Dual CapSense® Sigma-Delta Datasheet CSD2X V 2.10

Copyright © 2009-2013 Cypress Semiconductor Corporation. All Rights Reserved.

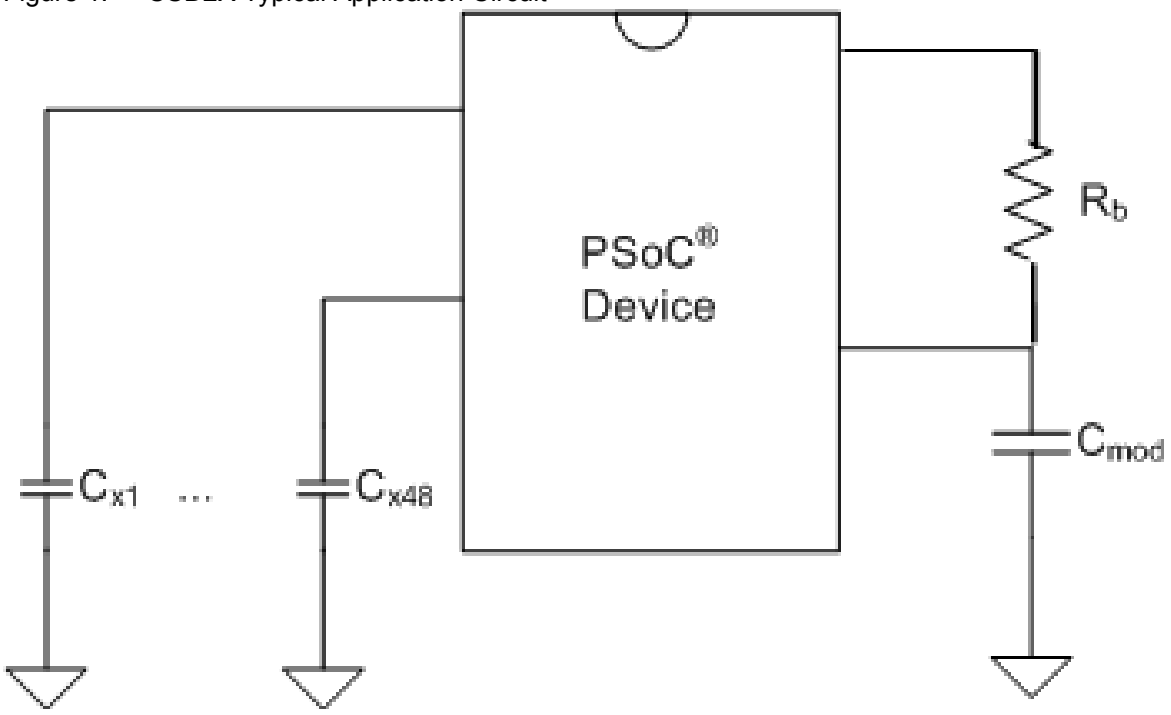
Resources	PSoC® Blocks				API Memory		Pins (per External I/O)
	Decimator	I²C/SPI	Digital	Analog	Flash	RAM	
CY8C28x45, CY8C28x52, CY8C28x13, CY8C28x33							
First Order Modulator with IDAC	2	–	0..2	4	–	–	2
First Order Modulator with Rb resistor	2	–	0..2	4	–	–	4
Second Order Modulator with IDAC	2	–	0..2	4	–	–	2
Second Order Modulator with Rb resistor	2	–	0..2	4	–	–	4

Features and Overview

- Scan up to 41 capacitive sensors (depending on the device pin count)
- Sensing possible with up to a 15 mm glass overlay.
- Proximity detection to 20 cm with a wire-based sensor.
- High immunity to AC mains noise, EMC noise, and power supply voltage changes.
- Supports different combinations of independent and slide capacitive sensors.
- Double slide sensor physical resolution using diplexing.
- Increase slide sensor resolution using interpolation.
- Touchpad support with two slide sensors.
- Sensing support through high resistive conductive materials (ITO films for example).
- Shield electrode support for reliable operation in the presence of water film or droplets.
- Guided sensor and pin assignments using the CSD2X Wizard.
- Integrated baseline update algorithm for handling temperature, humidity, and electrostatic discharge (ESD) events.
- Easily adjustable operational parameters.
- FMEA features such as Short test, Cmod test, and Cmod_Rb test
- Background scan
- PC GUI application support for raw data monitoring and parameter optimization in real-time.

The CSD2X User Module (Capacitive Sensing using a Sigma-Delta Modulator) provides capacitance sensing using the switched capacitor technique with a sigma-delta modulator to convert the sensing switched capacitor current to digital code. The CSD2X User Module can support double-channel CapSense scanning with First and Second order of Sigma Delta Modulator.

Figure 1. CSD2X Typical Application Circuit



Quick Start

1. Select and place user modules requiring dedicated pins (for example, I2C and LCD), if used. Assign ports and pins as required.
2. Select and place the CSD2X User Module.
3. Right-click the CSD2X User Module in the Workspace Explorer to access the CSD2X Wizard (the wizard is explained later in the datasheet).
4. Set the number of sensors, sliders, or rotary sliders that you want.
5. Set the sensor settings for each sensor.
6. Set pins and global parameters. Read all parameter descriptions and follow requirements and guidelines.
7. Generate the application and switch to the Application Editor.
8. Adapt the sample code as required to implement independent sensors, sliding sensors, or a touchpad.
9. Connect the I2C-USB bridge to the target board, and observe the signals.
10. Change the CSD2X parameters to optimize your settings and rebuild the application.
11. Program the PSoC device and verify module operation.Tune the CSD2X parameters to achieve a 5:1 SNR requirement as discussed in the CY8C21x34/B CapSense Design Guide.

See the *Troubleshooting* section in the *Appendices* if you encounter any problems.

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 (such as sliders and touchpads) 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.

While there are a number of methods to measure capacitance, the one used in this user module is combination switching capacitor with a delta-sigma modulator.

The sensor array consists of combinations of independent sensors, sliding sensors, and touchpads implemented as a pair of orthogonal sliders. 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.

The high level software functions accommodate slider diplexing so that a single electrical sensor may be used in two physical locations for resolution enhancement. The functions also provide further interpolation of resolved sensor position between physical sensor locations.

The following documents are recommended reading before you use the CSD2X User Module for the first time.

- *CY8C28X45 and CY8C21345 PSoC Programmable System-on-Chip Technical Reference Manual*, sections
  - CapSense System

The following design guides are recommended after reading the CSD User Module datasheet. These documents are available on the Cypress Semiconductor website at www.cypress.com:

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

Capacitance Measurement Operation

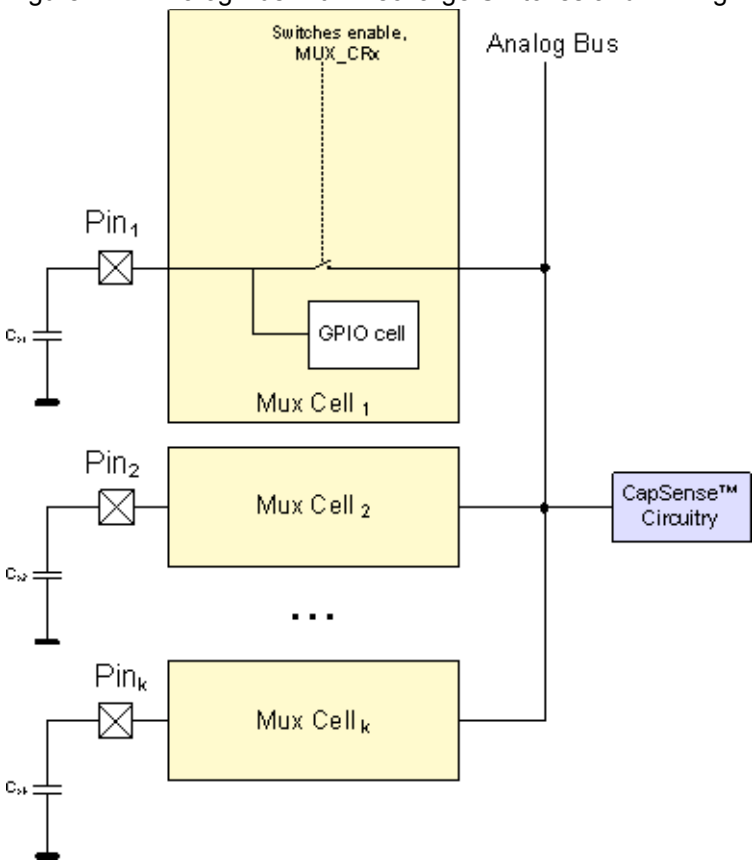
The decision logic is implemented in firmware. The firmware analyzes capacitance measurement, tracks the slow capacitance change due to environmental factors, and runs decision logic to detect button touches and calculate slider position.

Scanning an Array of Sensors

The CY8C28x45 family of devices have a built-in analog bus. It allows capacitive sensor connections to any PSoC pin. The CSD2X User Module uses internal precharge switches to charge active sensors at clock signal phase Ph_1 and connects the Analog Bus to the sensor at phase Ph_2. The sigma-delta modulator modulation capacitor and comparator inputs are connected to the analog bus permanently.

The firmware performs sensor scanning in series by setting corresponding bits in the MUX_CRx registers.

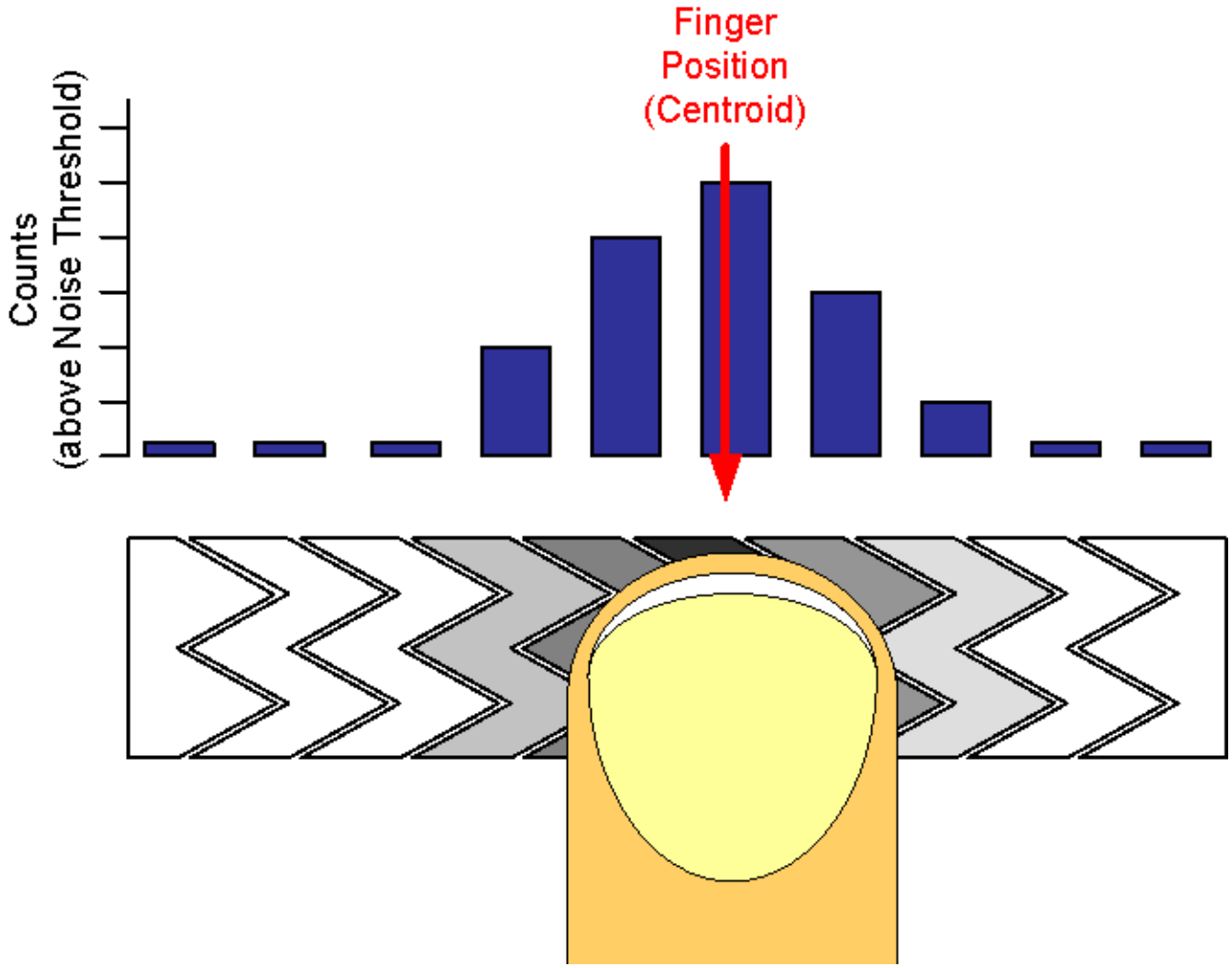
Figure 2. Analog Bus with Precharge Switches and Driving Waveforms



Sliders

Sliders are used for controls requiring gradual adjustments. Examples include a lighting control (dimmer), volume control, graphic equalizer, and speed control. These sensors are mechanically 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 CSD2X 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.

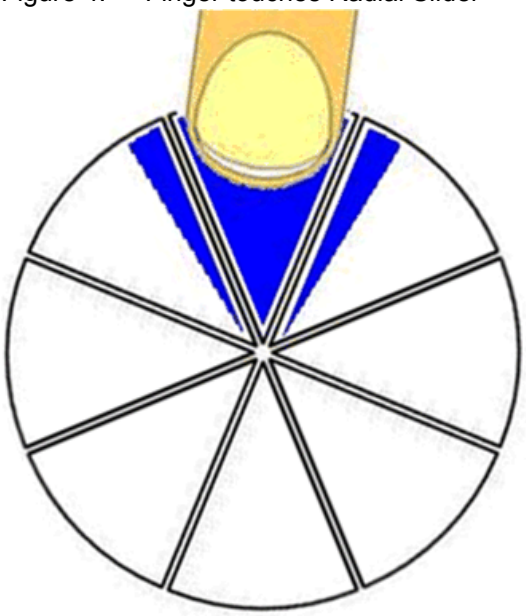
Figure 3. Ordering Physical Sensor Locations



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.

Radial Sliders

Figure 4. Finger touches Radial Slider



CSD2X User Module has two slider types: 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 CSD2X User Module has two API functions that support radial sliders. The first function CSD2X_wGetRadiaPos() returns centroid location and the second CSD2X_wGetRadialInc() returns finger shift in resolution units. When the finger moves in a clockwise direction it is a positive offset.

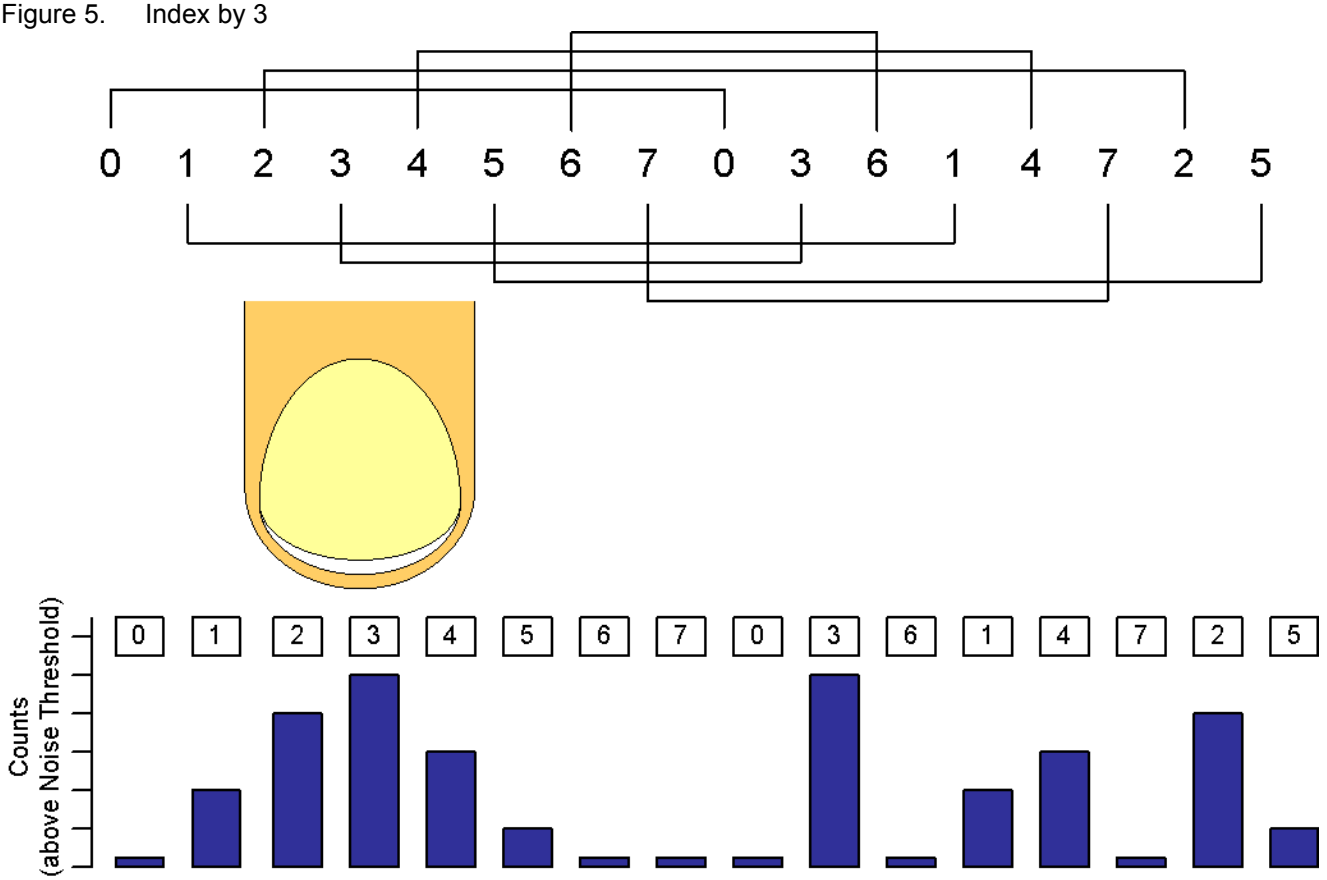
The reference point(0) is located in the middle of the first sensor. The Resolution for both linear and radial sliders is limited and is (number of pins used for sensors - 1) x 2^8 - 1 or (2 x number of pins used for sensors - 1) x 2^8 - 1 for diplexed sliders.

Diplexing

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 CSD2X Wizard. The second (or upper) half of the physical sensor locations is automatically mapped by an algorithm in the Wizard and listed in an include file. The order is established so that adjacent sensor actuation in one half does not result in adjacent sensor actuation in the other half. Exercise care to determine this order and map it onto the printed circuit board.

There are a number of methods to order the second half of the physical sensor locations. The simplest is to index the sensors in the upper half, all of the even sensors, followed by all of the odd sensors. Other methods include indexing by other values. The method selected for this user module is to index by three.

Figure 5. Index by 3



You should balance sensor capacitance in the slider. Depending on sensor or PCB layouts, there may be longer routes for some of the sensor pairs. The diplex sensor index table is automatically generated by the CSD2X 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

Total Slider Segment Count	Segment Sequence
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
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.

In order 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. In order 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 CSD2X 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:

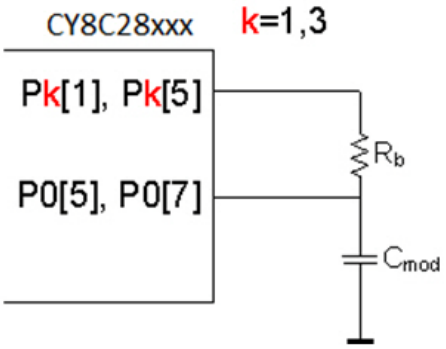
Resolution = (Number of Sensors – 1) x Multiplier

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

Feedback Component Selection Guidelines

The user module requires an external modulation capacitor C_{mod} and a modulator feedback resistor R_b. The capacitor can be connected to the P0[5], P0[7] port pins and Vss ground. The feedback resistor R_b can be connected to port pins P1[0], P1[1], P1[4], P1[5], P3[0], P3[1], P3[4], P3[5], and the capacitor pin. The pins are selected with the user module parameter setting. Do not use pins selected for modulator component connection for any other purposes.

Figure 6. External Component Connections



Modulation Capacitor

The recommended value for the modulation capacitor is 4.7 – 47 nF. The optimal capacitance can be selected by experiment to get maximum SNR. A value of 5.6 – 10 nF gives good results in the most cases. You can experiment with several capacitor values to get the best SNR after selecting the feedback resistor. A ceramic capacitor should be used. The temperature capacitance coefficient is not important. The resistor values depend on the total sensor capacitance C_s. The resistor value should be selected as follows:

- Monitor the raw counts for different sensor touches.
- Select a resistance value that provides maximum readings about 30% less than the full scale readings at the selected scanning resolution. The raw counts are increased when resistor values increase.

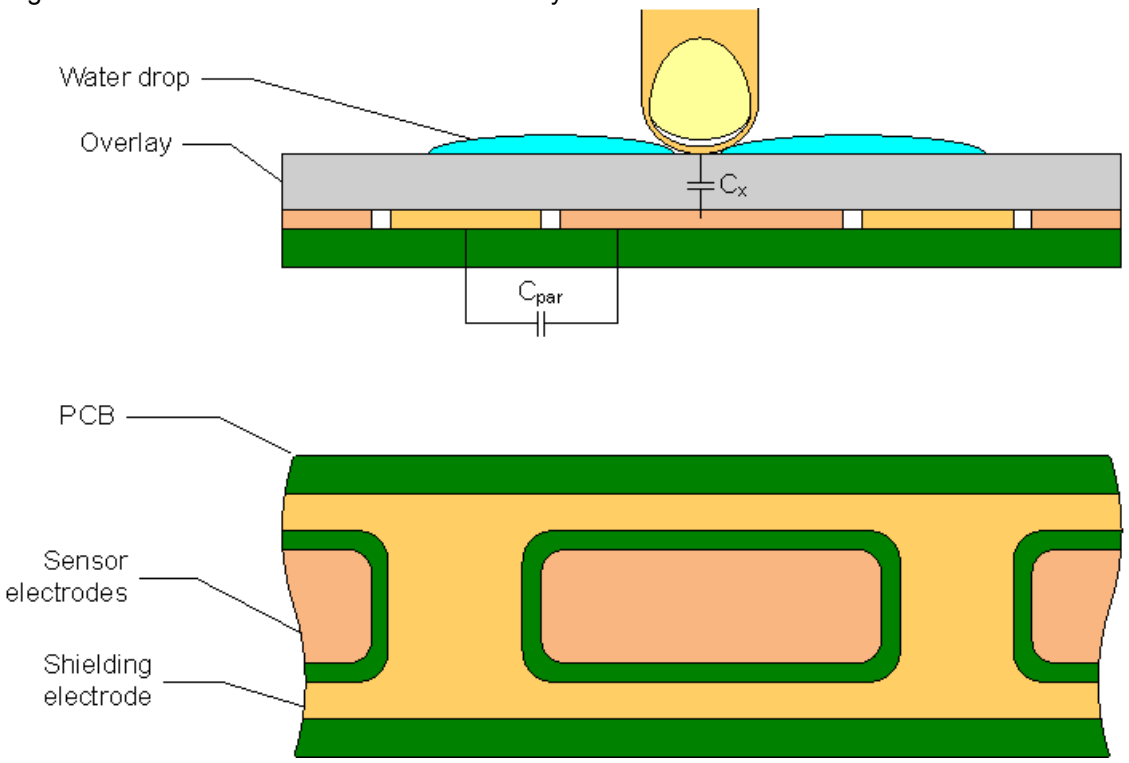
Shielding Electrode

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 device 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 CSD2X User Module supports separate output for the shielding electrode.

Figure 7. Possible Shield Electrode PCB Layout



The previous figure illustrates one possible layout configuration for the button's shield electrode. The shield electrode is especially useful for transparent ITO touchpad devices, where it blocks the LCD drive electrode's noise influence and reduces stray capacitance at the same time.

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 C_{par} 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.

The shield electrode can be connected to any PSoC pins to which it can be routed. Set the drive mode to **Strong Slow** to reduce ground noise and radiated emissions. Also, the slew limiting resistor can be connected between the PSoC device and the shielding electrode.

Comparator Reference Source

The comparator reference source used to form the comparator reference voltage. The reference voltage value determines the sensitivity.

The user module use different reference forming principle for first order and second order configurations.

For First Order Configuration the user module supports multiple selections for a reference source:

- Bandgap reference

- Analog modulator, driven by a PRSPWM or a prescaler-PWM signal
- External resistive voltage divider
- External RC filter for a PRSPWM or a prescaler-PWM signal

This table summarizes the reference selection options:

Type	External components	UM selection	When to use
Bandgap reference	none	VBG	Readings are proportional to the power supply voltage. Use only when power supply is well regulated
Analog modulator	none	ASE11	Recommended for most applications. Try starting testing from this option.
External resistive voltage divider	2	AnalogColumn Input Select	Readings are less dependent on power supply. Recommended R1 = 10k; R2 = 3.6k
External RC filter for a PRSPWM or a prescaler-PWM signal	2	AnalogColumn Input Select	If other reference selections have too much noise.

You probably use only the bandgap reference or the analog modulator. The external resistive voltage divider is useful for special cases.

Clock Source

The clock source is used to control the switches on the sensing capacitor. The user module supports four selection options as the clock source for the precharge switches:

- The 16-bit pseudo-random sequence generator (PRS16)
- The 8-bit PRS source
- The 8-bit PRS source with prescaler
- VC2

The required configuration should be selected when you first select the user module. To change this selection later, right click the CSD User Module icon in Interconnect View and select **User Module Selection Options**.

The PRS16 configuration uses the PRS16 module as a clock source. The PRS16 source provides spread-spectrum operation and ensures good immunity from external noise sources. In addition, designs with the spread-spectrum clock have lower electromagnetic emission levels. When your application is targeted to pass EMC/EMI tests or must provide reliable operation in harsh environments, the PRS16 configuration is recommended.

The PRS8 configuration uses the PRS8 clock source. The PRS8 is clocked by IMO directly. PRS8 saves one digital block by using shorter pseudo-random generator sequences. As a result, CapSense modules that use the PRS8 configuration are less robust against external noise signals.

The PRS8 configuration with prescaler uses an 8-bit counter as the PRS8 clock source. This counter is sourced by the IMO clock. The prescaler allows you to easily tune the operation frequency by changing the prescaler counter period. The main application area of the prescaler-based configuration is capacitive

sensing using high resistance materials, for example, sensing using the thin transparent ITO films over the display in a double layer touchpad device.

This table compares the four configurations:

Configuration	Operation Frequency	Digital Blocks Used	EMC Noise Immunity
PRS16	Spread-spectrum, average is $F_{IMO}/4$, peak is $F_{IMO}/2$	3	High. Sensitive points are multiples of the PRS sequence repeat period and PRS fundamental frequency F_{IMO}.
PRS8	Spread-spectrum, average is $F_{IMO}/4$, peak is $F_{IMO}/2$	2	Moderate. Sensitive at more points due to the shorter PRS repeat period.
PRS8 with prescaler	Adjustable spread spectrum, $F_{IMO}/4$ – $F_{IMO}/512$	1	Moderate. Sensitive at more points due to the shorter PRS repeat period.
VC2	fixed, IMO/(VC_1xVC_2)	0	Sensitive for EMC signals at operation frequency and its harmonics. Recommended only when no certification EMC/EMI tests are planned.

DC and AC Electrical Characteristics

Table 2. Power Supply Voltage

Parameter	Min	Typical	Max	Unit	Test Conditions and Comments
Value	2.7	5.0	5.25	V	

Table 3. Noise

Parameter[a]	Min	Typical	Max	Unit	Test Conditions(Vdd = 3.3 V, SysClk = 24 MHz, CPU Clock = 6 MHz,Baseline >= 70% of Resolution Max Count)
Noise Counts, peak-peak		0.2		% (noise counts)/ (baseline counts)	Resolution = 16
Noise Counts, peak-peak		1		% (noise counts)/ (baseline counts)	Resolution = 14
Noise Counts, peak-peak		10		% (noise counts)/ (baseline counts)	Resolution = 10

a. SNR increases as the Scan Speed slows and the Baseline counts increase.

Table 4. Power Consumption

Supply Voltage	Min	Typ	Max	Unit	Test Conditions and Comments
Active Current		10		mA	Average current during scan, 8 sensors
Standby Current		250		µA	Scanning Speed = Fast, Resolution = 9 100 ms report rate, 8 sensors
		1.6		mA	Scanning Speed = Fast, Resolution = 12 100 ms report rate, 8 sensors
Sleep/Wake Current		10		µA	1s report rate, 1 sensor

Placement

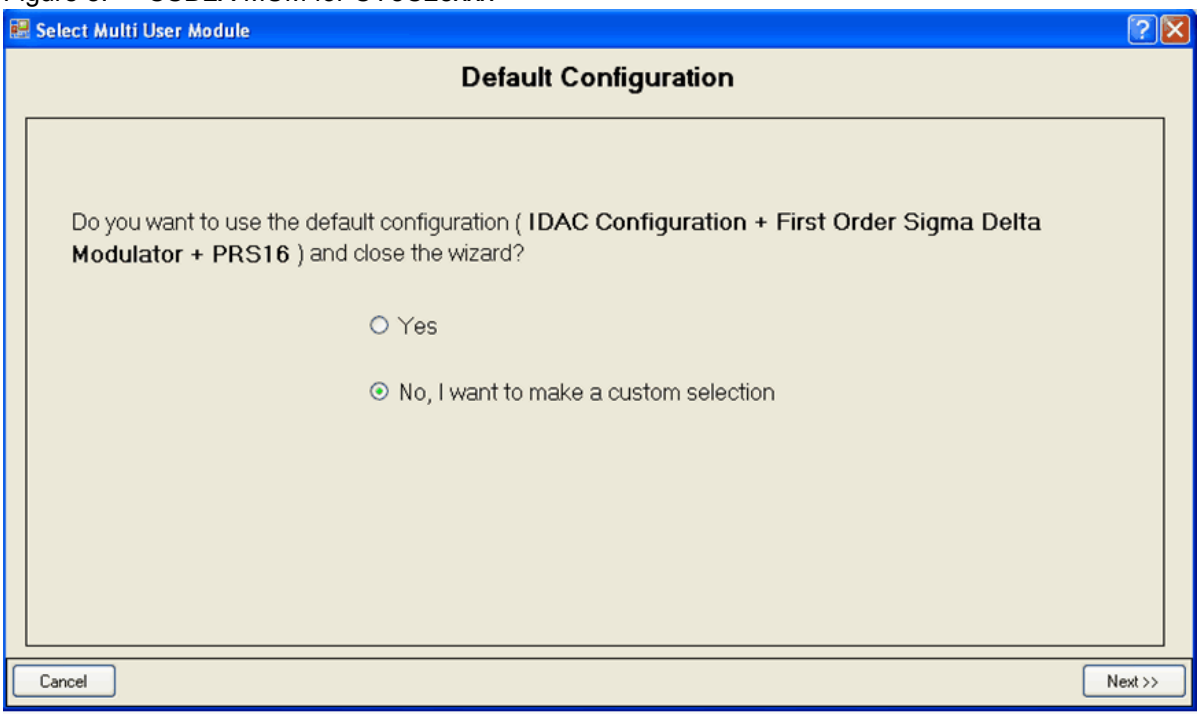
The blocks for the user module are automatically placed when the user module is instantiated. Alternate placements are available only for single-channel configurations. User modules (such as AMuxN) that occupy the same analog mux bus may have a conflict with the CSD2X User Module when they are used at the same time. If a simultaneous operation is needed, then the user modules should use pins that belong to different analog mux buses.

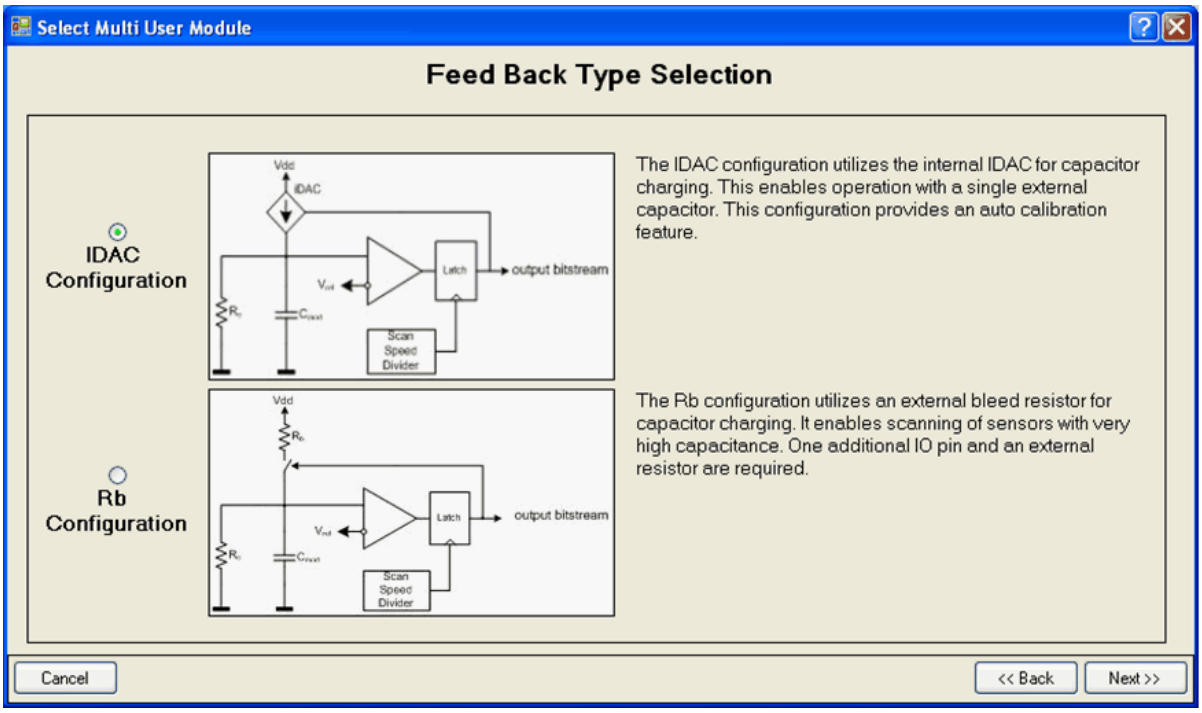
The user module supports 16 selection options.

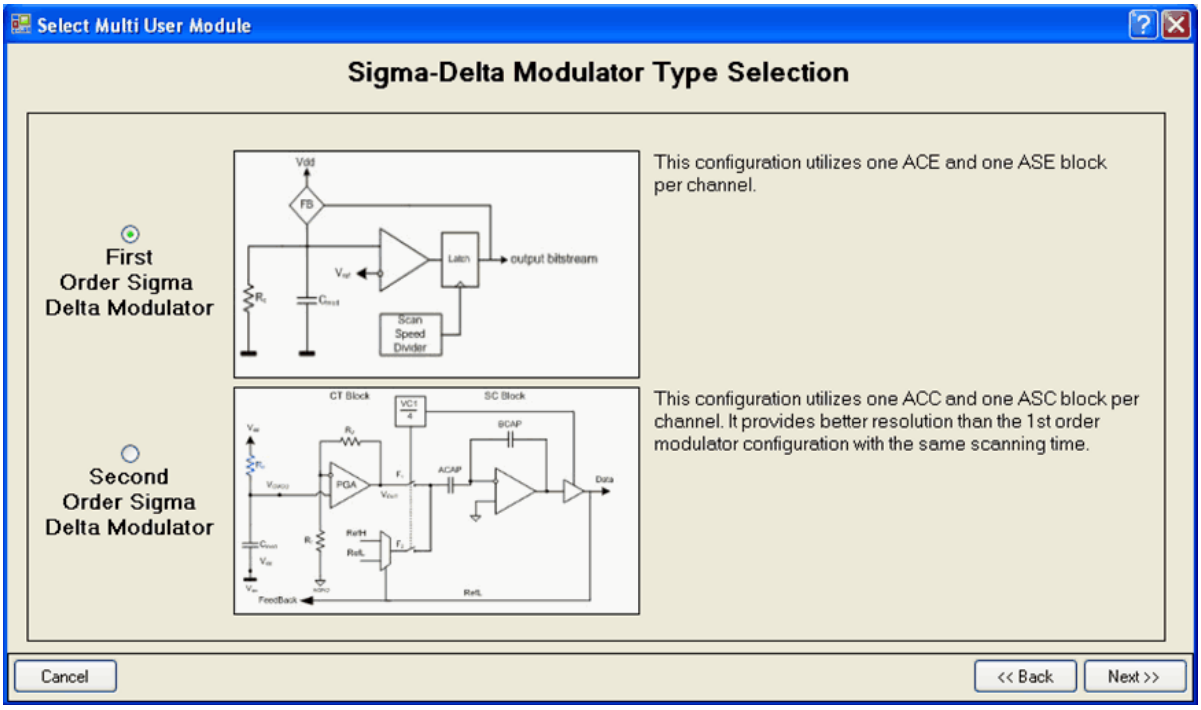
Configuration	Abbreviation
External resistor + Second Order Modulator + PWM8	RBCNT
External resistor + Second Order Modulator + PRS16	RBPRS
External resistor + Second Order Modulator + PRS8	RBPRS8
External resistor + Second Order Modulator + VC2	RBVC2
External resistor + First Order Modulator + PWM8	RECNT
External resistor + First Order Modulator + PRS16	REPRS
External resistor + First Order Modulator + PRS8	REPRS8
External resistor + First Order Modulator + VC2	REVC2
IDAC + Second Order Modulator + PWM8	IBCNT
IDAC + Second Order Modulator + PRS16	IBPRS
IDAC + Second Order Modulator + PRS8	IBPRS8
IDAC + Second Order Modulator + VC2	IBVC2
IDAC + First Order Modulator + PWM8	IECNT
IDAC + First Order Modulator + PRS16	IEPRS
IDAC + First Order Modulator + PRS8	IEPRS8
IDAC + First Order Modulator + VC2	IEVC2

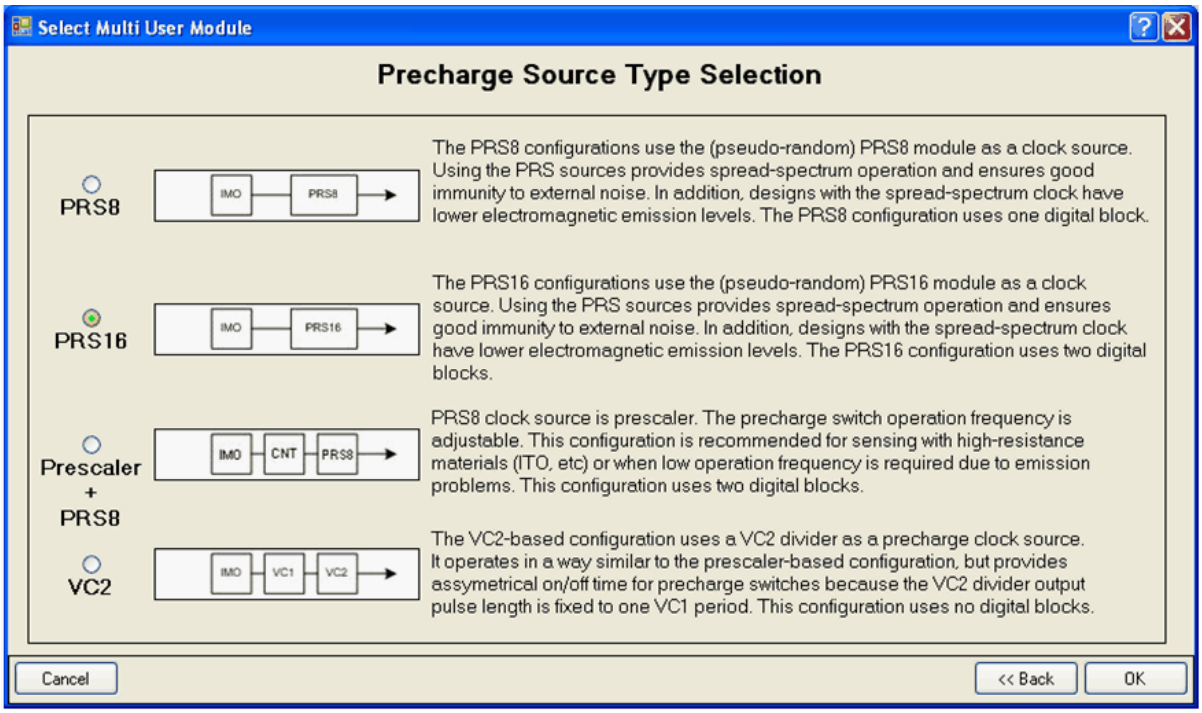
This wizard can be accessed through the MUM wizard. Follow the wizard to make your selection.

Figure 8. CSD2X MUM for CY8C28xxx









The blocks for the user module are automatically placed when the user module is instantiated; alternate placements are available only for single-channel configurations only. The consumed resources depend on the CSD2X User Module configuration. User modules that require specific pin resources, including the LCD and I2CHW, must be placed before starting the CSD2X Wizard to establish pin connections for the CSD2X 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.

Figure 9. Possible Digital Block Resources

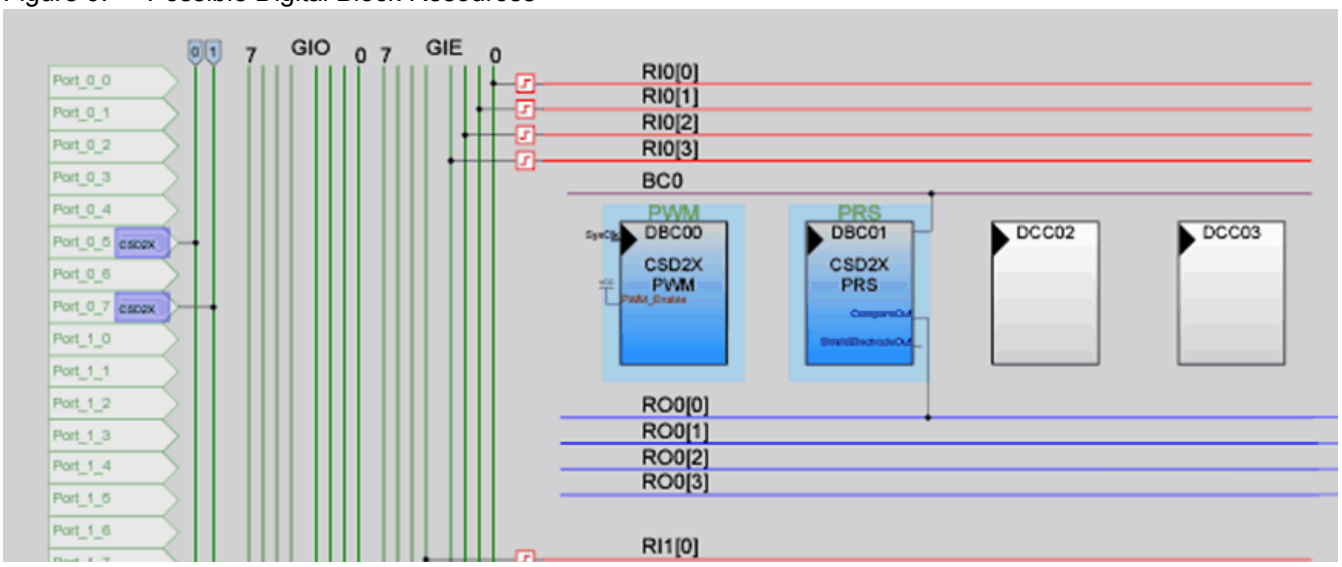


Figure 10. Possible Analog Block Resources - First Order Modulator

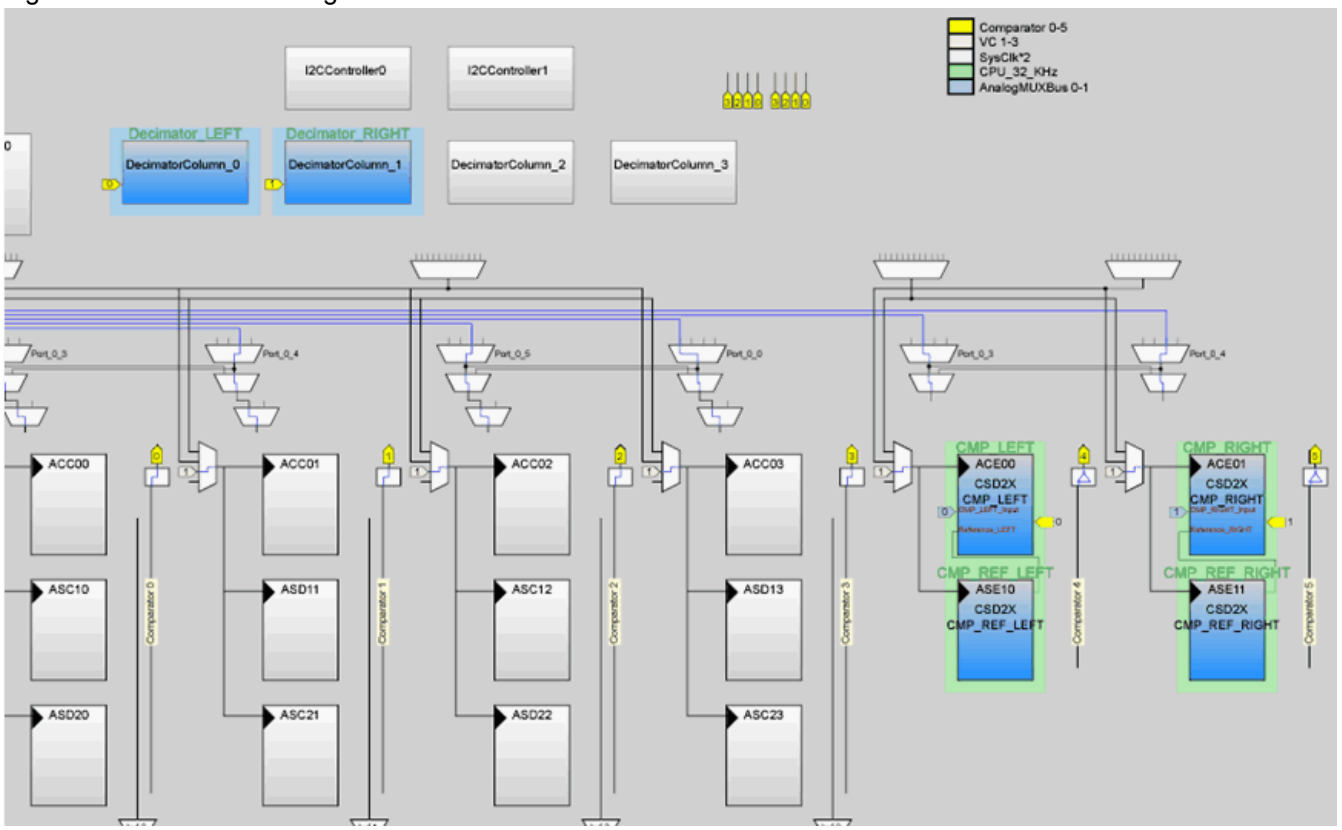
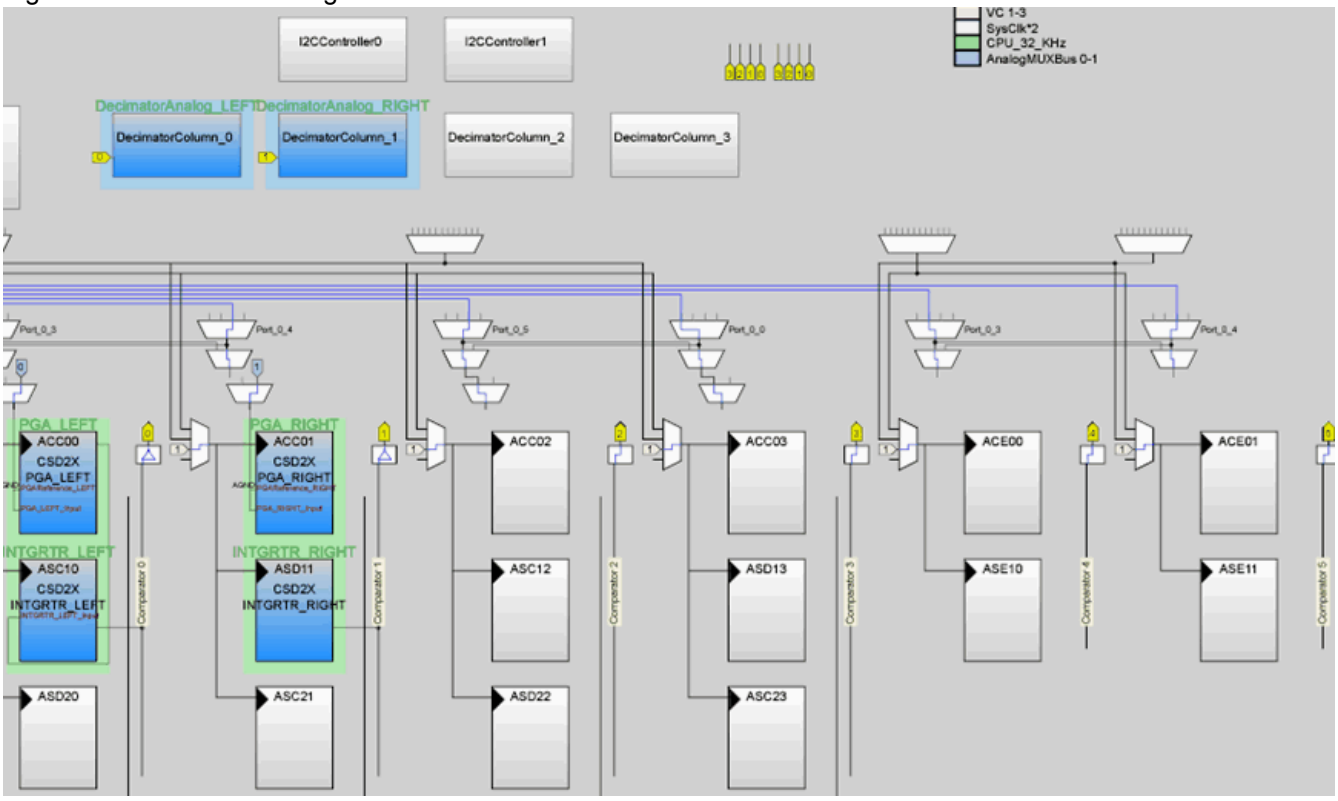


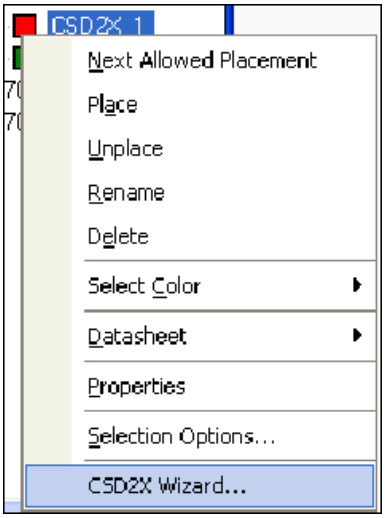
Figure 11. Possible Analog Block Resources - Second Order Modulator



Wizard

The CSD2X Wizard is used to set up the pinout for your CapSense buttons, sliders, and proximity sensors. You choose the configuration you want and assign the buttons and segments using a drag and drop interface.

1. To access the Wizard, right click the user module in the Workspace Explorer and select the CSD2X Wizard.



2. The Wizard opens showing the numeric entry boxes for the number of sensors, sliders, and radial slider sensors. The options in the wizard vary depending on the configuration. The following screen-shots show the wizard for different configurations.

Figure 12. CSD2X Wizard with IDAC Configuration

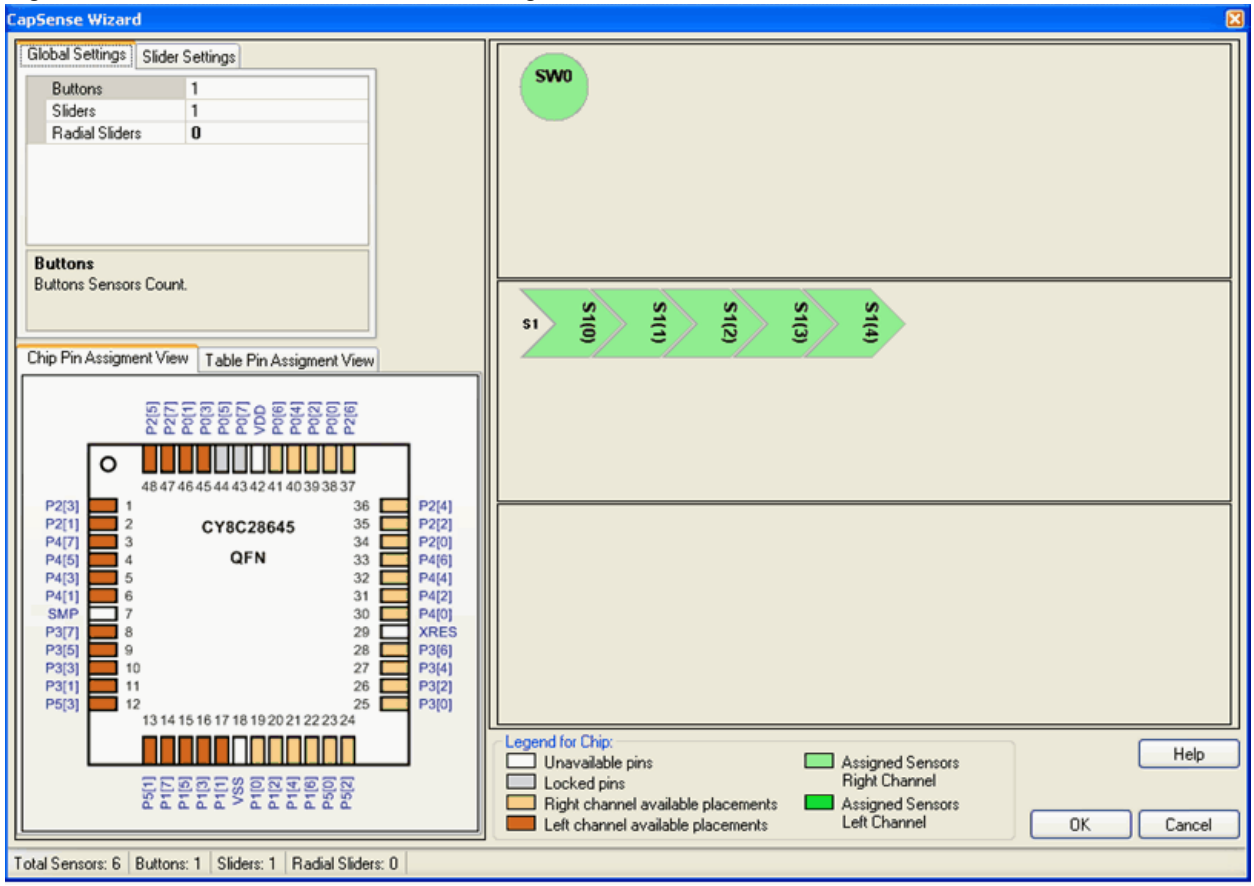
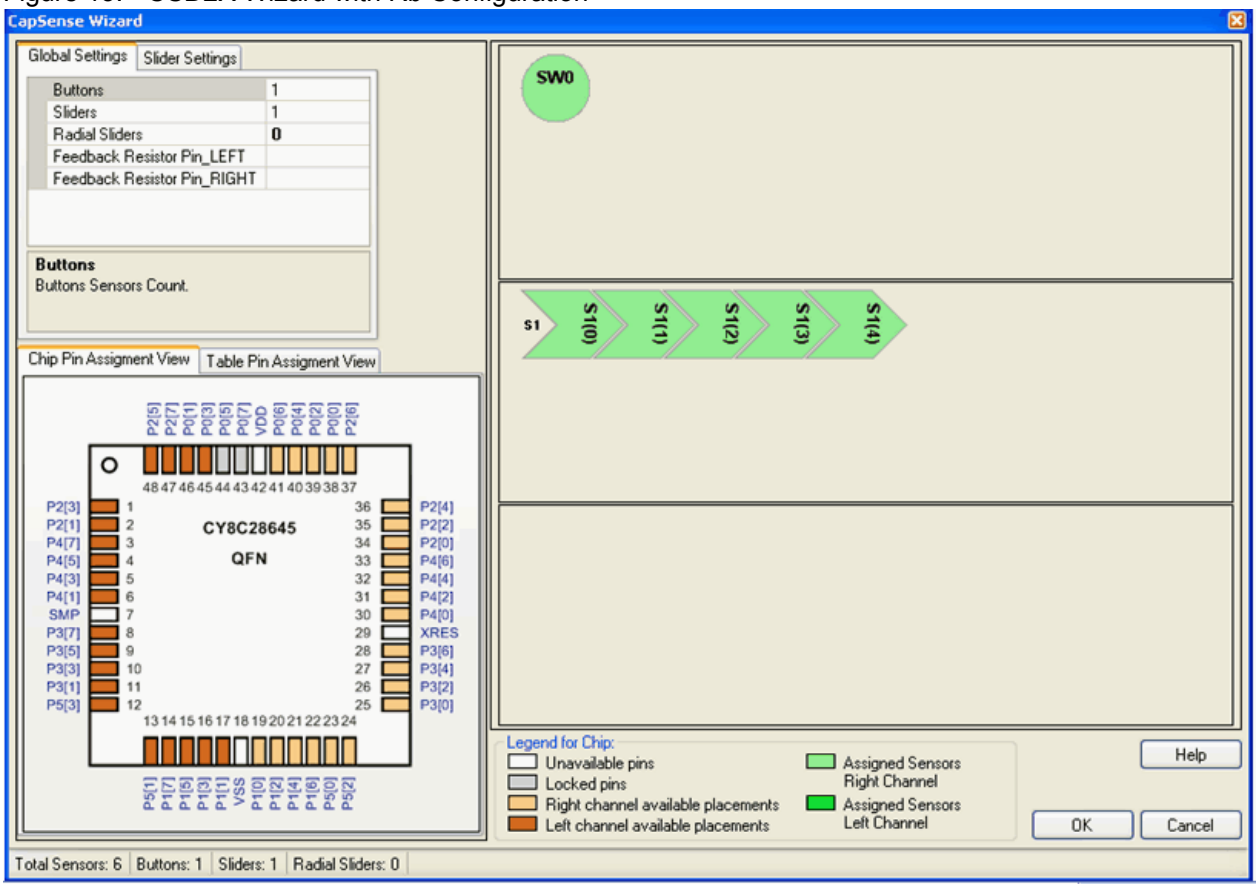


Figure 13. CSD2X Wizard with Rb Configuration



Wizard Pin Legend

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

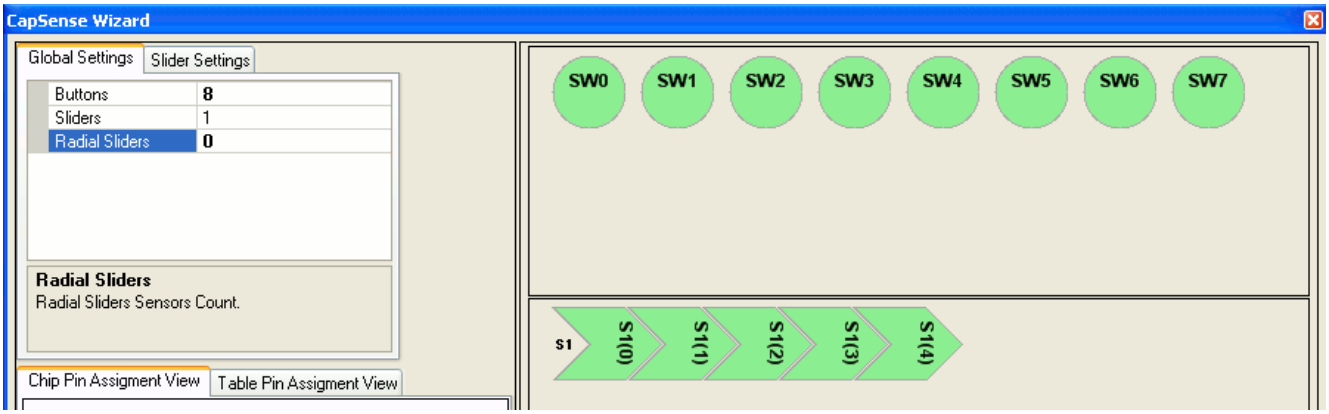
Gray – The pin is locked. There are two possible causes for this. The first possibility is that another user module such as the LCD or I^2C has claimed the pin. The second possibility is that the name of the pin has been changed from its default.To return the pin name to its default, in the Pinout view 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.

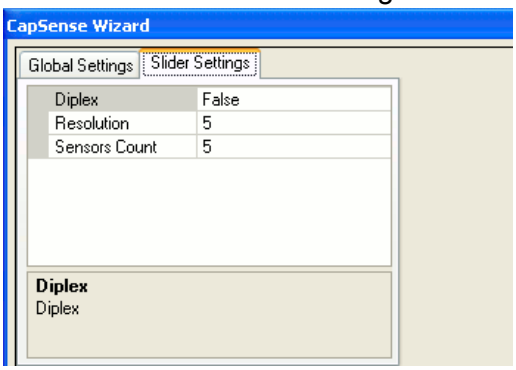
Green – The pin has been assigned as a CapSense input.

3. The right area of the wizard contains three areas where representations of your switches, sliders, and rotary sliders are added. Type the number of switches, sliders, or rotary sliders you want in the appropriate box and press [Enter]. The display updates with representations of your chosen configuration. In

the CSD2X Wizard with IDAC configuration, select the Feedback resistor pin from the drop-down menu

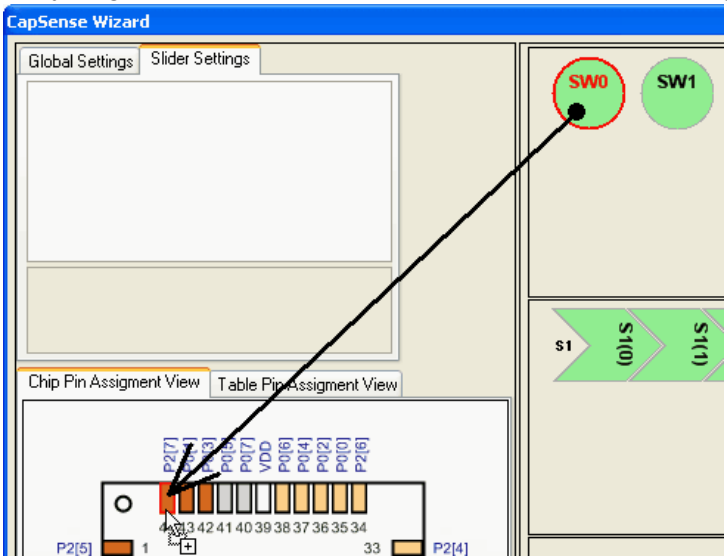


4. Switch to the Slider Settings tab to set parameters for your sliders.

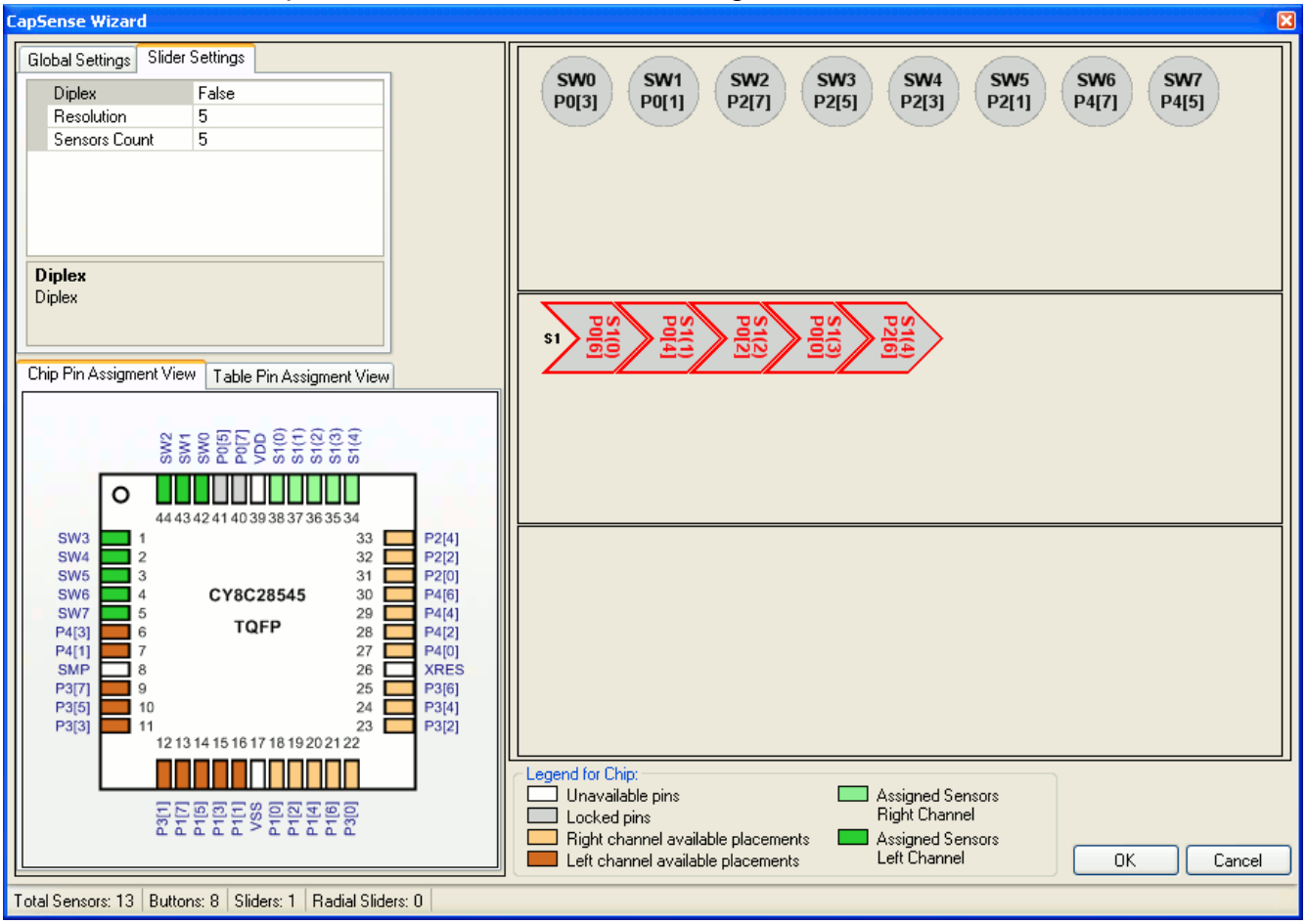


5. The Resolution of a slider is the number of positions your slider has. Touches that give readings on more than one segment of the slider are interpolated to this resolution. If you set the resolution to 100, the position of the finger on the slider is reported as being somewhere between 0 and 99. The minimum value is five. The maximum value is (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.
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. To map switches to pins, left-click the switch and drag it to any available pin. You may use either the Chip Pin Assignment view or the Table Pin Assignment view for this operation. The port pin is grayed out after selection and is no longer available. Change sensor assignments by dragging the sensor off

of the port pin. If you right-click on the Chip Pin Assignment View or the Table Pin Assignment View, you get the Clear All Pins option. This option unassigns all pins.



8. Repeat for the remaining independent sensors.
9. Mapping of slider segments to port pins is the same as for individual sensors.
10. Click **OK** to accept data and return to PSoC Designer.



Sensor placement is now complete. Set user module parameters and generate 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 pin assignment box. The pin is unassigned and you can then reassign it.

Wizard Slider Settings

Diplex

Allows you to use a single pin to monitor two electrical sensors for resolution enhancement. See the section on diplexing for more information.

Slider Resolution

For sliders and rotary sliders, this value sets the range of values returned by the CSD2X_wGetCentroidPos API. If any slider sensor is active, the function returns values from zero to the Resolution value set in the CSD Wizard. The CapSense algorithm interpolates the centroid position to this resolution based on readings from adjacent sensors.

Tables Produced by the Wizard

After completing the Wizard, click Generate Application. Based on your entries for sensor count, pin assignment, diplexing, and resolution, a set of tables is generated. The tables are located in *CSD2X_Table.asm* and *CSD2X.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). There are two tables for left and right channel each. The table includes all independent sensors, then each sensor in order. An example for a table with six sensors is:

```
CSD2X_Sensor_Table_Right:
_CSD2X_Sensor_Table_Right:
      dw    0x0140  //  Port 1 Bit 6
      dw    0x0301  //  Port 3 Bit 0
      dw    0x0304  //  Port 3 Bit 2

CSD2X_Sensor_Table_Left:
_CSD2X_Sensor_Table_Left:
      dw    0x0308  //  Port 3 Bit 3
      dw    0x0302  //  Port 3 Bit 1
      dw    0x0108  //  Port 1 Bit 3
```

This table is used by CSD2X_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 CSD2X wizard.

```
CSD2X_Group_Table:
_CSD2X_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 when it is a slider and is also diplexed. 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


 CSD2X_Diplex_Table:
_CSD2X_Diplex_Table:
 db >DiplexTable_0, <DiplexTable_0
 db >DiplexTable_1, <DiplexTable_1
```

Order Table

The Order Table consists of one byte for each sensor. This byte is the left sensor order in raw counts result array without channel dependence. The table is generated by wizard and reflect sensor order.

```
CSD2X_1_Order_Table_Left:
_CSD2X_1_Order_Table_Left:
DB 0x01 // Position 1

CSD2X_1_Order_Table_Right:
_CSD2X_1_Order_Table_Right:
DB 0x00,0x02 // Position 0 and 2
```

Parameters and Resources

PGAGain_RIGHT

The default value is 4.00.

PGAGain_LEFT

Parameter available in Second Order Modulator configurations only. Sets the PGA gain for the ADC on the left or right channels. The two PGAs can be set independently. Gain ranges are 1 to 48.00, using PSoC Designer or the CSD2X_SetPGAGain routine provided in the API. Gain settings less than 1 are not supported. The default value is 4.00.

Finger Threshold

This threshold is used to determine the state of each button sensor. If any sensor is active, the bIsAnySensorActive() function returns a 1. If all sensors are off, the bIsAnySensorActive() function returns a 0. The finger detection threshold values apply to all sensors and sliders. Possible values range from 5 to 255; the default value is 40.

Noise Threshold

For individual sensors, count values above this threshold do not update the baseline. For slider sensors, count values below this threshold are not counted in the calculation of the centroid. Possible values are 5 to 255; the default value is 20.

BaselineUpdate Threshold

When the new raw count value is above the current baseline and the difference is below the noise threshold (with the Sensors Autoreset parameter set to Disabled), the difference between the current baseline and the raw count is accumulated into what could be thought of as a bucket. When the bucket fills, the baseline is incremented by some value and the bucket is emptied. This parameter sets the threshold that the bucket must reach for the baseline to increment. Possible values are 0 to 255. Larger parameter values yield slower baseline update speeds. If you need more frequent baseline updates, decreases this parameter. The default value is 200.

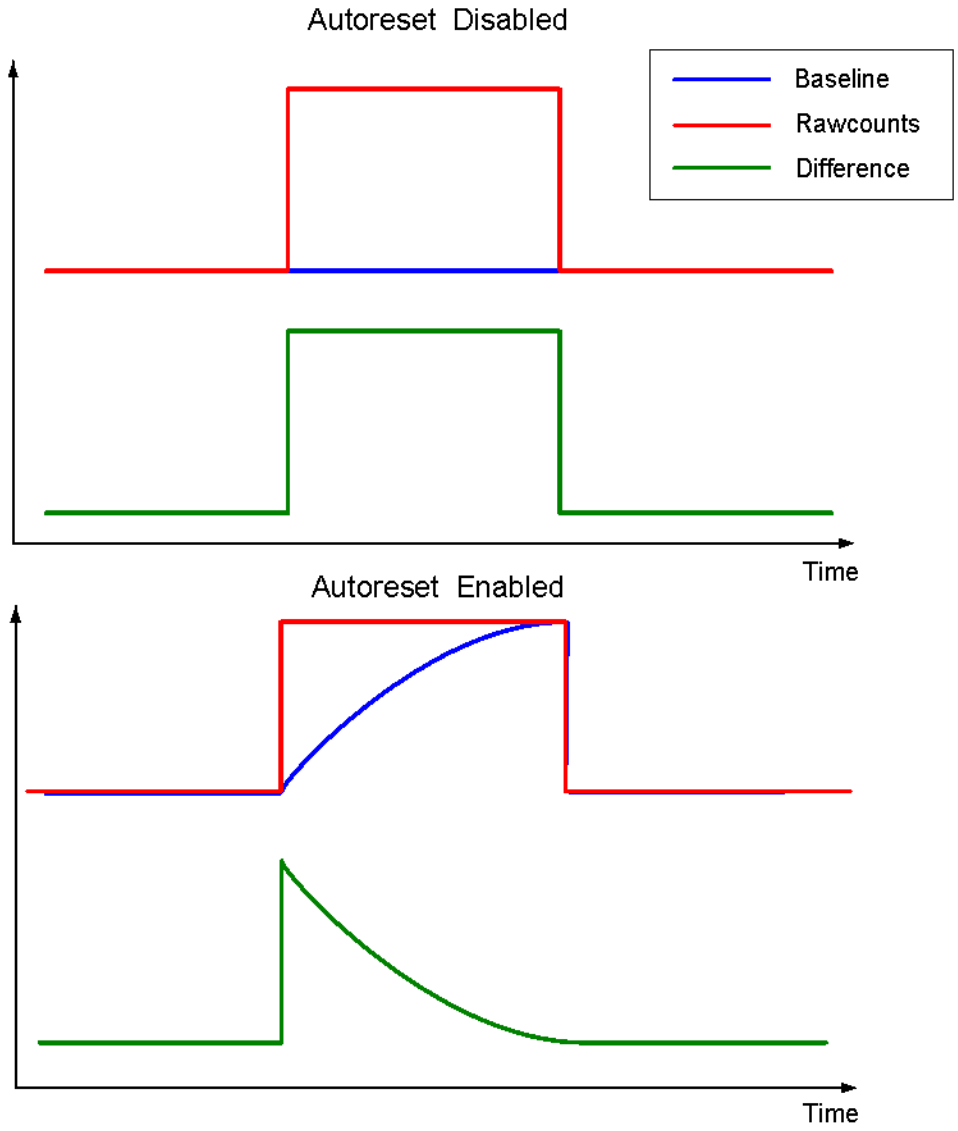
Sensors Autoreset

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 default setting is Disabled.

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

Figure 14. The Sensor Autoreset Parameter



Hysteresis

The Hysteresis parameter adds or subtracts from the finger threshold depending on whether the sensor is currently active or inactive. If the sensor is inactive, the difference count must overcome the finger threshold plus hysteresis. If the sensor is active, the difference count must go below the finger threshold minus hysteresis. It is used to add debouncing and stickiness to the finger detection algorithm. The threshold with hysteresis is evaluated when bIsSensorActive() or bIsAnySensorActive() is called. The sensor state can be monitored with the return value of bIsSensorActive() or the baSnsOnMask[] array. Possible values are 0 to 255, but must be lower than the Finger Threshold parameter setting.

Proper selection of high level decision logic parameters allows you to effectively compensate for environmental factors (temperature, humidity changes, and so on), suppress noisy signals (ESD, power supply spikes), and provide reliable touch detection under various conditions. The default value is 10.

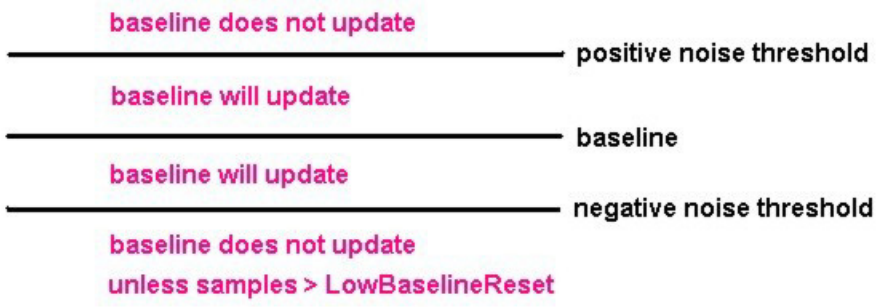
Debounce

The Debounce parameter adds a debounce counter to the sensor active transition. In order 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; the default value is 3. A setting of 1 provides no debouncing.

NegativeNoiseThreshold

The NegativeNoiseThreshold parameter adds a negative difference count threshold. If the current raw count is below the baseline and the difference between them is greater than this threshold, the baseline is not updated. However, if the current raw count stays in the low state (difference greater than threshold) for the number of samples specified by the LowBaselineReset parameter, the baseline is reset. The default value is 20.



LowBaselineReset

The LowBaselineReset parameter works together with the NegativeNoiseThreshold parameter. If the sample count values are below the baseline minus the NegativeNoiseTreshold for the specified number of samples, the baseline is set to the new raw count value. It essentially counts the number of abnormally low samples required to reset the baseline. It is generally used to correct for the finger-on-at-startup condition. The default value is 50.

Scanning Speed

This parameter affects the sensors' scanning speed. The available selections are **Fast**, **Normal**, and **Slow**. The default setting is Normal. Slower scanning speeds provide the following advantages:

- Improved SNR
- Better immunity to power supply and temperature changes
- Less demand for system interrupt latency; you can handle longer interrupts

The scanning speed and resolution affect the VC1 divider in the following way:

Scanning speed	VC1	
	Second Order Delta-Sigma Modulator ACC+ASD blocks	First Order Delta-Sigma Modulator ACE+ASE blocks
Fast	3	2
Normal	4	4
Slow	8	8

Resolution (First Order Modulator)

This parameter determines the scanning resolution in bits. Possible selection is 8, 10, and 12 bits. The maximum raw count for scanning resolution for N bits is 2^N-1.

Increasing the resolution improves sensitivity and the SNR of touch detection. The default value is 12.

Table 5. Scanning Time in µs vs. Scanning Speed and Resolution for 24 MHz IMO Operation

Resolution, bits	Scanning speed		
	Fast	Normal	Slow
8	85	130	260
10	130	260	510
12	260	510	1020

Note The scanning time was measured as the time interval between 2 sensor scans. This time includes the sensor setup time, modulator stabilization delay, sample conversion interval and data preprocessing time.

Resolution (Second Order Modulator)

This parameter determines the scanning resolution. The options are 0, 12, and 14 bits. If you need higher resolution with these configurations, please use oversampling and average several samples. The maximum raw count for scanning resolution for N bits is 2^N-1. Increasing the resolution improves sensitivity and the SNR of touch detection. The default value is 12.

Table 6. Scanning Time in µs vs. Scanning Speed and Resolution for 24 MHz IMO Operation

Resolution, bits	Scanning speed		
	Fast	Normal	Slow
10	124	136	296
12	220	220	548
14	428	552	1060

Note The scanning time was measured as the time interval between 2 sensor scans. This time includes the sensor setup time, modulator stabilization delay, sample conversion interval and data preprocessing time.

IDAC Value Left

The default value is 200.

IDAC Value Right

Parameter available in IDAC configurations only. The left or right channel capacitance measurement range depends on this parameter. Higher value corresponds to wider range. Adjust the IDAC value to get raw counts about 50-70% of full range. This parameter can be changed in run-time using corresponding API function.

Possible values are 1 to 255; the default value is 200.

IDAC Range

Parameter available in IDAC configurations only. Sets the IDAC current multiplier. The result of the setting is different for dual channel configurations. The results in dual channel configurations are shown:

Setting	Effect
1X[a]	Maximum IDAC current is 19.92 µA
4X	Maximum IDAC current is 91.03 µA
16X	Maximum IDAC current is 318.75 µA
32X	Maximum IDAC current is 637.50 µA

a. Not recommended for new designs.

Connect sensors that have large capacitance to the left channel in two channel configuration. The default value is x32.

Prescaler Period

Parameter available in the Prescaler + PRS8 configurations only. This parameter sets the prescaler period register and determines the precharge switch output frequency. This parameter is available for configuration with prescaler only. The prescaler period values can range from 1 to 255.

The recommended values are 2n – 1 to obtain the maximum signal to noise ratio (SNR).

- 1
- 3
- 7
- 15
- 31
- 63
- 127
- 255

Other values can result in more noise, especially at low resolution and high scan speed. The default value is 7.

Shield Electrode Out

The shielding electrode signal source can be selected from one of the free digital row buses (Row_0_Output_1 - Row_0_Output_3). Each row output can be routed to one of three pins. Set the Row LUT Function to A. The default setting is None.

PRS Polynomial

This parameter sets the PRS polynomial in the PRS-based configurations. There are two selection options:

- Short – The short polynomial setting yields better SNR, but due to the shorter repeat period, the end device can be more susceptible to external noise sources.
- Long – The long polynomial setting yields worse SNR, but the device is more robust against noise signals.

The default setting is Short.

Auto Calibration

Enable or disable auto calibration API functions.

Autocalibration is included in IDAC configurations only. Autocalibration automatically selects possible IDAC values to get raw counts in half of the resolution range. This reduces the overall sensitivity of the CapSense algorithm, but it allows you to quickly get raw counts in a readable range when you begin the tuning process. The autocalibration consumes ROM and RAM resources and increases the start time. If the raw count value after calibration is less than half of the resolution range then you should increase the IDAC range or reduce the precharge frequency. Autocalibration works to improve marginally functional configurations. The default setting is Enabled.

Reference_LEFT

The default value is ASE10.

Reference_RIGHT

Parameter available in First Order Modulator configurations only. This parameter sets the comparator reference value for the left and right channel. The reference comes from the internal resistive voltage divider. The default value is ASE11.

Ref Value

This parameter is applicable to First Order Delta Sigma Modulator only and sets the comparator reference value, when comparator reference comes from an analog modulator (ASE11) or an externally filtered PWM/PRSPWM signal (from AnalogColumn_InputSelect_1 with RC filter). This value has no effect when the reference comes from the bandgap (VBG) or from an external voltage divider (from AnalogColumn_InputSelect_1 with resistive voltage divider).

Zero is the minimum reference (1/4 Vdd). Eight is the maximum value (3/4Vdd). When the reference increases, the sensitivity is decreased, but the influence on the shielding electrode is increased.

If the design has sensors with noticeable capacitance differences (for example, sensors with different sized squares), you can balance raw counts by setting a higher reference for the sensors with larger capacitance using an API function. The default value is 4.

Feedback Resistor Pin_LEFT

Feedback Resistor Pin_RIGHT

This parameter is available for Rb configurations only. This parameter sets the pin to connect the external feedback resistor (R_b) for the left and right channels. Choose from the available pins: P1[0], P1[1], P1[4], P1[5], P3[0], P3[1], P3[4], P3[5] for the left chanel and P1[1], P1[2], P1[5], P1[6], P3[1], P3[2], P3[5], P3[6] for the right channel. Some pins are not available on some device packages or some configurations. Tip: if some of these pins are used for other purposes (for example, allocated for sensor connection), they are not available for selection in the user module parameter list. Future versions of the CSD User Module may allow additional pins to be used for connecting the feedback resistor. This allows the use of a second I^2C port on packages that have no P3 port. Use pins P1[5] or P3[1] to avoid programming problems.

BackgroundScanning

This parameter decides whether the background scanning is enabled or disabled in the project. Based on the selection, a code is generated for the corresponding APIs.

FMEA_Shorts_Test

This feature enables and disables the FMEA short tests.

FMEA_Cmod_Rb_Test

This feature enables the test for calculating Cmod and Rb.

Application Programming Interface

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

Only one instance of this user module can be placed in the project and this also applies to loadable configurations. Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns the CSD2X_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 CSD2X 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 CSD2X, start it sampling, and stop the CSD2X. In all cases the instance name of the module replaces the CSD2X 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:

- CSD2X_waSnsBaseline[]
- CSD2X_waSnsResult[]
- CSD2X_waSnsDiff[]
- CSD2X_baSnsOnMask[]
- CSD2X_bScanComplete
- CSD2X_baSensorShortGnd[]
- CSD2X_baSensorShortVdd[]
- CSD2X_wCmod_Val
- CSD2X_wCmod_Rb_Val

CSD2X_waSnsBaseline[] – This is an integer array that contains the baseline data of each sensor. The array size is equal to the sensor count. The CSD2X_waSnsBaseline[] array is updated by these functions:

- CSD2X_UpdateAllBaselines();
- CSD2X_UpdateSensorBaseline();
- CSD2X_InitializeBaselines().

CSD2X_waSnsResult[] – This is an integer array that contains the raw data of each sensor. The array size is equal to the sensor count. The CSD2X_waSnsResult[] data is updated by these functions:

- CSD2X_ScanSensor();
- CSD2X_ScanAllSensors().

CSD2X_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.

CSD2X_baSnsOnMask[] – This is a byte array that holds the sensor on or off state (for buttons or sliders). CSD2X_baSnsOnMask[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). CSD2X_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 CSD2X_baSnsOnMask[] data is updated by CSD2X_bIsSensorActive(BYTE bSensor) function or CSD2X_bIsAnySensorActive() routines.

CSD2X_bScanComplete[] – This variable is valid only when the background scanning feature is enabled. This variable is set when sensor scanning is complete and is updated in the CSD2X interrupt.

Masks	Value	Description
CSD2X_SCAN_COMPLETE	0X01	Scan All Sensors Complete Flag
CSD2X_SCAN_1COMPLETE	0X02	Scan One Sensor Complete Flag
CSD2X_SCAN_ALLSENSORS	0X04	ScanAllSensors Flag

CSD2X_baSensorShortGnd[] – This BYTE array is valid only when the FMEA feature is enabled. CSD2X_baSensorShortGND[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0 and sensor 1 is bit 1). CSD2X_baSensorShortGND[1] contains the masked bits for sensors 8 through 15 (if they are needed), and so on. This byte array contains only those elements that are necessary to contain

all the placed sensors.The CSD2X_baSensorShortGnd[] data is updated by the CSD2X_bFMEA_CheckGndShort() function.

CSD2X_baSensorShortVdd[] – This BYTE array is valid only when the FMEA feature is enabled. CSD2X_baSensorShortVdd[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0 and sensor 1 is bit 1). CSD2X_baSensorShortVdd[1] contains the masked bits for sensors 8 through 15 (if they are needed), and so on. This byte array contains only those elements that are necessary to contain all the placed sensors. The CSD2X_baSensorShortGnd[] data is updated by CSD2X_bFMEA_CheckVddShort() function.

CSD2X_wCmod_Val[] – This is a WORD variable, which is valid only when the FMEA feature is enabled and the IDAC method is selected. This variable contains the value of Cmod calculated during the CSD2X_FMEA_Cmod_Check function.

CSD2X_wCmod_Rb_Val – This is a WORD variable, which is valid only when the FMEA feature is enabled and the Rb method is selected. This variable contains the product of Cmod and Rb calculated during the CSD2X_FMEA_Cmod_Rb_Check function.

The following group of variables is used for parameters storage:

- bNoiseThreshold - keeps value of NoiseThreshold parameter;
- bNegativeNoiseThreshold - keeps value of NegativeNoiseThreshold parameter;
- bBaselineUpdateThreshold - keeps value of BaselineUpdateThreshold parameter;
- bHysteresis - keeps value of Hysteresis parameter;
- bDebounce - keeps value of Debounce parameter;
- bLowBaselineReset - keeps value of LowBaselineReset parameter;
- baBtnFThreshold - keeps value of default finger threshold for each sensor. It is initialized in CSD2X_SetDefaultFingerThresholds().

APIs	IBCNT	IBPRS	IBPRS8	IBVC2	IECNT	IEPRS	IEPRS8	IEVC2	RBCNT	RBPRS	RBPRS8	RBvC2	RECNT	REPRS	REPRS8	REVC2
CSD2X_Start()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_Stop()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_ScanSensor()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_ScanAllSensors()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_ClearSensors()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_wReadSensor()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_wGetPortPinLeft()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_wGetPortPinRight()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_EnableSensor()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_DisableSensor()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_SetScanMode()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•

APIs	IBCNT	IBPRS	IBPRS8	IBVC2	IECNT	IEPRS	IEPRS8	IEVC2	RBCNT	RBPRS	RBPRS8	RBvC2	RECNT	REPRS	REPRS8	REVC2
CSD2X_SetPGAGainLeft()	•	•	•	•					•	•	•	•				
CSD2X_SetPGAGainRight()	•	•	•	•					•	•	•	•				
CSD2X_SetPGARefLeft()	•	•	•	•					•	•	•	•				
CSD2X_SetPGARefRight()	•	•	•	•					•	•	•	•				
CSD2X_SetRefValue()					•	•	•						•	•	•	
CSD2X_Calibrate()	•	•	•	•	•	•	•	•								
CSD2X_UpdateSensorBaseline()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_bIsSensorActive()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_bIsAnySensorActive()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_SetDefaultFingerThresholds()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_InitializeSensorBaseline()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_InitializeBaselines()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_UpdateAllBaselines()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_wGetCentroidPos()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_wGetRadialPos()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_wGetRadialInc()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_baSensorShortGnd()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_baSensorShortVdd()	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
CSD2X_bFMEA_Left_Cmod_Check()	•	•	•	•	•	•	•	•								
CSD2X_bFMEA_Right_Cmod_Check()	•	•	•	•	•	•	•	•								
CSD2X_bFMEA_Left_Cmod_Rb_Check()									•	•	•	•	•	•	•	•
CSD2X_bFMEA_Right_Cmod_Rb_Check()									•	•	•	•	•	•	•	•

CSD2X_Start

Description:

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

C Prototype:

```
void  CSD2X_Start()
```

Assembly:

```
lcall  CSD2X_Start
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD2X_Stop

Description:

Stops the sensor scanner, disables internal interrupts, and calls CSD2X_ClearSensors() to reset all sensors to an inactive state.

C Prototype:

```
void  CSD2X_Stop()
```

Assembly:

```
lcall  CSD2X_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD2X_Resume

Description:

Resumes the user module operation after CSD2X_Stop call.

C Prototype:

```
void  CSD2X_Resume()
```

Assembly:

```
lcall  CSD2X_Resume
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD2X_SetPGAGainLeft

Description:

Sets the gain for the left PGA block, overwriting value set in the Device Editor.

C Prototype:

```
void  CSD2X_SetPGAGainLeft(byte bGainSetting)
```

Assembly:

```
mov   A, bGainSetting
lcall  CSD2X_SetPGAGainLeft
```

Parameters:

bGainSetting: Symbolic names provided in C and assembly include files, and their associated values, are given in the following table. PGA gain can be set from 1 to 48, settings below 1 are not supported. This function is common for ADC and CSD modes. In the first case it sets a ADC preamplifier gain, in the second it sets a modulator gain.

Symbolic Name	Value
PGA_G48_0	0x0C
PGA_G24_0	0x1C
PGA_G16_0	0x08
PGA_G8_00	0x18
PGA_G5_33	0x28
PGA_G4_00	0x38
PGA_G3_20	0x48
PGA_G2_67	0x58
PGA_G2_27	0x68
PGA_G2_00	0x78
PGA_G1_78	0x88
PGA_G1_60	0x98
PGA_G1_46	0xA8
PGA_G1_33	0xB8
PGA_G1_23	0xC8
PGA_G1_14	0xD8
PGA_G1_06	0xE8
PGA_G1_00	0xF8

For second Order configuration only.

Return Value:

None

Side Effects:

**

CSD2X_SetPGAGainRight

Description:

Sets the gain for the right PGA block, overwriting value set in the Device Editor.

C Prototype:

```
void  CSD2X_SetPGAGainRight(byte bGainSetting)
```

Assembly:

```
mov   A, bGainSetting
lcall  CSD2X_SetPGAGainRight
```

Parameters:

bGainSetting: symbolic names provided in C and assembly include files, and their associated values, are given in the following table. PGA gain can be set from 1 to 48, settings below 1 are not supported. This function is common for ADC and CSD modes. In the first case it sets a ADC preamplifier gain, in the second it sets a modulator gain.

Symbolic Name	Value
PGA_G48_0	0x0C
PGA_G24_0	0x1C
PGA_G16_0	0x08
PGA_G8_00	0x18
PGA_G5_33	0x28
PGA_G4_00	0x38
PGA_G3_20	0x48
PGA_G2_67	0x58
PGA_G2_27	0x68
PGA_G2_00	0x78
PGA_G1_78	0x88
PGA_G1_60	0x98
PGA_G1_46	0xA8
PGA_G1_33	0xB8

Symbolic Name	Value
PGA_G1_23	0xC8
PGA_G1_14	0xD8
PGA_G1_06	0xE8
PGA_G1_00	0xF8

For second Order configuration only.

Return Value:

None

Side Effects:

**

CSD2X_ScanSensor

Description:

Scans the selected pair of sensors. Sensor number is value from 0 to Maximum Channel Sensor Number. 0xFF means skip this channel scanning.

C Prototype:

```
void CSD2X_ScanSensor(BYTE bSensorLeft, byte bSensorRight);
```

Assembly:

```
mov A,  bSensorLeft
mov X,  bSensorRight
lcall CSD2X_ScanSensor
```

Parameters:

A => Sensor Number Left

X => Sensor Number Right

Return Value:

None

Side Effects

**

CSD2X_ScanAllSensors

Description:

Scans all of the configured sensors by calling CSD2X_ScanSensor() for each sensor index.

C Prototype:

```
void CSD2X_ScanAllSensors();
```

Assembly:

```
lcall CSD2X_ScanAllSensors
```

Parameters:

None

Return Value:

None

Side Effects

**

CSD2X_SetScanMode

Description:

Set clocking according to required Scan Speed and Resolution.

C Prototype:

```
void CSD2X_SetScanMode(BYTE bSpeed, BYTE bResolution);
```

Assembly:

```
mov X, bResolution
mov A, bSpeed
lcall CSD2X_SetScanMode
```

Parameters:

bSpeed: Scanning Speed

The following constants are given for the bSpeed parameter:

Constant	Value
CSD2X_FAST_SPEED	0x01
CSD2X_NORMAL_SPEED	0x02
CSD2X_SLOW_SPEED	0x03

bResolution: Scanning Resolution. Set this value to the required number of bits of resolution. The value range depends on the user module configuration.

The following possible constants are given for the bResolution parameter for First Order Modulator:

Constant	Value
CSD2X_8_BIT_RESOLUTION	8
CSD2X_10_BIT_RESOLUTION	10
CSD2X_12_BIT_RESOLUTION	12

The following possible constants are given for the bResolution parameter for Second Order Modulator:

Constant	Value
CSD2X_10_BIT_RESOLUTION	10
CSD2X_12_BIT_RESOLUTION	12
CSD2X_14_BIT_RESOLUTION	14

Return Value:

None

Side Effects

**

CSD2X_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 CSD2X_UpdateSensorBaseline() is called, a difference count is calculated by subtracting the previous baseline from the raw count value. This difference is stored in the CSD2X_waSnsDiff[] array and is provided to you.

2. If Sensors Autoreset is disabled, each time CSD2X_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. When the accumulated difference counts in the virtual bucket reaches the BaselineUpdateThreshold, the baseline is incremented by one and the bucket is reset to 0.

4. If the difference count is below the noise threshold, the value held in the 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 CSD2X_UpdateSensorBaseline(BYTE bSensor)
```

Assembly:

```
mov   A,   bSensor
lcall  CSD2X_UpdateSensorBaseline
```

Parameter:

A => Sensor Number

Return Value:

None

Side Effects:

**

CSD2X_UpdateAllBaselines

Description:

Uses the CSD2X_bUpdateSensorBaseline() function to update the baselines for all sensors.

C Prototype:

```
void  CSD2X_UpdateAllBaselines()
```

Assembly:

```
lcall CSD2X_UpdateAllBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD2X_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 CSD2X_baSnsOnMask[] array.

C Prototype:

```
BYTE  CSD2X_bIsSensorActive(BYTE bSensor)
```

Assembly:

```
mov  A,  bSensor
lcall CSD2X_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:

**

CSD2X_bIsAnySensorActive

Description:

Checks the difference count array for all sensors compared to their finger threshold. Calls CSD2X_bIsSensorActive() for each sensor so the CSD2X_baSnsOnMask[] array is up to date after calling this function.

C Prototype:

```
BYTE  CSD2X_bIsAnySensorActive()
```

Assembly:

```
lcall CSD2X_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:

**

CSD2X_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 CSD2X Wizard. This function is available only if slider is defined by the CSD2X Wizard.

C Prototype:

```
WORD  CSD2X_wGetCentroidPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall CSD2X_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 CSD2X 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 CSD2X_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.

CSD2X_wGetRadialPos

Description:

Checks a difference array for a centroid. If one exists, the centroid position is calculated to the resolution specified in the CSD2X Wizard. This function is available only for radial slider that is defined by the CSD2X Wizard.

C Prototype:

```
WORD  CSD2X_wGetRadialPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A, bSnsGroup
lcall CSD2X_wGetRadialPos
```

Parameters:

bSnsGroup A => Slider Number

This parameter is a number of radial slider you are working with. You can get its number through CSD2X 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 CSD2X 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.

CSD2X_wGetRadialInc

Description:

Returns actual finger shift, the difference between current and previous finger positions. This function works in pair with CSD2X_wGetRadialPos() and takes data generated by the latter (data is saved in internal variables).

C Prototype:

```
WORD  CSD2X_wGetRadialInc(BYTE bSnsGroup)
```

Assembly:

```
mov  A, bSnsGroup
lcall CSD2X_wGetRadialInc
```

Parameters:

bSnsGroup A => Slider Number

This parameter is a number of radial slider you are working with. You can get its number through CSD2X 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 CSD2X_wGetRadialPos() returned -1 (FFFFh)) or there is no touch at the moment (this time CSD2X_wGetRadialPos() returned -1 (FFFFh))

Side Effects:

The routine should be called only after CSD2X_wGetRadialPos() API. Because it uses internal data CSD2X_waSliderPrevPos and CSD2X_waSliderCurrPos that are set by the CSD2X_wGetRadialPos().

CSD2X_InitializeSensorBaseline

Description:

Loads the CSD2X_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 CSD2X_InitializeSensorBaseline(BYTE bSensor)
```

Assembly:

```
mov A, bSensor
lcall CSD2X_InitializeSensorBaseline
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

CSD2X_InitializeBaselines

Description:

Loads the CSD2X_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 CSD2X_InitializeBaselines()
```

Assembly:

```
lcall CSD2X_InitializeBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD2X_SetDefaultFingerThresholds

Description:

Loads the CSD2X_baBtnFThreshold[] array with the FingerThreshold parameter value. This function must be called prior to scanning if the CSD2X_baBtnFThreshold[] array is not manually loaded with custom values.

C Prototype:

```
void CSD2X_SetDefaultFingerThresholds()
```

Assembly:

```
lcall CSD2X_SetDefaultFingerThresholds
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD2X_SetPGARefLeft

Description:

This function overwrites the user module parameter settings left PGA reference value. Use it if some sensors need to be scanned with a different reference setting. This function can be used in conjunction with CSD2X_ScanSensor(). This function is available in Rb configurations.

C Prototype:

```
void CSD2X_SetPGARefLeft(BYTE bRefValue);
```

Assembly:

```
mov     A, bRefValue
lcall    CSD2X_SetPGARefLeft
```

Parameters:

bRefValue - sets the reference value. Accepted values are 1(AGND) or 2(VSS).

Return Value:

None

Side Effects:

**

CSD2X_SetPGARefRight

Description:

This function overwrites the user module parameter settings right PGA reference value in Rb configuration. Use it if some sensors need to be scanned with a different reference setting. This function can be used in conjunction with CSD2X_ScanSensor(). This function is available in Rb configurations.

C Prototype:

```
void CSD2X_SetPGARefRight(BYTE bRefValue);
```

Assembly:

```
mov     A, bRefValue
lcall    CSD2X_SetPGARefRight
```

Parameters:

bRefValue - sets the reference value. Accepted values are 1(AGND) or 2(VSS).

Return Value:

None

Side Effects:

**

CSD2X_Calibrate

Description:

This function overwrites the user module parameter settings for DAC values for move raw counts on half of range. This function should be run before baseline initialization. This function is not available in Rb configurations.

C Prototype:

```
void CSD2X_Calibrate(void);
```

Assembly:

```
lcall    CSD2X_Calibrate
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD2X_ClearSensors

Description:

Clears all sensors to the non-sampling state by sequentially calling CSD2X_wGetPortPinLeft() or CSD2X_wGetPortPinRight() and CSD2X_DisableSensor() for each of the sensors.

C Prototype:

```
void  CSD2X_ClearSensors()
```

Assembly:

```
lcall  CSD2X_ClearSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSD2X_wReadSensor

Description:

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

C Prototype:

```
WORD  CSD2X_wReadSensor(BYTE bSensor)
```

Assembly:

```
mov A, bSensor
lcall CSD2X_wReadSensor
```

Parameters:

A => Sensor Number

Return Value:

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

Side Effects:

**

CSD2X_wGetPortPinLeft

Description:

Returns the port number and pin mask for a given sensor. The passed parameter indexes and selects the data from the CSD2X_Sensor_Table_Left[]. The return value can be passed to the CSD2X_EnableSensor(), CSD2X_DisableSensor(). This function is available in single channel configurations only.

C Prototype:

```
WORD  CSD2X_wGetPortPinLeft(BYTE bSensor)
```

Assembly:

```
mov  A,  bSensor
lcall     CSD2X_wGetPortPinLeft
```

Parameters:

bSensor – Sensor Number, the range is 0 to (n – 1) where 'n' is the total of the number of sensors set in the CSD2X Wizard for left channel (which includes number of slider segments connected to left channel). The sensor number is used by CSD2X_wGetPortPinLeft() to determine port and bit mask for the selected active sensor.

Return Value:

A => Sensor Bitmap

X => Port Number

Side Effects:

**

CSD2X_wGetPortPinRight

Description:

Returns the port number and pin mask for a given sensor that is connected to right channel. The passed parameter indexes and selects the data from the CSD2X_Sensor_Table_Right[]. The return value can be passed to the CSD2X_EnableSensor(), CSD2X_DisableSensor(). This function is available in double channel configurations only.

C Prototype:

```
WORD  CSD2X_wGetPortPinRight(BYTE bSensor)
```

Assembly:

```
mov  A,  bSensor
lcall     CSD2X_wGetPortPinRight
```

Parameters:

bSensor – Sensor Number, the range is 0 to (n – 1) where 'n' is the total of the number of sensors set in the CSD2X Wizard for right channel (which includes number of slider segments connected to right channel). The sensor number is used by CSD2X_wGetPortPinRight() to determine port and bit mask for the selected active sensor.

Return Value:

A => Sensor Bitmap

X => Port Number

Side Effects:

**

CSD2X_EnableSensor

Description:

Configures the selected sensor to measure during the next measurement cycle. The port and sensor can be selected using the CSD2X_wGetPortPinLeft() or CSD2X_wGetPortPinRight() 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  CSD2X_EnableSensor(BYTE bMask, BYTE bPort)
```

Assembly:

```
mov  X,  bPort
mov  A,  bMask
lcall  CSD2X_EnableSensor
```

Parameters:

A => Sensor Bitmap

X => Port Number

Return Value:

None

Side Effects:

**

CSD2X_DisableSensor

Description:

Disables the sensor selected by the CSD2X_wGetPortPinLeft() or CSD2X_wGetPortPinRight() 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.

C Prototype:

```
void  CSD2X_DisableSensor(BYTE bMask, BYTE bPort)
```

Assembly:

```
mov  X,  bPort
mov  A,  bMask
lcall  CSD2X_DisableSensor
```

Parameters:

A => Sensor Bitmap

X => Port Number

Return Value:

None

Side Effects:

**

CSD2X_SetRefValue

Description:

Set reference voltage by adjusting compare value for PRS or PWM.

C Prototype:

```
void CSD2X_SetRefValue(BYTE bRefValue);
```

Assembly:

```
mov A, bRefValue
lcall CSD2X_SetRefValue
```

Parameters:

bRefValue - Ref voltage code from 0 to 8 (0 -> Ref = 1/4 * Vcc, 8 -> Ref = 3/4 * Vcc)

Return Value:

None

Side Effects

**

APIs for background scanning and FMEA:

BYTE CSD2X_bIsScanComplete(void)

Description

Checks the scan complete flag in the bScanComplete variable and returns TRUE of FASLE to customer. Also this API should reset this flag after call.

C Prototype

```
BYTE CSD2X_bIsScanComplete(void);
```

Assembly

```
lcall  CSD2X_bIsScanComplete
```

Parameters

None

Returns

This parameter returns a value of 1 if active and 0 if not active. If A => 1, scan is complete; if A = 0, scan is incomplete.

BYTE CSD2X_bFMEA_CheckGndShort(void)

Description

This API checks if any of the sensors (including shield electrode, if enabled) is shorted to ground.

C Prototype

```
BYTE CSD2X_bFMEA_CheckGndShort(void);
```

Assembly

```
lcall  CSD2X_bFMEA_CheckGndShort
```

Parameters

None

Returns

This parameter returns a value of 1 if any sensor is shorted to GND. It returns a value of 0 if none of the sensors is shorted to GND.

BYTE CSD2X_bFMEA_CheckVddShort(void)

Description

This API checks if any of the sensors (including shield electrode, if enabled) is shorted to Vdd.

C Prototype

```
BYTE CSD2X_bFMEA_CheckVddShort(void);
```

Assembly

```
lcall  CSD2X_bFMEA_CheckVddShort
```

Parameters

None

Returns

This parameter returns a value of 1 if any sensor is shorted to Vdd. It returns a value of 0 if none of the sensors is shorted to Vdd.

BYTE CSD2X_bFMEA_CheckSensorShort(void)

Description

This API checks if any of the sensors (including shield electrode, if enabled) is shorted to another sensor.

C Prototype

```
BYTE CSD2X_bFMEA_CheckSensorShort(void);
```

Assembly

```
lcall  CSD2X_bFMEA_CheckSensorShort
```

Parameters

None

Returns

This parameter returns a value of 1 if any sensor is shorted to another sensor. It returns a value of 0 if no sensor is shorted to another sensor.

The CSD2X_bFMEA_CheckSensorShort API does not check the sensor-to-sensor short correctly if one of the sensors is shorted to V_{DD}. Use the CSD2X_bFMEA_CheckVddShort() and CSD2X_bFMEA_CheckGndShort() APIs to check Gnd or Vdd short.

BYTE CSD2X_bFMEA_Left_Cmod_Check(void)

BYTE CSD2X_bFMEA_Right_Cmod_Check(void)

Description

This API is available only if you select the IDAC method. This checks C_{mod} for short test and also checks whether it is within the valid range. The valid range is defined by recommended minimum and maximum C_{mod} values with an error of 20%. The minimum C_{mod} value is 1 nF and maximum C_{mod} value is 20 nF. The Cmod range's maximum and minimum values are defined in the CSD2X.inc file and the user can change them during generate/build in the PSoC Designer.

CSD2X_bFMEA_Left_Cmod_Check() and CSD2X_bFMEA_Right_Cmod_Check() functions update the CSD2X_wCmod_L_Val and CSD2X_wCmod_R_Val WORD variables. These variables contain the value of C_{mod} in nF scaled to 100. For example, CSD2X_wCmod_Val = 2200 corresponds 22 nF (or 0.022 uF). This value is valid only if the CSD2X_bFMEA_Left_Cmod_Check() or CSD2X_bFMEA_Left_Cmod_Check() API returns bRetVal.4.

C Prototype

```
BYTE CSD2X_bFMEA_Left_Cmod_Check(void);
BYTE CSD2X_bFMEA_Right_Cmod_Check(void)
```

Assembly

```
lcall  CSD2X_bFMEA_Left_Cmod_Check
lcall  CSD2X_bFMEA_Right_Cmod_Check
```

Parameters

None

Returns

bRetVal.0:C_{mod} shorted to GND

bRetVal.1:C_{mod} shorted to Vdd

bRetVal.4:Cmod within +20%

bRetVal.8:Cmod is low (under the valid range) or Cmod is disconnected

bRetVal.16:C_{mod} is high (over the valid range)

Constant	Value	Description
CSD2X_LEFT_CMOD_SHORTED_TO_GND	0	Left Cmod shorted to GND
CSD2X_ LEFT_CMOD_SHORTED_TO_VDD	1	Left Cmod shorted to Vdd
CSD2X_ LEFT_CMOD_WITHIN_20	4	Left Cmod within +20%
CSD2X_ LEFT_CMOD_IS_LOW	8	Left Cmod is out of range
CSD2X_ LEFT_CMOD_IS_HIGH	16	Left Cmod is out of range
CSD2X_RIGHT_CMOD_SHORTED_TO_GND	0	Right Cmod shorted to GND
CSD2X_RIGHT_CMOD_SHORTED_TO_VDD	1	Right Cmod shorted to Vdd
CSD2X_RIGHT_CMOD_WITHIN_20	4	Right Cmod within +20%
CSD2X_RIGHT_CMOD_IS_LOW	8	Right Cmod is out of range
CSD2X_RIGHT_CMOD_IS_HIGH	16	Right Cmod is out of range

Notes

1. The background scanning should be completed before using CSD2X_bFMEA_Left_Cmod_Rb_Check() and CSD2X_bFMEA_Right_Cmod_Rb_Check() APIs.
2. The Global Interrupts are disabled while checking Cmod range.

BYTE CSD2X_bFMEA_Left_Cmod_Rb_Check(void)

BYTE CSD2X_bFMEA_Right_Cmod_Rb_Check(void)

Description

This API is available only if you select the Rb method and enable the FMEA feature. This checks C_{mod} and Rb values for short test and also checks whether the product is within the valid range. The valid range is defined by C_{mod} and Rb product with an error of 40%. The minimum value is $3.3*10^{-6}$ sec and maximum value is $700*10^{-6}$ sec.

CSD2X_bFMEA_Left_Cmod_Rb_Check() and CSD2X_bFMEA_Right_Cmod_Rb_Check() update WORD CSD2X_wCmod_Rb_L_Val and CSD2X_wCmod_Rb_R_Val variables. These variables contain the product of C_{mod} and Rb in µsec calculated during the CSD2X_bFMEA_Left_Cmod_Rb_Check() and CSD2X_bFMEA_Right_Cmod_Rb_Check() functions. For example CSD2X_wCmod_Rb_Val contains 120 which corresponds to 120 10-6 sec (or 120 usec). This value is valid only if CSD2X_bFMEA_Left_Cmod_Rb_Check() (or CSD2X_bFMEA_Right_Cmod_Rb_Check()) API returns bRetVal.4.

C Prototype

```
BYTE CSD2X_bFMEA_Left_Cmod_Rb_Check(void);
BYTE CSD2X_bFMEA_Right_Cmod_Rb_Check(void)
```

Assembly

```
lcall  CSD2X_bFMEA_Left_Cmod_Rb_Check
lcall  CSD2X_bFMEA_Right_Cmod_Rb_Check
```

Parameters

None

Returns

bRetVal.0:C_{mod} shorted to GND

bRetVal.1:C_{mod} shorted to Vdd

bRetVal.2:Rb shorted to Gnd

bRetVal.3:Rb shorted to Vdd

bRetVal.4:Cmod and Rb product is within +40%

bRetVal.8: Cmod and Rb product is low (under the valid range) or Cmod is disconnected

bRetVal.16:Cmod and Rb product is high (over the valid range) or Rb is disconnected

Constant	Value	Description
CSD2X_LEFT_CMOD_SHORTED_TO_GND	0	Left Cmod shorted to GND
CSD2X_ LEFT_CMOD_SHORTED_TO_VDD	1	Left Cmod shorted to Vdd
CSD2X_LEFT_RB_SHORTED_TO_GND	2	Left Rb shorted to Gnd
CSD2X_LEFT_RB_SHORTED_TO_VDD	3	Left Rb shorted to Vdd
CSD2X_LEFT_CMODRB_WITHIN_40	4	Left Cmod and Rb product is within +40%

Constant	Value	Description
CSD2X_LEFT_CMODRB_IS_LOW	8	Left Cmod and Rb product is out of range
CSD2X_LEFT_CMODRB_IS_HIGH	16	Left Cmod and Rb product is out of range
CSD2X_RIGHT_CMOD_SHORTED_TO_GND	0	Right Cmod shorted to GND
CSD2X_RIGHT_CMOD_SHORTED_TO_VDD	1	Right Cmod shorted to Vdd
CSD2X_RIGHT_RB_SHORTED_TO_GND	2	Right Rb shorted to Gnd
CSD2X_RIGHT_RB_SHORTED_TO_VDD	3	Right Rb shorted to Vdd
CSD2X_RIGHT_CMODRB_WITHIN_40	4	Right Cmod and Rb product is within +40%
CSD2X_RIGHT_CMODRB_IS_LOW	8	Right Cmod and Rb product is out of range
CSD2X_RIGHT_CMODRB_IS_HIGH	16	Right Cmod and Rb product is out of range

Notes

1. The background scanning should be completed before using CSD2X_bFMEA_Left_Cmod_Rb_Check() and CSD2X_bFMEA_Right_Cmod_Rb_Check() APIs.
2. If Rb is less than 6 k ohms, then API can return "Cmod shorted to GND" instead of "Rb shorted to Gnd". An additional Pull Up resistor to Cmod is needed to differ between Rb short to Gnd and Cmod short to Gnd.
3. If Rb is less than 6 k ohms, then API can return "Cmod shorted to Vdd" instead of "Rb shorted to Vdd". An additional Pull Down resistor to Cmod is needed to differ between Rb short to Vdd and Cmod short to Vdd.
4. The Global Interrupts are disabled while checking Cmod and Rb range.

The CSD2X_wCmod_Rb_L_Val and CSD2X_wCmod_Rb_R_Val variables contain Rb*Cmod value with large error up to 80% (in the configurations with Rb) for Rb*Cmod values that are close to the low limit of the valid range.

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 CSD2X 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;
    CSD2X_Start();
    CSD2X_InitializeBaselines() ; //scan all sensors first time, init baseline
    CSD2X_SetDefaultFingerThresholds() ;
    //
    // Loop Forever
    //
    while (1)
    {
        CSD2X_ScanAllSensors();    //scan all sensors in array (buttons and sliders)
        CSD2X_UpdateAllBaselines(); //Update all baseline levels;

        //detect if any sensor is pressed
        if(CSD2X_bIsAnySensorActive())
        {
        // Add user code here to proceed with sensor touching
        }

        // now we are ready to send all status variables to chart program
        // communication here
        //
        // OUTPUT CSD2X_waSnsResult[x] <- Raw Counts
        // OUTPUT CSD2X_waSnsDiff[x] <- Difference
        // OUTPUT CSD2X_waSnsBaseline[x] <- Baseline
        // OUTPUT CSD2X_baSnsOnMask[x] <- Sensor On/Off
    }
}
```

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

```
//-------------------------------------------------------------------------
// Sample C code for the CSD2X module
//-------------------------------------------------------------------------

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

void main(void)
{
    M8C_EnableGInt;

    CSD2X_Start(); // Start CSD2X UM
    CSD2X_SetDefaultFingerThresholds(); // Set default thresholds for buttons
    // Initialize baseline for sensor number "3"
    CSD2X_InitializeSensorBaseline(3);

    while (1)
    {
        // Scan continuously sensor number "3" which is connected
        CSD2X_ScanSensor(0xFF, 3);
        CSD2X_UpdateSensorBaseline(3); // Update Baseline for sensor 3
        if(CSD2X_bIsSensorActive(3)) // check if sensor 3 is touched
```

```
        {
        // Add user code here to proceed with buttons pressing
        }
    }
}
```

Example 3. The following example demonstrates the ability to set the different Finger Threshold levels for each sensor. It is useful when different sensors are placed on different locations and some sensors are more sensitive than others.

```
//--------------------------------------------------------------------------
// Sample C code for the CSD2X 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;

    CSD2X_Start();
    CSD2X_InitializeBaselines();

    // set finger threshold for sensor "0"
    CSD2X_baBtnFThreshold[0] = 10;
    // set finger threshold for sensor "1"
    CSD2X_baBtnFThreshold[1] = 20;
    // set finger threshold for sensor "2"
    CSD2X_baBtnFThreshold[2] = 30;
    // set finger threshold for sensor "3"
    CSD2X_baBtnFThreshold[3] = 40;
    // set finger threshold for sensor "4"
    CSD2X_baBtnFThreshold[4] = 50;
    // set finger threshold for sensor "5"
    CSD2X_baBtnFThreshold[5] = 255;
    // set finger threshold for sensor "6"
    CSD2X_baBtnFThreshold[6] = 200;

        while (1) {
        // Scan continuously all sensors
        CSD2X_ScanAllSensors();
        CSD2X_UpdateAllBaselines();
        //detect if any sensor is pressed
        if(CSD2X_bIsAnySensorActive())
        {
        // Add user code here to proceed the buttons pressing
        }
    }
}
```

Example 4. The following example shows how the CSD2X_wGetRadialPos() and CSD2X_wGetRadialInc() APIs work. You should have one radial slider defined in the CSD2X Wizard.

```
//-------------------------------------------------------------------------
// Sample C code for the CSD2X module
// Define one radial slider in CSD2X Wizard
//-------------------------------------------------------------------------

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

WORD Pos, Inc;
void main(void)
{
    M8C_EnableGInt;
    CSD2X_Start();

    CSD2X_InitializeBaselines() ;
    CSD2X_SetDefaultFingerThresholds();

        while (1)
        {
        CSD2X_ScanAllSensors();
        CSD2X_UpdateAllBaselines();

        Pos = CSD2X_wGetRadialPos(1);
        Inc = CSD2X_wGetRadialInc(1);
    }
}
```

Example 5. This code starts the user module and continuously scans the sensors when FMEA and background scanning are enabled. The communication section can be used to communicate values to a PC charting tool.

```
//-------------------------------------------------------------------------
// Sample C code for the CSD2X 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;
    CSD2X_Start();
    CSD2X_InitializeBaselines() ; //scan all sensors first time, init baseline
    CSD2X_SetDefaultFingerThresholds();
    if(CSD2X_bIsScanComplete())
    {
        //Short tests
        if (CSD2X_bFMEA_CheckGndShort())
        {
            // Add user code here to Sensor to Ground short alarm
            // CSD2X_baSensorShortGnd[0]
            // contains contains the masked bits for sensors 0 through 7
        }
```

```
        if (CSD2X_bFMEA_CheckVddShort())
        {
            // Add user code here to Sensor to Vdd short alarm
            // CSD2X_baSensorShortVdd[0]
            // contains contains the masked bits for sensors 0 through 7
        }
        if (CSD2X_bFMEA_CheckSensorShort())
        {
           // Add user code here to Sensor to Sensor short alarm
        }
        if (CSD2X_bFMEA_Left_Cmod_Check() != 4)
        {
            // Add user code here to alarm Cmod issues
            // CSD2X_wCmod_Val
        }
        if (CSD2X_bFMEA_Right_Cmod_Check() != 4)
        {
            // Add user code here to alarm Cmod and Rb issues
        }
    }
    //
    // Loop Forever
    //
    while (1)
    {
            CSD2X_ScanAllSensors(); //scan all sensors in array (buttons and sliders)

            if(CSD2X_bIsScanComplete())
            {
                CSD2X_UpdateAllBaselines(); //Update all baseline levels;
            }

            //detect if any sensor is pressed
            if(CSD2X_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 CSD2X_waSnsResult[x] <- Raw Counts
        // OUTPUT CSD2X_waSnsDiff[x] <- Difference
        // OUTPUT CSD2X_waSnsBaseline[x] <- Baseline
        // OUTPUT CSD2X_baSnsOnMask[x] <- Sensor On/Off

        // Do other tasks
    }
}
```

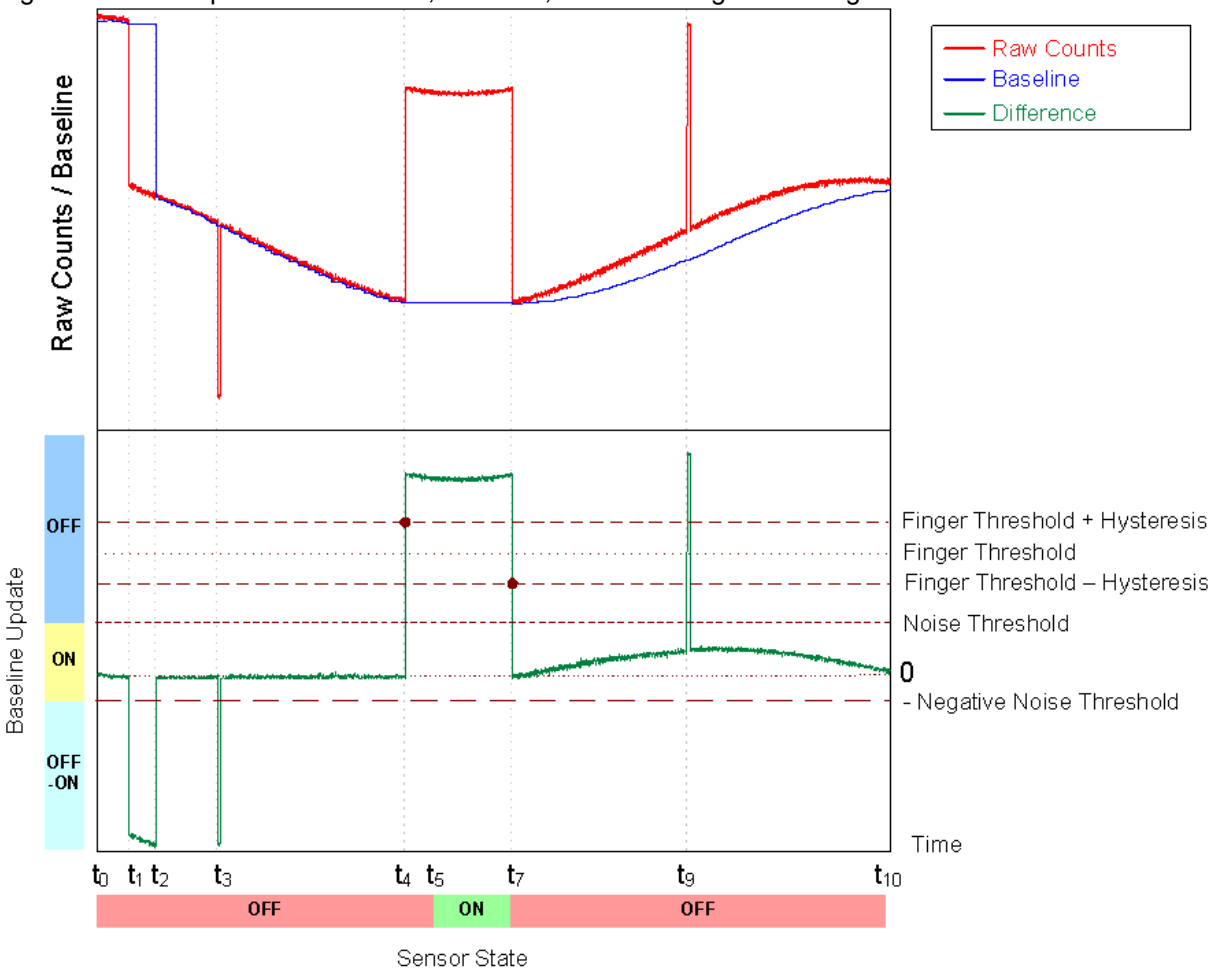
Appendices

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. Some of this information may be moved into application notes in the future.

Interaction of CSD2X 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 15. 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.

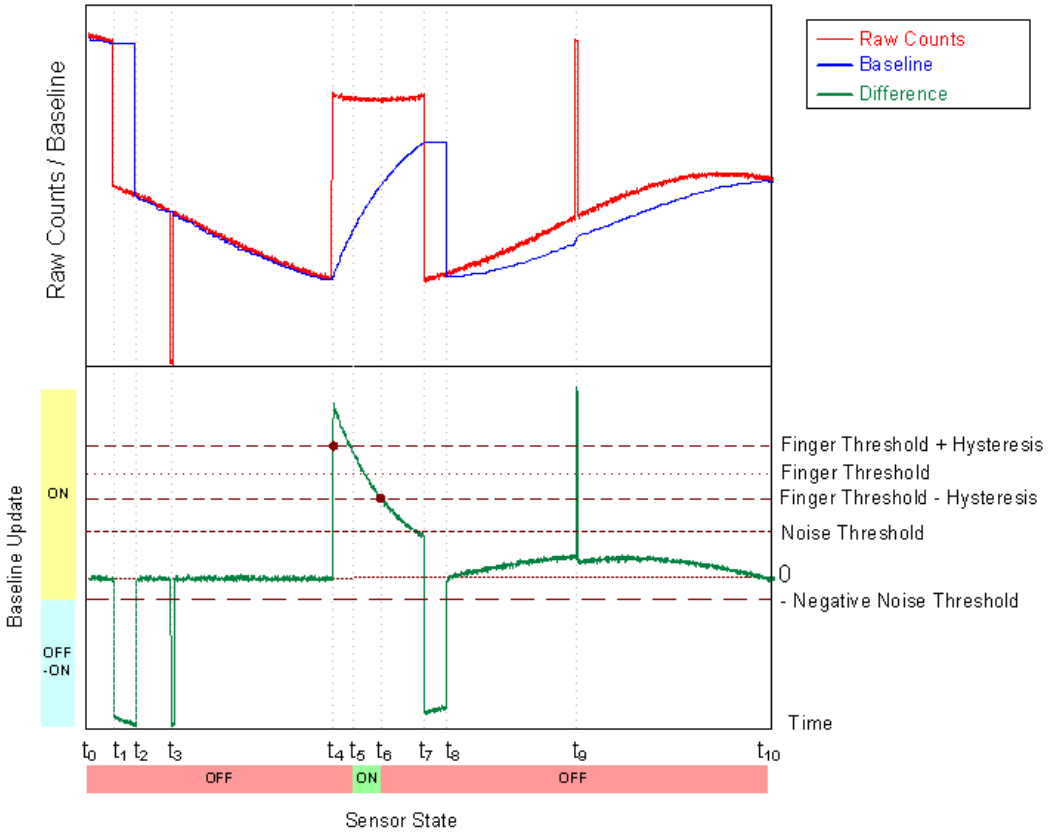
At t_1 the raw count drops sharply and the negative difference exceeds the NegativeNoiseThreshold. This situation can happen if the device is powered on when a finger is on the sensor and then the finger is removed after a period of time. At this time the baseline update mechanism is frozen and an internal 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 16. 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.

Double Channel Scanning

Double channel scanning is done as a synchronous block function.

Synchronous means that the left and right channel sensors are scanned at the same time and scanning begins on the next pair of sensors only after the previous pair is scanned. It is possible that the left and right sensors have a different resolution and scanning time. In this case the “Scan Sensor" function waits for the slower sensor and doesn’t do anything with the other. If the left and right sensors arrays consist of different numbers of sensors then the “ScanAllSensors” function:

- Scans all possible sensors in pairs
- If there are more sensors on one side than the other, scans the remaining sensors on that channel one at a time

The scanning starts from the end of the array. The sensors position are assigned in the GUI according to sensor placement from left top to right bottom. To make sure that your dual channel scanning is as efficient as possible, do the following:

- Have the same number of sensors in each channel
- Arrange sensors so that sensors with longer scanning times are in pairs
- Place all slider segments to the same channel
- Place large sensors on the left channel
- If different references are used, place sensors with higher reference values first

Version History

Version	Originator	Description
1.0	DHA	Initial version.
1.10	DHA	1. The resolution maximum is (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. 2. Removed 0.5 shift and added compensation for negative values. 3. The connection between AnalogComparatorColumn and GlobalOutput line is displayed in the Interconnect view.
1.10.b	DHA	Added help file to wizard.
1.20	DHA	1. Transferred the DiplexTable from "AREA UserModules" to "AREA lit". 2. Set the default "DiplexTable" parameter value to 0x0112. 3. Added the "DiplexUsed" parameter to improve code compression. 4. Changed the call instruction to lcall in the CSD2X_InitializeBaselines API to add support for CSD2X_28IEPRS on CY8C28xxx devices. 5. Updated descriptions of the CSD2X_wGetPortPinLeft() and CSD2X_wGetPortPinRight() API functions.
1.30	DHA	1. Updated area declarations to support Imagecraft optimization. 2. Updated this user module datasheet to add symbolic names for the Resolution parameter. 3. Updated the function description for SetScanMode() API. 4. Added the max value limitation on the Resolution parameter for the Slider and Radial Slider. 5. Updated the user module wizard help. Added a description of the slider resolution parameter min/max values.
1.30.b	DHA	Added note that 1X value of IDAC range parameter is not recommended for new designs.
2.00	DHA	1. Added FMEA and Background Scanning functionality to the user module. 2. Updated user module block diagram in the user module datasheet. 3. Added `@INSTANCE_NAME`_IDAC_RANGE to the inc file. 4. Renamed "Slider Settings" tab to "Sensor Settings". 5. Explained limitations of dynamic reconfiguration in the datasheet.

Version	Originator	Description
2.10	MYKZ	1. Added Resume() function to User Module API. 2. Fixed problem with saving information for sliders. 3. Updated baseline algorithm to check for negative difference counts. 4. Added CUR_PP and IDX_PP handling to the interrupt service routine to cover the case when background scanning is enabled. 5. Added build error message when user attempts to build project without first calling the user module wizard. 6. Optimized Start User Modul+L143e function code. 7. Added decimator placement initialization for CY8C28xxx families. 8. Updated UM Wizard sliders setting algorithm to take into account free pins. 9. Added User Code section in User Module *.inc file to provide the ability to customize the Cmod range.

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 © 2009-2013 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.