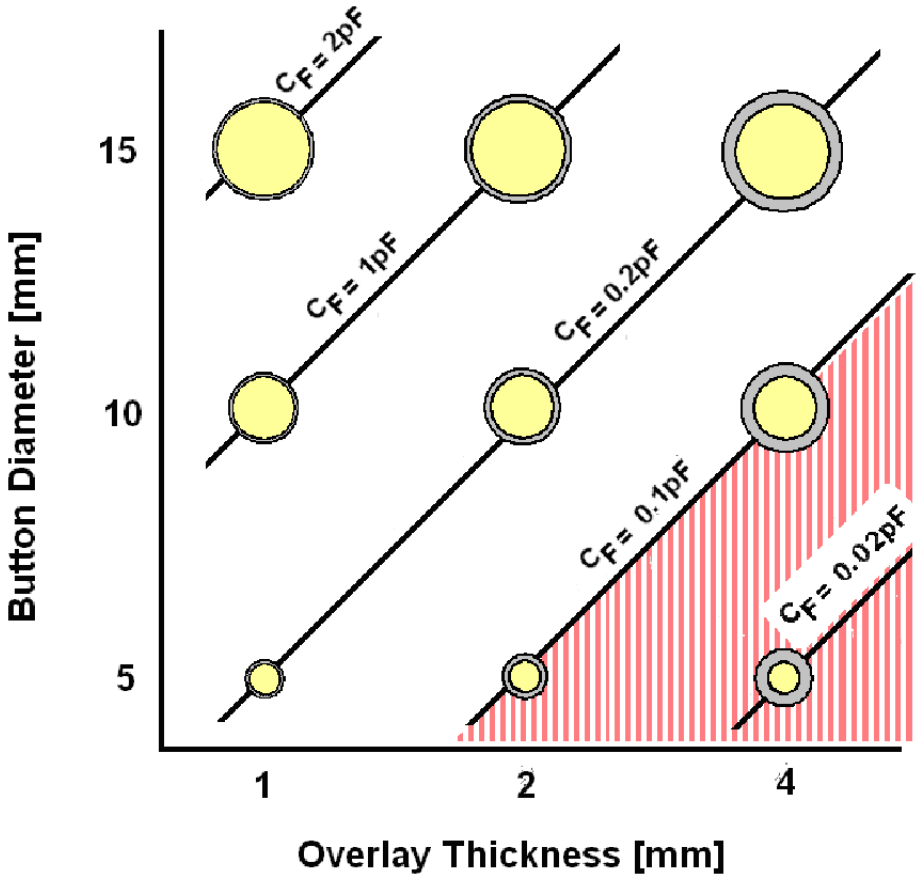
1. **按键大小：** 使用图 3 中的图示选择针对特定外覆层厚度可产生正确水平的手指电容的按键大小。 最小手指电容 C_F 为 0.1 pF，但建议采用 0.2 pF 以获得额外的设计余量。 按键与接地填充物之间的间隙等于外覆层厚度。

   板设计与外覆层是否均满足了手指电容的最低要求？

   如果回答为 “ 否 ”，请使用薄一些的外覆层，或增加传感器的大小。

   如果回答为 “ 是 ”，请转到第 2 步。

Figure 3. 按键直径与外覆层厚度



示例： 塑料外覆层为 2 mm 厚。 设计目标为 C_F = 0.2 pF。 显图 3 示按键直径需为 10 mm，并且传感器板与接地填充物之间的间隙为 2 mm。

[+] Feedback

2. **CP 位于 5 到 50 pF 范围内：** 检查将由 CSA 用户模块监控的 C_P 的范围。 使用经验法则估算每个 PSoC 引脚上的电容负载，即在具有 8mil 走线的 63mil 厚的 2-layer 板上，走线电容大约为 2 pF/ 英寸。

   是否每个传感器都位于 5 pF 到 50 pF 的 C_P 范围内？

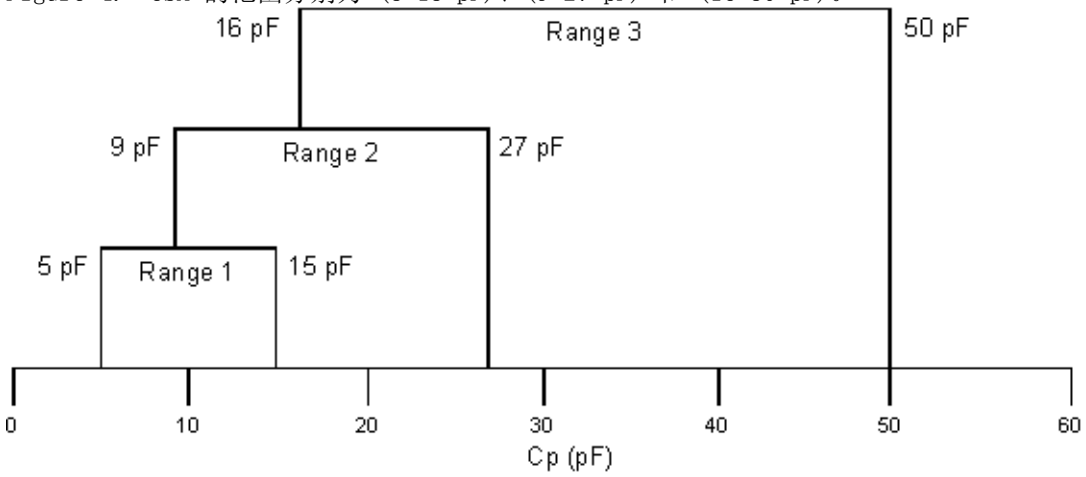
   如果回答为 “ 否 ”，请将传感器板更靠近 PSoC 器件一些，或考虑使用另一种 CapSense 方法，如 CSD。 任何长于 24″ 的走线都将超过 50pF 限值。

   如果回答为 “ 是 ”，请转到第 3 步。

   示例： 系统有 12 个触摸传感器。 所有传感器的传感器板到 PSoC 引脚的距离都小于 4″。 这些传感器的 PCB 空板的 C_P 范围为 9 pF 到 18 pF。 考虑到 PSoC 器件的封装效应（增加 3 pF），PSoC 引脚上的总负载在 12 pF 到 21 pF 之间，可轻松满足 5-50 pF 的 C_P 限值要求。

3. **CP 范围：** 手指灵敏度和转换时间按 C_P 范围中的一个范围指定，如图 4 所示。5-15 pF、9-27 pF 和 16-50 pF。

Figure 4. CSA 的范围分别为 (5-15 pF)、(9-27 pF) 和 (16-50 pF)。



   示例： PSoC 引脚上的总负载在 12 pF 到 21 pF 之间。 最适合这些值的 C_P 范围为范围 2，即 9 pF 到 27 pF。

[+] Feedback

4. **差值和阈值：** 估算差值，并设置手指阈值和噪声阈值参数。 差值是由手指触摸传感器引起的计数增加。 差值可使用手指灵敏度 S_{FINGER} 和手指电容 C_F 按 用公式 1 计算得出。 。公式 2

Equation 1

$$DifferenceCounts = S_{Finger} \times C_F$$

Equation 2

$$FingerThreshold = 0.75 \times DifferenceCounts$$

Equation 3

$$NoiseThreshold = 0.5 \times DifferenceCounts$$

计算这三个值并转到第 5 步。

示例： 系统包含下列参数。

C_P = 10 至 22 pF，C_F = 0.2 pF

IDACSetting = 7

SettlingTime = 245

C_{mod} = 2700 pF，X7R (+/-20%)

CSA 时钟 = 6 MHz

IMO = 12 MHz

CPU = 6 MHz

解出差值、手指阈值和噪声阈值。

差值 = 500 计数/pF * 0.2 pF = 100 计数

手指阈值 = .75 * 100 = 75 计数

噪声阈值 = .5 * 100 = 50 计数

5. **扫描时间：** 使用转换时间 t_C 按公式 4 估算扫描所有传感器的扫描时间 t_{SCAN}。每个传感器的转换时间为 900 µs。

Equation 4

$$t_{SCAN} = t_C \times (number of sensors)$$

计算此值并转到第 6 步。

示例： 系统在第 4 步定义了 12 个传感器。

tSCAN = 900 ms/传感器 * 12 个传感器 = 10.8 ms

[+] Feedback

6. **平均供电电流：** 使用有功电流 I_{active} 和睡眠电流 I_{sleep} 以及报告速率按公式 5 估算平均供电电流 I_{DDCS}。

Equation 5

$$I_{DDCS} = [t_{SCAN} \times I_{active} + (ReportRate - t_{SCAN}) \times I_{sleep}] / (ReportRate)$$

示例： 从第 5 步继续，添加下列参数。

ReportRate = 100 ms

I_active = 1.13 mA

I_sleep = 2.6 mA

解出平均供电电流 IDDSC。

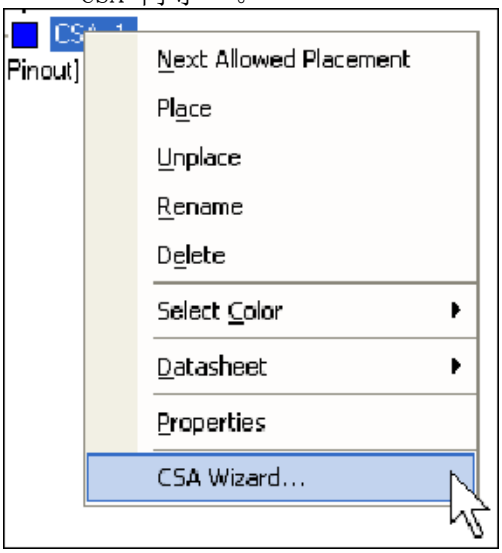
I_{DDCS} = [10.8 ms * 1.13 mA + 89.2 ms * 2.6 mA] / 100 ms = 124 mA

7. **串联电阻：** 确定抗射频干扰所需的串联电阻。
示例： PCB 走线为铜质，长度超过 25 mm。 在这些条件下，建议为所有 CapSense 输入使用 560Ω 的串联电阻。

向导

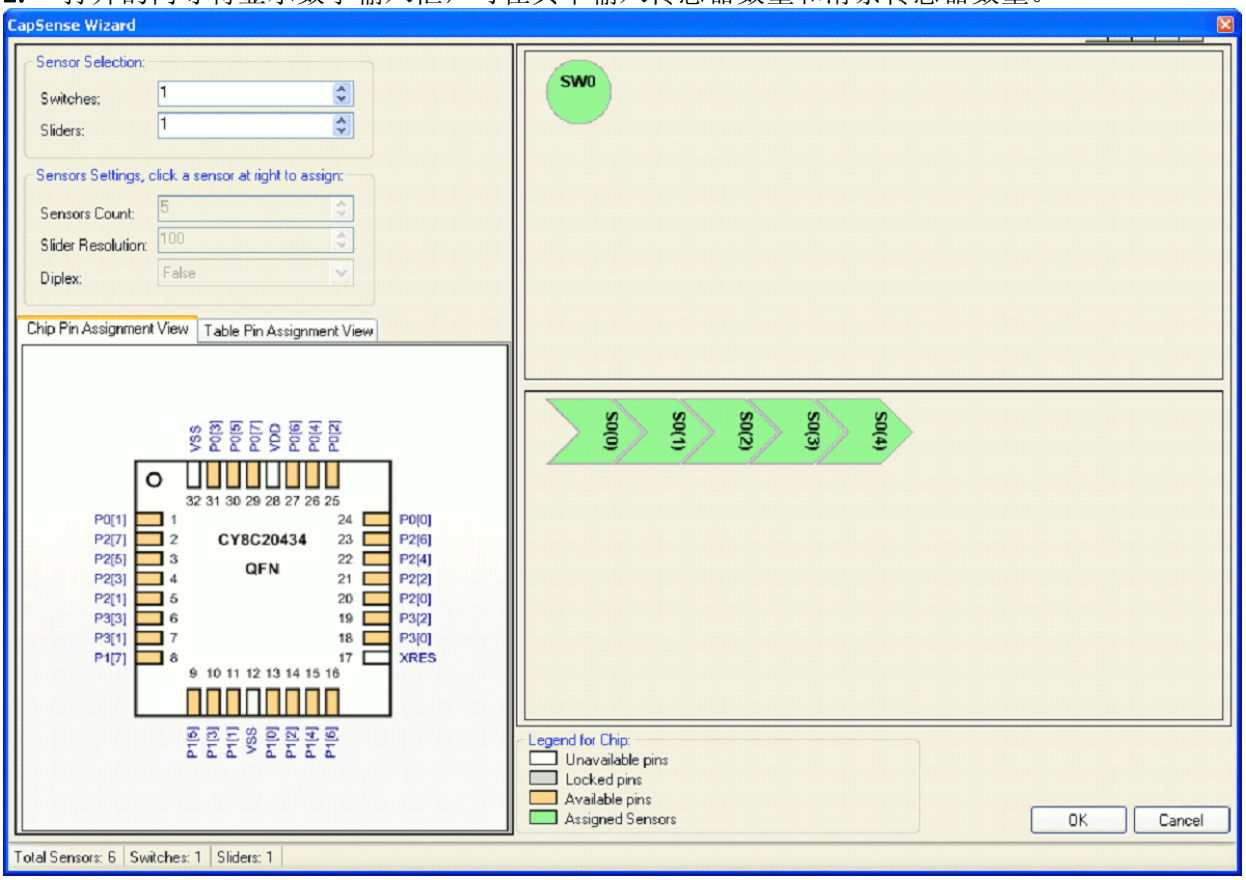
CSA 向导用于设置 CapSense 按键、滑条和接近传感器的引脚分布。 可以选择所需的配置，使用拖放界面来分配按键和段。

1. 要访问向导，请在 “ 器件编辑器互连视图 ” 中右键单击任意 CSA 模块，然后单击鼠标左键选择 “CSA 向导 ”。



[+] Feedback

2. 打开的向导将显示数字输入框，可在其中输入传感器数量和滑条传感器数量。



向导引脚图示

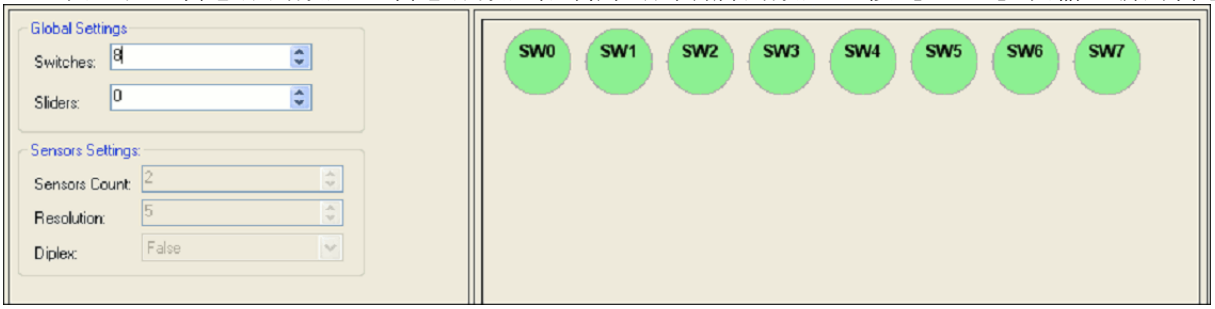
白色 - 引脚不可用作 CapSense 输入端。

灰色 - 引脚已锁定。 产生此情况的可能原因有两种。 第一种可能性是另一个用户模块，例如 LCD 或 I^2C 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**。 现在可在向导中分配引脚了。

橙色 - 引脚可分配。

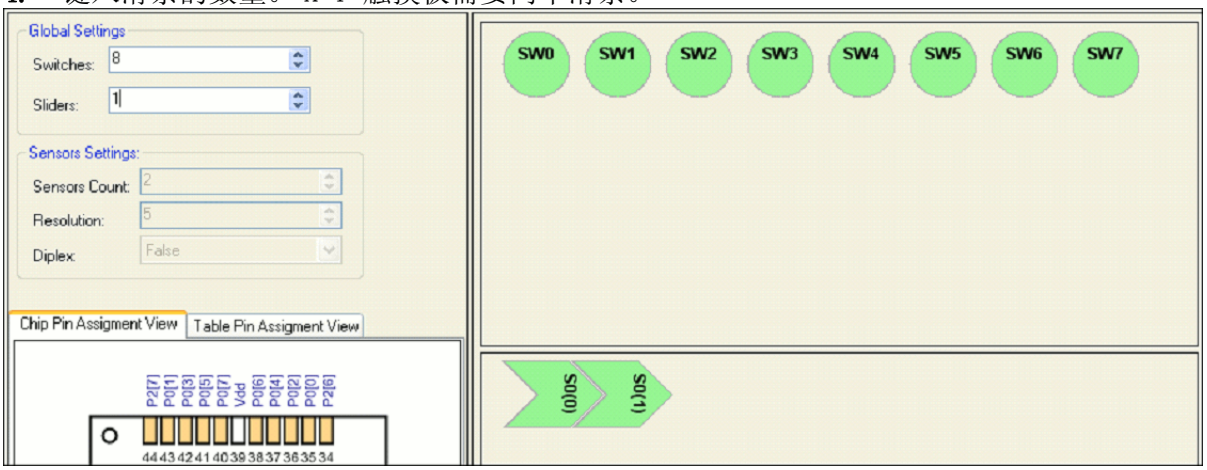
绿色 - 引脚已分配为 CapSense 输入端。

3. 键入独立传感器的数量。 传感器数量限制为可用引脚的数量。 按 [Enter] 键输入新的传感器数量值。



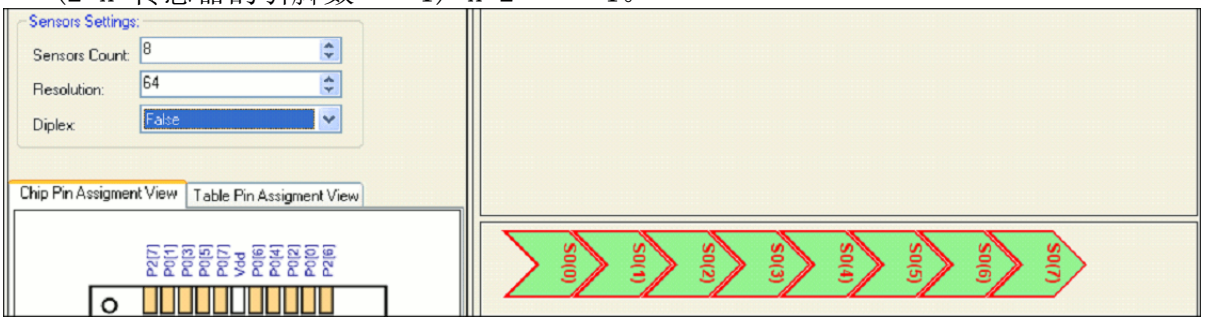
[+] Feedback

4. 键入滑条的数量。 X-Y 触摸板需要两个滑条。

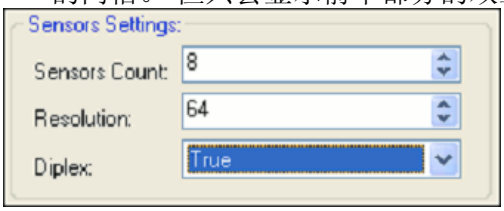


5. 单击滑条以启用传感器设置。 键入每个滑条中传感器元件的数量。 滑条传感器中的传感器实际最小数量为五，最大值受限于引脚数量。 输入数据后，按 [Enter] 键输入新值。

6. 键入输出分辨率。 最小值为五。 对于双工型滑条，最大值 = （传感器的引脚数 - 1） x 2^{16} - 1 或 (2 x 传感器的引脚数 - 1） x 2^{16} - 1。

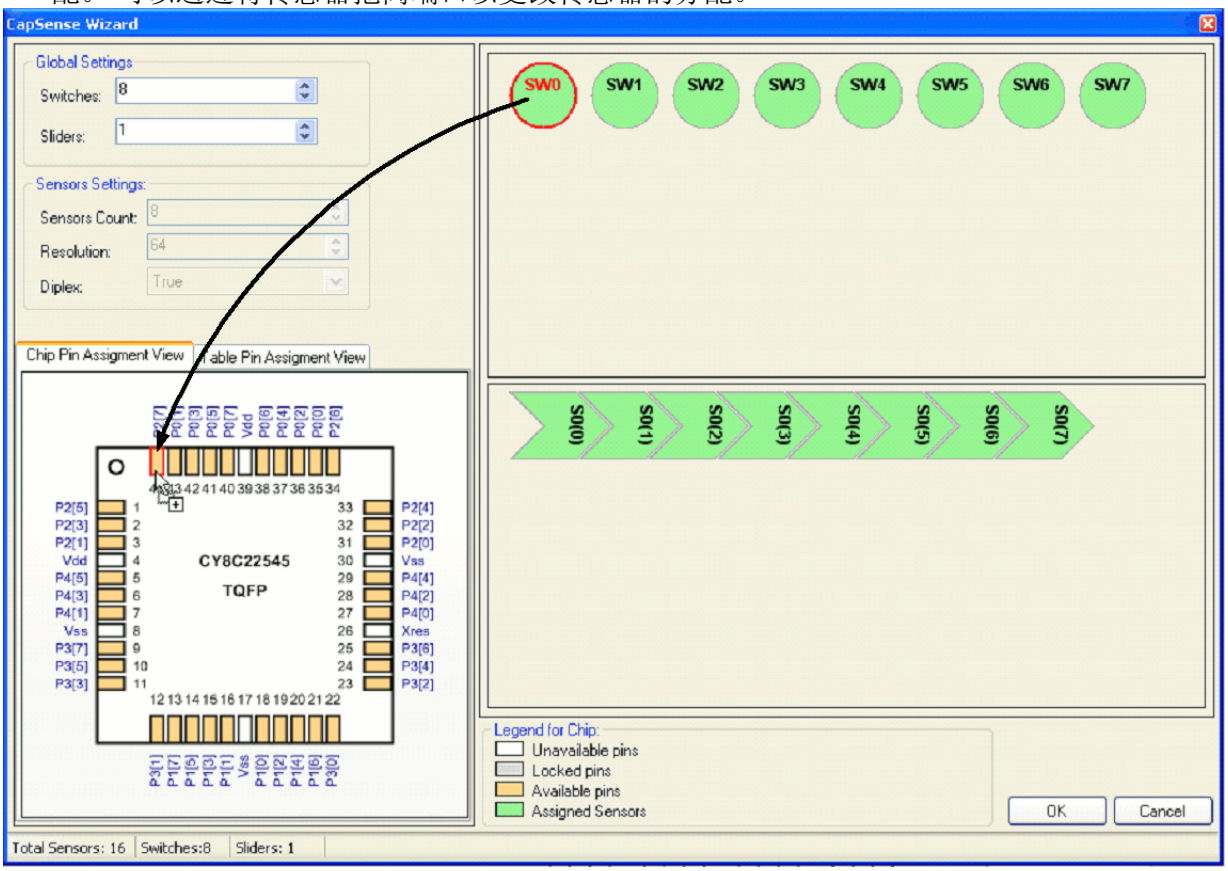


7. 如果需要，请选择 “ 双工 ”。 这样会将选定用于传感器的引脚数量映射成电路板上传感器位置数量的两倍。 但只会显示前半部分的双工传感器。 请参见 AN2292 查看引脚连接的双工表。



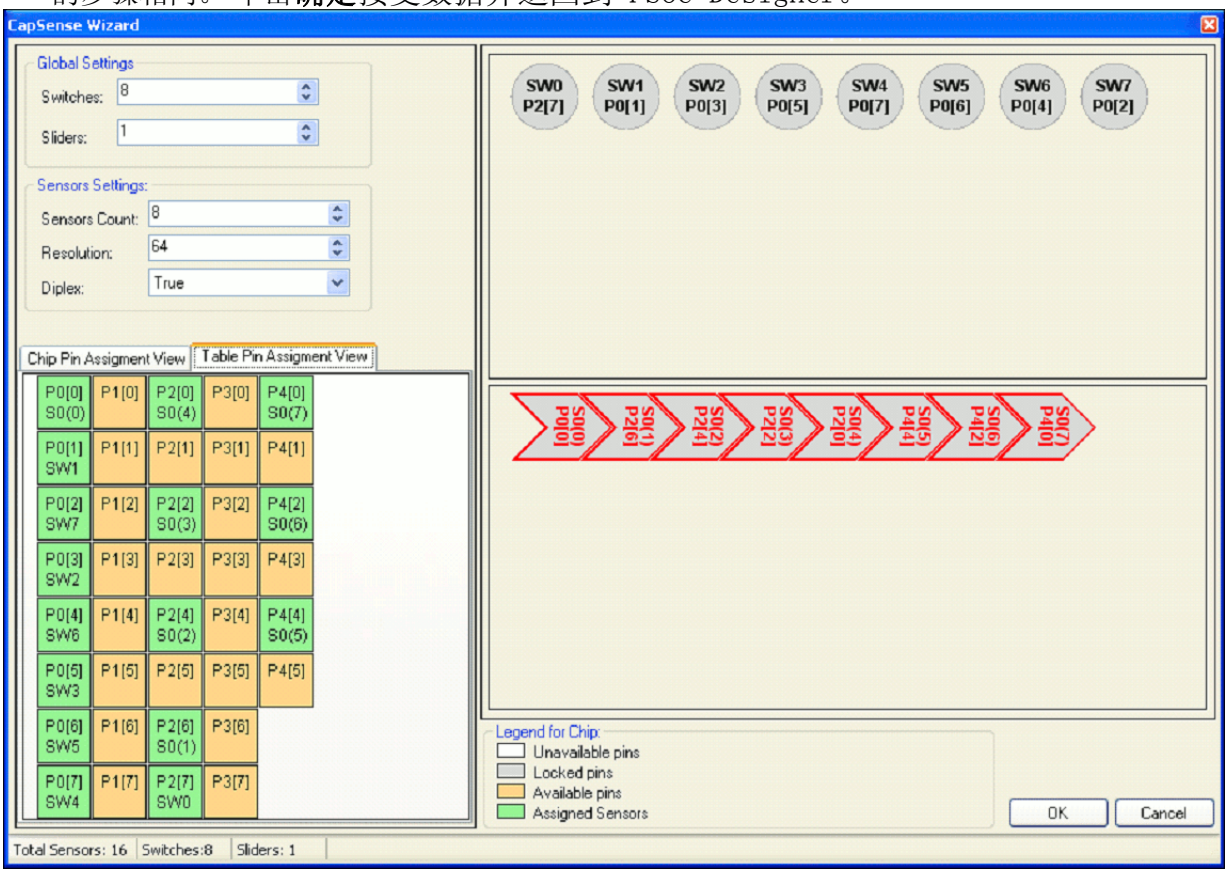
[+] Feedback

8. 在 “ 引脚分配视图 ” 中将开关或传感器拖到引脚上，以此将开关或传感器分配到各个引脚。 您可以选择在 “ 芯片引脚分配视图 ”（Chip Pin Assignment View）或 “ 表引脚分配视图 ”（Table Pin Assignment View）中将开关或传感器拖放到引脚上。 端口引脚在选定后将变为绿色，无法再用于分配。 可以通过将传感器拖离端口以更改传感器的分配。



[+] Feedback

9. 对于其余独立传感器，重复此操作即可。 将单个滑条传感器映射到物理端口引脚的操作与单个传感器的步骤相同。 单击**确定**接受数据并返回到 PSoC Designer。



这时就完成了传感器放置操作。 在 “ 器件编辑器 ” 窗口中单击鼠标右键，选择**刷新**更新引脚连接。

设置用户模块参数并生成应用程序。 如果愿意，可以立即对示例项目进行调整。

要更改引脚分配情况，请将光标置于已分配的引脚上，单击此引脚，然后将其拖到开关框外。 此时引脚便会处于未分配状态，您可以重新分配。

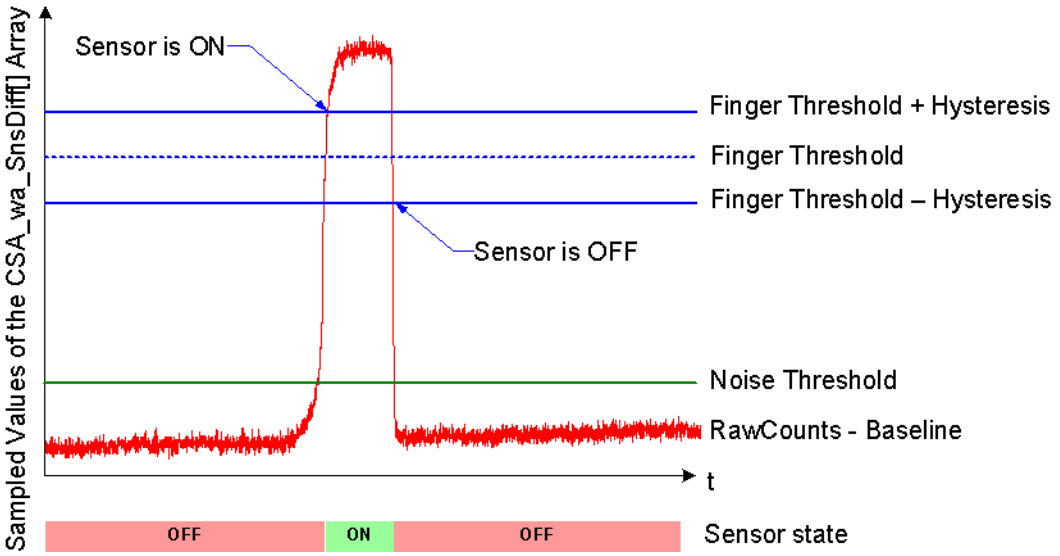
完成向导后，单击 “ 生成应用程序 ”。 向导将根据您输入的传感器数量、引脚分配、双工和分辨率，生成一组表格。 这些表格位于 CSD_Table.asm 中

[+] Feedback

参数和资源

用户模块参数的最佳设置取决于电路板布局和按键尺寸。 请参考 的应用信息参数值。

Figure 5. 基于差值波形的 CSA 参数术语



手指阈值

此阈值使用差值波形来决定每个按键传感器的状态。 如果 wa_SnsDiff[] 阵列中存储的差值达到或超过手指阈值，传感器将激活。

可能值的范围为 3 到 255。

噪声阈值 (Noise Threshold)

对于单个传感器，此参数设置一个计数值，超出此值时将不更新基准线。

对于滑条传感器，它设置一个计数值，那些低于它的扫描结果将会在计算质心的时候被忽略。

可能值为 3 到 255。

基准线更新阈值 (Baseline Update Threshold)

当新的原始计数值超过当前基准线且差值低于噪声阈值时，当前基准线与原始计数之间的差值将累计到一个 “ 水桶 ” 中。 当水桶已满时，基准线将递增并清空水桶。 此参数设置了基准线递增时水桶必须要达到的阈值。

可能值为 0 到 255。

建立时间 (SettlingTime)

“ 建立时间 ”(SettlingTime) 参数控制等待 C_{mod} 电容上的电压稳定下来的软件延迟。 该循环每个迭代有 9 个 CPU 周期。 选择一个至少为 5 × R × C 的 “ 建立时间 ”(SettlingTime)，其中 R = 1 ÷ （时钟频率 × C_P），C = C_{mod}。

可能值为 2 到 255。

IDAC 设置 (IDACSetting)

此参数控制 C_{mod} 上 ADC 斜坡电压的斜率。 设置较低时，电压缓慢上升，计数值较大。 设置较高时，电压快速上升，计数值较小。

可能值为 4 到 255。

[+] Feedback

外部电容 (ExternalCap)

迟滞 (Hysteresis)

“ 迟滞 ”(Hysteresis) 参数会与手指阈值相加或相减，具体取决于传感器当前处于活动还是非活动状态。 如果传感器关闭，则差值必须超过手指阈值与迟滞之和。 如果传感器开启，则差值必须低于手指阈值与迟滞之差。 此参数用于增加手指检测算法的平稳性和 “ 牢固性 ”。

可能值为 0 到 255。 但是，该设置必须低于 “ 手指阈值 ”(FingerThreshold) 参数设置。

防反跳 (Debounce)

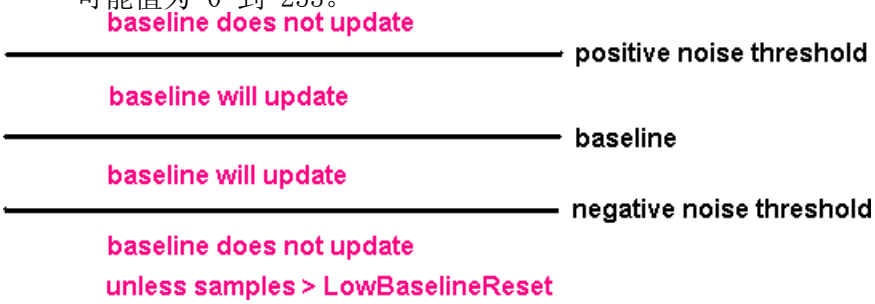
“ 防反跳 ”(Debounce) 参数为传感器的活动状态切换添加了一个防反跳计数器。 传感器要想从非活动状态切换到活动状态，差值必须在指定的采样数内保持超过手指阈值与迟滞之和。

可能值为 1 到 255。 设置为 1 则不提供防反跳。

负噪声阈值 (NegativeNoiseThreshold)

“ 负噪声阈值 ”(NegativeNoiseThreshold) 参数会添加一个负的差值阈值。 如果当前原始计数低于基准线的计数超过 “ 负噪声阈值 ”(NegativeNoiseThreshold) 参数，即两者之差大于此阈值，则不会更新基准线。 但是，如果当前原始计数一直保持低状态 （差值大于阈值），并且次数超过 “ 低基准线复位 ”(LowBaselineReset) 参数指定的采样次数，基准线会被重置。

可能值为 0 到 255。



低基准线复位 (LowBaselineReset)

“ 低基准线复位 ”(LowBaselineReset) 参数与 “ 负噪声阈值 ”(NegativeNoiseThreshold) 参数配合使用。 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。 它计算重设基准线所需的异常低的采样的次数， 它用来纠正启动时手指已经在传感器上面的情况。

可能值为 0 到 255。

传感器自动复位 (Sensors Autoreset)

此参数决定是在所有情况下都更新基准线，还是仅当信号差值低于噪声阈值时才更新。 当设置为**启用**时，基准线将持续更新。 此设置限制了传感器的最大持续时间 （典型值为 5-10 秒），但它防止了没有任何物体触碰到传感器时原始计数突然上升所导致的传感器始终开启的现象。 较大的供电电压波动、高能射频噪声源或极快的温度变化都会导致这种突然升高。

当此参数设置为**禁用**时，则仅当原始计数与基准线之差低于 “ 噪声阈值 ”(Noise Threshold) 参数时基准线才进行更新。

可能值为 “ 启用 ” 和 “ 禁用 ”。

[+] Feedback

高层 API (High Level API)

“ 高层 API”(High Level API) 参数可设置为 “ 启用 ” 或 “ 禁用 ”。 将此参数设置为 “ 禁用 ”，可通过排除用户模块提供的高层 API 来节省 RAM 和 ROM 空间。 除非您需要额外的 RAM 和 ROM，且不需要高层 API，否则请将此参数的状态保留为 “ 启用 ”。

可能值为 “ 启用 ” 和 “ 禁用 ”。

时钟 (Clock)

“ 时钟 ”(Clock) 参数可用于增加传感器的有效电阻值。 如果传感器面积较大，有效电阻可能过高，以至于无法自动校准开关电容电路。 触摸板行 / 列或较大的接近传感器的灵敏度可能会下降。 在这种情况下，稳定电压会远远低于比较器阈值。 设置较大的 IMO 分频器可增加有效电阻，以补偿高电容。

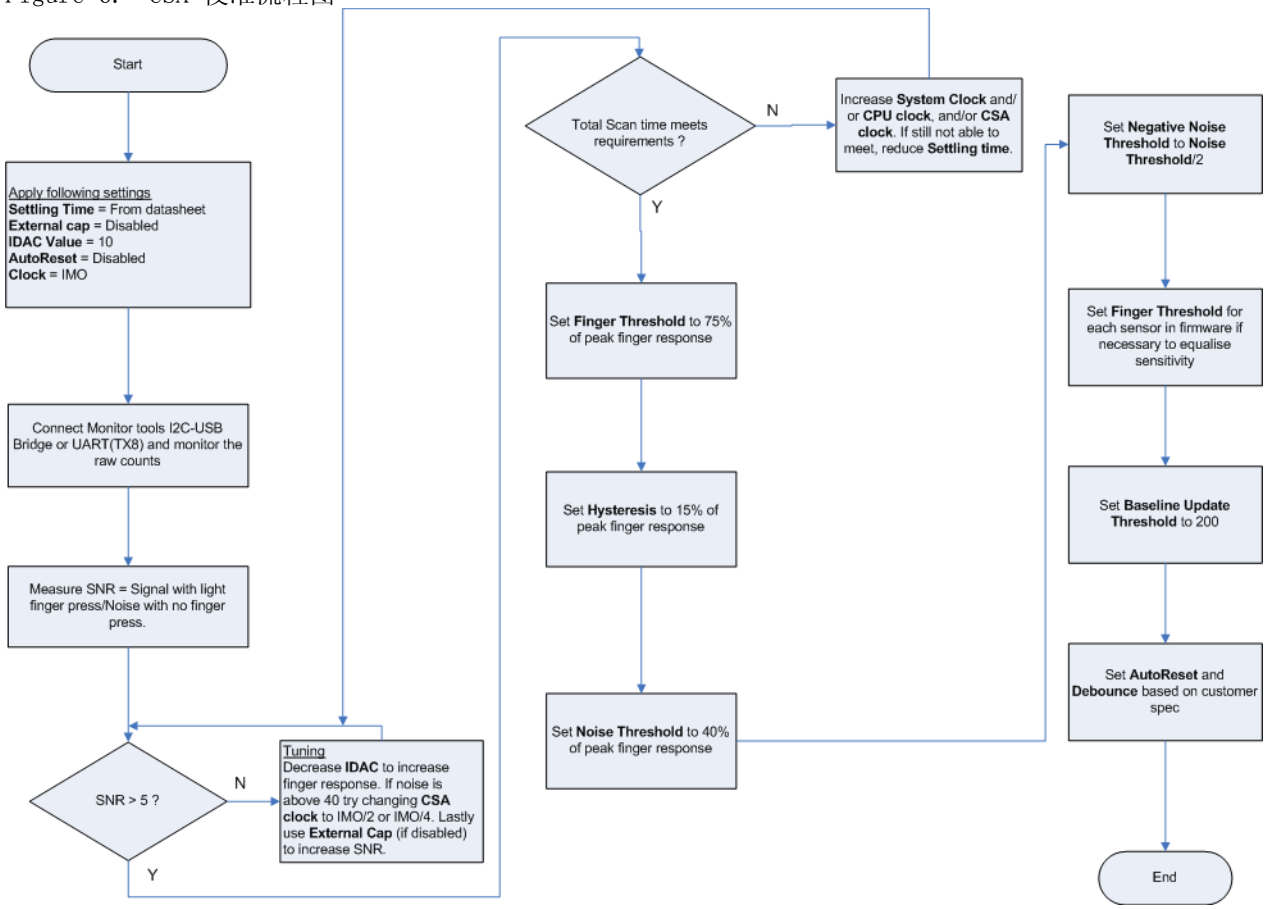
可能值为 IMO、IMO/2、IMO/4 和 IMO/8。

[+] Feedback

CSA 校准

为了获得最佳性能，可以通过实际的 CapSense 硬件和外覆层调整 CSA 参数。 下面的流程图显示了如何校准 CSA：

Figure 6. CSA 校准流程图



1. 开始操作时采用 CSA 用户模块的默认设置。
2. 使用 I^2C-USB 桥接器或 UART 以及实际的硬件和外覆层，捕获传感器的原始计数、基准线和差值。
3. **粗调。** 检查信噪比 (SNR) 是否大于 5。 如果信噪比小于 5，则通过下列建议的 PCB 指南可增加信噪比：降低 IDAC 值，更改 CSA 时钟并使用外部电容。 有关 PCB 指南，请参考应用笔记 “CapSense 最佳实践 ”。 AN2394. 有关信噪比和信噪比测量方法的详细信息，请参考应用笔记 “ 电容式感应 – CapSense 应用的信噪比要求 ”。 AN2403.
4. 检查所有传感器的总扫描时间是否符合要求。 如果未达到要求，则需增加系统时钟、CPU 时钟和 / 或 CSA 时钟。 由于这些参数也影响信噪比，请返回到步骤 3。 通过几次尝试，找到可以生成最佳信噪比和所需扫描时间的最佳 IDAC 和 CSA 时钟参数。
5. 当激活按键时捕获差值。 将 “ 手指阈值 ”(Finger Threshold) 参数设置为手指峰值的 75%。
6. 将 “ 迟滞 ”(Hysteresis) 参数设置为手指峰值的 15%。
7. 将 “ 噪声阈值 ”(noise threshold) 设置为手指峰值的 40%。
8. 将 “ 负噪声阈值 ”(negative noise threshold) 设置为 “ 噪声阈值 ”(noise threshold) 的一半。
9. 如果需要，设置各个传感器的手指阈值。 这可以通过写入到固件中的 CSA_baBtnFThreshold 阵列来实现。
10. 根据需要设置基准线更新阈值。 更新基准线的频率必须依据项目之间的关系决定。 基准线应该是慢速移动参考，这有助于减少噪声和温度对电容式传感器的影响。

[+] Feedback

- **快速更新基准线速率：** 如果使手指缓慢移向按键，则会出现问题。 这称为 “ 基准线速率超过手指速率 ”。
- **慢速更新基准线速率：** 这样会使按键较易受到温度波动的影响，可能导致 “ 按键锁定 ”。

11. 根据需要设置 “ 自动复位 ”(AutoReset) 和 “ 防反跳 ”(Debounce) 参数。 有关更多信息，请参考 “ 参数 ” 一节。

应用程序编程接口

应用程序编程接口 (API) 例程作为用户模块的一部分提供，允许您编写用于与用户模块进行交互的代码而无需考虑其实现细节。 本节具体说明了每个接口函数以及 include 文件所提供的相关常量。

Note 在这里，与所有用户模块 API 中的情况相同，调用 API 函数会改变 A 和 X 寄存器的值。 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 “ 寄存器易失 ” 策略是为了提高效率，并且从 PSoC Designer 的 1.0 版本起已开始使用。 C 编译器自动遵循此要求。 汇编语言编程人员需确保其代码遵循此策略。 虽然一些用户模块 API 函数可以保留 A 和 X 不变，但是无法保证它们将来也会如此。

所提供的进入点用于初始化、启动和停止 CSA 用户模块。 在所有情况下都要将以下进入点中所示的 CSA 前缀替换为模块的实例名称。 未能使用正确的实例名称是产生语法错误的常见原因。

软件控制参数

传递给 API 的控制参数包括：

bSnsGroup

引用用作滑条的一组特定传感器。 CSA_bGetCentroidPos 使用它来选择要更新的传感器组。

按键包含在组 0 中。 滑条包含在组 1 及更高编号的组中。

bSensor

CSA_wGetPortPin 使用传感器编号为选定活动传感器确定端口和位掩码 （bPort 和 bMask）。 CSA_wGetPortPin 会返回 bMask 和 bPort。 CSA_EnableSensor 和 CSA_DisableSensor 使用这两项选择特定传感器。 CSA_wReadSensor 还会使用它们设置要返回哪个传感器的计数。

CSA 数据阵列

API 函数使用多个全局阵列。 不应手动更改这些阵列。 您可以出于调试目的对这些值进行检查。

CSA_waSnsResult

此阵列存储每个传感器的 16-bit 原始计数值。

CSA_waSnsBaseline

此阵列存储每个传感器的 16-bit 基准线值。

CSA_waSnsDiff

此阵列存储每个传感器的 16-bit 差值 （原始计数 - 基准线）。

CSA_baSnsOnMask

此 8-bit 阵列存储传感器的开 / 关数据 （对于按键或滑条）。 阵列中每个元素可存储最多 8 个传感器的传感器状态。CSA_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）。CSA_baSnsOnMask[1] 包含传感器 8 到 15 （如果需要）的掩码位，以此类推。 此字节阵列包含的元素数足以包含所有放置的传感器。 如果传感器开启，则位值为 1 ； 如果传感器关闭，则位值为 0。

[+] Feedback

CSA_baDACCodeScan

此阵列存储 8-bit IDACSetting 参数值，这些参数值将设定 Cmod 上的 ADC 线性斜坡电压的斜率。只要在调用 CSA_Start 之后立即加载该阵列 （在初始化基准线或扫描任何传感器之前），便可以分别为每个传感器配置此参数。

CSA_baDACCodeBaseline

该阵列存储在 CSA_Start. 中自动确定的每个传感器的 8-bit 校准设置。该阵列中的值为加载每个 CapSense 输入的寄生电容提供了一个相对度量。

数据表

向导会基于您输入的传感器个数、引脚分配、双工和分辨率生成一组数据表。 传感器表位于 CSA_table.asm 中。 分组和双工表位于 CSAHL.asm 中。

CSA_Sensor_Table

在传感器表中，每个传感器对应一个 2 字节条目。 第一个字节是端口号，第二个字节是位的位掩码（而非位的编号）。 该表包含所有的独立传感器，然后是依序排列的各个滑条传感器。 以下是包含六个传感器的示例表格：

```
CSA_Sensor_Table:
_CSA_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
```

CSA_Group_Table

分组表定义每个传感器组或滑条传感器组。 表中每一行的第一个条目是该组的起始传感器编号。 第二个条目是该组中的传感器数。 第三个条目是 0 （如果未使用双工）。 第四、第五和第六个条目组合在一起形成用于质心计算的固定点乘数值。 下面是一个包含七个传感器的示例：

```
CSA_Group_Table:
CSA_Group_Table:
// Group Table
//        Origin Count  Diplex  SliceMultiplier
      db    0,    0x7,    0x0,    0x00     // Buttons
```

在具有独立传感器和滑条传感器的项目中，独立传感器组以及每个滑条传感器在分组表中都有一个单独的条目，如下所示：

```
CSA_Group_Table:
CSA_Group_Table:
; Group Table
;         Origin Count  Diplex   DivBtwSns(wholeMSB, wholeLSB, fractByte)
      db    0,    0x7,    0x0,    0x00,    0x00,    0x00 ; Buttons
      db    0x6,  0xA,    0x4,    0x0,     0x7,     0xE5 ; Slider 1
```

CSA_Diplex_Table

双工表定义每个滑条传感器的传感器全范围映射。 该表包含两部分：每个滑条的传感器映射，以及每个单独滑条对其表的引用。 下面是一个包含 10 个传感器的滑条的典型示例。

```
DiplexTable_0:
```


[+] Feedback

```
; This group is not a diplexed slider

DiplexTable_1:
    db    0,1,2,3,4,5,6,7,8.9,0,3,6,9,1,4,7,2,5,8   // 10 switch slider

CSA_Diplex_Table:
_CSA_Diplex_Table:
    db >DiplexTable_0, <DiplexTable_0
    db >DiplexTable_1, <DiplexTable_1
```

基本 API

基本 API 用于启动和停止用户模块。

CSA_Start()

说明：

校准每个传感器的 CSA 用户模块。 断开所有传感器引脚与模拟复用器总线的连接。 所有传感器引脚将转为接地。 将 C_{mod} 电容连接到系统。

C 原型：

```
void  CSA_Start()
```

汇编：

```
lcall  CSA_Start
```

参数：

无

返回值：

无

副作用：

**

CSA_Stop()

说明：

禁用 CapSense 模块。 调用 CSA_ClearSensors 以断开所有传感器引脚与模拟复用器总线的连接并将其转为接地。

C 原型：

```
void  CSA_Stop()
```

汇编：

```
lcall  CSA_Stop
```

参数：

无

返回值：

无

[+] Feedback

副作用：

**

低层 API

低层 API 用于获取传感器数据。

CSA_ScanSensor

说明：

扫描单个传感器以确定代表其电容的原始计数值。 该例程假定在执行之前调用了 CSA_Start 函数。 该例程是一个封锁调用，即它会等待 CapSense 模块中断 CSA_ISR 完成。 然后，它会将来自 16-bit 计数器的数据传递给 CSA_waSnsResult 阵列。

C 原型：

```
void  CSA_ScanSensor(BYTE bSensor)
```

汇编：

```
mov  A,  bSensor
lcall CSA_ScanSensor
```

参数：

bSensor： 范围为 0 到 n-1，其中 n 是在 CSA 向导中设置的传感器总数与包含在滑条传感器中的传感器数之和。

返回值：

无

副作用：

**

CSA_ScanAllSensors

说明：

通过为每个传感器索引调用 CSA_ScanSensor 扫描所有配置的传感器。

C 原型：

```
void  CSA_ScanAllSensors()
```

汇编：

```
lcall  CSA_ScanAllSensors
```

参数：

无

返回值：

无

副作用：

**

[+] Feedback

CSA_ClearSensors

说明：

针对每个传感器执行 CSA_DisableSensor。 传感器引脚将断开与模拟复用器总线的连接并转为接地。

C 原型：

```
void  CSA_ClearSensors()
```

汇编：

```
lcall  CSA_ClearSensors
```

参数：

无

返回值：

无

副作用：

**

CSA_wReadSensor

说明：

返回指定传感器的原始计数值。

C 原型：

```
WORD  CSA_wReadSensor(BYTE bSensor)
```

汇编：

```
mov A, bSensor
lcall CSA_wReadSensor
```

参数：

bSensor： 范围为 0 到 n-1，其中 n 是在 CSA 向导中设置的传感器总数与包含在滑条传感器中的传感器数之和。

返回值：

该传感器的原始计数值，A 中的 LSB 和 X 中的 MSB。

副作用：

**

CSA_wGetPortPin

说明：

返回给定传感器的端口号和引脚掩码。 传递的参数对 CSA_Sensor_Table. 中的数据编制索引并进行选择。

C 原型：

```
WORD  CSA_wGetPortPin(BYTE bSensor)
```


[+] Feedback

汇编：

```
mov  A,  bSensor
lcall     CSA_wGetPortPin
```

参数：

bSensor：范围为 0 到 n-1，其中 n 是在 CSA 向导中设置的传感器总数与包含在滑条传感器中的传感器数之和。

返回值：

bPort 和 bMask：用于确定所选择的特定传感器的端口号和位掩码。

副作用：

**

CSA_EnableSensor

说明：

为下一个测量周期中的扫描配置选定的传感器。传感器的端口和引脚是使用 CSA_wGetPortPin 例程选择的，其中端口号和传感器位掩码分别加载到 X 和 A 中。驱动模式会修改以便将选中的端口和引脚设定为模拟 High Z 模式，并启用正确的模拟复用器总线输入。

C 原型：

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

汇编：

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

参数：

bPort 和 bMask：用于确定所选择的特定传感器的端口号和位掩码。

返回值：

无

副作用：

**

CSA_DisableSensor

说明：

断开传感器的连接，使其不再是一个输入。通常，该调用与 CSA_wGetPortPin. 配合使用。这将断开端口引脚与模拟复用器总线的连接。驱动模式将更改为 Strong (01)，数据寄存器位设置为 0。这会将传感器接地。

C 原型：

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

汇编：

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


[+] Feedback

参数：

bPort 和 bMask： 用于确定所选择的特定传感器的端口号和位掩码。

返回值：

无

副作用：

**

高层 API

高层 API 用于处理低层 API 获取的传感器数据。

CSA_UpdateSensorBaseline

说明：

更新某个传感器的传感器基准线。 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 CSA_UpdateSensorBaseline 时，通过从原始计数值中减去以前的基准线来计算差值。 该差值存储在 waSnsDiff 阵列中。
2. 每次调用 CSA_UpdateSensorBaseline 时，都会将该差值与噪声阈值进行比较。 如果差值低于 “ 噪声阈值 ”(noise threshold)，则会将差值的一半加到 / 累加到一个虚拟水桶中。 如果差值高于 “ 噪声阈值 ”(noise threshold)，则不更新基准线。
3. 当虚拟水桶中的累计差值达到 “ 基准线更新阈值 ”(BaselineUpdateThreshold) 时，基准线将递增，且水桶将复位为 0。
4. 如果差值低于 “ 噪声阈值 ”(noise threshold)，则 waSnsDiff 阵列中储存的值将设置为 0。

C 原型：

```
void CSA_UpdateSensorBaseline(BYTE bSensor)
```

汇编：

```
mov   A,   bSensor
lcall  CSA_UpdateSensorBaseline
```

参数：

bSensor： 范围为 0 到 n-1，其中 n 是在 CSA 向导中设置的传感器总数与包含在滑条传感器中的传感器数之和。

返回值：

无

副作用：

**

[+] Feedback

CSA_UpdateAllBaselines

说明：

使用 CSA_bUpdateSensorBaseline 例程更新所有传感器的基准线。

C 原型：

```
void  CSA_UpdateAllBaselines()
```

汇编：

```
lcall CSA_UpdateAllBaselines
```

参数：

无

返回值：

无

副作用：

**

CSA_bIsSensorActive

说明：

与手指阈值进行比较，检查给定传感器的差值阵列。 将迟滞考虑在内。 根据传感器的状态，“ 迟滞 ”(Hysteresis) 值将与 “ 手指阈值 ”(Finger Threshold) 相加或相减。 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。 此函数还会更新 CSA_baSnsOnMask 阵列中传感器的位。

C 原型：

```
BYTE  CSA_bIsSensorActive(BYTE bSensor)
```

汇编：

```
mov  A,  bSensor
lcall CSA_bIsSensorActive
```

参数：

bSensor： 范围为 0 到 n-1，其中 n 是在 CSA 向导中设置的传感器总数与包含在滑条传感器中的传感器数之和。

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。

副作用：

**

[+] Feedback

CSA_bIsAnySensorActive

说明：

与手指阈值进行比较，检查所有传感器的差值阵列。 针对每个传感器调用 CSA_bIsSensorActive，以便在调用此函数后 CSA_baSnsOnMask 阵列保持为最新状态。

C 原型：

```
BYTE  CSA_bIsAnySensorActive()
```

汇编：

```
lcall CSA_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。

副作用：

**

CSA_SetDefaultFingerThresholds

说明：

通过 “ 手指阈值 ”(FingerThreshold) 参数值加载 CSA_baBtnFThreshold 阵列。 如果 CSA_baBtnFThreshold 阵列不是通过自定义值手动加载的，则必须在扫描之前调用此函数。

C 原型：

```
BYTE  CSA_SetDefaultFingerThresholds()
```

汇编：

```
lcall CSA_SetDefaultFingerThresholds
```

参数：

无

返回值：

无

副作用：

**

CSA_InitializeBaselines

说明：

通过扫描每个传感器，加载含有初始值的 CSA_waSnsBaseline 阵列。

C 原型：

```
void  CSA_InitializeBaselines()
```

汇编：

```
lcall CSA_InitializeBaselines
```


[+] Feedback

参数：

无

返回值：

无

副作用：

**

CSA_InitializeSensorBaseline

说明：

为 CSA_waSnsBaseline 阵列加载特定传感器的初始值。 用于为特定传感器复位基准线。

C 原型：

```
void  CSA_InitializeSensorBaseline(BYTE bSensor)
```

汇编：

```
lcall CSA_InitializeSensorBaseline
```

参数：

bSensor： 范围为 0 到 n-1，其中 n 是在 CSA 向导中设置的传感器总数与包含在滑条传感器中的传感器数之和。

返回值：

无

副作用：

**

CSA_wGetCentroidPos

说明：

检查差值阵列的质心。 如果存在一个质心，则会在临时变量中存储偏移和长度，并根据 CSA 向导中指定的分辨率计算质心位置。

C 原型：

```
WORD  CSA_wGetCentroidPos(BYTE bSnsGroup)
```

汇编：

```
mov  A,  bSnsGroup
lcall CSA_wGetCentroidPos
```

参数：

bSnsGroup： 引用用作滑条的一组特定传感器。

返回值：

滑条的位置值、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。 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信号，则在差值数据传输后调用此例程。

[+] Feedback

注： 如果滑条段的噪声计数大于噪声阈值，此子例程可能会生成错误的质心结果。 噪声阈值应小心设置 （高出噪声水平足够程度），以确保噪声不会生成错误的质心。

固件源代码示例

扫描按键与开关 LED

此代码是为 CSA UCC 板 (CY3280-20x34) 和线性滑条模块板 (CY3280-SLM) 编写的。

```
//------ Sample code for CSA buttons that LEDs On and Off   ----------
//------     pin assignments for Linear Slider Module plugged   ----------
//-------     into CSA UCC board, CY3280-20x43+SLM            ----------

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

void main(void)
{
//initialize LED states
  PRT0DR |= 0b00100010;  //turn-off LED on P0[5],P0[1]
  PRT1DR |= 0b00000100;  //turn-off LED on P1[2]
  PRT2DR |= 0b10100000;  //turn-off LED on P2[7],P2[5]

//Set port drive modes for LEDs
  PRT0DM0 |= 0b00100010;  //strong on P0[5],P0[1]
  PRT0DM1 &=~0b00100010;

  PRT1DM0 |= 0b10100100;  //strong on P1[2]
  PRT1DM1 &=~0b10100100;

  PRT2DM0 |= 0b10100000;  //strong on P2[5],P2[7]
  PRT2DM1 &=~0b10100000;

  M8C_EnableGInt;  //enable global interrupts for use with CSA

  CSA_Start();     //initialize the CSA User Module
  CSA_SetDefaultFingerThresholds();   //Load finger thresholds
  CSA_InitializeBaselines();  //Set baselines to current count

  while(1)  //infinite loop scanning buttons
  {
    CSA_ScanAllSensors();    //sample all buttons
    CSA_UpdateAllBaselines();  //compute baseline, all buttons

// control the LEDs using the sensor states.
// LED ON if active, OFF if not active.
// Check buttons in sequence.
    if (CSA_bIsSensorActive(0))
    {
      PRT1DR &= ~0b00000100;   //turn-on LED on P1[2]
    }
    else
    {
      PRT1DR |= 0b00000100;  //turn-off LED on P1[2]
```


[+] Feedback

```
    }
    if (CSA_bIsSensorActive(1))
    {
      PRT0DR &= ~0b00100000;  //turn-on LED on P0[5]
    }
    else
    {
      PRT0DR |= 0b00100000;  //turn-off LED on P0[5]
    }
    if (CSA_bIsSensorActive(2))
    {
      PRT0DR &= ~0b00000010;  //turn-on LED on P0[1]
    }
    else
    {
      PRT0DR |= 0b00000010;  //turn-off LED on P0[1]
    }
    if (CSA_bIsSensorActive(3))
    {
      PRT2DR &= ~0b10000000;  //turn-on LED on P2[7]
    }
    else
    {
      PRT2DR |= 0b10000000;  //turn-off LED on P2[7]
    }
    if (CSA_bIsSensorActive(4))
    {
      PRT2DR &= ~0b00100000;  //turn-on LED on P2[5]
    }
    else
    {
      PRT2DR |= 0b00100000;  //turn-off LED on P2[5]
    }
  }
}
```

使用线性滑条控制 LED 亮度

此代码是为 CSA UCC 板 (CY3280-20x34) 和线性滑条模块板 (CY3280-SLM) 编写的。

```
//------ Sample code for CSA slider controlling LED intensity ----------
//------   pin assignments for Linear Slider Module plugged   ----------
//-------  into CSA UCC board, CY3280-20x43+SLM               ----------

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

int wCentroid = 0;  //estimated finger position; 0xffff for no finger
int wPos = 0;  //estimated finger position
int wLED_PWM; //controls LED intensity

void main(void)
{
//initialize LED states
  PRT0DR |= 0b00100010;  //turn-off LED on P0[5],P0[1]
  PRT1DR |= 0b00000100;  //turn-off LED on P1[2]
  PRT2DR |= 0b10100000;  //turn-off LED on P2[7],P2[5]

//Set port drive modes for LEDs
  PRT0DM0 |= 0b00100010;  //strong on P0[5],P0[1]
  PRT0DM1 &=~0b00100010;

  PRT1DM0 |= 0b10100100;  //strong on P1[2]
  PRT1DM1 &=~0b10100100;

  PRT2DM0 |= 0b10100000;  //strong on P2[5],P2[7]
  PRT2DM1 &=~0b10100000;

  M8C_EnableGInt;  //enable global interrupts for use with CSA

  CSA_Start();     //initialize the CSA User Module
  CSA_SetDefaultFingerThresholds();   //Load finger thresholds
  CSA_InitializeBaselines();  //Set baselines to current count

  while(1)  //infinite loop scanning slider
  {
    CSA_ScanAllSensors();    //sample all sensors
    CSA_UpdateAllBaselines();  //compute baseline, all sensors

    wCentroid = CSA_wGetCentroidPos(1); /estimated position
    if (wCentroid != 0xffff)  //0xffff means finger off slider
    {
      wPos = wCentroid;  //get position, range is 0 to 100
    }

    if (wPos > 0)  //if position>0, then pulse all LEDs ON
    {
      PRT1DR &= ~0b00000100;  //turn-on LED on P1[2]
      PRT0DR &= ~0b00100000;  //turn-on LED on P0[5]
      PRT0DR &= ~0b00000010;  //turn-on LED on P0[1]
      PRT2DR &= ~0b10000000;  //turn-on LED on P2[7]
```


[+] Feedback

```
      PRT2DR &= ~0b00100000;  //turn-on LED on P2[5]

      for (wLED_PWM = 0; wLED_PWM < wPos*wPos/100; wLED_PWM++)
      {   //control LED pulse width by position^2
      }  //this control function looks nice

    }
    // LED pulse ON is over for this period, turn all off
    PRT1DR |= 0b00000100;  //turn-off LED on P1[2]
    PRT0DR |= 0b00100000;  //turn-off LED on P0[5]
    PRT0DR |= 0b00000010;  //turn-off LED on P0[1]
    PRT2DR |= 0b10000000;  //turn-off LED on P2[7]
    PRT2DR |= 0b00100000;  //turn-off LED on P2[5]


  }  //do next scan (while loop)
}
```

更多参考资料

在阅读 CSA 用户模块文档之后，建议您阅读以下应用笔记。 这些应用笔记可在赛普拉斯半导体公司的网站 www.cypress.com 上找到：

- *CapSense 最佳实践* - AN2394
- *CapSense 应用的信噪比要求* - AN2403
- *用于调试 CapSense 应用的绘图工具* - AN2397
- *PSoC CapSense 应用的 EMC 设计注意事项* - AN2318
- *电容式感应应用中的功耗和睡眠注意事项* - AN2360
- *PSoC CapSense 布线指南* - AN2292
- *通用异步发送器的软件实现* - AN2399
- *防水电容式感应* - AN2398

[+] Feedback

版本历史记录

版本	创作者	说明
1.3	DHA	更改建立时间的默认值，从 20 改为 160。 添加附加滑条的功能。 在用户模块和配置向导中添加了辐射状滑条功能。
1.40	DHA	新增的 iDAC 代码减法的溢出条件检查。 如果发生溢出，iDAC 代码将设置为 0。

Note PSoC Designer 5.1 在所有的用户模块数据手册中提供了版本历史记录。 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 © 2006-2011 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.