SmartSense™, Autotuning CapSense® Sigma-Delta Datasheet SmartSense V 1.60

Copyright © 2010-2013 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks				API Memory		External I/O
	CapSense®	I²C/SPI	Timer	Comparator	Flash	RAM	
CY8C20xx6, CY8C20xx6A, CY8C20xx6AN, CY8C20xx6AS, CY8C20xx6H, CY8C20045, CYRF89435, CY8C20065, CY7C69xxx							
User Module	1	-	1	1	3182	63	1
Slider APIs					694	184	0
Each Sensor					5	29	1

For one or more fully configured, functional example projects that use this user module go to www.cypress.com/psocexampleprojects.

Features and Overview

- Implements CapSense® capacitive sensing in the CY8C20xx6A family of PSoC® devices.
- Auto-tunes configurable system parameters in runtime to account for sensor, IC, and PCB characteristics.
- Supports up to 36 capacitive sensors and 6 sliders.
- Supports parasitic sensor capacitance range of 5 pF to 45 pF.
- Detects touches as low as 0.1 pF, that is, a finger can be detected through up to 15 mm of glass or 5 mm of plastic.
- High immunity to AC mains noise, other EMI, and power supply noise.
- Supports capacitive sensors configured as independent buttons and also as dependent arrays to form sliders.
- Provides diplexing option which allows two slider elements to be assigned per dedicated I/O pin.
- Supports slider resolution greater than physical pitch through interpolation.
- Provides shield electrode for reliable operation with high parasitic capacitance and also in the presence of water film.
- Enables guided sensor and pin assignments using the SmartSense™ Wizard.
- The CY8C20045 family does not support sliders.
- The CY8C20065 family does not support sliders.

Note This user module supports only C language projects. ASM (Assembly language) projects are not supported.

Quick Start

1. Select and place user modules that require dedicated pins (for example, I^2C and LCD). Assign ports and pins as required.
2. Select and place the SmartSense User Module.
3. Right-click the SmartSense User Module in the Workspace Explorer to access the SmartSense Wizard (refer to the SmartSense Wizard section).
4. Set the desired number of sensors, sliders, and radial sliders.
5. For sliders, enter the parameters specific to sliders.
6. Assign each of the sensors to an unused pin.
7. Enter the pin that will be connected to the external modulation capacitor.
8. Right-click the SmartSense User Module in the Workspace Explorer to access the Properties list. Enter the pin that will be used to shield the sensor, if required (see the Parameters and Resources section).
9. Generate the application and switch to the Application Editor.
10. Adapt the sample code as required to implement independent sensors, sliding sensors, and a touch-pad.
11. Program the PSoC on the target board with the .hex file generated by PSoC Designer™.

Introduction

The SmartSense User Module implements CapSense capacitive sensing. CapSense is a human interface technology that operates by detecting the capacitance of the human body. This is done using sensors that consist of a conductive surface, usually a pad etched on the PCB. Because CapSense detects body capacitance, it can sense through insulating layers such as plastic or glass overlays. These overlays usually constitute the external enclosure of the device. These attributes make CapSense an attractive alternative to mechanical input devices like push buttons and potentiometers. The major benefits of CapSense are:

- Cleaner, more aesthetically pleasing designs
- Reduced form factor possibilities for smaller end products.
- Addition of advanced user interface features, such as LED effects and proximity sensing.
- Improved reliability with no components that wear or have finite cycle life.
- Improved spill resistance due to lack of mechanical interface penetrations.
- Reduced tooling cost by eliminating penetrations or other mechanical featured needed for mechanical input devices.

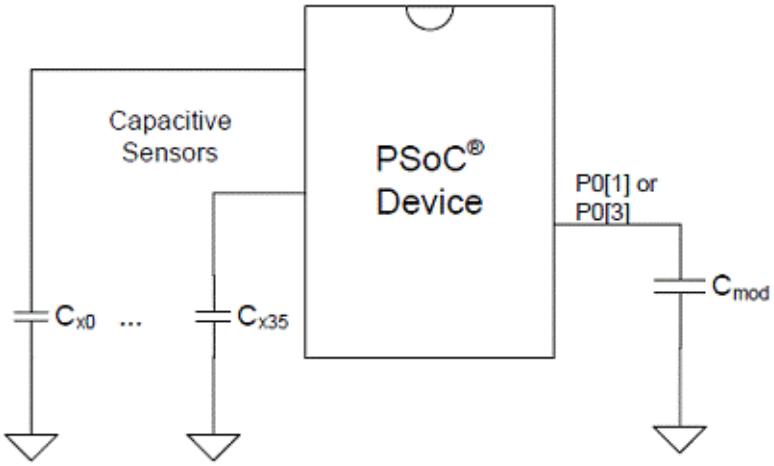
Like all Cypress CapSense solutions, SmartSense offers superior immunity to conducted and radiated noise interference, with the additional benefit of auto-tuning. Auto-tuning gives run time compensation for IC and PCB characteristics and environmental changes to ensure reliable sensor operation. For example, prototype and production PCBs frequently exhibit different material properties that affect sensor parasitic capacitance. This effect can be significant enough to require retuning of the CapSense system parameters. Auto-tuning automatically compensates for such changes, with no need for retuning. Furthermore, the auto-tuning algorithms in SmartSense continuously monitor sensor data to compensate for environment conditions, such as temperature and ambient noise level, to maintain proper sensor function.

The SmartSense User Module consists of PCB level, IC level, and software components.

PCB Level

Figure 1 shows a schematic of the SmartSense User Module. The physical sensor is typically a conductive pattern constructed on a PCB connected to a PSoC I/O pin with an insulating overlay. See the design guide Getting Started with CapSense for more information on PCB level CapSense implementation.

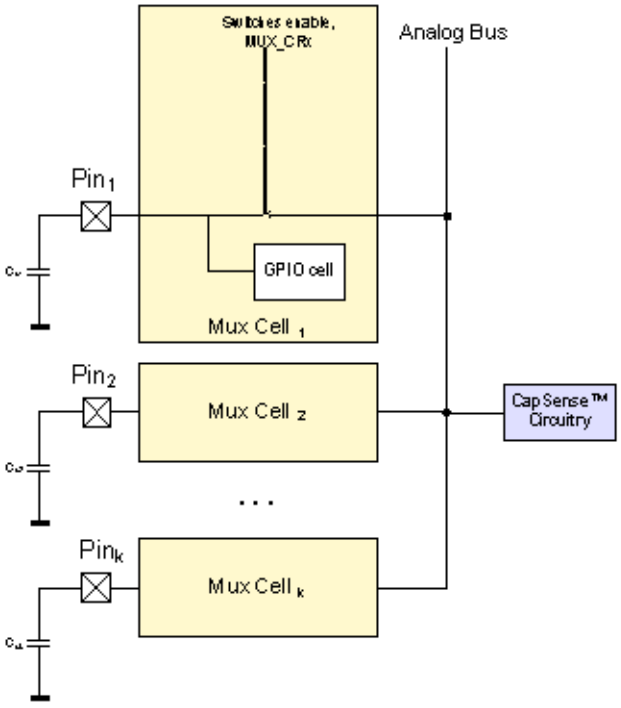
Figure 1. SmartSense Schematic



IC Level

The CY8C20xx6A devices have an Analog MUX (AMUX) Bus that allows connecting capacitive sensing analog circuitry to any PSoC pin. The SmartSense User Module connects the active sensor to the AMUX Bus, which allows the CapSense circuitry to measure its capacitance and translate that capacitance into a digital code. The firmware serially scans the sensors by sequentially setting corresponding bits in the MUX_CRx registers. This is represented in Figure 2.

Figure 2. CY8C20xx6A AMUX Block Diagram



Software

The attributes of the SmartSense software component are:

- Auto-tuning algorithms configure the analog capacitive sensing circuitry in runtime for optimal performance. These algorithms take into account physical sensor characteristics, IC characteristics, and the Sensor Sensitivity user module parameter.
- The raw count value from the capacitance converter circuitry is analyzed in runtime by API functions to make sensor state decisions and to compensate for environmental changes.
- In the case of consecutive, dependent sensors (for example, sliders and touchpads) API functions are given to interpolate a position with greater resolution than the physical pitch of the sensors.
- High level software functions accommodate slider diplexing so that one I/O pin can be routed to two physical sensors. This reduces by half the number of I/O pins consumed for a given number of slider elements.

Recommended Reading

Cypress recommends reading the following documents before implementing a CapSense design using the SmartSense User Module. These documents are available at the Cypress Semiconductor website: www.cypress.com.

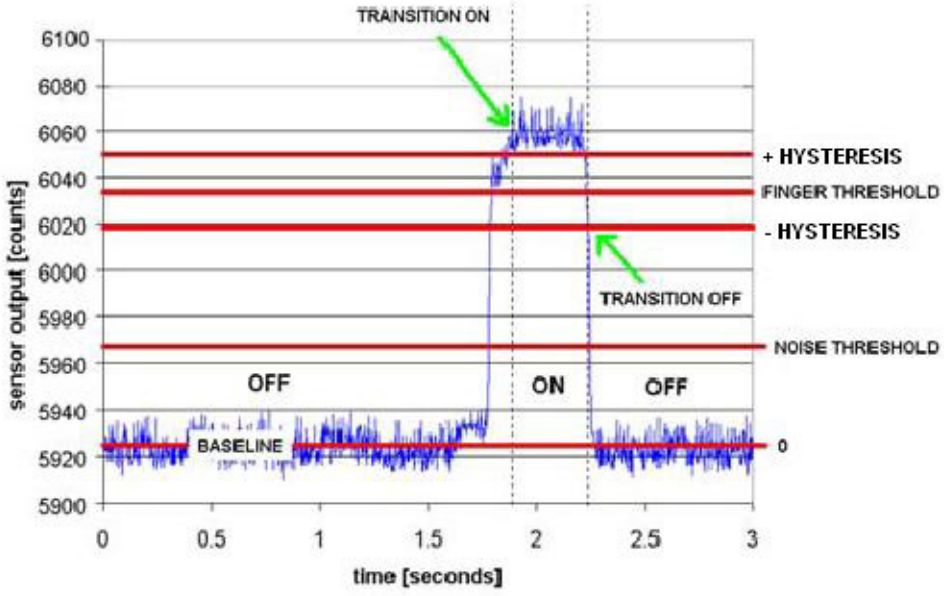
- *CY8C20x66 Series PSoC Mixed Signal Array Technical Reference Manual, section:* CapSense System
- Getting Started with CapSense
- CY8C20xx6A/H CapSense Design Guide
- CY8C21x34/B CapSense Design Guide
- CY8C20x34 CapSense Design Guide
- CY8CMBR2044 CapSense Design Guide

Capacitance Sensing Implementation

Buttons

CapSense buttons are analogous to mechanical push-buttons. They are used for discreet controls such as on/off switches, function keys, menu keys, and so on. API functions monitor the capacitance signals from each sensor and compare them to threshold levels calculated by the SmartSense auto-tuning algorithms. When a sensor is touched, its capacitance signal increases; if the increase is enough as determined by the SmartSense decision logic, that sensor will become activated. Figure 3 shows a typical signal (blue line) from a sensor when it is being activated. SmartSense automatically sets the thresholds (red lines) based on user input to provide the desired system behavior.

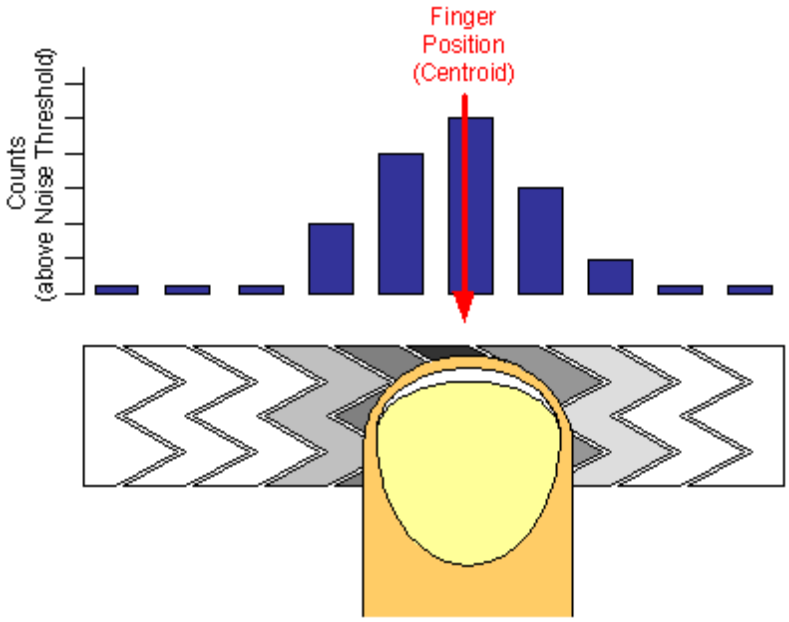
Figure 3. Capacitance Signal from a Sensor when it is Being Activated



Sliders

CapSense sliders are analogous to mechanical potentiometers. Sliders are used for controls that require a continuum of levels, such as lighting dimmers, volume control, graphic equalizers, speed controls, and so on. A CapSense slider is implemented with an array of adjacent sensors. When a slider is actuated by a finger, several adjacent sensors register an increase in capacitance signal. This is shown in Figure 4. The exact position of the touch is found by computing the centroid location of the set of activated sensors. The practical minimum number of sensors in a slider is five, and the maximum is limited only by the number of available I/O pins on the PSoC device.

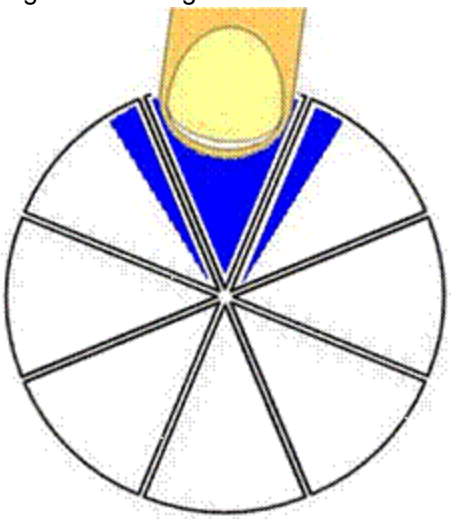
Figure 4. Interpolated Centroid Position of a Finger on a Slider



Radial Sliders

SmartSense supports two slider types: linear and radial. Linear sliders have a beginning and an end, whereas radial sliders, as shown in Figure 5, do not. In either case, when a touch occurs, the centroid algorithm takes into account signal from sensors adjacent to the sensor with the largest signal to interpolate the exact position of the touch. Radial sliders are not diplexed. The SmartSense User Module has two special API functions for radial sliders. The first function SmartSense_wGetRadiaPos() returns the centroid location, and the second function SmartSense_wGetRadialInc() returns the finger shift in resolution units. When the finger moves in a clockwise direction, SmartSense_wGetRadialInc() returns a positive offset. The reference point (0) is located in the center of the first sensor. The Resolution for both linear and radial sliders is limited to (number of pins used for sensors – 1) x 2^8 – 1 or (2 x pins used for sensors – 1) x 2^8 -1 for diplexed sliders.

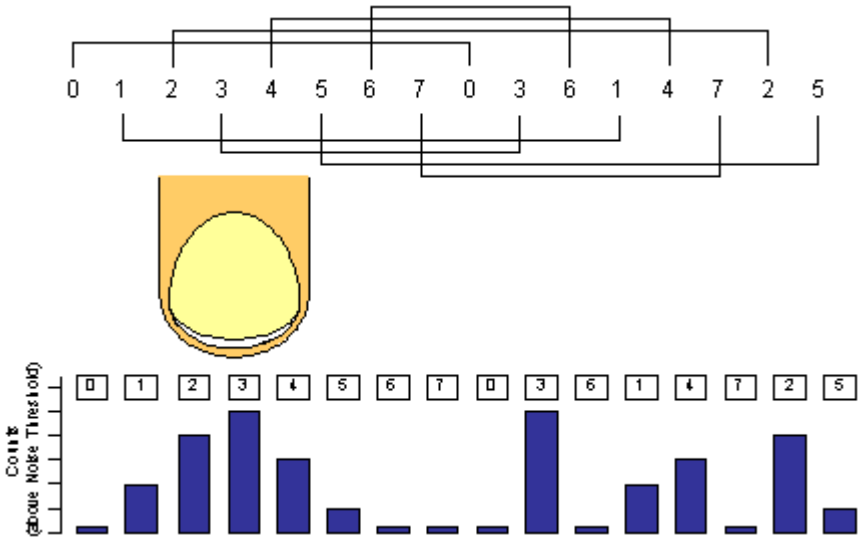
Figure 5. Finger touches Radial Slider



Diplexing

When Diplexing is used, each pin on the PSoC device that is designated as a slider element is mapped to two physical locations in the array of slider sensors. The first (or numerically lower) half of the physical locations is mapped according to the port pin assigned in the SmartSense Wizard. The second (or upper) half of the physical sensor locations is automatically mapped using the pattern shown in Figure 6.

Figure 6. Indexing of Diplexed Slider Array by SmartSense



The close proximity of strong signals in lower half of the slider results in the same levels aliased into the upper half. However, in the upper half, the results are scattered and non-contiguous. The centroid algorithm searches for strong adjacent sets of signals to declare the resolved slider position. The pattern used for mapping upper half sensors ensures that a valid signal pattern in one half does not result in a valid signal pattern in the other half, as shown in Figure 6.

Care must be exercised to ensure the mapping of sensors to pins on the PCB matches the Index by 3 sequence used by the diplexing algorithm. The capacitance of sensor pairs in a diplexed slider must be reasonably well matched (within 10 pF). The diplex sensor index table is automatically generated by the SmartSense Wizard when you select diplexing. Table 1 shows the diplexing sequences for up to 56 slider segments diplexed into 28 PSoC I/O pins.

Table 1. Diplexing Sequence for Different Slider Segment Counts

Total Slider Segment Count	Segment Sequence
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

Total Slider Segment Count	Segment Sequence
24	0,1,2,3,4,5,6,7,8,9,10,11,0,3,6,9,1,4,7,10,2,5,8,11
26	0,1,2,3,4,5,6,7,8,9,10,11,12,0,3,6,9,12,1,4,7,10,2,5,8,11
28	0,1,2,3,4,5,6,7,8,9,10,11,12,13,0,3,6,9,12,1,4,7,10,13,2,5,8,11
30	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,0,3,6,9,12,1,4,7,10,13,2,5,8,11,14
32	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0,3,6,9,12,15,1,4,7,10,13,2,5,8,11,14
34	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,0,3,6,9,12,15,1,4,7,10,13,16,2,5,8,11,14
36	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,0,3,6,9,12,15,1,4,7,10,13,16,2,5,8,11,14,17
38	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,0,3,6,9,12,15,18,1,4,7,10,13,16,2,5,8,11,14,17
40	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,0,3,6,9,12,15,18,1,4,7,10,13,16,19,2,5,8,11,14,17
42	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,0,3,6,9,12,15,18,1,4,7,10,13,16,19,2,5,8,11,14,17,20
44	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,2,5,8,11,14,17,20
46	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20
48	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20,23
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

Slider Segment Selection Guidelines for the Diplex Slider

Selecting the number of segments needed for a slider mainly depends on the physical length of the slider. However, special care must be taken when you decide the number of segments for a diplexing slider.

In a diplexing slider design, one sensor is used as two different physical slider segments to increase the physical length of slider. The number of segments that are completely covered by a finger touch must be less than the number of sensors between two segments derived from the same sensor. This ensures the proper working of the diplex slider.

For example, in the case of a 10-segment slider (5 sensors), two slider segments derived from sensor 3 are separated by only two sensors (sensor 4 and 0). In this case, a finger touch must not completely cover more than two sensor segments to ensure the proper working of the slider.

For a 12-segment slider, one finger touch must not cover more than 3 segments. Similarly, for a 18-segment slider, one finger touch must not completely cover more than 4 segments.

Interpolation and Scaling

In slider applications, it is necessary to determine position with finer granularity than the physical pitch of the sensors. SmartSense accomplishes this by interpolating the position of the finger using a centroid calculation on the signal from the sensors adjacent to the sensor with the largest signal. The array is first scanned to verify that the signal pattern is valid. The passing requirement is for some number of adjacent sensor signals to be above a noise threshold. When the strongest signal is found, this signal and those contiguous signals larger than the noise threshold are used to compute a centroid. As few as two and as many as (typically) eight sensors are used to calculate the centroid in the form of:

Equation 1

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

The calculated value is typically fractional. To report the centroid to a specific resolution, for example, a range of 0 to 100 for 12 sensors, the centroid value is multiplied by a calculated scalar. It is more efficient to combine the interpolation and scaling operations into a single calculation and report this result directly in the desired scale. This is handled in the high-level APIs.

Slider sensor count and resolution are set in the SmartSense Wizard. A scaling value is calculated by the Wizard and stored as fractional values. The multiplier for the centroid resolution is contained in three bytes with definitions given in Table 2.

Table 2. Centroid Multiplier Bit Definition for Sliders

Resolution Multiplier MSB								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Multiplier	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
Resolution Multiplier ISB								
Multiplier	128	64	32	18	16	8	4	2
Resolution Multiplier LSB								
Multiplier	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256

The resolution is found by using this equation:

Resolution = (Number of Sensors - 1) x Multiplier

The centroid is held in a 24-bit unsigned integer, and its resolution is a function of the number of sensors in the slider and the multiplier.

External Component Selection (C_{mod})

SmartSense requires an external modulation capacitor, C_{mod}, connected from Vss to one of two dedicated PSoC pins P0[1] or P0[3]. The C_{mod} pin assignment is made in the SmartSense Wizard under **Global Settings > Modulator Capacitor Pin**. The selected pin must not be used for any other purpose. The recommended value for the external modulation capacitor is 2.2 nF. A ceramic capacitor must be used. The temperature capacitance coefficient is not important. Cypress strongly recommends using a 560 ohm[1] series resistor on all CapSense sensor traces for RF interference suppression. This resistor must be placed as close to the PSoC device as possible.

Note If the "Modulator Capacitor Pin" Wizard parameter is set to 'None', then the internal capacitor (of approximately 100 pF) is enabled. However, use an external capacitor to get a better SNR.

Driven Shield Electrode

A driven shield electrode is an optional feature to reduce parasitic sensor capacitance. The benefits of this feature include improved sensor sensitivity and prevention of false sensor triggering when there is water on the overlay.

A shield electrode must be located behind or outside the sensing electrode, as shown in Figure 7. When water is on the overlay and there is no driven shield electrode, the capacitive coupling or parasitic capacitance between sensors and other conductors of the PCB increases. This causes a corresponding increase in sensor capacitance signal that might be large enough to falsely activate the sensor. The driven shield electrode nulls out parasitic capacitive coupling, so the presence of water has negligible influence on sensor capacitance signal. This prevents false activations.

Figure 7. Driven Shielding Electrode PCB Layout

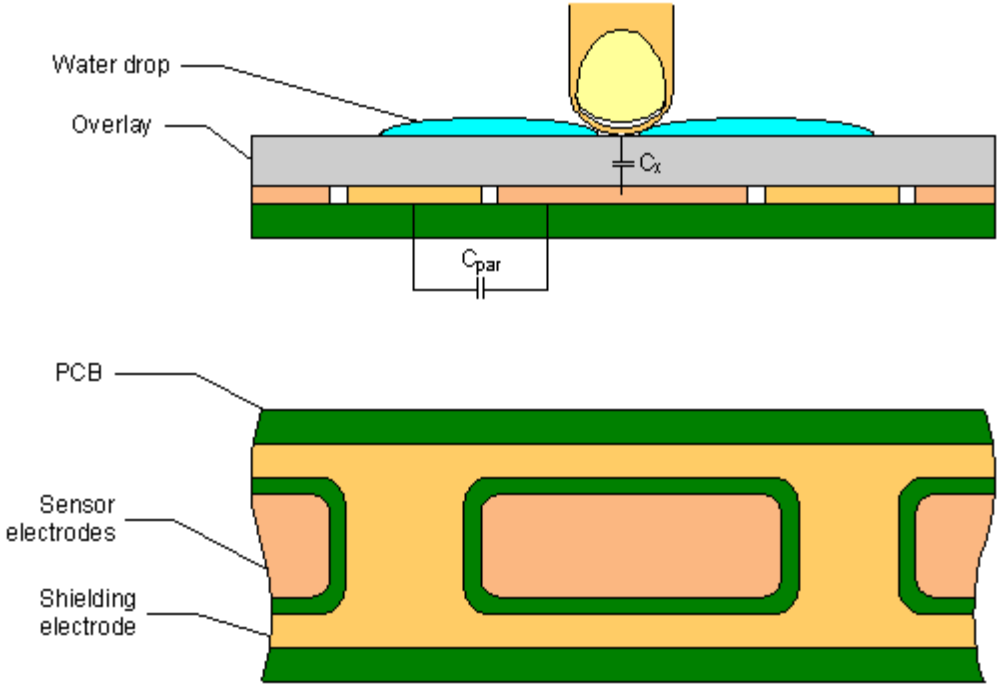


Figure 7 illustrates a driven shielding electrode for a button. As an alternative, the shielding electrode can be located on the opposite PCB layer, including the plane under the button. A hatch pattern is recommended in this case, with a fill ratio of about 30 to 40%. No additional ground plane is required.

1. Maximum value of the series resistor that can be connected to a CapSense trace is 560 ohm.

The shield electrode must be connected to one of two dedicated PSoC pins: P0[7] or P1[2]. The drive mode for selected pin must be set to **Strong**. A 560 ohm slew limiting resistor can be connected between the PSoC device and the shielding electrode to reduced emitted EMI.

Power Supply Requirement

Table 3. SmartSense Power Supply Requirement

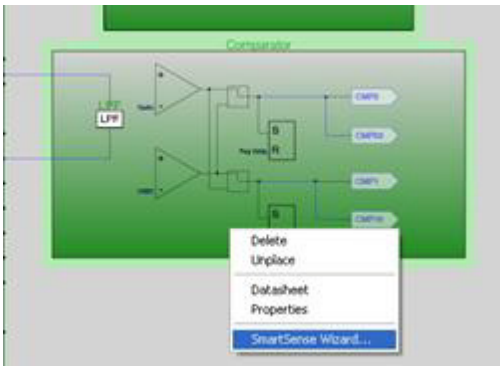
Parameter	Min	Typical	Max	Unit	Test Conditions and Comments
V_{DD}	1.8	-	5.50	V	If the V_{DD} drop in an application exceeds 5% of the base V_{DD}, the rate at which V_{DD} drops and recovers must not exceed 200 mV/s.

Placement

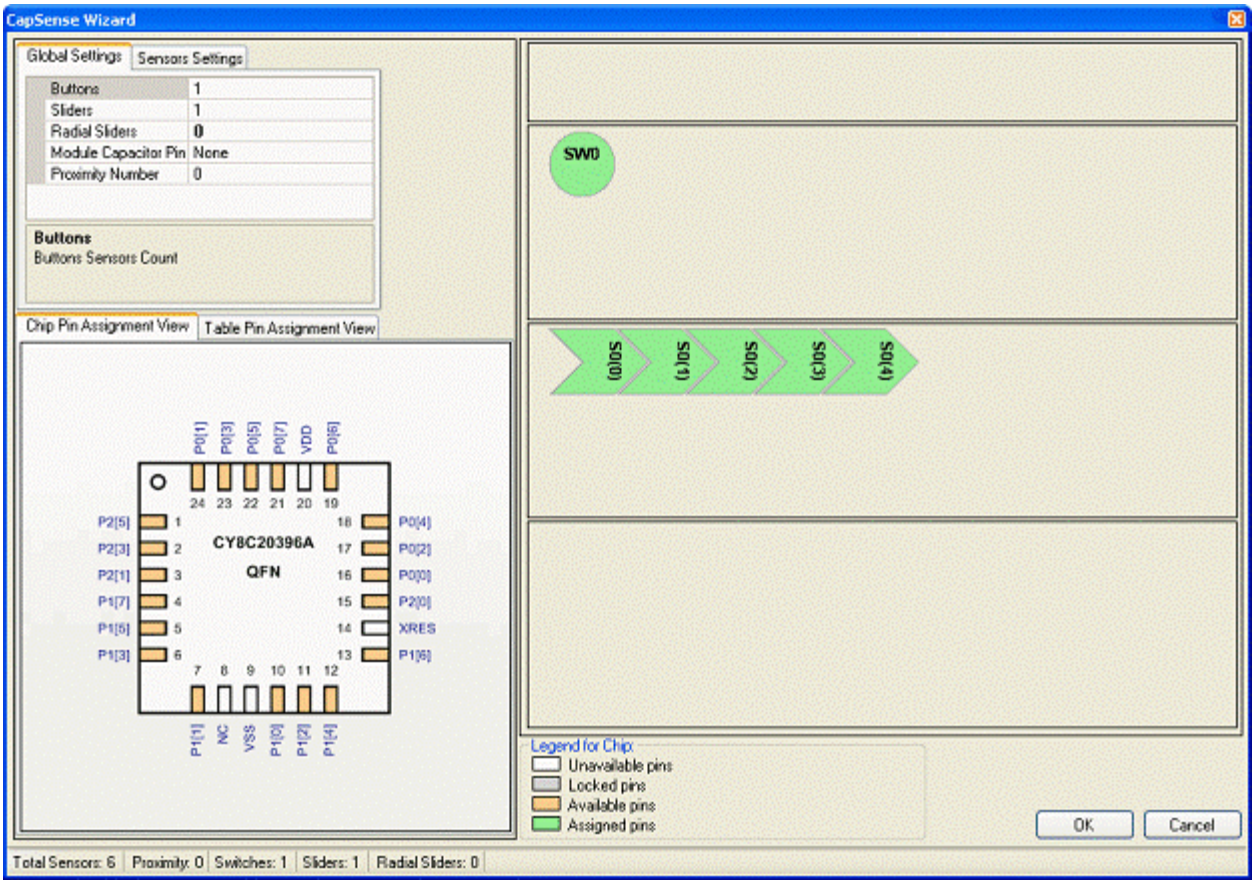
The CapSense and Timer1 blocks are assigned to SmartSense when the user module is instantiated. Alternate placements are not available. User modules that require dedicated pin resources, including the LCD and I2CHW, must be placed before starting the SmartSense Wizard. This ensures that the dedicated pins are reserved and cannot be inadvertently assigned as sensors when sensors are mapped to I/O pins in the SmartSense Wizard. Avoid P1[0] and P1[1] when placing capacitive sensor connections. These pins are used for programming the part and may have excess routing capacitance, which adversely affects sensor sensitivity.

SmartSense Wizard

1. To access the SmartSense Wizard, right click any block occupied by SmartSense in the Device Editor Interconnect View and select the SmartSense Wizard with a left mouse click.



2. The Wizard opens showing the numeric entry boxes for the number of sensors and the number of slider sensors.



Wizard Pin Legend

White – The pin cannot be used as a CapSense input.

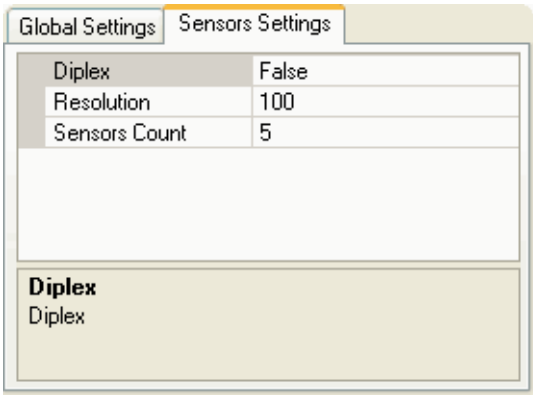
Gray – The pin is locked. There are two possible causes for this. The first possibility is that another user module such as the LCD or I^2C has claimed the pin. The second possibility is that the name of the pin has been changed from its default. To return the pin name to its default, expand the pin in the Pinout view, and select **Default** from the **Select** menu. The pin is now available for assignment in the Wizard.

Orange – The pin is available for assignment.

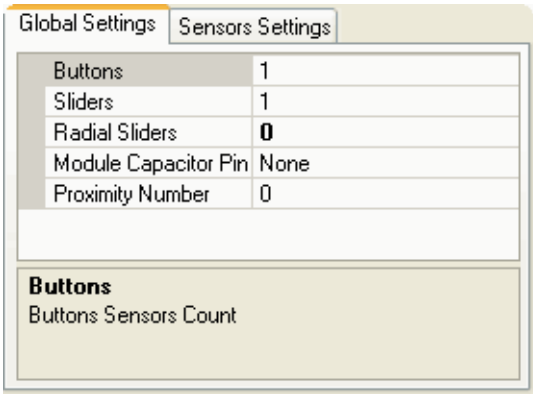
Green – The pin has been assigned as a CapSense input.

3. Type the number of independent buttons, sliders, and radial sliders. The total number of sensors (buttons plus slider elements) is limited to the number of pins available. After entering the data, press the [Enter] key to update the display with the new value.

4. Select **Sensor Settings** to set the settings for sliders and radial sliders. To alter settings, click one of your sliders to activate it. Type the number of sensor elements in each slider. The practical minimum number of sensors in a slider sensor is five, the maximum is limited only by pin count. After entering the data, press the [Enter] key to update the display.

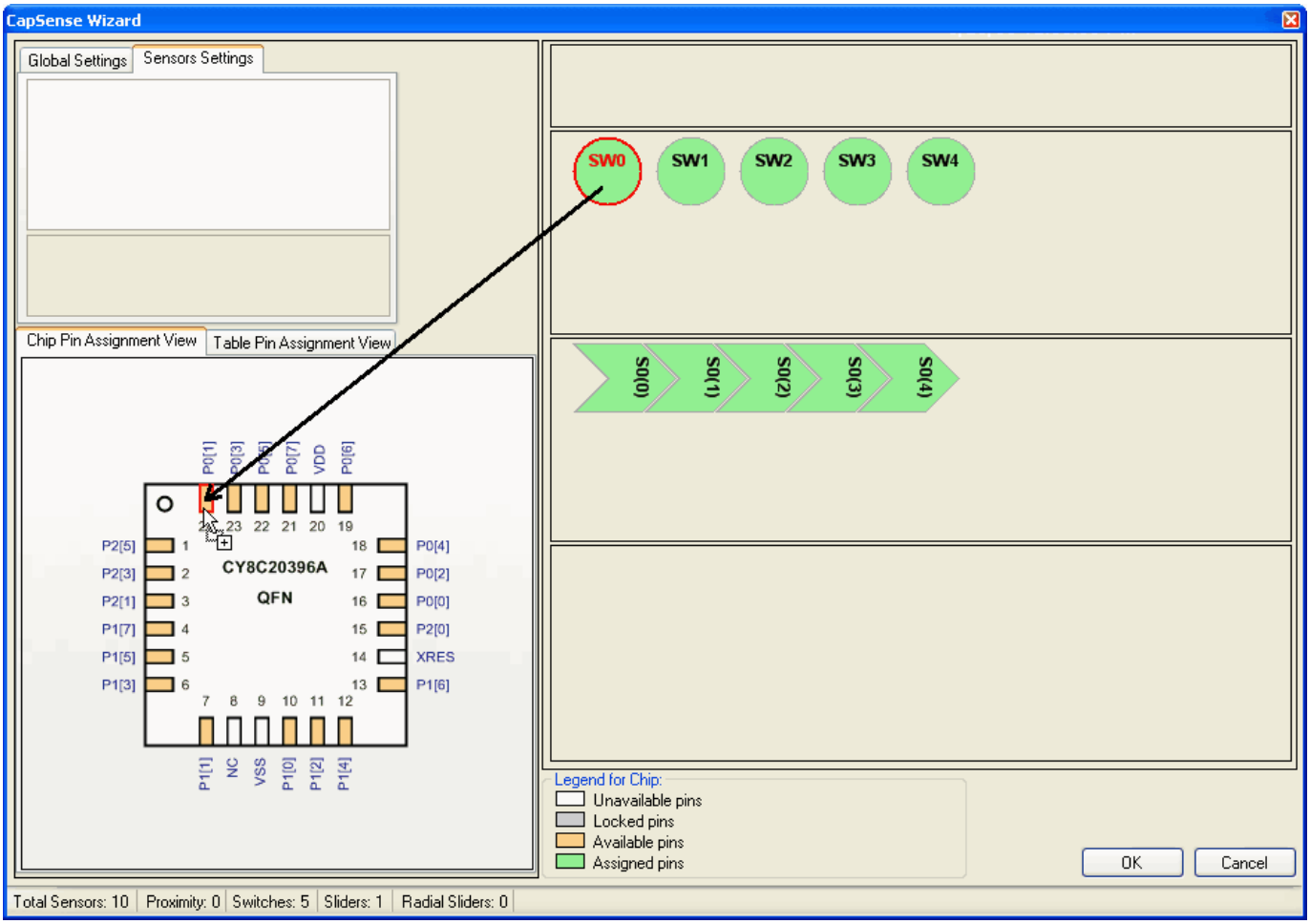


5. Select modulator capacitor (C_{mod}) pin. Choose P0[1] or P0[3].



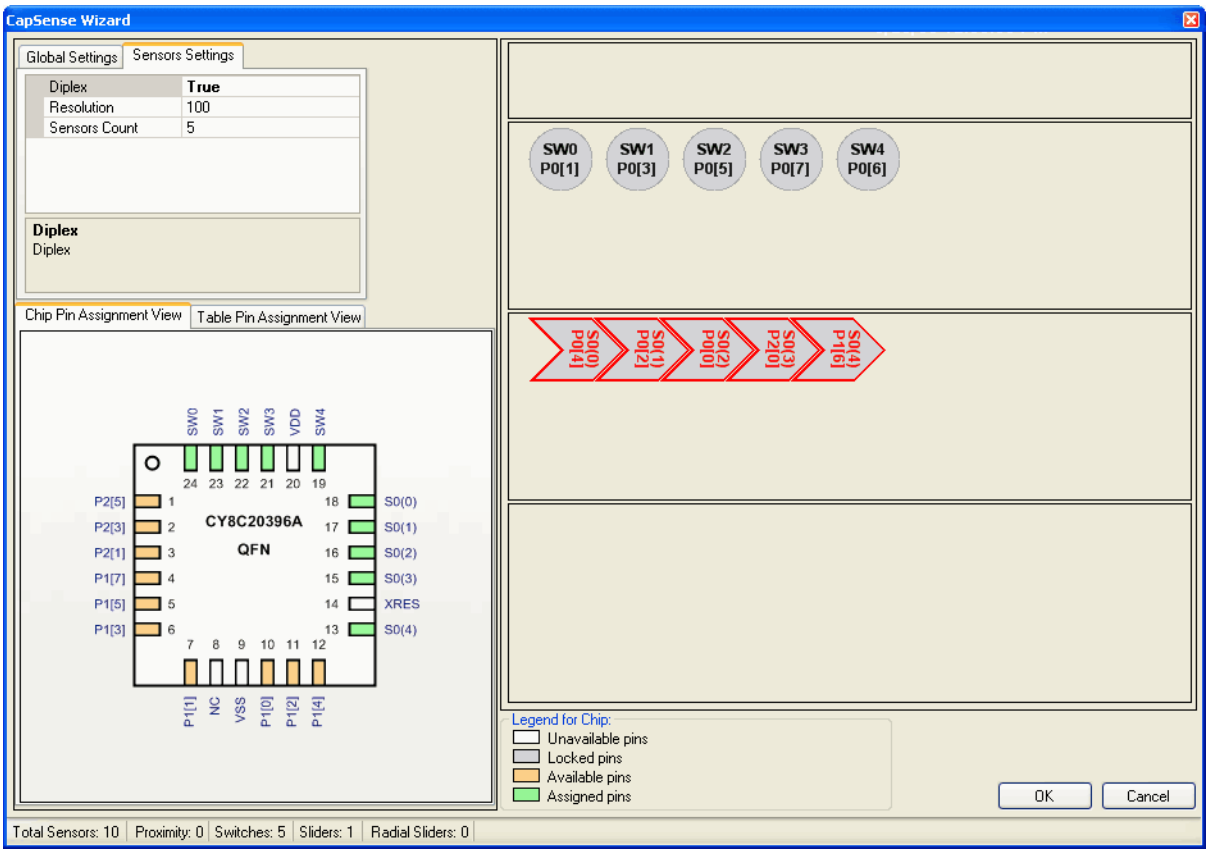
6. Type the output resolution. The minimum value is five. SmartSense attempts to interpolate the touch results to the specified resolution using the relative strength of adjacent segments. The software reports touch results on the slider between zero and the resolution - 1.
7. Select Diplex, if required. This maps the number of pins selected for sensors to twice as many sensor locations on the board. Only the first half of the diplex sensors is shown; the second half is automatically mapped as outlined in the previous section on Diplexing. See the Diplexing section to find Diplexing tables for pin connections.

8. Assign sensors to pins by dragging the sensor onto the pin in the Pin Assignment View. You can choose to drag sensors onto pins in the Chip Pin Assignment View or the Table Pin Assignment View. The I/O pin is green after selection and is no longer available. Change sensor assignments by dragging the port pin back to the uncommitted table. Avoid selecting pins already committed to other user modules.



9. Repeat for the remaining sensors. Click **OK** to accept data and return to PSoC Designer.

Sensor placement is now complete. Right click in the Device Editor window and select **Refresh** to update the pin connections.



To change the pin assignment, place your cursor on the assigned pin, click the pin, and drag and drop it outside the switches box. The pin is now unassigned and you can reassign it.

After completing the Wizard, click **Generate Application**. Based on your entries for sensor count, pin assignment, diplexing, and resolution, a set of tables is generated. The tables are located in SmartSense_Table.asm.

Sensor Table

The Sensor table consists of a 2-byte entry for each sensor. The first byte is the port number and the second byte is the bit mask for the bit (not the bit number). An example for a table with six sensors is:

```
SmartSense_Sensor_Table:
_SmartSense_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
```

Group Table

The Group table defines each of the groups of button sensors or sliders. There is one entry for each slider plus one for the independent button sensors. The first entry is always the independent buttons. Each entry is six bytes. The first byte is the index in the Sensor Table where the group starts. The second byte is the number of sensors in that group. The third byte signifies whether the slider is diplexed or not (4 is diplexed, 0 is not diplexed). The fourth, fifth, and sixth bytes are the fixed point multiplier by which the slider's centroid is scaled to achieve the resolution specified in the SmartSense Wizard.

```
SmartSense_Group_Table:
_SmartSense_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
```

Diplex Table

Diplex table scan order data is produced for a group that is a slider and with diplexing enabled. Otherwise, a label is created, but no data is placed. The table consists of two parts: sensor mapping for each slider, and a reference for each separate slider to its table. A typical example for an eight sensor slider is shown here:

```
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_Diplex_Table:
_SmartSense_Diplex_Table:
 db >DiplexTable_0, <DiplexTable_0
 db >DiplexTable_1, <DiplexTable_1
```

Parameters and Resources

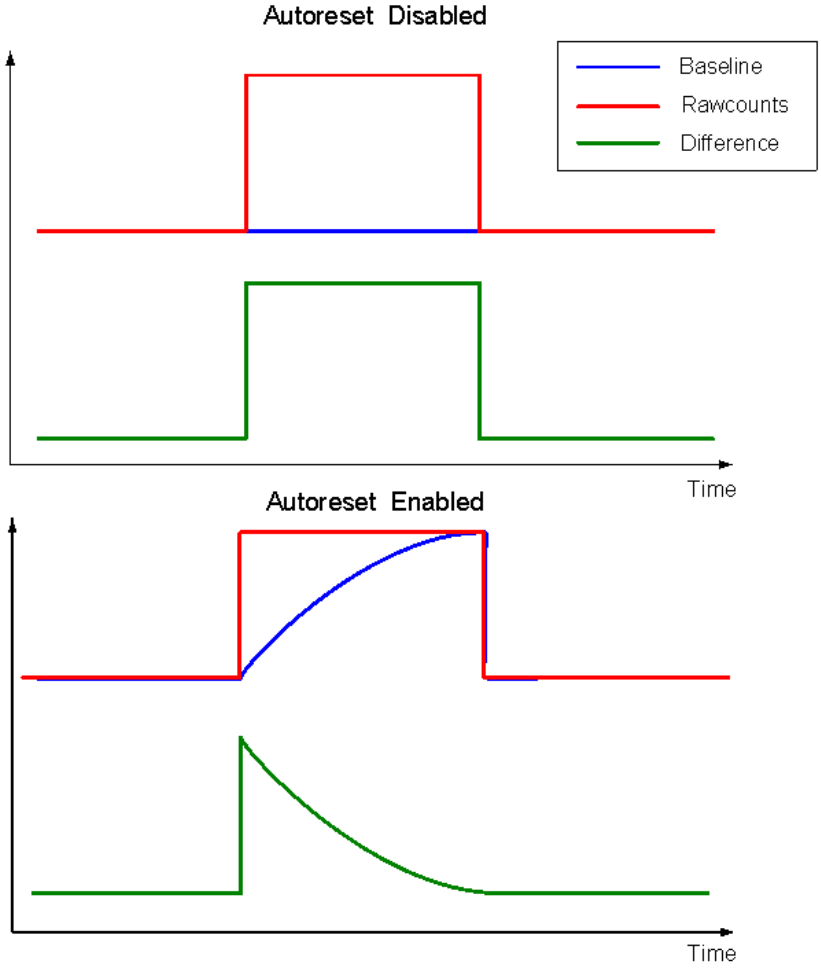
After completing configuration and I/O pin assignment in the SmartSense Wizard, the user module parameters must be set. Note that for any user module parameter change to take affect, the project must be regenerated.

Sensors Autoreset

This parameter determines whether the baseline is updated at all times or only when the signal difference is below the Noise Threshold. The default value for this parameter is "Disabled", that is, the baseline is updated only when the difference between the raw count and the baseline is below the Noise Threshold. Figure 8 illustrates this parameter's effect on baseline update. When Sensors Autoreset is set to **Enabled**, the baseline is always updated without regard to Noise Threshold. This limits the maximum activated time of sensors (typically to 5 - 10s). However, this provides the benefit of preventing sensors from getting stuck due to sudden rises in raw counts that are not caused by a touch. Such sudden rises can be caused by a large power supply voltage fluctuation, a high energy RF noise source, or rapid temperature change.

When Sensors Autoreset is Disabled, the baseline is updated only when the difference between raw count and baseline is below the Noise Threshold. This parameter should generally be left in its default "Disabled" state. See the Appendix section for additional explanation of this parameter.

Figure 8. Affect of the Sensor Autoreset Parameter on Baseline Update



Debounce

This parameter adds a debounce counter to the sensor active transition. For a sensor to transition from inactive to active, the difference count value must stay above finger threshold plus hysteresis for the number of samples specified by this parameter. The debounce counter is incremented by the bIsSensorActive or bIsAnySensorActive API functions.

Possible values are 1 to 255. A setting of '1' has no debounce, but gives the fastest response. The default setting is 3.

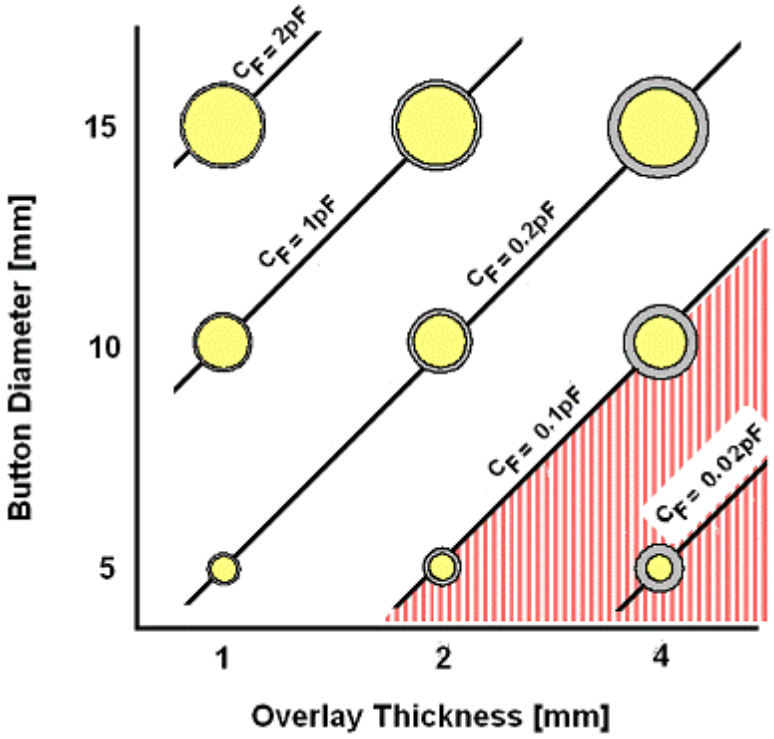
ShieldElectrodeOut

This parameter routes the shielding electrode signal to P0[7] or P1[2]. The default is None, which is applicable when no shielding electrode is used.

Sensor Sensitivity

Sensor Sensitivity sets the capacitance signal change (sensor response) in pF needed to activate a button sensor. The available settings are 0.1, 0.2, 0.3, and 0.4 pF. The default setting is 0.1 pF. Figure 9 shows the relationship between the button size, overlay thickness (acrylic plastic), and sensor response (CF), and can be used as a guide for setting Sensor Sensitivity. The Sensor Sensitivity should always be set at or below the sensor response indicated in Figure 9. This ensures robust operation. Note that the area shaded in red must be avoided, because the sensor response is below the minimum 0.1 pF that can be detected by SmartSense.

Figure 9. Relationship Between Button Size, Overlay Thickness, and Sensor Response in pF



This parameter setting is the default setting that is applied to all sensors in a design. Example 3 in the Sample Code section shows how the sensitivity of individual sensors can be set to a value other than the default specified by this parameter.

Scan Time of a Sensor

To maintain the consistent sensitivity (finger response) over a wide range of parasitic capacitance, the SmartSense User Module automatically finds out the hardware parameters of the user module. As a result, sensor scan time does not remain constant for a design in mass production; it could vary based on the parasitic capacitance variation of the PCB. The total scan time of a sensor is decided by four factors: parasitic capacitance of the sensor, IMO frequency, CPU operating frequency, and sensitivity level of the SmartSense User Module. The scan time of a sensor can be calculated using the following equation and with the help of the following tables:

Scan Time = Sampling Time + Processing Time

The values for sampling time with various IMO and sensitivity levels are available in the following tables:

Table 4. Sampling Time for a Sensor with IMO = 24 MHz

Sensitivity = 0.2 pf		Sensitivity = 0.3 pf		Sensitivity = 0.4 pf	
Cp (pf)	**Sampling Time (us)**	**Cp (pf)**	**Sampling Time (us)**	**Cp (pf)**	**Sampling Time (us)**
8 to 10	340	8 to 17	340	8 to 10	170
10 to 23	680	17 to 35	680	10 to 23	340
23 to 41	1360	35 to 41	1360	23 to 41	680
41 to 45	2730	41 to 45	2730	41 to 45	1360

Table 5. Sampling Time for a Sensor with IMO = 12 MHz

Sensitivity = 0.2 pf		Sensitivity = 0.3 pf		Sensitivity = 0.4 pf	
Cp (pf)	**Sampling Time (us)**	**Cp (pf)**	**Sampling Time (us)**	**Cp (pf)**	**Sampling Time (us)**
8 to 10	680	8 to 17.2	680	8 to 10	340
10.1 to 23	1360	17.2 to 35.8	1360	10 to 23	680
23 to 41	2730	35.8 to 41.8	2730	23 to 41	1360
41 to 45	5460	41.8 to 45	5460	41 to 45	2730

Table 6. Sampling Time for a Sensor with IMO = 6 MHz

Sensitivity = 0.2 pf		Sensitivity = 0.3 pf		Sensitivity = 0.4 pf	
Cp (pf)	**Sampling Time (us)**	**Cp (pf)**	**Sampling Time (us)**	**Cp (pf)**	**Sampling Time (us)**
8 to 11	680	8 to 10	680	8 to 10	680
11 to 23	1360	10 to 17	1360	11 to 23	1360
23 to 42	2730	17 to 35	2730	23 to 41	2730
42 to 45	5460	35 to 41	5460	41 to 45	5460
		41 to 45	10920		

Note The Cp Range provided in Tables 4,5, and 6 have an accuracy of +/- 10%.

The value for processing time with various CPU frequencies can be found in the following table:

Table 7. Processing time for a Sensor

CPUCLK (MHz)	Processing time (PT) in us
24	71
12	142
6	284
3	568

For example, if a CapSense system is designed with 24 MHz IMO frequency, 6 MHz CPU clock (IMO/4), and SmartSense sensitivity level of 0.3 pF, the scan time of the sensor that has parasitic capacitance ~15 pF can be calculated in the following manner:

The sampling time for 24MHz IMO and 0.3pF sensitivity configuration must be selected from Table 1.

Sampling time = 680 uS

The processing time for the above mentioned configuration (CPU clock of 6 MHz) must be selected from Table 4.

Processing time = 284 uS

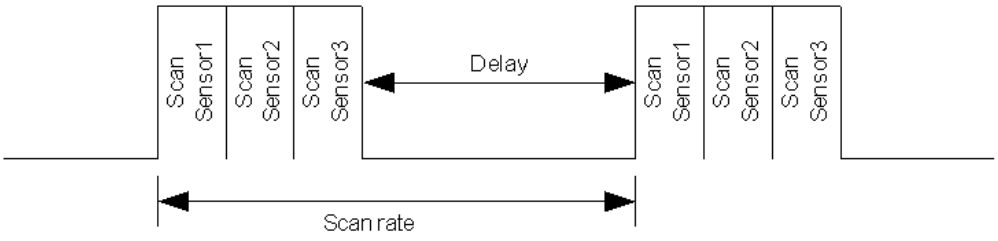
Scan time = 680 uS. + 284 uS. = 964 uS

In the above mentioned configuration, the scan time for more than one sensor is the sum of the scan time of each sensor.

Sensor Scan Rate Selection Guidelines

Scan rate is the rate at which sensors are scanned. An example of a 3-button design is shown in the following figure. All sensors in the design are scanned sequentially and there is a delay before the next sensor scan is initiated.

Figure 10. Typical Sensor Scan



To ensure proper working of the baseline, it is recommended to maintain a scan rate of 15 ms or more in a design. This indicates that a design with less number of sensors must add a delay to make the sensor scan rate equal to or greater than 15 ms. A design with more number of sensors may not need any delay as scanning all sensors itself may consume 15 ms. A good design may put the CapSense controller in sleep mode, instead of the firmware delay routine, to create a low power design.

Application Programming Interface

The Application Programming Interface (API) functions are provided as part of the user module to enable you to deal with the module at a higher level. This section specifies the interface to each function together with related constants provided by the include files.

Only one instance of this user module can be placed in the project and this also applies to loadable configurations. Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns the SmartSense_1 to the first instance of this user module in a given project. It can be changed to any unique value that follows the syntactic rules for identifiers. The assigned instance name becomes the prefix of every global function name, variable and constant symbol. In the following descriptions the instance name has been shortened to SmartSense for simplicity.

Note ** In this, as in all user module APIs, the values of the A and X register may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X before the call if those values are required after the call. This "registers are volatile" policy was selected for efficiency reasons and has been in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers must also ensure their code observes the policy. Though some user module API functions may leave A and X unchanged, there is no guarantee they may do so in the future.

For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

Entry Points are supplied to initialize the SmartSense, start it sampling, and stop the SmartSense. In all cases, the instance name of the module replaces the SmartSense prefix shown in the following entry points. Failure to use the correct instance name is a common cause of syntax errors.

API functions use different global arrays. Do not alter these arrays manually. You can inspect these values for debugging purposes, however. For example, you can use a charting tool to display the contents of the arrays. There several global arrays:

- SmartSense_waSnsBaseline[]
- SmartSense_waSnsResult[]
- SmartSense_waSnsDiff[]
- SmartSense_baSnsOnMask[]

SmartSense_waSnsBaseline[] – This is an integer array that contains the baseline data of each sensor. The array size is equal to the sensor count. The SmartSense_waSnsBaseline[] array is updated by these functions:

- SmartSense_UpdateAllBaselines()
- SmartSense_UpdateSensorBaseline()
- SmartSense_InitializeBaselines()

SmartSense_waSnsResult[] – This is an integer array that contains the raw data of each sensor. The array size is equal to the sensor count. The SmartSense_waSnsResult[] data is updated by these functions:

- SmartSense_ScanSensor()
- SmartSense_ScanAllSensors().

SmartSense_waSnsDiff [] – This is an integer array that contains the difference between the raw data and the baseline data of each sensor. The array size is equal to the sensor count.

SmartSense_baSnsOnMask[] – This is a byte array that holds the sensor on or off state (for buttons or sliders). SmartSense_baSnsOnMask[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). SmartSense_baSnsOnMask[1] contains the masked bits for sensors 8 through 15 (if they are needed), and so on. This byte array contains as many elements as are necessary to contain all the placed sensors. The value of a bit is 1 if the button is on and 0 if the button is off. The SmartSense_baSnsOnMask[] data is updated by these functions:

- SmartSense_bIsSensorActive()
- SmartSense_bIsAnySensorActive()

SmartSense_SnsErrorStatus[] – This is a byte array that reserves one bit for every CapSense sensor to indicate C_P out of the design limit. The size of this array variable is equal to the total number of sensors divided by eight bytes (similar to the _baSnsOnMask[] array variable in the existing user module to indicate the sensor on/off status). If the measured C_P of any sensor is outside the design limits, the bit corresponding to that sensor is set to indicate the error status.

SmartSense_Start

Description:

Initializes registers and starts the user module. This function must be called before calling any other user module functions.

C Prototype:

```
void  SmartSense_Start()
```

Assembly:

```
lcall  SmartSense_Start
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_Stop

Description:

Restores the CapSense block to its idle default configuration, releases the AMUX bus for other purposes, disables internal interrupts, and calls SmartSense_ClearSensors() to reset all sensors to their inactive state.

C Prototype:

```
void  SmartSense_Stop()
```

Assembly:

```
lcall  SmartSense_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_Resume

Description:

Resumes the user module operation after SmartSense_Stop call.

C Prototype:

```
void SmartSense_Resume()
```

Assembly:

```
lcall SmartSense_Resume
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_ScanSensor

Description:

Scans the selected sensor. Each sensor is uniquely identified by its position in the Sensor Table. This position or Sensor Number is assigned by the SmartSense Wizard.

C Prototype:

```
void SmartSense_ScanSensor(BYTE bSensor);
```

Assembly:

```
mov A,  bSensor
lcall SmartSense_ScanSensor
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects

**

SmartSense_ScanAllSensors

Description:

Scans all of the configured sensors by calling SmartSense_ScanSensor() for each sensor.

C Prototype:

```
void SmartSense_ScanAllSensors()
```

Assembly:

```
lcall  SmartSense_ScanAllSensors
```

Parameter:

None

Return Value:

None

Side Effects:

**

SmartSense_UpdateSensorBaseline

Description:

The historical count value, calculated independently for each sensor, is called the sensor's baseline. This baseline is updated using the “Bucket Method”.

The Bucket Method uses the following algorithm:

1. Each time the SmartSense_UpdateSensorBaseline() is called, a difference count is calculated by subtracting the previous baseline from the raw count value. This difference is stored in the SmartSense_waSnsDiff[] array.
2. If Sensors Autoreset is disabled, each time SmartSense_UpdateSensorBaseline() is called, the difference count is compared to the noise threshold. If the difference is below the noise threshold, it is accumulated into a virtual bucket. If the difference is above the noise threshold, the bucket is not updated. If Sensors Autoreset is enabled, the difference is accumulated into a virtual bucket regardless of the noise threshold parameter.
3. After the accumulated difference counts in the virtual bucket reach the BaselineUpdateThreshold, the baseline is incremented by one and the bucket is reset to 0.
4. If the difference count is below the noise threshold, the value held in the waSnsDiff[] array is reset to 0. As a result, this array does not contain elements with values greater than 0, but below the NoiseThreshold.

C Prototype:

```
void SmartSense_UpdateSensorBaseline(BYTE bSensorNum)
```

Assembly:

```
mov   A,   bSensorNum
lcall  SmartSense_UpdateSensorBaseline
```

Parameter:

A => Sensor Number

Return Value:

None

Side Effects:

**

SmartSense_UpdateAllBaselines

Description:

Uses the SmartSense_bUpdateSensorBaseline() function to update the baselines for all sensors.

C Prototype:

```
void SmartSense_UpdateAllBaselines()
```

Assembly:

```
lcall  SmartSense_UpdateAllBaselines
```

Parameter:

None

Return Value:

None

Side Effects:

**

SmartSense_bIsSensorActive

Description:

Checks the difference count array for the given sensor compared to its finger threshold. Hysteresis is taken into account. The Hysteresis value is added or subtracted from the finger threshold based on whether the sensor is currently on. If it is active, the threshold is lowered. If it is inactive, the threshold is raised. This function also updates the sensor's bit in the SmartSense_baSnsOnMask[] array.

C Prototype:

```
BYTE  SmartSense_bIsSensorActive(BYTE bSensorNum)
```

Assembly:

```
mov  A,  bSensorNum
lcall SmartSense_bIsSensorActive
```

Parameters:

bSensor A => Sensor Number

Return Value:

Returns value of 1 if active, 0 if not active

A => 1 – Selected sensor is active, 0 – Selected sensor is not active.

Side Effects:

**

SmartSense_bIsAnySensorActive

Description:

Checks the difference count array for all sensors compared to their finger threshold. Calls SmartSense_bIsSensorActive() for each sensor so that the SmartSense_baSnsOnMask[] array is up to date after calling this function.

C Prototype:

```
BYTE  SmartSense_bIsAnySensorActive()
```

Assembly:

```
lcall SmartSense_bIsAnySensorActive
```

Parameters:

None

Return Value:

Returns value of 1 if active, 0 if not active

A => 1 – One or more sensors are active, 0 – No sensors are active.

Side Effects:

**

SmartSense_wGetCentroidPos

Description:

Checks a linear slider array for a centroid. If there is a centroid, the offset and length are stored in temporary variables and the centroid position is calculated to the resolution specified in the SmartSense Wizard. This function is available only if slider is defined by the SmartSense Wizard.

C Prototype:

```
WORD  SmartSense_wGetCentroidPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall SmartSense_wGetCentroidPos
```

Parameters:

bSnsGroup A => Group Number

This parameter is the Group Number of the slider. Group 0 is always the independent buttons. Sliders are contained in group 1 and higher.

Return Value:

Position value of the slider, LSB in A and MSB in X.

Side Effects:

This routine modifies the difference counts by subtracting the noise threshold value and should only be called once after each scan to avoid getting negative difference values. If your application monitors difference count signals, call this routine after difference count data transmission.

If any slider sensor is active, the function returns values from zero to the Resolution value set in the Wizard. If no sensors are active, the function returns -1 (FFFFh). If an error occurs during execution

of the centroid/diplexing algorithm, the function returns -1 (FFFFh). You can use the SmartSense_blsSensorActive() routine to determine which slider segments are touched.

SmartSense_wGetRadialPos

Description:

Checks a radial slider array for a centroid. If there is a centroid, the centroid position is calculated to the resolution specified in the SmartSense Wizard.

C Prototype:

```
WORD  SmartSense_wGetRadialPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall SmartSense_wGetRadialPos
```

Parameters:

bSnsGroup A => Group Number

This parameter is the Group Number of radial slide. You can get its number through the SmartSense Wizard on the left side of radial slider representation (for example, "s2" means the radial slider Group Number is 2).

Return Value:

Position value of the radial slider, LSB in A and MSB in X.

Side Effects:

This routine modifies the difference counts by subtracting the noise threshold value and should only be called once after each scan to avoid getting negative difference values and baseline update. If your application monitors difference count signals, call this routine after difference count data transmission.

If any slider sensor is active, the function returns values from zero to the Resolution value set in the Wizard. If no sensors are active, the function returns -1 (FFFFh). If an error occurs during execution of the centroid algorithm, the function returns -1 (FFFFh). You can use the SmartSense_blsSensorActive() routine to determine which slider segments are touched.

SmartSense_wGetRadialInc

Description:

Returns actual finger shift, the difference between current and previous finger positions. This function works in conjunction with SmartSense_wGetRadialPos().

C Prototype:

```
WORD  SmartSense_wGetRadialInc(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall SmartSense_wGetRadialInc
```

Parameters:

bSnsGroup A => Group Number

This parameter is the Group Number of radial slide. You can get its number through the SmartSense Wizard on the left side of radial slider representation (for example, "s2" means the radial slider Group Number is 2).

Return Value:

Finger shift value, positive if clockwise and negative if anti-clockwise, LSB in A and MSB in X.

Finger shift value is the difference between current and previous finger positions.

Side Effects:

The routine must be called only after calling SmartSense_wGetRadialPos(), because it uses internal data stored in global variables by the latter.

SmartSense_InitializeSensorBaseline

Description:

Loads the SmartSense_waSnsBaseline[bSensor] array element with an initial value by scanning the selected sensor. The raw count value is copied into the baseline array element for the selected sensor. This function can be used to reset the baseline of an individual sensor.

C Prototype:

```
void SmartSense_InitializeSensorBaseline(BYTE bSensorNum)
```

Assembly:

```
mov A, bSensorNum
lcall SmartSense_InitializeSensorBaseline
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

SmartSense_InitializeBaselines

Description:

Loads the SmartSense_waSnsBaseline[] array with initial values by scanning each sensor. The raw count values are copied in to baseline array for each sensor.

C Prototype:

```
void SmartSense_InitializeBaselines()
```

Assembly:

```
lcall SmartSense_InitializeBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_ClearSensors

Description:

Clears all sensors to the non-sampling state by sequentially calling SmartSense_wGetPortPin() and SmartSense_DisableSensor() for each of the sensors.

C Prototype:

```
void  SmartSense_ClearSensors()
```

Assembly:

```
lcall  SmartSense_ClearSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_wReadSensor

Description:

Returns the key Raw scan value in A (LSB) and X (MSB).

C Prototype:

```
WORD  SmartSense_wReadSensor(BYTE bSensor)
```

Assembly:

```
mov A, bSensor
lcall SmartSense_wReadSensor
```

Parameters:

A => Sensor Number

Return Value:

Scan value of sensor, LSB in A and MSB in X.

Side Effects:

**

SmartSense_wGetPortPin

Description:

Returns the port number and pin mask for a given sensor. The passed parameter indexes and selects the data from the SmartSense_Sensor_Table2[]. The return value can be passed to the SmartSense_EnableSensor(), SmartSense_DisableSensor().

C Prototype:

```
WORD  SmartSense_wGetPortPin(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall SmartSense_wGetPortPin
```

Parameters:

bSensor – the range is from 0 to (n – 1), where n is the total number of sensors set in the SmartSense Wizard plus the number of sensors included in sliders. The sensor number is used by SmartSense_wGetPortPin() to determine port and bit mask for the selected active sensor.

Return Value:

A => Sensor Bitmask

X => Port Number

Side Effects:

**

SmartSense_EnableSensor

Description:

Configures the selected sensor to measure during the next measurement cycle. The port and sensor can be selected using the SmartSense_wGetPortPin() function, with the port number and sensor bitmask loaded into X and A, respectively. Drive modes are modified to place the selected port and pin into Analog High Z mode and to enable the correct Analog Mux Bus input.

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Bitmask

X => Port Number

bMask - Bitmask for given sensor

bPort - Port Number for given key

Return Value:

None

Side Effects:

**

SmartSense_DisableSensor

Description:

Disables the sensor selected by the SmartSense_wGetPortPin() function. The drive mode is changed to Strong (001). This effectively grounds the sensor. The connection from the port pin to the Analog-MuxBus is turned off. The function parameters are returned by SmartSense_wGetPortPin() function.

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Bitmask

X => Port Number

bMask - Bitmask for given sensor

bPort - Port Number for given key

Return Value:

None

Side Effects:

**

SmartSense_GetSnsParasiticCapacitance

Description:

This API returns sensor parasitic capacitance in pF.

C Prototype:

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

Parameters:

bSensor A => Sensor Number

Return Value:

A => sensor parasitic capacitance in pF

Side Effects:

**

Sample Firmware Source Code

Example 1. This code starts the user module and continuously scans the sensors. The communication section can be used to communicate values to a PC charting tool.

```
//-------------------------------------------------------------------------
// Sample C code for the SmartSense 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_Start();
SmartSense_InitializeBaselines() ; //scan all sensors first time, init baseline
  //
// Loop Forever
//
while (1) {
SmartSense_ScanAllSensors(); //scan all sensors in array (buttons and sliders)
    SmartSense_UpdateAllBaselines(); //Update all baseline levels;

//detect if any sensor is pressed
if(SmartSense_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_waSnsResult[x] <- Raw Counts
// OUTPUT SmartSense_waSnsDiff[x] <- Difference
// OUTPUT SmartSense_waSnsBaseline[x] <- Baseline
// OUTPUT SmartSense_baSnsOnMask[x] <- Sensor On/Off
}
}
```

Example 2. This code starts the user module and continuously scans the sensors, except unlike Example 1, looping through the sensors is done in the user code. The communication section can be used to communicate values to a PC charting tool.

```
//-------------------------------------------------------------------------
// Sample C code for the SmartSense 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_Start();
SmartSense_InitializeBaselines() ; //Scan all sensors first time, init baseline

while (1) //Loop forever
  {
for(bIndex=0; bIndex < SmartSense_TotalSensorCount; bIndex++)//Loop through all sensors
```

```
{
   SmartSense_ScanSensor(bIndex);     // Scan Sensors
   SmartSense_UpdateSensorBaseline(bIndex); // Run baseline filter
if(SmartSense_bIsSensorActive(bIndex))
{
// Add user code here to process the sensor touching
}

}

//detect if any sensor is pressed

    // now we are ready to send all status variables to chart program
    // communication here
//
// OUTPUT SmartSense_waSnsResult[x] <- Raw Counts
// OUTPUT SmartSense_waSnsDiff[x] <- Difference
// OUTPUT SmartSense_waSnsBaseline[x] <- Baseline
// OUTPUT SmartSense_baSnsOnMask[x] <- Sensor On/Off
}
}
```

Configuration Registers

Table 8. Block CapSense, Register: CS_CR0

Bit	7	6	5	4	3	2	1	0
Value	0	0	SmartSense_PRSCLK	0	1	0	0	EN

Table 9. Block CapSense, Register: CS_CR1

Bit	7	6	5	4	3	2	1	0
Value	1	Scan Speed		0	0	0	0	0

Power: 0x01 Turns on power to analog block. 0x00 Turns off power to analog block.

Table 10. Block CapSense, Register: CS_CR2

Bit	7	6	5	4	3	2	1	0
Value	1	0	0	0	0	1	0	0

Table 11. Block CapSense, Register: CS_CR3

Mode/Bit	7	6	5	4	3	2	1	0
Value	0	1	1	1	0	0	0	0

Table 12. Block CapSense, Register: CS_CNTH

Bit	7	6	5	4	3	2	1	0
Data Out MSB								

Table 13. Block CapSense, Register: CS_CNTL

Bit	7	6	5	4	3	2	1	0
Data Out LSB								

Table 14. Block CapSense, Register: PRS_CR

Mode/Bit	7	6	5	4	3	2	1	0
Value	1	0	8/12 bit	1	Prescaler			

Table 15. Block Timer, Register: PT1_CFG

Mode/Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	1	Start

Table 16. Block Timer, Register: PT1_DATA0

Mode/Bit	7	6	5	4	3	2	1	0
Value	Data LSB							

Table 17. Block Timer, Register: PT1_DATA1

Mode/Bit	7	6	5	4	3	2	1	0
Value	Data MSB							

Appendix

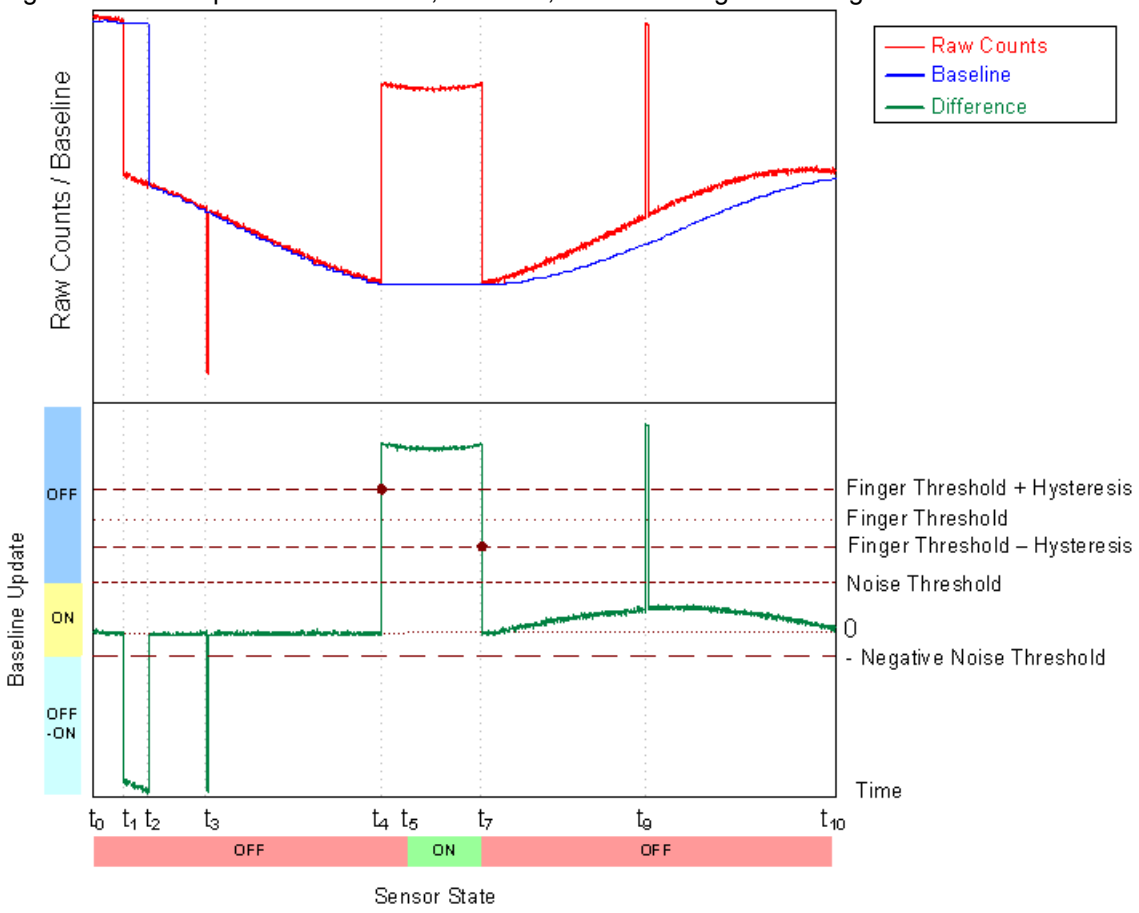
The following sections contain information beyond what is usually included in user module datasheets. The detailed information was developed by Cypress engineers to help you successfully design CapSense applications. Some of this information may be moved into application notes in the future.

Interaction of SmartSense Parameters

The following figures illustrate the baseline update and decision logic operation, and can be useful to better understand how to set user module parameters for optimum performance. Figure 10 illustrates system operation when the Sensors Autoreset parameter is set to **Disabled**. Figure 11 illustrates the Sensors Autoreset parameter **Enabled**. The Finger Threshold, Noise Threshold, Hysteresis, and Negative Noise Threshold are shown together with Difference signal (Raw Count - Baseline). Data was collected during some artificial tests that demonstrate system operation at both slow and rapid raw data changes.

The slow changes can be caused by temperature or humidity variations, and the rapid changes can be triggered by a sensor touch, an ESD event, or the influence of a strong RF field.

Figure 11. Example of Raw Counts, Baseline, Difference Signals Change With SensorsAutoreset Set to Disabled



At t_0, the raw counts are close to the baseline level and start to drop slowly because of humidity or temperature changes. Because the raw count change between two successive conversions does not exceed the NegativeNoiseThreshold parameter (by absolute value), the baseline is updated by tracking the Raw Count minimum value, holding the lower value of raw count signal.

At t_1, the raw count drops sharply and the negative difference exceeds the NegativeNoiseThreshold. This situation can happen if the device is powered on when a finger is on the sensor and then the finger is removed after a period of time. At this time, the baseline update mechanism is frozen and an internal timeout counter is activated. The baseline is reset when the difference signal is below the NegativeNoiseThreshold for LowBaselineReset samples. This happens at t_2.

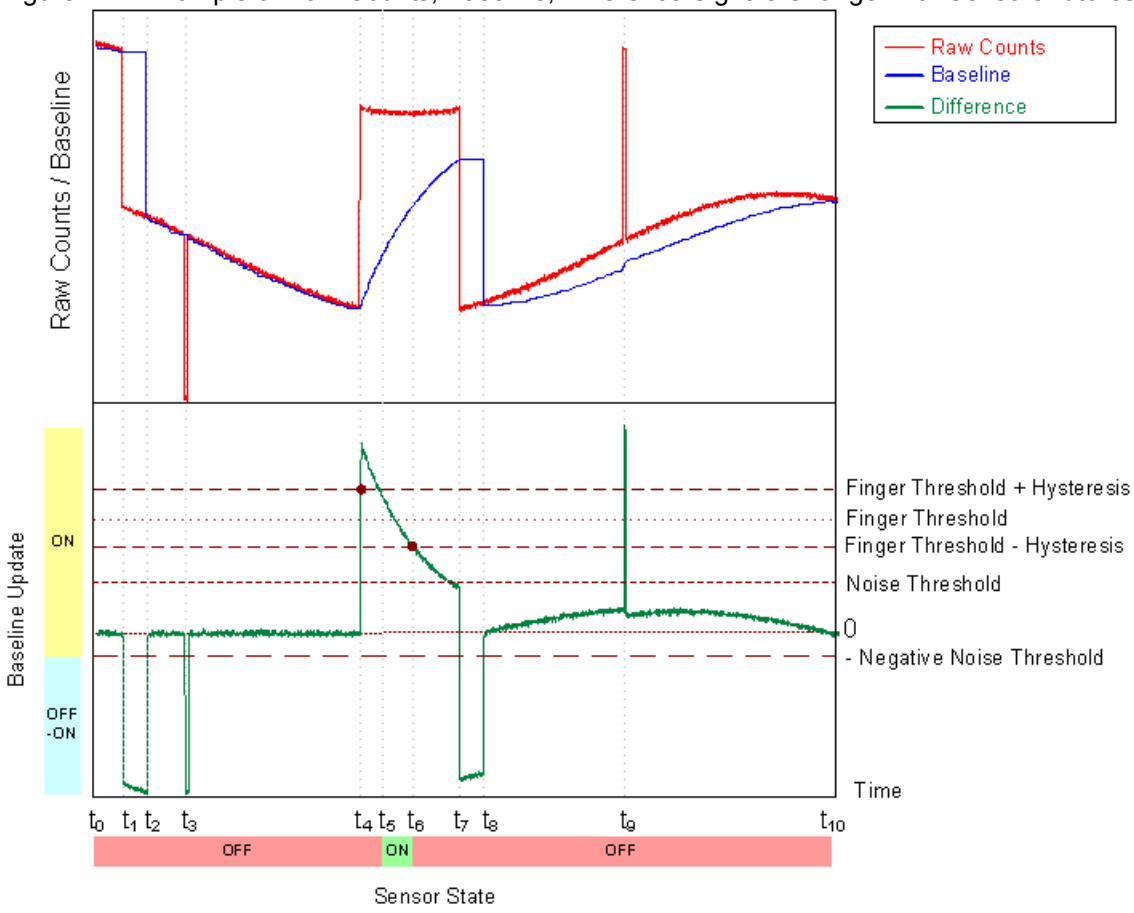
The second large negative difference signal spike happens at t_3. this spike may have been triggered by an ESD event, for example. Because the spike duration in the sample count is less than the LowBaselineReset parameter, the baseline is kept on hold and the spike is filtered. This prevents a false baseline reset and the resulting false touch detection.

The sensor is touched at t_4. When the difference signal exceeds the FingerThreshold + Hysteresis value, the internal debounce counter is activated. If the signal exceeds this value for more than Debounce samples, the sensor state is set to on. This happens at t_5. The sensor state reverts back to the off state immediately when the difference signal drops below the FingerThreshold – Hysteresis level at t_7. The

short positive spike at t_9 is filtered by the debounce counter because the spike duration in sample units does not exceed the Debounce value.

The raw count drifts up slowly between t_7 and t_{10}. The baseline is updated using the bucket algorithm when the difference signal is below the NoiseThreshold (SensorsAutoreset is set to Disabled), the difference signal is proportional to the drift rate. It is possible to control the baseline update speed using the BaselineUpdate Threshold parameter. Lower parameter values give faster baseline update speeds.

Figure 12. Example of Raw Counts, Baseline, Difference Signals Change With SensorsAutoreset Set to Enabled



The system operation in Figure 11 is similar to the operation in the previous case, except for the following differences:

- The touch duration is decreased because of the active baseline update algorithm while the sensor is touched, t_6.
- After the finger is removed, the baseline is reset after LowBaselineReset samples (t_8), which blocks touch detection for a short time. This serves as an additional debounce mechanism.

Version History

Version	Originator	Description
1.00	DHA	Initial version.
1.10	DHA	1. Added conditional code to prevent build errors when no sensor is added. 2. Added descriptions of the SmartSense_wGetPortPin(), SmartSense_EnableSensor(), and SmartSense_DisableSensor() API functions in this user module datasheet.
1.20	DHA	1. Transferred the Diplex Table from "AREA UserModules" to "AREA lit". 2. Set the default "DiplexTable" parameter value to 0x0112. 3. Added the "DiplexUsed" parameter to improve code compression. 4. Updated the SmartSense_bIsAnySensorActive() API function to address an issue with returned sensors mask. 5. Moved the user module API functions SmartSense_Start(), SmartSense_UpdateAllBaselines(), and SmartSense_UpdateSensorBaseline() from the "UserModules" to the "text AREA". 6. Added new parameters: Approaching speed, Proximity IIR Filter, and Proximity IIF Filter Coefficient. 7. Updated the CalibrateSingleSensor() API function. 8. Updated the SmartSense_Calibrate API to fix incorrect variables initialization. 9. Renamed "Switches" to "Buttons" in the wizard GUI. 10. Updated the SmartSense_Start function for the Proximity sensor. 11. Updated the SmartSense_wGetCentroidPos function to improve the functionality of the Slider centroid algorithm. 12. Updated the InternalMainScanSensor() API to modify CPU_CLK, stop and enable PRS, and disable and enable interrupts. 13. Updated the “Features and Overview” and “Sensor Scan Time” sections.

Version	Originator	Description
1.30	DHA	1. Updated area declarations to support Imagecraft optimization. 2. The LoadParameters, SetPrescaler, SetScanMode, and Resume APIs were made identical to the SmartSense_EMC User Module. 3. Deleted SmartSenseTUNE.c. 4. For CY8C21x34 devices, the LoadParameters, SetDefaultFingerThreshold, and UpdateSensorSignal APIs are moved to asm from C. 5. For CY8C20xx6 devices, the LoadParameters API is moved to asm. 6. Updated the wizard to properly indicate the available pins. 7. Added conditional compilation in case 0 sensors is used. 8. Updated the SmartSense_ClearSensors API description in this datasheet. 9. Updated the user module wizard help. Added a description of the slider resolution parameter min/max values. 10. Added the M8C_ClearWDT macro to the InternalMainScanSensor function code to resolve watchdog timer coexistence.
1.40	DHA	1. Corrected resolution value calculation in UM wizard to address the error after change in diplexing. 2. Updated code for baSensorSensitivity array initialization. 3. For CY8C21x34 devices, modified ScanSensor API to avoid Watchdog reset. Exported LoadFixedParameters API function from the assembly file. Removed redundant InitializeBaselines API function call from Start API function. 4. For CY8C20x66 devices, updated code for baSensorSensitivity array initialization. 5. Removed CS_MISC register use. 6. Datasheet updates: Added Cmod section; added note about wizard behavior when Cmod is not assigned.
1.50	DHA	1. Increased number of proximity sensors to max allowed pins. 2. Explained limitations of dynamic reconfiguration in the datasheet. 3. Added CYRF89435 and CY8C200x5 device support.

Version	Originator	Description
1.60	MYKZ	1. Added Resume() function to User Module API. 2. Fixed problem with saving information for sliders. 3. Updated baseline algorithm to check for negative difference counts. 4. Added GetSnsParasiticCapacitance to User Module API. 5. Added build error message when user attempts to build project without first calling the user module wizard. 6. Removed default value for feedback resistor pin and fixed feedback pin handling in User Module wizard. 7. Updated Stop() function to support Rb pins. 8. Added CY7C69000 support. 9. Updated Precharge() function to correct Cmod connection to GND. 10. Limited slider resolution to 100.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.

Copyright © 2010-2013 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.