Preliminary

CapSense® Sigma-Delta Datasheet CSDe V 2.10

Copyright © 2011-2013 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks				API Memory (Bytes)		Pins per senor
	CapSense®	I²C/SPI	Timer	Comparator	Flash	RAM	
CY8C20xx6L							
User Module	1	–	1	1	1344	39	1
Slider APIs	–	–	–	–	584	79	0
Each Sensor	–	–	–	–	5	13	1

Features and Overview

- Implements CapSense® capacitive sensing in the CY8C20xx6L family of PSoC® devices using the sigma-delta data conversion
- Immune to GPIO current transience, V_{DD} fluctuation, entry and exit from sleep, and IDAC RTS noise
- Configurable system parameters enable tuning so that performance is optimized in a broad range of applications
- Supports as many as 36 capacitive sensors and six sliders
- Capable of detecting 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
- Supports capacitive sensors configured as independent buttons and as dependent arrays to form sliders
- Effective numbers of slider elements can double the number of dedicated I/O pins using the diplexing technique
- Supports slider resolution greater than the physical pitch through interpolation
- Gives shield electrode for reliable operation with high parasitic capacitance and in the presence of a water film
- Guided sensor and pin assignments using the CSDe Wizard
- External components and PCB layout requirements identical to CSD, except in the 48-QFN package where the P4[2] pin must be connected to an isolated (ungrounded) pad on the PCB
- No hardware tuning (manual or automated) across the effective 5–45 pF Cp range
- Constant 400 us integration time for each sensor regardless of Cp
- The CSDe User Module works with the SmartLED User Module

The CSDe User Module implements a CSD based on the differential capacitive sensing method.

Quick Start

1. Select and place user modules that require dedicated pins (for example, I^2C and LCD), if used. Assign ports and pins as required.
2. Select and place the CSDe User Module.
3. Right-click the CSDe User Module in the Workspace Explorer to access the CSDe Wizard (the wizard is explained in a later section in this user module datasheet).
4. Set the required number of sensors, sliders, and rotary 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 the external modulation capacitor will be connected to in the CSDe Wizard.
8. If required, enter the pin that will be used to shield the sensor 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, or a touch-pad.
11. Program the PSoC device on the target board with the hex generated by PSoC Designer™.

Introduction

CapSense is a human interface technology that operates by detecting 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 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. The major benefits of CapSense include:

- Cleaner, more aesthetically pleasing designs.
- Reduces form factor for smaller end products.
- Advanced user interface features, such as LED effects and proximity sensing.
- Improves reliability with components that do not wear or have a finite cycle life.
- Improves spill resistance due to lack of mechanical interface penetrations.
- Reduces tooling cost by eliminating penetrations or other mechanical features needed for mechanical input devices.

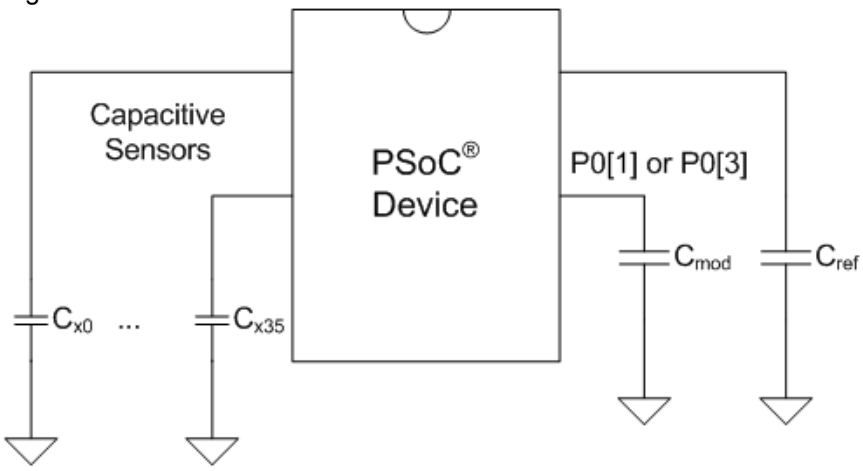
CSDe uses a sigma-delta capacitance to digital code conversion circuit, key attributes of which include low EMC emission and superior immunity against EMI.

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

PCB Level

Figure 1 shows a schematic of CSDe. 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 CY8C20xx6A/H CapSense and Getting Started with CapSense design guides for complete information on PCB level CapSense implementation.

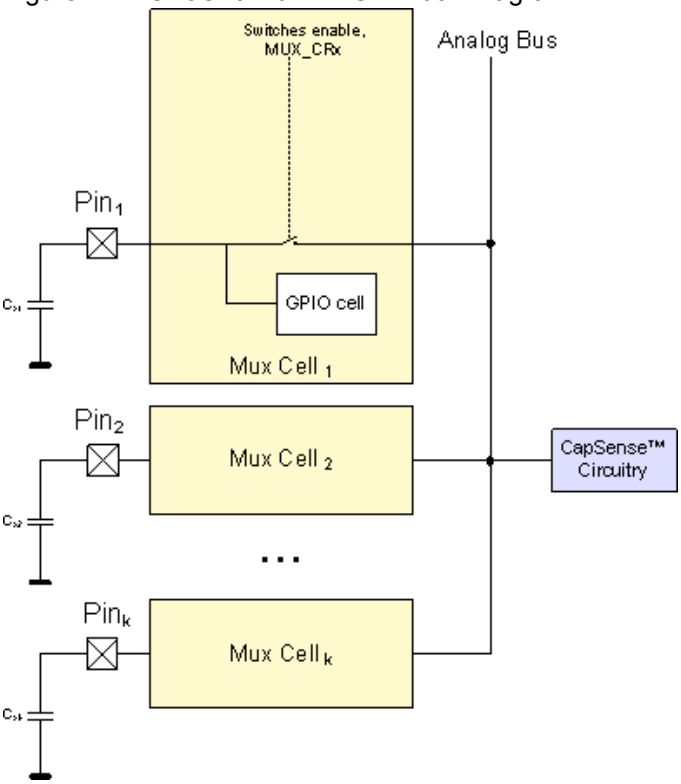
Figure 1. CSDe Schematic



IC Level

CY8C20xx6L devices have an analog mux bus that enables connecting capacitive sensing analog circuitry to any PSoC pin. The CSDe User Module connects the active and reference 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 represented in Figure 2.

Figure 2. CY8C20xx6L AMUX Block Diagram



Software

The following are attributes of the CSDe software component:

- Tunable system configuration parameters allow tuning to optimize performance in a broad range of applications.
- The raw count value from the capacitance converter circuitry is analyzed in runtime by API functions to make sensor state decisions and to compensate for environmental changes.
- In the case of consecutive, dependent sensors (for example, sliders and touchpads) API functions are provided 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 to reduce by half the number of I/Os consumed for a specific number of slider elements.

Recommended Reading

The following documents are recommended reading before implementing a CapSense design using the CSDe User Module:

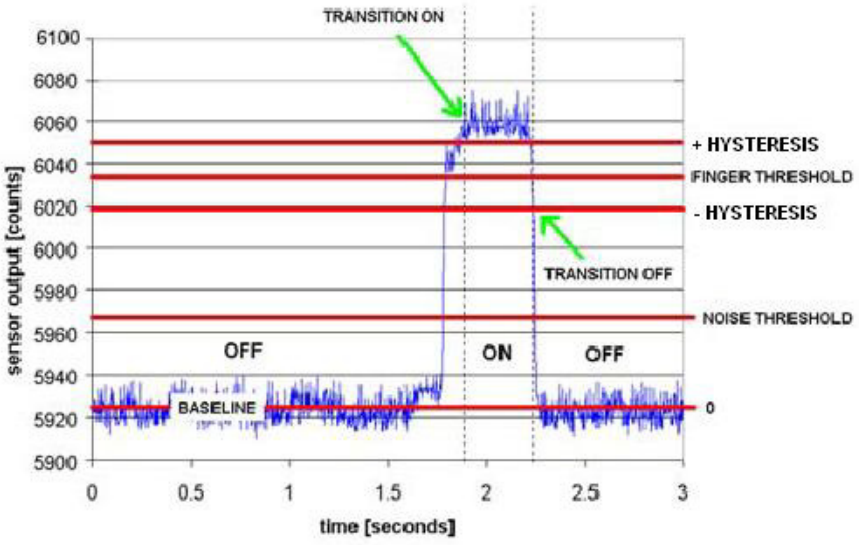
- Getting Started with CapSense
- CY8C20xx6A/H CapSense Design Guide
- CY8C21x34/B CapSense Design Guide
- CY8C20x34 CapSense Design Guide
- CY8CMBR2044 CapSense Design Guide

Capacitance Sensing Implementation

Buttons

CapSense buttons are analogous to mechanical push-buttons. They are used for discreet controls such as on/off switches, function keys, menu keys, and so on. API functions monitor the capacitance signals (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.

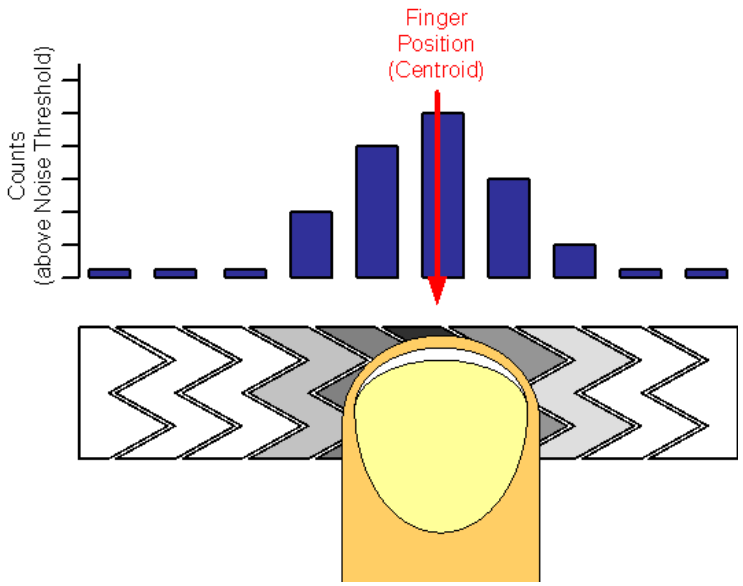
Figure 3. Capacitance Signal from a Sensor as it is Activated



Sliders

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

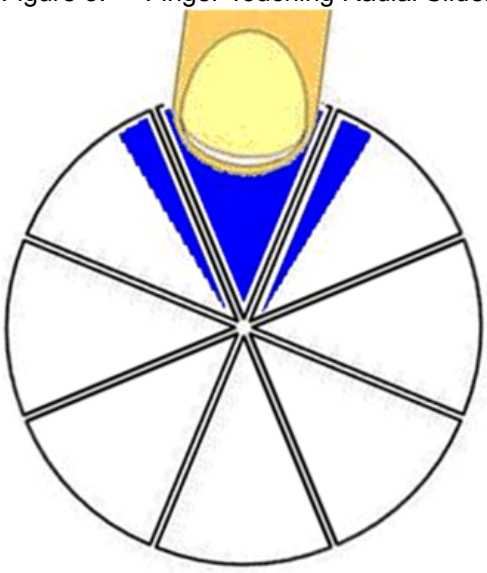
Figure 4. Interpolated Centroid Position of a Finger on a Slider



Radial Sliders

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

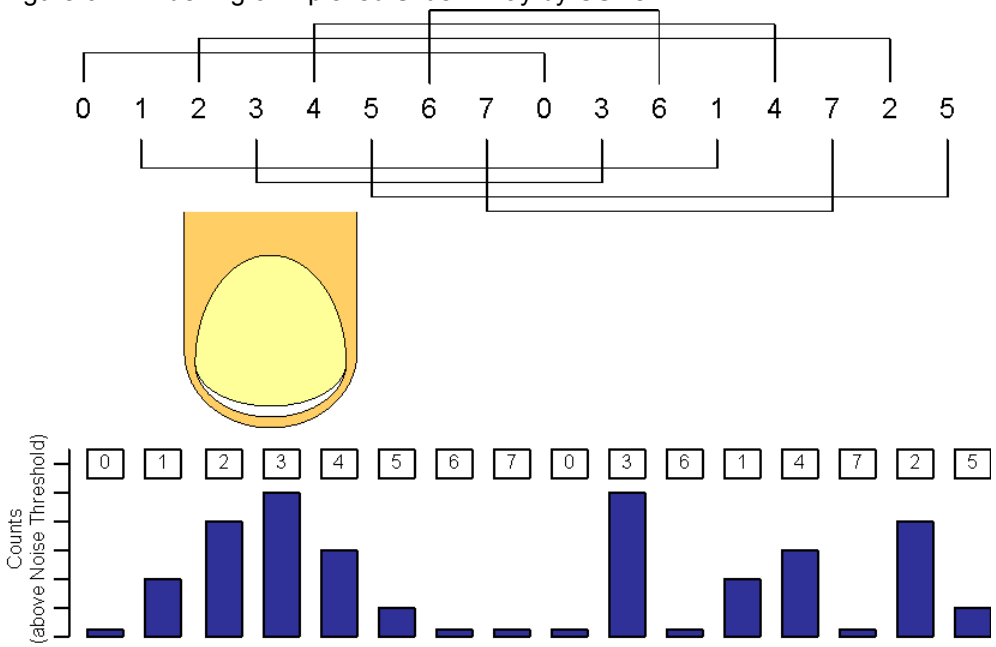
Figure 5. Finger Touching Radial Slider



Diplexing

When diplexing is used, each pin on the PSoC that is designated as a slider element is mapped to two physical locations in the array of slider sensors. The first (or numerically lower) half of the physical locations is mapped according to the port pin assigned in the CSDe 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 CSDe



The close proximity of strong signals in the lower half of the slider results in the same levels aliased to 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.

Ensure that 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 should also be reasonably well-matched (within 10 pF). The diplex sensor index table is automatically generated by the CSDe Wizard when you select diplexing. Table 1 shows the diplexing sequences for up to 56 slider segments diplexed into 28 PSoC I/O pins.

Table 1. Diplexing Sequence for Different Slider Segment Counts

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

Total Slider Segment Count	Segment Sequence
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})

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

Driven Shield Electrode

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

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

Figure 7. Driven Shielding Electrode PCB Layout

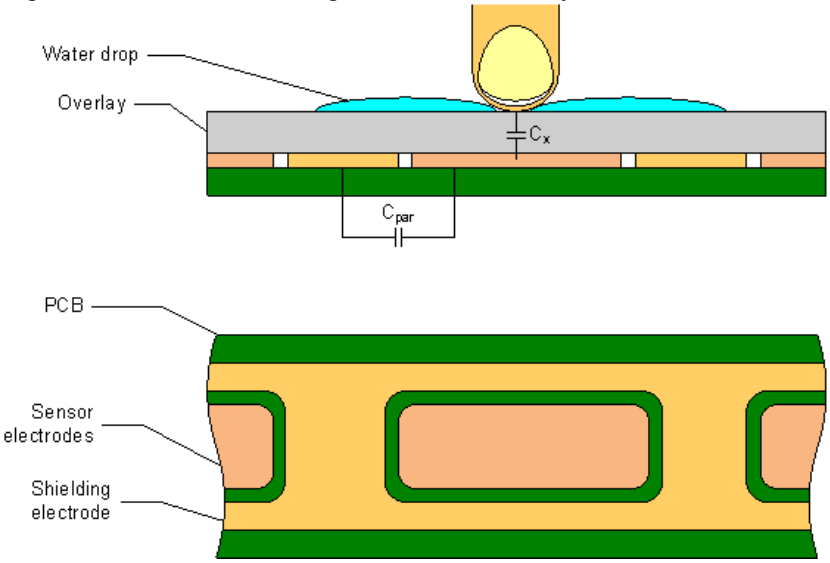


Figure 7 illustrates a driven shielding electrode for a button. As an alternative, the shielding electrode can be located on the opposite PCB layer, including the plane under the button. A hatch pattern is recommended in this case, with a fill ratio of about 30 to 40%. No additional ground plane is required.

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

Power Supply Requirement

Table 2. CSDe Power Supply Requirement

Parameter	Min	Typ	Max	Unit	Test Conditions and Comments
V_{DD}	1.8[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.

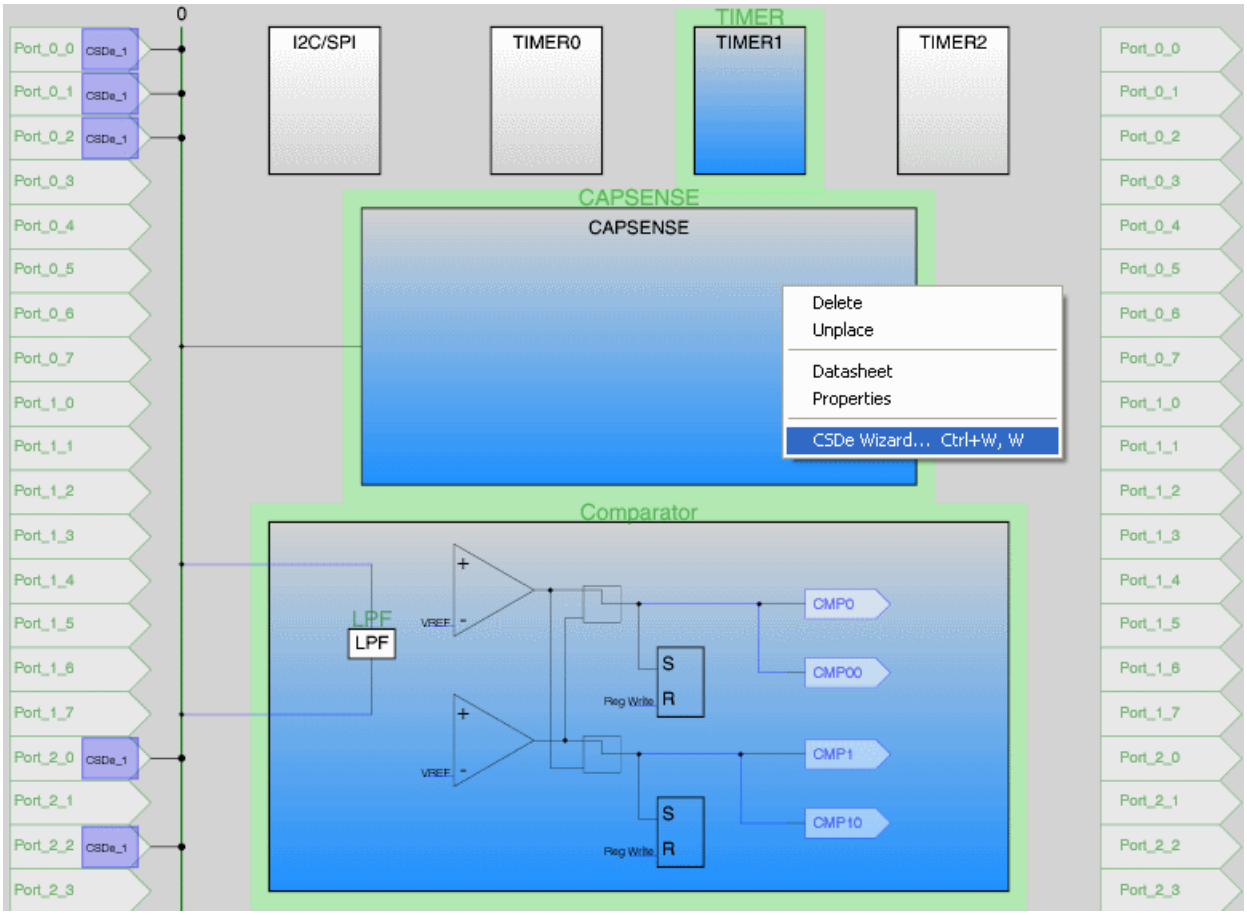
a. 1.8 V is an absolute minimum VDD spec. Allowing V_{DD} to droop below 1.8 V can introduce excessive noise.

Placement

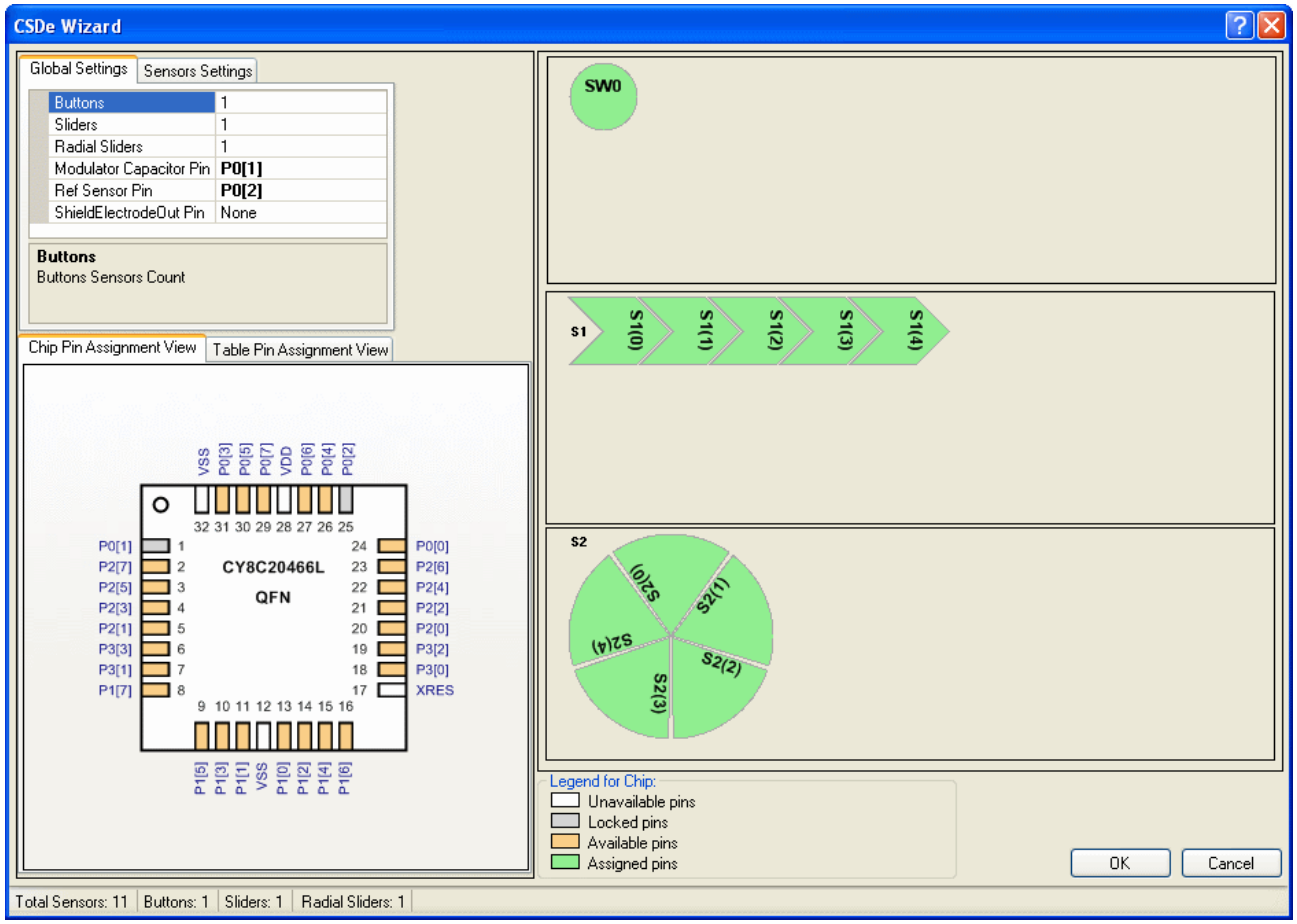
The Timer1, CapSense, and Comparator blocks are assigned to CSDe when the user module is instantiated. Alternate placements are not available. User modules that require dedicated pin resources, including the LCD and I2CHW, must be placed before starting the CSDe Wizard. This ensures that the dedicated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

CSDe Wizard

1. To access the CSDe Wizard, right-click on any user module block in the Chip Editor and select the CSDe Wizard.



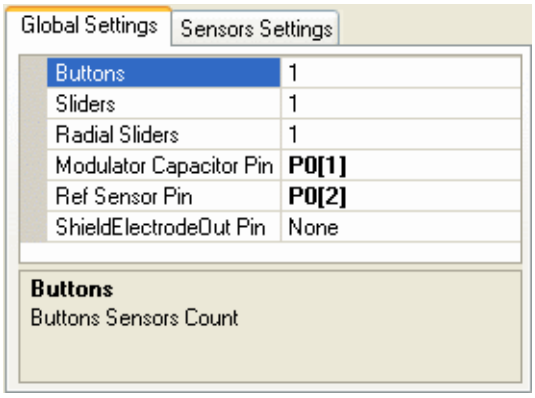
2. The Wizard opens showing the numeric entry boxes for the number of sensors, slider sensors, and radial sliders.



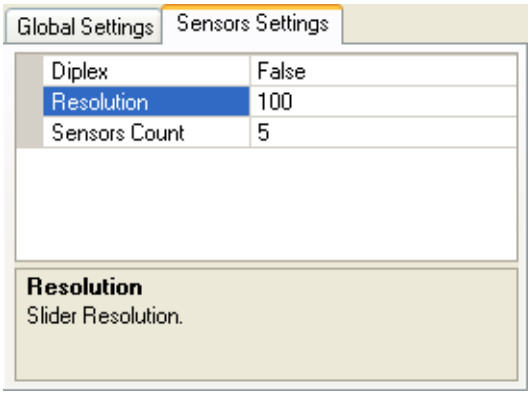
Wizard Pin Legend

- White – The pin cannot be used as a CapSense input.
- Gray – The pin is locked. There are two possible causes for this. 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 been changed from its default.To return the pin name to its default, expand the pin from the ‘Select’ menu in the Pinout view, and 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.

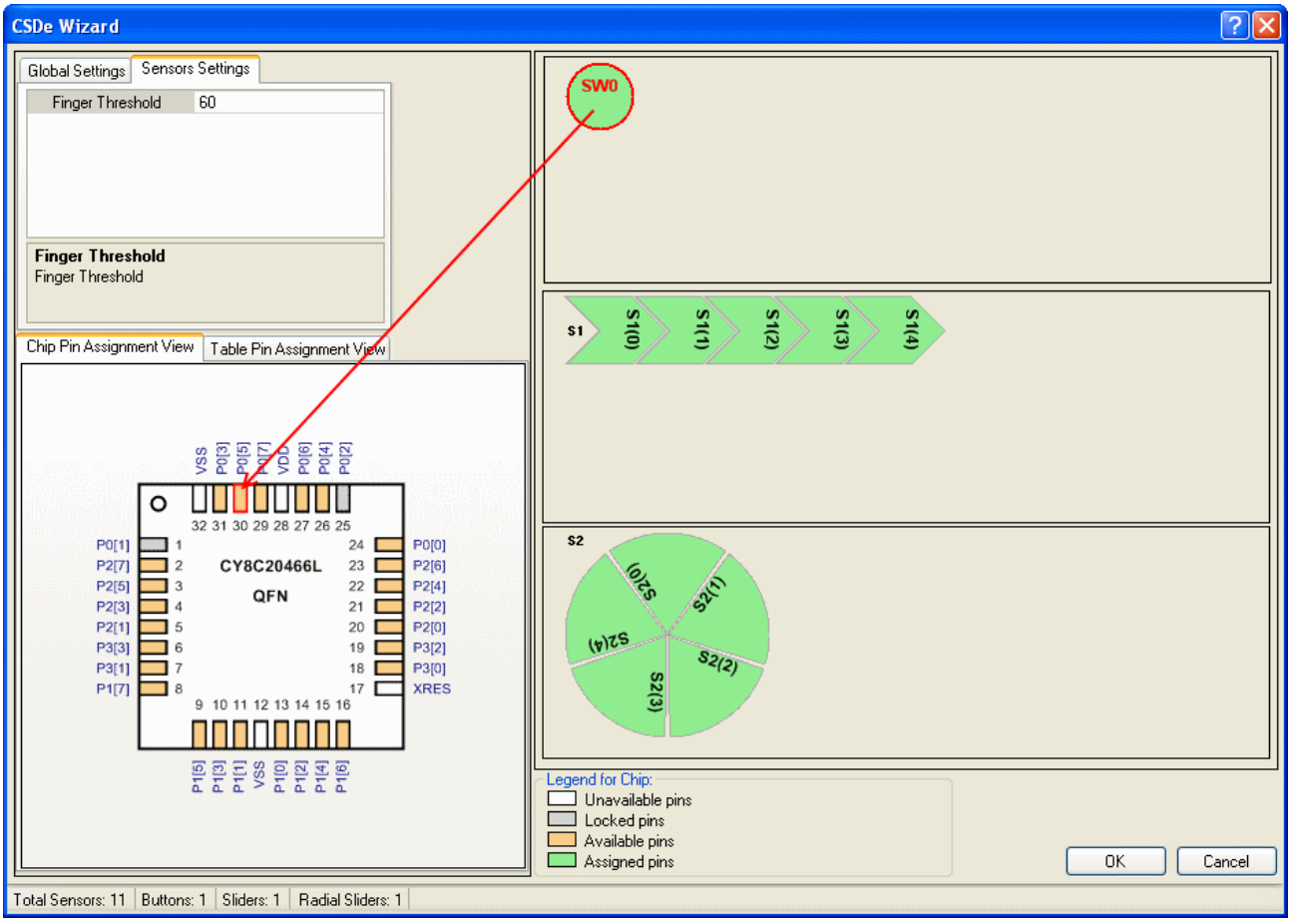
3. Select the Global Settings tab to type the number of independent buttons, sliders, and radial sliders. The total number of sensors (buttons and slider elements) is limited to the number of pins available. After entering the data, press the [Enter] key to update the display with the new value.
4. Select the modulator capacitor pin (C_{mod}). Choose from either P0[1] or P0[3].
5. Select the reference sensor pin (C_{ref}). Choose one from the drop-down list.
6. Select the shield electrode output pin, if desired. Choose from either P0[7] or P1[2].



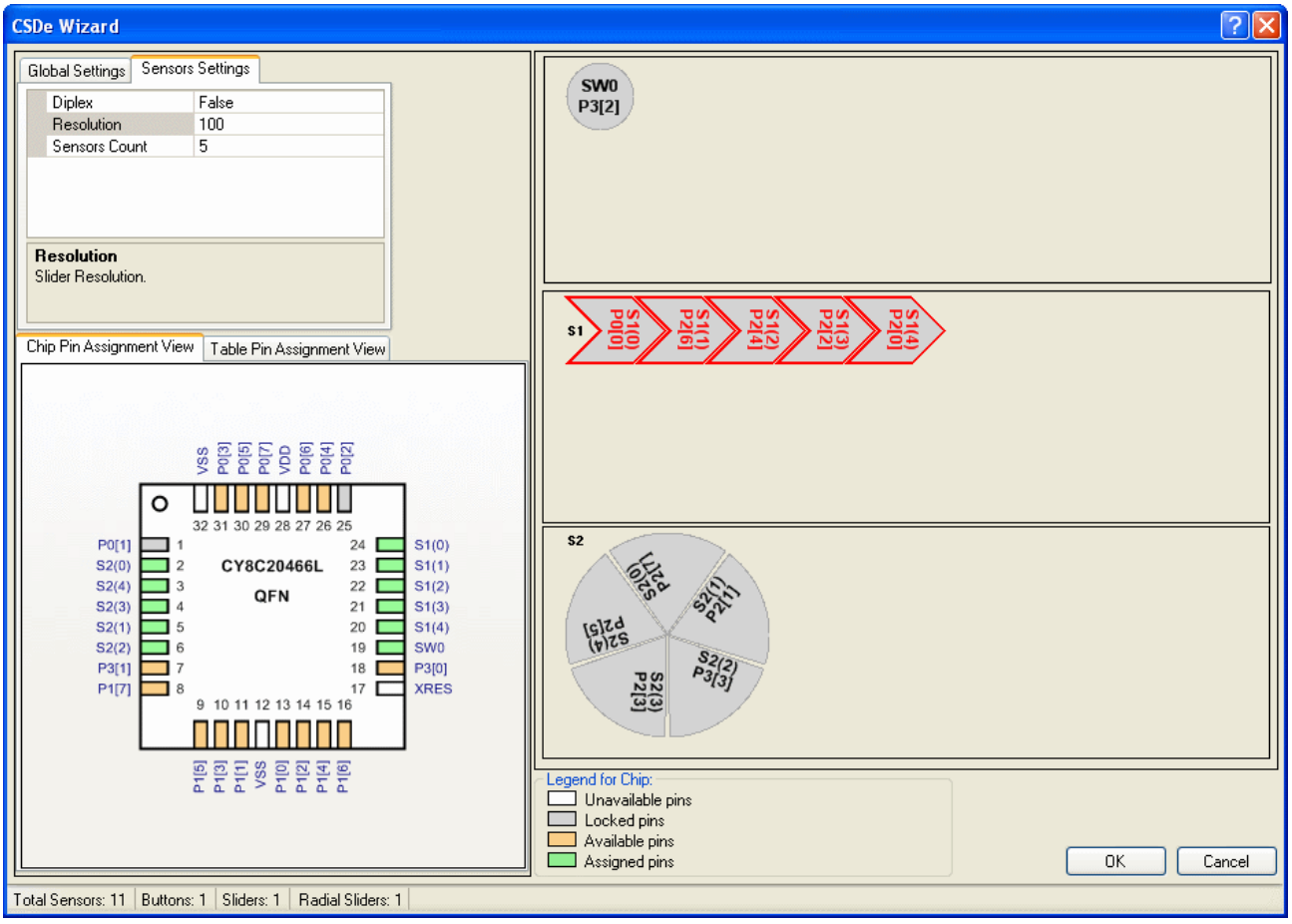
7. Select the Sensors Settings tab to set the settings for buttons, sliders, and radial sliders. To alter settings, click one of your sliders to activate it. Type the number of sensor elements in each slider. The practical minimum number of sensors in a slider is five, the maximum is limited only by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



8. Type the output resolution. The minimum value is five. CSDe 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.
9. 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.



10. Assign sensors to pins by dragging the sensor to the pin in the Pin Assignment View. You can choose to drag sensors to pins in the Chip Pin Assignment View or the Table Pin Assignment View. The I/O pin is green after selection and is no longer available. Change sensor assignments by dragging the port pin back to the uncommitted table. Avoid selecting pins already committed to other user modules.
11. Repeat for the remaining sensors. Click OK to accept data and return to PSoC Designer. Sensor placement is now complete. Right-click in the Device Editor window and select Refresh to update the pin connections.



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

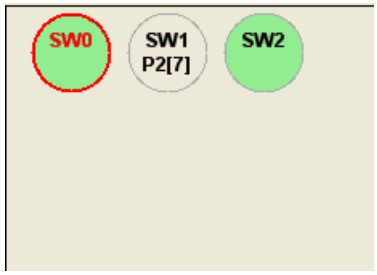
Sensors Representation Section

This section graphically represents all sensor types available in a project. This section is used to drag-and-drop sensors to the Chip or 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 the following screenshot. 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

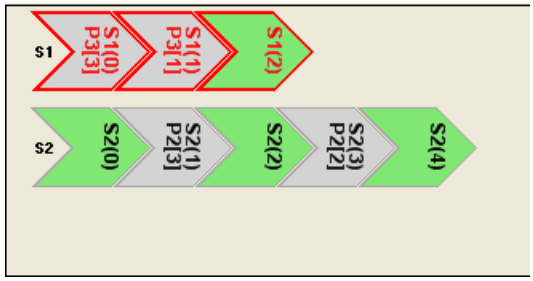
the button title if a button is already assigned. The button widget is set in a red frame if the button sensor is selected.

Figure 8. Buttons Representation



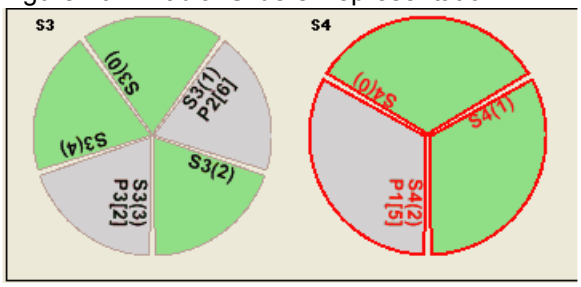
- Sliders Representation – The sliders are displayed as a sequence of sensor segments as shown in the following screenshot. The specific sensor name is displayed for each segment in a slider. All sensors in a slider support the drag-and-drop capability for the Chip and Table Pin Assignment Views. The 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 9. Sliders Representation



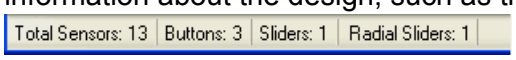
- Radial Sliders Representation – The radial sliders are displayed as a pie tile of segments as shown in the following screenshot. The sensor name is displayed for each segment in a radial slider. All sensors in a radial slider support the drag-and-drop capability for the Chip and Table Pin Assignment 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 10. 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
- 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 CSDe User Module 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 parameters and closes, or it shows an appropriate warning message, does not save parameters, and stays opened.
2. “Cancel” – this button closes the wizard without saving any parameters.
3. “Close” – 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 then the wizard closes without saving any parameters.
4. “Help” – this button calls the help page giving a reference information on how to use the CSDe User Module wizard. It briefly describes the CSDe User Module wizard features. The Help is available through 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.

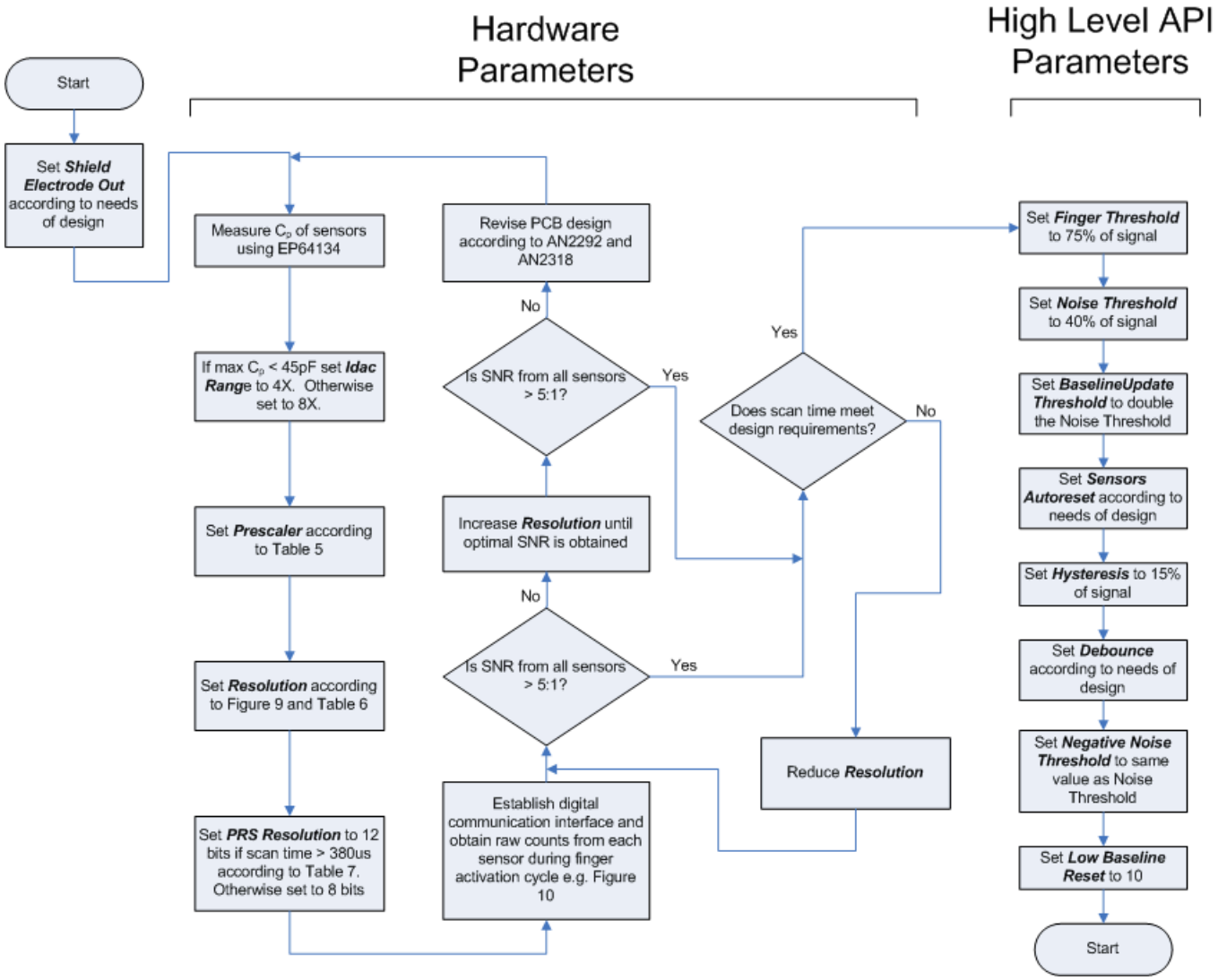
User Module Parameters - Tuning Guide

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

Figure 8 is a flowchart showing the tuning process for CSDe UM parameters. CSDe UM parameters can be separated into two broad categories, namely Hardware parameters and High Level API parameters. The parameters in these categories affect the behavior of the capacitive sensing system in different ways and are therefore treated separately in this section. There is, however, a complementary relationship between the sensitivity of each sensor as determined by the Hardware parameter settings and many of the High Level API parameter settings. The designer must remember this when any hardware parameter

is changed to ensure that the corresponding High Level API parameters are adjusted accordingly. Tuning of CSDe User Module Parameters should always begin with the Hardware Parameter.

Figure 11. CSDe User Module Parameters Tuning Flowchart



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 provides guidance on how each should be tuned based on system characteristics or other parameters.

By default, hardware parameters are global settings that apply to all CapSense sensors in a design. In designs where total parasitic capacitance of each sensor (C_p) and/or sensor sensitivity vary over a wide range, there may not be global Hardware parameter settings that are suitable for all sensors. In such cases, the SetPrescaler(i), and SetScanResolution(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 an example of this.

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 CSDe design. Prescaler depends on the selected Precharge Source, IMO, and the C_p of the sensor/sensors being scanned. Recommended Prescaler settings based on these parameters are provided in Table 3. The default setting is 2.

Table 3. Prescaler Setting 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 raises 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 6 provides recommended Resolution settings based on C_p and the finger capacitance C_f. C_f 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 is 12.

Figure 12. Finger Capacitance (C_f) Based on Overlay Thickness and Circular Sensor Diameter

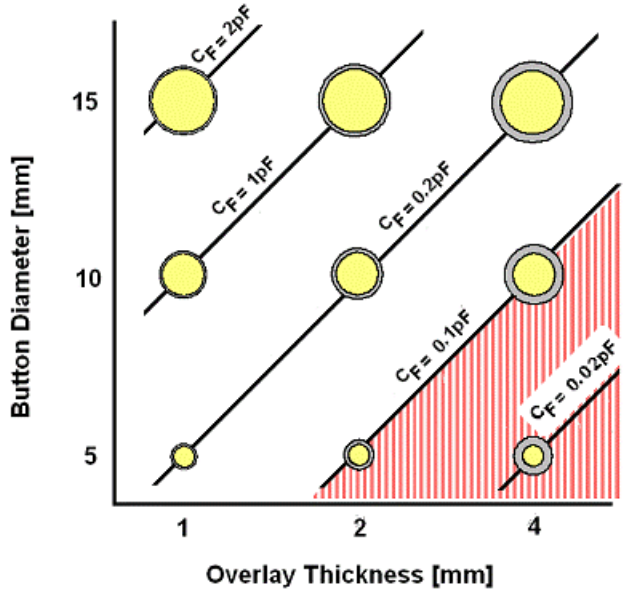


Table 4. Resolution Setting Based on Finger Capacitance and Cp

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

PRS Resolution

This parameter changes the PRS sequence length. Possible values are 8 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.

Filter Coefficient

This parameter sets the IIR filter coefficients. Possible values are 2, 4, 8, 16, 32, and 64. The default setting is 2.

High-Level API Parameters

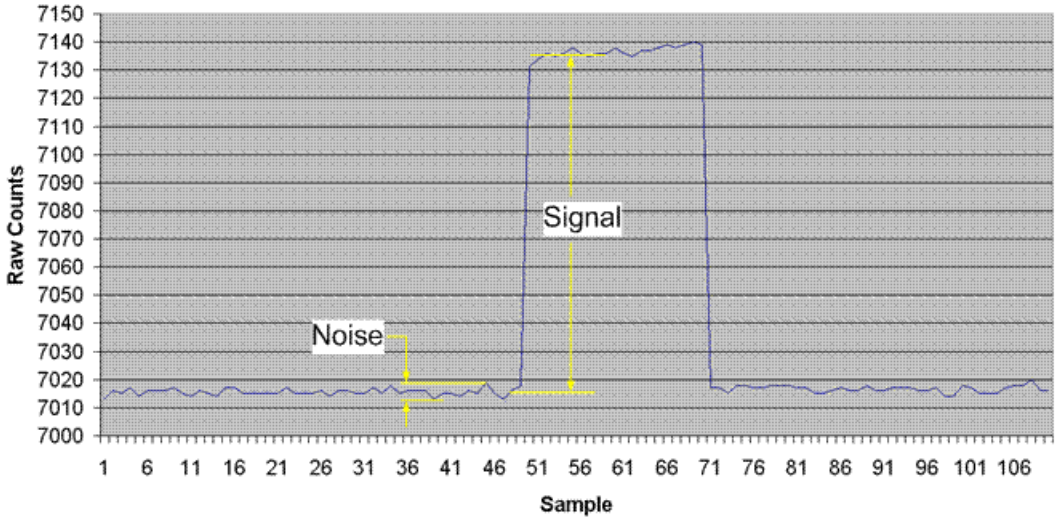
High Level API parameters determine the behavior of high level firmware algorithms that discriminate between sensor activations and noise, and compensate for signal drift caused by environmental conditions. To determine the proper values for these parameters, a digital communication interface must

be established with the system to monitor the raw counts, and the baseline and difference counts during a finger activation event for each sensor. This data is stored in arrays named CSDe_waSnsBaseline[], CSDe_waRawCount[], CSDe_wRefSnsResult[], CSDe_waSnsResult[], and CSDe_waSnsDiff[] respectively. CSDe_waRawCount - stores the real raw counts which do not take into account the value of the reference sensor. CSDe_waSnsResult - stores the difference between the real raw counts of sensors and the raw counts of the reference sensor.

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. As a result, the data used as the basis for setting these parameters must be taken with the system in its final assembled state and in 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 that is, 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. According to the CY8C20xx6A/H CapSense design guide, the ratio of signal to noise (SNR) must be at least 5:1 for robust operation of the CapSense system. If SNR is less than 5:1 the Hardware Parameters must be adjusted and/or the PCB layout changed according to the guidelines of the Getting Started with CapSense design guide to raise the SNR to at least 5:1.

Figure 13. Typical Raw Counts from a Sensor During Finger Activation Cycle



Noise Threshold

Positive changes in raw counts below this level are accumulated for the purpose of updating 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

CSDe uses a "bucket" method to update the baseline count in the CSDe_UpdateSensorBaseline() API function. Half the difference between the raw count value and the baseline count is added to the "bucket" when:

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.

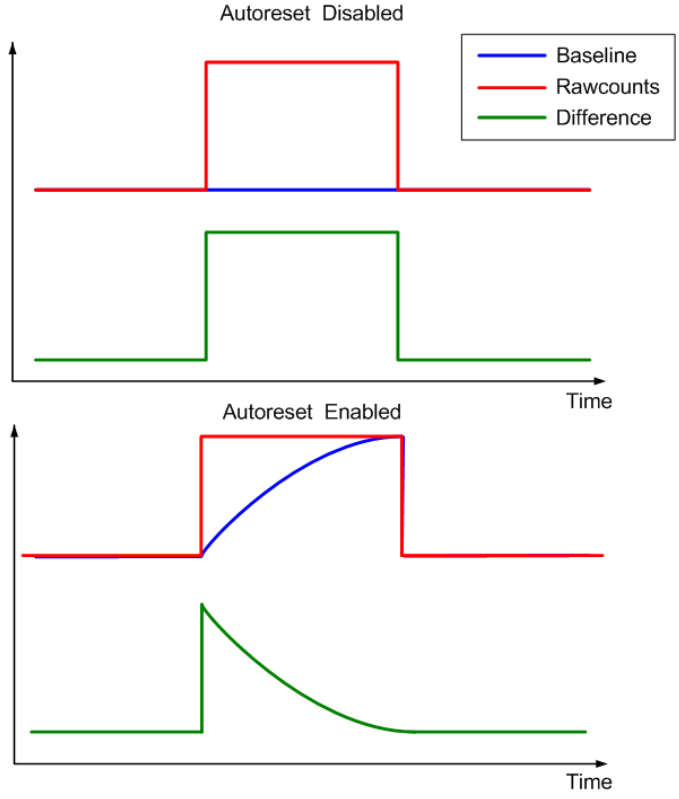
   The BaselineUpdateThreshold sets the value that the "bucket" must reach for the baseline to be incremented. A good starting point for this parameter is double the Noise Threshold. Possible values are 0 to 255; the default value is 100.

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 on due to sudden rises in raw counts that are not caused by a touch. Such sudden rises can be caused by a large power supply voltage fluctuation, a high energy RF noise source, or rapid temperature change.

When Sensors Autoreset is Disabled, 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 additional explanation of this parameter.

Figure 14. Affect of the Sensor Autoreset Parameter on Baseline

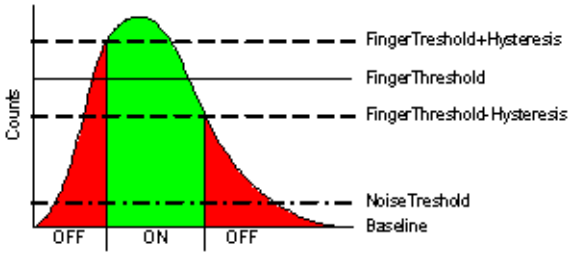


Hysteresis

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 15. Relationship Between Finger Threshold and Hysteresis on Button Sensor State



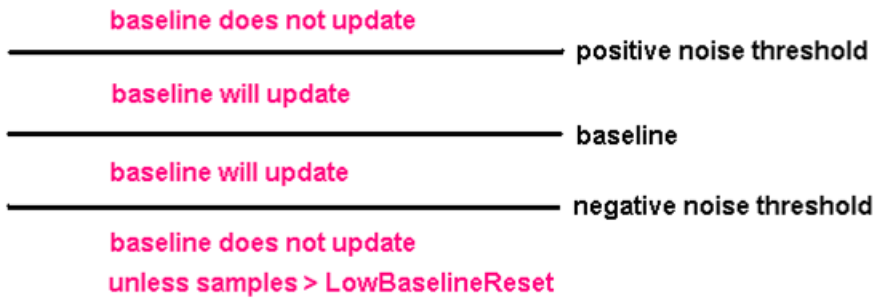
Debounce

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 plus hysteresis for the number of samples specified by this parameter. The Debounce counter is incremented by the CSDe_bIsSensorActive or CSDe_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. It establishes a level relative to the current baseline line above which the baseline resets, that is, snaps down to the raw count value. If raw counts are below this level, the baseline does 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 16. Relationship Between Noise Thresholds and Baseline Reset



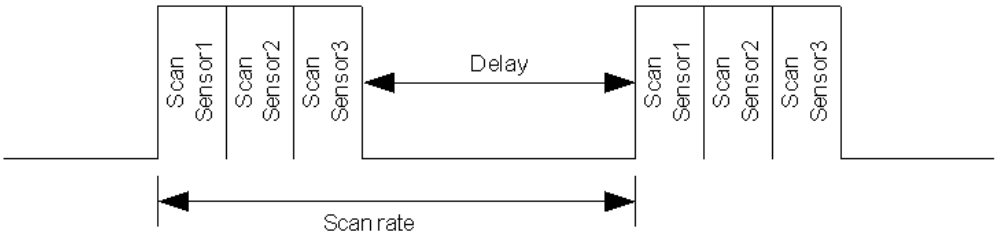
Low Baseline Reset

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.

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 17. 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 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.

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 CSDe_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 CSDe for simplicity.

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

Entry Points are supplied to initialize the CSDe, start it sampling, and stop the CSDe. In all cases the instance name of the module replaces the CSDe 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. You should 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:

- CSDe_waRawCount[]
- CSDe_waSnsResult[]
- CSDe_waSnsBaseline[]

- CSDe_waSnsDiff[]
- CSDe_baSnsOnMask[]
- CSDe_baBtnFThreshold[]

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

CSDe_waSnsResult[] – This is an integer array that contains the raw data of each sensor. The raw count of the reference sensor scan is subtracted from this value to give the final scan result. The array size is equal to the sensor count. The CSDe_waSnsResult[] data is updated by these functions:

- CSDe_ScanSensor()
- CSDe_ScanAllSensors()

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

- CSDe_UpdateAllBaselines();
- CSDe_UpdateSensorBaseline();
- CSDe_InitializeBaselines().

CSDe_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.

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

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

CSDe_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 CSDe_CalibrateSensor() and used by CSDe_ScanSensor(). The array size is equal to the sensor count.

The following group of variables is used by the user module API:

- **CSDe_wRefSnsResult** - stores the scan result of the reference sensor;
- **CSDe_fCSDeFlags** - byte flag that indicates the CSDe states:

Name	Value	Description
CSDe_CALIBRATION_FLAG	0x01	Used to suppress subtraction of the reference sensor raw counts from that of the actual sensor and application of the IIR filter in CSDe_ScanSensor(). This is necessary during calibration to determine an appropriate IDAC setting for each actual sensor.
CSDe_INITIALIZATION_FLAG	0x02	Used to suppress subtraction of the reference sensor raw counts from that of the actual sensor and application of the IIR filter in CSDe_ScanSensor(). This is necessary during initialization to determine an appropriate IDAC setting for each actual sensor.

- **CSDe_bFilterCoef1** - byte variable used to store the IIR filter multiplier. This variable is initialized in CSDe_Start() and accessed in CSDe_ScanSensor(). The variable is initialized depending on the “Filter Coefficient” parameter selection. This first coefficient is calculated by the formula:
  Coef1 = $\log_2$(“Filter Coefficient”), where “Filter Coefficient” is the value that you select in the CSDe User Module parameter window.
- **CSDe_bFilterCoef2** - byte variable used to store the IIR filter divider. This variable is initialized in CSDe_Start() and accessed in CSDe_ScanSensor(). The variable is initialized depending on the “Filter Coefficient” parameter selection. This first coefficient is calculated by the formula:
  Coef2 = (“Filter Coefficient” - 1), where “Filter Coefficient” is the value that you select in the CSDe User Module parameter window.
- **CSDe_bRefDAC** - byte variable used to store the AutoCalibrated IDAC value for the reference sensor. This value is set by CSDe_CalibrateRefSensor() and is used by CSDe_ScanRefSensor() to scan the reference sensor.

CSDe_Start

Description:

This function starts the CSDe User Module, initializes the global variables, configures and connects C_{mod} to the AMUX bus, configures and connects the driven shield (if selected), and configures the CapSense block and the associated hardware.

C Prototype:

```
void  CSDe_Start(void)
```

Assembly:

```
lcall CSDe_Start
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDe_Stop

Description:

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

C Prototype:

```
void  CSDe_Stop(void)
```

Assembly:

```
lcall CSDe_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDe_Resume

Description:

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

C Prototype:

```
void  CSDe_Resume(void)
```

Assembly:

```
lcall CSDe_Resume
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDe_ScanRefSensor

Description:

This function scans the reference sensor. The result of the scan is stored in RAM at CSDe_wRefSnsResult.

C Prototype:

```
void  CSDe_ScanRefSensor(void)
```

Assembly:

```
lcall CSDe_ScanRefSensor
```

Parameters:

None

Return Value:

None

Side Effects

**

CSDe_ScanSensor

Description:

This function scans the actual sensor specified by bSensor. Prior to the actual sensor scan this function calls CSDe_ScanRefSensor() to scan the reference sensor. Then the actual sensor is scanned

and the result from the reference sensor scan is subtracted from the actual sensor scan. The constants used to AutoCalibrate the reference and actual sensors ensure that the reference sensor raw counts are always less than the actual sensor raw counts to prevent a carry out condition. An IIR filter is applied to the difference and the result is stored in RAM at CSDe_waSnsResult(). The coefficients for the IIR filter are defined by the constants CSDe_FILTER_COEF1 and CSDe_FILTER_COEF2.

C Prototype:

```
void  CSDe_ScanSensor(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall CSDe_ScanSensor
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects

**

CSDe_ScanAllSensors

Description:

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

C Prototype:

```
void  CSDe_ScanAllSensors(void)
```

Assembly:

```
lcall CSDe_ScanAllSensors
```

Parameters:

None

Return Value:

None

Side Effects

**

CSDe_ClearSensors

Description:

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

C Prototype:

```
void  CSDe_ClearSensors(void)
```

Assembly:

```
lcall CSDe_ClearSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDe_wReadSensor

Description:

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

C Prototype:

```
WORD  CSDe_wReadSensor(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall CSDe_wReadSensor
```

Parameters:

A => Sensor Number

Return Value:

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

Side Effects:

**

CSDe_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 CSDe_Sensor_Table[]. The return value can be passed to the CSDe_EnableSensor(), CSDe_DisableSensor().

C Prototype:

```
WORD  CSDe_wGetPortPin(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall CSDe_wGetPortPin
```

Parameters:

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

Return Value:

A => Sensor Bitmap

X => Port Number

Side Effects:

**

CSDe_EnableSensor

Description:

This function configures the selected sensor to measure during the next measurement cycle. The port and sensor can be selected using the CSDe_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  CSDe_EnableSensor(BYTE bMask, BYTE bPort)
```

Assembly:

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

Parameters:

A => Sensor Bitmap

X => Port Number

Return Value:

None

Side Effects:

**

CSDe_DisableSensor

Description:

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

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Bitmap

X => Port Number

Return Value:

None

Side Effects:

**

CSDe_SetScanResolution

Description:

This function sets the 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. The function is effective when some sensors need to be scanned with different scanning speed and resolution, for example, 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 longer scanning time for long-range detection.

C Prototype:

```
void  CSDe_SetScanResolution(BYTE bResolution)
```

Assembly:

```
mov   A, bResolution
lcall CSDe_SetScanResolution
```

Parameters:

bResolution - sets the Resolution value.

Return Value:

None

Side Effects:

**

CSDe_SetPrescaler

Description:

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

C Prototype:

```
void  CSDe_SetPrescaler(BYTE bPrescaler)
```

Assembly:

```
mov   A, bPrescaler
lcall CSDe_SetPrescaler
```

Parameters:

bPrescaler - sets the Prescaler value. Accepted values are listed in the following table:

Name	Value	Prescaler
CSDe_PRESCALER_1	0x00	1
CSDe_PRESCALER_2	0x01	2
CSDe_PRESCALER_4	0x02	4
CSDe_PRESCALER_8	0x03	8
CSDe_PRESCALER_16	0x04	16
CSDe_PRESCALER_32	0x05	32
CSDe_PRESCALER_64	0x06	64
CSDe_PRESCALER_128	0x07	128
CSDe_PRESCALER_256	0x08	256

Return Value:

None

Side Effects:

**

CSDe_CalibrateRefSensor

Description:

This function performs a binary search to AutoCalibrate the reference sensor to the level specified by the constants CSDe_REFCALIBRATION_TARGET_MSB and CSDe_REFCALIBRATION_TARGET_LSB. The resulting IDAC_D setting for the reference sensor is stored in the RAM at CSDe_bRefDAC.

C Prototype:

```
void  CSDe_CalibrateRefSensor(void)
```

Assembly:

```
lcall CSDe_CalibrateRefSensor
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDe_CalibrateSensor

Description:

This function performs a binary search to AutoCalibrate the actual sensor specified by bSensor to the level specified by the constants CSDe_SNSCALIBRATION_TARGET_MSB and

CSDe_SNSCALIBRATION_TARGET_LSB. The resulting IDAC_D setting for bSensor is stored in RAM at CSDe_baDAC.

C Prototype:

```
void  CSDe_CalibrateSensor(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall CSDe_CalibrateSensors
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

CSDe_UpdateSensorBaseline

Description:

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

The Bucket Method uses the following algorithm:

1. Each time CSDe_UpdateSensorBaseline() is called, a difference count is calculated by subtracting the previous baseline from the raw count value. This difference is stored in the CSDe_waSnsDiff[] array and is provided to you.

2. If Sensors Autoreset is disabled, each time CSDe_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 has 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 waSnsDiff[] array is reset to 0. Therefore, this array does not contain elements with values greater than 0 but below the NoiseThreshold.

C Prototype:

```
void  CSDe_UpdateSensorBaseline(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall CSDe_UpdateSensorBaseline
```

Parameter:

A => Sensor Number

Return Value:

None

Side Effects:

**

CSDe_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 CSDe_baSnsOnMask[] array.

C Prototype:

```
BYTE  CSDe_bIsSensorActive(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall CSDe_bIsSensorActive
```

Parameters:

A => Sensor Number

Return Value:

Return value of 1 if active, 0 if not active

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

Side Effects:

**

CSDe_bIsAnySensorActive

Description:

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

C Prototype:

```
BYTE  CSDe_bIsAnySensorActive(void)
```

Assembly:

```
lcall CSDe_bIsAnySensorActive
```

Parameters:

None

Return Value:

Return value of 1 if active, 0 if not active

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

Side Effects:

**

CSDe_SetDefaultFingerThresholds

Description:

This function loads the CSDe_baBtnFThreshold[] array with the FingerThreshold parameter value. This function must be called before scanning if the CSDe_baBtnFThreshold[] array is not manually loaded with custom values.

C Prototype:

```
void  CSDe_SetDefaultFingerThresholds(void)
```

Assembly:

```
lcall CSDe_SetDefaultFingerThresholds
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDe_InitializeSensorBaseline

Description:

This function loads the CSDe_waSnsBaseline[bSensor] 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 for resetting the baseline of an individual sensor.

C Prototype:

```
void  CSDe_InitializeSensorBaseline(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall CSDe_InitializeSensorBaseline
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

CSDe_InitializeBaselines

Description:

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

C Prototype:

```
void  CSDe_InitializeBaselines(void)
```

Assembly:

```
lcall CSDe_InitializeBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDe_UpdateAllBaselines

Description:

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

C Prototype:

```
void  CSDe_UpdateAllBaselines(void)
```

Assembly:

```
lcall CSDe_UpdateAllBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDe_SetSliderIdac

Description:

This function sets iDAC current for slider elements to the highest for each slider group.

C Prototype:

```
void  CSDe_SetSliderIdac(void)
```

Assembly:

```
lcall CSDe_SetSliderIdac
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDe_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 CSDe Wizard. This function is available only if slider is defined by the CSDe Wizard.

C Prototype:

```
WORD  CSDe_wGetCentroidPos(BYTE bSnsGroup)
```

Assembly:

```
mov   A, bSnsGroup
lcall CSDe_wGetCentroidPos
```

Parameters:

bSnsGroup A => 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:

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. The routine should 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 CSDe 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 CSDe_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. The noise threshold should be set carefully (high enough above the noise level) so that noise does not generate a false centroid.

CSDe_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 CSDe Wizard. This function is available only for radial slider that is defined by the CSDe Wizard.

C Prototype:

```
WORD CSDe_wGetRadialPos(BYTE bSnsGroup)
```

Assembly:

```
mov   A, bSnsGroup
lcall CSDe_wGetRadialPos
```

Parameters:

bSnsGroup A => Group Number

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

Return Value:

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

Side Effects:

The routine should be called only once after each scan to avoid getting negative difference values and baseline update. If your application monitors difference count signals, call this routine after difference count data transmission.

If any slider sensor is active, the function returns values from zero to the Resolution value set in the CSDe 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.

CSDe_wGetRadialInc

Description:

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

C Prototype:

```
WORD  CSDe_wGetRadialInc(BYTE bSnsGroup)
```

Assembly:

```
mov   A, bSnsGroup
lcall CSDe_wGetRadialInc
```

Parameters:

bSnsGroup A => Group Number

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

Return Value:

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

Finger shift value is the difference between current and previous finger positions. If there was no touch during previous scan (the last but one time CSDe_wGetRadialPos() returned -1 (FFFFh)) or there is no touch at the moment (this time CSDe_wGetRadialPos() returned -1 (FFFFh)).

Side Effects:

The routine should be called only after CSDe_wGetRadialPos() API. Because it uses internal data CSDe_waSliderPrevPos and CSDe_waSliderCurrPos that are set by the CSDe_wGetRadialPos().

Sample Firmware Source Code

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

```
//--------------------------------------------------------------------------
// Sample C code for the CSDe 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;
  CSDe_Start();
CSDe_InitializeBaselines() ; //scan all sensors first time, init baseline
  CSDe_SetDefaultFingerThresholds() ;
//
// Loop Forever
//
while (1) {
CSDe_ScanAllSensors(); //scan all sensors in array (buttons and sliders)
    CSDe_UpdateAllBaselines(); //Update all baseline levels;

//detect if any sensor is pressed
 if(CSDe_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 CSDe_waSnsResult[x] <- Raw Counts
// OUTPUT CSDe_waSnsDiff[x] <- Difference
// OUTPUT CSDe_waSnsBaseline[x] <- Baseline
// OUTPUT CSDe_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 CSDe 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;
```

```
CSDe_Start(); // Start CSDe UM
     CSDe_SetDefaultFingerThresholds(); // Set default thresholds for butt
// Initialize baseline for sensor number "3"
     CSDe_InitializeSensorBaseline(3);

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

Configuration Registers

The CSDe User Module uses the Timer1, CapSense, and Comparator PSoC blocks. Each block is personalized and parameterized through a set of registers. The set of registers used by the user module with brief descriptions are given in this section. 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 CSDe User Module

- Bank 0
  - Analog Mux Configuration Register: AMUX_CFG
    This register is used to configure 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 Registers: MUX_CR0
This register is used to control the connection between the analog mux bus and the corresponding pin.

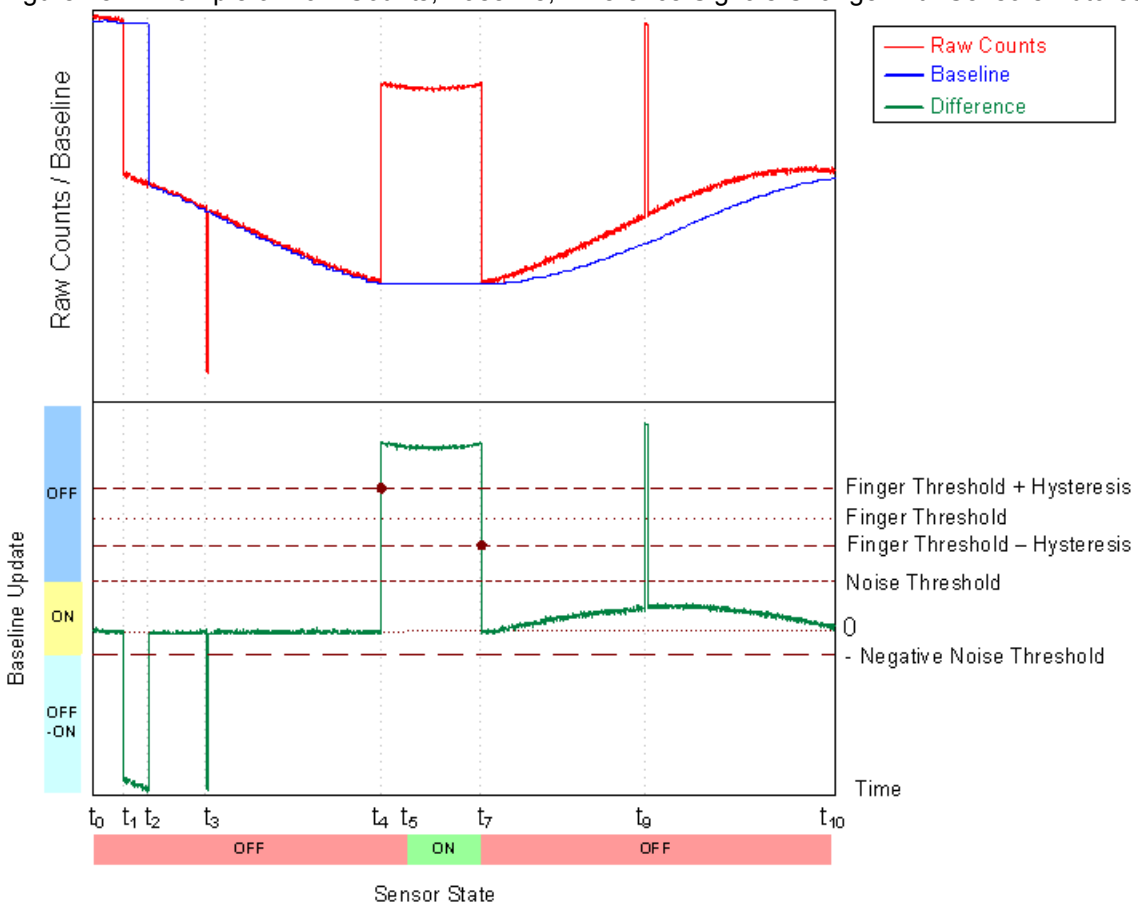
Appendix

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

Interaction of CSDe Parameters

Figure 15 and Figure 16 illustrate the baseline update and decision logic operation and can be useful for better understanding how to set user module parameters for optimum performance. Figure 15 illustrates system operation when the Sensors Autoreset parameter is set to **Disabled**. Figure 16 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 rawcount changes.The slow changes can be caused by temperature or humidity variations and the rapid changes can be triggered by a sensor touch, an ESD event, or the influence of a strong RF field.

Figure 18. 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.

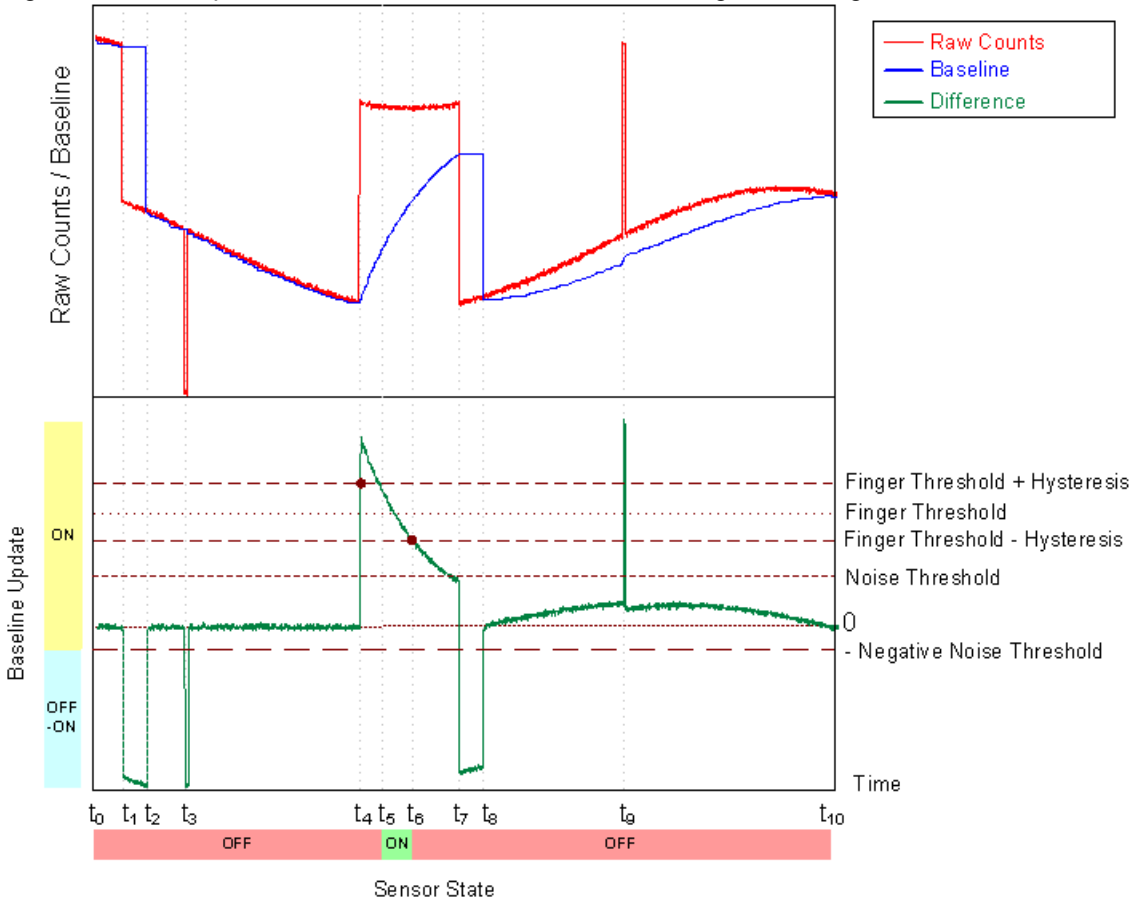
At t_1, the raw count drops sharply and the negative difference exceeds the NegativeNoiseThreshold. This situation can happen if the device is powered on when a finger is on the sensor and then the finger is removed after a period of time. At this time the baseline update mechanism is frozen and an internal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

The second large negative difference signal spike happens at t_3; this spike may have been triggered, 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 t_4. When the difference signal exceeds the FingerThreshold + Hysteresis value, the internal debounce counter is activated. If the signal exceeds this value for more than Debounce samples, the sensor state is set to on. This happens at t_5. The sensor state reverts back to the off state immediately when the difference signal drops below the FingerThreshold – Hysteresis level at t_7. The short positive spike at t_9 is filtered by the debounce counter, because the spike duration in sample units does not exceed the Debounce value.

The raw count drifts up slowly between t_7 and t_{10}. The baseline is updated using the bucket algorithm when the difference signal is below the NoiseThreshold (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 19. Example of Raw Counts, Baseline, Difference Signals Change With Sensors Autoreset Set to Enabled



The system operation in Figure 16 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. Corrected resolution value calculation in UM wizard to address error after change in diplexing. 2. Moved setting of CSD_MODE bit from ScanSensor API to Start API.
2.00	DHA	1. Deleted redundant constants in .inc file. 2. Updated list of available values for "Filter Coefficient" property. 3. Set PRS Precharge Source for the sensor and Prescaler for the reference sensor. 4. Set PreScaler frequency for sourcing RefSensor. 5. CSDe_waRawCount variable was moved to newly created ram3 area to avoid build error on max sensor count. 6. Updated Filter Coefficient section and explained limitations of dynamic reconfiguration in datasheet. 7. Changed calibration algorithm for Refsensor.
2.10	MYKZ	1. Fixed problem with saving information for sliders. 2. Updated baseline algorithm to check for negative difference counts. 3. Moved User Module to Legacy state.

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 © 2011-2013 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.