CapSense® Sigma-Delta Datasheet CSD V 2.20

Copyright © 2008-2015 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks				API Memory (Bytes)		Pins per senor
	CapSense®	I²C/SPI	Timer	Comparator	Flash	RAM	
CY8C20x66A, CY8C20x36A, CY8C20x46A, CY8C20x96A, CY8C20xx6AS, CY8C20xx6H, CY8C20045, CYONS2110-LBXC, CYONSFN2053-LBXC, CYONSFN2061-LBXC, CYONSFN2151-LBXC, CYONSFN2161-LBXC, CYONSFN2162-LBXC, CYRF89435, CY8C20065, CY7C69xxx							
User Module	1	–	1	1	1143	35	1
Slider APIs	–	–	–	–	584	79	0
Each Sensor	–	–	–	–	5	10	1

For one or more fully configured, functional code examples that use this user module go to www.cypress.com/psoccodeexamples.

Features and Overview

- Implements CapSense® capacitive sensing in the CY8C20xx6A family of PSoC® devices using sigma-delta data conversion.
- Configurable system parameters allow tuning to optimize performance in a broad range of applications.
- Supports up to 36 capacitive sensors and 6 sliders.
- Capable of detecting touches as low as 0.1 pF. Detecting a finger is possible through up to 15 mm of glass or 5 mm of plastic.
- High immunity to AC mains noise, other EMI, and power supply noise.
- Supports capacitive sensors configured as independent buttons and/or as dependent arrays to form sliders.
- Effective number of slider elements can halve the number of dedicated I/O pins using diplexing technique.
- Supports slider resolution greater than physical pitch through interpolation.
- Shield electrode provided for reliable operation with high parasitic capacitance and/or in the presence of water film.
- Guided sensor and pin assignments using the CSD wizard.
- The CY8C20045 family does not support sliders.
- The CY8C20065 family does not support sliders.

Quick Start

1. Select and place user modules that require dedicated pins (I2C and LCD for example). Assign ports and pins as required.
2. Select and place the CSD User Module.
3. Right-click the CSD User Module in the Workspace Explorer to access the CSD wizard (the wizard is explained later in the 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 that the external modulation capacitor will be connected to in the CSD Wizard.
8. Enter the pin that will be used to 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 file 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 is able to sense through insulating layers such as plastic or glass overlays. These overlays usually constitute the external enclosure of the device. These attributes make CapSense an attractive alternative to mechanical input devices like push-buttons and potentiometers. Major benefits of CapSense include:

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

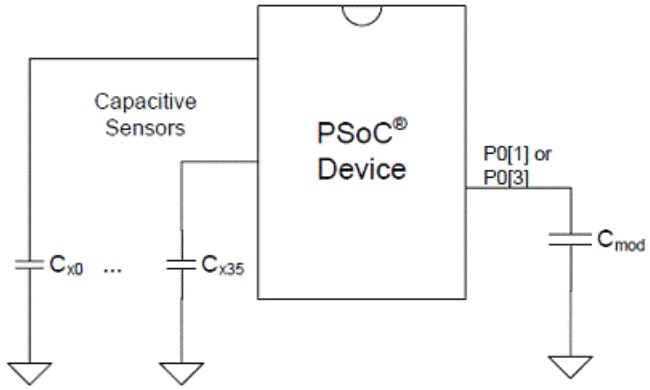
CSD is one method in Cypress' CapSense capacitive sensing portfolio. CSD uses a sigma-delta capacitance to digital code conversion circuit, key attributes of which include low EMC emission and superior immunity against EMI.

The CSD User Module consists of PCB level, IC level, and software components:

PCB Level

Figure 1 shows a schematic of CSD. 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 Design Guide for complete information on PCB level CapSense implementation.

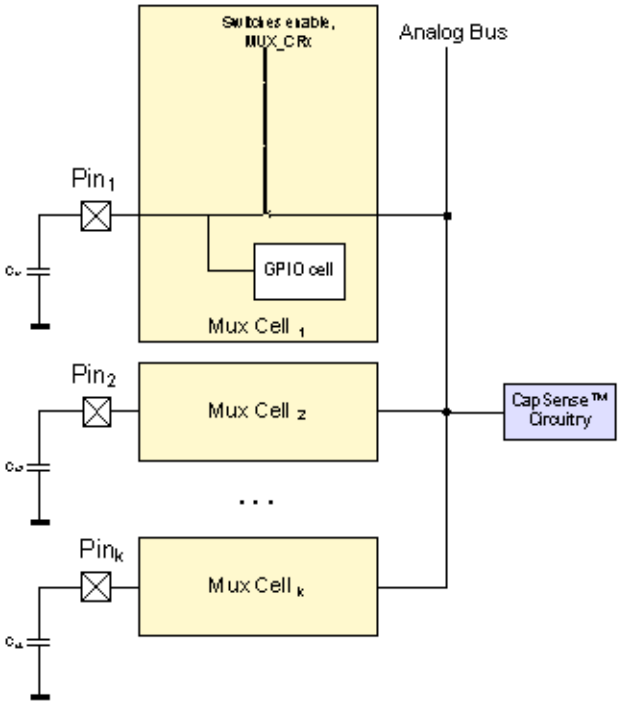
Figure 1. CSD Schematic



IC Level

CY8C20xx6A devices have an analog mux bus that allows connecting capacitive sensing analog circuitry to any PSoC pin. The CSD 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 as represented in Figure 2.

Figure 2. CY8C20xx6A AMUX Block Diagram



Software

The following are attributes of the CSD 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/O consumed for a given number of slider elements.

Recommended Reading

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

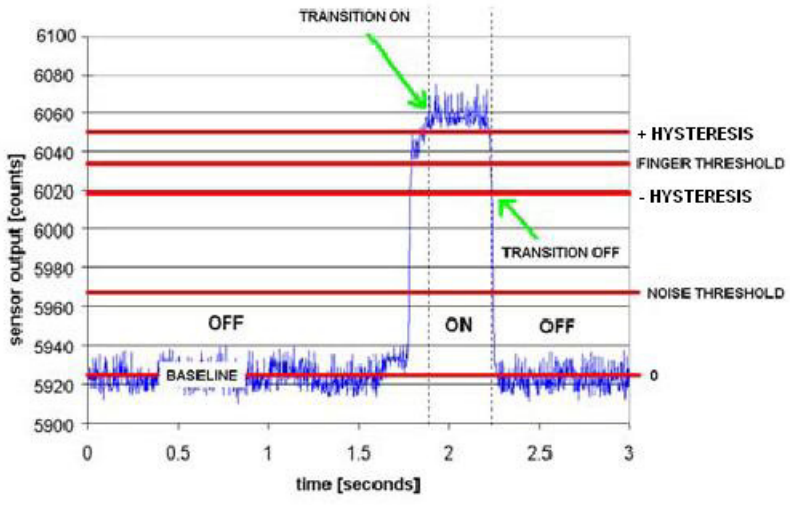
- Getting Started with CapSense
- CY8C20xx6A/H CapSense Design Guide
- CY8C21x34/B CapSense Design Guide
- CY8C20x34 CapSense Design Guide
- CY8CMBR2044 CapSense Design Guide

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 becomes activated. Figure 3 shows a typical signal (blue line) from a sensor as it is being activated. The thresholds (red lines) are all tunable to provide the desired behavior.

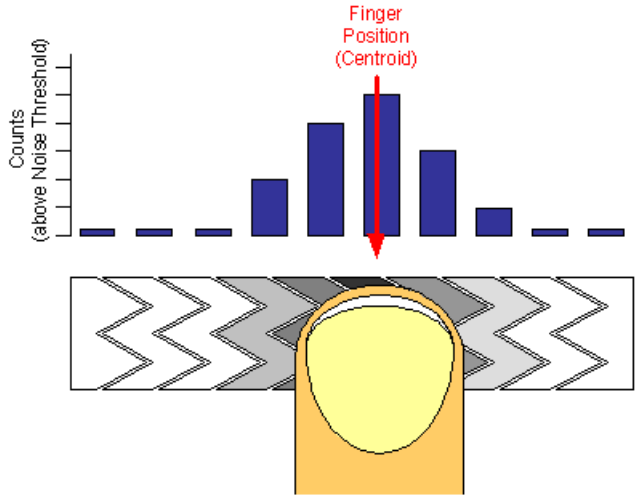
Figure 3. Capacitance Signal from a Sensor as it is Being Activated



Sliders

CapSense sliders are analogous to mechanical potentiometers. Sliders are used for controls requiring 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.

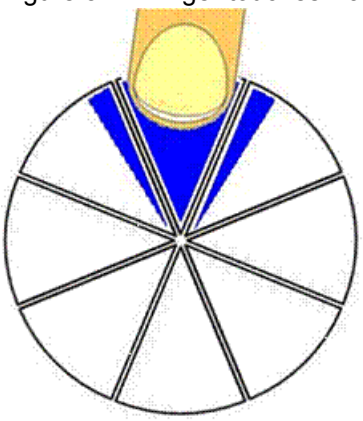
Figure 4. Interpolated Centroid Position of a Finger on a Slider



Radial Sliders

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

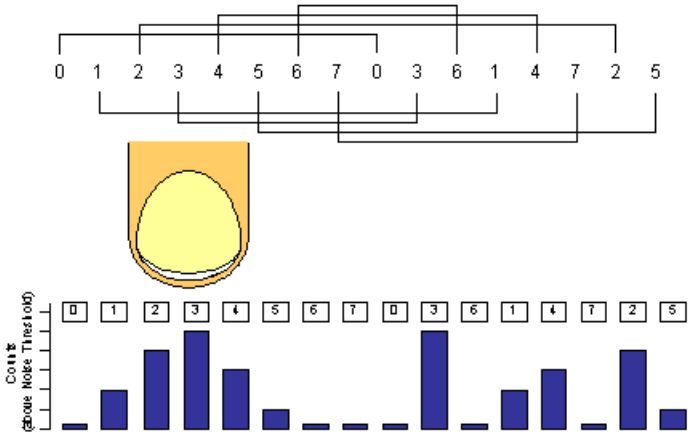
Figure 5. Finger touches Radial Slider



Diplexing

When diplexing is used, each pin on the PSoC 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 CSD 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 CSD



The close proximity of strong signals in lower half of the slider results in the same levels aliased into the upper half. However, in the upper half, the results are scattered and non-contiguous. The centroid algorithm searches for strong adjacent sets of signals to declare the resolved slider position. The pattern used for 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.

Care must be exercised to ensure the mapping of sensors to pins on the PCB matches the Index by 3 sequence used by the diplexing algorithm. The capacitance of sensor pairs in a diplexed slider should also be reasonably well matched (within 10 pF). The diplex sensor index table is automatically generated by the CSD 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

Total Slider Segment Count	Segment Sequence
28	0,1,2,3,4,5,6,7,8,9,10,11,12,13,0,3,6,9,12,1,4,7,10,13,2,5,8,11
30	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,0,3,6,9,12,1,4,7,10,13,2,5,8,11,14
32	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0,3,6,9,12,15,1,4,7,10,13,2,5,8,11,14
34	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,0,3,6,9,12,15,1,4,7,10,13,16,2,5,8,11,14
36	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,0,3,6,9,12,15,1,4,7,10,13,16,2,5,8,11,14,17
38	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,0,3,6,9,12,15,18,1,4,7,10,13,16,2,5,8,11,14,17
40	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,0,3,6,9,12,15,18,1,4,7,10,13,16,19,2,5,8,11,14,17
42	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,0,3,6,9,12,15,18,1,4,7,10,13,16,19,2,5,8,11,14,17,20
44	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,2,5,8,11,14,17,20
46	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20
48	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20,23
50	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20,23
52	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23
54	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23,26
56	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,0,3,6,9,12,15,18,21,24,27,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23,26

Slider Segment Selection Guidelines for the Diplex Slider

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

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

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

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

External Component Selection (C_{mod})

CSD 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 CSD 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. It is also strongly recommended that a 560 Ω series resistor be used 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. The driven shield electrode nulls out parasitic capacitive coupling, so the presence of water has negligible influence on sensor capacitance signal, thus preventing false activations.

Figure 7. Driven Shielding Electrode PCB Layout

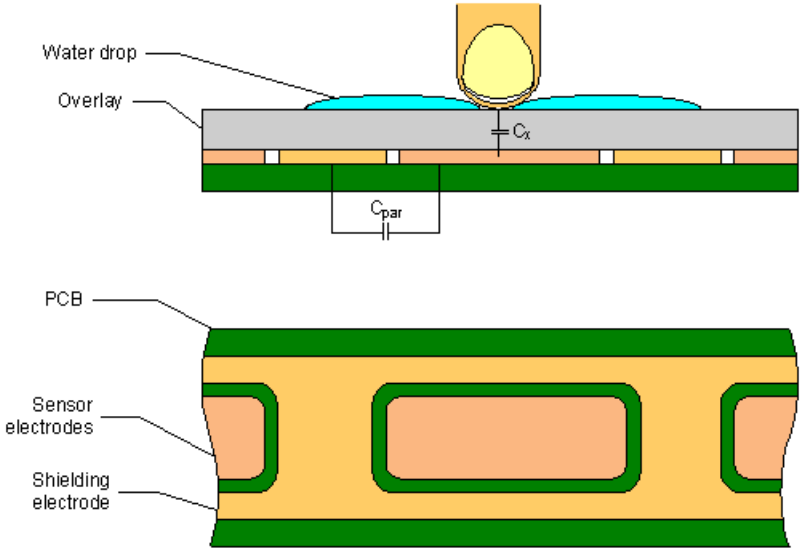


Figure 7 illustrates a driven shielding electrode for a button. As an alternative, the shielding electrode can be located on the opposite PCB layer, including the plane under the button. A hatch pattern is recommended in this case, with a fill ratio of about 30 to 40%. No additional ground plane would be 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 ohm slew limiting resistor can be connected between the PSoC device and the shielding electrode to reduced emitted EMI.

Power Supply Requirement

Table 2. CSD 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.

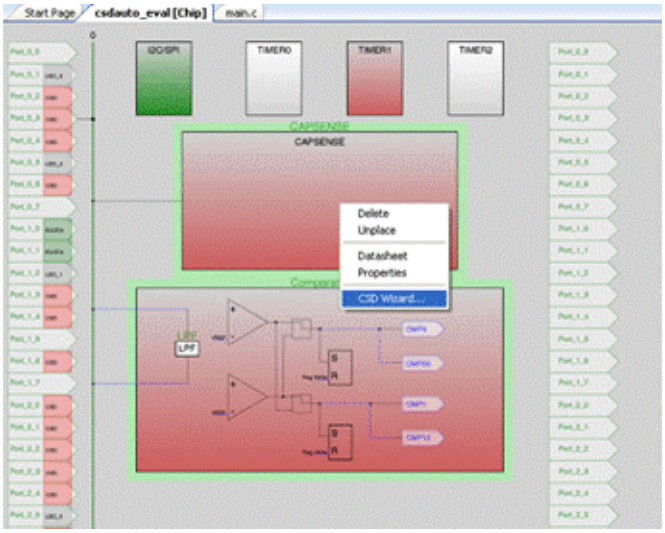
a. 1.8V is an absolute minimum VDD spec. Allowing V_{DD} to droop below 1.8V can introduce excessive noise.

Placement

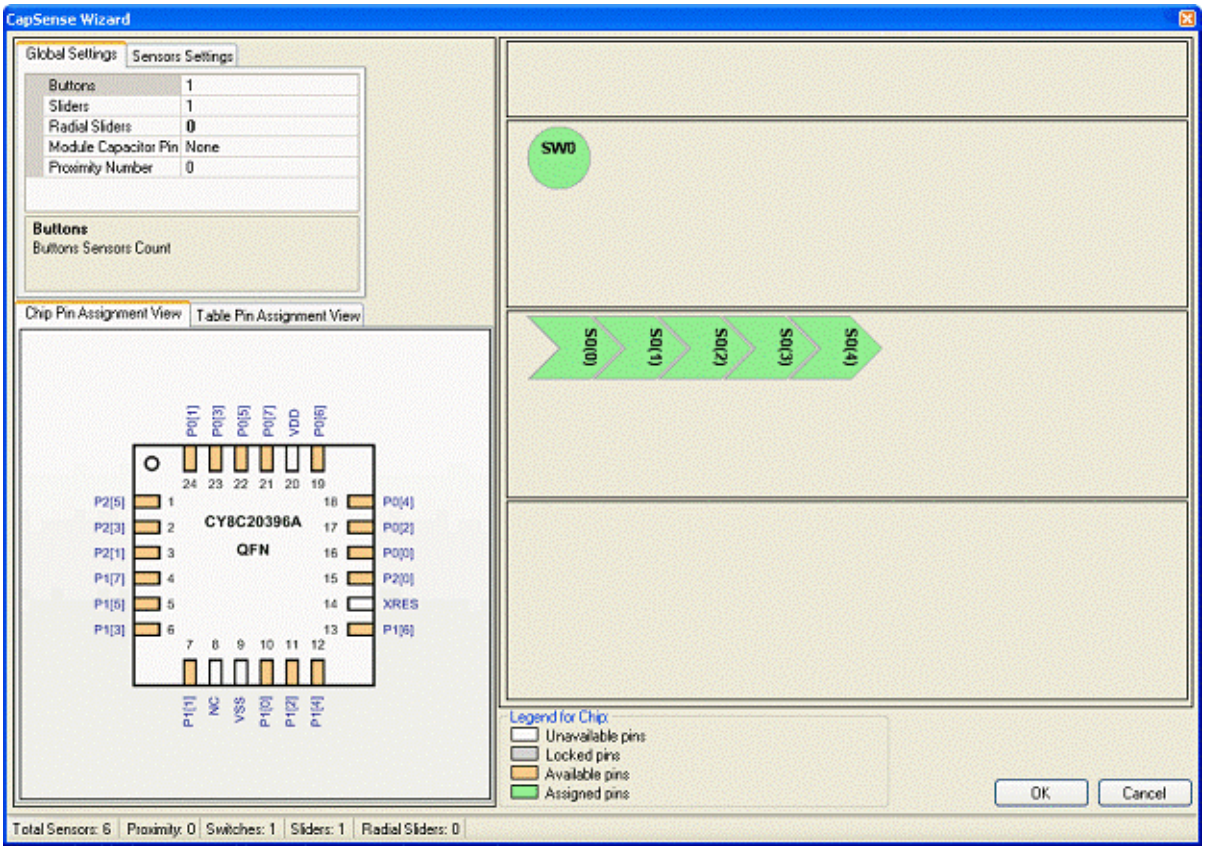
The CapSense and Timer1 blocks are assigned to CSD when the user module is instantiated. Alternate placements are not available. User modules that require dedicated pin resources, including the LCD and I2CHW, must be placed before starting the CSD Wizard. This ensures that the dedicated pins are reserved and cannot be inadvertently assigned as sensors when sensors are mapped to I/O pins in the CSD 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.

CSD Wizard

1. To access the CSD Wizard, right-click on the CapSense Block in the Device Editor Interconnect View and select the CSD Wizard with a left mouse click.



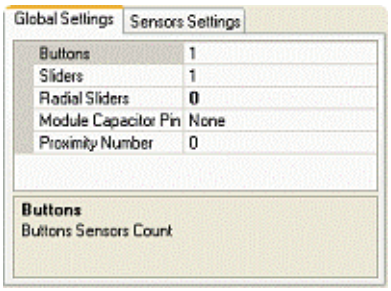
2. The Wizard opens showing the numeric entry boxes for the number of sensors and the number of slider sensors.



Wizard Pin Legend

- White – The pin cannot be used as a CapSense input.
- Gray – The pin is locked. There are two possible causes for this. The first possibility is that another user module such as the LCD or I^2C has claimed the pin. The second possibility is that the name of the pin has been changed from its default.To return the pin name to its default, in the Pinout view expand the pin, from the **Select** menu, select **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.

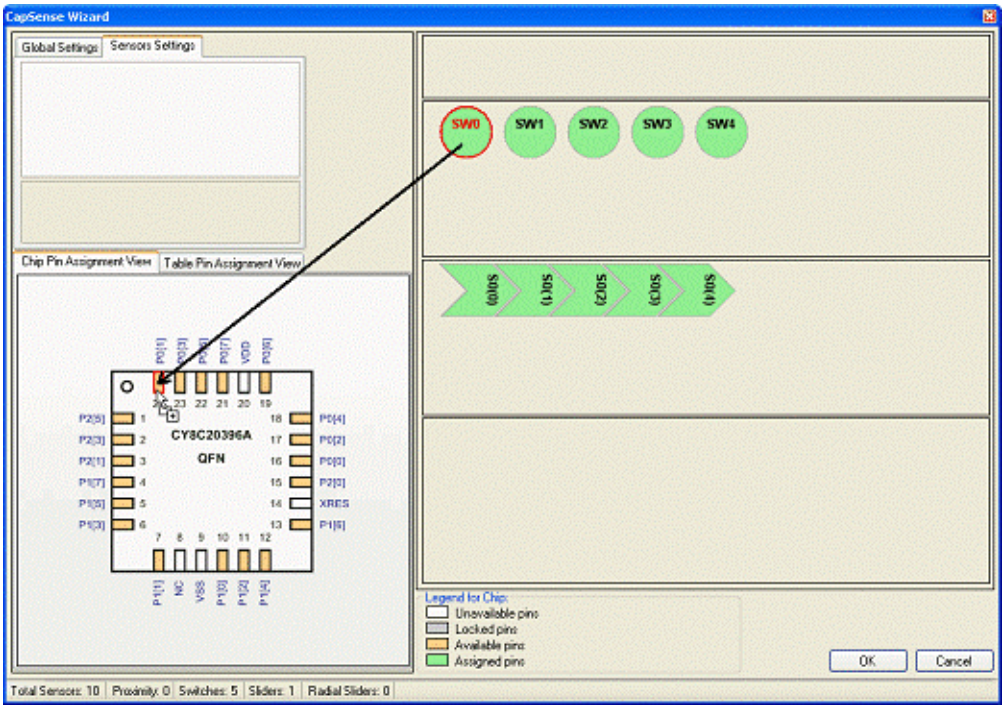
3. Type the number of independent buttons, sliders, and radial sliders. The total number of sensors (buttons plus slider elements) is limited to the number of pins available. After entering the data, press the [Enter] key to update the display with the new value.



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

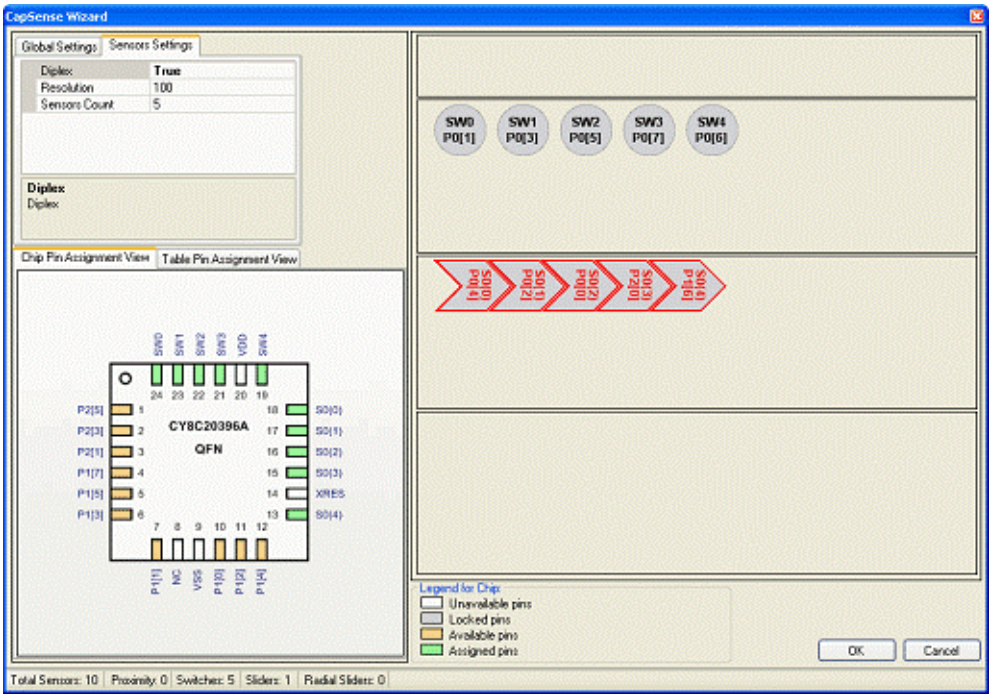


5. Select modulator capacitor (C_{mod}) pin. Choose from either P0[1] or P0[3].
6. Type the output resolution. The minimum value is five. The maximum value can be calculated using the equation (number of pins used for sensors - 1) x 2^8 - 1 or (2 x number of pins used for sensors - 1) x 2^8 - 1 for diplexed sliders.CSD attempts to interpolate the touch results to the specified resolution using the relative strength of adjacent segments. The software reports touch results on the slider between zero and the resolution - 1.
7. Select Diplex, if 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 onto the pin in the Pin Assignment View. You can choose to drag sensors onto pins in the Chip Pin Assignment View or the Table Pin Assignment View. The I/O pin is green after selection and is no longer available. Change sensor assignments by dragging the port pin back to the uncommitted table. Make sure to 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 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 and you can then re-assign it.

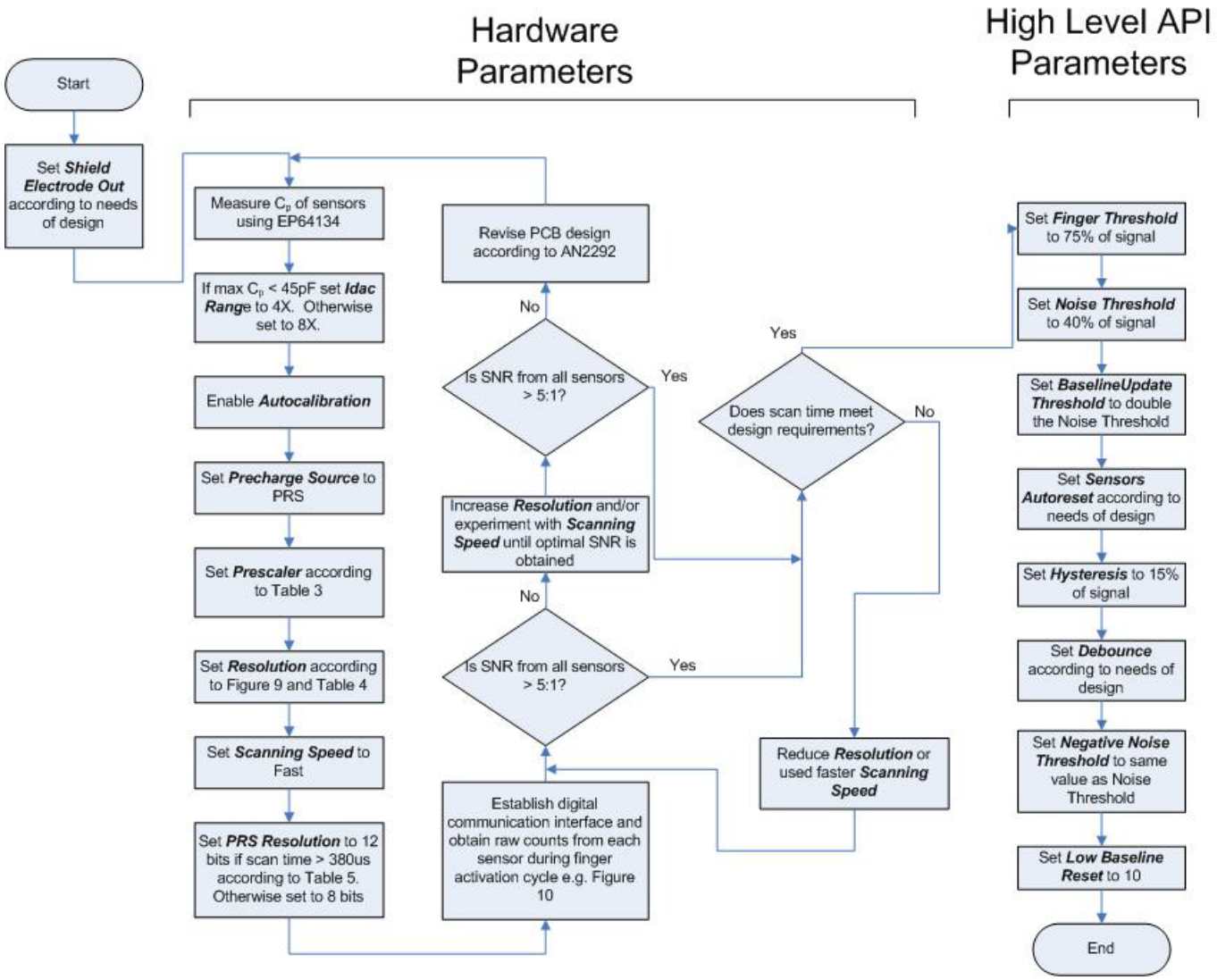
After completing the Wizard, click Generate Application. Based on your entries for sensor count, pin assignment, diplexing, and resolution, a set of tables will be generated. The tables are located in CSD_Table.asm

User Module Parameters - Tuning Guide

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

Figure 8 is a flowchart showing the tuning process for the CSD User Module parameters. CSD User Module parameters can be separated into two broad categories: 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 ensure that when any hardware parameter is changed, the corresponding high level API parameters are adjusted accordingly. Tuning of CSD User Module parameters should always begin with the Hardware Parameter.

Figure 8. CSD 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 and/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 SetIdacValue(), SetPrescaler(), and SetScanMode() API functions can be used to configure the respective Hardware parameters for each sensor. In this case, these functions should be used in conjunction with CSD_ScanSensor() API.

Tables 3 and 5 provide tuning recommendations for several key Hardware Parameters based on sensor C_p. C_p values depend on characteristics of the PSoC, PCB layout, and proximity of other components in the assembled product. That being the case, C_p must be measured in situ with the system in its final assembled state, that is, in the same enclosure and with the same overlay as the system will have in service. The best way to measure C_p is to use the example project entitled "Measuring Absolute Sensor Capacitance with a CY8C20xx6 CapSense Controller" (EP64134) available on www.cypress.com. This project measures the absolute capacitance of each sensor in a system using the PSoC itself thus taking into account all factors affecting C_p. See the documentation associated with the example project for instructions on its set up and use.

ShieldElectrodeOut

The ShieldElectrodeOut parameter selects whether and to which pin a shield signal is driven out in CSD designs. A driven shield in useful in applications where the sensor overlay may become wet (see CY8C20xx6A/H CapSense Design Guide), to shield against EMI, or to mitigate excessively high C_p. Available pins for driven shield are P0[7] or P1[2]. Enabling the shield electrode is not required for CSD operation, is disabled by default, and is generally not needed. The default setting is None.

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 value is 4X.

Autocalibration

This parameter enables a successive approximation for the iDAC setting that establishes a raw count baseline at or above 85 percent. Autocalibration should always be set to enabled in CY8C20xx6A CSD designs. The ability of the Autocalibration algorithm to successfully set the iDAC relies on a proper Prescaler setting and that C_{mod} be of the recommended size. 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 value is 20.

Precharge Source

Note that using Prescaler as the Precharge source can introduce dead zones where signal counts due to finger drop to zero. Hence, for robust CapSense operation, PRS should be selected over Prescaler as the precharge source. This parameter selects the sensor switching clock source. The available options are Prescaler, which uses the IMO through a divider, or PRS, which 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 is therefore the recommended default setting for Precharge Source. In some instances, the Prescaler Precharge Source can provide higher SNR (signal-to-noise ratio). However, when using copper circuitry, this SNR improvement is usually marginal and rarely justifies foregoing the benefits of PRS. The default setting is PRS.

Prescaler

Prescaler is the divider applied to the IMO to develop the Precharge Clock. This is the most critical Hardware user module parameter for properly tuning a CSD 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 value is 2.

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

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

Note 1: This combination of Precharge Source, Prescaler and C_p is not recommended

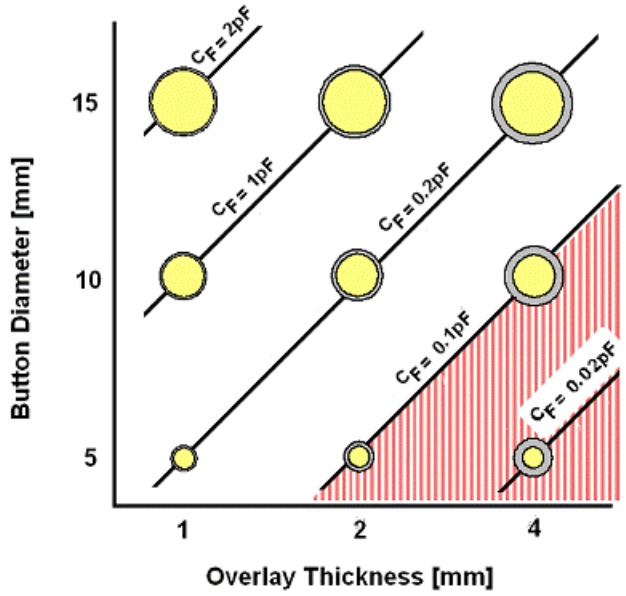
Resolution

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 4 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 value is 12.

Table 4. Resolution Setting Based on Finger Capacitance and Cp

Cp (pF)	C_f = 0.1pF	C_f = 0.2 pF	C_f = 0.4pF	C_f = 0.8pF
<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

Figure 9. Finger Capacitance (C_f) Based on Overlay Thickness and Circular Sensor Diameter



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, but not all, cases slower scanning speed can yield higher SNR at the expense of longer scan time and more power consumption. Table 5 shows the actual scan time in µs for a single sensor based on Resolution and Scanning Speed. The default setting is Normal.

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

	Scanning Speed			
Resolution (bits)	Ultra Fast	Fast	Normal	Slow
9	57	78	125	205
10	78	125	205	380
11	125	205	380	720
12	205	380	720	1400
13	380	720	1400	2800
14	720	1400	2800	5600
15	1400	2800	5600	11000
16	2800	5600	11000	22000

PRS Resolution

This parameter changes the PRS sequence length. Possible values are 8 and 12 bit. Corresponding sequence length is 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) and Scanning Speed parameters. 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 value is 8 bits.

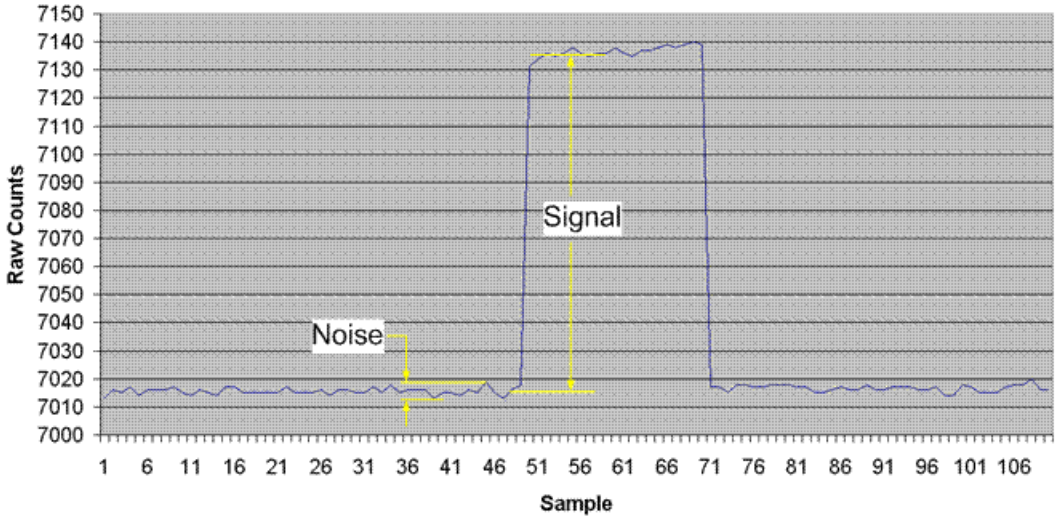
High Level API Parameters

High Level API parameters determine the behavior of high level firmware algorithms that discriminate between sensor activations and noise, and compensate for signal drift caused by environmental conditions. In order to determine proper values for these parameters a digital communication interface must be established with the system to monitor raw counts,and baseline and difference counts during a finger activation event for each sensor. This data is stored in arrays named CSD_waSnsBaseline[], CSD_waSnsResult[], and CSD_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 and in the same EMI environment as will exist in service.

Figure 10 show the typical raw counts obtained from a sensor during a finger activation cycle, that is, sensor is activated then deactivated. Superimposed over the data are labels that indication 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 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 CY8C20xx6A/H CapSense Design Guide to raise SNR to at least 5:1.

Figure 10. 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, the raw count response of each sensor to finger activation must be analyzed. 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, that is, smallest finger. Finger Threshold has no meaning and function for sensors that are elements in a slider. The default value is 60.

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 value is 10.

BaselineUpdate Threshold

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

- the raw count value returned from a scan is above the current baseline and

- the difference between the raw count value and baseline is below the Noise Threshold

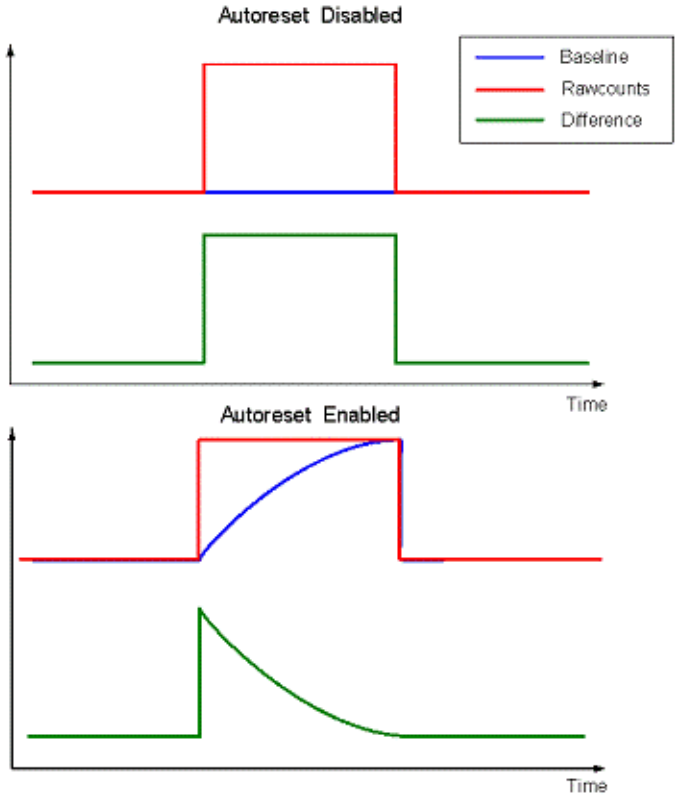
The BaselineUpdateThreshold sets the value that the "bucket" must reach for the baseline to be incremented. A good starting point for this parameter is double the Noise Threshold. The default value is 100.

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 - 10 s), however it provides the benefit of preventing sensors from getting stuck on due to sudden rises in raw counts that are not cause 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.

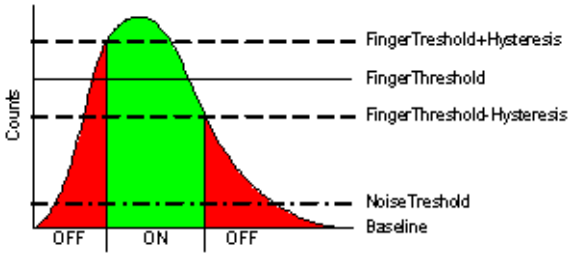
Figure 11. Effect of the Sensor Autoreset Parameter on Baseline



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-Hystersis to change state from ON to OFF. A good starting point for the Hysteresis is 15 percent of the raw count signal (see Figure 9). The default value is 10.

Figure 12. Relationship Between Finger Threshold and Hysteresis on Button Sensor State



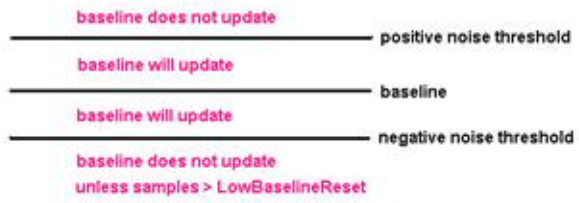
Debounce

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 CSD_bIsSensorActive or CSD_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 baseline will reset, that is, snap down to the raw count value. If raw counts are below this level, 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 value is 10.

Figure 13. Relationship Between Noise Thresholds and Baseline Reset



Low Baseline Reset

This parameter works with Negative Noise Threshold to set the number samples with raw counts less than baseline needed to make baseline snap down to the raw count level. If the raw count value is less than 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 value is 50.

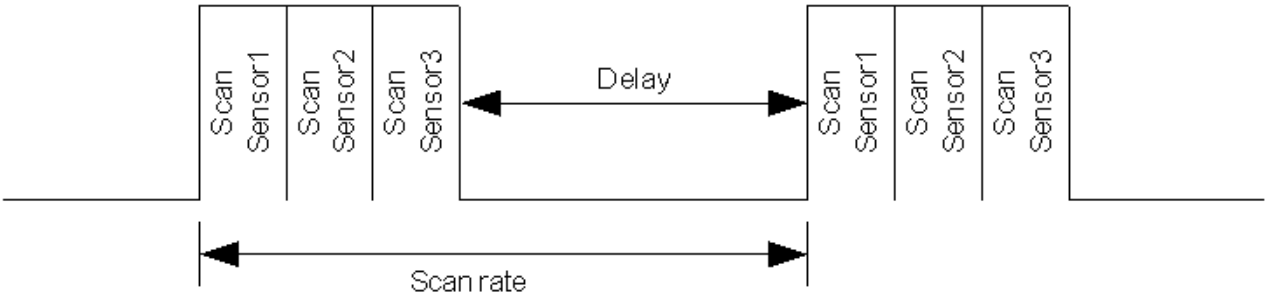
FMEA_Cp_Range_Test

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 14. 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 fewer 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 CSD_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 CSD for simplicity.

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

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

Entry Points are supplied to initialize the CSD, start it sampling, and stop the CSD. In all cases the instance name of the module replaces the CSD 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 several global arrays:

- CSD_waSnsBaseline[]
- CSD_waSnsResult[]
- CSD_waSnsDiff[]
- CSD_baSnsOnMask[]

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

- CSD_UpdateAllBaselines();
- CSD_UpdateSensorBaseline();
- CSD_InitializeBaselines().

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

- CSD_ScanSensor();
- CSD_ScanAllSensors().

CSD_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.

CSD_baSnsOnMask[] – This is a byte array that holds the sensor on or off state (for buttons or sliders). CSD_baSnsOnMask[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). CSD_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 CSD_baSnsOnMask[] data is updated by CSD_bIsSensorActive(BYTE bSensnor) function or CSD_bIsAnySensorActive() routines.

CSD_Start

Description:

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

C Prototype:

```
void  CSD_Start()
```

Assembly:

```
lcall  CSD_Start
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD_Stop

Description:

Stops the sensor scanner, disables internal interrupts, and calls CSD_ClearSensors() to reset all sensors to an inactive state.

C Prototype:

```
void  CSD_Stop()
```

Assembly:

```
lcall  CSD_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD_Resume

Description:

Resumes the user module operation after CSD_Stop call.

C Prototype:

```
void  CSD_Resume()
```

Assembly:

```
lcall  CSD_Resume
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD_ScanSensor

Description:

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

C Prototype:

```
void CSD_ScanSensor(BYTE bSensor);
```

Assembly:

```
mov A,  bSensor
lcall CSD_ScanSensor
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects

**

CSD_ScanAllSensors

Description:

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

C Prototype:

```
void CSD_ScanAllSensors();
```

Assembly:

```
lcall CSD_ScanAllSensors
```

Parameters:

None

Return Value:

None

Side Effects

**

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

2. If Sensors Autoreset is disabled, each time CSD_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 Noise-Threshold.

C Prototype:

```
void CSD_UpdateSensorBaseline(BYTE bSensor)
```

Assembly:

```
mov   A,   bSensor
lcall  CSD_UpdateSensorBaseline
```

Parameter:

A => Sensor Number

Return Value:

None

Side Effects:

**

CSD_UpdateAllBaselines

Description:

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

C Prototype:

```
void  CSD_UpdateAllBaselines()
```

Assembly:

```
lcall CSD_UpdateAllBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD_bIsSensorActive

Description:

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

C Prototype:

```
BYTE  CSD_bIsSensorActive(BYTE bSensor)
```

Assembly:

```
mov  A,  bSensor
```

```
lcall CSD_bIsSensorActive
```

Parameters:

bSensor 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:

**

CSD_bIsAnySensorActive

Description:

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

C Prototype:

```
BYTE  CSD_bIsAnySensorActive()
```

Assembly:

```
lcall CSD_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:

**

CSD_wGetCentroidPos

Description:

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

C Prototype:

```
WORD  CSD_wGetCentroidPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall CSD_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 CSD 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 CSD_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.

CSD_wGetRadialPos

Description:

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

C Prototype:

```
WORD  CSD_wGetRadialPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
call CSD_wGetRadialPos
```

Parameters:

bSnsGroup A => Group Number

This parameter is a number of radial slider you are working with. You can get its number using the CSD User Module 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 CSD 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.

CSD_wGetRadialInc

Description:

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

C Prototype:

```
WORD  CSD_wGetRadialInc(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall CSD_wGetRadialInc
```

Parameters:

bSnsGroup A => Group Number

This parameter is a number of radial slider you are working with.You can get its number using the CSD User Module 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 CSD_wGetRadialPos() returned -1 (FFFFh)) or there is no touch at the moment (this time CSD_wGetRadialPos() returned -1 (FFFFh))

Side Effects:

The routine should be called only after CSD_wGetRadialPos() API. Because it uses internal data CSD_waSliderPrevPos and CSD_waSliderCurrPos that are set by the CSD_wGetRadialPos().

CSD_InitializeSensorBaseline

Description:

Loads the CSD_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 CSD_InitializeSensorBaseline(BYTE bSensor)
```

Assembly:

```
mov A, bSensor
lcall CSD_InitializeSensorBaseline
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

CSD_InitializeBaselines

Description:

Loads the CSD_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 CSD_InitializeBaselines()
```

Assembly:

```
lcall CSD_InitializeBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD_SetDefaultFingerThresholds

Description:

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

C Prototype:

```
void CSD_SetDefaultFingerThresholds()
```

Assembly:

```
lcall CSD_SetDefaultFingerThresholds
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD_SetScanMode

Description:

Sets scanning speed and resolution. This function can be called at runtime to change the scanning speed and resolution. The function overwrites the user module parameter settings. This 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. This function can be used in conjunction with CSD_ScanSensor() function.

C Prototype:

```
void CSD_SetScanMode(BYTE bSpeed, BYTE bResolution);
```

Assembly:

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

Parameters:

bSpeed: Scanning Speed

The following constants are given for the bSpeed parameter:

Constant	Value
CSD_ULTRA_FAST_SPEED	0x00
CSD_FAST_SPEED	0x01
CSD_NORMAL_SPEED	0x02
CSD_SLOW_SPEED	0x03

bResolution: Scanning Resolution. Set this value to the required number of bits of resolution. Thisparameter value must not be lower than 9 or greater than 16.

The following possible constants are given for the bResolution parameter:

Constant	Value
CSD_9_BIT_RESOLUTION	9
CSD_10_BIT_RESOLUTION	10
CSD_11_BIT_RESOLUTION	11
CSD_12_BIT_RESOLUTION	12
CSD_13_BIT_RESOLUTION	13
CSD_14_BIT_RESOLUTION	14
CSD_15_BIT_RESOLUTION	15
CSD_16_BIT_RESOLUTION	16

Return Value:

None

Side Effects:

**

CSD_SetSliderIdac

Description:

Sets iDAC current for slider elements to the highest for each slider group.

C Prototype:

```
void CSD_SetSliderIdac(void)
```

Assembly:

```
lcall  CSD_ SetSliderIdac
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD_SetIdacValue

Description:

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

C Prototype:

```
void CSD_SetIdacValue(BYTE bRefValue);
```

Assembly:

```
mov     A, bIdacValue
lcall    CSD_SetIdacValue
```

Parameters:

bIdacValue - sets the iDAC value. Accepted values are 1..255.

Return Value:

None

Side Effects:

**

CSD_SetPrescaler

Description:

This function overwrites the user module parameter settings Prescaler Value. Use it if some sensors need to be scanned with Prescaler setting. This function can be used in conjunction with CSD_ScanSensor().

C Prototype:

```
void CSD_SetPrescaler(BYTE bPrescaler);
```

Assembly:

```
mov     A, bPrescaler
lcall    CSD_SetPrescaler
```

Parameters:

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

Name	Value	Prescaler
CSD_PRESCALER_1	0x00	1
CSD_PRESCALER_2	0x01	2
CSD_PRESCALER_4	0x02	4
CSD_PRESCALER_8	0x03	8
CSD_PRESCALER_16	0x04	16
CSD_PRESCALER_32	0x05	32
CSD_PRESCALER_64	0x06	64
CSD_PRESCALER_128	0x07	128
CSD_PRESCALER_256	0x08	256

Return Value:

None

Side Effects:

**

CSD_CalibrateSensors

Description:

Adjusts iDAC current to obtain raw count near wLevel value and stores results in the global array CSD_baDAC[].

C Prototype:

```
void CSD_CalibrateSensors(WORD wLevel)
```

Assembly:

```
lcall  CSD_CalibrateSensors
```

Parameters:

wlevel - the target raw data value.

Return Value:

None

Side Effects:

**

CSD_ClearSensors

Description:

Clears all sensors to the nonsampling state by sequentially calling CSD_wGetPortPin() and CSD_DisableSensor() for each of the sensors.

C Prototype:

```
void  CSD_ClearSensors()
```

Assembly:

```
lcall  CSD_ClearSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD_wReadSensor

Description:

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

C Prototype:

```
WORD  CSD_wReadSensor(BYTE bSensor)
```

Assembly:

```
mov A, bSensor
lcall CSD_wReadSensor
```

Parameters:

A => Sensor Number

Return Value:

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

Side Effects:

**

CSD_wGetPortPin

Description:

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

C Prototype:

```
WORD  CSD_wGetPortPin(BYTE bSensorNum)
```

Assembly:

```
mov  A,  bSensorNumber
lcall     CSD_wGetPortPin
```

Parameters:

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

Return Value:

A => Sensor Bitmap

X => Port Number

Side Effects:

**

CSD_EnableSensor

Description:

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

Assembly:

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

Parameters:

A => Sensor Bitmap

X => Port Number

Return Value:

None

Side Effects:

**

CSD_DisableSensor

Description:

Disables the sensor selected by the CSD_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 CSD_wGetPortPin() function.

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Bitmap

X => Port Number

Return Value:

None

Side Effects:

**

CSD_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 continuously scans the sensors. The communication section can be used to communicate values to a PC charting tool.

```
//-------------------------------------------------------------------------
// Sample C code for the CSD 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;
  CSD_Start();
CSD_InitializeBaselines() ; //scan all sensors first time, init baseline
  CSD_SetDefaultFingerThresholds() ;
//
// Loop Forever
```

```
//
while (1) {
CSD_ScanAllSensors(); //scan all sensors in array (buttons and sliders)
    CSD_UpdateAllBaselines(); //Update all baseline levels;

//detect if any sensor is pressed
 if(CSD_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 CSD_waSnsResult[x] <- Raw Counts
// OUTPUT CSD_waSnsDiff[x] <- Difference
// OUTPUT CSD_waSnsBaseline[x] <- Baseline
// OUTPUT CSD_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 CSD module
//-------------------------------------------------------------------------

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

void main(void)
{
     M8C_EnableGInt;
CSD_Start(); // Start CSD UM
     CSD_SetDefaultFingerThresholds(); // Set default thresholds for buttons

// Initialize baseline for sensor number "3"
     CSD_InitializeSensorBaseline(3);

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

Example 3. The following example demonstrates the ability to scan different sensors with different scanning parameters using the CSD_SetScanMode() function. Useful when need to performs buttons touch detection and proximity detection. The buttons are scanned with low resolution to reduce the scan

time, the proximity is scanned with higher resolution to get maximum sensitivity. You can adapt this code to scan proximity less frequently and only when no button touch is detected.

```
//-------------------------------------------------------------------------
// Sample C code for the CSD module
// Scanning sensors with different scanning speed and resolution
//-------------------------------------------------------------------------

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

void main(void)
{

     M8C_EnableGInt;

CSD_Start();
     CSD_SetDefaultFingerThresholds();

// Set UltraFast, 9-bit resolution mode for baseline calculations
     CSD_SetScanMode(0, 9);

     // Initialize baselines for all of the sensors which operate in
     // Ultra Fast mode and 9-bit resolution
     CSD_InitializeSensorBaseline(0);
     CSD_InitializeSensorBaseline(1);
     CSD_InitializeSensorBaseline(2);

     // Set Slow, 14-bit resolution mode for baseline calculations
     CSD_SetScanMode(3, 14);
    // Initialize baselines for all of the sensors which operate in
     // Slow mode and 14-bit resolution
     CSD_InitializeSensorBaseline(3);

     while (1) {
          // Set UltraFast, 9-bit resolution mode for the following buttons
          CSD_SetScanMode(0, 9);
          // Scan sensor number "0"
          CSD_ScanSensor(0);
          // Scan sensor number "1"
          CSD_ScanSensor(1);
          // Scan sensor number "2"
          CSD_ScanSensor(2);

          // Set Slow, 14-bit resolution mode for the following sensor
          CSD_SetScanMode(3, 14);
          // Scan sensor number "3"
          CSD_ScanSensor(3);

          CSD_UpdateAllBaselines();
//detect if any sensor is pressed
          if(CSD_bIsAnySensorActive()){
               // Add user code here to proceed the buttons pressing
       }
     }
}
```

Example 4. The following example demonstrates the ability to set the different Finger Threshold levels for each sensor. Useful when different sensors are placed on different locations and some sensors are more sensitive than others.

```
//-------------------------------------------------------------------------
// Sample C code for the CSD module
// Set individual finger threshold parameter for each sensor
//-------------------------------------------------------------------------

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

void main(void)
{
 M8C_EnableGInt;

CSD_Start();
 CSD_InitializeBaselines();

 // set finger threshold for sensor "0"
 CSD_baBtnFThreshold[0] = 10;
 // set finger threshold for sensor "1"
 CSD_baBtnFThreshold[1] = 20;
 // set finger threshold for sensor "2"
 CSD_baBtnFThreshold[2] = 30;
 // set finger threshold for sensor "3"
 CSD_baBtnFThreshold[3] = 40;
 // set finger threshold for sensor "4"
 CSD_baBtnFThreshold[4] = 50;
 // set finger threshold for sensor "5"
 CSD_baBtnFThreshold[5] = 255;
 // set finger threshold for sensor "6"
 CSD_baBtnFThreshold[6] = 200;

 while (1) {
  // Scan continuously all sensors
  CSD_ScanAllSensors();
  CSD_UpdateAllBaselines();
  //detect if any sensor is pressed
  if(CSD_bIsAnySensorActive()){
   // Add user code here to proceed the buttons pressing
  }
 }
}
```

Configuration Registers

Table 6. Block CapSense, Register: CS_CR0

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

Table 7. Block CapSense, Register: CS_CR1

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

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

Table 8. Block CapSense, Register: CS_CR2

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

Table 9. Block CapSense, Register: CS_CR3

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

Table 10. Block CapSense, Register: CS_CNTH

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

Table 11. Block CapSense, Register: CS_CNTL

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

Table 12. Block CapSense, Register: PRS_CR

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

Table 13. Block Timer, Register: PT1_CFG

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

Table 14. Block Timer, Register: PT1_DATA0

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

Table 15. Block Timer, Register: PT1_DATA1

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

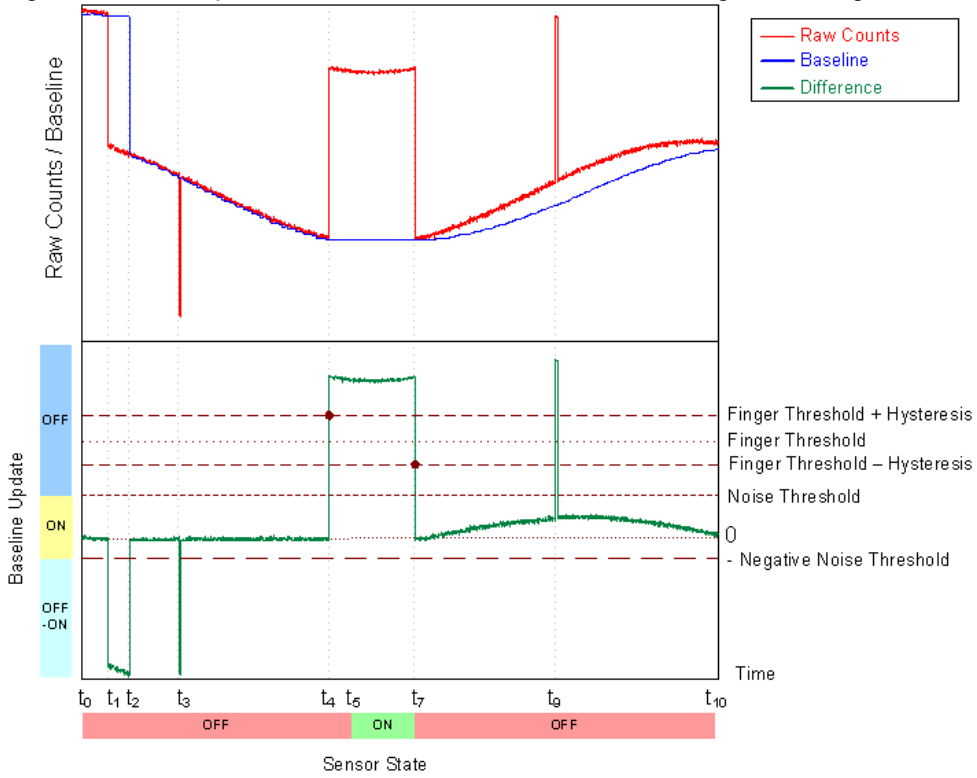
Appendix

The following sections contain information beyond what is usually included in user module datasheets. The detailed information was developed by Cypress engineers to help you successfully design CapSense applications.

Interaction of CSD Parameters

Figure 14 and Figure 15 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 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 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 15. Example of Raw Counts, Baseline, Difference Signals Change With SensorsAutoreset 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.

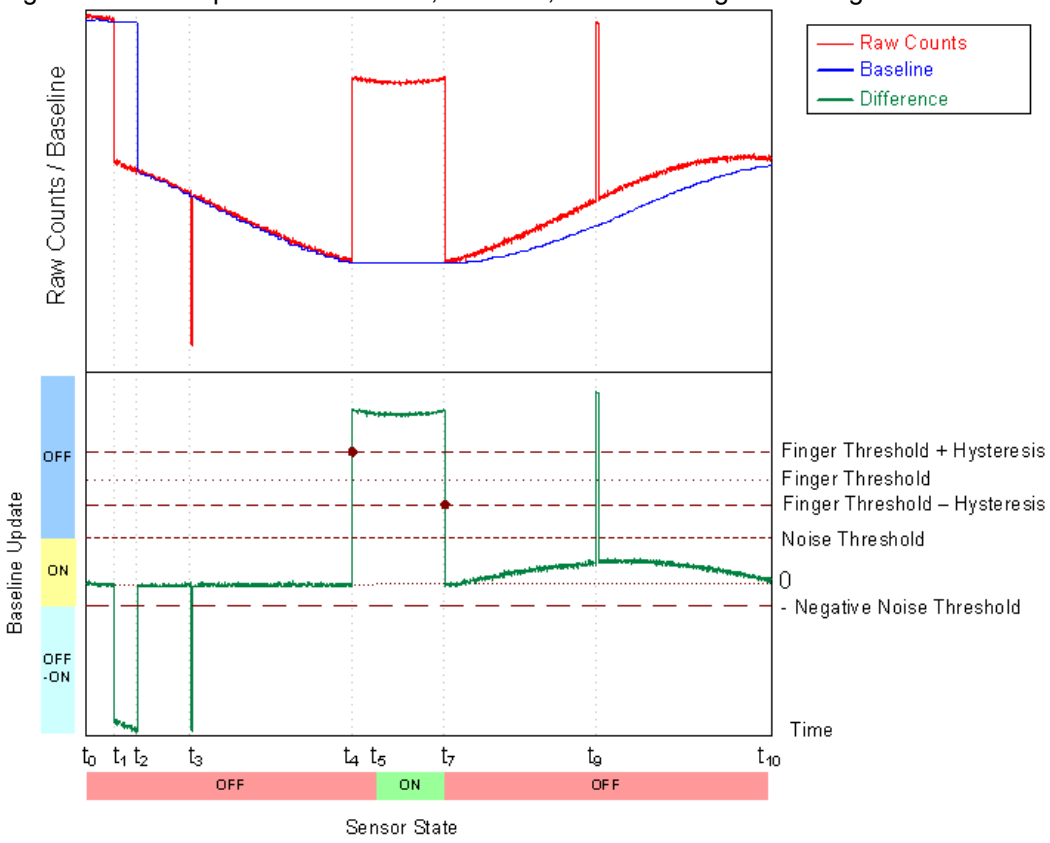
The second large negative difference signal spike happens at t_3; this spike may have been triggered, for example, by an ESD event. Because the spike duration in the sample count is less than the LowBaselineReset parameter, the baseline is kept on hold and the spike is filtered. This prevents a false 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 (SensorsAutoreset is set to Disabled), the difference signal is proportional to the drift rate. It is possible to control the baseline update speed using

the BaselineUpdate Threshold parameter. Lower parameter values provide faster baseline update speeds.

Figure 16. Example of Raw Counts, Baseline, Difference Signals Change With SensorsAutoreset 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
1.1	DHA	1. This version has the capability to add an additional slider. 2. Added Autocalibration parameter.
1.20	DHA	Added radial Slider functionality for CY8C20xx6 and CY8CTMA3xx evice families.
1.30	DHA	1. Transferred the Diplex Table from "AREA UserModules" to "AREA lit" to fix code compression issues. 2. Set the default "DiplexTable" parameter value to 0x0112 to address build errors when wizard was not run. 3. Added the "DiplexUsed" parameter to fix code compression issues. 4. Optimized code to decrease time to save user module parameters in the wizard. 5. Renamed "Switches" to "Buttons" in the wizard GUI.
2.00	DHA	1. Updated area declarations to support Imagecraft optimization. 2. Added symbolic names for the Resolution parameter in this user module datasheet. 3. Deleted the redundant modification of Bit 1 CS_MISC register usage from the CSD User Module code. 4. Removed 1x and 2x values from IDAC range from the user module property. 5. Updated user module wizard help. Added description of the slider resolution parameter min/max values.
2.10	DHA	1. Corrected resolution value calculation in user module wizard to address the error after change in diplexing. 2. Corrected calculation of maximum sensor count of sliders when diplexing is enabled. 3. Changed calibration resolution from 9 bits to 12 bits in CSD_Start API. 3. Datasheet updates: a. Removed 1x and 2x values from IDAC range. b. Updated section "Precharge Source". Added note about dead zones when Prescaler is used as precharge source. c. Updated example 2 sample code d. Updated PRS resolution parameter description.
2.10.b	DHA	Added CYRF89435 device support.

Version	Originator	Description
2.20	HPHA	1. Fixed problem with saving information for sliders. 2. Updated baseline algorithm to check for negative difference counts. 3. Added a build error message when user attempts to build project without first calling the user module wizard. 4. Added GetSnsParasiticCapacitance to User Module API; added FMEA_Cp_Range_Test parameter. 5. Optimized Start User Module function code; corrected large memory usage in Start function. 6. Added CY7C69000 support. 7. PRS reset added to ScanSensor API to eliminate high noise. 8. Corrected wGetCentroidPos function to return valid value when sensor is not touched.

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 © 2008-2015 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.