SmartSense_EMCPlus Datasheet SmartSense_EMCplus V 1.20

Copyright © 2012-2013 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks				API Memory		External I/O
	CapSense®	I²C/SPI	Timer	Comparator	Flash	RAM	
CY8C20xx7S, CY8C20055							
User Module	1	–	1	1	3494	85	1
Slider APIs	–	–	–	–	696	92	0
Each Sensor	–	–	–	–	6	31	1

Features and Overview

- Implements CapSense capacitive sensing in the CY8C20xx7S family of PSoC® devices
- Auto-tunes configurable system parameters in runtime to account for sensor, IC, and PCB characteristics
- Supports as many as 35 capacitive sensors and six sliders
- Supports parasitic sensor capacitance range of 5 pF to 45 pF
- Detects touches as low as 0.1 pF, that is, a finger touch 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 enables two linear 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 in the presence of water film
- Enables guided sensor and pin assignments using the SmartSense_EMCplus™ Wizard
- The SmartSense Electromagnetic Compatible (SmartSense_EMCplus) User Module provides superior noise immunity against external noise
- The CY8C20055 family does not support sliders

Note This user module supports only C language projects: Assembly language (ASM) 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_EMCplus User Module.
3. Right-click the SmartSense_EMCplus User Module in the Workspace Explorer to access the SmartSense_EMCplus Wizard (refer to the SmartSense_EMCplus Wizard section).
4. Set the required number of sensors, sliders, and radial sliders.
5. For sliders, enter the parameters specific to the 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_EMCplus User Module in the Workspace Explorer to access the Properties list. Enter the pin that will be used to shield the sensor, if required (refer to the Parameters and Resources section).
9. Generate the application and switch to the Application Editor.
10. Adapt the sample code to implement independent sensors, sliding sensors, and a touchpad.
11. Program the PSoC on the target board with the .hex file generated by PSoC Designer™.

Introduction

The SmartSense_EMCplus User Module enables CapSense capacitive sensing. CapSense is a human interface technology that detects the capacitance of the human body. The technology relies on sensors, which 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. The following attributes make CapSense an attractive alternative to mechanical input devices, such as push buttons and potentiometers:

- Clean, aesthetically pleasing designs
- Reduced form factor for lower-end products.
- Addition of advanced user interface features, such as LED effects and proximity sensing.
- Improved reliability because no components wear or have finite life cycle.
- Improved spill resistance due to a lack of mechanical interface penetrations.
- Reduced tooling because penetrations or other mechanical features needed for mechanical input devices are eliminated.

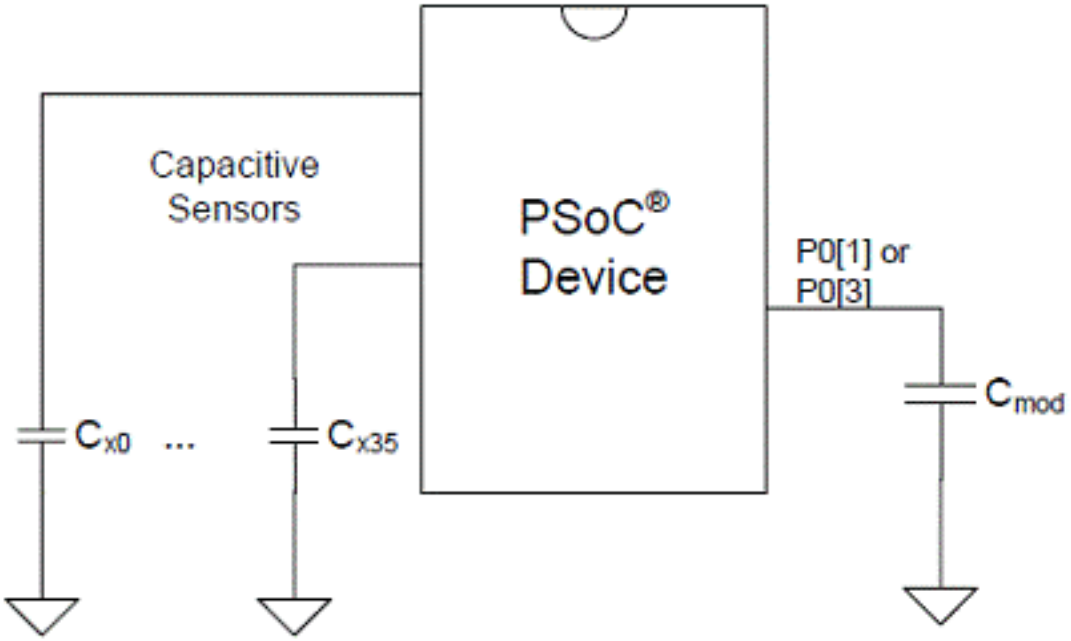
Like all CapSense solutions, SmartSense_EMCplus offers superior immunity to conducted and radiated noise interference, with the additional benefit of auto-tuning. Auto-tuning gives run time compensation for IC and PCB characteristics and environmental changes to ensure reliable sensor operation. For example, prototype and production PCBs frequently exhibit different material properties that affect sensor parasitic capacitance. This effect can be significant enough to require retuning the CapSense system parameters. Auto-tuning automatically compensates for such changes without the need to retune. Furthermore, the auto-tuning algorithms in SmartSense_EMCplus continuously monitor sensor data to compensate for environment conditions, such as temperature and ambient noise level, to maintain proper sensor function.

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

PCB Level

Figure 1 shows a schematic of the SmartSense_EMCplus 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.

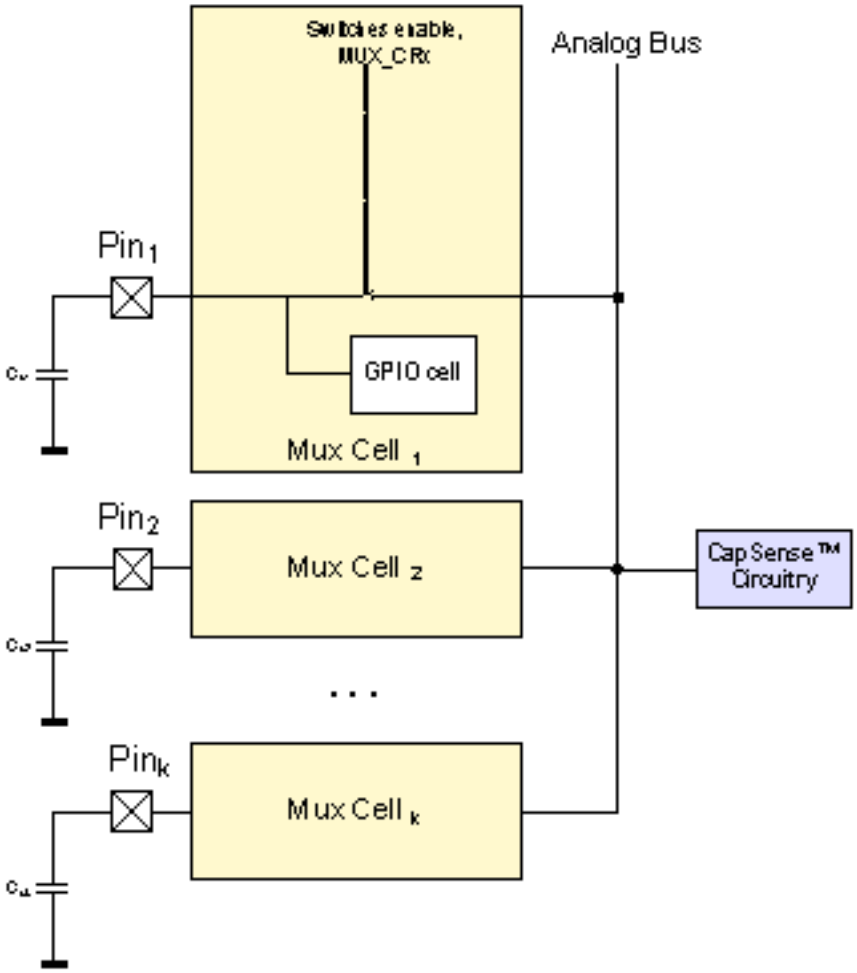
Figure 1. SmartSense_EMCplus Schematic



IC Level

The CY8C20xx7S devices have an Analog MUX (AMUX) Bus that enables connecting capacitive sensing analog circuitry to any PSoC pin. The SmartSense_EMCplus 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 (see Figure 2).

Figure 2. CY8C20xx7S AMUX Block Diagram



Software

The attributes of the SmartSense_EMCplus 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.
- To make sensor state decisions and to compensate for environmental changes, API functions analyze in runtime the raw count value from the capacitance converter circuitry.
- 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 linear slider diplexing to allow one I/O pin to be routed to two physical sensors. As a result, the number of I/O pins consumed for a given number of slider elements is cut in half.

Recommended Reading

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

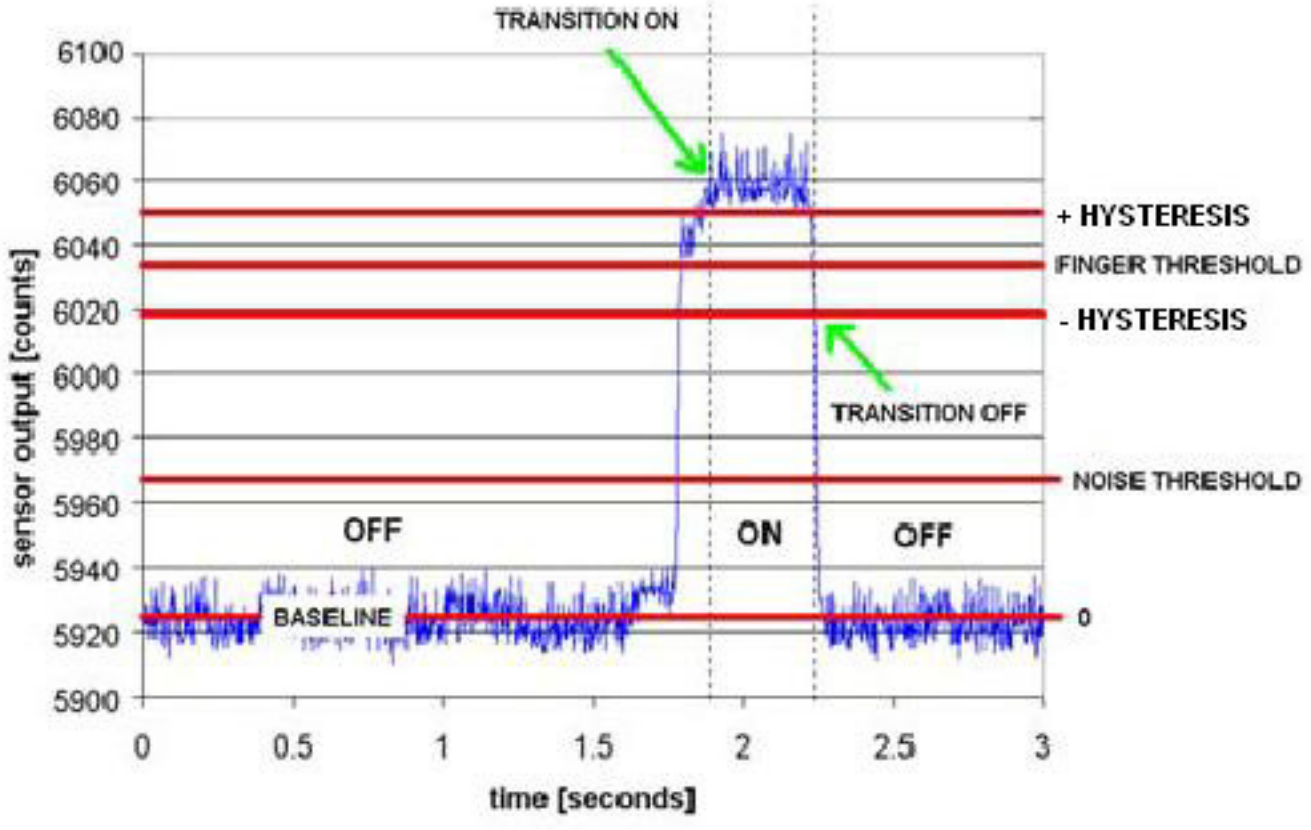
- *CY8C20xx7 Series PSoC Mixed Signal Array Technical Reference Manual, section: CapSense System*
- CY8C20xx7 Device Datasheet
- Getting Started with CapSense

Capacitance Sensing Implementation

Buttons

Analogous to mechanical push buttons, CapSense buttons are. used for discreet controls, These include on/off switches, function keys, and menu keys. API functions monitor the capacitance signals from each sensor and compare them to threshold levels calculated by the SmartSense_EMCplus auto- tuning algorithms. When a sensor is touched, its capacitance signal increases. If the SmartSense_EMCplus decision logic determines that the increase is sufficient, the sensor is activated. Figure 3 shows a typical signal (blue line) from a sensor when it is being activated. SmartSense_EMCplus automatically sets the thresholds (red lines) based on your input to give the desired system behavior. In addition, the Manual Mode is accessible that lets you set the finger threshold for each sensor.

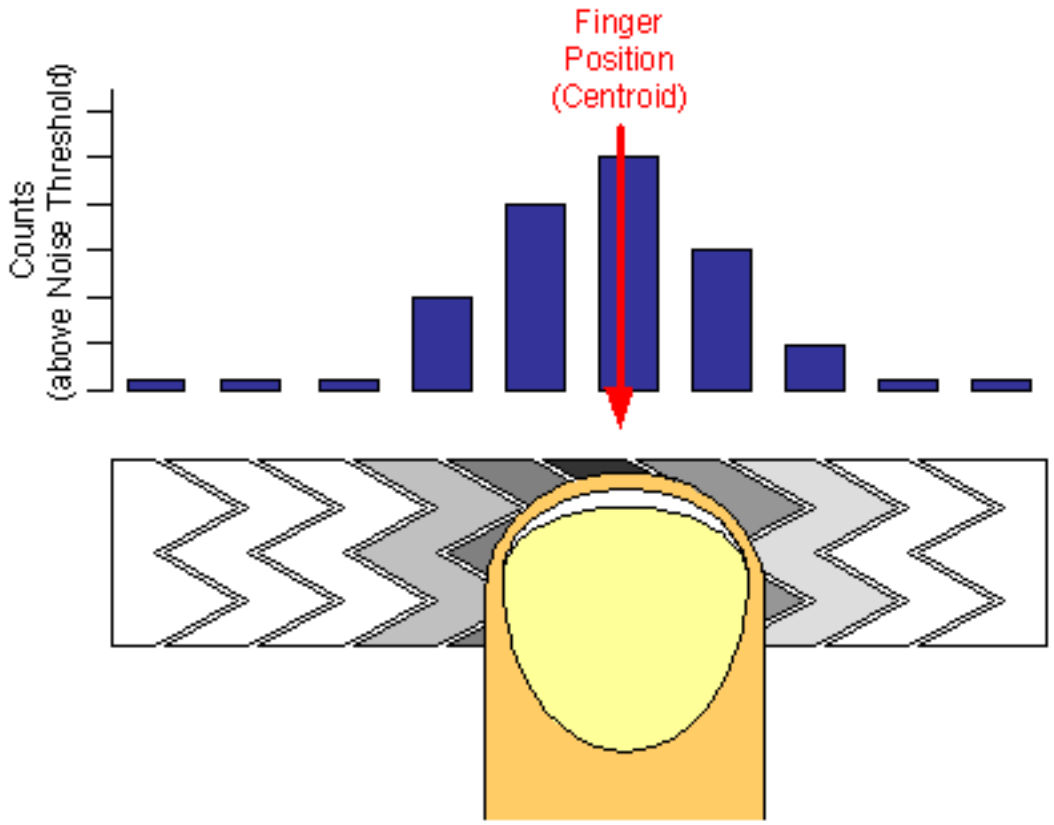
Figure 3. Capacitance Signal from a Sensor when it is Being Activated



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, and speed controls. 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 the capacitance signal. This action is shown in Figure 4. To find the exact position of the touch, compute 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.

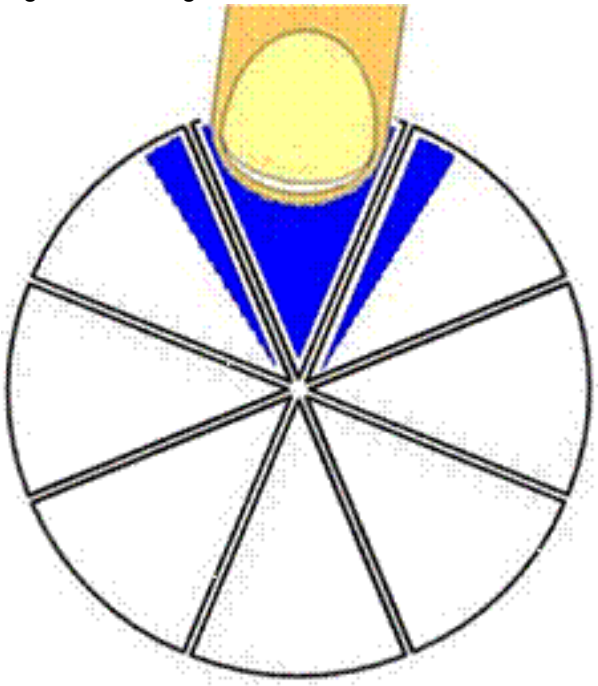
Figure 4. Interpolated Centroid Position of a Finger on a Slider



Radial Sliders

SmartSense_EMCplus 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 the signals 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_EMCplus User Module has two special API functions for radial sliders: The first function SmartSense_EMCplus_wGetRadiaPos() returns the centroid location, and the second function, SmartSense_EMCplus_wGetRadialInc() returns the finger shift in resolution units. When the finger moves in a clockwise direction, SmartSense_EMCplus_wGetRadialInc() returns a positive offset. The reference point (0) is located in the center of the first sensor. The Resolution is limited to (number of pins used for sensors – 1) x 2^8 – 1 or (2 x number of pins used for sensors – 1) x 2^8 – 1 for diplexed sliders.

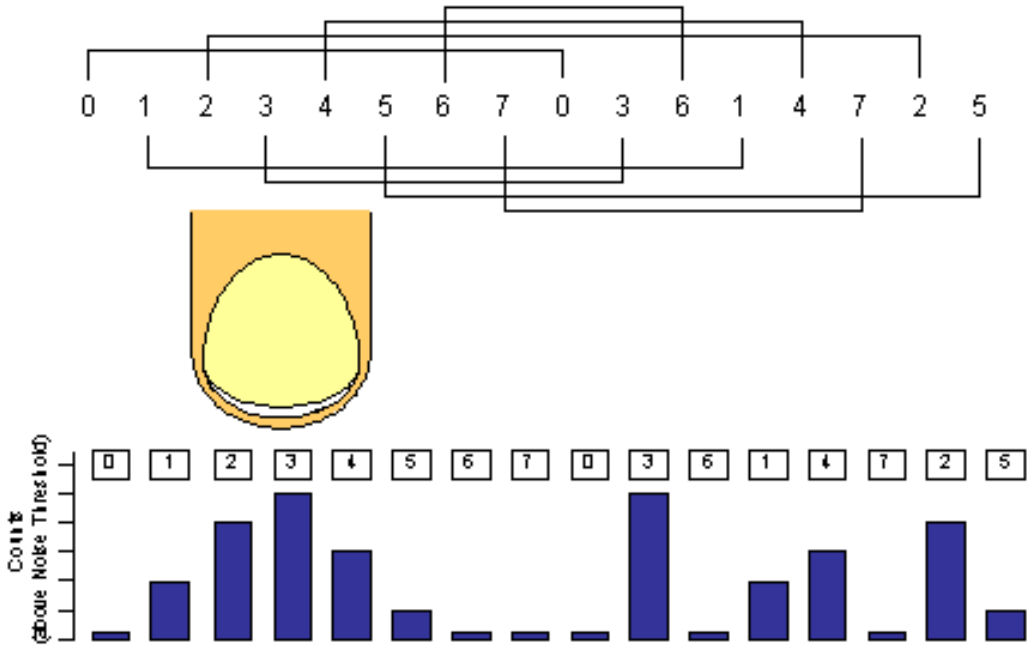
Figure 5. Finger touches Radial Slider



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 location is mapped according to the port pin assigned in the SmartSense_EMCplus Wizard. The second (or upper) half of the physical sensor location is automatically mapped using the pattern shown in Figure 6.

Figure 6. Indexing of Diplexed Slider Array by SmartSense_EMCplus



The close proximity of strong signals in the 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 the 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.

Ensure that the mapping of the 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_EMCplus Wizard when you select diplexing. Table 1 lists the diplexing sequences for as many as 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
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

Total Slider Segment Count	Segment Sequence
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 its physical length. However, 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 physical slider segments to increase the physical length. 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. That ensures the proper operation of the diplex slider. For example, in the case of a 10-segment slider (five sensors), two slider segments derived from sensor 3 are separated by only two sensors (sensors 4 and 0). In that case, to ensure the proper operation of the slider, a finger touch must not completely cover more than two sensor segments. For a 12-segment slider, one finger touch must not cover more than three segments. Similarly, for an 18- segment slider, one finger touch must not completely cover more than four segments.

Interpolation and Scaling

In slider applications, determine the position with finer granularity than the physical pitch of the sensors. SmartSense_EMCplus accomplishes that 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. First, the array is 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 using Equation 1:

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 one calculation and report the result directly in the desired scale. This is handled in the high-level APIs.

Slider sensor count and resolution are set in the SmartSense_EMCplus 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

Bit	7	6	5	4	3	2	1	0
Resolution Multiplier MSB								
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

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

Driven Shield Electrode

A driven shield electrode is an optional feature that reduces parasitic sensor capacitance. The benefits include improved sensor sensitivity and the prevention of false sensor triggering when water is detected 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 the 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 a negligible influence on the sensor capacitance signal. This prevents false activations.

Figure 7. Driven Shielding Electrode PCB Layout

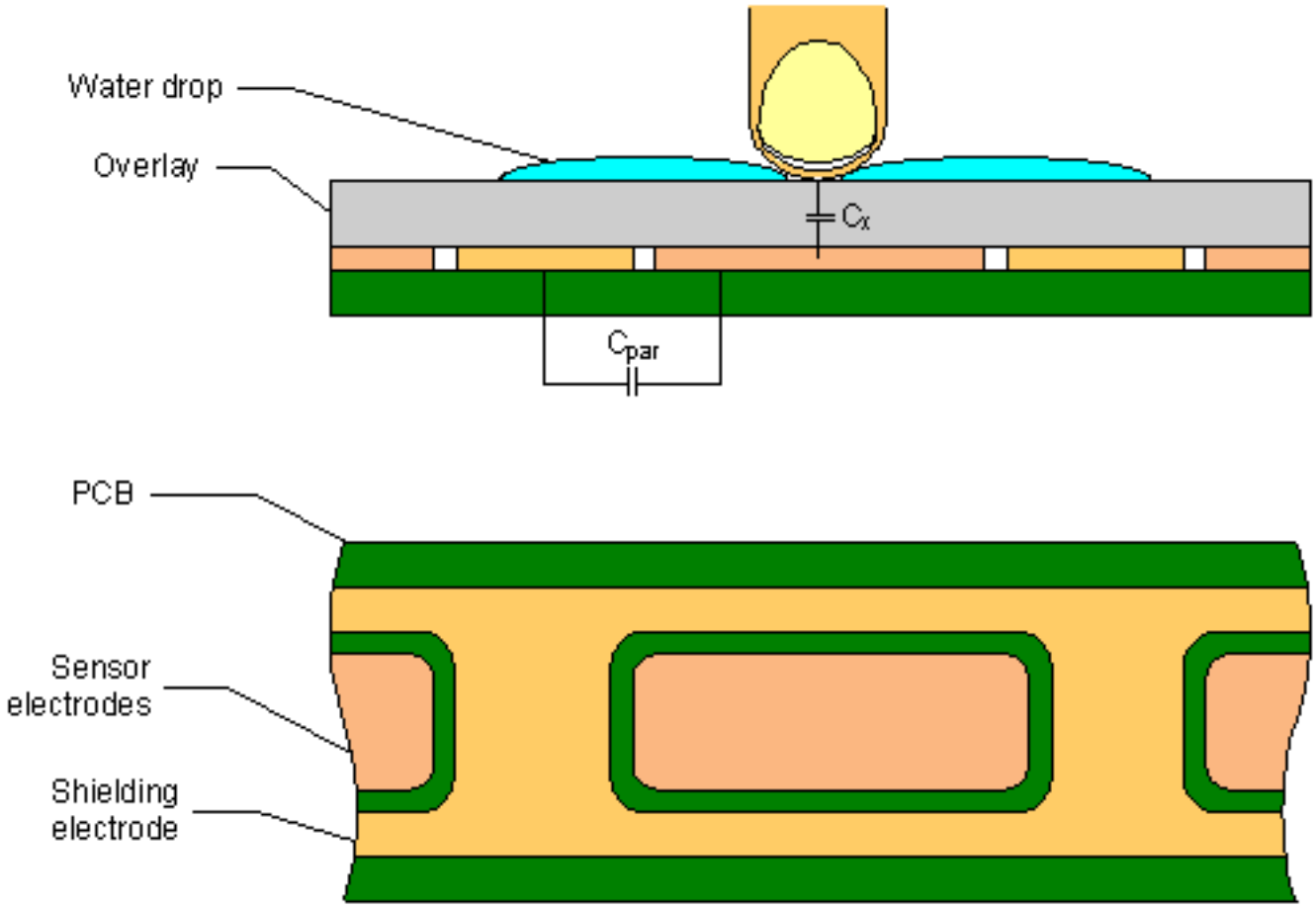
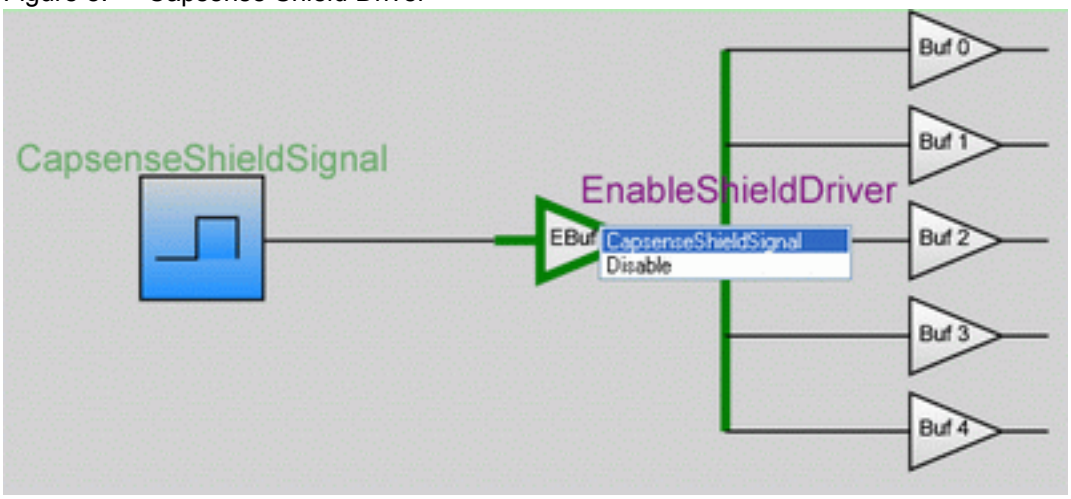


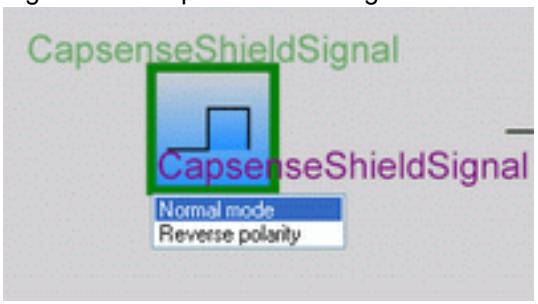
Figure 7 illustrates a driven shielding electrode for a button. As an alternative, the shielding electrode can be located on the opposite PCB layer, including the plane under the button. A hatch pattern is recommended in this case, with a fill ratio of about 30 percent to 40 percent. No additional ground plane is required. The shield electrode can be connected to any of the dedicated PSoC pins: P2[4],P2[2], P0[2], P0[0], or P1[2]. The drive mode for the selected pin must be set to High Z Analog. To reduce emitted EMI, connect a 560-Ω slew-limiting resistor between the PSoC device and the shielding electrode. The CapSense Shield Signal can be routed to P2[4], P2[2], P0[2], P0[0], or P1[2]pins through five ShieldBuffers.TheEnableShieldDriver block in the PSoC Designer Chip Editor enables this action.

Figure 8. Capsense Shield Driver



The Shield Driver can be configured to drive Shield Signal in two modes: normal and Reverse polarity. When you select reverse polarity, Shield driver clock is inverted.

Figure 9. CapsenseShieldSignal block



Power Supply Requirement

Table 3. SmartSense_EMCplus 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.

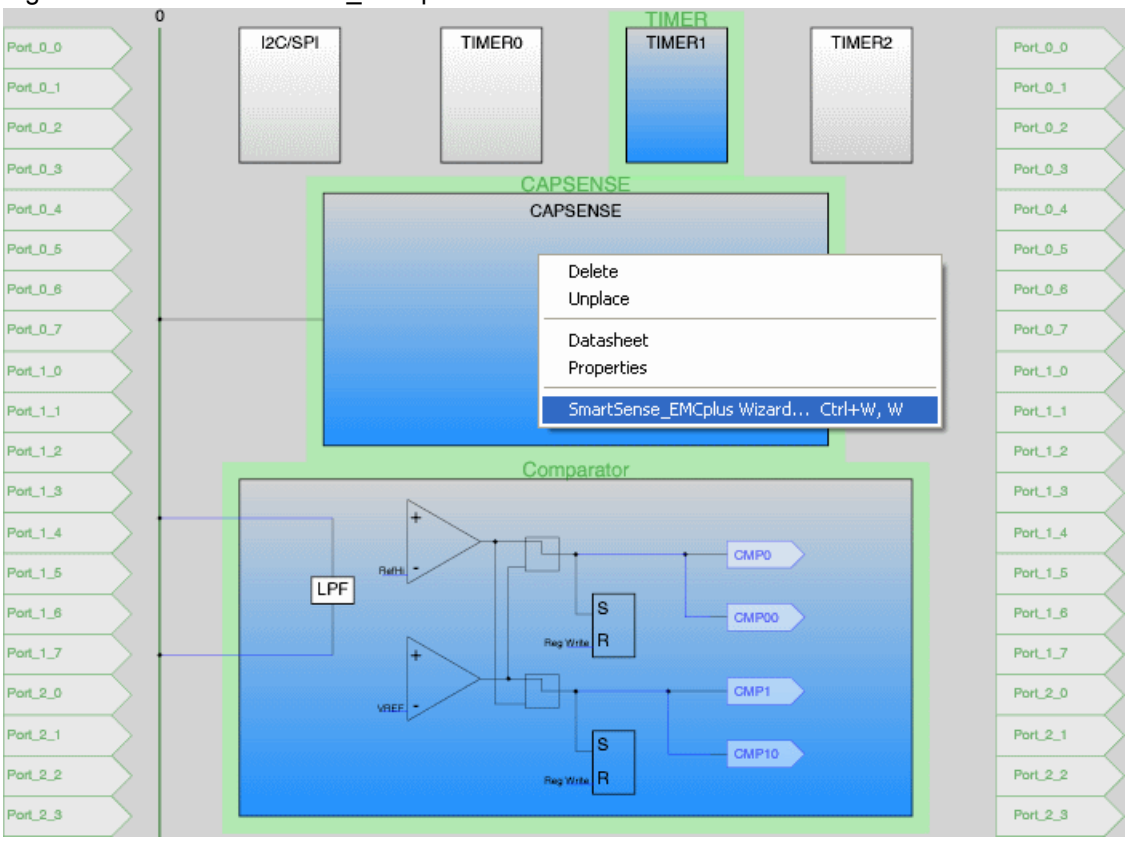
Placement

The CapSense, Comparator, CapsenseShieldSignal, and Timer1 blocks are assigned to SmartSense_EMCplus when the user module is instantiated. Alternate placements are not available. Place user modules that require dedicated pin resources, including the LCD and I2CHW, before starting the SmartSense_EMCplus Wizard. In addition, configure ShieldBuffers before using SmartSense_EMCplus Wizard. Doing so ensures that the dedicated pins are reserved and cannot be inadvertently assigned as sensors when they are mapped to I/O pins in the SmartSense_EMCplus Wizard. Avoid P1[0] and P1[1] when placing capacitive sensor connections. These pins are used to program the part and may have excessive routing capacitance, which damages sensor sensitivity.

SmartSense_EMCplus Wizard

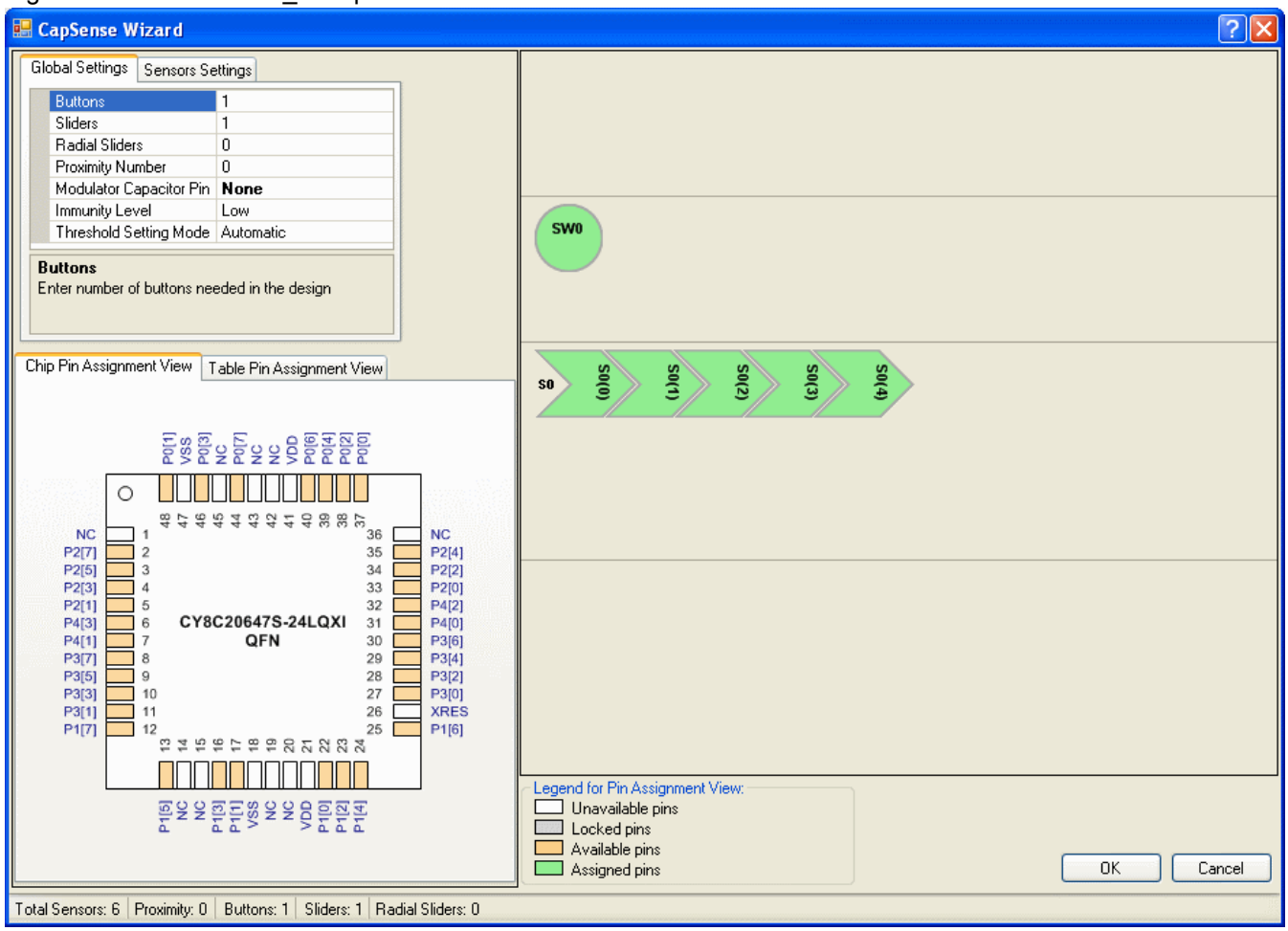
1. To access the SmartSense_EMCplus Wizard, right-click any block occupied by SmartSense_EMCplus in the Device Editor Interconnect View and select the SmartSense_EMCplus Wizard with a left click.

Figure 10. The SmartSense_EMCplus Wizard Access



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

Figure 11. SmartSense_EMCplus Wizard



Wizard Pin Legend

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

Gray – The pin is locked. There are two possible causes: The first 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.

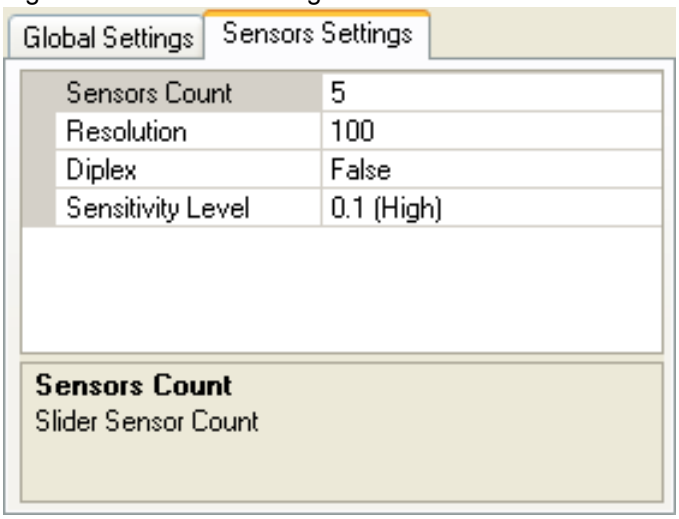
Orange – The pin is available for assignment.

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 sliders and radial sliders. To alter the settings, click one of your sliders to activate it. Type the number of sensor elements in each slider. The practical

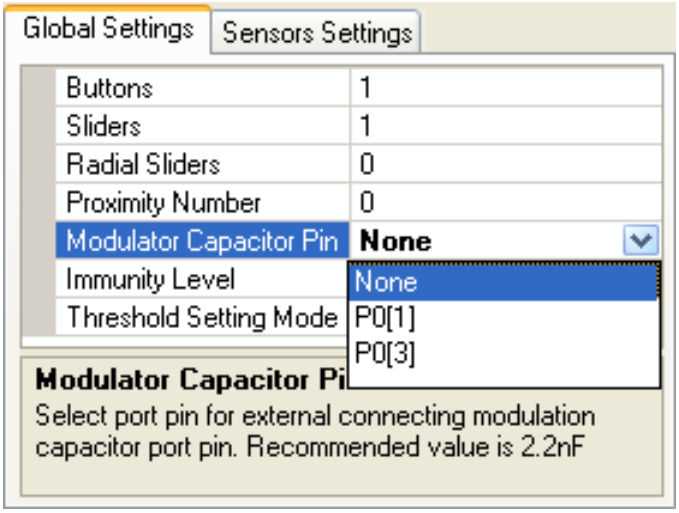
minimum number of sensors in a slider sensor is five, and the maximum is limited only by pin count. After entering the data, press the [Enter] key to update the display.

Figure 12. Sensor Settings Tab



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

Figure 13. Global Settings Tab

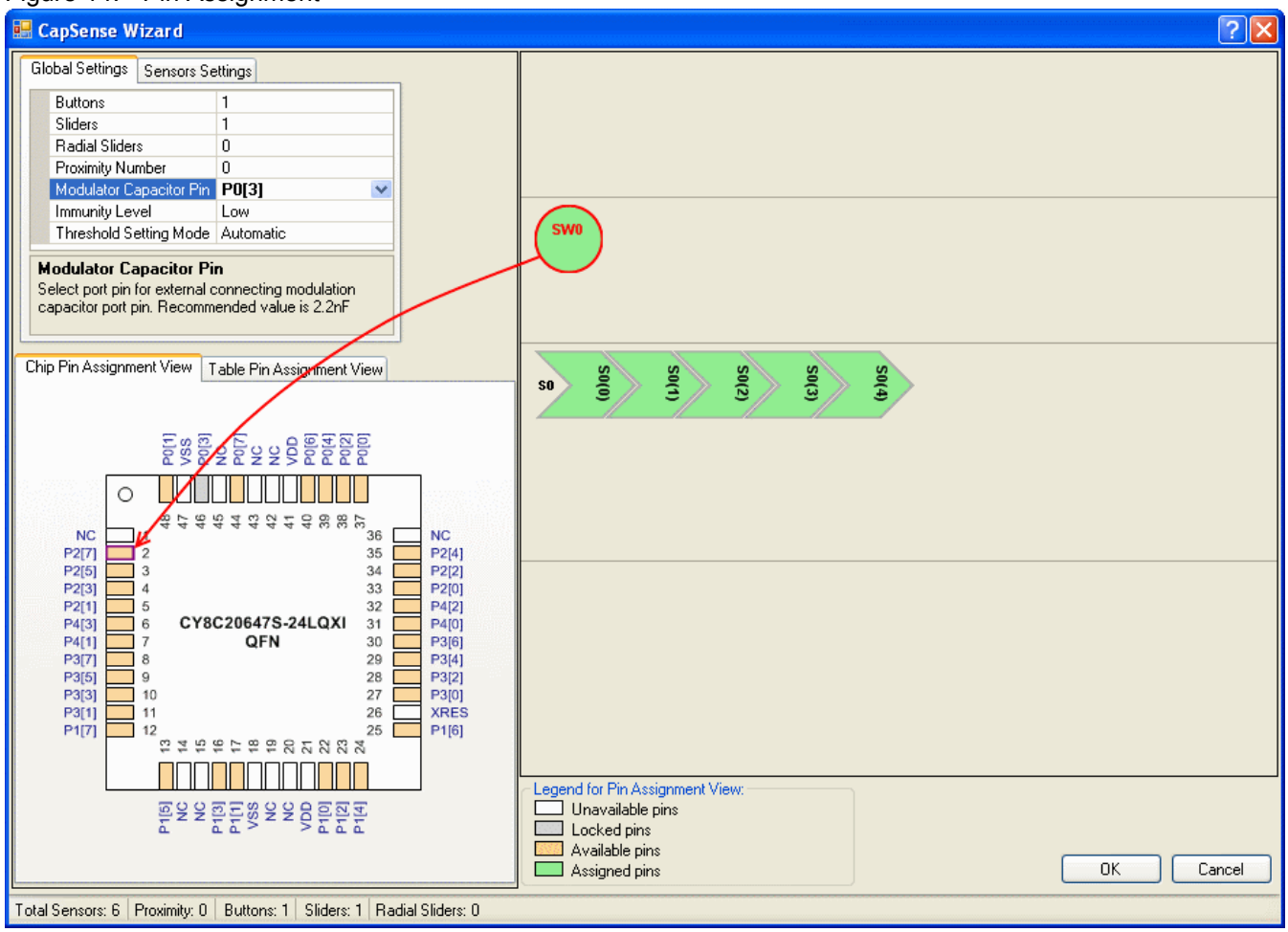


6. Type the output resolution. The minimum value is five. SmartSense_EMCplus attempts to interpolate the touch results to the specified resolution using the relative strength of adjacent segments. The software reports touch results on the slider between zero and the resolution - 1.

7. Select Diplex, if required. Doing so 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 each 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. If you select the I/O pin, it turns green, indicating it is no longer available. Change the sensor assignments by dragging the port pin back to the uncommitted table. Avoid selecting pins that are already committed to other user modules.

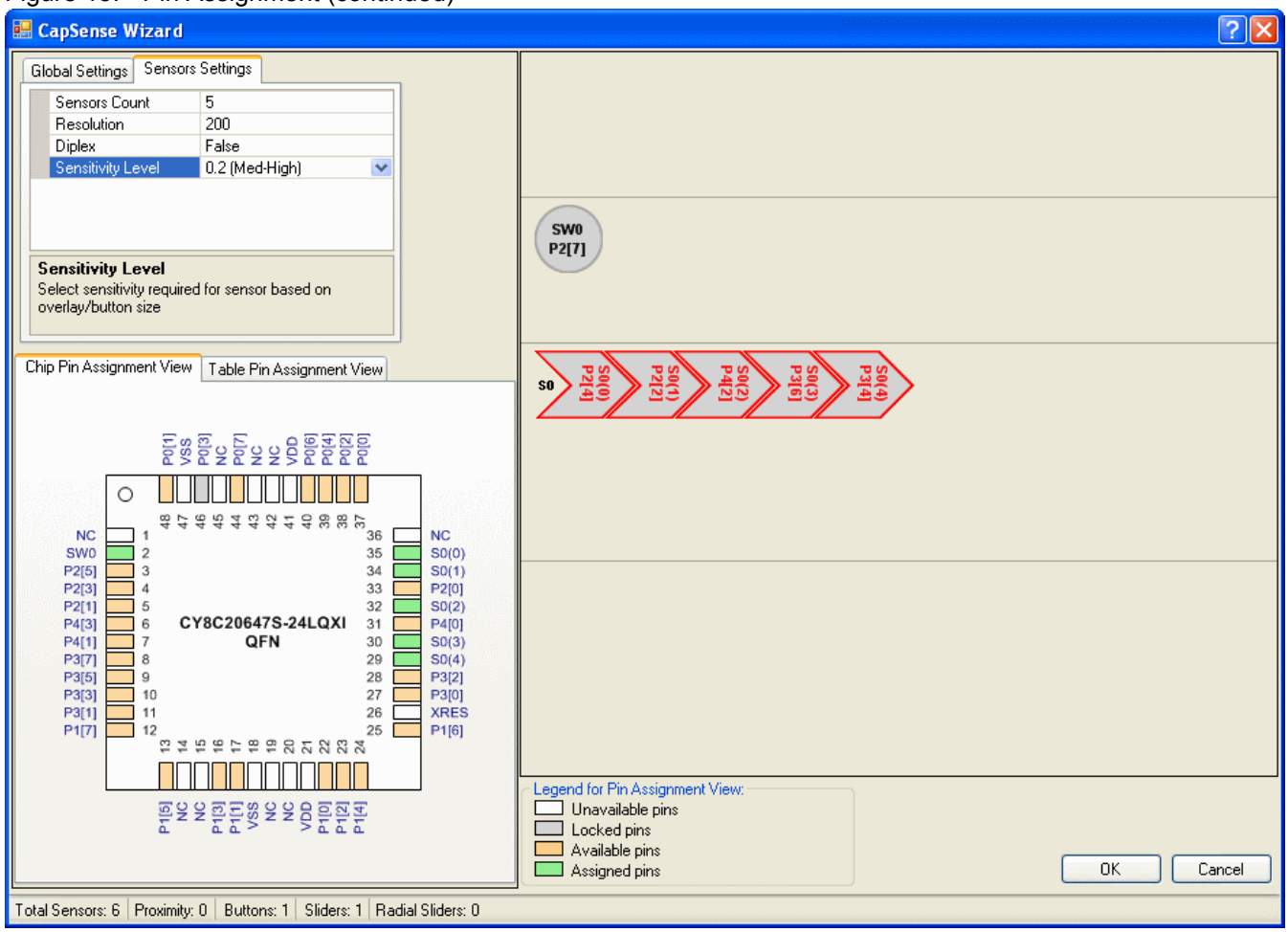
Figure 14. Pin Assignment



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

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

Figure 15. Pin Assignment (continued)



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_EMCplus_Table.asm and SmartSense_EMCplusHL.asm.

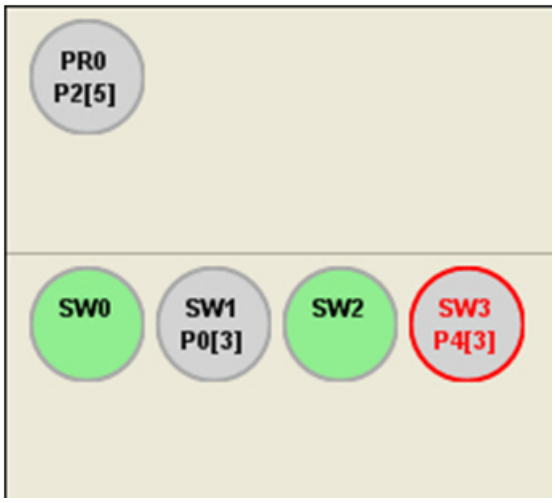
Sensor Representation Section

This section graphically represents all sensor types available in a project. This section is used to drag-and-drop sensors to the Chip or Table Pin Assignment View. The assigned sensors are marked grey. This section consists of three tiles:

- Buttons representation – The Button sensors are displayed in the wizard as shown in the following screenshot. Each button sensor element has its own title. All buttons support drag-and-drop capability for the Chip and Table Pin Assignment View. The assigned Port Pin number is displayed under the but-

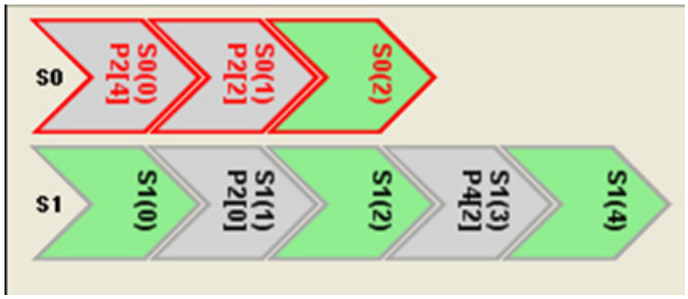
ton title if a button is already assigned. The button widget is set in a red frame if the button sensor is selected.

Figure 16. Buttons Representation



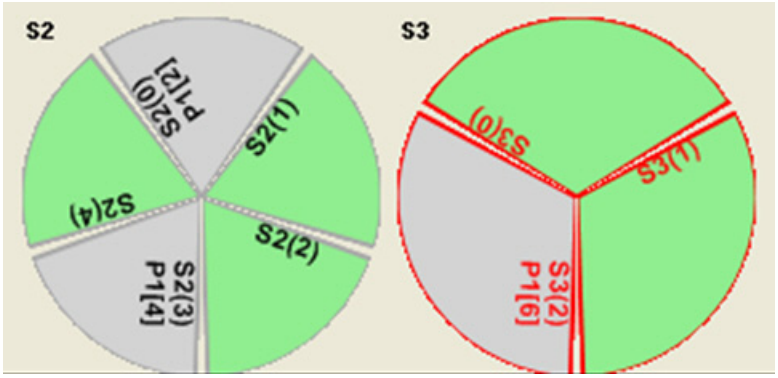
- Sliders representation – The Sliders are displayed as a sequence of sensor segments as shown in the following screenshot. The specific sensor name is displayed for each segment in a slider. All sensors in a slider support the drag-and-drop capability for the Chip and Table Pin Assignment View. The assigned Port Pin number is displayed for each segment if a slider segment is already assigned. All sensors in a slider are set in a red frame if the slider is selected.

Figure 17. Sliders Representation



- Radial Sliders representation – The Radial Sliders are displayed as a pie tile of segments as shown in the following screenshot. The sensor name is displayed for each segment in a radial slider. All sensors in a radial slider support the drag-and-drop capability for the Chip and Table Pin Assignment View. The assigned Port Pin number is displayed for each segment if a radial slider segment is already assigned. All sensors in a radial slider are set in a red frame if the radial slider is selected.

Figure 18. Radial Sliders Representation



Status Bar

The status bar displays common information about the design, such as the following:

- Total sensors – displays the overall number of sensors used in a design
- Proximity – displays the number of proximity sensors used in a design
- Buttons – displays the number of buttons used in a design
- Sliders – displays the number of linear sliders used in a design
- Radial sliders – displays the number of radial sliders used in a design



Wizard Buttons

The SmartSense_EMCplus wizard offers buttons with predefined functionality:

1. “OK” – This button checks if the wizard parameters are correct and all sensors are assigned. If yes, then the wizard saves the parameters and closes; if not, it shows an appropriate warning message, does not save parameters, and remains opened.
2. “Cancel” – This button closes the wizard without saving any parameters.
3. “Close” – This is a standard window close button located on the title bar, at the top right corner of the wizard form. If you click on the close button, the wizard closes without saving any parameters.
4. “Help” – This button calls the Help page giving a reference information on how to use the SmartSense_EMCplus user module wizard. It briefly describes the SmartSense_EMCplus User Module wizard features. The Help is available via a standard window help button, labeled with the question mark, and located on the title bar, at the top right corner of the wizard form.

Wizard Parameters

Wizard Parameters are described in the Global Settings and Sensor Settings tabs.

Global Settings Tab

Global Settings Tab consists of the following parameters: buttons, sliders, radial sliders, proximity, modulator capacitor pin, immunity level, and threshold setting mode.

Buttons

This is the number of physical button sensors.

Sliders

This is the number of physical linear slider sensors.

Radial Sliders

This is the number of physical radial slider sensors.

Proximity

This is the number of proximity sensors.

Modulator Capacitor Pin

This parameter sets the pin to connect to the external modulator capacitor (C_{mod}). Choose from the available pins P0[1] and P0[3].

Immunity level

This parameter defines the immunity level. The options are low, Medium, and High. Selecting High improves performance in a high-noise environment. However, this level also increases memory usage. As a result, the maximum number of supported sensors decreases. A High immunity level also increases the sensor scan time. Setting immunity to Medium consumes double the scan time, and setting the immunity to High consumes three times the scan time when compared with the Low mode. If you choose Medium, the sliders are disabled

Threshold Setting Mode

Choose between an automatic and manual threshold setting. You can choose a Threshold Setting Mode only if the Immunity Level is Low. In all other immunity modes, the threshold setting mode parameter is grayed out and the Manual status is maintained.

Sensor Settings Tab

Sensor Settings Tab consists of the following parameters: Finger Threshold, Sensitivity Level, Diplex, Resolution, and Sensors Count.

Finger Threshold

The Finger Threshold parameter is available only if you set Threshold Setting Mode parameter to Manual. This threshold is used to determine the state of each sensor. If any sensor is active, the bIsAnySensorActive() function returns a ‘1’. If all the sensors are off, the bIsAnySensorActive() function returns a ‘0’. Possible values range from 1 to 255.

This parameter should be set to 80% of sensor signal received from each sensor when it is touched. The signal received from each sensor is stored in the array variable SmartSense_EMCplus_baSnsSignal[].

Diplex

This option enables or disables the diplex for the slider. See the Diplexing section of this user module datasheet for a detailed description.

Resolution

This option sets the sensor resolution in the range from 5 to (number of pins used for sensors - 1) x 2^8 - 1 or (2 x number of pins used for sensors - 1) x 2^8 - 1 for diplexed sliders.

Sensors Count

This is the number of physical sensors in a slider or a radial slider.

Sensitivity Level

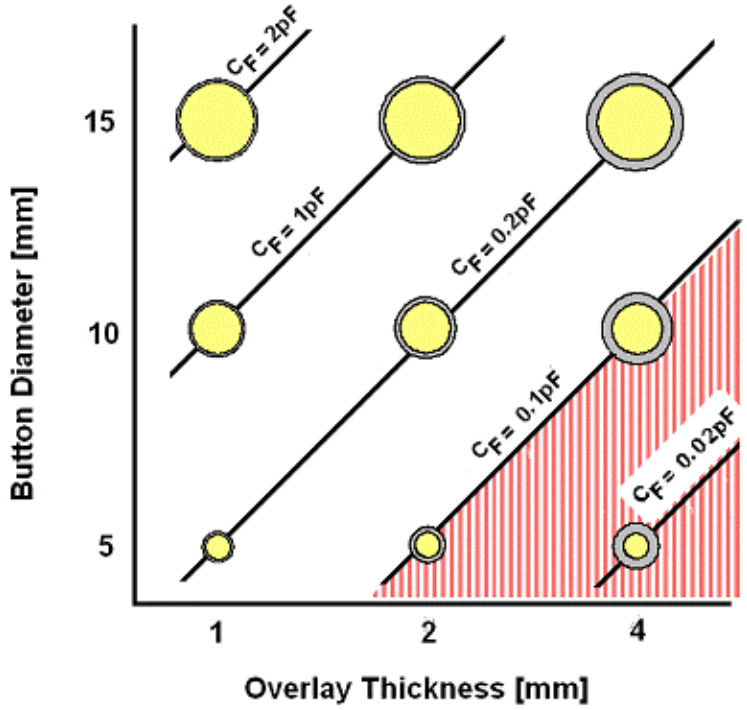
Sensitivity is used to increase or decrease the strength of a signal from a sensor. Setting a lower value for sensitivity (0.1 pF) leads to a stronger signal from the sensor. Designs with thicker overlays require a stronger signal from the sensor for proper implementation. The available options for sensitivity are High (0.1 pF), Medium High (0.2 pF), Medium Low (0.3 pF), and Low (0.4 pF).

To produce a stronger signal from a sensor (High sensitivity), the SmartSense_EMCplus User Module uses more time for sensor scanning. This means setting 0.1 pF (High) sensitivity for a sensor consumes more scan time compared to a sensor that has a sensitivity level set to 0.2 pF (Medium High).

One of the tuning best practices is to find the highest sensitivity value for a sensor that produces the required 5:1 signal-to-noise ratio (SNR). You can start tuning with the highest value for sensitivity (0.4 pF), and reduce the required value to meet the 5:1 SNR.

Figure 16 shows the relationship among the button size, overlay thickness (acrylic plastic), and sensor response (CF). This figure can be used as a guide for setting the sensor sensitivity. The sensor sensitivity must always be set at or below the sensor response shown in Figure 16. 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_EMCplus.

Figure 19. Relationship Between Button Size, Overlay Thickness, and Sensor Response in pF



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.

Tables Produced by the Wizard

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. Those tables are located in SmartSense_EMCplus_Table.asm and SmartSense_EMCplusHL.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_EMCplus_Sensor_Table:
_SmartSense_EMCplus_Sensor_Table:
      dw    0x0140  //  Port 1 Bit 6
      dw    0x0301  //  Port 3 Bit 0
      dw    0x0304  //  Port 3 Bit 2
      dw    0x0308  //  Port 3 Bit 3
      dw    0x0302  //  Port 3 Bit 1
      dw    0x0108  //  Port 1 Bit 3
```

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 button. 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_EMCplus Wizard.

```
SmartSense_EMCplus_Group_Table:
_SmartSense_EMCplus_Group_Table:
; Group Table:
;    Origin    Count    Diplex?    DivBtwSw(wholeMSB, wholeLSB, fractByte)
 db   0x0,     0x3,     0x00,      0x00,    0x00,    0x00 ; Buttons
 db   0x3,     0x8,     0x4,       0x0,     0x0,     0x44 ; Slider 1
```

Diplex Table

Diplex table scan order data is produced for a group that has a slider 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_EMCplus_Diplex_Table:
_SmartSense_EMCplus_Diplex_Table:
 db >DiplexTable_0, <DiplexTable_0
 db >DiplexTable_1, <DiplexTable_1
```

Finger Threshold Table

The Finger Threshold table defines the Normalized Finger Threshold for each sensor. A typical example for a one-button, five-sensor slider is shown here:

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

Sensitivity Level Table

The Sensitivity Level table defines the Sensor Sensitivity for each sensor. A typical example for a one-button, five-sensor slider is shown here:

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

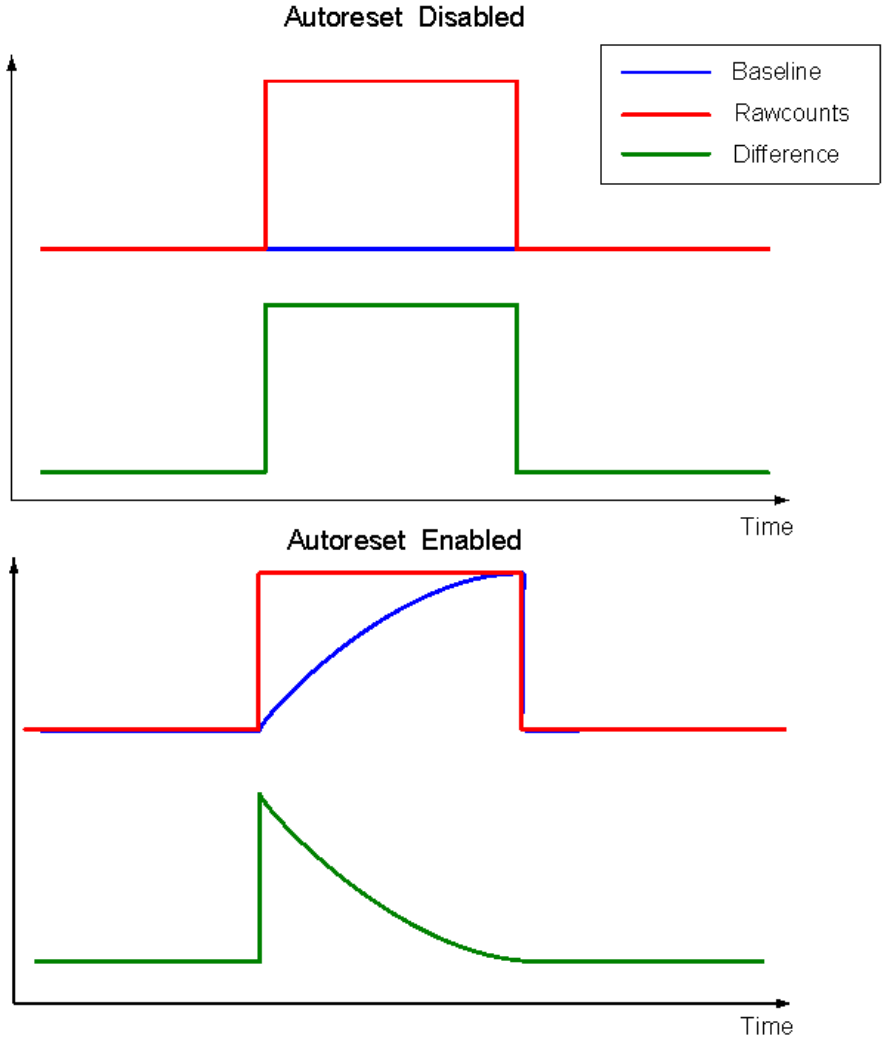
Parameters and Resources

After you have completed the configuration and I/O pin assignment in the SmartSense_EMCplus Wizard, it's time to set the user module parameters. To ensure that a user module parameter change takes effect, you must regenerate the project.

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, which means the baseline is updated only when the difference between the raw count and the baseline is below the noise threshold. Figure 17 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 between 5 and 10 seconds). However, this prevents sensors from getting stuck because of sudden rises in raw counts that are not caused by a touch. Such 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 for more information about this parameter.

Figure 20. Effect of the Sensor Autoreset Parameter on Baseline Update



Debounce

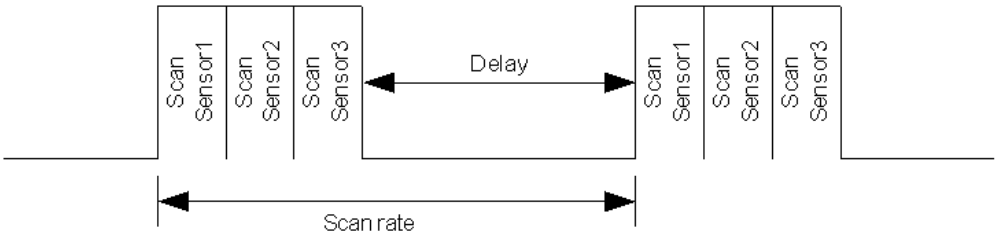
This parameter adds a debounce counter to the sensor active transition. For a sensor to transition from inactive to active, the difference count value must stay above finger threshold plus hysteresis for the number of samples specified by this parameter. The debounce counter is incremented by the bIsSensorActive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

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

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 21. 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 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_EMCplus_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_EMCplus 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. The calling function is responsible for preserving 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 its efficiency and has been in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers also must ensure their code observes the policy. Though some user module API function may leave A and X unchanged, there is no guarantee they may do so in the future.

For Large Memory Model devices, the caller is responsible for preserving 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_EMCplus, start the sampling, and stop the SmartSense_EMCplus. In all cases, the instance name of the module replaces the SmartSense_EMCplus 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.However,you can inspect these values for debugging purposes. For example, you can use a charting tool to display the contents of the arrays. There are several global arrays:

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

SmartSense_EMCplus_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_EMCplus_waSnsBaseline[] array is updated by these functions:

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

SmartSense_EMCplus_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_EMCplus_waSnsResult[] data is updated by these functions:

- SmartSense_EMCplus_ScanSensor()
- SmartSense_EMCplus_ScanAllSensors().

SmartSense_EMCplus_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. The SmartSense_EMCplus_waSnsDiff[] data is updated by these functions:

- SmartSense_EMCplus_UpdateAllBaselines()
- SmartSense_EMCplus_UpdateSensorBaseline()

SmartSense_EMCplus_baSnsOnMask[] – This is a byte array that holds the sensor ON or OFF state (for buttons or sliders). SmartSense_EMCplus_baSnsOnMask[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). SmartSense_EMCplus_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_EMCplus_baSnsOnMask[] data is updated by these functions:

- SmartSense_EMCplus_bIsSensorActive()
- SmartSense_EMCplus_bIsAnySensorActive()

SmartSense_EMCplus_baSnsSignal[] – This is a byte array that contains the normalized finger touch signal data of each sensor. The value of this array is used to monitor the signal of a finger touch and set the finger threshold when you select the Manual threshold setting mode. The array size is equal to the

sensor count. The SmartSense_EMCplus_baSnsSignal array is updated by the SmartSense_EMCplus_UpdateSensorSignal() function.

SmartSense_EMCplus_baDAC[] – This is a byte array that contains the IDAC1_CODE Register value for each sensor. The values from this array are copied to SmartSence_EMCplus_bIdacValue variable before scan.

SmartSense_EMCplus_baCompensationDAC[] – This is a byte array that contains the IDAC0_CODE Register value for each sensor. The values from this array are copied to SmartSense_EMCplus_bCompensationIdacValue variable before scan.

SmartSense_EMCplus_SnsErrorStatus[] – This is a byte array that reserves one bit for every CapSense sensor to indicate that the C_P is 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_EMCplus_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_EMCplus_Start(void)
```

Assembly:

```
lcall SmartSense_EMCplus_Start
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_Stop

Description:

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

C Prototype:

```
void  SmartSense_EMCplus_Stop(void)
```

Assembly:

```
lcall SmartSense_EMCplus_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_Resume

Description:

Resumes the user module operation after SmartSense_EMCplus_Stop call.

C Prototype:

```
void  SmartSense_EMCplus_Resume(void)
```

Assembly:

```
lcall SmartSense_EMCplus_Resume
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_ScanSensor

Description:

Scans the selected sensor. Each sensor is uniquely identified by its position in the Sensor Table. The SmartSense_EMCplus Wizard assigns this position or sensor number.

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_ScanAllSensors

Description:

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

C Prototype:

```
void  SmartSense_EMCplus_ScanAllSensors(void)
```

Assembly:

```
lcall SmartSense_EMCplus_ScanAllSensors
```

Parameter:

None

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_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, which uses the following algorithm:

1. Each time the SmartSense_EMCplus_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_EMCplus_waSnsDiff[] array.
2. If Sensors Autoreset is disabled, each time SmartSense_EMCplus_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 reaches 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 SmartSense_EMCplus_waSnsDiff[] array is reset to 0. As a result, this array does not contain elements with values greater than 0, but, rather, below the noise threshold.

C Prototype:

```
SmartSense_EMCplus_UpdateSensorBaseline(BYTE bSensorNum)
```

Assembly:

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

Parameter:

A => Sensor Number

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_UpdateAllBaselines

Description:

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

C Prototype:

```
void  SmartSense_EMCplus_UpdateAllBaselines(void)
```

Assembly:

```
lcall SmartSense_EMCplus_UpdateAllBaselines
```

Parameter:

None

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_bIsSensorActive

Description:

Checks the difference count array for the given sensor compared with 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_EMCplus_baSnsOnMask[] array.

C Prototype:

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

Assembly:

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

Parameters:

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_EMCplus_bIsAnySensorActive

Description:

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

C Prototype:

```
BYTE  SmartSense_EMCplus_bIsAnySensorActive(void)
```

Assembly:

```
lcall SmartSense_EMCplus_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_EMCplus_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_EMCplus Wizard. This function is available only if the slider is defined by the SmartSense_EMCplus Wizard.

C Prototype:

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

Assembly:

```
mov   A, bSnsGroup
lcall SmartSense_EMCplus_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 must be called only 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_EMCplus_blsSensorActive() routine to determine which slider segments are touched.

SmartSense_EMCplus_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_EMCplus Wizard.

C Prototype:

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

Assembly:

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

Parameters:

bSnsGroup A => Group Number

This parameter is the group number of the radial slider. You can get its number through the SmartSense_EMCplus 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. To avoid getting negative difference values and baseline update, this routine must be called only once after each scan. 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_EMCplus_blsSensorActive() routine to determine which slider segments are touched

SmartSense_EMCplus_wGetRadialInc

Description:

Returns actual finger shift, which is the difference between current and previous finger positions. This function works with SmartSense_EMCplus_wGetRadialPos().

C Prototype:

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

Assembly:

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

Parameters:

bSnsGroup A => Group Number

This parameter is the group number of radial slider. You can get its number in the SmartSense_EMCplus 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 counter-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_EMCplus_wGetRadialPos(), because it uses internal data stored in global variables by the latter.

SmartSense_EMCplus_InitializeSensorBaseline

Description:

Loads the SmartSense_EMCplus_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_EMCplus_InitializeSensorBaseline(BYTE bSensorNum)
```

Assembly:

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

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_InitializeBaselines

Description:

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

C Prototype:

```
void SmartSense_EMCplus_InitializeBaselines(void)
```

Assembly:

```
lcall SmartSense_EMCplus_InitializeBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_ClearSensors

Description:

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

C Prototype:

```
void  SmartSense_EMCplus_ClearSensors(void)
```

Assembly:

```
lcall SmartSense_EMCplus_ClearSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_wReadSensor

Description:

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

C Prototype:

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

Assembly:

```
mov   A, bSensor
lcall SmartSense_EMCplus_wReadSensor
```

Parameters:

A => Sensor Number

Return Value:

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

Side Effects:

**

SmartSense_EMCplus_wGetPortPin

Description:

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

C Prototype:

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

Assembly:

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

Parameters:

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

Return Value:

A => Sensor Bitmap, X => Port Number

Side Effects:

**

SmartSense_EMCplus_EnableSensor

Description:

Configures the selected sensor to measure during the next measurement cycle. The port and sensor can be selected using the SmartSense_EMCplus_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_EMCplus_EnableSensor(BYTE bMask, BYTE bPort)
```

Assembly:

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

Parameters:

A => Sensor Bitmap, X => Port Number

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_DisableSensor

Description:

Disables the sensor selected by the SmartSense_EMCplus_wGetPortPin() function. The drive mode is changed to Strong (001), which effectively grounds the sensor. The connection from the port pin to

the Analog Mux Bus is turned OFF. The function parameters are returned by SmartSense_EMCplus_wGetPortPin() function.

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Bitmap, X => Port Number

Return Value:

None

Side Effects:

**

SmartSense_EMCplus_UpdateSensorSignal

Description:

Updates the SmartSense_EMCplus_baSnsSignal[] array with normalized difference count.

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Number

Return Value:

A => Normalized difference count

Side Effects:

**

SmartSense_EMCplus_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 continually scans the sensors. The communication section can convey values to a PC charting tool.

```
//-------------------------------------------------------------------------
// Sample C code for the SmartSense_EMCplus module
// Scanning all sensors continuously
//-------------------------------------------------------------------------

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

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

// Detect if any sensor is pressed
        if(SmartSense_EMCplus_bIsAnySensorActive())
        {
            // Add user code here to proceed the sensor touching
        }

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

Example 2. This code starts the user module and continually scans the sensors; however, unlike Example 1, looping through the sensors is done in the user code. The communication section can convey values to a PC charting tool.

```
//-------------------------------------------------------------------------
// Sample C code for the SmartSense_EMCplus module
// Scanning all sensors continuously
```

```
//-------------------------------------------------------------------------

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

void main(void)
{
    BYTE bIndex;
    M8C_EnableGInt;

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

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

Example 3. This code is similar to Example 1, except this code changes the Sensor Sensitivity of Sensors 2 and 3 to 0.2 pF and 0.4 pF, respectively.

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

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

void main(void)
{
```

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

//detect if any sensor is pressed
        if(SmartSense_EMCplus_bIsAnySensorActive())
        {
            // Add user code here to proceed the sensor touching
        }

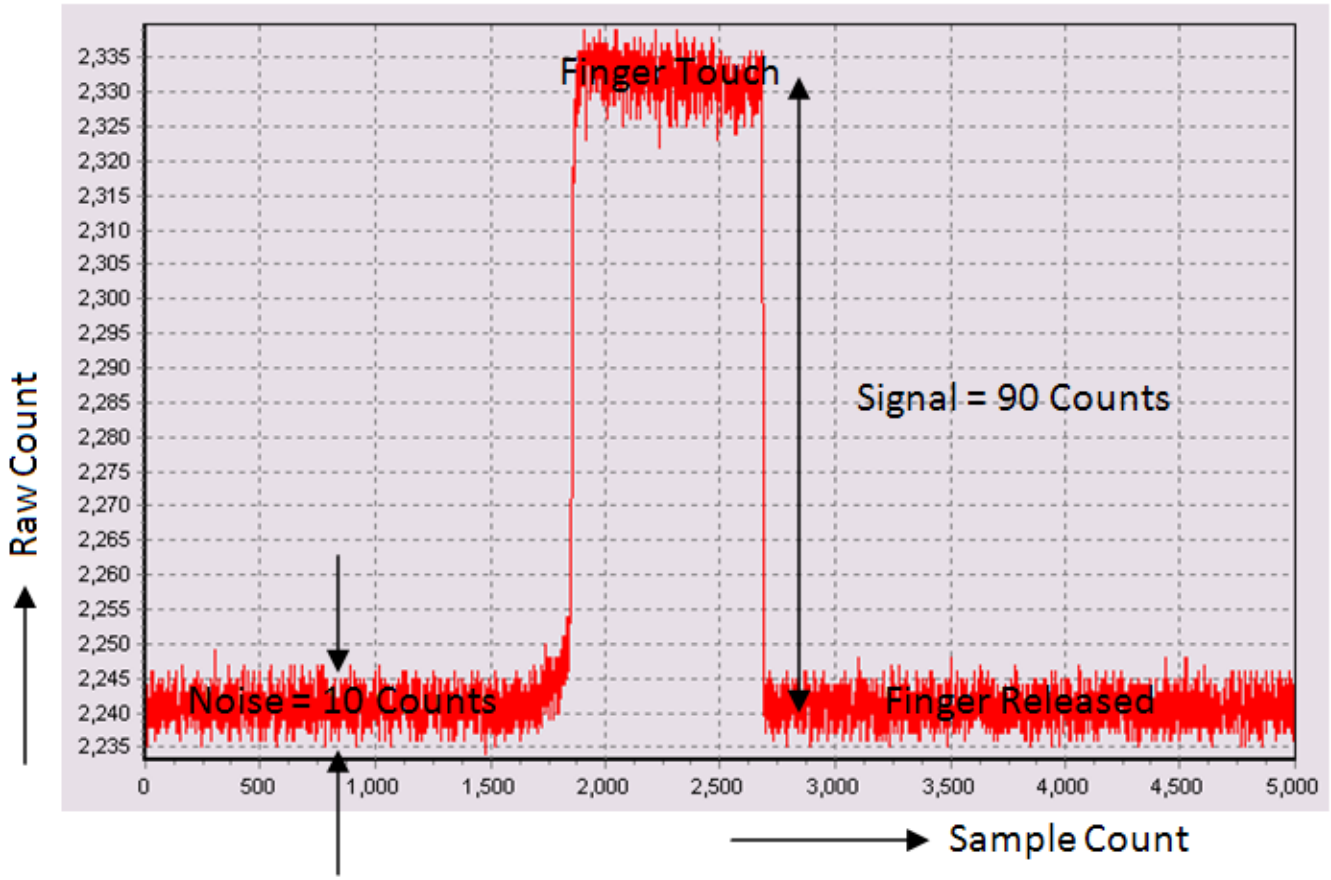
        // Now we are ready to send all status variables to chart program
        // communication here
//
// OUTPUT SmartSense_EMCplus_waSnsResult[x] <- Raw Counts
// OUTPUT SmartSense_EMCplus_waSnsDiff[x] <- Difference
// OUTPUT SmartSense_EMCplus_waSnsBaseline[x] <- Baseline
// OUTPUT SmartSense_EMCplus_baSnsOnMask[x]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

How to Set Optimal Sensitivity and Finger Threshold

SmartSense_EMCplus is an advanced electromagnetic compliance design of the CSD-based SmartSense User Module that does not require a tedious tuning process. However, two simple steps can ensure the robustness of the design when you use the SmartSense_EMCplus User Module:

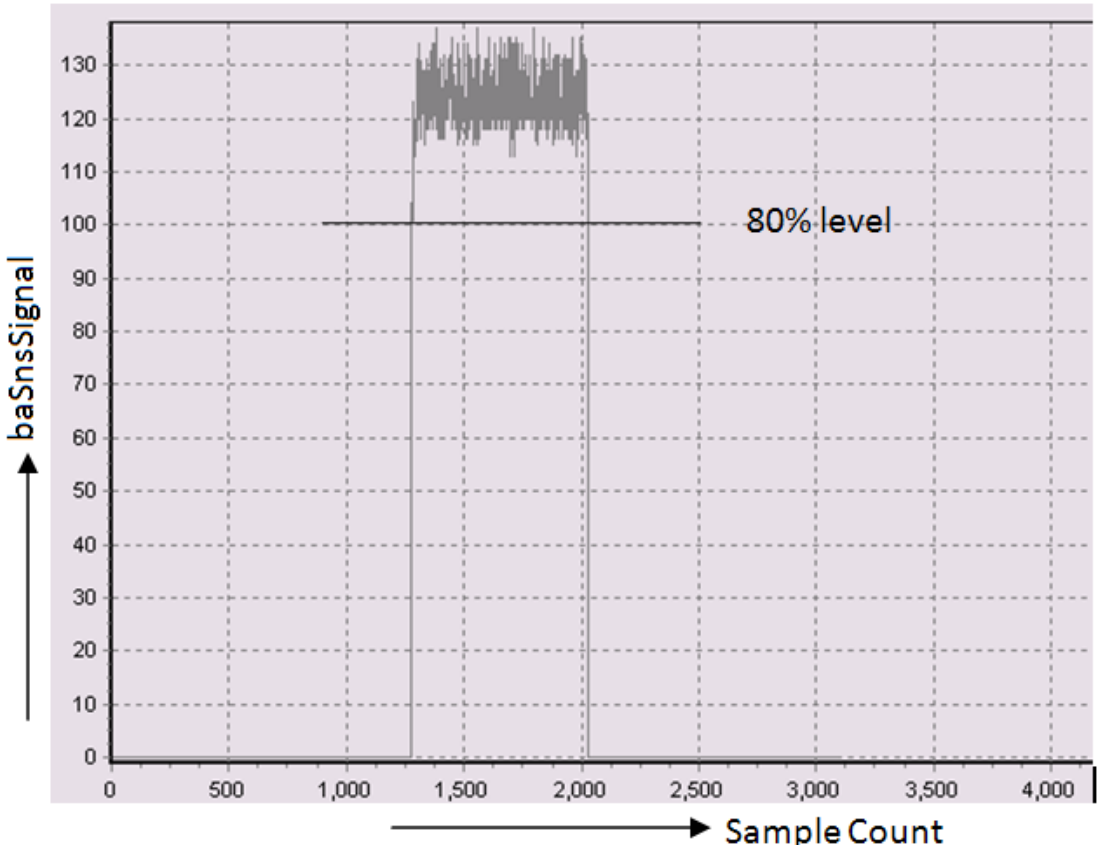
1. Set up a real-time monitoring tool for the CapSense User Module parameter to measure the SNR using the "Example code 2" given in this user module datasheet. The sensor raw count (SmartSense_EMCplus_waSnsResult), baseline (SmartSense_EMCplus_waSnsBaseline), difference counts (SmartSense_EMCplus_waSnsDiff), normalized signal (SmartSense_EMCplus_baSnsSignal), and sensor finger threshold (baBtnFThreshold) must be observed during the tuning process. Do not use the LCD or any other numerical display to monitor the data, because these are slow and do not allow visualizing the data dynamics. Recommended data monitoring tools are multi-chart or I2C USB Bridge Control panel.
2. Set the sensitivity level to 0.4 pF (Low), monitor the sensor raw count (SmartSense_EMCplus_waSnsResult), and calculate the SNR. Figure 18 shows a typical raw count graph with a finger touch. According to CapSense best practices, the SNR for a robust design must be greater than 5:1. If the measured SNR is less than 5:1, reduce the sensitivity level value to the next possible step until you achieve 5:1 or higher.

3. Raw Count Graph for a Typical Sensor with Finger Touch



4. If automatic finger threshold is used in the design, the previous step (step 2) completes the tuning process. If the design uses flexible finger threshold, then set the finger threshold as well to complete the tuning process. To set the finger threshold, monitor the sensor signal (SmartSense_EMCplus_baSnsSignal) and set the finger threshold value to 80 percent of the sensor signal value when the sensor is touched. Figure 19 shows a typical sensor signal and the finger threshold value.

Figure 22. Sensor Signal for a Typical Sensor with a Finger Touch



SmartSense_EMCplus User Module-Specific Guidelines

All guidelines applicable to the SmartSense User Module also apply to the SmartSense_EMCplus User Module. For general guidelines on CapSense design and SmartSense_EMCplus based design, refer to the CapSense Getting Started Guide.

This section discusses some important aspects of the SmartSense_EMCplus User Module:

Sensor Scan Time, Response Time, and Memory Usage

When a sensor uses the SmartSense_EMCplus User Module, the scan and response times of the sensor as well as the RAM memory usage depend on the immunity mode you select in the user module:

- With immunity mode Medium, the sensor scan time is two times higher than the sensor with immunity mode Low. With immunity mode High, the scan time of the sensor is three times higher than the scan time of the sensor with immunity mode Low.
- Increase in scan time proportionally increases the response time of a sensor. With immunity mode Medium, the response time is two times higher than that of a sensor with immunity mode Low. Similarly, the response time of a sensor with immunity mode 'High' is three times higher than that of a sensor with immunity mode Low.
- To implement a robust electromagnetic compliant algorithm, the SmartSense_EMCplus User Module uses RAM. As a result, the highest immunity mode (High) needs about three times the amount of RAM compared with the immunity mode Low. Medium uses only about double the RAM of that of immunity mode Low.

IMO Tolerance and Time-Critical Task

The tolerance of the internal main oscillator (IMO) for SmartSense_EMCplus enabled parts is +5 percent and -20 percent.

- When implementing a time-critical algorithm and logic, the tolerance of the IMO must be considered to ensure that the firmware logic or algorithm does not break.
- If your project uses interrupts, the tolerance of the IMO must be considered when analyzing the interrupt latency and ISR execution time, among other factors.
- For every timing analysis that depends on the IMO (for example, the timer clocked by IMO, delay created using loop in firmware, API execution time) you must consider the tolerance of the IMO to ensure a robust application firmware.

I^2C Operating Speed

The I2C interface operation frequency is limited to a maximum 80 percent of the actual operating frequency of the user module in the SmartSense_EMCplus enabled devices. The reason for that is the tolerance of the IMO.

- Therefore, when clock speed of 400 kHz is selected in I2C User Module, the I^2C interface can be operated to a maximum of 320 kHz. Similarly, the operating frequency is limited to a maximum of 80 kHz and 40 kHz when the 100-kHz and 50-kHz clock mode are selected in the I2C User Module.
- When using the I^2C slave interface, the master clock must operate within the reduced specification mentioned earlier. Failure to do so leads to data corruption, I^2C bus conjunction, or inconsistent behavior from the I2C User Module.
- Using the I^2C master module affects only the throughput of the interface.

Configuration Registers

Table 4. Block CapSense, Register: CS_CR0

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

Table 5. 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 6. Block CapSense, Register: CS_CR2

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

Table 7. Block CapSense, Register: CS_CR3

Mode/Bit	7	6	5	4	3	2	1	0
Value	SHIELD_IO_EN[4]	0	0	0	SHILED_IO_EN[3:0]			

Table 8. Block CapSense, Register: CS_CNTH

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

Table 9. Block CapSense, Register: CS_CNTL

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

Table 10. Block CapSense, Register: PRS_CR

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

Table 11. 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 12. Block Timer, Register: PT1_DATA0

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

Table 13. Block Timer, Register: PT1_DATA1

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

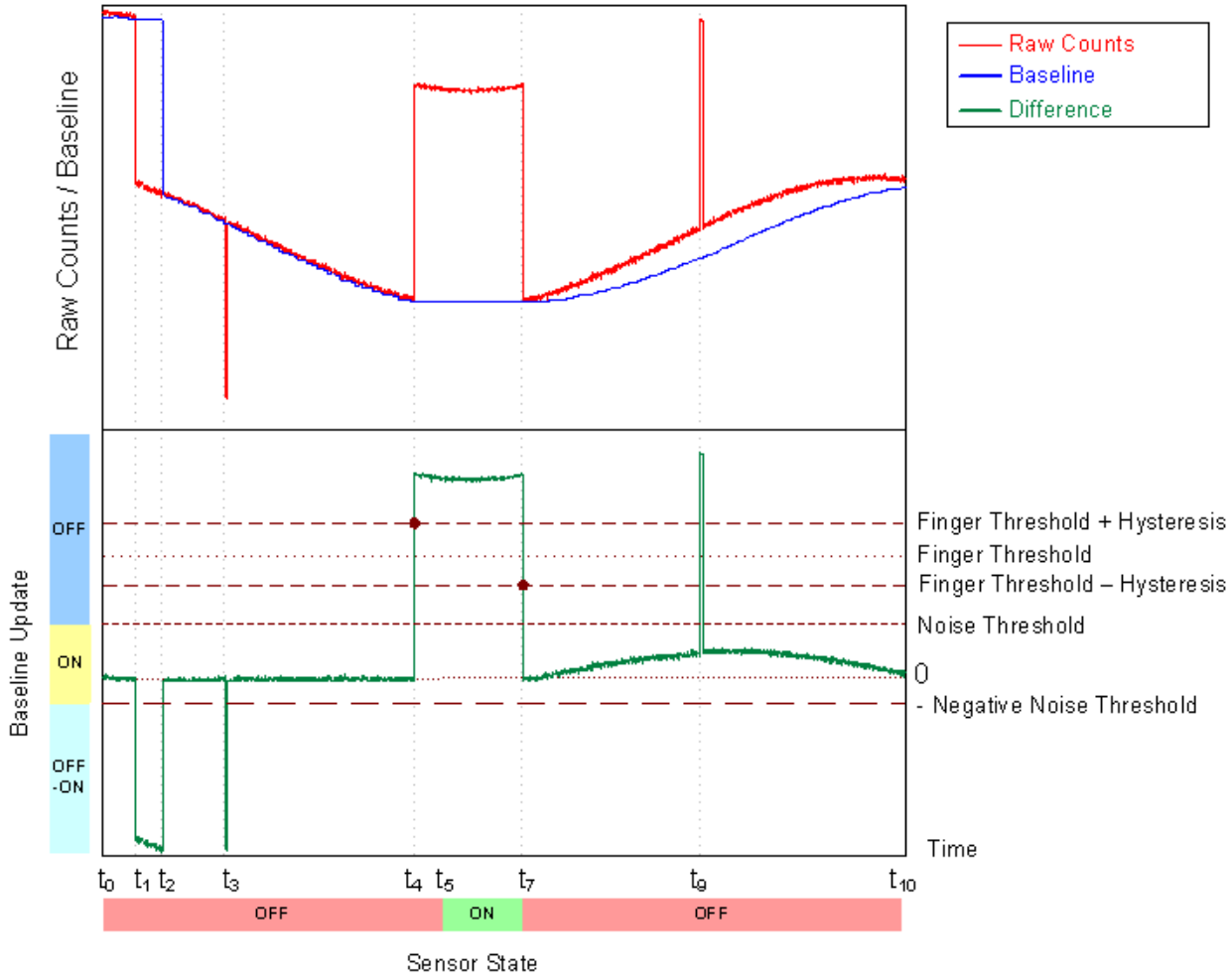
Appendix

The following sections contain information that goes beyond what is usually included in user module datasheets. Cypress engineers developed this information to help you design CapSense applications. In the future, some of this information may be moved into application notes.

Interaction of SmartSense_EMCplus Parameters

The following figures illustrate the baseline update and decision logic operation. They can help you better understand how to set user module parameters for optimum performance. Figure 20 illustrates system operation when the Sensors Autoreset parameter is set to Disabled. Figure 21 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 variations in temperature or humidity, and the rapid changes can be triggered by a sensor touch, an ESD event, or the influence of a strong RF field.

Figure 23. 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.

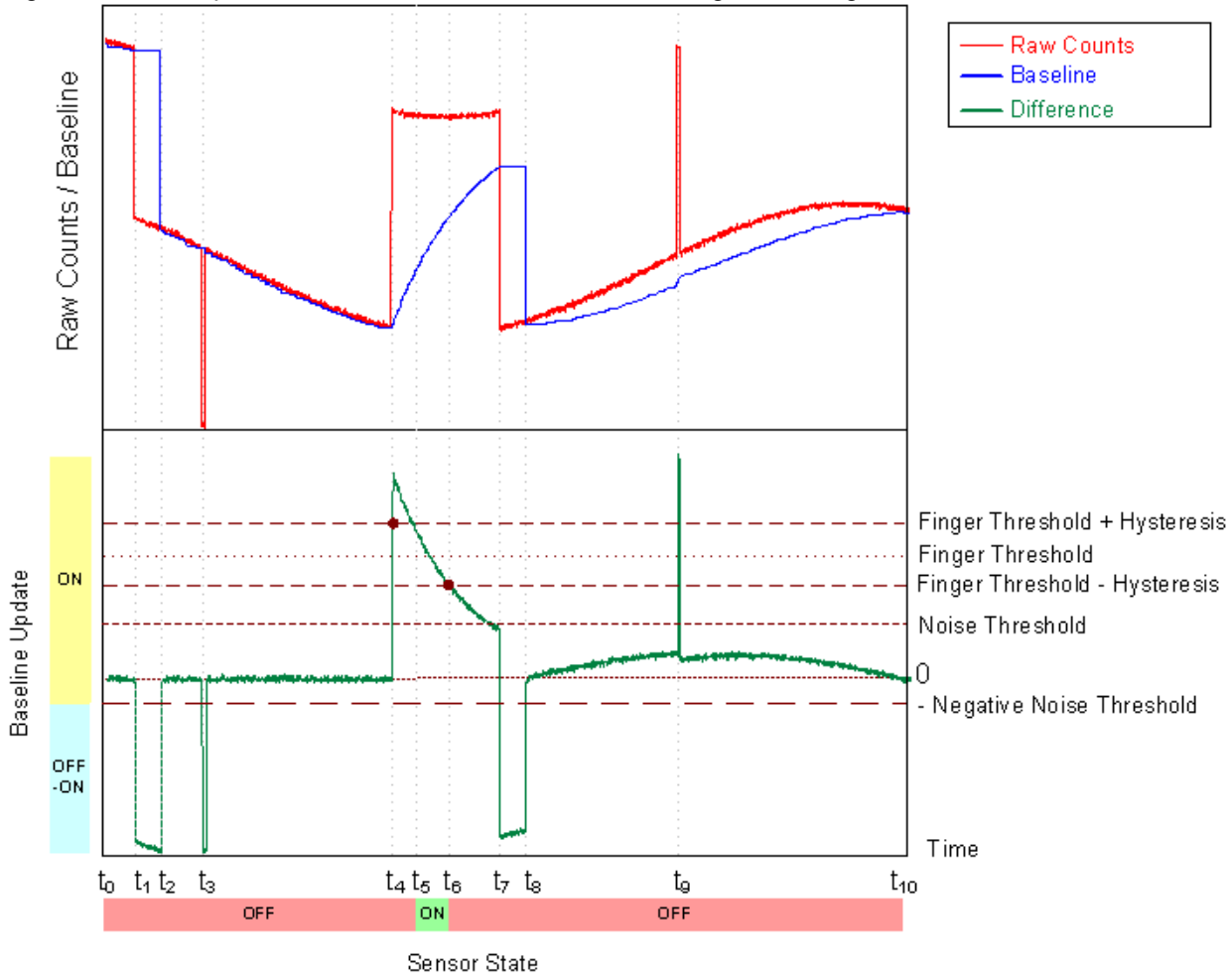
At t1, the raw count drops sharply, and the negative difference exceeds the NegativeNoiseThreshold. This situation can happen if the device is powered on when a finger is on the sensor and is later removed. At this time, the baseline update mechanism is frozen and an internal timeout counter is activated. 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 t5. The sensor state reverts back to the OFF state immediately when the difference signal drops below the FingerThreshold – Hysteresis level at t7. 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. You can control the baseline update speed using the BaselineUpdate Threshold parameter. Lower parameter values give faster baseline update speeds.

Figure 24. Example of Raw Counts, Baseline, Difference Signals Change With SensorsAutoreset Set to Enabled



The system operation in Figure 21 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. Updated Sensitivity Level property default value to 0.4. 2. CSD mode disabled and PRS clock source bit reset in SmartSense_EMCplus_Stop API. 3. Added CY8C200x5 device support.
1.20	MYKZ	1. Fixed problem with saving information for sliders. 2. Updated baseline algorithm to check for negative difference counts. 3. Added GetSnsParasiticCapacitance to User Module API. 4. Added build error message to prevent User Module usage when Wizard was not called. 5. Implemented large memory model in InitializeSensorBaseline() function. 6. Initialized default value of GlobalSettingsProxy properties in User Module wizard.

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 © 2012-2013 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.