CapSense® Sigma-Delta Plus Datasheet CSDPLUSV 1.20

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

Resources	PSoC® Blocks				API Memory (Bytes)		Pins per Sensor
	CapSense®	I²C/SPI	Timer	Comparator	Flash	RAM	
CY8C20xx7/S, CY8C20055							
User Module	1	–	1	1	1203	42	1
Slider APIs	–	–	–	–	1863	94	0
Each Sensor	–	–	–	–	5	13	1

For one or more fully configured, functional example projects that use this user module, click here.

Features and Overview

- Implements CapSense capacitive sensing in the CY8C20xx7/S family using the latest sigma-delta plus (CSDPLUS) sensing architecture
- Configurable system parameters allow tuning to optimize performance in a range of applications such as proximity sensors, buttons, or sliders
- Supports as many as 31 capacitive sensors and six sliders
- Capable of detecting touches as low as 0.1 pF; therefore, detecting a finger is possible through up to 15 mm of glass or 5 mm of plastic
- CSDPLUS compensates for higher parasitic capacitances
- CSDPLUS improves sensitivity, increases proximity detection range
- CSDPLUS offers improved noise performance
- New Driven Shield electrode is available on multiple GPIOs (package dependent, up to a maximum of five)
- Driven Shield electrode improves performance with high sensor parasitic capacitance
- Driven Shield electrode ensures robust performance in the presence of water and improves proximity detection range
- Improved immunity to GPIO current transient and VDD fluctuation
- Robust entry and exit from I2C Sleep with wake-up from HW address match
- Supports capacitive sensors configured as independent buttons or as dependent arrays to form sliders, or both
- Effective number of slider elements can be double the number of dedicated I/O pins using the diplexing technique
- Supports slider resolution greater than physical pitch by using interpolation
- Guided sensor and pin assignments using the CSDPLUS Wizard
- The CY8C20055 family does not support sliders

The CSDPLUS User Module is based on the differential capacitive sensing method. This user module uses the analog mux bus to connect a capacitive sensing analog circuitry to any PSoC pin. The CSDPLUS

User Module connects the active sensor to the analog mux bus allowing the CapSense circuitry to measure its capacitance and translate that capacitance into a digital code. Firmware serially scans the sensors by sequentially setting corresponding bits in the MUX_CRx registers.

Quick Start

1. Select and place user modules that require dedicated pins (for example, I2C and LCD), if they are used. Assign ports and pins as required.
2. Select and place the CSDPLUS User Module.
3. Right-click the CSDPLUS User Module in Workspace Explorer to access the CSDPLUS Wizard (the wizard is explained later in this user module datasheet).
4. Set the required number of sensors, sliders, or rotary sliders.
5. For sliders, enter the parameters specific to sliders.
6. Assign each of the sensors to an unused pin.
7. Enter the pin to which the external modulation capacitor will be connected in the CSDPLUS Wizard.
8. Enter the pin that will shield the sensor, if appropriate, in the User Module Parameters list (see explanation in the following 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/or a touchpad.
11. Program the PSoC on the target board with the hex generated by PSoC Designer™.

Introduction

CapSense is a human interface technology that detects the capacitance of the human body using sensors consisting of a conductive surface, usually a pad etched on the PCB. Because CapSense detects body capacitance, it can sense even 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 such as push-buttons and potentiometers. Major benefits of CapSense include:

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

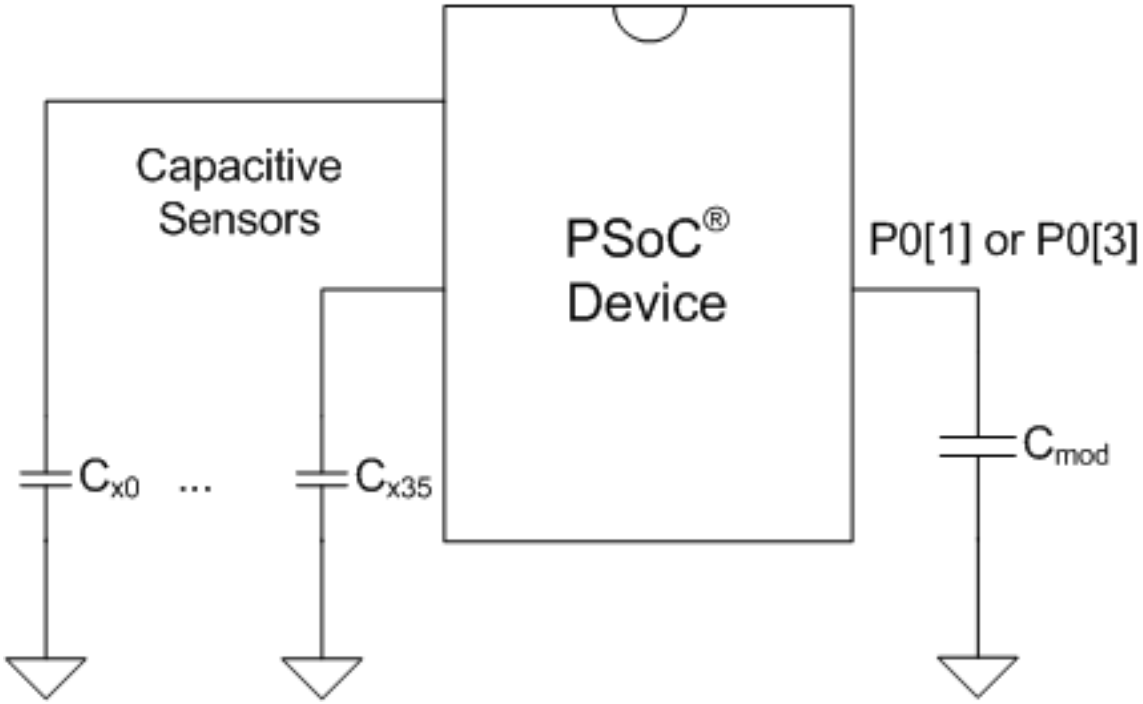
CSDPLUS uses a circuit that converts sigma-delta capacitance to digital code. The circuit's key attributes include low EMC emission and superior immunity against EMI.

The CSDPLUS User Module consists of a PCB level, an IC level, and software components, which are explained in the following sections.

PCB Level

Figure 1 shows a schematic of CSDPLUS. 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 Cypress guide, Getting Started with CapSense, for complete information on PCB-level CapSense implementation.

Figure 1. CSDPLUS Schematic

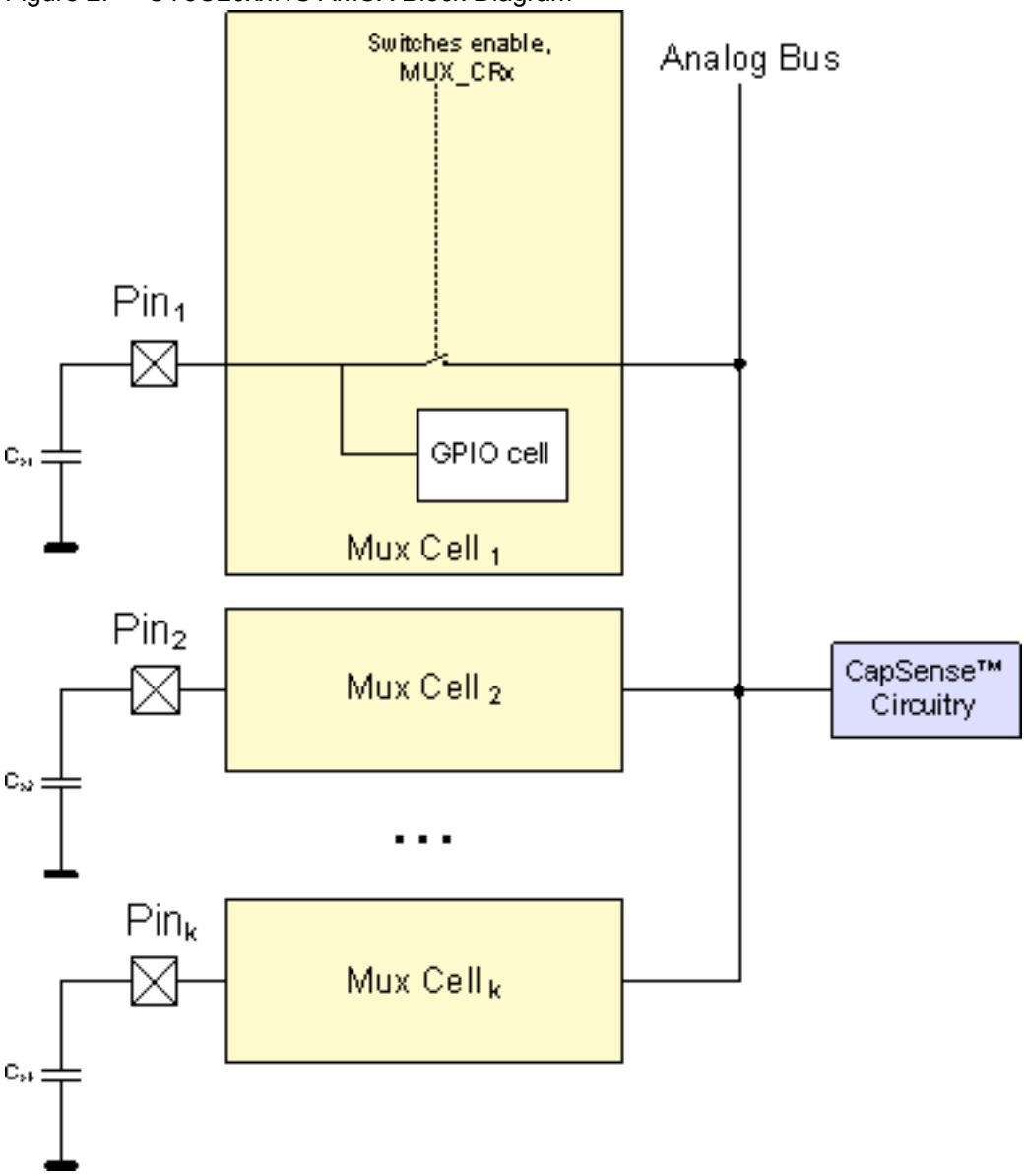


IC Level

CY8C20xx7/S devices have an analog mux bus that allows connecting capacitive sensing analog circuitry to any PSoC pin. The CSDPLUS User Module connects the active sensors to the analog mux bus allowing the CapSense circuitry to measure its capacitance and translate that capacitance into a digital code.

Firmware serially scans the sensors by sequentially setting corresponding bits in the MUX_CRx registers as shown in Figure 2.

Figure 2. CY8C20xx7/S AMUX Block Diagram



Software

The CSDPLUS software component has the following attributes:

- Tunable system configuration parameters allow tuning to optimize performance in a range of applications.
- 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 supplied to interpolate a position with greater resolution than the physical pitch of the sensors.
- High-level software functions accommodate slider diplexing so that a single I/O pin can be routed to two physical sensors. Therefore, the number of I/Os consumed for a given number of slider elements is halved.

Recommended Reading

The following documents are recommended reading before you implement a CapSense design using the CSDPLUS User Module:

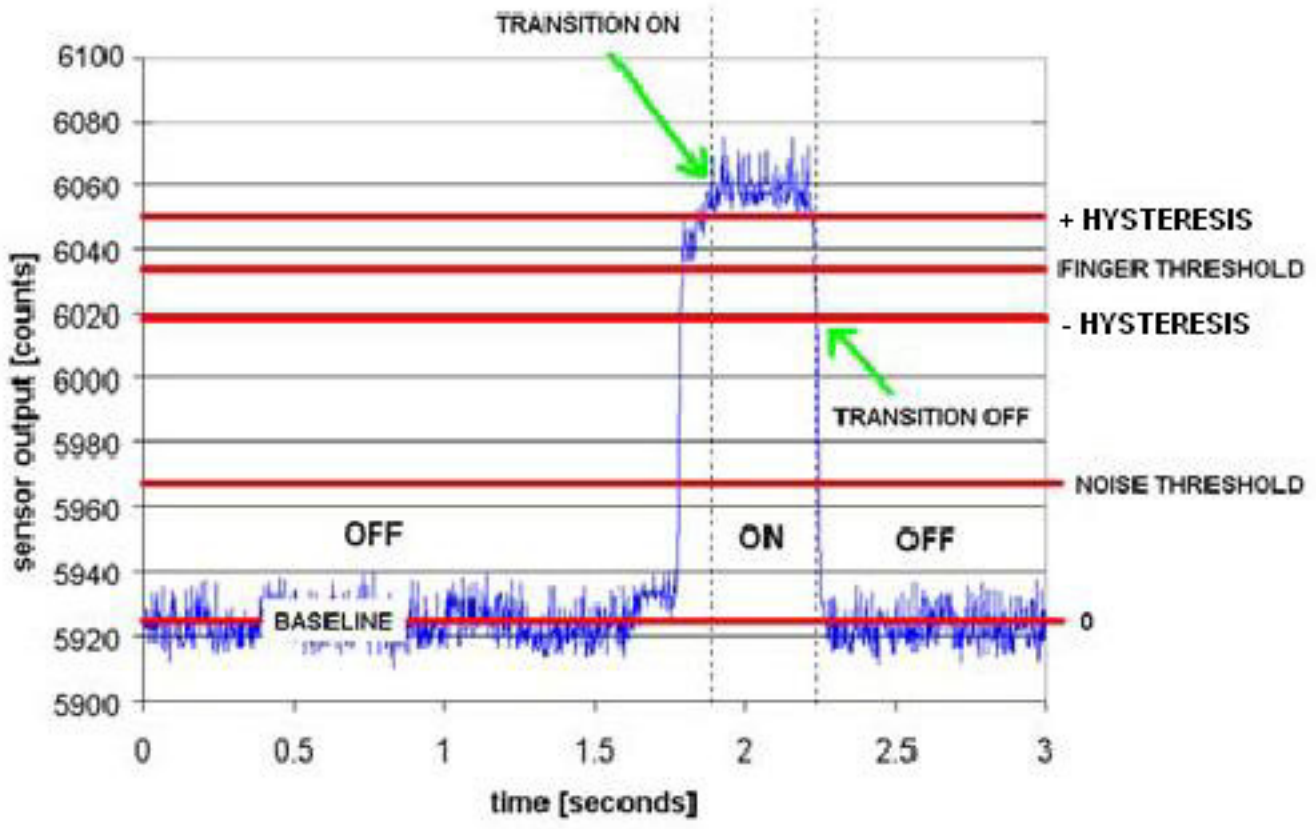
- Getting Started with CapSense
- CY8C20xx6A/H/AS CapSense® Design Guide
- CY8C21x34/B CapSense® Design Guide
- CY8C20x34 CapSense® Design Guide
- CY8CMBR2044 CapSense® Design Guide

Capacitance Sensing Implementation

Buttons

Analogous to mechanical push-buttons, CapSense buttons, are used for discreet controls, such as on/off switches, function keys, and menu keys. API functions monitor the capacitance signals (raw counts) from each sensor and compare them to tunable threshold parameters. When a sensor is touched, its capacitance signal increases; if the increase is sufficient, as determined by the CSD decision logic, that sensor is activated. Figure 3 shows a typical signal (blue line) from a sensor as it is activated. The thresholds (red lines) can be tuned to provide the desired behavior.

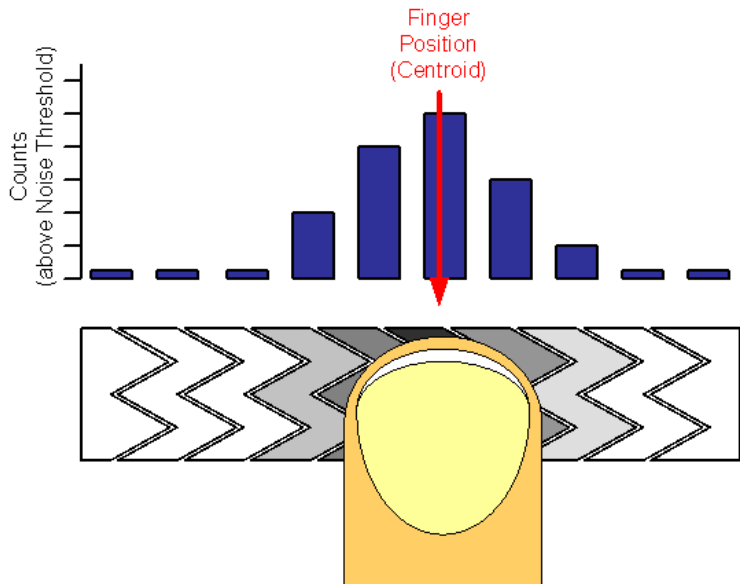
Figure 3. Capacitance Signal from a Sensor as it is Being Activated



Sliders

Analogous to mechanical potentiometers, CapSense sliders are used for controls that require a continuum of levels. Those include 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 capacitance signal, as shown in Figure 4. Find the exact position of the touch 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

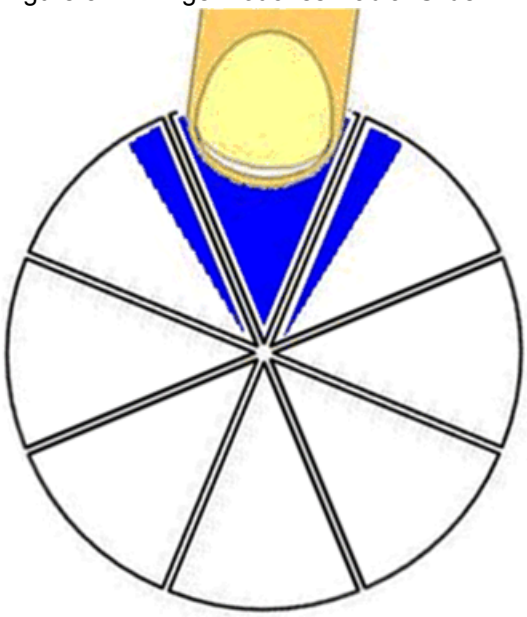


Radial Sliders

CSDPLUS supports two slider types: linear and radial. Linear sliders have a beginning and an end, whereas radial sliders do not (see Figure 5). In either case, when a touch occurs, the centroid algorithm considers the largest signal from sensors adjacent to the sensor to interpolate the exact position of the touch. Radial sliders are not diplexed. The CSDPLUS User Module has two special API functions for radial sliders. The first function, CSDPLUS_wGetRadiaPos(), returns centroid location. The second function, CSDPLUS_wGetRadialInc(), returns finger shift in resolution units. When the finger moves in a clockwise direction, CSDPLUS_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. For diplexed sliders, use (2 x pins used for sensors – 1) x 2^8 – 1).

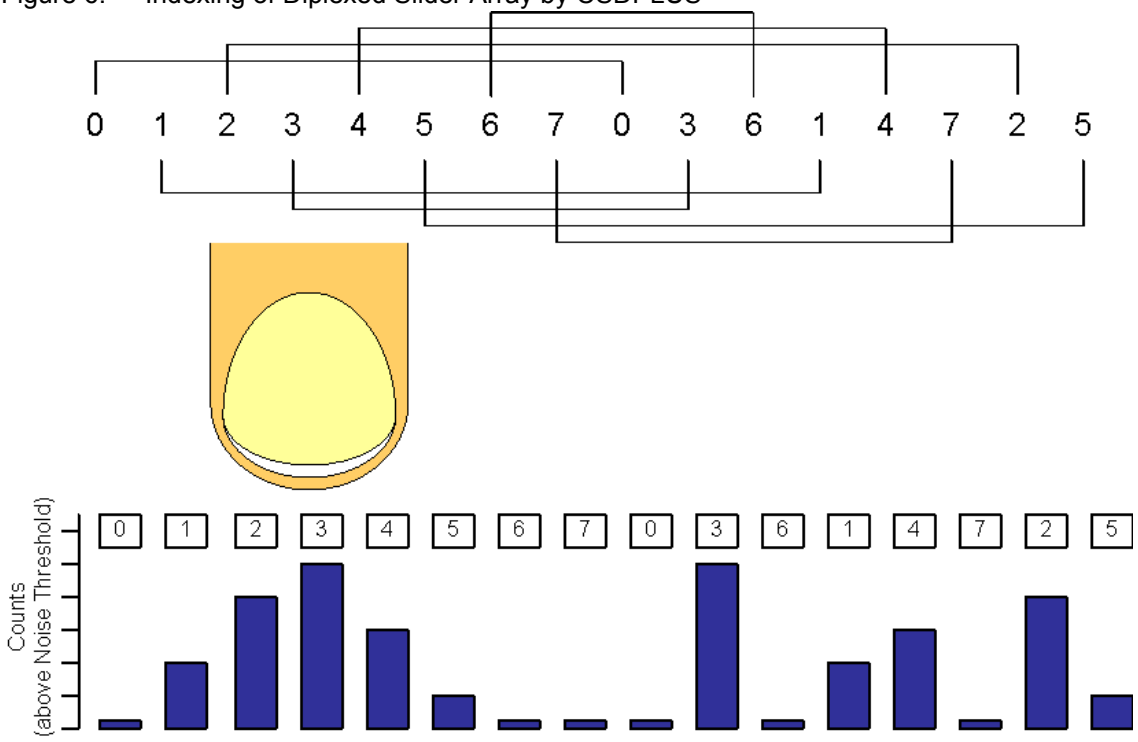
Figure 5. Finger Touches Radial Slider



Diplexing

In diplexing, each pin on the PSoC 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 CSDPLUS 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 CSDPLUS



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 mapped 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.

Take care 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 also needs to be reasonably well matched (within 10 pF). When you select diplexing, the diplex sensor index table is automatically generated by the CSDPLUS Wizard. Table 1 shows 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

External Component Selection (C_{mod})

CSDPLUS requires an external modulation capacitor, C_{mod}, connected from VSS to one of two dedicated PSoC pins, either P0[1], or P0[3]. The C_{mod} pin assignment is made in the CSDPLUS wizard under "Global Settings > Modulator Capacitor Pin". Do not use the selected pin for any other purpose. Use a ceramic capacitor, and note that the recommended value for the external modulation capacitor is 2.2 nF. The temperature capacitance coefficient is not important. Moreover, to suppress interference, use a 560-Ω series resistor on all CapSense sensor traces. Place this resistor as close to the PSoC as is practical.

Driven Shield Electrode

A driven shield electrode is an optional feature that helps to reduce parasitic sensor capacitance (C_p). Benefits include improved sensor sensitivity and prevention of false sensor triggering when water is detected on the overlay.

Position a shield electrode 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. A corresponding increase in sensor capacitance signal occurs that might be large enough to falsely activate the sensor. The driven shield electrode nulls out parasitic capacitive coupling. Therefore, the presence of water has a negligible influence on sensor capacitance signal, thus preventing false activations.

Figure 7. Driven Shielding Electrode PCB Layout

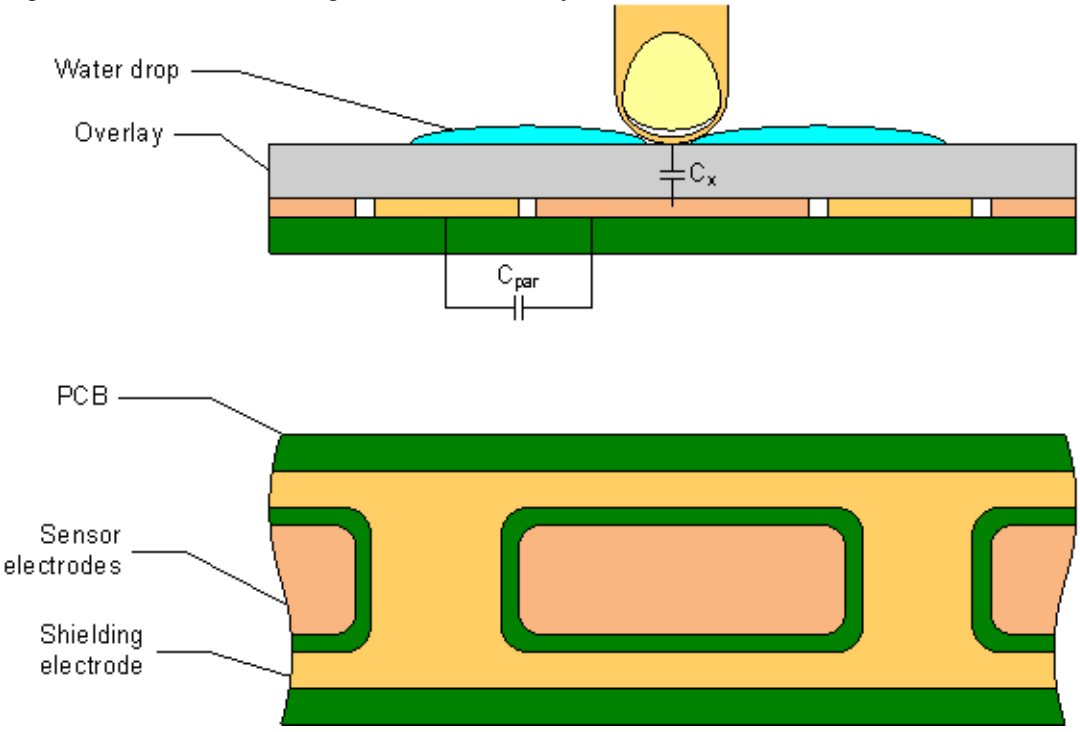
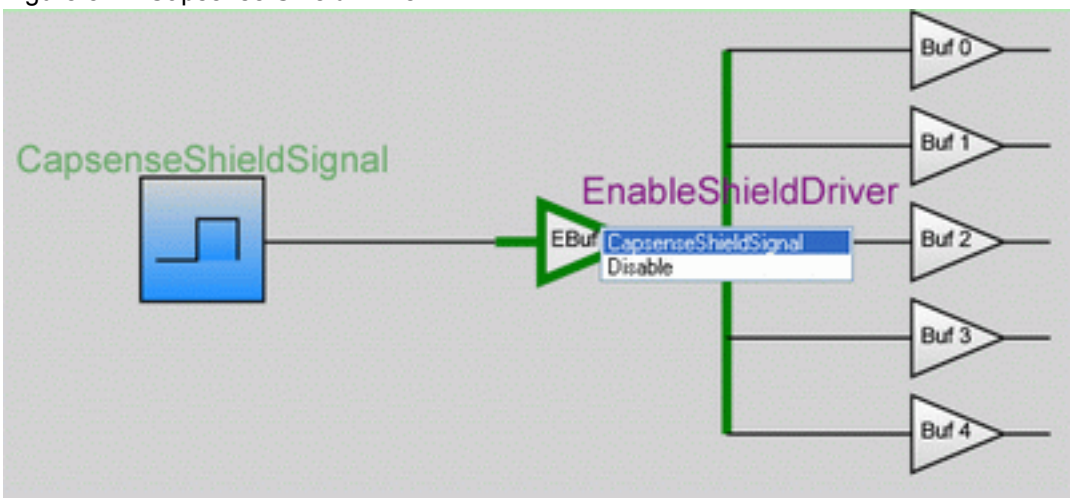


Figure 7 illustrates a driven shielding electrode for a button. As an alternative, you can position the shielding electrode on the opposite PCB layer, including the plane under the button. In this case, a hatch pattern with a fill ratio of 30 percent to 40 percent is recommended. No additional ground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

The shield electrode must be connected to any of the dedicated PSoC pins: P2[4], P2[2], P0[2], P0[0] or P1[2]. Set the drive mode for the selected pin to **Strong**. To reduce emitted EMI, connect a 560-Ω slew limiting resistor between the PSoC device and the shielding electrode.

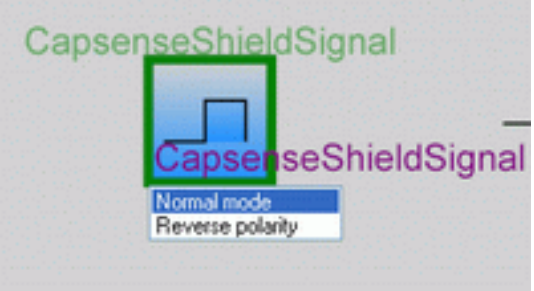
The Capsense Shield Signal can be routed to P2[4], P2[2], P0[2], P0[0] or P1[2] pins through 5 ShieldBuffers and is enabled by EnableShieldDriver block in the PSoC Designer Chip Editor.

Figure 8. Capsense Shield Driver



The Shield Driver can be configured to drive Shield Signal in two modes: Normal mode and Reverse polarity. When Reverse polarity is selected Shield driver clock will be inverted.

Figure 9. CapsenseShieldSignal block



Power Supply Requirement

Table 2. CSDPLUS Power Supply Requirement

Parameter	Min	Typ	Max	Unit	Test Conditions and Comments
V_{DD}	1.71[a]	-	5.50	V	If the V_{DD} droop exceeds 5% of the base V_{DD}, the rate at which V_{DD} droops and recovers must not exceed 200 mV/s.

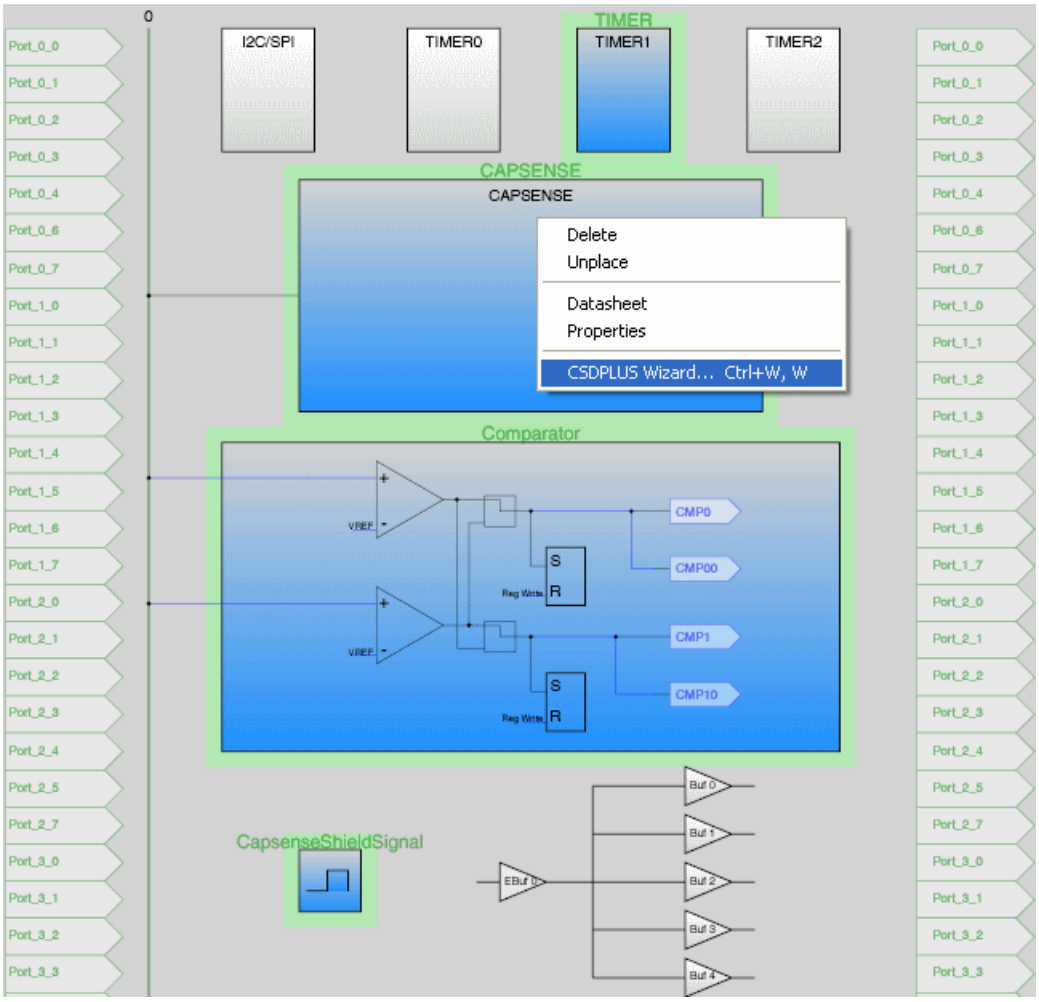
a. 1.8 V is an absolute minimum V_{DD} spec. Allowing V_{DD} to droop below 1.8 V can introduce excessive noise.

Placement

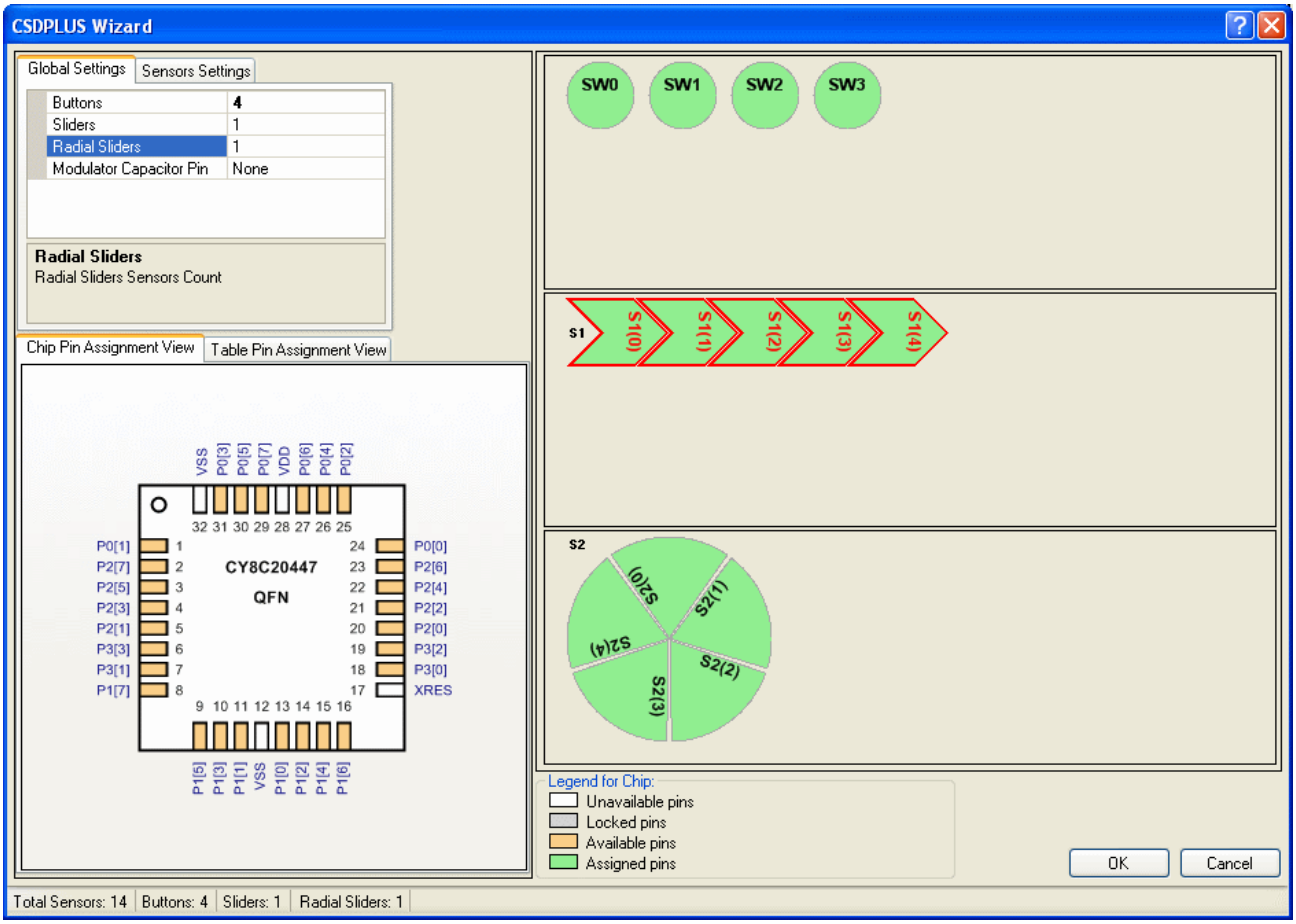
The Timer1, CapSense, Comparator, and Capsense Shield Signal blocks are assigned to CSDPLUS when the user module is instantiated. Alternate placements are not available. User modules that require dedicated pin resources, including the LCD and I2CHW, must be placed before starting the CSDPLUS Wizard. Doing so ensures that the dedicated pins are reserved and cannot be inadvertently assigned as sensors when sensors are mapped to I/O pins in the CSDPLUS Wizard. Avoid P1[0] and P1[1] when placing capacitive sensor connections. Those pins are used to program the part and may have excessive routing capacitance, which hampers sensor sensitivity.

CSDPLUS Wizard

1. To access the CSDPLUS Wizard, right-click on any user module block in the Chip Editor and select the CSDPLUS Wizard with a left mouse click.



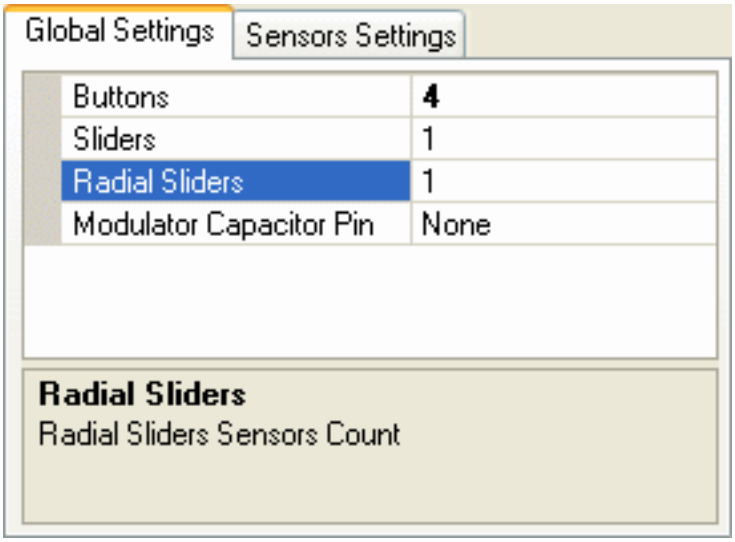
2. The Wizard opens, showing the numeric entry boxes for the number of sensors, slider sensors, and radial sliders.



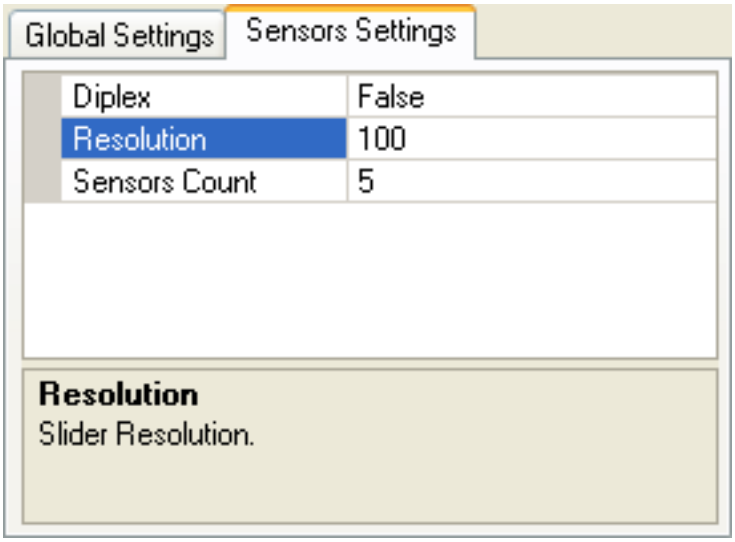
Wizard Pin Legend

- White – The pin cannot be used as a CapSense input.
- Gray – The pin is locked, for one of two reasons. The first possibility is that another user module, such as the LCD or I2C, has claimed the pin. The second possibility is that the name of the pin has changed from its default.To return the pin name to its default, do the following: While in the pinout view, expand the pin, from the ‘Select’ menu, choose ‘Default’. 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.

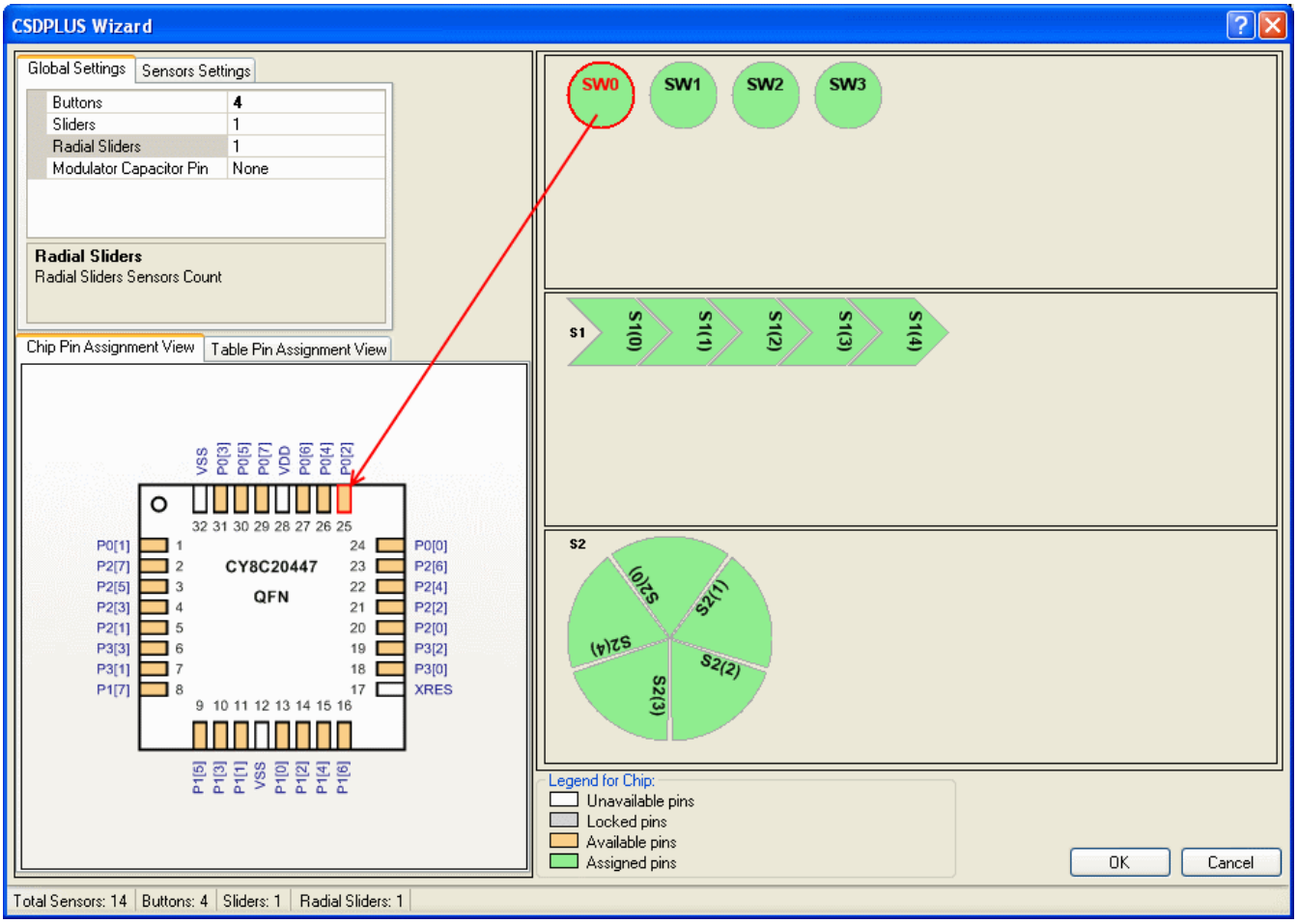
3. Select the Global Settings tab to type the number of independent buttons, sliders, and radial sliders. The total number of sensors (buttons and slider elements) is limited to the number of pins available. After entering the data, press the [Enter] key to update the display with the new value.
4. Select the modulator capacitor pin (C_{mod}). Choose from either P0[1] or P0[3].



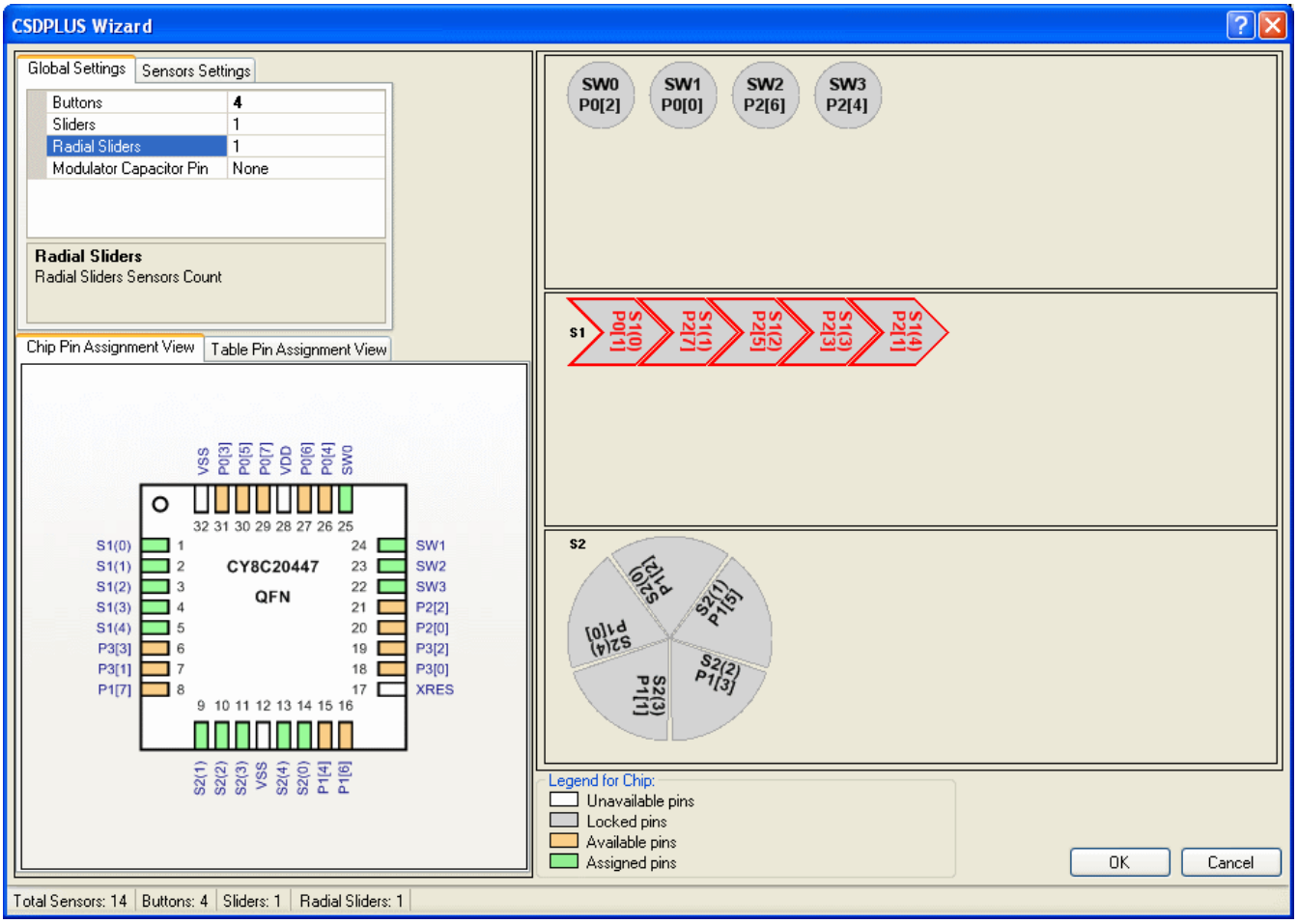
5. Select the Sensors Settings tab to make choices for buttons, sliders, and radial sliders. To alter the settings, click one of your sliders to activate it. Type the number of sensor elements in each slider. The practical minimum number of sensors in a slider is five, while the maximum is limited only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



6. Type the output resolution. The minimum value is five. CSDPLUS tries 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 desired. 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 to the pin in the Pin Assignment View. You can use either the Chip Pin Assignment View or the Table Pin Assignment View. The I/O pin turns green after selection and is no longer available. To change sensor assignments, drag 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 will be unassigned.

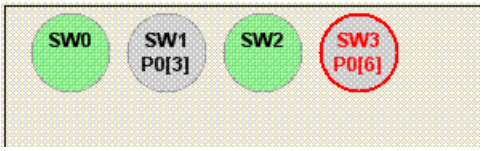
After completing the Wizard, click Generate Application. Based on your entries for sensor count, pin assignment, diplexing, and resolution, a set of tables will be generated.

Sensors 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 and Table Pin Assignment Views. The assigned sensors are marked grey. This section consists of the following three tiles:

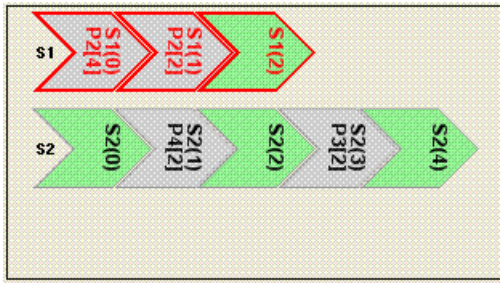
- Buttons representation – The Buttons sensors are displayed in the wizard as shown in Figure 10. Each button sensor element has its own title. All buttons support the drag-and-drop capability for the Chip and Table Pin Assignment Views. The assigned Port Pin number is displayed under the button title if the button is already assigned. The button widget is set in a red frame if the button sensor is selected.

Figure 10. Buttons Representation Section



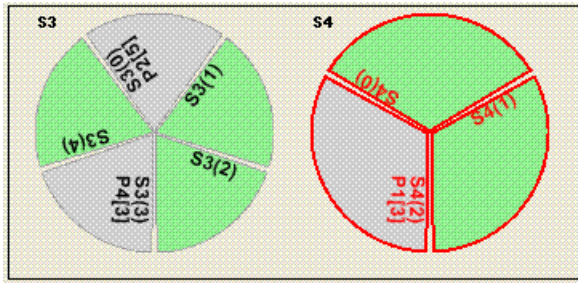
- Sliders representation – The Sliders are displayed as a sequence of sensor segments as shown in Figure 11. The specific sensor name is displayed for each segment in a slider. All sensors in a slider support drag-and-drop capability for the Chip and Table Pin Assignment Views. The assigned Port Pin 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 11. Sliders Representation Section



- Radial sliders representation – The Radial Sliders are displayed as a pie tile of segments as shown in Figure 12. The sensor name is displayed for each segment in a radial slider. All sensors in a radial slider support drag-and-drop capability for the Chip and Table Pin Assignment Views. 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 12. Radial Sliders Representation Section

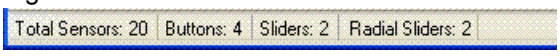


Status Bar

The status bar displays common information about the design (see Figure 13):

- Total sensors – displays the overall number of 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

Figure 13. Status Bar



Wizard Buttons

The CSDPLUS User Module wizard offers buttons with predefined functionality.

1. “OK” – This button checks to determine if the wizard parameters are correct and all sensors are assigned. If yes, then the wizard saves parameters and closes; otherwise, it shows an appropriate warning message, does not save parameters, and remains open.
2. “Cancel” – This button closes the wizard without saving parameters.
3. “Close” – A standard window close button is located on the title bar, at the top right corner of the wizard form. If you click on Close, the wizard closes without saving any parameters.

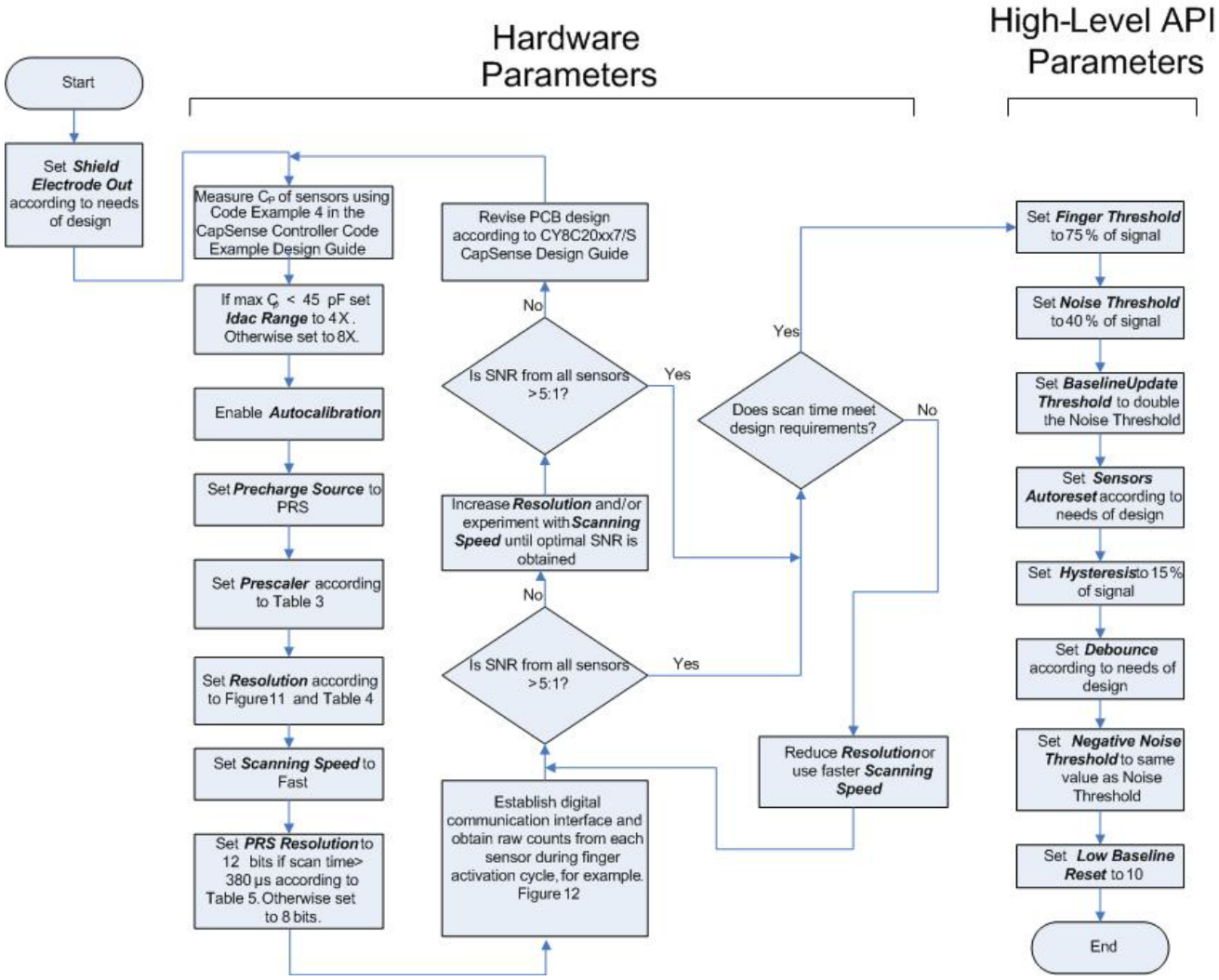
4. “Help” – This button calls the help page, giving reference information on how to use the CSDPLUS User Module wizard. It briefly describes the CSDPLUS User Module wizard features. Help is available through a standard window help button, labeled with a question mark, and located on the title bar, at the top right corner of the wizard form.

User Module Parameters - Tuning Guide

After you complete configuration and I/O pin assignment in the CSDPLUS Wizard, set the user module parameters. Note that to enact change in any user module parameter, you must regenerate the project.

Figure 8 is a flowchart showing the tuning process for CSDPLUS UM parameters. These parameters are separated into two categories: hardware and high-level API. The parameters in these categories affect the behavior of the capacitive sensing system in different ways and, therefore, are treated separately in this section. However, a complementary relationship exists between the sensitivity of each sensor, as determined by the hardware parameter settings, and many of the high level API parameter settings. Keep that in mind when you change any hardware parameter to ensure that the corresponding high level API parameters are adjusted accordingly. Tuning of CSDPLUS User Module Parameters should always begin with the hardware parameters.

Figure 14. CSDPLUS User Module Parameters Tuning Flowchart



- For measuring sensor C_P, refer to the CapSense® Controller Code Examples Design Guide

- For revising PCB design, refer to the CY8C20xx7/S CapSense Design Guide

Hardware Parameters

Hardware parameters configure the hardware that the CSD method uses to convert the physical capacitance of each sensor into a digital code. This section describes these parameters and guides you on how each should be tuned based on system characteristics and other parameters.

By default, hardware parameters are global settings that apply to all CapSense sensors in a design. In designs where the total parasitic capacitance of each sensor (Cp) or sensor sensitivity vary over a wide range, you may not find global hardware parameters suitable for all sensors. In such cases, the SetPrescaler(i) and SetScanMode(i) API functions can be used to configure the respective hardware parameters for each sensor where (i) is the sensor index before calling the ScanSensor(i) API function. The Sample Code section contains such an example.

iDAC Range

This parameter sets the output range of the iDAC. The choices are 4X and 8X. For projects where the maximum sensor C_p is less than 45 pF use 4X; otherwise use 8X. The default setting is 4X.

Autocalibration

This parameter enables a successive approximation for the iDAC setting that establishes a raw count baseline at or above 85 percent. Always set autocalibration to enabled in CSD designs. The ability of the autocalibration algorithm to successfully set the iDAC relies on a proper Prescaler setting and the recommended size for Cmod. The default setting is Enabled.

iDAC Value

This parameter determines the current output of iDAC when autocalibration is disabled. When autocalibration is enabled, as recommended, this parameter is overridden and has no effect. When autocalibration is disabled, raising this parameter lowers the raw count baseline and vice versa. The default setting for this parameter is 20.

Compensation iDAC Value

This parameter determines the current output of compensation iDAC when autocalibration is disabled. When autocalibration is enabled, as recommended, this parameter is overridden and has no effect. When autocalibration is disabled, raising this parameter lowers the raw count baseline and vice versa. The default setting is 0.

Precharge Source

This parameter selects the sensor switching clock source. The available options are Prescaler, which uses the IMO through a divider, or PRS. The default setting is PRS. The PRS passes the divided IMO clock through a Pseudo Random Generator, providing a spread spectrum clock. PRS provides superior noise immunity and lower noise emissions and, therefore, is the recommend default setting for Precharge Source. In some instances, the Prescaler Precharge Source can provide a higher signal-to-noise ratio (SNR). However, when using copper circuitry, this SNR improvement is usually marginal and rarely justifies forgoing the benefits of PRS.

Prescaler

Prescaler is the divider applied to the IMO to develop the precharge Clock. This is the most critical hardware UM parameter in properly tuning a CSDPLUS design. Prescaler depends on the selected precharge source, IMO, and the Cp of the sensor/sensors being scanned. For the recommended prescaler settings based on these parameters, see Table 3. The default setting is 2.

Table 3. Prescaler Settings Based on Precharge Source, IMO, and C_p

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

Note 1: This combination of precharge source, prescaler, and C_p is not recommended

Resolution

This parameter sets the iDAC resolution. Available choices are 9 to 16 bits. Raising the resolution increases sensitivity, SNR, and noise immunity at the expense of scan time. The maximum raw count (full-scale range) for scanning resolution n is 2^n-1. Table 4 provides recommended resolution settings based on Cp and the finger capacitance Cf. Cf is the change in capacitance of a sensor when a finger is placed on the sensor. C_f depends on overlay thickness, sensor size, and proximity of the sensor to other large conductors. Figure 9 provides C_f values as a function of overlay thickness and circular sensor diameter. The default setting for this parameter is 12.

Figure 15. Finger Capacitance (C_f) Based on Overlay Thickness and Circular Sensor Diameter

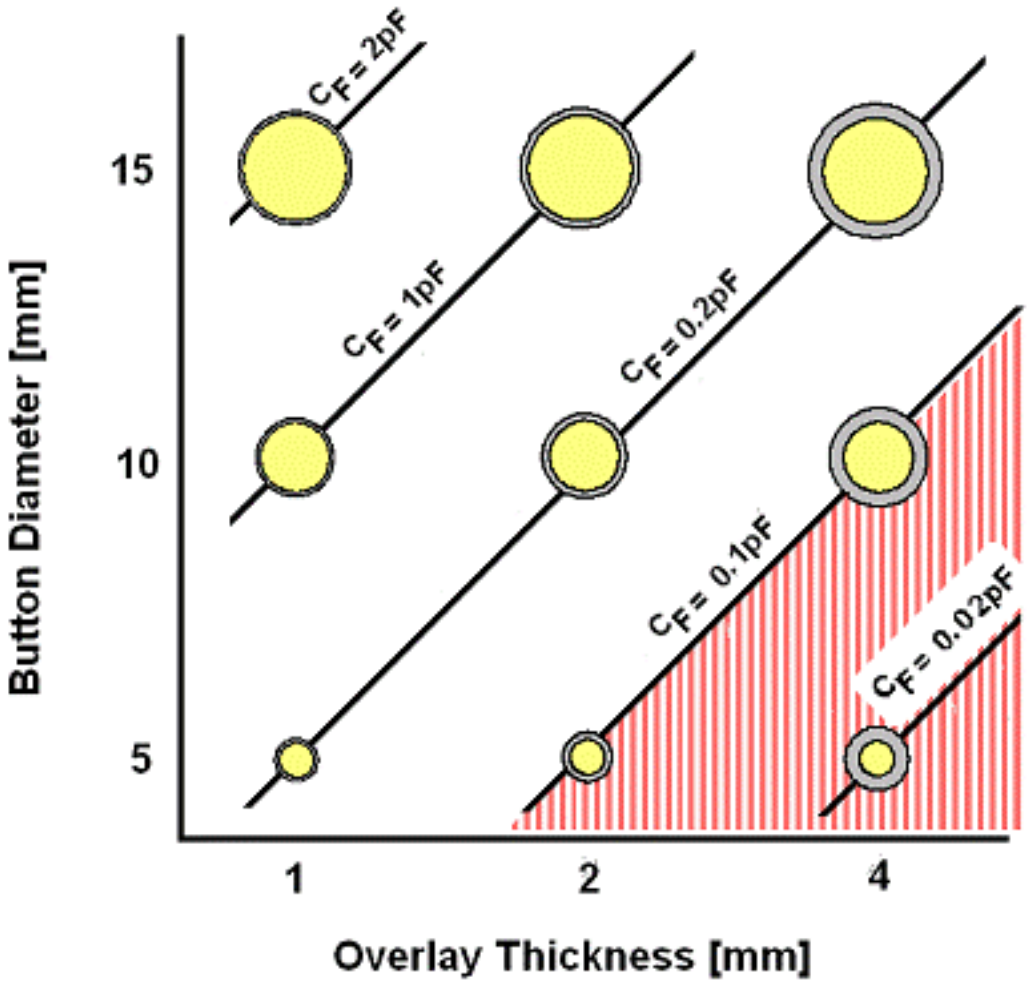


Table 4. Resolution Settings Based on Finger Capacitance and Cp

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

Scanning Speed

This parameter controls the integration time for each LSB of the scan result. The choices are Ultra Fast, Fast, Normal, and Slow. Fast is generally a good starting point. In some cases, slower scanning can yield higher SNR at the expense of longer scan time and higher power consumption. The default

setting for this parameter is Normal. Table 5 shows the actual scan time in µs for a single sensor based on resolution and scanning speed.

Table 5. Scan Time (µs) for a Single Sensor Based on Resolution and Scanning Speed

Resolution (bits)	Scanning Speed			
	Ultra Fast	Fast	Normal	Slow
9	110	130	175	260
10	130	175	260	430
11	175	260	430	780
12	260	430	780	1500
13	430	780	1500	2900
14	780	1500	2900	5600
15	1500	2900	5600	11250
16	2900	5600	11250	22500

PRS Resolution

This parameter changes the PRS sequence length. Possible values are 8 bit and 12 bit. Corresponding sequence lengths are 511 and 2047 input clock periods, respectively. When very short scan times are needed, an 8-bit PRS must be used to avoid excessive noise. Scan time is determined by the resolution (not to be confused with PRS resolution) parameter. For scan times of ≤ 380 µs, PRS resolution should be set to 8 bits; for scan times of > 380 µs, PRS resolution should be set to 12 bits. The default setting is 8 bits.

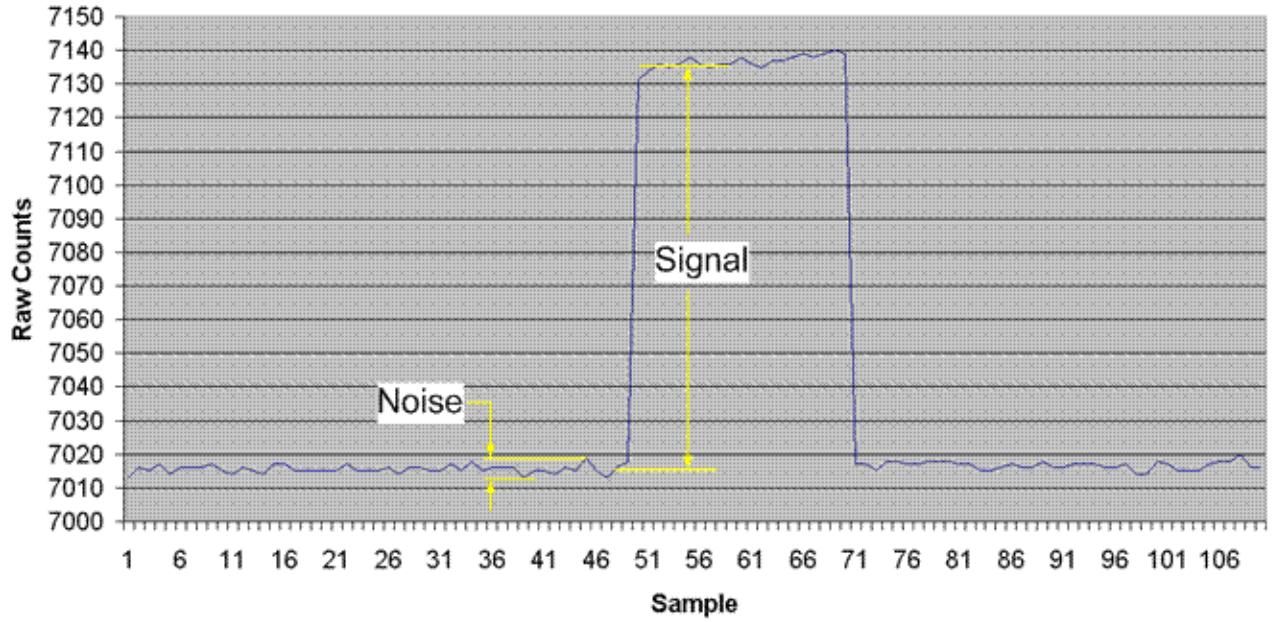
High Level API Parameters

High level API parameters determine the behavior of high level firmware algorithms that discriminate between sensor activations and noise, as well as compensate for signal drift caused by environmental conditions. To determine proper values for these parameters, establish a digital communication interface with the system to monitor raw counts, baseline and difference counts during a finger activation event for each sensor. This data is stored in arrays called CSDPLUS_waSnsResult[], CSDPLUS_waSnsBaseline[], and CSDPLUS_waSnsDiff[] respectively.

The high level API parameters settings are based primarily on ambient noise and finger signal strength as indicated by this data. Noise and signal strength depend on EMI environment, PCB layout, overlay thickness, and other physical characteristics of the system. Therefore, the data used as the basis for setting these parameters must be taken in situ with the system in its final assembled state. Moreover, you must use the same EMI environment as will exist in service.

Figure 10 shows the typical raw counts obtained from a sensor during a finger activation cycle (when a sensor is activated, then deactivated). Superimposed over the data are labels that indicate how noise and signal are to be calculated based on the raw data. Where appropriate, the high level parameter descriptions that follow include information on how to set each parameter based on these noise and signal values. The SNR must be at least 5:1 for robust operation of the CapSense system. If the SNR is less than 5:1, the hardware parameters must be adjusted or the PCB layout must be changed to raise the SNR to at least 5:1.

Figure 16. Typical Raw Counts from a Sensor During Finger Activation Cycle



Finger Threshold

Finger threshold quantifies the nominal change in raw counts, not including hysteresis, needed for a sensor to be considered activated. Possible values range from 5 to 255. By default, a global finger threshold is applied to all sensors by the CSD_SetDefaultFingerThresholds() API function. For independent buttons (not contained in a slider group), unique finger thresholds can be set for each button by writing the desired values into the index position corresponding to the sensor number in the CSD_baBtnFThreshold[] global array. To set this parameter, you must analyze the raw count response of each sensor to finger activation. Finger threshold for each sensor should be set to 75 percent of the raw count signal (see Figure 10) when the sensor is touched by the most limiting, or the smallest, finger. Finger threshold has no meaning and function for sensors that are elements in a slider. The default setting for this parameter is 60.

Noise Threshold

Positive changes in raw counts below this level are accumulated to update raw count baseline. Possible values are 5 to 255. For button sensors, when Sensors Autoreset is disabled, (default) count values above this threshold do not update the baseline; and for slider elements, count values below this threshold are not counted in the centroid calculation. Noise threshold is a global parameter that applies to all sensors. A good starting point for noise threshold is 40 percent of the raw count signal (see Figure 10). The default setting is 10.

BaselineUpdate Threshold

CSDPLUS uses a so-called bucket method to update the baseline count in the CSDPLUS_UpdateSensorBaseline() API function. Half the difference between the raw count value and the baseline count is added to the bucket when both of these conditions are met:

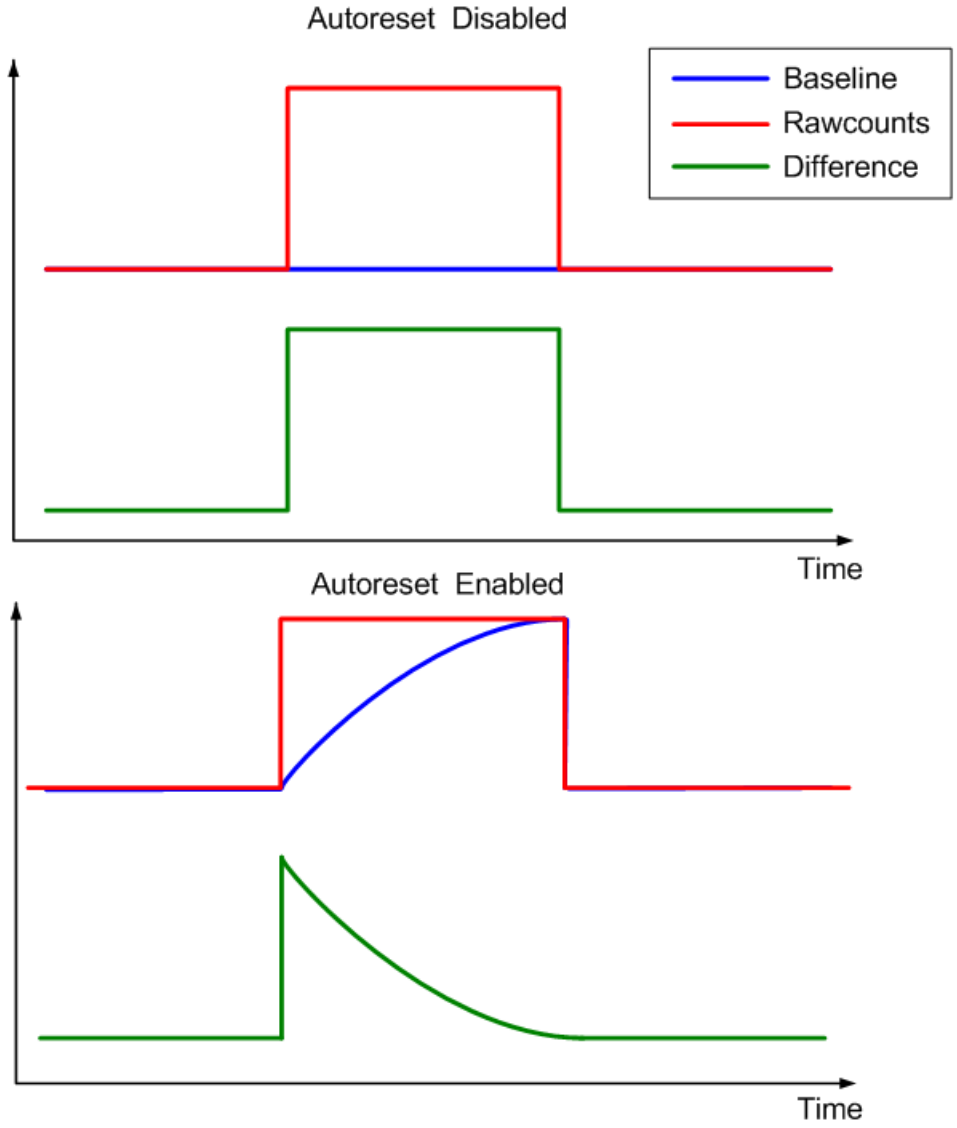
1. The raw count value returned from a scan is above the current baseline AND.
2. The difference between the raw count value and the baseline is below the noise threshold.

The BaselineUpdateThreshold sets the value that the "bucket" must reach for the baseline to be incremented. A good starting point for this parameter is double the Noise Threshold. Possible values are 0 to 255; the default setting is 100.

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, baseline is updated only when the difference between raw count and baseline is below the noise threshold. Figure 11 illustrates this parameter's effect on baseline update. When Sensors Autoreset is set to Enabled, baseline is always updated without regard to noise threshold. This limits the maximum activated time of sensors (typically to 5 - 10s). However, it provides the benefit of preventing 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 a rapid temperature change. When Sensors Autoreset is Disabled, 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 appendices for more details about this parameter.

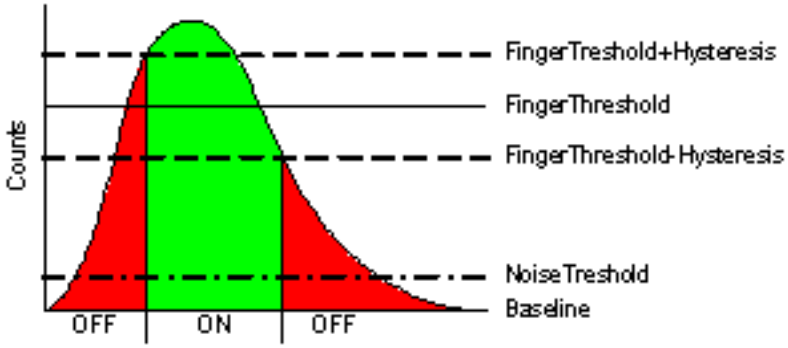
Figure 17. Affect of the Sensor Autoreset Parameter on Baseline



Hysteresis

To improve button sensor activation state recognition and provide more stable operation, hysteresis is used to gate sensor activation status from OFF to ON and back to OFF (see Figure 12). Count values must be greater than FingerThreshold+Hystersis to change the state from OFF to ON. Count values must be less than FingerThreshold-Hysteresis to change state from ON to OFF. A good starting point for the Hysteresis is 15 percent of the raw count signal (see Figure 10). The default setting is 10.

Figure 18. Relationship Between Finger Threshold and Hysteresis on Button Sensor State



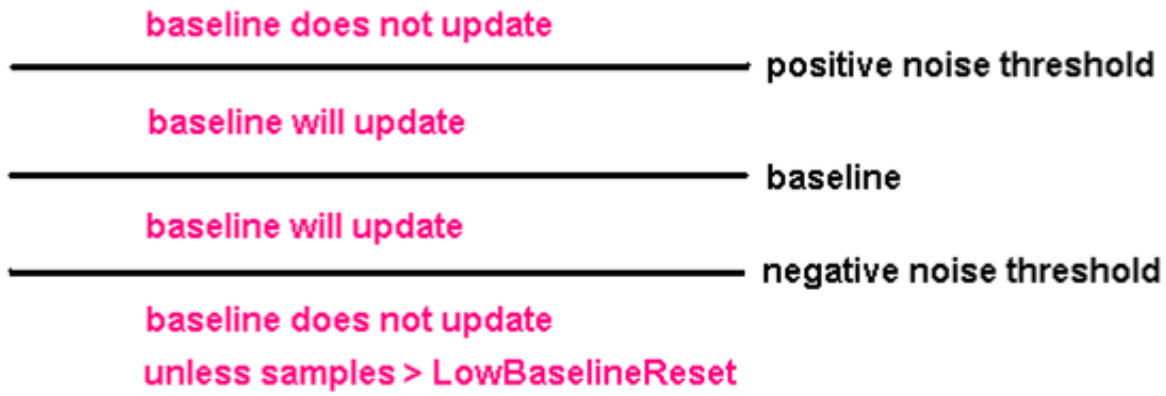
Debounce

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 + hysteresis for the number of samples specified by this parameter. The Debounce counter is incremented by the CSDPLUS_bIsSensorActive or CSDPLUS_bIsAnySensorActive API functions. Possible values are 1 to 255. A setting of '1' has no debounce but provides the fastest response. The default setting is 3.

Negative Noise Threshold

This parameter applies when raw counts fall below baseline. Negative noise threshold establishes a level relative to the current baseline line above which the baseline will reset, that is, snap down to the raw count value. If raw counts are below this level, the baseline will not reset unless the Low Baseline Reset parameter limit is reached. In that case, the baseline will reset. Figure 13 shows the relationship between the noise thresholds and baseline reset. A good starting point for negative noise threshold is to use the same value as noise threshold. The default setting is 10.

Figure 19. Relationship Between Noise Thresholds and Baseline Reset



Low Baseline Reset

This parameter works with negative noise threshold to set the number of samples with raw counts less than the baseline needed to make baseline snap down to the raw count level. If the raw count value is less than the Baseline - Negative Noise Threshold for the number of samples set by Low Baseline Reset, the baseline will snap down to the raw count value. Low Baseline Reset is generally used to correct a finger-on-at-startup condition. A good starting point is 10; the default setting is 50.

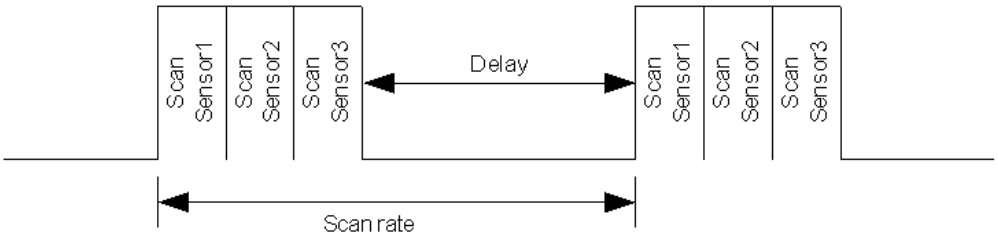
FMEA_Cp_Range_Test

This parameter enables the GetSnsParasiticCapacitance API. It adds the conditional compilation of the GetSnsParasiticCapacitance, Calibrate, div_24_16_24, and mul_16x16_32 functions.

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 20. 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 allow 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.

Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns the CSDPLUS_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 CSDPLUS 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 satisfies this requirement. Assembly language programmers must also 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.

Entry Points are supplied to initialize the CSDPLUS, begin the sampling, and stop the CSDPLUS. In all cases, the instance name of the module replaces the CSDPLUS 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 are several global arrays such as:

- CSDPLUS_waSnsResult[]
- CSDPLUS_waSnsBaseline[]
- CSDPLUS_waSnsDiff[]
- CSDPLUS_baSnsOnMask[]
- CSDPLUS_baBtnFThreshold[]
- CSDPLUS_baDAC[]
- CSDPLUS_baCompensationDAC[]

CSDPLUS_waSnsResult[]– This is an integer array used by the CSDPLUS_ScanSensor() function to store the raw count of each actual sensor scan. The array size is equal to the sensor count.

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

- CSDPLUS_UpdateAllBaselines();
- CSDPLUS_UpdateSensorBaseline();
- CSDPLUS_InitializeBaselines().

CSDPLUS_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.

CSDPLUS_baSnsOnMask[] – This is a byte array that holds the sensor on or off state (for buttons or sliders). CSDPLUS_baSnsOnMask[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). CSDPLUS_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 CSDPLUS_baSnsOnMask[] data is updated by CSDPLUS_bIsSensorActive(BYTE bSensor) function or CSDPLUS_bIsAnySensorActive() routines.

CSDPLUS_baBtnFThreshold[]– This is a byte array used to store the threshold for each sensor. The array size is equal to the sensor count.

CSDPLUS_baDAC[]– This is a byte array used to store the autocalibrated IDAC value for each actual sensor. The value for each sensor is set by CSDPLUS_CalibrateSensor() and used by CSDPLUS_ScanSensor(). The array size is equal to the sensor count.

CSDPLUS_baCompensationDAC[]– This is a byte array used to store compensated IDAC value corresponding to each sensor. The array size is equal to the sensor count.

CSDPLUS_Start

Description:

This function starts the CSDPLUS User Module. It initializes the global variables, configures and connects Cmod to the amux bus, and configures the CapSense block and the associated hardware. This function should be called before calling any other user module functions.

This API fills the global array CSDPLUS_baCompensationDAC[] with the values entered by the user in accordance to user module parameter "Compensation iDAC Value".

C Prototype:

```
void  CSDPLUS_Start(void)
```

Assembly:

```
lcall CSDPLUS_Start
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDPLUS_Stop

Description:

This function stops the sensor scanner, disables internal interrupts, and calls CSDPLUS_ClearSensors() to reset all sensors to an inactive state.

C Prototype:

```
void  CSDPLUS_Stop(void)
```

Assembly:

```
lcall CSDPLUS_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDPLUS_Resume

Description:

This function resumes the user module operation after CSDPLUS_Stop() call.

C Prototype:

```
void  CSDPLUS_Resume(void)
```

Assembly:

```
lcall CSDPLUS_Resume
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDPLUS_ScanSensor

Description:

This function scans the selected sensor. Each sensor has a unique number within the sensor array that is assigned by the CSDPLUS Wizard in sequence. Sw0 is sensor 0, Sw1 is sensor 1, and so on.

C Prototype:

```
void  CSDPLUS_ScanSensor(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall CSDPLUS_ScanSensor
```

Parameters:

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

Return Value:

None

Side Effects

**

CSDPLUS_ScanAllSensors

Description:

This function scans all of the configured sensors by calling CSDPLUS_ScanSensor() for each sensor index.

C Prototype:

```
void  CSDPLUS_ScanAllSensors(void)
```

Assembly:

```
lcall CSDPLUS_ScanAllSensors
```

Parameters:

None

Return Value:

None

Side Effects

**

CSDPLUS_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 CSDPLUS_UpdateSensorBaseline() is called, a difference count is calculated by subtracting the previous baseline from the raw count value. This difference is stored in the CSDPLUS_waSnsDiff[] array and is provided to you.

2. If Sensors Autoreset is disabled, each time CSDPLUS_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 have reached 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 CSDPLUS_waSnsDiff[] array is reset to 0. Therefore, this array does not contain elements with values greater than 0 but below the noise threshold.

C Prototype:

```
void  CSDPLUS_UpdateSensorBaseline(BYTE bSensorNum)
```

Assembly:

```
mov   A, bSensorNum
lcall CSDPLUS_UpdateSensorBaseline
```

Parameter:

A => Sensor Number

Return Value:

None

Side Effects:

**

CSDPLUS_UpdateAllBaselines

Description:

This function uses the CSDPLUS_bUpdateSensorBaseline() function to update the baselines for all sensors.

C Prototype:

```
void CSDPLUS_UpdateAllBaselines(void)
```

Assembly:

```
lcall CSDPLUS_UpdateAllBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDPLUS_bIsSensorActive

Description:

This function 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 CSDPLUS_baSnsOnMask[] array.

C Prototype:

```
BYTE  CSDPLUS_bIsSensorActive(BYTE bSensorNum)
```

Assembly:

```
mov   A, bSensorNum
lcall CSDPLUS_bIsSensorActive
```

Parameters:

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

Return Value:

BYTE: 1 – Selected sensor is active; BYTE 0 – Selected sensor is not active.

Side Effects:

**

CSDPLUS_bIsAnySensorActive

Description:

This function checks the difference count array for all sensors compared to their finger threshold. Calls CSDPLUS_bIsSensorActive() for each sensor so the CSDPLUS_baSnsOnMask[] array is up to date after calling this function.

C Prototype:

```
BYTE  CSDPLUS_bIsAnySensorActive(void)
```

Assembly:

```
lcall CSDPLUS_bIsAnySensorActive
```

Parameters:

None

Return Value:

BYTE: 1 – One or more sensors are active; BYTE 0 – No sensors are active.

Side Effects:

**

CSDPLUS_wGetCentroidPos

Description:

This function checks a difference array for a centroid. If one exists, the offset and length are stored in temporary variables and the centroid position is calculated to the resolution specified in the CSDPLUS Wizard. This function is available only if slider is defined by the CSDPLUS Wizard.

C Prototype:

```
WORD  CSDPLUS_wGetCentroidPos(BYTE bSnsGroup)
```

Assembly:

```
mov   A, bSnsGroup
lcall CSDPLUS_wGetCentroidPos
```

Parameters:

bSnsGroup => Group Number

This parameter is a reference to a specific group of sensors used as a slider. Group 0 is for buttons. Sliders are contained in group 1 and higher.

Return Value:

WORD: 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. To avoid getting negative difference values, call the routine 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 CSDPLUS 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 CSDPLUS_blsSensorActive() routine to determine which slider segments are touched, if required.

Note If noise counts on the slider segments are greater than the noise threshold, this subroutine may generate a false centroid result. Set the noise threshold carefully (high enough above the noise level) so that noise does not generate a false centroid.

CSDPLUS_wGetRadialPos

Description:

This function checks a difference array for a centroid. If one exists, the centroid position is calculated to the resolution specified in the CSDPLUS Wizard. This function is available only for radial slider that is defined by the CSDPLUS Wizard.

C Prototype:

```
WORD  CSDPLUS_wGetRadialPos(BYTE bSnsGroup)
```

Assembly:

```
mov   A, bSnsGroup
lcall CSDPLUS_wGetRadialPos
```

Parameters:

bSnsGroup => Group Number

This parameter is a number of radial slider with which you are working. Find its number through the CSDPLUS UM wizard on the left hand of radial slider representation (for example, s2, the radial slider number is 2).

Return Value:

WORD: position value of the radial slider, LSB in A and MSB in X.

Side Effects:

To avoid getting negative difference values and baseline update, call the routine 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 CSDPLUS Wizard. If no sensors are active, the function returns -1 (FFFFh).

Note If noise counts on the slider segments are greater than the noise threshold, this subroutine may generate a false centroid result. The noise threshold should be set carefully (high enough above the noise level) so that noise does not generate a false centroid.

CSDPLUS_wGetRadialInc

Description:

This function returns actual finger shift, which is the difference between current and previous finger positions. This function works in pair with CSDPLUS_wGetRadialPos() and takes data generated by the latter (data is saved in internal variables).

C Prototype:

```
WORD  CSDPLUS_wGetRadialInc(BYTE bSnsGroup)
```

Assembly:

```
mov   A, bSnsGroup
lcall CSDPLUS_wGetRadialInc
```

Parameters:

bSnsGroup => Group Number

This parameter is a number of radial slider with which you are working. Find it through the CSDPLUS UM wizard on the left hand of radial slider representation (for example, for s2, the radial slider 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. If there was no touch during previous scan (the last but one time CSDPLUS_wGetRadialPos() returned -1 (FFFFh)) or there is no touch at the moment (this time CSDPLUS_wGetRadialPos() returned -1 (FFFFh)).

Side Effects:

The routine should be called only after CSDPLUS_wGetRadialPos() API. That is because it uses internal data CSDPLUS_waSliderPrevPos and CSDPLUS_waSliderCurrPos that are set by the CSDPLUS_wGetRadialPos().

CSDPLUS_InitializeSensorBaseline

Description:

This function loads the CSDPLUS_waSnsBaseline[bSensorNum] array element with an initial value by scanning the selected sensor. The raw count value is copied in to the baseline array element for the selected sensor. This function can be used to reset the baseline of an individual sensor.

C Prototype:

```
void  CSDPLUS_InitializeSensorBaseline(BYTE bSensorNum)
```

Assembly:

```
mov   A, bSensorNum
lcall CSDPLUS_InitializeSensorBaseline
```

Parameters:

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

Return Value:

None

Side Effects:

**

CSDPLUS_InitializeBaselines

Description:

This function loads the CSDPLUS_waSnsBaseline[] array with initial values by scanning each sensor. The raw count values are copied to the baseline array for each sensor.

C Prototype:

```
void  CSDPLUS_InitializeBaselines(void)
```

Assembly:

```
lcall CSDPLUS_InitializeBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDPLUS_SetDefaultFingerThresholds

Description:

This function loads the CSDPLUS_baBtnFThreshold[] array with the FingerThreshold parameter value. If the CSDPLUS_baBtnFThreshold[] array is not manually loaded with custom values, this function must be called before scanning.

C Prototype:

```
void  CSDPLUS_SetDefaultFingerThresholds(void)
```

Assembly:

```
lcall CSDPLUS_SetDefaultFingerThresholds
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDPLUS_SetScanMode

Description:

This function sets scanning speed and resolution. The function can be called at runtime to change the scanning speed and resolution. This function overwrites the user module parameter settings. This function sets scanning speed and resolution and can be called at runtime to change the scanning speed and resolution. This function overwrites the user module parameter settings. Use it when you need to scan sensors that differ in scanning speed and resolution. One example would be regular buttons and a proximity detector. The regular buttons can be scanned with 9-bit resolution. The proximity detector can be scanned less often with 16-bit resolution and a longer scanning time for long-range detection. This function can be used with CSDPLUS_ScanSensor() function.

C Prototype:

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

Assembly:

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

Parameters:

bSpeed – sets the scanning speed; accepted values are listed in the following table.

bResolution – sets the scanning resolution; accepted values are listed in the following table.

Name	Value
Scan Speed Values	
CSDPLUS_ULTRA_FAST_SPEED	0
CSDPLUS_FAST_SPEED	1
CSDPLUS_NORMAL_SPEED	2
CSDPLUS_SLOW_SPEED	3
Resolution Values	

Name	Value
CSDPLUS_9_BIT_RESOLUTION	9
CSDPLUS_10_BIT_RESOLUTION	10
CSDPLUS_11_BIT_RESOLUTION	11
CSDPLUS_12_BIT_RESOLUTION	12
CSDPLUS_13_BIT_RESOLUTION	13
CSDPLUS_14_BIT_RESOLUTION	14
CSDPLUS_15_BIT_RESOLUTION	15
CSDPLUS_16_BIT_RESOLUTION	16

Return Value:

None

Side Effects:

**

CSDPLUS_SetSliderIdac

Description:

This function sets iDAC current for slider elements to the highest for each slider group. This function is available only if the "AutoCalibration" user property is set to "Enabled".

C Prototype:

```
void  CSDPLUS_SetSliderIdac(void)
```

Assembly:

```
lcall CSDPLUS_SetSliderIdac
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDPLUS_SetIdacValue

Description:

This function overwrites the user module parameter settings iDAC Value. Use it if some sensors need to be scanned with different iDAC settings. This function can be used with CSDPLUS_ScanSensor().

C Prototype:

```
void  CSDPLUS_SetIdacValue(BYTE bIdacValue, BYTE bCompensationIDACValue)
```

Assembly:

```
mov   A, bIdacValue
mov   X, bCompensationIDACValue
lcall CSDPLUS_SetIdacValue
```

Parameters:

bIdacValue – sets the iDAC value. Accepted values are in the range 0-127.

bCompensationIDACValue - sets the compensation iDAC value. Valid values are in range 0-127.

Return Value:

None

Side Effects:

**

CSDPLUS_SetPrescaler

Description:

This function overwrites the value of the prescaler user module parameter. Use it if some sensors need to be scanned with the prescaler setting.

C Prototype:

```
void  CSDPLUS_SetPrescaler(BYTE bPrescaler)
```

Assembly:

```
mov   A, bPrescaler
lcall CSDPLUS_SetPrescaler
```

Parameters:

bPrescaler – sets the prescaler value. Accepted values are listed in the following table:

Name	Value	Prescaler
CSDPLUS_PRESCALER_1	0x00	1
CSDPLUS_PRESCALER_2	0x01	2
CSDPLUS_PRESCALER_4	0x02	4
CSDPLUS_PRESCALER_8	0x03	8
CSDPLUS_PRESCALER_16	0x04	16
CSDPLUS_PRESCALER_32	0x05	32
CSDPLUS_PRESCALER_64	0x06	64
CSDPLUS_PRESCALER_128	0x07	128
CSDPLUS_PRESCALER_256	0x08	256

Return Value:

None

Side Effects:

**

CSDPLUS_CalibrateSensors

Description:

This function adjusts iDAC current to obtain raw count near wValue value and stores results in the CSDPLUS_baDAC[] and CSDPLUS_baCompensationDAC[] global arrays. This function is available only if the "AutoCalibration" user property is set to "Enabled".

C Prototype:

```
void  CSDPLUS_CalibrateSensors(WORD wValue)
```

Assembly:

```
mov   A, [wValue+1]
mov   X, [wValue]
lcall CSDPLUS_CalibrateSensors
```

Parameters:

WORD: wValue – the target raw data value.

Return Value:

None

Side Effects:

**

CSDPLUS_ClearSensors

Description:

This function clears all sensors to the nonsampling state by sequentially calling CSDPLUS_wGetPortPin() and CSDPLUS_DisableSensor() for each of the sensors.

C Prototype:

```
void  CSDPLUS_ClearSensors(void)
```

Assembly:

```
lcall CSDPLUS_ClearSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDPLUS_wReadSensor

Description:

This function returns the key Raw scan value in A (LSB) and X (MSB).

C Prototype:

```
WORD  CSDPLUS_wReadSensor(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall CSDPLUS_wReadSensor
```

Parameters:

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

Return Value:

WORD: scan value of sensor, LSB in A and MSB in X.

Side Effects:

**

CSDPLUS_wGetPortPin

Description:

This function returns the port number and pin mask for a given sensor. The passed parameter indexes and selects the data from the CSDPLUS_Sensor_Table[]. The return value can be passed to the CSDPLUS_EnableSensor(), CSDPLUS_DisableSensor().

C Prototype:

```
WORD  CSDPLUS_wGetPortPin(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall CSDPLUS_wGetPortPin
```

Parameters:

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

Return Value:

A => Sensor Bitmap

X => Port Number

Side Effects:

**

CSDPLUS_EnableSensor

Description:

This function configures the selected sensor to measure during the next measurement cycle. Select the port and sensor using the CSDPLUS_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. This also enables the comparator function.

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Bitmap

X => Port Number

Return Value:

None

Side Effects:

**

CSDPLUS_DisableSensor

Description:

This function disables the sensor selected by the CSDPLUS_wGetPortPin() function. The drive mode is changed to Strong (001), effectively grounding the sensor. The connection from the port pin to the AnalogMuxBus is turned off. The function parameters are returned by CSDPLUS_wGetPortPin() function.

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Bitmap

X => Port Number

Return Value:

None

Side Effects:

**

CSDPLUS_GetSnsParasiticCapacitance

Description:

This API returns sensor parasitic capacitance in pF.

C Prototype:

```
BYTE CSDPLUS_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 conveys values to a PC charting tool.

```
//-------------------------------------------------------------------------
// Sample C code for the CSDPLUS User 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;
    CSDPLUS_Start();
CSDPLUS_InitializeBaselines(); //scan all sensors first time, init baseline
    CSDPLUS_SetDefaultFingerThresholds();
//
// Loop Forever
//
while (1)
    {
CSDPLUS_ScanAllSensors(); //scan all sensors in array (buttons and sliders)
       CSDPLUS_UpdateAllBaselines(); //Update all baseline levels;

//detect if any sensor is pressed
        if(CSDPLUS_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 CSDPLUS_waSnsResult[x] <- Raw Counts
```

```
// OUTPUT CSDPLUS_waSnsDiff[x] <- Difference
// OUTPUT CSDPLUS_waSnsBaseline[x] <- Baseline
// OUTPUT CSDPLUS_baSnsOnMask[x] <- Sensor On/Off
}
}
```

Example 2.The following code demonstrates the example of one sensor usage when a couple of sensors configured in the UM Wizard.

```
//-------------------------------------------------------------------------
// Sample C code for the CSDPLUS User Module
//-------------------------------------------------------------------------

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

void main(void)
{
    M8C_EnableGInt;

CSDPLUS_Start(); // Start CSDPLUS UM
    CSDPLUS_SetDefaultFingerThresholds(); // Set default thresholds for button
    // Initialize baseline for sensor number "3"
    CSDPLUS_InitializeSensorBaseline(3);

    while (1)
    {
        // Scan continuously sensor number "3" which is connected
CSDPLUS_ScanSensor(3);
        CSDPLUS_UpdateSensorBaseline(3); //Update Baseline for sensor 3
        if(CSDPLUS_bIsSensorActive(3)) // check if sensor 3 is touched
        {
            // Add user code here to proceed the buttons pressing
        }
    }
}
```

Configuration Registers

The CSDPLUS User Module uses the Timer1, CapSense, Comparator and Capsense Shield Signal PSoC blocks. The CSDPLUS User Module uses the Timer1, CapSense, and Comparator PSoC blocks. Each one is personalized and parameterized through a set of registers. This section gives brief descriptions of those sets. Symbolic names for these registers are defined in the user module instance's C and assembly language interface files (the ".h" and ".inc" files).

Timer1 Block Registers

- Bank 0
  - Timer1 Configuration Register: PT1_CFG

    The Programmable Timer Configuration Register (PT1_CFG) configures the PSoC's programmable timer.

- Timer1 Data Register 0: PT1_DATA0

  The Programmable Timer Data Register 0 (PT1_DATA0) holds the lower 8 bits of the programmable timer's count value.

- Timer1 Data Register 1: PT1_DATA1

  The Programmable Timer Data Register 1 (PT1_DATA1) holds the 8 bits of the programmable timer's count value for the device.

CapSense Block Registers

- Bank 0
  - CapSense Control Register 0: CS_CR0

    The CapSense Control Register 0 (CS_CR0) controls the operation of the CapSense counters.

  - CapSense Control Register 1: CS_CR1

    The CapSense Control Register 1 (CS_CR1) contains additional CapSense system control options.

  - CapSense Control Register 2: CS_CR2

    The CapSense Control Register 2 (CS_CR2) contains additional CapSense system control options.

  - CapSense Control Register 3: CS_CR3

    The CapSense Control Register 3 (CS_CR3) contains control bits primarily for the low-pass filter and reference buffer.

  - CapSense Counter Low Byte Register: CS_CNTL

    The CapSense Counter Low Byte Register (CS_CNTL) contains the current count for the low byte counter.

  - CapSense Counter High Byte Register: CS_CNTH

    The CapSense Counter High Byte Register (CS_CNTH) contains the current count value for the high byte counter.

  - CapSense Status Register: CS_STAT

    The CapSense Status Register (CS_STAT) controls CapSense counter options.

  - CapSense Slew Control Register: CS_SLEW

    The CapSense Slew Control Register (CS_SLEW) enables and controls a fast slewing mode for the relaxation oscillator.

Comparator Block Registers

- Bank 0

- Comparator Control Register 0: CMP_CR0

  The Comparator Control Register 0 (CMP_CR0) enables and configures the input range of the comparators.

- Comparator Control Register 1: CMP_CR1

  The Comparator Control Register 1 (CMP_CR1) configures the comparator output options.

- Comparator Multiplexer Register: CMP_MUX

  The Comparator Multiplexer Register (CMP_MUX) contains control bits for input selection of comparators 0 and 1.

- Comparator LUT Control Register: CMP_LUT

  The Comparator LUT Control Register (CMP_LUT) selects the logic function.

Additional Registers affected by CSDPLUS User Module

- Bank 0
  - Analog mux Configuration Register: AMUX_CFG

    This register configures the integration capacitor pin connections to the analog global bus.

  - Pseudo Ransom Sequence and Prescaler Control Register: PRS_CR

    This register controls the prescaler and pseudo random sequence generator output.

  - Current DAC Data Register: IDAC_D

    This register specifies the 8-bit multiplying factor that determines the output IDAC current.

- Bank 1
  - Output Override to Port 0 Register: OUT_P0

    This register enables specific internal signals to output to Port 0 pins.

  - Output Override to Port 1 Register: OUT_P1

    This register enables specific internal signals to be output to Port 1 pins.

  - Analog Mux Port Bit Enable Register: MUX_CR0

    This register is used to control the connection between the analog mux bus and the corresponding pin.

  - AnalogMuxDAC0 Register: IDAC_0_REG

    This register is used to to set the IDAC0 code.

  - AnalogMuxDAC1 Register: IDAC_1_REG

    This register specifies the 8-bit multiplying factor that determines the output IDAC current.

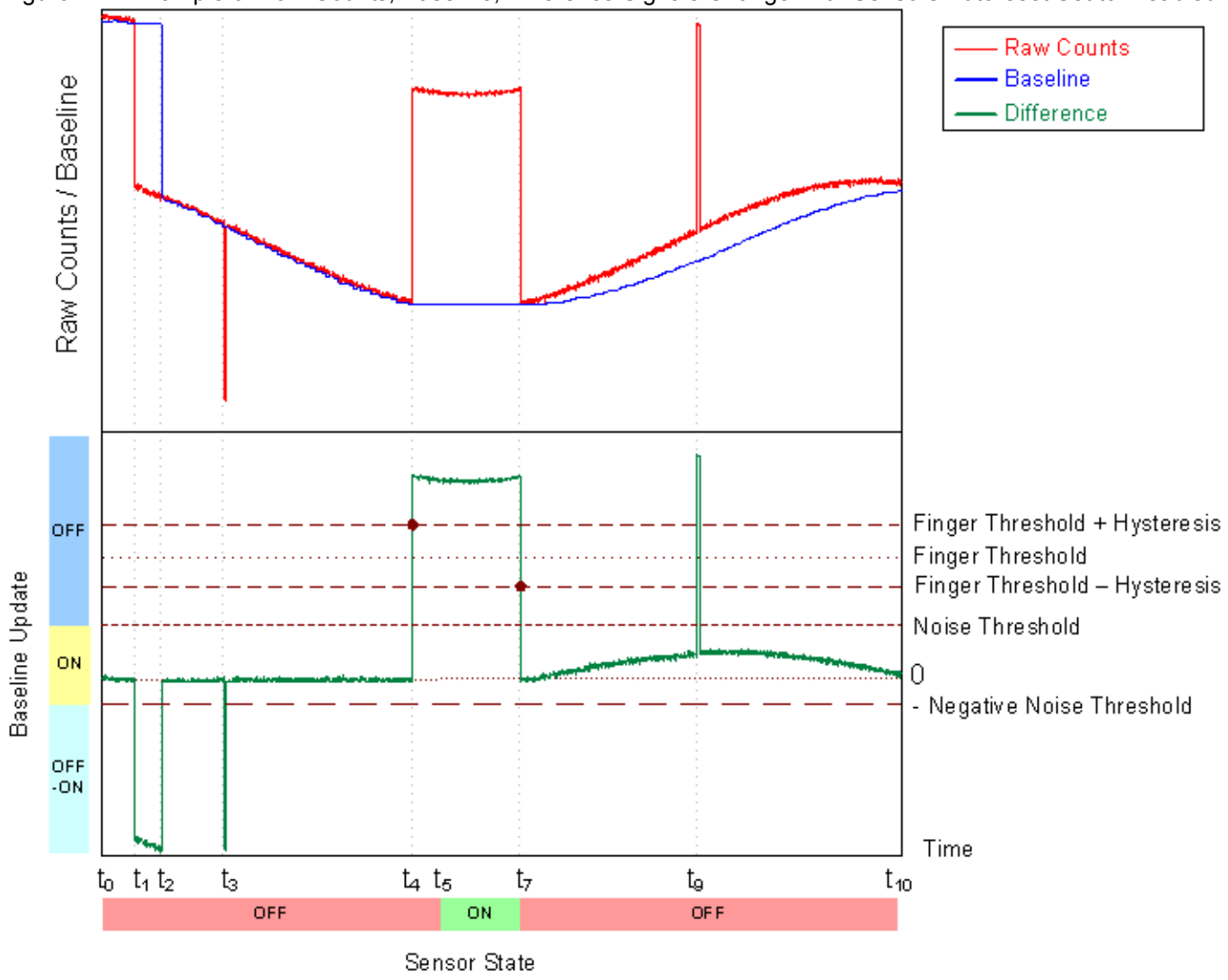
Appendices

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

Interaction of CSDPLUS Parameters

Figure 14 and Figure 15 illustrate the baseline update and decision logic operation and can help you better understand how to set user module parameters for optimum performance. Figure 14 illustrates system operation when the Sensors Autoreset parameter is set to Disabled. Figure 15 illustrates the Sensors Autoreset parameter Enabled. The Finger Threshold, Noise Threshold, Hysteresis, and Negative Noise Threshold are shown together with Difference Signals (Raw Count – Baseline). Data was collected during some artificial tests that demonstrate system operation at both slow and rapid raw count 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 21. Example of Raw Counts, Baseline, Difference Signals Change With Sensors Autoreset Set to Disabled



At t_0, the raw counts that are close to the baseline level 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.

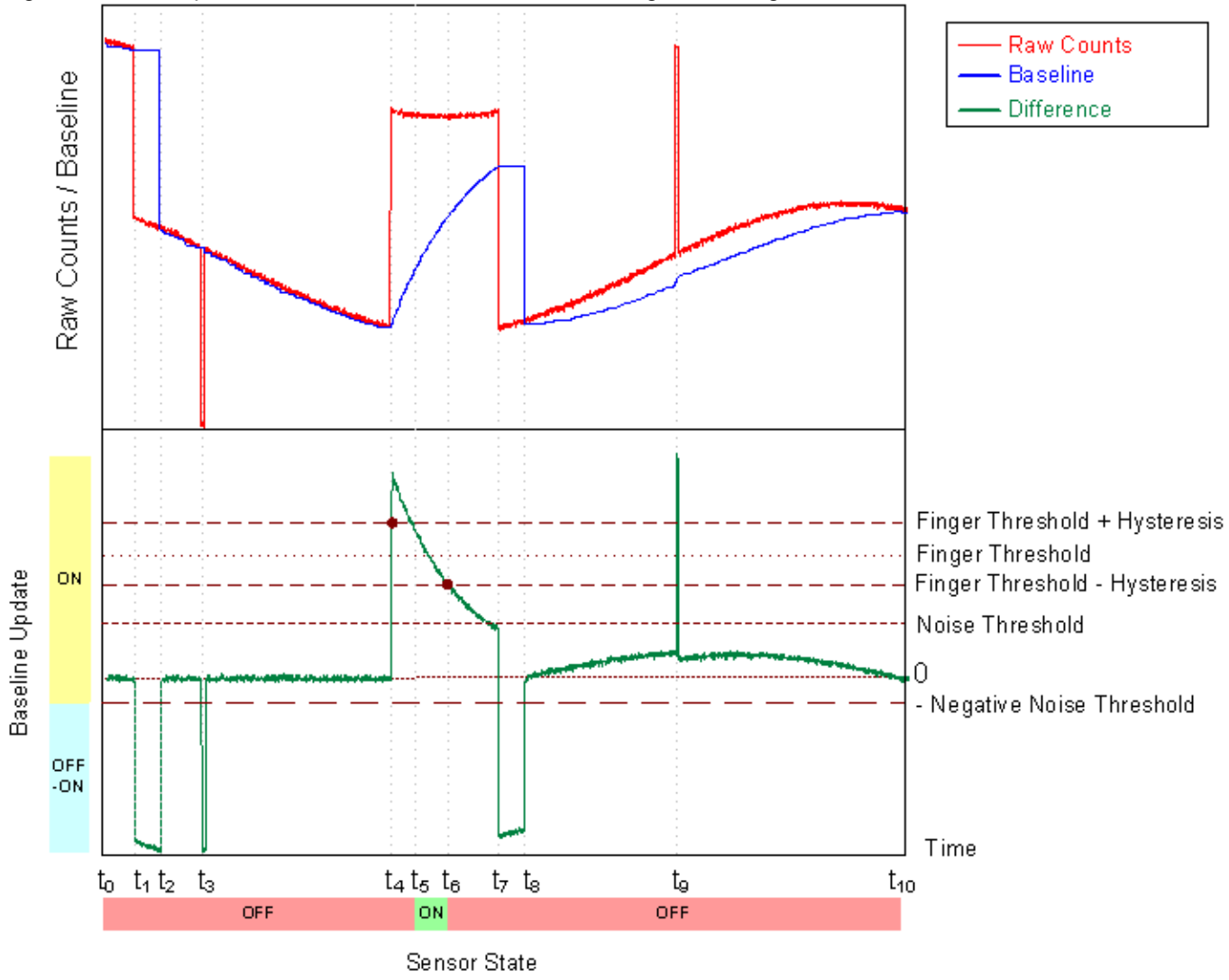
At t_1, the raw count drops sharply and the negative difference exceeds the NegativeNoiseThreshold. This situation can happen if the device is powered on when a finger is on the sensor and then the finger is removed after some time. At this time, the baseline update mechanism is frozen, and an internal 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, for example, by an ESD event. 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 t4. 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 t9 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 (sensors Autoreset 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 provide faster baseline update speeds.

Figure 22. Example of Raw Counts, Baseline, Difference Signals Change With Sensors Autoreset Set to Enabled



The system operation in Figure 15 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 RAM and ROM usage in the user module datasheet. 2. Exposed the following user module variables to application code: CSD_bNoiseThreshold, CSD_bNegativeNoiseThreshold, CSD_bBaselineUpdateThreshold, CSD_bHysteresis, CSD_bDebounce, and CSD_bLowBaselineReset”.
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. Updated ScanSensor() function to reset PRS. 5. Corrected the wGetCentroidPos function to return a valid value when the sensor is not touched. 6. Fixed memory paging problem in the GetSnsParasiticCapacitance function.

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.