Autotuning CapSense® Sigma-Delta Datasheet CSDAUTO V 1.0

Copyright © 2009-2014 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks				API Memory		Pins (per External IO)
	CapSense	I²C/SPI	Timer	Comparator	Flash	RAM	
CY8C20x66, CY8C20x36, CY8C20336AN, CY8C20436AN, CY8C20636AN, CY8C20x46, CY8C20x96, CY8C20xx6AS, CY7C645xx, CY7C643/4/5xx, CY7C60424, CY7C6053x, CYONS2110, CYONS21L1T, CYONSFN2162							
	1	-	1	1	1540	35	0

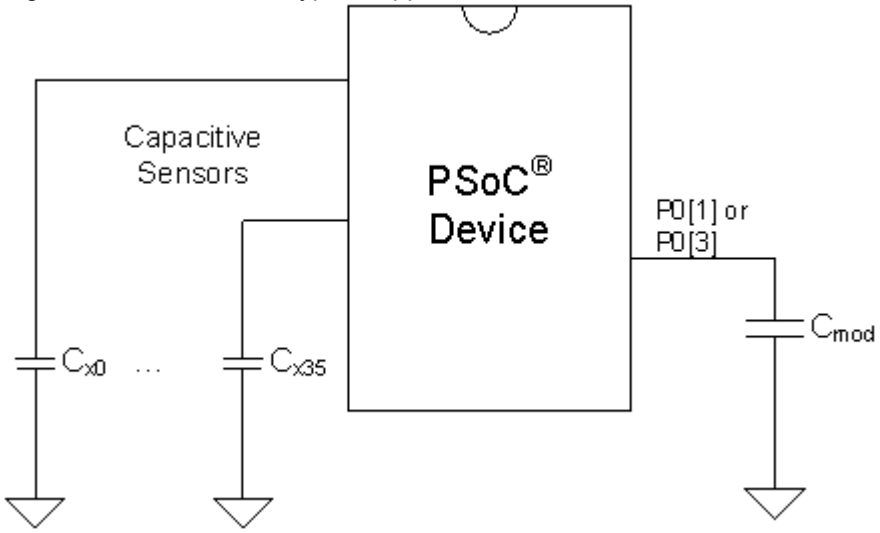
Features and Overview

- AutoTuning algorithms optimize the operational parameters at runtime based on the parasitic capacitance of each sensor.
- Scans 1 to 36 capacitive sensors.
- Capable of detecting 0.1 pF touch with parasitic sensor capacitance (Cp) up to 50 pF, as long as the layout guidelines given in the application note “Capacitance Sensing - Layout Guidelines for PSoC CapSense”, AN2292, are followed.
- Sensing possible through up to a 15 mm glass overlay.
- High immunity to AC mains noise, other EMI, and power supply voltage changes.
- Supports capacitive sensors configured as independent buttons, proximity sensors, and/or as dependent arrays to form sliders. Sliders and proximity sensors not fully supported in this beta version.
- Effective number of slider elements can double the number of dedicated IO pins using diplexing technique.
- Supports slider resolution greater than physical pitch through interpolation.
- Touchpads can be implemented as pairs of interwoven orthogonal sliders.
- Shield electrode provides for reliable operation with high parasitic capacitance and/or in the presence of water film.
- Guided sensor and pin assignments using the CSDAUTO Wizard.
- PC GUI application support for raw data monitoring in real-time.

The CSDAUTO (Autotuning CapSense® using a Sigma-Delta Modulator) User Module provides capacitance sensing using the switched capacitor technique with a sigma-delta modulator to convert the sensor capacitance into digital code.

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

Figure 1. CSDAUTO Typical Application



Functional Description

The capacitive sensor consists of physical, electrical, and software components:

- Physical
  - The physical sensor itself, typically a conductive pattern constructed on a PCB connected to the PSoC with an insulating cover, a flexible membrane, or a transparent overlay over a display.
- Electrical
  - A method to convert the sensor capacitance to digital format. The conversion system consists of a sensing switched capacitor, a sigma-delta modulator, and a counter-based digital filter to convert the modulator output bit stream to a readable digital format.
- Software
  - Detection and compensation software algorithms convert the count value into a sensor detection decision.
  - In the case of consecutive, dependent sensors (sliders and touchpads, for example) APIs are provided to interpolate a position with greater resolution than the physical pitch of the sensors. For example, you can create a volume slider with 10 sensors and use the provided firmware to expand the number of volume levels to 100. Alternatively, using the same APIs, you can use two capacitive sensors that taper into each other and determine the position of a conductive object (such as a finger) between them.
  - High level decision logic provides compensation for environmental factors, such as temperature, humidity, and power supply voltage change. A separate shield electrode can be used for shielding the sensor array to reduce stray capacitance, providing more reliable operation in the presence of a water film or droplets.
  - 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 IO consumed for a given number of slider elements.

The following documents are recommended reading before implementing a CapSense design using the CSDAUTO user module:

- The *PSoC® CY8C20x66, CY8C20x66A, CY8C20x46/96, CY8C20x46A/96A, CY8C20x36, CY8C20x36A, CY8CTMG20x, CY8CTMG20xA, CY8CTST200, CY8CTST200A Technical Reference Manual* (TRM) sections:
  - CapSense System

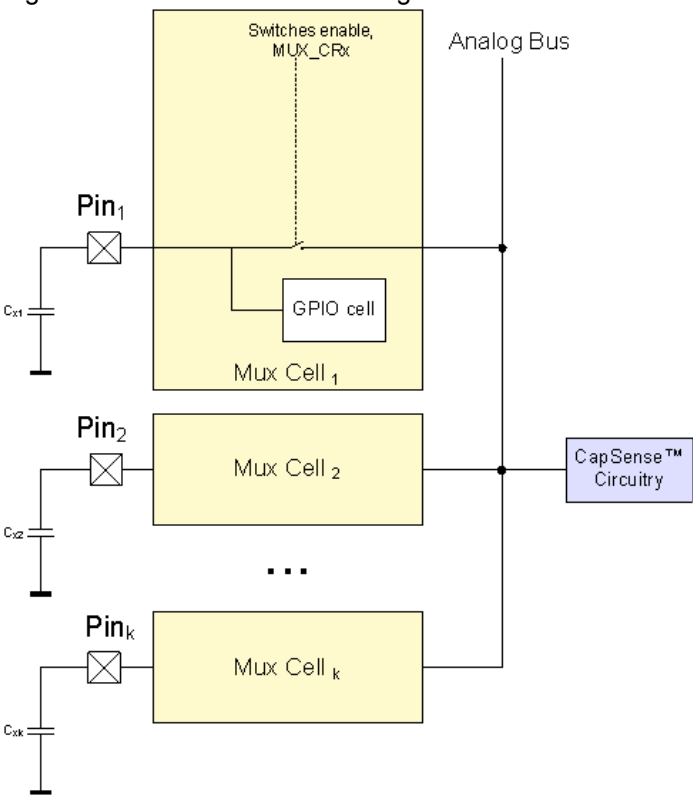
The following application notes are recommended after reading the CSD User Module documentation. Application notes can be found on the Cypress Semiconductor web site at www.cypress.com:

- *CapSense Best Practices* – AN2394
- *Signal-to-Noise Ratio Requirements for CapSense Applications* – AN2403
- *Design Aids - CapSense Data Viewing Tool* – AN2397
- *EMC Design Considerations for PSoC CapSense Applications* – AN2318
- *Power and Sleep Considerations* – AN2360
- *Layout Guidelines for PSoC CapSense* – AN2292
- *Software Implementation of a Universal Asynchronous Transmitter* – AN2399
- *Waterproof Capacitance Sensing* – AN2398

Capacitance Sensing Operation

CY8C20x66 family of devices have an Analog MUX Bus. It allows connection of the CapSense Circuitry to any PSoC pin. The CSDAUTO user module connects the active sensor to the analog Mux bus so the always-connected CapSense circuitry can measure the capacitance of the sensor and translate that capacitance into a digital code. The firmware performs sensor scanning in series by setting corresponding bits in the MUX_CRx registers.

Figure 2. CSDAUTO Block Diagram

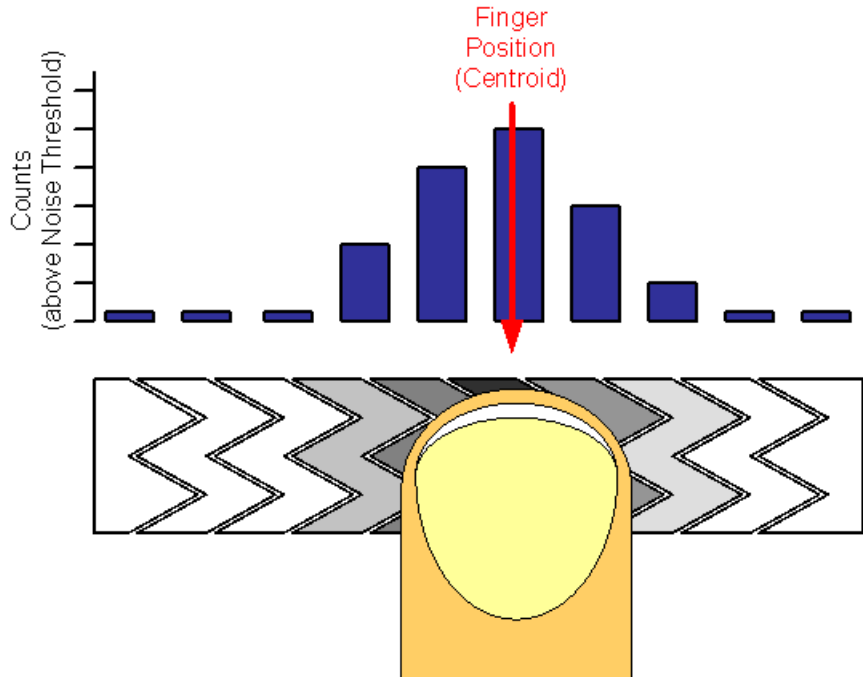


Sliders

Note Sliders and proximity sensors are not fully supported in this beta version.

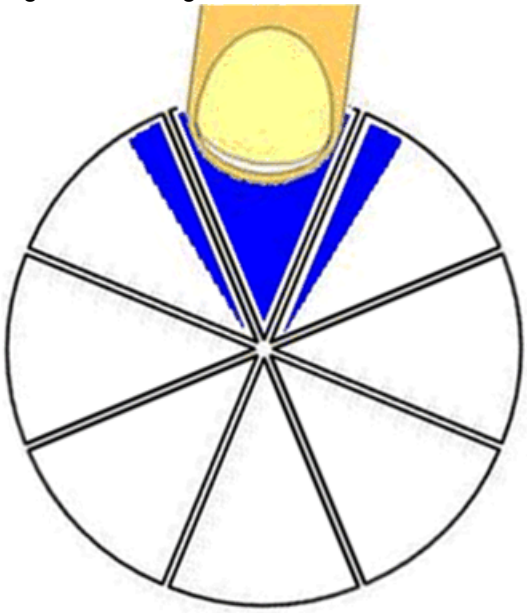
Sliders are used for controls requiring gradual adjustments. Examples include a lighting controls (dimmer), volume controls, graphic equalizers, and speed controls. The sensors that make up a slider are adjacent to one another. Actuation of one sensor results in partial actuation of physically adjacent sensors. The actual position in the slider is found by computing the centroid location of the set of activated sensors. Sliders are accommodated in the CSDAUTO Wizard, by establishing groups in which each group of sliders has a specific order. The practical lower limit number for sensors slider is five, the upper limit is simply the number of sensor positions available on the PSoC device selected.

Figure 3. Interpolated Centroid Position of a Finger on a Slider



Radial Sliders

Figure 4. Finger touches Radial Slider



For the CSDAUTO user module, two slider types are available: linear and radial. Radial sliders are similar to linear ones. While linear sliders have a beginning and an end, radial sliders do not. When a touch happens, the centroid calculation algorithm takes into account sensor counts of the switches to the right and left of the current switch. Radial sliders are not diplexed.

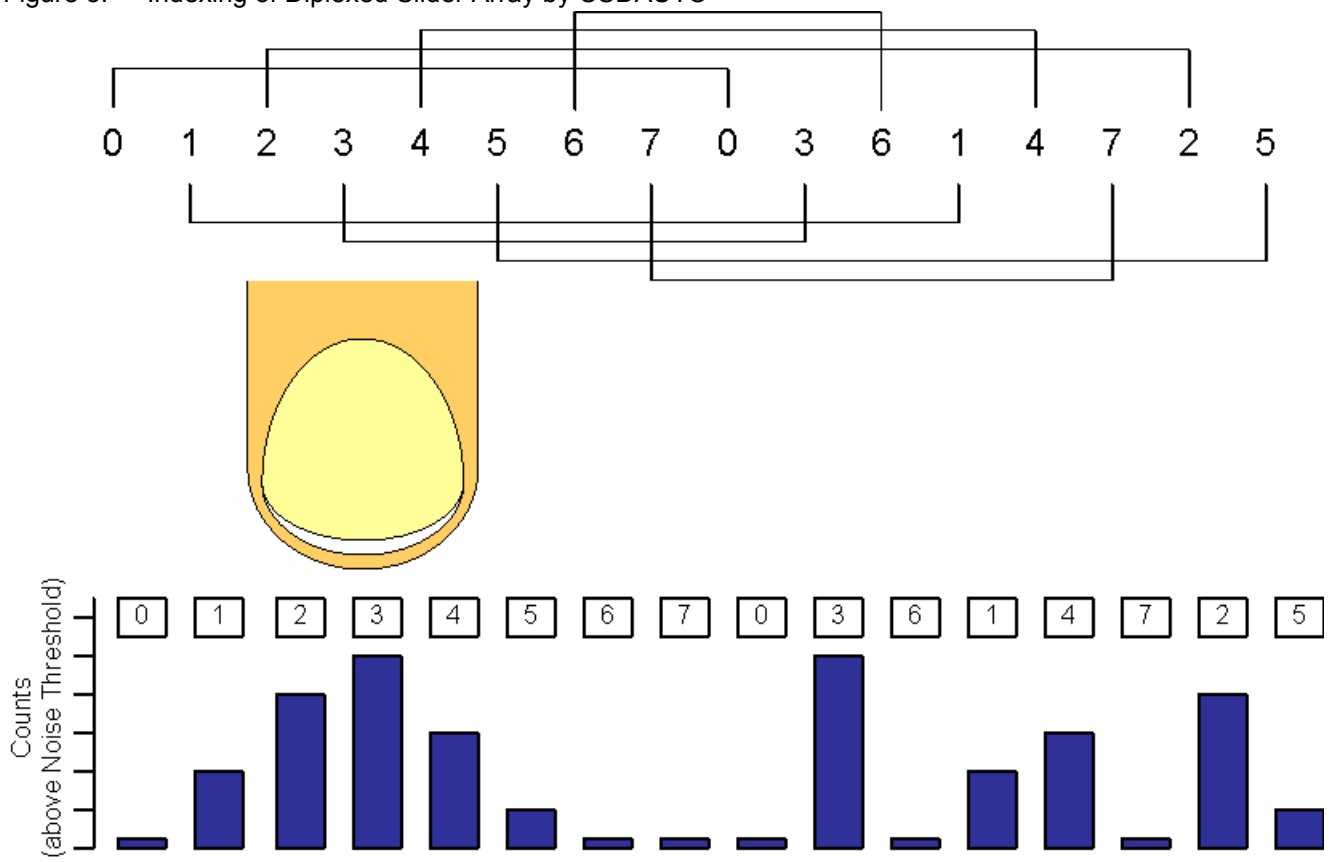
The CSDAUTO user module has two API functions that support radial sliders. The first function CSDAUTO_wGetRadiaPos() returns centroid location and the second CSDAUTO_wGetRadialInc() returns finger shift in resolution units. When the finger moves in a clockwise direction it is a positive offset.

The reference point(0) is located in the middle of the first sensor. The Resolution for both linear and radial sliders is limited and is 3000.

Diplexing

When Diplexing is used, each PSoC sensor connection in a slider is mapped to two physical locations in the array of slider sensors. The first (or numerically lower) half of the physical locations is mapped sequentially to the base assigned sensors, with the port pin assigned by the designer using the CSDAUTO Wizard. The second (or upper) half of the physical sensor locations is automatically mapped by an algorithm as shown in the following figure.

Figure 5. Indexing of Diplexed Slider Array by CSDAUTO



The close proximity of strong signals in one half of the slider results in the same levels aliased into the upper half, but the results are scattered. The sensing algorithms search for strong adjacent sets of signals to declare the resolved slider position. The order is established so that adjacent sensor actuation in one half does not result in adjacent sensor actuation in the other half.

Ensure that the mapping of sensors to pins on the printed circuit board matches the Index by 3 pattern used by the user module. The capacitance of sensor pairs in a diplexed slider should be reasonably well matched (within 10 pF).

The diplex sensor index table is automatically generated by the CSDAUTO Wizard when you select diplexing. This table illustrates the diplexing sequences for different slider segments count:

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

Interpolation and Scaling

In applications for sliding sensors and touchpads it is often necessary to determine finger (or other capacitive object) position to more resolution than the native pitch of the individual sensors. The contact area of a finger on a sliding sensor or a touchpad is often larger than any single sensor.

To calculate the interpolated position using a centroid, the array is first scanned to verify that a given sensor location is valid. The requirement is for some number of adjacent sensor signals to be above a noise threshold. When the strongest signal is found, this signal and those contiguous signals larger than the noise threshold are used to compute a centroid. As few as two and as many as (typically) eight sensors are used to calculate the centroid in the form of:

Equation 1

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

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

Slider sensor count and resolution are set in the CSDAUTO Wizard. A scaling value is calculated by the wizard and stored as fractional values.

The multiplier for the centroid resolution is contained in three bytes with these bit definitions:

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

The resolution is found by using this equation:

Resolution = (Number of Sensors – 1) × Multiplier

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

External Component Selection Guidelines

CSDAUTO can use internal or external modulation capacitor, C_{mod}, connected from Vss ground to either P0[1], or P0[3] port pins. The pins are selected by the Pin Selection Wizard setting. The selected pin should not be used for any other purposes.

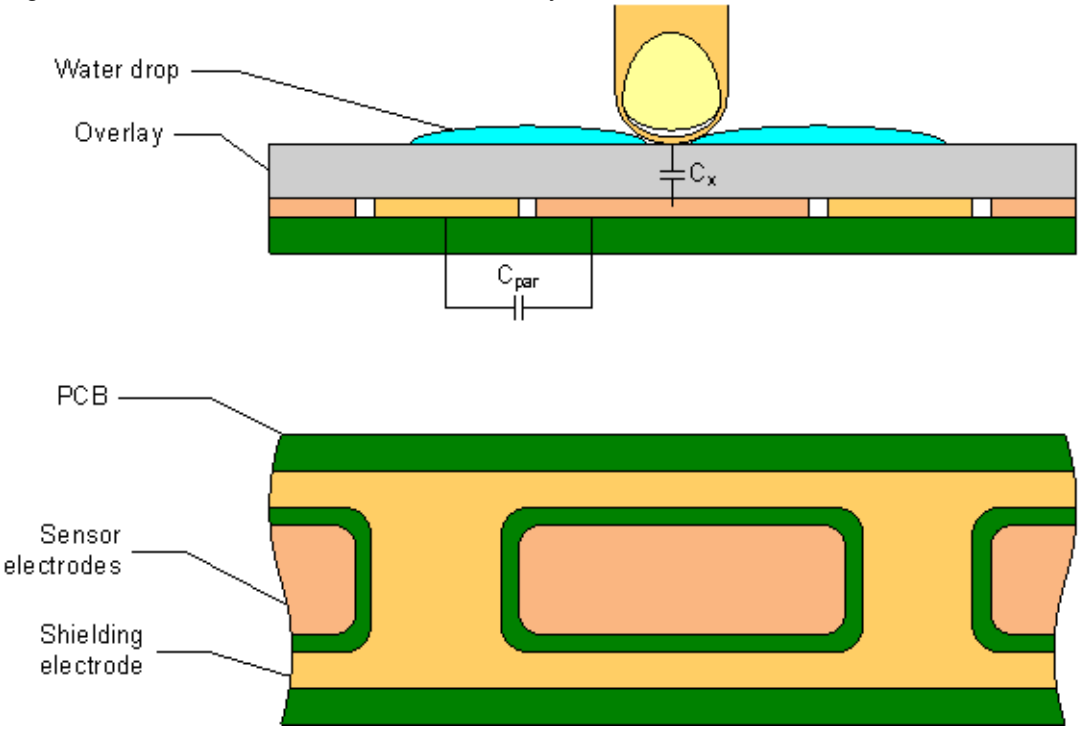
Unless CSDAUTO can work with internal capacitor the external capacitor is recommended. The recommended value for the external modulation capacitor is 2.2 nF. The optimal capacitance can be selected by experiment to get maximum SNR. A value of 2.2 nF gives good results in the most cases. A ceramic capacitor should be used. The temperature capacitance coefficient is not important.

Shield Electrode

Some applications require reliable operation in the presence of water films or droplets. White goods, automotive applications, various industrial applications, and others need capacitive sensors that do not provide false triggering because of water, ice, and humidity changes. In this case a separate shielding electrode can be used. This electrode is located behind or outside the sensing electrode. When water films are located on the insulation overlay surface, the coupling between the shielding and sensing electrodes is increased. The shielding electrode allows you to reduce the influence of parasitic capacitance, which gives you more dynamic range for processing sense capacitance changes.

In some applications it is useful to select the shielding electrode signal and its placement relative to the sensing electrode such that increasing the coupling between these electrodes causes the opposite of the touch change of the sensing electrode capacitance measurement. This simplifies the high-level software API work. The CSDAUTO User Module supports separate output for the shielding electrode.

Figure 6. Possible Shield Electrode PCB Layout



The previous figure illustrates one possible layout configuration for the button's shield electrode. In this example, the button is covered by a shielding electrode plane. 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 in this case.

When water drops are located between the shielding and sensing electrodes, the Cpar is increased and modulator current can be reduced. In practical tests, the modulator reference voltage can be increased by the API so that the raw count increase from water drops should be close to zero or be slightly negative. You can achieve this by selecting the appropriate modulator reference.

CSDAUTO uses the same signal used for the precharge clock to drive the shielding electrode.

The shield electrode can be connected to dedicated PSoC pins (P0[7] or P1[2]). The drive mode for selected pin should be set to Strong. A slew limiting resistor can be connected between the PSoC device and the shielding electrode to reduced emitted EMI.

DC and AC Electrical Characteristics

Table 2. Power Supply Requirement

Parameter	Min	Typical	Max	Unit	Test Conditions and Comments
Vdd	1.7	5.0	5.25	V	NA

Table 3. Signal and Noise at Various Cp's

Parasitic Capacitance, Cp (pF)	Signal for 0.1 pF Touch (Counts)	Noise[a] (Counts)	SNR	Scan Time (µs)
10				
15				
20				

a. This noise value is a typical value based on a reference design following the guidelines of AN2292.

Placement

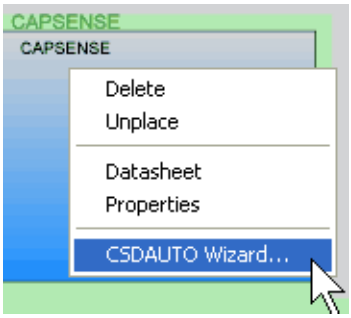
The blocks for the user module are automatically placed when the user module is instantiated, alternate placements are not available. The CSDAUTO User Module consumes the CapSense block and one Timer (Timer1).

User modules that require specific pin resources, including the LCD and I2CHW, must be placed before starting the CSDAUTO Wizard to establish pin connections for the CSDAUTO User Module.

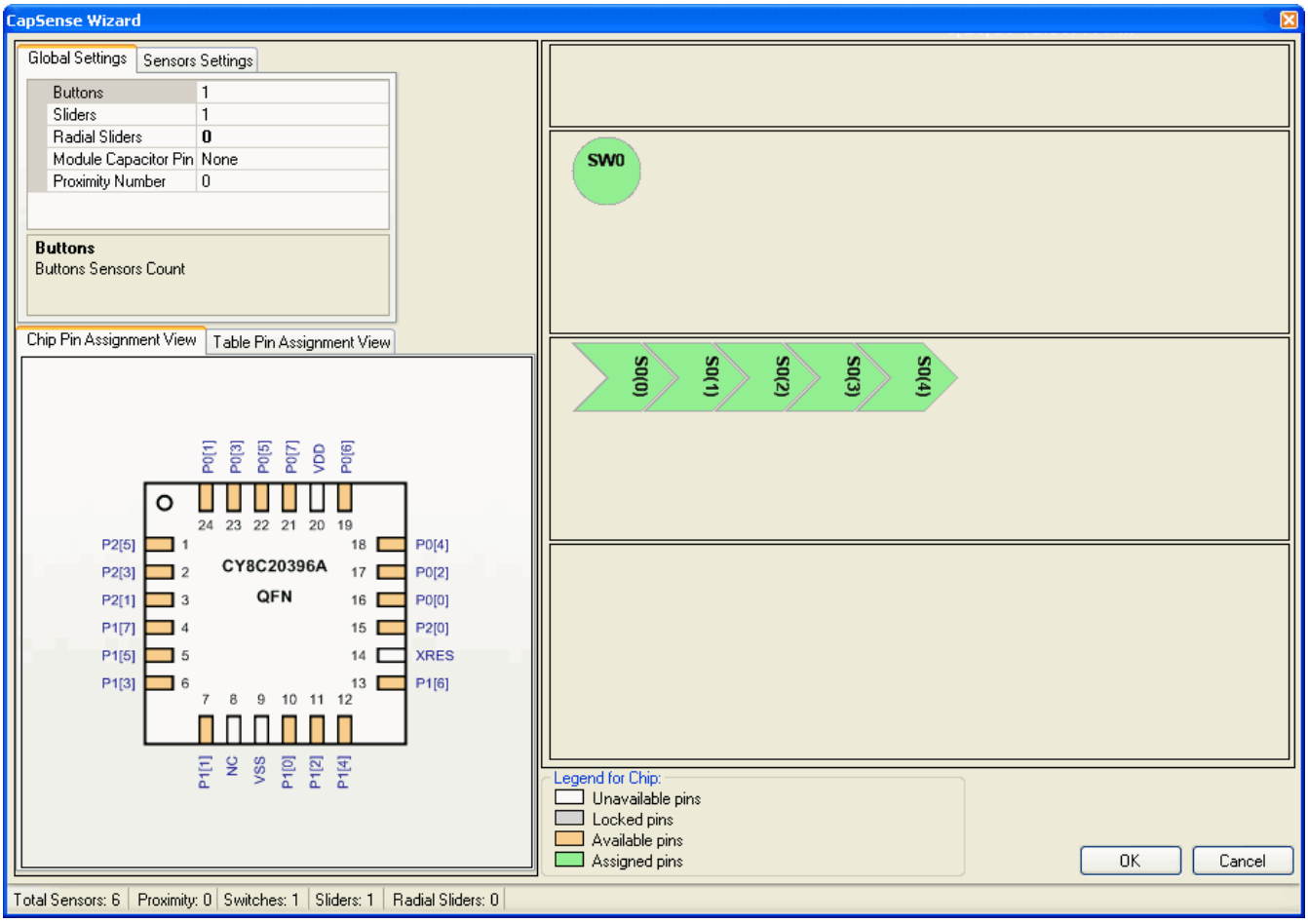
Avoid P1[0] and P1[1] when placing capacitive sensor connections. These pins are used for programming the part and may have excess routing capacitance affecting sensor sensitivity and noise.

Wizard Access

1. To access the Wizard, right click any block of the CSDAUTO in the Device Editor Interconnect View, then select the CSDAUTO Wizard with a left mouse click.



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



Wizard Pin Legend

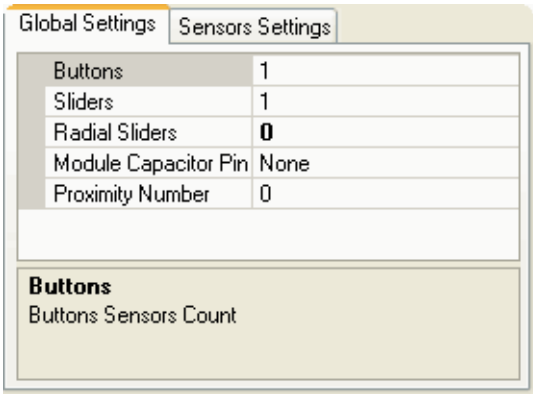
White – The pin can not be used as a CapSense input.

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.

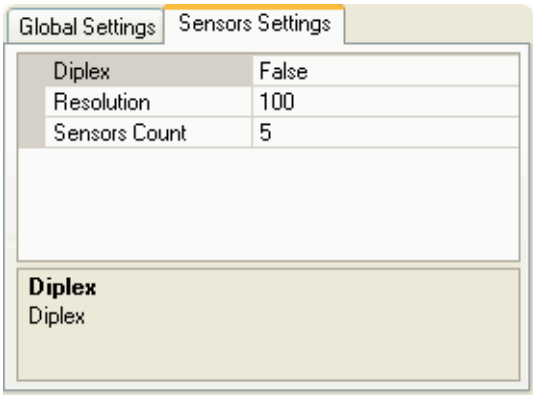
Orange – The pin is available for assignment.

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

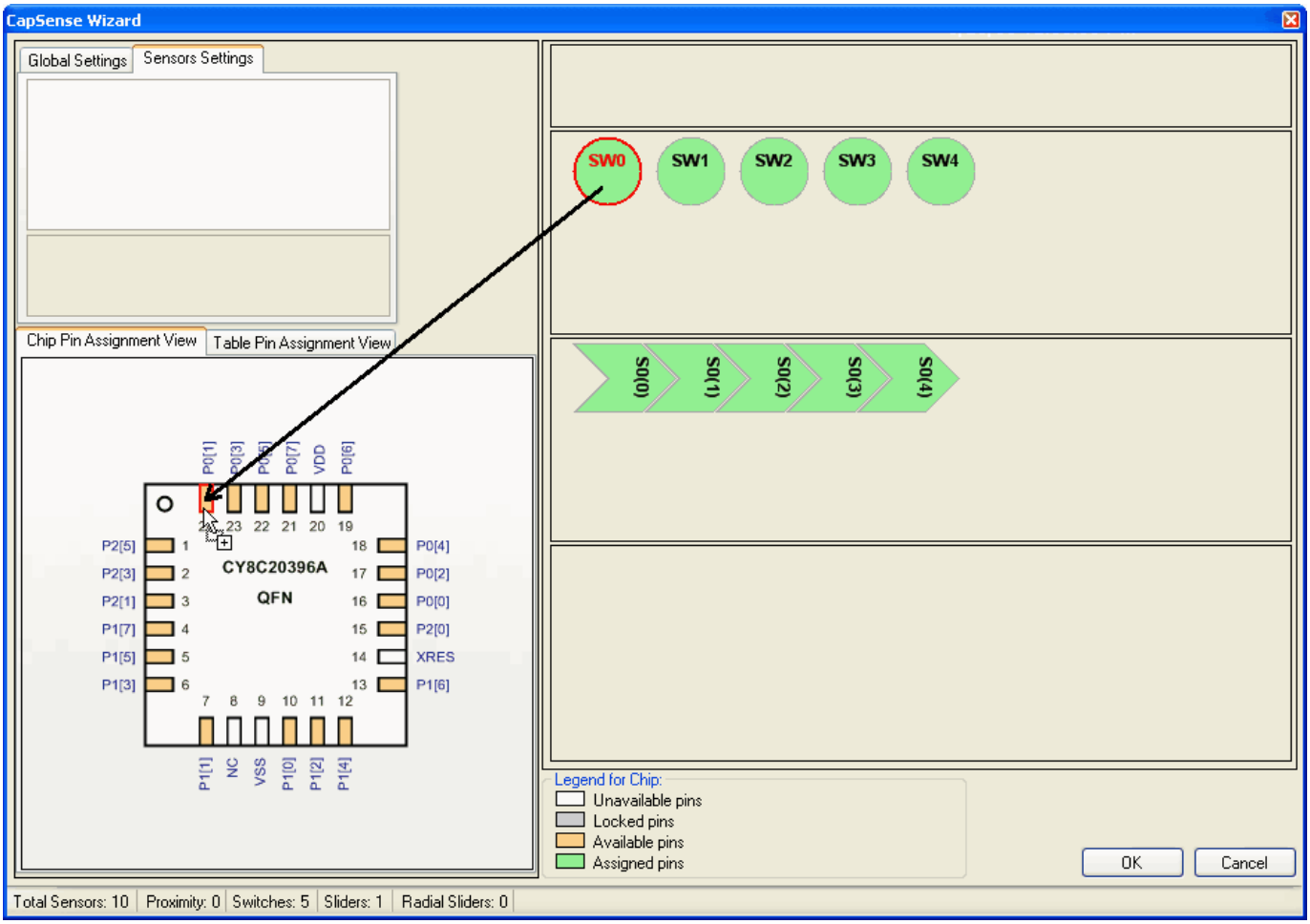
3. Type the number of independent buttons, sliders, and radial sliders. The total number of sensors (buttons plus slider elements) is limited to the number of pins avail-able. After entering the data, press the [Enter] key to update the display with the new value. X-Y touchpads require two sliders.



4. Select Sensor Settings to set the settings for sliders and radial sliders. To alter settings, click one of your sliders to activate it. Type the number of sensor elements in each slider. The practical minimum number of sensors in a slider sensor is five, the maximum is limited by pin count. After entering the data, press the [Enter] key to update the display. Select modulator capacitor (C_{mod}) pin. Choose from the available pins P0[1], P0[3]. If the internal capacitor is used select None. You usually get better SNR with an external 2.2 nF capacitor.

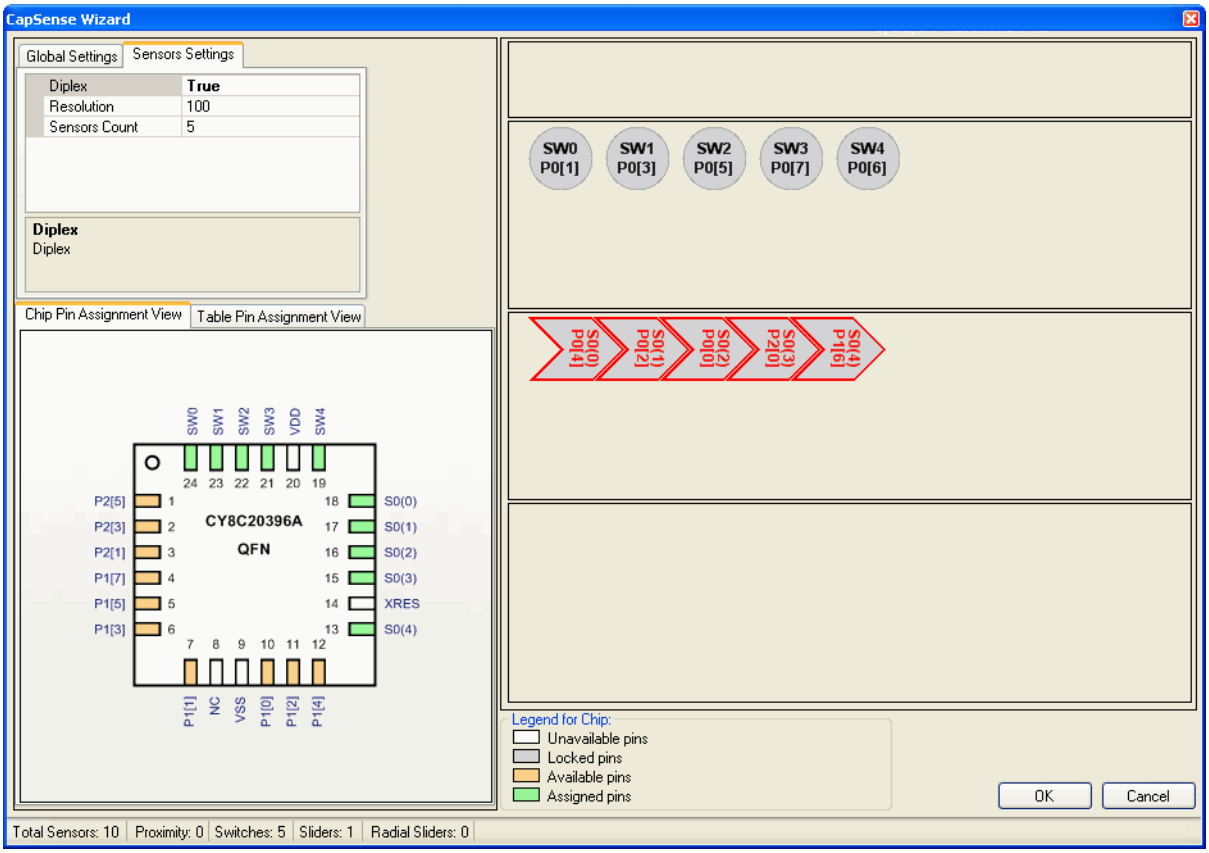


5. Type the output resolution. The minimum value is five. The maximum value is (# of pins used for sensors – 1) × 2_{16} – 1 or (2 x pins used for sensors – 1) × 2_{16} – 1 for diplexed sliders. The software attempts to interpolate the touch results to the specified resolution using the relative strength of adjacent segments. The software reports touch results on the slider between zero and the resolution – 1.
6. 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.
7. Assign switches or sensors to pins by dragging the switch or sensor onto the pin in the Pin Assignment View. You can choose to drag switches or sensors onto pins in the Chip Pin Assignment View or the Table Pin Assignment View. The port 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.



8. Repeat for the remainder of independent sensors. Mapping of individual slider sensors onto physical port pins is the same as for individual 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.



Set user module parameters and generate the application. You can adapt a sample project now, if you wish.

If you want change 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. The tables are located in CSDAUTO_Table.asm

Sensor Table

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

```
CSDAUTO_Sensor_Table:
_CSDAUTO_Sensor_Table:
      dw    0x0140  //  Port 1 Bit 6
```

```
      dw    0x0301  //  Port 3 Bit 0
      dw    0x0304  //  Port 3 Bit 2
      dw    0x0308  //  Port 3 Bit 3
      dw    0x0302  //  Port 3 Bit 1
      dw    0x0108  //  Port 1 Bit 3
```

This table is used by CSDAUTO_wGetPortPin() routine.

Group Table

The Group Table defines each of the groups of button sensors or sliders. There is one entry for each slider plus one for the free button sensors. The first entry is always the free sensors. Each entry is six bytes. The first byte is the index in the Sensor Table where the group starts. The second byte is how many sensors are in that group. The third byte signifies whether the slider is diplexed or not (4 is diplexed, 0 is not diplexed). The fourth, fifth, and sixth bytes are the fixed point multiplier that the slider's calculated centroid is multiplied by to achieve the resolution desired in the CSDAUTO wizard.

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

Diplex Table

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

```
DiplexTable_0:
; This group is not a diplexed slider
DiplexTable_1:
 db 0,1,2,3,4,5,6,7,0,3,6,1,4,7,2,5// 8 switch slider


 CSDAUTO_Diplex_Table:
_CSDAUTO_Diplex_Table:
 db >DiplexTable_0, <DiplexTable_0
 db >DiplexTable_1, <DiplexTable_1
```

Parameters and Resources

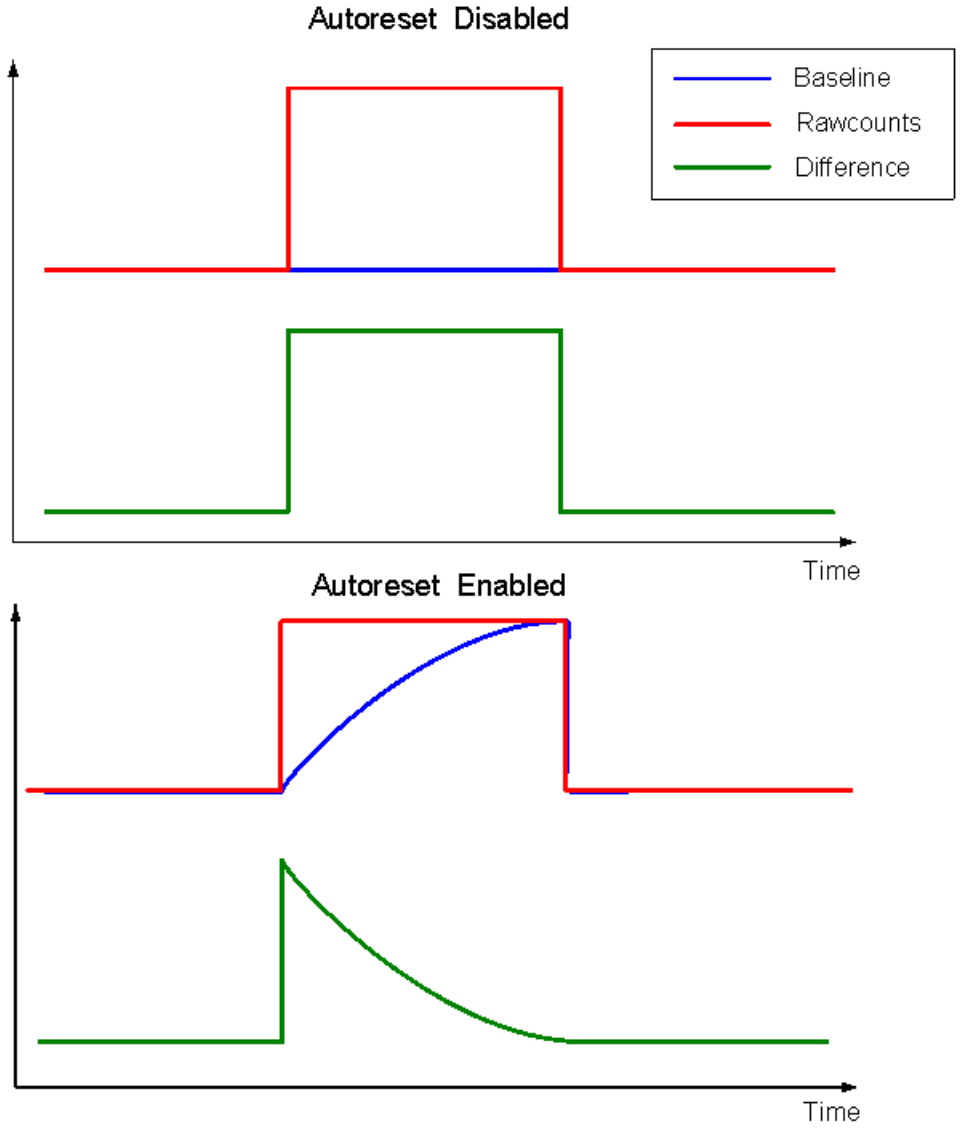
Sensors Autoreset

This parameter determines whether the baseline is updated at all times or only when the signal difference is below the Noise Threshold. When set to **Enabled** the baseline is updated constantly. This setting limits the maximum time duration of the sensor (typical values are 5 – 10s), but it prevents the sensors from permanently turning on when the raw count suddenly rises without anything touching the sensor. This sudden rise can be caused by a large power supply voltage fluctuation, a high energy RF noise source, or a very quick temperature change.

When the parameter is set to **Disabled** the baseline is updated only when raw count and baseline difference is below the Noise Threshold parameter. You should leave this parameter Disabled unless you have problems with sensors permanently turning on when the raw count suddenly rises without anything touching the sensor.

The following figure illustrates this parameter's influence on the baseline update.

Figure 7. The Sensor Autoreset Parameter



Debounce

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

Possible values are 1 to 255. A setting of 1 provides no debouncing.

ShieldElectrodeOut

The shielding electrode signal source can be routed to P0[7] or P1[2].

Application Programming Interface

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

Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns the CSDAUTO_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 CSDAUTO 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 function may leave A and X unchanged, there is no guarantee they may do so in the future.

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

- CSDAUTO_waSnsBaseline[]
- CSDAUTO_waSnsResult[]
- CSDAUTO_waSnsDiff[]
- CSDAUTO_baSnsOnMask[]

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

- CSDAUTO_UpdateAllBaselines();
- CSDAUTO_UpdateSensorBaseline();
- CSDAUTO_InitializeBaselines().

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

- CSDAUTO_ScanSensor();
- CSDAUTO_ScanAllSensors().

CSDAUTO_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.

CSDAUTO_baSnsOnMask[] – This is a byte array that holds the sensor on or off state (for buttons or sliders). CSDAUTO_baSnsOnMask[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). CSDAUTO_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

CSDAUTO_baSnsOnMask[] data is updated by CSDAUTO_bIsSensorActive(BYTE bSensnor) function or CSDAUTO_bIsAnySensorActive() routines.

CSDAUTO_Start

Description:

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

C Prototype:

```
void  CSDAUTO_Start()
```

Assembly:

```
lcall  CSDAUTO_Start
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDAUTO_Stop

Description:

Stops the sensor scanner, disables internal interrupts, and calls CSDAUTO_ClearSensors() to reset all sen-sors to an inactive state.

C Prototype:

```
void  CSDAUTO_Stop()
```

Assembly:

```
lcall  CSDAUTO_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDAUTO_ScanSensor

Description:

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

C Prototype:

```
void CSDAUTO_ScanSensor(BYTE bSensor);
```

Assembly:

```
mov A,  bSensor
lcall CSDAUTO_ScanSensor
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects

**

CSDAUTO_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 CSDAUTO_UpdateSensorBaseline() is called, a difference count is calculated by subtracting the previous baseline from the raw count value. This difference is stored in the CSDAUTO_waSnsDiff[] array and is provided to you.
2. If Sensors Autoreset is disabled, each time CSDAUTO_UpdateSensorBaseline() is called the difference count is compared to the noise threshold. If the difference is below the noise threshold, it is accumu-lated 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. Once the accumulated difference counts in the virtual bucket has reached the BaselineUpdateTh-reshold, 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 CSDAUTO_UpdateSensorBaseline(BYTE bSensor)
```

Assembly:

```
mov   A,   bSensor
lcall  CSDAUTO_UpdateSensorBaseline
```

Parameter:

A => Sensor Number

Return Value:

None

Side Effects:

**

CSDAUTO_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 CSDAUTO_baSnsOnMask[] array.

C Prototype:

```
BYTE  CSDAUTO_bIsSensorActive(BYTE bSensor)
```

Assembly:

```
mov  A,  bSensor
lcall CSDAUTO_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:

**

CSDAUTO_bIsAnySensorActive

Description:

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

C Prototype:

```
BYTE  CSDAUTO_bIsAnySensorActive()
```

Assembly:

```
lcall CSDAUTO_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:

**

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

C Prototype:

```
WORD  CSDAUTO_wGetCentroidPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall CSDAUTO_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 CSDAUTO Wizard. If no sensors are active, the function returns –1 (FFFFh). If an error occurs during execu-tion of the centroid/diplexing algorithm, the function returns –1 (FFFFh). You can use the CSDAUTO_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.

CSDAUTO_wGetRadialPos

Description:

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

C Prototype:

```
WORD  CSDAUTO_wGetRadialPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall CSDAUTO_wGetRadialPos
```

Parameters:

bSnsGroup A => Group Number

This parameter is a number of radial slider you are working with. You can get its number through CSDAUTO 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.

CSDAUTO_wGetRadialInc

Description:

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

C Prototype:

```
WORD  CSDAUTO_wGetRadialInc(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall CSDAUTO_wGetRadialInc
```

Parameters:

bSnsGroup A => Group Number

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

Side Effects:

The routine should be called only after CSDAUTO_wGetRadialPos() API. Because it uses internal data CSDAUTO_waSliderPrevPos and CSDAUTO_waSliderCurrPos that are set by the CSDAUTO_wGetRadialPos().

CSDAUTO_InitializeSensorBaseline

Description:

Loads the CSDAUTO_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 CSDAUTO_InitializeSensorBaseline(BYTE bSensor)
```

Assembly:

```
mov A, bSensor
lcall CSDAUTO_InitializeSensorBaseline
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

CSDAUTO_InitializeBaselines

Description:

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

C Prototype:

```
void CSDAUTO_InitializeBaselines()
```

Assembly:

```
lcall CSDAUTO_InitializeBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDAUTO_SetScanMode

Description:

Sets scanning speed and resolution. This function can be called at runtime to change the scanning speed and resolution. 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 res-olution. 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 CSDAUTO_ScanSensor() func-tion.

C Prototype:

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

Assembly:

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

Parameters:

bSpeed, bResolution

Return Value:

None

Side Effects:

**

CSDAUTO_SetIdacValue

Description:

This function sets the iDAC current value. Use it if some sensors need to be scanned with different iDAC setting. This function can be used in conjunction with CSDAUTO_ScanSensor().

C Prototype:

```
void CSDAUTO_SetIdacValue(BYTE bRefValue);
```

Assembly:

```
mov     A, bIdacValue
lcall    CSDAUTO_SetIdacValue
```

Parameters:

bIdacValue – Sets the iDAC value. Accepted values are 1.. 255.

Return Value:

None

Side Effects:

**

CSDAUTO_SetPrescaler

Description:

This function sets the Prescaler value. Use it if some sensors need to be scanned with Prescaler setting. This function can be used in conjunction with CSDAUTO_ScanSensor().

C Prototype:

```
void CSDAUTO_SetPrescaler(BYTE bPrescaler);
```

Assembly:

```
mov     A, bPrescaler
lcall    CSDAUTO_SetPrescaler
```

Parameters:

bPrescaler – Sets the Prescaler value. Accepted values are listed in the following table:

Name	Value	Prescaler
CSDAUTO_PRESCALER_1	0x00	1
CSDAUTO_PRESCALER_2	0x01	2
CSDAUTO_PRESCALER_4	0x02	4
CSDAUTO_PRESCALER_8	0x03	8
CSDAUTO_PRESCALER_16	0x04	16
CSDAUTO_PRESCALER_32	0x05	32
CSDAUTO_PRESCALER_64	0x06	64
CSDAUTO_PRESCALER_128	0x07	128
CSDAUTO_PRESCALER_256	0x08	256

Return Value:

None

Side Effects:

**

CSDAUTO_ClearSensors

Description:

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

C Prototype:

```
void  CSDAUTO_ClearSensors()
```

Assembly:

```
lcall  CSDAUTO_ClearSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSDAUTO_wReadSensor

Description:

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

C Prototype:

```
WORD  CSDAUTO_wReadSensor(BYTE bSensor)
```

Assembly:

```
mov A, bSensor
lcall CSDAUTO_wReadSensor
```

Parameters:

A => Sensor Number

Return Value:

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

Side Effects:

**

CSDAUTO_wGetPortPin

Description:

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

C Prototype:

```
WORD  CSDAUTO_wGetPortPin(BYTE bSensorNum)
```

Assembly:

```
mov  A,  bSensorNumber
lcall     CSDAUTO_wGetPortPin
```

Parameters:

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

Return Value:

A => Sensor Bitmap

X => Port Number

Side Effects:

**

CSDAUTO_EnableSensor

Description:

Configures the selected sensor to measure during the next measurement cycle. The port and sensor can be selected using the CSDAUTO_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 compara-tor function.

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Bitmap

X => Port Number

Return Value:

None

Side Effects:

**

CSDAUTO_DisableSensor

Description:

Disables the sensor selected by the CSDAUTO_wGetPortPin() function. The drive mode is changed to Strong (001) and set to zero. This effectively grounds the sensor. The connection from the port pin to the AnalogMuxBus is turned off. The function parameters are returned by CSDAUTO_wGetPortPin() func-tion.

C Prototype:

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

Assembly:

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

Parameters:

A => Sensor Bitmap

X => Port Number

Return Value:

None

Side Effects:

**

Sample Firmware Source Code

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 CSDAUTO module
// Scanning all sensors continuously
//-------------------------------------------------------------------------

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

void main(void)
{
  BYTE bIndex;

  M8C_EnableGInt;
  CSDAUTO_Start();
 CSDAUTO_InitializeBaselines() ; //Scan all sensors first time, init baseline

 while (1) {     //Loop forever
```

```
    for(bIndex=0; bIndex < CSDAUTO_TotalSensorCount; bIndex++) //Loop through all
sensors
  {
      CSDAUTO_ScanSensor(bIndex);      // Scan Sensors
      CSDAUTO_UpdateSensorBaseline(bIndex);  // Run baseline filter
  }

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

    // now we are ready to send all status variables to chart program
    // communication here
   //
   // OUTPUT CSDAUTO_waSnsResult[x] <- Raw Counts
   // OUTPUT CSDAUTO_waSnsDiff[x] <- Difference
   // OUTPUT CSDAUTO_waSnsBaseline[x] <- Baseline
   // OUTPUT CSDAUTO_baSnsOnMask[x] <- Sensor On/Off
 }
}
```

Configuration Registers

Table 4. Block CapSense, Register: CS_CR0

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

Table 5. Block CapSense, Register: CS_CR1

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

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

Table 6. Block CapSense, Register: CS_CR2

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

Table 7. Block CapSense, Register: CS_CR3

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

Table 8. Block CapSense, Register: CS_CNTH

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

Table 9. Block CapSense, Register: CS_CNTL

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

Table 10. Block CapSense, Register: PRS_CR

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

Appendices

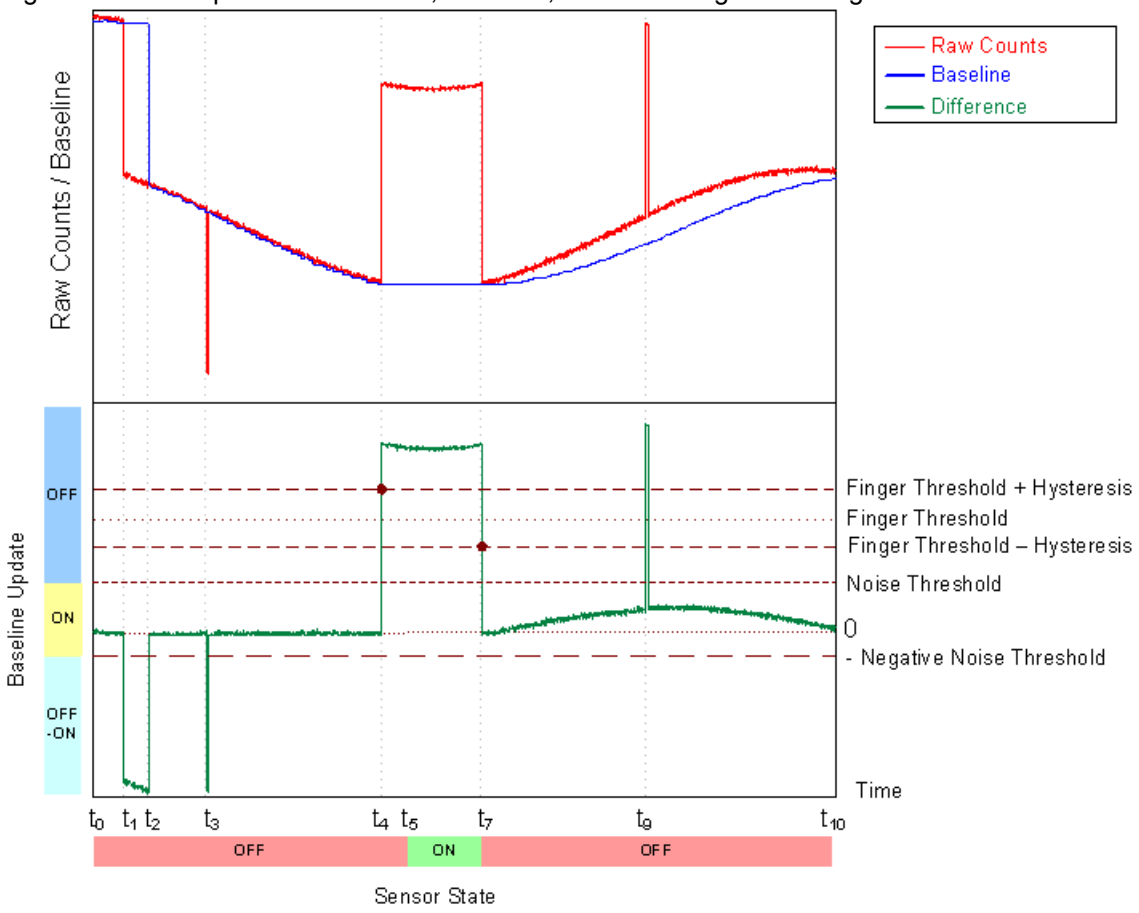
The following sections contain information beyond what is usually included in user module data sheets. The detailed information was developed by Cypress engineers to help you successfully design CapSense applications. Some of this information may be moved into application notes in the future.

Interaction of CSDAUTO Parameters

The following figures illustrate the baseline update and decision logic operation and can be useful for better understanding how to set user module parameters for optimum performance. The first figure illustrates system operation when the Sensors Autoreset parameter is set to **Disabled**. The second 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 8. Example of Raw Counts, Baseline, Difference Signals Change With SensorsAutoreset Set to Disabled



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

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

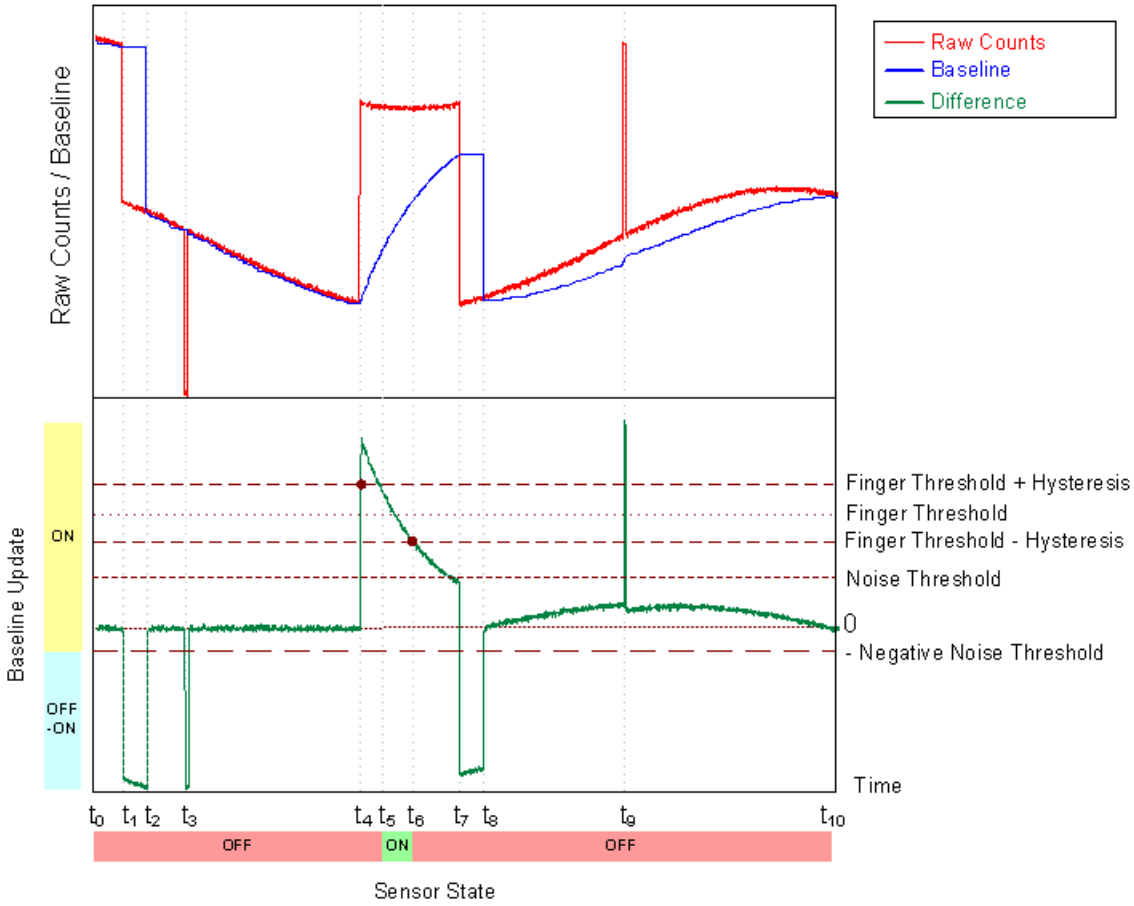
The second large negative difference signal spike happens at t_3, this spike may have been triggered by an ESD event for example. Because the spike duration in the sample count is less than the LowBaselineReset parameter, the baseline is kept on hold and the spike is filtered. This prevents a false baseline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

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

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

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

Figure 9. Example of Raw Counts, Baseline, Difference Signals Change With SensorsAutoreset Set to Enabled



The system operation in the previous figure 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
1.0	DHA	1. Updated Radial Slider Position when no Linear Slider is present. The resolution maximum is now 3000. 2. Removed 0.5 shift and added compensation for negative values. 3. Added capability to add additional slider. 4. Moved ModCap selection from User Module parameter to Wizard Setting section. 5. Added new parameter "Sensor Sensitivity". 6. Added CSD and CSDAUTO support for TMA3xx series chips.

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 © 2009-2014 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.