# Dual CapSense® Sigma-Delta Datasheet CSD2X V 3.10

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

| Resources | PSoC® Blocks | | | | API Memory (Bytes) | | Pins (in addition to sensors) |
|---|---|---|---|---|---|---|---|
| | CapSense® | I²C/SPI | Timer | Comparator | Flash | RAM | |
| CY8C21x45, CY8C22x45 | | | | | | | |
| Single Channel with IDAC* | 1 | – | – | 1 | 1006 | 29 | 1 |
| Autocalibration enabled | 1 | – | – | 1 | 1198 | 35 | 1 |
| Double Channel with IDAC* | 2 | – | – | 2 | 1390 | 30 | 2 |
| Autocalibration enabled | 2 | – | – | 2 | 1576 | 36 | 2 |
| Single Channel with Rb resistor* | 1 | – | – | 1 | 1000 | 28 | 2 |
| Double Channel with Rb resistor* | 2 | – | – | 2 | 1400 | 30 | 4 |
| Each additional CapSense button | – | – | – | – | 8 | 12 | 1 |
| Static code and RAM increase when capacitive slider with 5 elements is used | – | – | – | – | 613 | 90 | 5 |
| Each additional slider element | – | – | – | – | 8 | 12 | 1 |
| Static code and RAM increase when slider diplexing is used (5 sensors) | – | – | – | – | 10 | – | – |

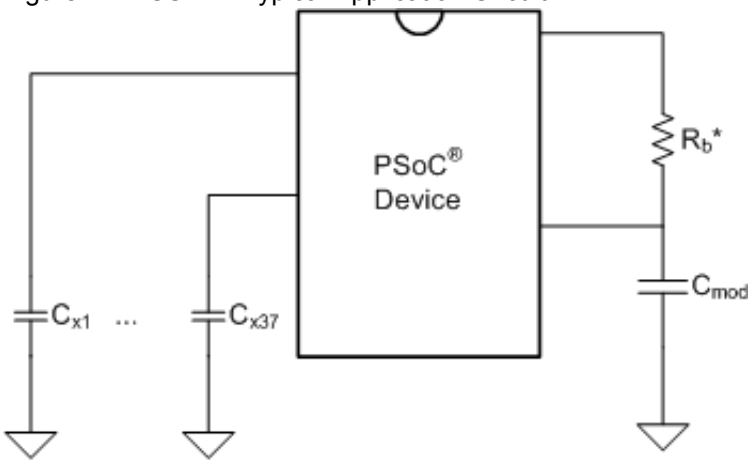
*Calculated for configuration with 1 button

## Features and Overview

- Scan 1 to 37 capacitive sensors
- Sensing possible with up to 15 mm glass overlay
- Proximity detection to 20 cm with a wire-based sensor
- Reduced scan time with dual-channel scanning
- 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
- FMEA features such as Short test, Cmod test, and Cmod_Rb test
- Enhanced hardware architecture to support background scan
- Easily adjustable operational parameters
- 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) gives 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 single-channel CapSense scanning and dual-channel CapSense scanning (scans two sensors simultaneously).

Figure 1. CSD2X Typical Application Circuit




* This resistor is available in $R_b$ configurations only

## 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. Configure the settings for each sensor.
6. Set pins and global parameters. Read all the parameter descriptions and follow the 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 given 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 given 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 a 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 compensates 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 a 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 enhancing resolution. The functions also give further interpolation of resolved sensor position between physical sensor locations.

The following document is recommended reading before you use the CSD2X User Module for the first time:

*'CapSense System' section in the CY8C22X45 and CY8C21345 PSoC Programmable System-on-Chip Technical Reference Manual*

The following design guides are recommended after reading the CSD2X 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

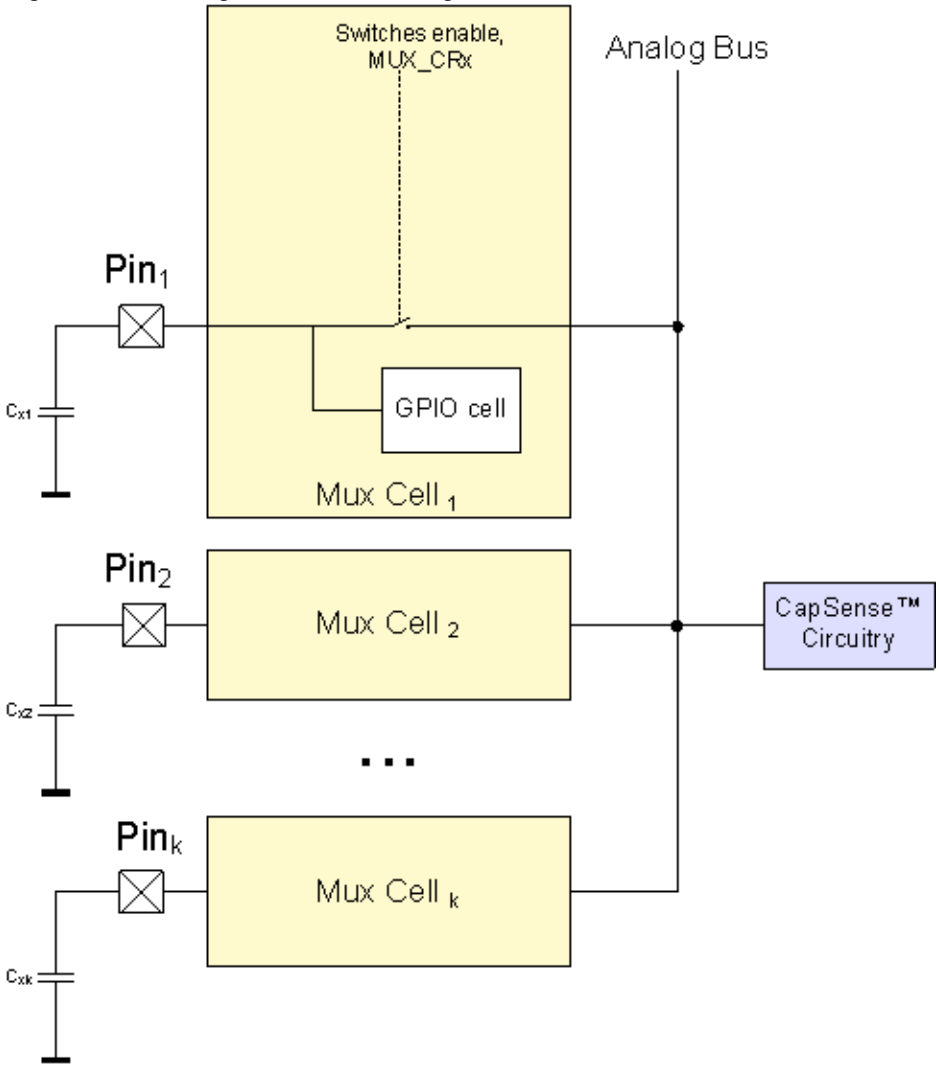
The decision logic is implemented in firmware. The firmware analyzes capacitance measurement, tracks the slow capacitance change due to environmental factors, and runs decision logic to detect button touches and calculate slider position.

### Scanning an Array of Sensors

The CY8C22x45 family of devices has 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.

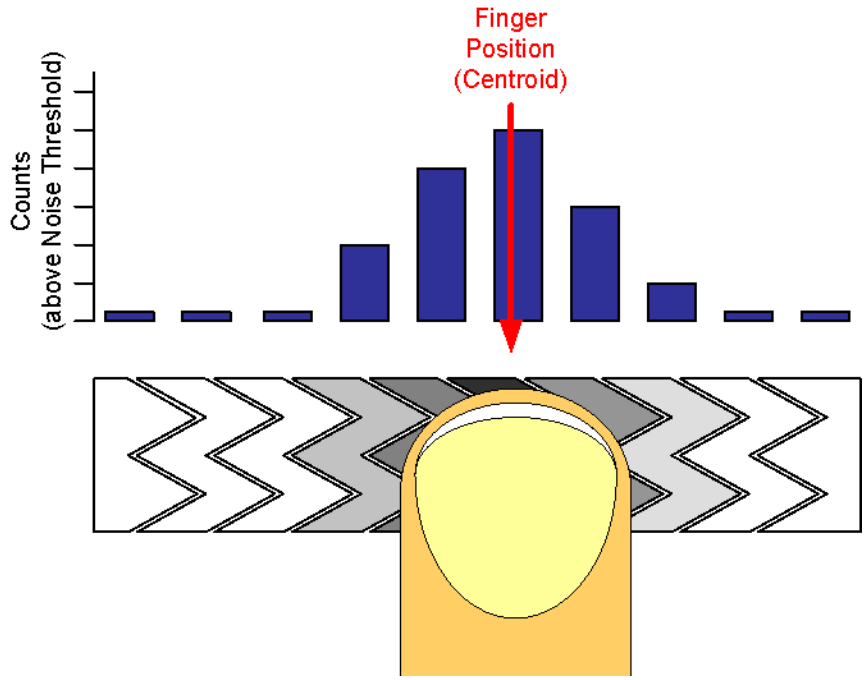
Figure 2. Analog Bus with Precharge Switches




### *Sliders*

Sliders are used for controls that require 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 the sensor sliders is five and the upper limit is simply the number of sensor positions available on the PSoC device selected.

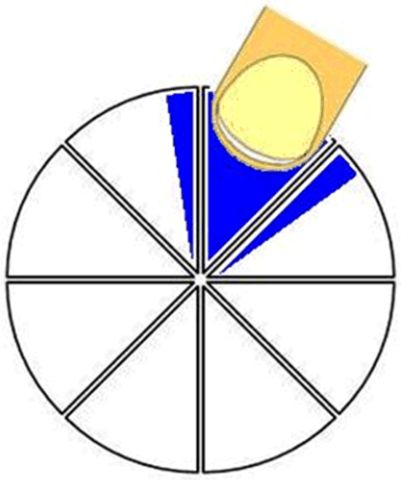
Figure 3. Ordering Physical Sensor Locations




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



The CSD2X UM has two slider types: linear and radial. Radial sliders are similar to linear sliders, except for these differences. 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.

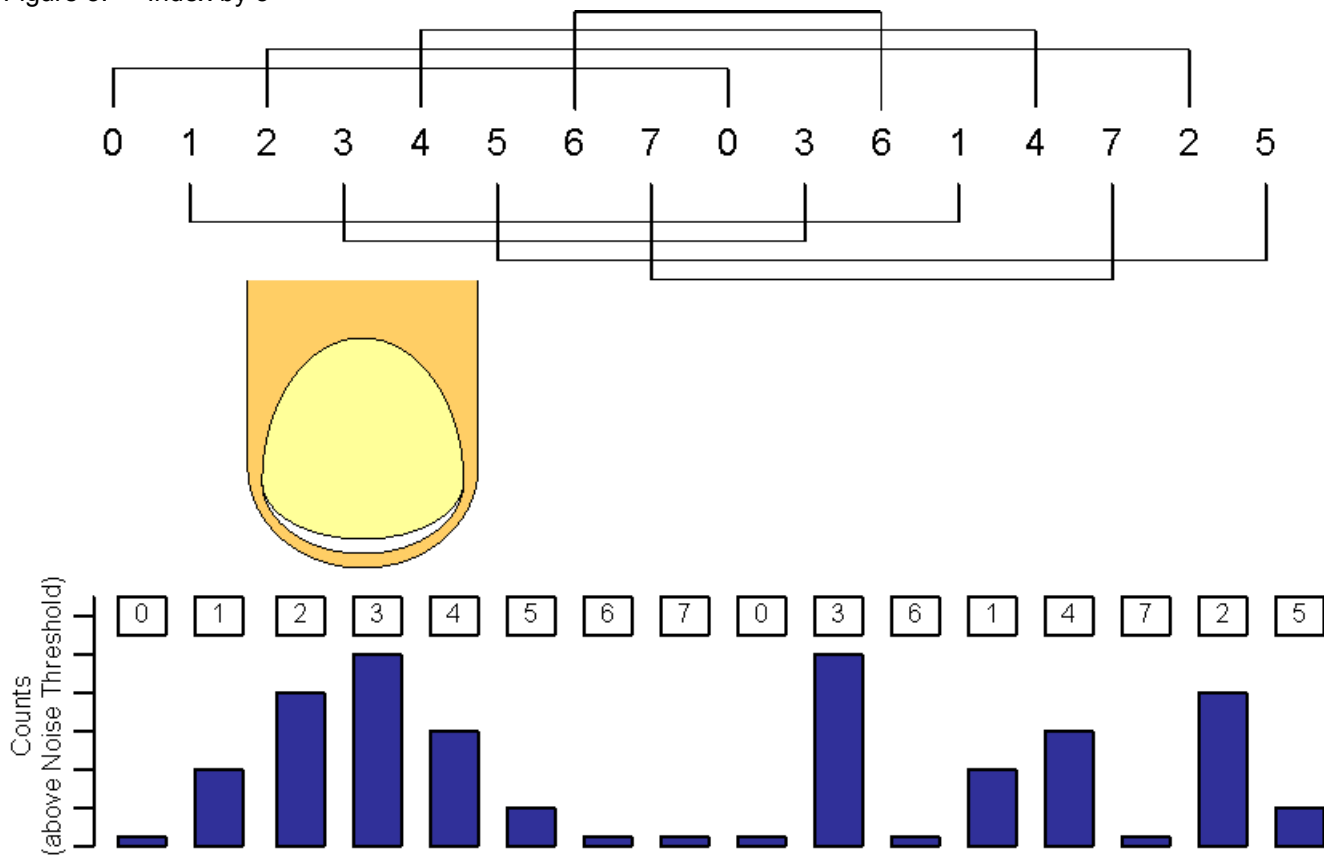
The CSD2X UM has two API functions that support radial sliders. The first function, CSD2X_wGetRadiaPos(), returns centroid location and the second, CSD2X_wGetRadialInc(), returns finger shift in resolution units. When the finger moves in a clockwise direction it is a positive offset.

### *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 the adjacent sensor actuation in one half does not result in adjacent sensor actuation in the other half. Make certain 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. The following table illustrates the diplexing sequences for different slider segment counts.

Table 1. Diplexing Sequence for Different Slider Segment Counts

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

| Total Slider Segment Count | Segment Sequence |
|---|---|
| 50 | 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20,23 |
| 52 | 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23 |
| 54 | 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23,26 |
| 56 | 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,0,3,6,9,12,15,18,21,24,27,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23,26 |

### *Interpolation and Scaling*

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

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

**Equation 1**

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

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

Slider sensor count and resolution are set in the 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 MSB | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Bit** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| 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

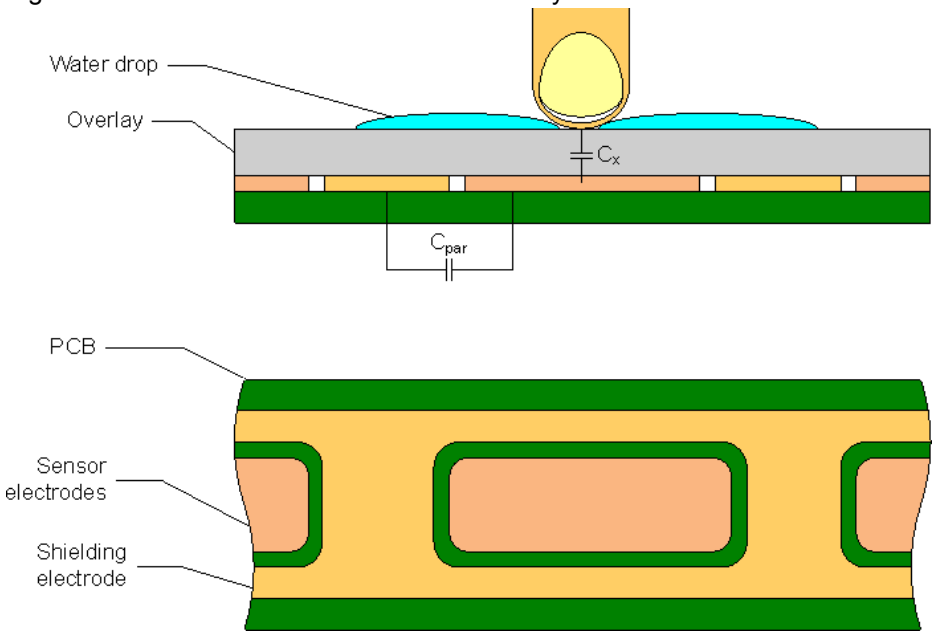
The centroid is held in a 24-bit unsigned integer and its resolution is a function of the number of sensors and the multiplier.

## CSD2X 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 give a false trigger 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 a 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 6. 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 with a fill ratio of about 30 to 40%. No additional ground plane is required.

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 slightly negative. You can achieve this by selecting the appropriate modulator reference.

### Comparator Reference Source

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

The user module uses different reference forming principles for the IDAC and Rb configurations.

For Rb configuration, the user module supports the following selections for a reference source:

- Bandgap reference
- Analog modulator, driven by a special PWM signal
- External resistive voltage divider

This table summarizes the reference selection options for Rb configuration:

| Type | External Components | UM Selection | When to Use |
|---|---|---|---|
| Bandgap reference | none | VBG | Recommended for most applications. Try starting testing from this option. |
| Analog modulator | none | ASExx | Readings are proportional to the power supply voltage. Use only when power supply is well regulated. |
| External resistive voltage divider | 2 | AnalogColumn Input Select | Readings are less dependent on power supply. Recommended R1 = 10k; R2 = 3.6k |

For IDAC configuration, the user module supports the following reference sources:

- Bandgap reference
- VDD reference
- External resistive voltage divider

This table summarizes the reference selection options for IDAC configuration:

| Type | External Components | UM Selection | When to Use |
|---|---|---|---|
| Bandgap reference | none | VBG | Recommended for most applications. Try starting testing from this option. |
| Power Supply reference | none | VDD | Readings are proportional to the power supply voltage. Use only when power supply is well regulated. |

You may 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 two selection options as the clock source for the precharge switches:

- 16-bit pseudo-random sequence generator (PRS)
- IMO prescaler directly

The required configuration can be selected by corresponding user module parameters.

The PRS source gives 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 the EMC/EMI tests or must give reliable operation in harsh environments, the PRS clock source is recommended.

The following table compares the two clock sources:

| Clock Source | Operation Frequency | EMC Noise Immunity |
|---|---|---|
| PRS | Spread-spectrum, average is $F_{IMO}$/4/Prescaler, peak is $F_{IMO}$/2/Prescaler | High. Sensitive points are multiples of the PRS sequence repeat period and PRS fundamental frequency $F_{IMO}$/Prescaler. |
| IMO prescaler direct | Fixed frequency $F_{IMO}$/Prescaler | Moderate. Sensitive at more points. |

## Placement

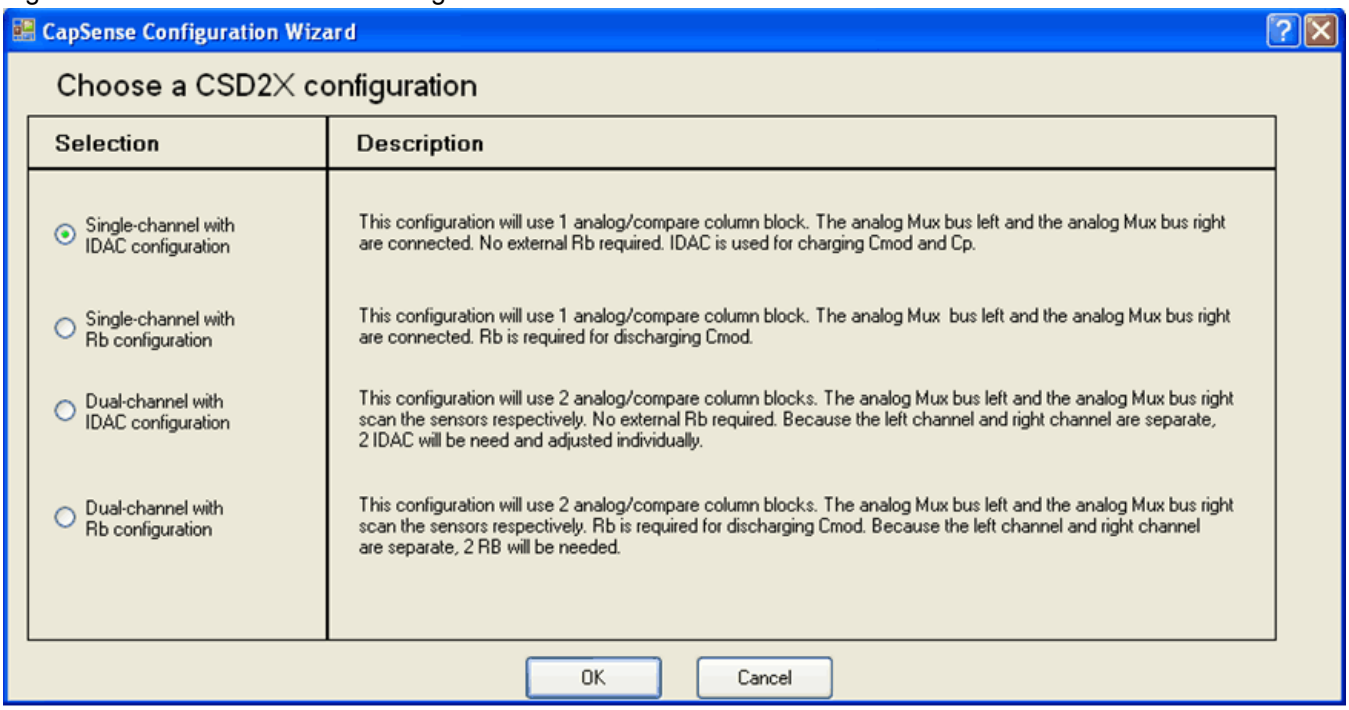
The blocks for the user module are automatically placed when the user module is instantiated, alternate placements are available for single channel configurations only. The CSD2X User Module consumes the CapSense block and one Comparator block. User modules (such as AMuxN) that occupy the same analog 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 the pins that belong to different analog mux buses

This CSD2X Multi User Module (MUM) has four configurations:

- CSD2X single-channel with IDAC configuration
- CSD2X Single-channel with Rb configuration
- CSD2X Dual-channel with IDAC configuration

- CSD2X Dual-channel with Rb configuration

Figure 7. Multi 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 8. Block Resources: Single-Channel with IDAC Configuration

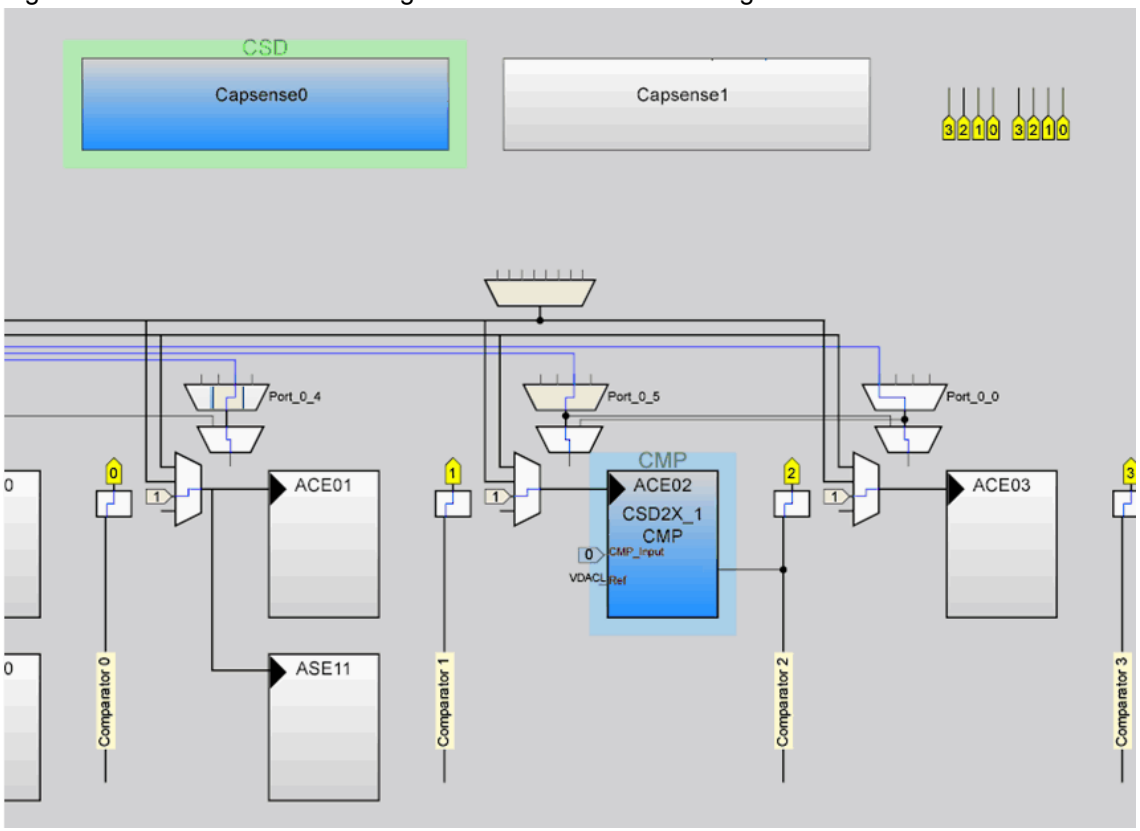



Figure 9. Block Resources: Single-Channel with Rb Configuration

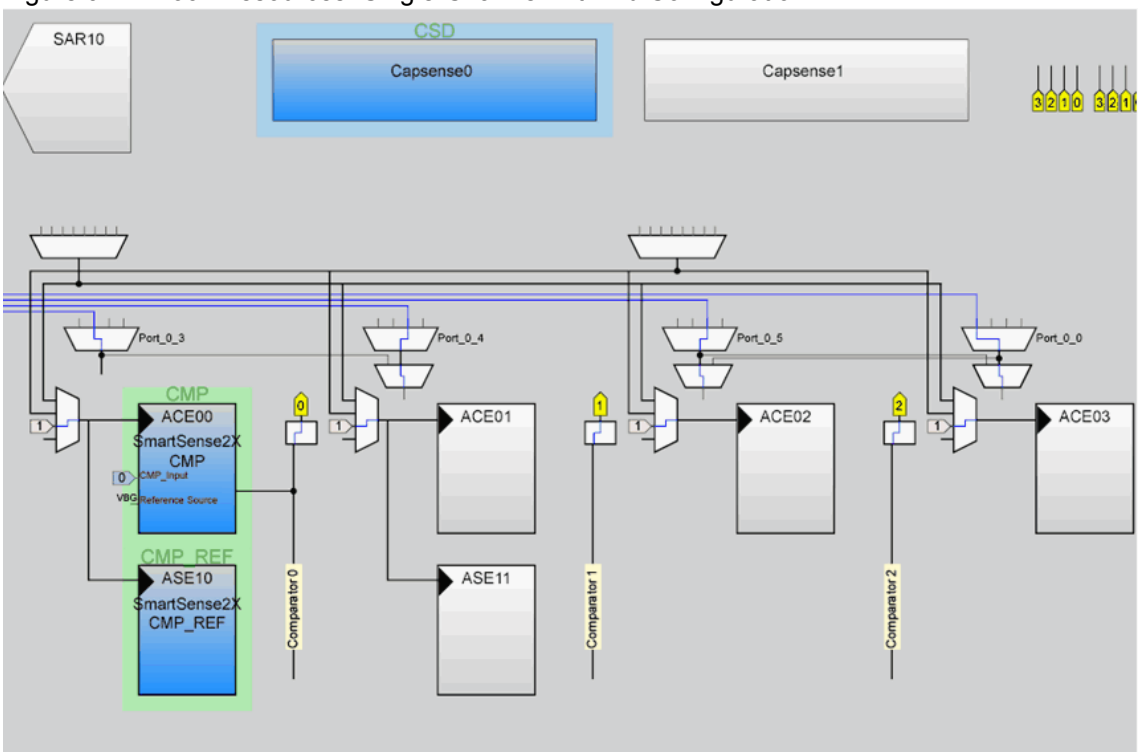


Figure 10. Block Resources: Dual-Channel with IDAC Configuration

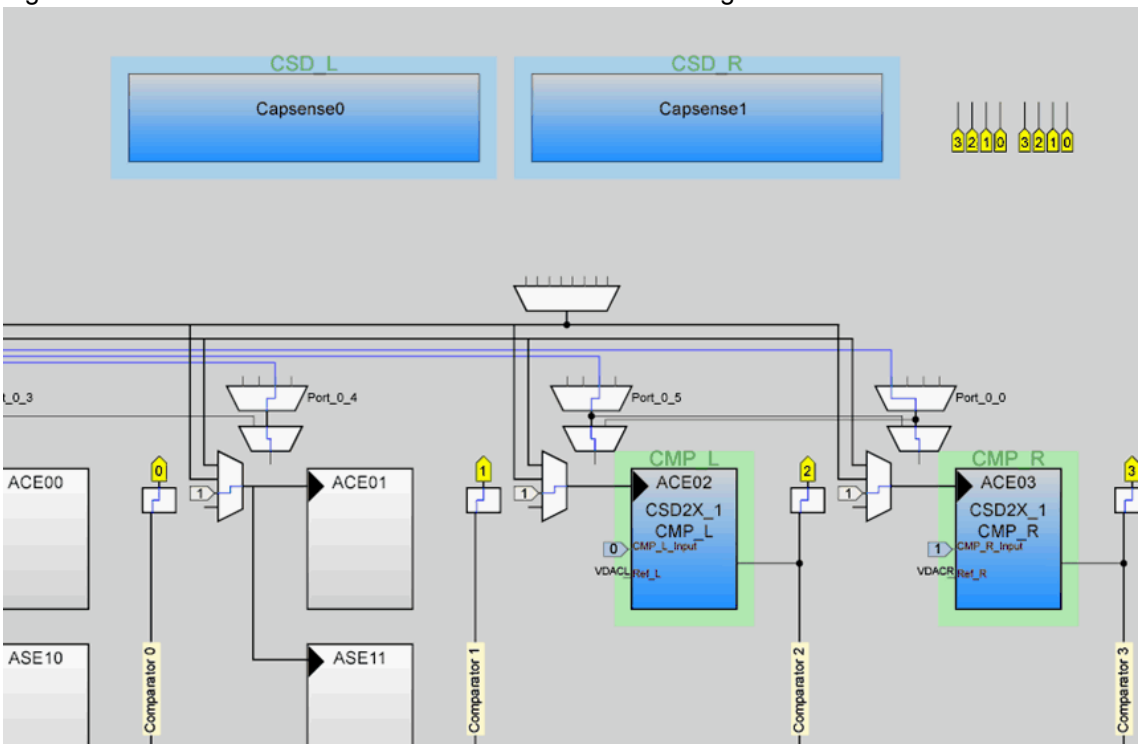
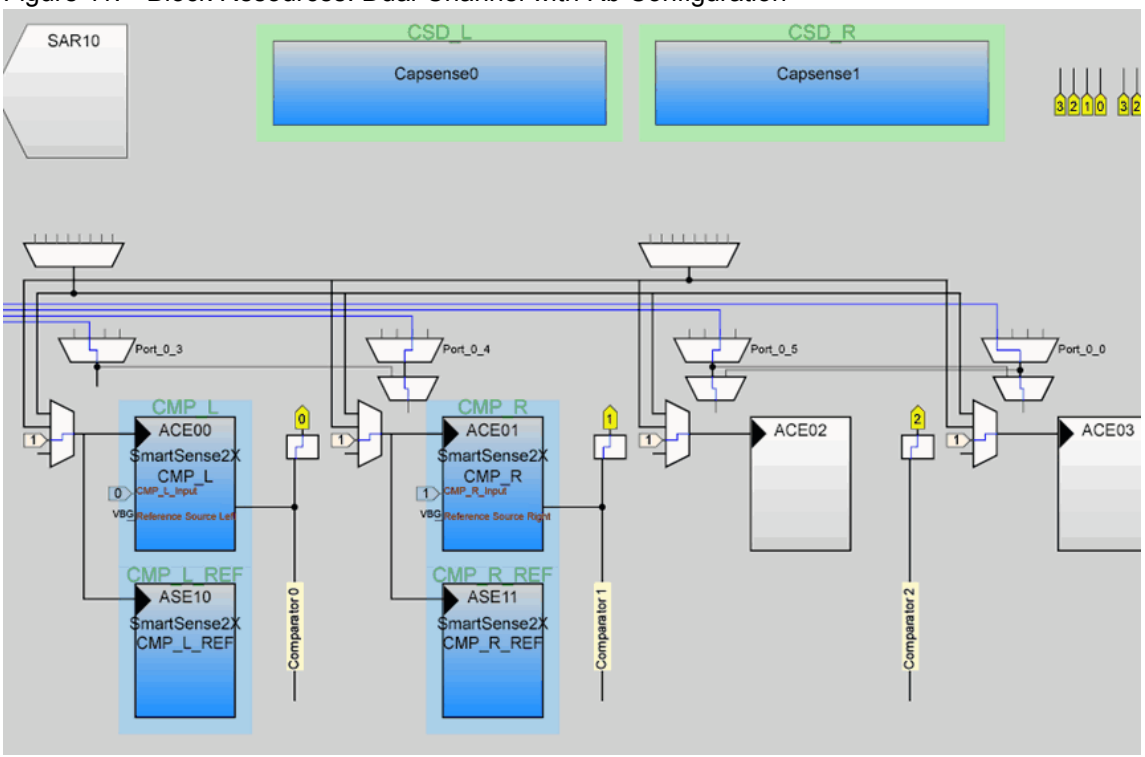


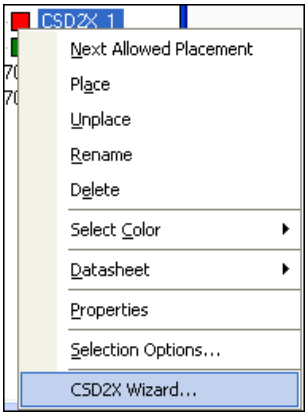

Figure 11. Block Resources: Dual-Channel with Rb Configuration



## Wizard

The CSD2X Wizard is used to set the pinout for your CapSense buttons, sliders, and proximity sensors. Choose the configuration and assign the buttons and segments using the drag-and-drop interface.

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



**2.** Based on the selection in the MUM wizard, the options in this wizard vary. The Wizard has numeric entry boxes for the number of sensors and the number of slider sensors

Figure 12. Wizard for Single-channel with IDAC Configuration

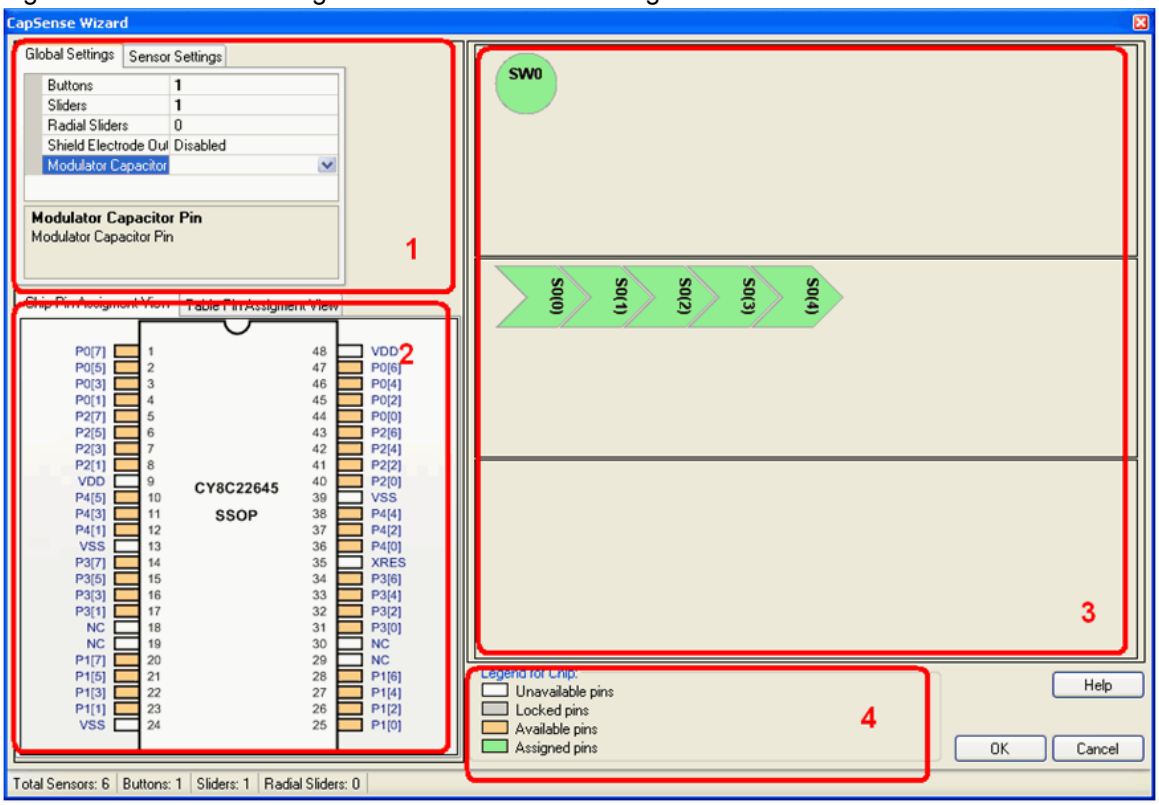



Figure 13. Wizard for Dual-Channel with IDAC Configuration

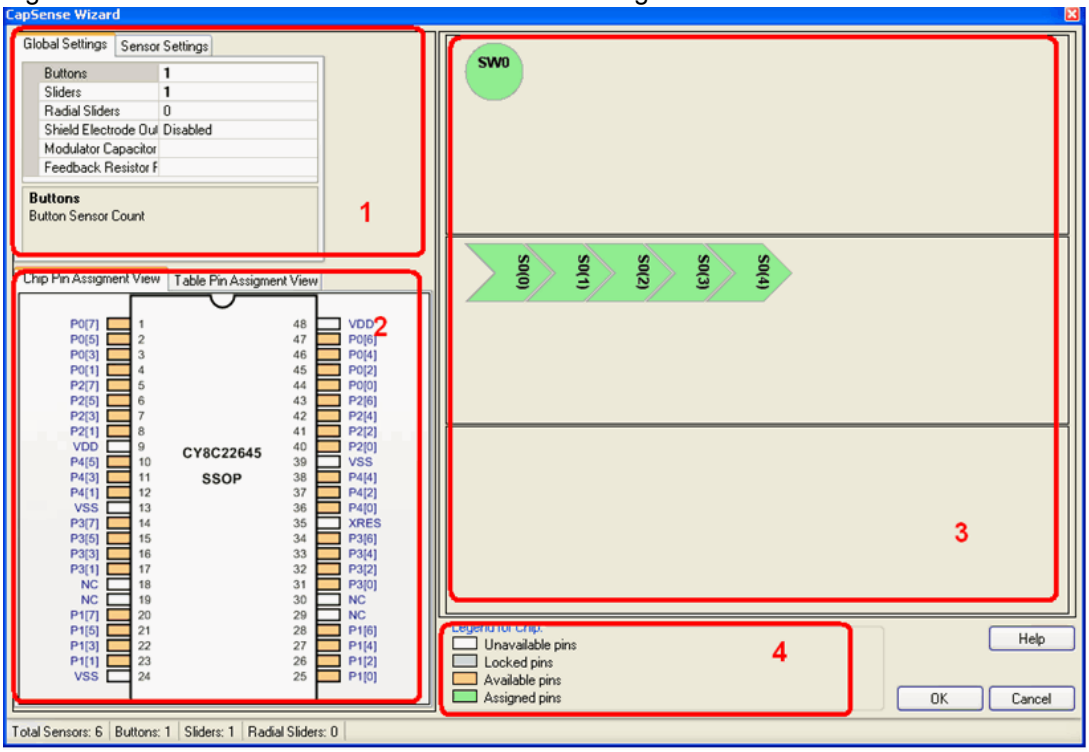


Figure 14. Wizard for Single-Channel with Rb Configuration

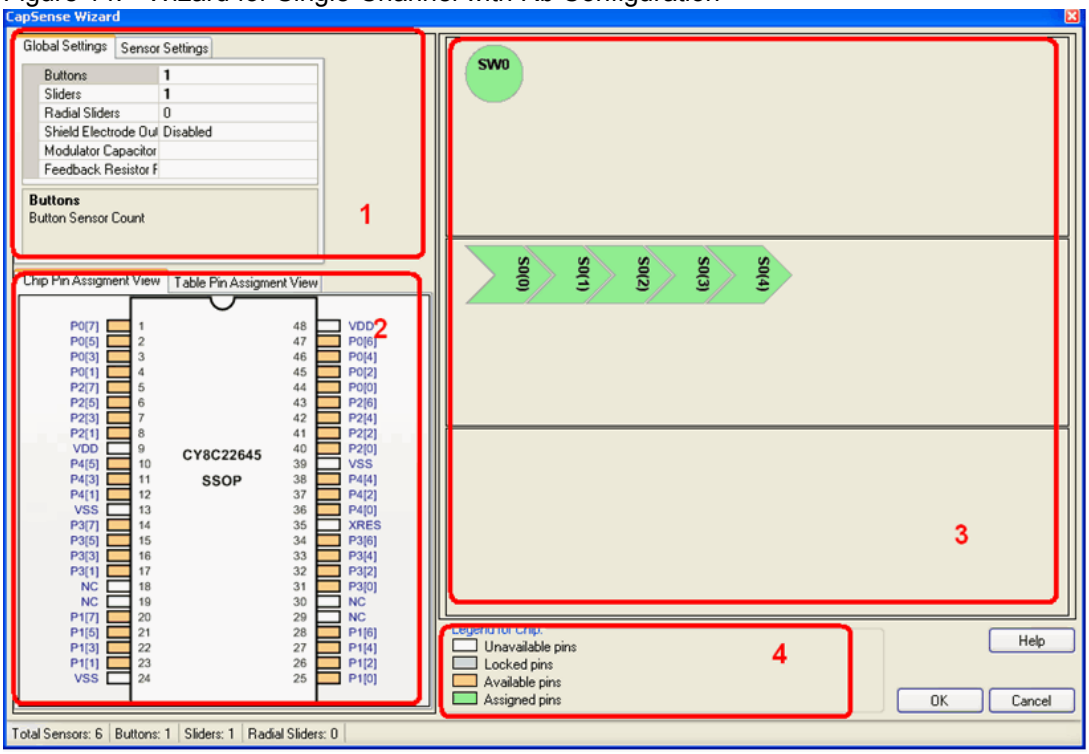
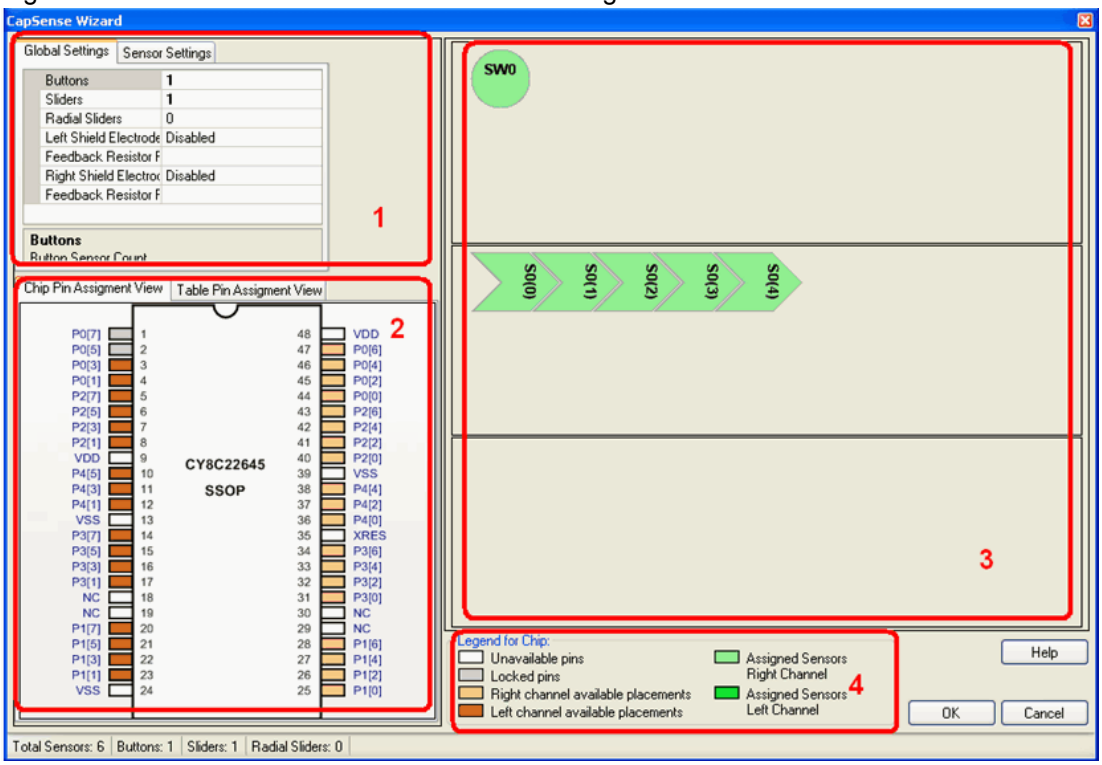



Figure 15. Wizard for Dual-Channel with Rb Configuration



### Wizard Pin Legend

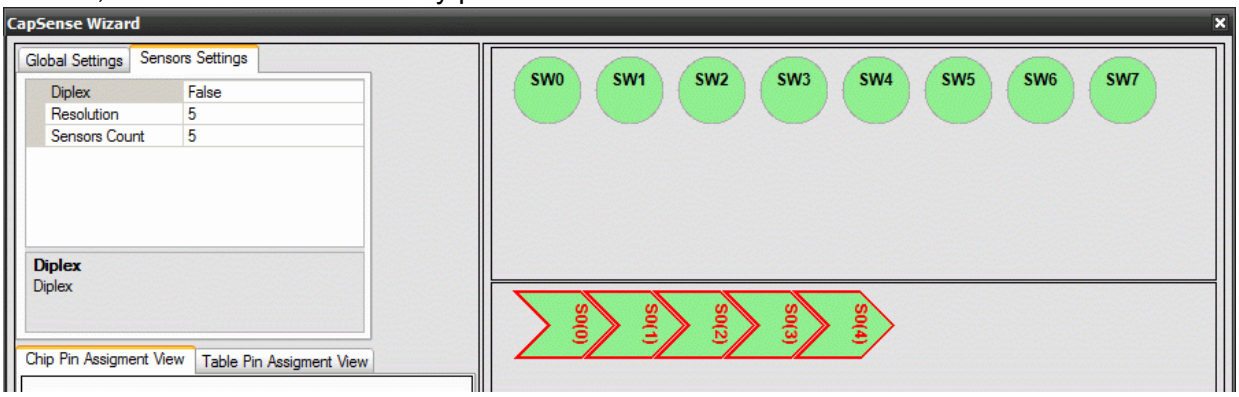
White – The pin cannot be used as a CapSense input.

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.

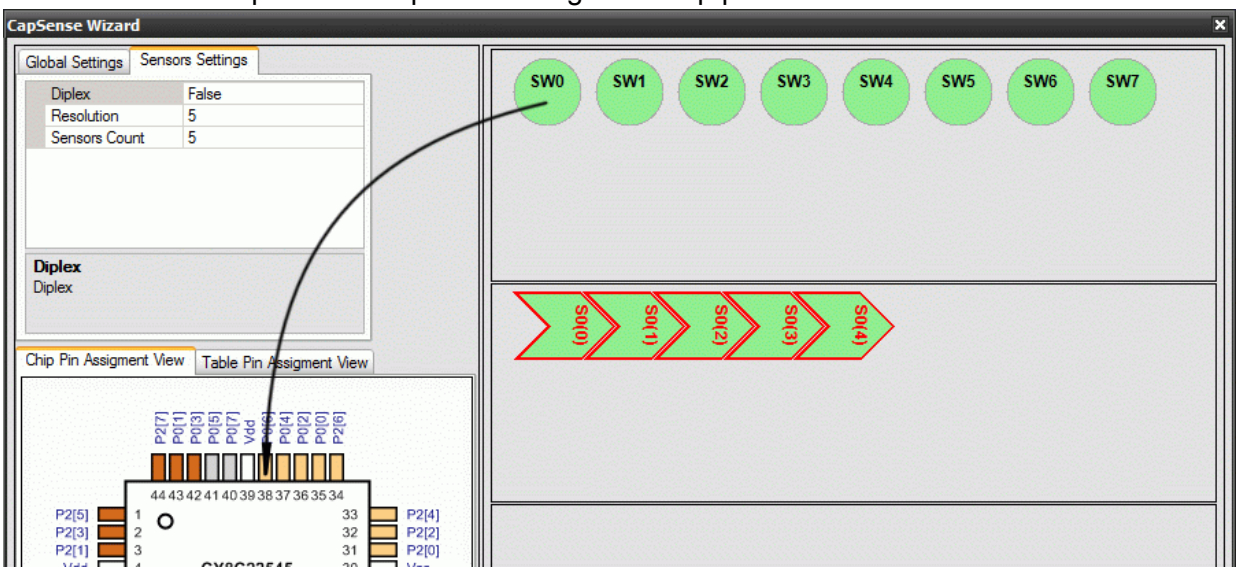
Green – The pin has been assigned as a CapSense input.

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.
4. Switch to the Sensor Settings tab to set parameters for your sliders. Setting for sliders and rotary sliders (including the number of sensors in the slider or rotary slider) are set collectively. X-Y touchpads require two sliders but one is selected. The practical minimum number of sensors in a slider sensor is five, the maximum is limited by pin count.

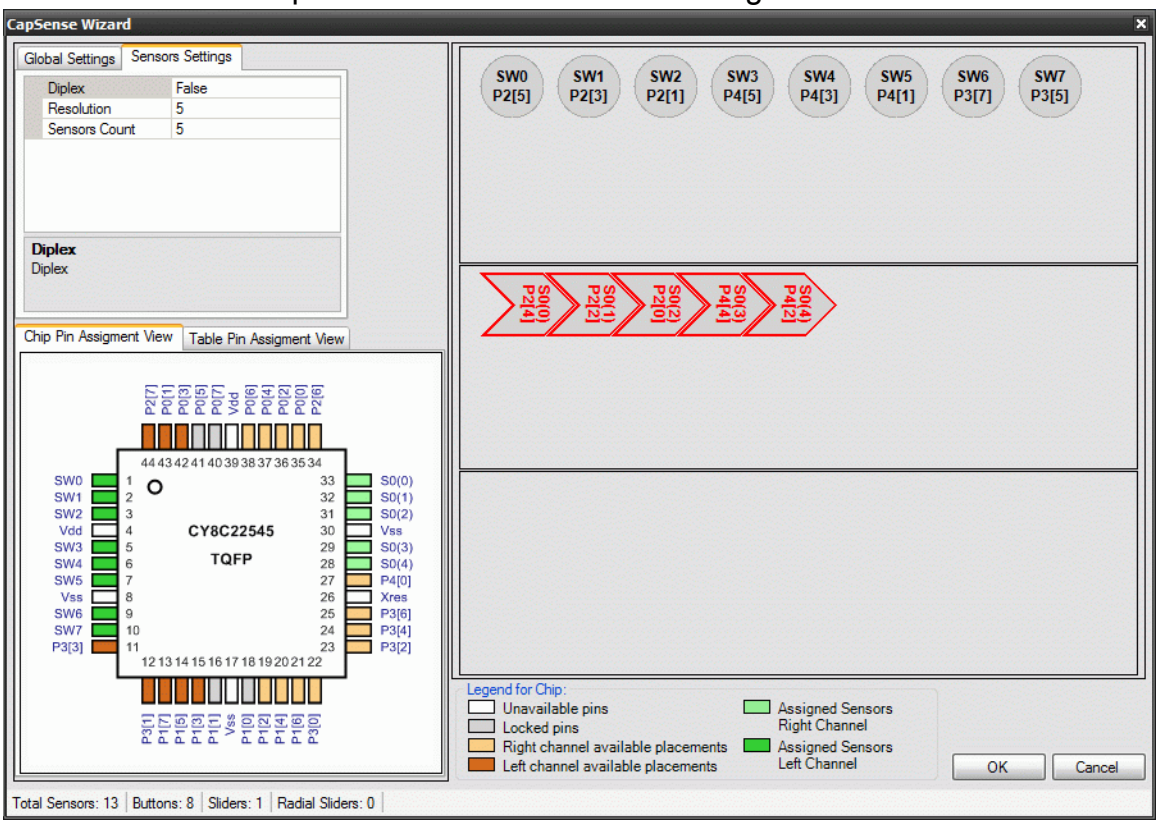



5. The Resolution of a slider is the number of positions your slider has. Touches that give readings on more than one segment of the slider are interpolated to this resolution. If you set the resolution to 100, the position of the finger on the slider is reported as being somewhere between 0 and 99. The minimum value is five. The maximum value is (number of pins used for sensors – 1) x $2^8$ – 1 or (2 x number of 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, select 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 the

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




8. Repeat step 7 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 needed.

If you want to change the 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.

## Modulator Capacitor Pin

CSD2X requires an external modulation capacitor, $C_{mod}$, connected from Vss to one of the two dedicated PSoC pins – P0[5] or P0[7]. The CSD2X dual-channel configuration requires two external modulation capacitors that must be connected to the P0[5] and P0[7] pins (note that the Modulator Capacitor Pin property is not available and pins P0[5] and P0[7] are locked).

The selected pins must not be used for any other purpose. The recommended value for the external modulation capacitor is 2.2 nF. Make certain to use a ceramic capacitor. The temperature capacitance coefficient is not important

## Feedback Resistor Pin

This parameter is available for Rb configurations only. This parameter sets the pin to connect the external feedback resistor ($R_b$). Dual-channel with $R_b$ configuration has **Feedback Resistor Pin L** and **Feedback Resistor Pin R** properties for left and right channels respectively. Choose from the available pins: P1[4], P3[4], GOO[4] for left channel and P1[5], P3[5], GOO[5] for right channel. If the GOO[4] or GOO[5] option is chosen, then the respective Rb signals must be routed manually to a pin from the GOO[4] and/or GOO[5] global output bus nets. In this case, the Rb pins must be set to 'Open Drain Low' drive mode. Some pins are not available on some device packages.

## Shield Electrode Output

This parameter enables or disables the algorithm that supports the optional shield electrode. The dual-channel configuration has Left Shield Electrode Output and Right Shield Electrode Output properties for its left and right channels. The shield electrode, if enabled, is routed to P1[6] or P3[6] for the left channel or P1[7] or P3[7] for the right channel in the CSD2X Wizard. On devices with fewer pins, P3[6] and P3[7] may be unavailable. For a single channel, the connection depends on whether the right or left CapSense channel is used. The drive mode for the selected pin should be set to Strong.

Cypress recommends using a 560-ohm series resistor on all CapSense sensor traces for RF interference suppression. This resistor must be placed as close to the PSoC device as possible.

## Wizard Slider Settings

### Diplex

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

### 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.

### Sensors Count

This value sets the number of sensors in each slider or rotary slider.

## Tables Produced by the Wizard

After completing the Wizard, click **Generate Application**. Based on your entries for sensor count, pin assignment, and diplexing, 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 each for the left and right channels. The table includes all independent sensors, and 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 the 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*

The 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 the wizard and reflects 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
```

### *IDAC Table*

The IDAC Table is generated for CSD2X with IDAC configurations when AutoCalibration is enabled. This table contains calibrated current values for each sensor, which can be changed in runtime. This can be useful in complex projects.

For Double Channel configuration:

```
 CSD2X_1_baDACCodeBaselineL:
_CSD2X_1_baDACCodeBaselineL:
BLK CSD2X_1_TotalLeftSensorCount
CSD2X_1_baDACCodeBaselineR:
_CSD2X_1_baDACCodeBaselineR:
BLK CSD2X_1_TotalRightSensorCount
```

For Single Channel configuration:

```
 CSD2X_1_baDACCodeBaselineL:
_CSD2X_1_baDACCodeBaselineL:
BLK CSD2X_1_TotalSensorCount
CSD2X_1_baDACCodeBaselineR:
_CSD2X_1_baDACCodeBaselineR:
BLK CSD2X_1_TotalSensorCount
```

## Software Interrupts Compatibility

**WARNING:** The CSD2X User Module overwrites contents of INT_CLR2 register with logical '1' values during its operation. This means that Software Interrupts feature cannot be used with the CSD2X User Module. Do not alter the value of the ENSWINT bitfield in INT_MSK3 register - this breaks the project operation.

## Parameters and Resources

**Autocalibration**

Enable or disable autocalibration API functions.

Autocalibration is included only in IDAC configurations. 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. When this parameter is ‘Enabled’, the CSD2X_Calibrate API

function is automatically called at the end of CSD2X_Start API during user module startup. The default setting is Enabled.

**Note** Compensation IDAC is disabled during autocalibration when the Compensation IDAC Value parameter equals to null.

**Noise Threshold**

Difference 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.

**Shield Electrode Out**

This parameter enables or disables the algorithm that supports the optional shield electrode. The shield electrode, if enabled, is routed to P1[6] or P3[6] for left channel or P1[7] or P3[7] for right channel in the CSD2X Wizard. On devices with fewer pins, P3[6] and P3[7] may be unavailable. For a single channel, the connection depends on whether the right or left CapSense channel is used. The default setting is Disabled.

The drive mode for selected pin should be set to **Strong**.

**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.

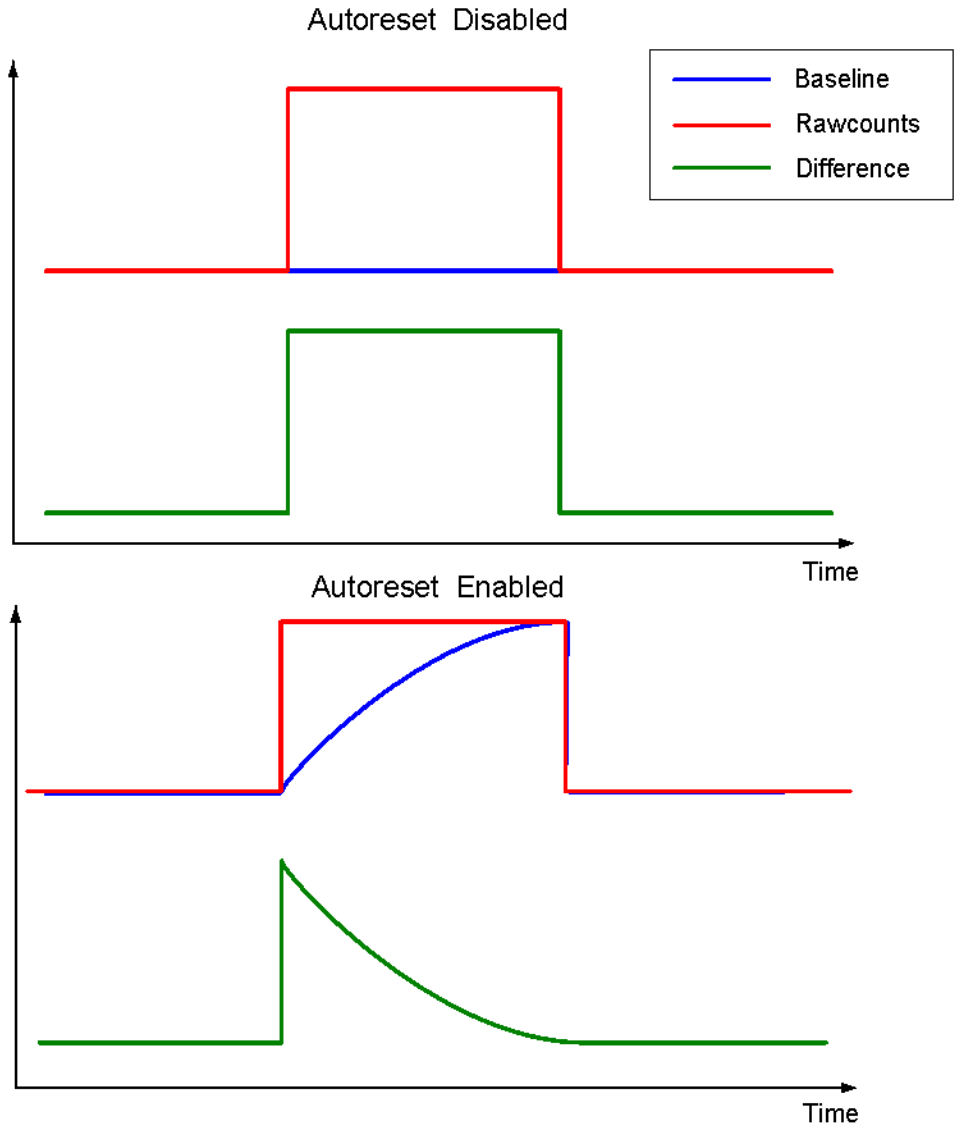
**Sensors Autoreset**

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 Enabled.

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

Figure 16. 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 give reliable touch detection under various conditions. The default value is 10.

**Debounce**

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

Possible values are 1 to 255. A setting of 1 gives no debouncing; the default value is 3.

**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. For individual sensors (not contained in a slider group), these thresholds are variable and given in the baBtnFThreshold[] array. The CSD2X_SetDefaultFingerThresholds() function may be used to set the thresholds to the default value set in the User Module Parameters. To adjust the sensitivity for individual sensors, change the baBtnFThreshold[] value for each sensor. (The size of this byte array is equal to the count of implemented individual sensors.)

Possible values range from 1 to 255; the default value is 20.

**Scanning Speed**

This parameter affects the sensor scanning speed. The available selections are: **Ultra Fast, Fast, Normal, Slow**. The default setting is Normal. Slower scanning speeds give the following advantages:

- Improved SNR
- Better immunity to power supply and temperature changes

The scanning speed affects the Scan Speed Divider in the following way:

| Scanning Speed | Divider |
|---|---|
| Ultra fast | 1 |
| Fast | 2 |
| Normal | 4 |
| Slow | 8 |

**Scanning Resolution**

This parameter determines the scanning resolution in bits. The sensors can be scanned with resolutions ranging from 9 to 16 bits. The maximum raw count for scanning resolution for N bits is $2^N$-1.

Increasing the resolution improves sensitivity and the touch detection's SNR. Use a high resolution for proximity detection. A 16-bit resolution, slow scanning mode, and a 20-cm wire allows you to detect a hand at 20 cm or more. The default value is 12.

Table 2. **Single-Channel Configuration:** Scanning Time in µs vs. Scanning Speed and Resolution for 24 MHz CPU Frequency Operation with Prescaler = SYSCLK/1

| Resolution, Bits | Scanning Speed | | | |
|---|---|---|---|---|
| | Ultra Fast | Fast | Normal | Slow |
| 9 | 51.49 | 72.39 | 114.8 | 199.8 |
| 10 | 72.43 | 114.9 | 200.1 | 370 |
| 11 | 115.4 | 200.3 | 370 | 710 |
| 12 | 200.3 | 370 | 710.1 | 1391 |
| 13 | 370.1 | 710.3 | 1392 | 2752 |
| 14 | 710.8 | 1390 | 2752 | 5474 |
| 15 | 1390 | 2752 | 5472 | 10910 |
| 16 | 2752 | 5472 | 10910 | 21800 |

Table 3. **Dual-Channel Configuration:** Scanning Time in µs vs. Scanning Speed and Resolution for 24 MHz CPU Frequency Operation with Prescaler = SYSCLK/1

| Resolution, Bits | Scanning Speed | | | |
|---|---|---|---|---|
| | Ultra Fast | Fast | Normal | Slow |
| 9 | 83.4 | 104 | 146.5 | 231.7 |
| 10 | 104 | 146.5 | 231.7 | 402 |
| 11 | 146.5 | 231.6 | 402 | 741 |
| 12 | 231.6 | 402.2 | 741 | 1422 |
| 13 | 402 | 741.2 | 1422 | 2784 |
| 14 | 741 | 1422 | 2784 | 5504 |
| 15 | 1422 | 2784 | 5508 | 10950 |
| 16 | 2784 | 5504 | 10950 | 21840 |

### Note

1. The scanning time was measured as the time interval between two sensor scans. This time includes the sensor setup time, modulator stabilization delay, sample conversion interval and data preprocessing time.
2. Scanning time for a given resolution and scan speed setting increases in proportion to an increase in the prescaler value. For example, changing the prescaler from SYSCLK/1 to SYSCLK/2 increases the scanning times in this table by a factor of two.

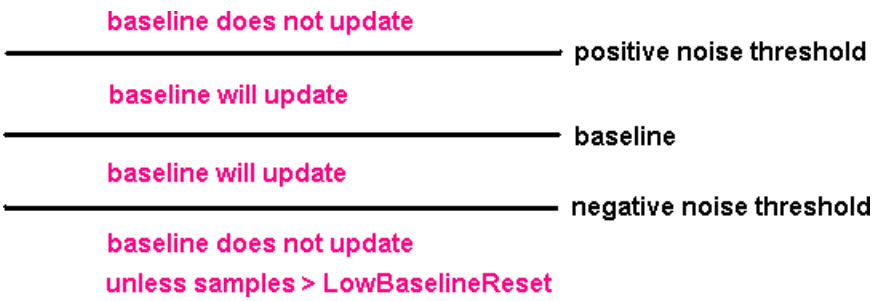
**LowBaselineReset**

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 the finger-on-at-startup condition. The possible value range is 0...255. The default value is 50.

**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. Possible value range is 0...255; the default value is 20.




**Modulator Capacitor Pin**

The modulator capacitor pin is used only with single channel configurations. This parameter sets the pin to connect the external modulator capacitor ($C_{mod}$). Available pins are P0[5] and P0[7]. For Double Channel configuration both pins are used. The default setting is Empty.

**Reference Source**

This parameter sets the source of the comparator reference. Possible choices are bandgap (Vbg), analog modulator (ASExx), or an external voltage (from AnalogColumn_InputSelect_x). For IDAC configurations, the possible choices are Vbg or VDD. For additional information, see the Comparator Reference Source section. The default setting is VBG.

**Reference Value**

For both IDAC and Rb configurations, the value 0 corresponds to a reference voltage close to GND. The value 31 corresponds to a reference voltage close to VDD. This gives the reference voltage a 5-bit resolution.

In Rb configurations, the Reference Value parameter does not have any effect if the reference comes from an external pin or from Vbg. In this case, the reference value setting can be ignored. In Rb configurations, the actual resolution of the reference voltage is 4 bits. However, the value used is still on a 5-bit scale (that is, 0 to 31).

Using a higher reference value gives a higher reference voltage. A higher reference voltage causes the raw counts of the sensors to increase. Similarly, a lower reference value/voltage causes the raw counts of the sensors to decrease. The default value is 23.

**Precharge Source**

This parameter selects the clock source for precharge switches. The allowed options are PRS and Timer. The recommended setting for precharge source is PRS because it not only provides better EMI immunity and lower emission, but also ensures linear response in change of raw counts with the change in capacitance The default setting is PRS.

**Prescaler**

This parameter sets the prescaler ratio and determines the precharge switch output frequency. This parameter also affects the PRS output frequency and the sensor scanning time for a given resolution and scan speed setting.

Possible values are:

- SYSCLK/1
- SYSCLK/2
- SYSCLK/4
- SYSCLK/8
- SYSCLK/16
- SYSCLK/32
- SYSCLK/128
- SYSCLK/256

The default settings are:

- SYSCLK/2 - single conf
- SYSCLK/1 - dual conf

**Feedback Resistor Pin**

This parameter is available for Rb configurations only. This parameter sets the pin to connect the external feedback resistor ($R_b$). Dual-channel with Rb configuration has **Feedback Resistor Pin L** and **Feedback Resistor Pin R** properties for left and right channels accordingly. Choose from the available pins: P1[4], P3[4], GOO[4] for left channel and P1[5], P3[5], GOO[5] for right channel. Some pins are not available on some device packages. Tip: if some of these pins are used for other purposes (for example, allocated for sensor connection), they are not available for selection in UM 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. The default setting is Empty.

**IDAC Range**

This parameter is only available on iDAC configurations. Sets the iDAC current multiplier. The result of the setting is different for dual channel configurations. The results in dual channel configurations are shown.

Connect the sensors that have large capacitance to the left channel in two channel configuration. The default values are: x32 (single conf) and x4 (dual conf).

**IDAC Value**

The 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 at run time using the CSD2X_SetLeftDACValue or CSD2X_SetRightDACValue API function. Possible values are 1 to 255; the default value is 200.

**Compensation IDAC Value**

Compensation iDAC is intended to compensate initial capacitance of sensors. While tuning this parameter can improve sensitivity and SNR, setting too high value can break CSD2X operation. During tuning process, start from 0 and increase this value to achieve maximal SNR and sensitivity. This parameter is available only in Single Channel IDAC Configuration.

Possible values are 0 to 255; the default value is 0.

**Note**

1. Compensation IDAC is disabled during Auto Calibration when Compensation IDAC Value parameter equals to null.
2. The nonzero Compensation IDAC Value can cause to additional noise when project has long interrupts (EzI2Cs interrupts with low CPU Clock for example).

**BackgroundScanning**

This parameter decides whether 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/disables the FMEA short tests.

**FMEA_Cmod_Rb_Test**

This feature enables the test for Cmod and Rb calculation.

## Application Programming Interface

The Application Programming Interface (API) functions are given 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 given 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. Do not alter these arrays manually. You can inspect these values for debugging purposes, however. For example, you can use a charting tool to display the contents of the arrays. There several global arrays:

- CSD2X_waSnsBaseline[]
- CSD2X_waSnsResult[]
- CSD2X_waSnsDiff[]
- CSD2X_baSnsOnMask[]

- CSD2X_baDACCodeBaseline[]
- 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. The data is updated by CSD_UpdateSensorBaseline().

**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_baDACCodeBaseline[] –** This table contains calibrated current values for each sensor, which can be changed during runtime. It is useful in complex projects.

For double channel configuration:

- CSD2X_baDACCodeBaselineL:
- _CSD2X_baDACCodeBaselineL:
- BLK CSD2X_TotalLeftSensorCount
- CSD2X_baDACCodeBaselineR:
- _CSD2X_baDACCodeBaselineR:
- BLK CSD2X_TotalRightSensorCount

For single channel configuration:

- CSD2X_baDACCodeBaselineL:
- _CSD2X_baDACCodeBaselineL:
- BLK CSD2X_TotalSensorCount
- CSD2X_baDACCodeBaselineR:
- _CSD2X_baDACCodeBaselineR:
- BLK CSD2X_TotalSensorCount

**CSD2X_bScanComplete[] –** This variable is valid only when the background scanning feature is enabled. This variable must be set when sensor scanning is complete.

The CSD2X_bScanComplete variable is updated in the CSD2X interrupt. CSD2X_bScanComplete contains various flags that define the scanning state. CSD2X_SCAN_COMPLETE flag says that all background scanning scans are complete. The CSD2X_SCAN_1COMPLETE flag says that one sensor is scanned after the ScanAllSensors API call. The CSD2X_SCAN_ALLSENSORS flag says that not all sensors are scanned yet after the ScanAllSensors API call.

| 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 o, 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 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 o, 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 those elements as that are necessary to contain all the placed sensors.The CSD2X_baSensorShortGnd[] data is updated by the CSD2X_bFMEA_CheckVddShort() function.

**CSD2X_wCmod_Val[] –** This is a WORD variable, which is valid only when the FMEA feature is enabled and when 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 when the Rb method is selected. This variable contains the product of Cmod and Rb calculated during the CSD2X_FMEA_Cmod_Rb_Check function.

## CSD2X_Start

**Description:**

Initializes registers and starts the user module. This function should be called before calling any other user module functions. When autocalibration is enabled, CSD2X_Calibrate API is automatically called within this function, after all other user module parameters have been applied.

**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_SetScanMode

**Description:**

This function overrides Scanning Speed and Scanning Resolution set in User Module Parameters for all subsequent scans. Resolution is an integer value between 9 and 16.

**C Prototype:**

In Singe Channel Configuration:

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

In Double Channel Configuration:

```
void  CSD2X_SetLeftScanMode(BYTE bSpeed, BYTE bResolution);
void  CSD2X_SetRightScanMode(BYTE bSpeed, BYTE bResolution);
```

**Assembly:**

In Single Channel Configuration:

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

In Double Channel Configuration:

```
mov  A,  bSpeedL
mov  X,  bResolutionL
lcall  CSD2X_SetLeftScanMode
mov  A,  bSpeedR
mov  X,  bResolutionR
lcall  CSD2X_SetRightScanMode
```

**Parameters:**

bSpeed: Scanning Speed

The following constants are given for the bSpeed parameter:

| Constant | Value |
|---|---|
| CSD2X_ULTRAFAST_SPEED | 0x00 |
| 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. Thisparameter value must not be lower than 9 or greater than 16.

Following possible constants are given for bResolution parameter:

| Constant | Value |
|---|---|
| CSD2X_9_BIT_RESOLUTION | 9 |
| CSD2X_10_BIT_RESOLUTION | 10 |
| CSD2X_11_BIT_RESOLUTION | 11 |

| Constant | Value |
|---|---|
| CSD2X_13_BIT_RESOLUTION | 13 |
| CSD2X_14_BIT_RESOLUTION | 14 |
| CSD2X_15_BIT_RESOLUTION | 15 |
| CSD2X_16_BIT_RESOLUTION | 16 |

**Return Value:**

None

**Side Effects**

**

## CSD2X_SetPrescaler

**Description:**

This function overrides the Prescaler value set in User Module Parameters for all subsequent scans.

**C Prototype:**

In Singe Channel Configuration:

```
void  CSD2X_SetPrescaler(BYTE bPrescaler);
```

In Double Channel Configuration:

```
void  CSD2X_SetLeftPrescaler(BYTE bPrescaler);
void  CSD2X_SetRightPrescaler(BYTE bPrescaler);
```

**Assembly:**

In Single Channel Configuration:

```
mov  A,  bPrescaler
lcall  CSD2X_SetPrescaler
```

In Double Channel Configuration:

```
mov  A,  bPrescalerL
lcall  CSD2X_SetLeftPrescaler
mov  A,  bPrescalerR
lcall  CSD2X_SetRightPrescaler
```

**Parameters:**

A => Prescaler value.

Following constants are given for bPrescaler parameter:

| Constant | Value |
|---|---|
| CSD2X_PRESCALER_SYSCLK1 | 0x30 |
| CSD2X_PRESCALER_SYSCLK2 | 0x20 |
| CSD2X_PRESCALER_SYSCLK4 | 0x10 |
| CSD2X_PRESCALER_SYSCLK8 | 0x00 |
| CSD2X_PRESCALER_SYSCLK16 | 0x40 |
| CSD2X_PRESCALER_SYSCLK32 | 0x50 |
| CSD2X_PRESCALER_SYSCLK128 | 0x60 |
| CSD2X_PRESCALER_SYSCLK256 | 0x70 |

**Return Value:**

None

**Side Effects**

**

## CSD2X_ScanSensor

**Description:**

Scans the selected sensors. Each sensor has a unique number within the sensor array. For single-channel configurations, this number is assigned by the CSD2X Wizard in sequence. Sw0 is sensor 0, Sw1 is sensor 1, and so on.

For two-channel configurations, the sensor number is a value from 0 to Maximum Channel Sensor Number. For example, if there are two sensors on the left channel, their values are 0 and 1, respectively. If there are also two sensors on the right channel, their values are also 0 and 1, respectively. If a value of 0xFF is passed into this function as a sensor number, no sensor is scanned for that channel.

**C Prototype:**

In Singe Channel Configuration:

```
void  CSD2X_ScanSensor(BYTE bSensor);
```

In Double Channel Configuration:

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

**Assembly:**

```
mov  A,  bSensor
lcall  CSD2X_ScanSensor
```

In Double Channel Configuration:

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

**Parameters:**

A => Sensor Number

In Double Channel Configuration:

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_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 given 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. Once the accumulated difference counts in the virtual bucket has reached the BaselineUpdateThreshold, the baseline is incremented by one and the bucket is reset to 0.

4. If the difference count is below the noise threshold, the value held in the waSnsDiff[] array is reset to 0. Therefore, this array does not contain elements with values greater than 0 but below the NoiseThreshold.

**C Prototype:**

```
void  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 UM wizard on the left hand of radial slider representation (for example, s2, the radial slider number is 2).

**Return Value:**

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

**Side Effects:**

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

If any slider sensor is active, the function returns values from zero to the Resolution value set in the 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 UM wizard on the left hand of radial slider representation (for example, s2, the radial slider number is 2).

**Return Value:**

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

Finger shift value is the difference between current and previous finger positions. If there was no touch during previous scan (the last but one time 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_SetLeftDACValue

**Description:**

This function overwrites the user module parameter settings left IDAC Value. Use it if some sensors need to be scanned with different iDAC setting. This function can be used in conjunction with CSD2X_ScanSensor(). This function is not available in Rb configurations.

**C Prototype:**

```
void  CSD2X_SetLeftDACValue(BYTE bIdacValue);
```

**Assembly:**

```
mov  A,  bIdacValue
lcall  CSD2X_SetLeftDACValue
```

**Parameters:**

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

**Return Value:**

None

**Side Effects:**

**

## CSD2X_SetRightDACValue

**Description:**

This function overwrites the user module parameter settings right IDAC Value. Use it if some sensors need to be scanned with a different iDAC setting. This function can be used in conjunction with CSD2X_ScanSensor(). This function is not available in Rb configurations.

**C Prototype:**

```
void  CSD2X_SetRightDACValue(BYTE bIdacValue);
```

**Assembly:**

```
mov  A,  bIdacValue
lcall  CSD2X_SetRightDACValue
```

**Parameters:**

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

**Return Value:**

None

**Side Effects:**

**

## CSD2X_SetIdacValue

**Description:**

This function overwrites the user module settings for IDAC and Compensation IDAC Values. Use it if some sensors need to be scanned with a different IDAC setting. This function can be used in conjunction with CSD2X_ScanSensor(). This function is available only in Single Channel IDAC configuration.

**C Prototype:**

```
void  CSD2X_SetIdacValue(BYTE bIDACVal, BYTE bCompIDACVal);
```

**Assembly:**

```
mov  A,  bIDACVal
mov  X,  bCompIDACVal
lcall  CSD2X_SetIdacValue
```

**Parameters:**

bIDACVal - Sets the IDAC value. Accepted values are 1..255.

bCompIDACVal - Sets the Compensation IDAC value. Accepted values are 0..255.

**Return Value:**

None

**Side Effects:**

**

## CSD2X_SetRefValue

**Description:**

This function overwrites the user module parameter settings 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 has no effect when using an Rb configuration and the reference is set to be Vbg or come from an external pin. This is because the Vbg reference voltage or a reference voltage from an external pin cannot be adjusted at run-time.

**C Prototype:**

In Single Channel Configuration:

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

In Double Channel Configuration:

```
 void  CSD2X_SetLeftRefValue(BYTE bLeftRefValue);
void  CSD2X_SetRightRefValue(BYTE bRightRefValue);
```

**Assembly:**

In Single Channel Configuration:

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

In Double Channel Configuration:

```
 mov  A,  bLeftRefValue
lcall  CSD2X_SetLeftRefValue
mov  A,  bRightRefValue
lcall  CSD2X_SetRightRefValue
```

**Parameters:**

bRefValue - Sets the reference value. Accepted values are 0..31.

**Return Value:**

None

**Side Effects:**

**

## CSD2X_SetRefSource

**Description:**

This function overwrites the user module parameter settings reference source. Use it if some sensors need to be scanned with a different reference setting. This function can be used in conjunction with CSD2X_ScanSensor().

**C Prototype:**

In all Single Channel and Double Channel with IDAC Configurations:

```
void  CSD2X_SetRefSource(BYTE bRefSource);
```

In Double Channel with Rb Configuration:

```
void  CSD2X_SetLeftRefSource(BYTE bLeftRefSource);
void  CSD2X_SetRightRefSource(BYTE bRightRefSource);
```

**Assembly:**

In all Single Channel and Double Channel with IDAC Configurations:

```
mov  A,  bRefSource
lcall  CSD2X_SetRefSource
```

In Double Channel with Rb Configuration:

```
mov  A,  bLeftRefSource
lcall  CSD2X_SetLeftRefSource
mov  A,  bRightRefSource
lcall  CSD2X_SetRightRefSource
```

**Parameters:**

bRefSource - Reference source constant. Accepted values are shown in tables below.

| Constant (For Rb configurations) | Value |
|---|---|
| CSD2X_REFERENCE_ACOLUMN_MUX | 0x01 |
| CSD2X_REFERENCE_VBG | 0x03 |
| CSD2X_REFERENCE_ASE | 0x04 |

| Constant (For IDAC configurations) | Value |
|---|---|
| CSD2X_REFERENCE_VDD | 0x00 |
| CSD2X_REFERENCE_2VBG | 0x10 |

**Return Value:**

None

**Side Effects:**

**

## CSD2X_SetIdacRange

**Description:**

This function overwrites the user module parameter settings for DAC ranges. Use it if some sensors need to be scanned with a different range setting. This function can be used in conjunction with CSD2X_ScanSensor(). This function is not available in Rb configurations.

**C Prototype:**

```
void  CSD2X_SetIdacRange(const bRange);
```

**Assembly:**

```
mov  A,  bRange
lcall  CSD2X_SetIdacRange
```

**Parameters:**

bRange - Sets the reference value. Accepted values are one of the following constants.

| Setting | Value | Effect |
|---|---|---|
| CSD2X_IDAC_RANGE_1X | 0x00 | Maximum IDAC current is 19.92 µA |
| CSD2X_IDAC_RANGE_4X | 0x01 | Maximum IDAC current is 91.03 µA |
| CSD2X_IDAC_RANGE_16X | 0x08 | Maximum IDAC current is 318.75 µA |
| CSD2X_IDAC_RANGE_32X | 0x09 | Maximum IDAC current is 637.50 µA |

**Return Value:**

None

**Side Effects:**

**

## CSD2X_Calibrate

**Description:**

This function performs CDS2X Auto Calibration. When AutoCalibration is enabled, this function if called automatically during user module startup in CSD2X_Start() function. 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_wGetPortPin() 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_wGetPortPin

**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[]. 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_wGetPortPin(BYTE bSensor)
```

**Assembly:**

```
mov  A,  bSensor
lcall  CSD2X_wGetPortPin
```

**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 plus the number of sensors included in sliders. The sensor number is used by CSD2X_wGetPortPin() to determine port and bit mask for the selected active sensor.

**Return Value:**

A => Sensor Bitmap

X => Port Number

**Side Effects:**

**

## CSD2X_wGetPortPinLeft

**Description:**

Returns the port number and pin mask for a given sensor that is connected to left channel. 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 double 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_wGetPortPin() function, with the port number and sensor bitmask loaded into X and A, respectively. Drive modes are modified to place the selected port and pin into Analog High Z mode and to enable the correct Analog Mux Bus input. This also enables the comparator function.

**C Prototype:**

```
void  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_wGetPortPin() function. The drive mode is changed to Strong (001). This effectively grounds the sensor. The connection from the port pin to the Analog-MuxBus is turned off. The function parameters are returned by CSD2X_wGetPortPin() function.

**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:**

**

APIs for Background Scanning and FMEA:

## CSD2X_bIsScanComplete

**Description:**

Checks the scan complete flag in the bScanComplete variable and returns TRUE or FASLE to the customer.

**C Prototype:**

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

**Assembly:**

```
lcall  CSD2X_bIsScanComplete
```

**Parameters:**

None

**Return Value:**

Returns non-zero value if active, 0 if not active.

A => 1 - scan is complete, 0 - scan is incomplete

**Side Effects:**

**

## CSD2X_bFMEA_CheckGndShort

**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

**Return Value:**

Returns 1 if any sensor is shorted to ground and returns a zero if none of the sensors is shorted to ground.

**Side Effects:**

**

## CSD2X_bFMEA_CheckVddShort

**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

**Return Value:**

Returns 1 if any sensor is shorted to VDD and zero if none of the sensors is shorted to VDD.

**Side Effects:**

**

## CSD2X_bFMEA_CheckSensorShort

**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

**Return Value:**

Returns 1 if any sensor is shorted to another sensor and zero if no sensor is shorted to another sensor.

**Side Effects:**

**

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.

## CSD2X_bFMEA_Cmod_Check – Single Channel

## CSD2X_bFMEA_Left_Cmod_Check

## CSD2X_bFMEA_Right_Cmod_Check – Dual Channel

**Description:**

This API is available only if the IDAC method is selected and if the FMEA feature is enabled. This checks $C_{mod}$ for short test and also whether it is within the valid range or not. 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 the 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 PSoC Designer.

The CSD2X_bFMEA_Cmod_Check() function updates CSD2X_wCmod_Val (CSD2X_wCmod_L_Val and CSD2X_wCmod_R_Val for Dual Channel Configurations) WORD variable. This variable contains the value of Cmod in nF scaled to 10. For example, CSD2X_wCmod_Val = 220 corresponds to 22 nF (or 0.022 uF). This value is valid only if CSD2X_bFMEA_Cmod_Check() API returns bRetVal.4 or bRetVal.5.

**C Prototype:**

Single channel:

```
BYTE CSD2X_bFMEA_Cmod_Check(void);
```

Dual channel:

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

**Assembly:**

```
lcall  CSD2X_bFMEA_Cmod_Check
```

**Parameters:**

None

**Return Value:**

bRetVal.0:Cmod shorted to GND.

bRetVal.1:Cmod shorted to VDD.

bRetVal.4:Cmod is within +20%.

bRetVal.5:Cmod is out of range or Cmod is unconnected.

Single Channel:

| Constant | Value | Description |
|---|---|---|
| CSD2X_CMOD_SHORTED_TO_GND | 0x00 | Cmod shorted to GND |
| CSD2X_CMOD_SHORTED_TO_VDD | 0x01 | Cmod shorted to VDD |
| CSD2X_CMOD_WITHIN_20 | 0x04 | Cmod within +20% |
| CSD2X_CMOD_CMOD_WITHOUT_20 | 0x05 | Cmod is out of range |

Dual Channel:

| Constant | Value | Description |
|---|---|---|
| CSD2X_LEFT_CMOD_SHORTED_TO_GND | 0x00 | Left Cmod shorted to GND |
| CSD2X_ LEFT_CMOD_SHORTED_TO_VDD | 0x01 | Left Cmod shorted to VDD |
| CSD2X_LEFT_CMOD_WITHIN_20 | 0x04 | Left Cmod within +20% |
| CSD2X_LEFT_CMOD_CMOD_WITHOUT_20 | 0x05 | Left Cmod is out of range |
| CSD2X_RIGHT_CMOD_SHORTED_TO_GND | 0x00 | Right Cmod shorted to GND |
| CSD2X_RIGHT_CMOD_SHORTED_TO_VDD | 0x01 | Right Cmod shorted to VDD |
| CSD2X_RIGHT_CMOD_WITHIN_20 | 0x04 | Right Cmod within +20% |
| CSD2X_RIGHT_CMOD_CMOD_WITHOUT_20 | 0x05 | Right Cmod is out of range |

**Note**

Disable the background scanning parameter before using FMEA_bFMEA_Cmod_Check.

### CSD2X_bFMEA_Cmod_Rb_Check – Single Channel

### CSD2X_bFMEA_Left_Cmod_Rb_Check

### CSD2X_bFMEA_Right_Cmod_Rb_Check – Dual Channel

**Description:**

This API is available only if the Rb method is selected and if the FMEA feature is enabled. This checks $C_{mod}$ and Rb values for short test and also whether the product is within the valid range or not. The valid range is defined by $C_{mod}$ and Rb product, with an error of 40%. The minimum value is $3.3*10^{-6}$ and maximum value is 700*10-6.

CSD2X_bFMEA_Cmod_Rb_Check() updates the WORD CSD2X_wCmod_Rb_Val (CSD2X_wCmod_Rb_L_Val and CSD2X_wCmod_Rb_R_Val for Dual Channel Configuration) variable. This variable contains the product of $C_{mod}$ and Rb in µsec, calculated during the CSD2X_FMEA_Cmod_Rb_Check function. For example, CSD2X_wCmod_Rb_Val contains 120,

which corresponds to 120 10-6 sec (or 120 µsec). This value is valid only if the CSD2X_bFMEA_Cmod_Rb_Check() API returns bRetVal.4.

**C Prototype:**

Single channel:

```
BYTE CSD2X_bFMEA_Cmod_Rb_Check(void);
```

Dual channel:

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

**Assembly:**

```
lcall  CSD2X_bFMEA_Cmod_Rb_Check
```

**Parameters:**

None

**Return Value:**

bRetVal.0:Cmod shorted to GND.

bRetVal.1:Cmod 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.5:Cmod and Rb product is out of range or Cmod and/or Rb are unconnected.

Single Channel:

| Constant | Value | Description |
|---|---|---|
| CSD2X_CMOD_SHORTED_TO_GND | 0x00 | Cmod shorted to GND |
| CSD2X_CMOD_SHORTED_TO_VDD | 0x01 | Cmod shorted to VDD |
| CSD2X_RB_SHORTED_TO_GND | 0x02 | Rb shorted to GND |
| CSD2X_RB_SHORTED_TO_VDD | 0x03 | Rb shorted to VDD |
| CSD2X_CMODRB_WITHIN_40 | 0x04 | Cmod and Rb product is within +40% |
| CSD2X_CMODRB_WITHOUT_40 | 0x05 | Cmod and Rb product is out of range |

Dual Channel:

| Constant | Value | Description |
|---|---|---|
| CSD2X_LEFT_CMOD_SHORTED_TO_GND | 0x00 | Left Cmod shorted to GND |
| CSD2X_ LEFT_CMOD_SHORTED_TO_VDD | 0x01 | Left Cmod shorted to VDD |
| CSD2X_LEFT_RB_SHORTED_TO_GND | 0x02 | Left Rb shorted to GND |
| CSD2X_LEFT_RB_SHORTED_TO_VDD | 0x03 | Left Rb shorted to VDD |
| CSD2X_LEFT_CMODRB_WITHIN_40 | 0x04 | Left Cmod and Rb product is within +40% |
| CSD2X_LEFT_CMODRB_WITHOUT_40 | 0x05 | Left Cmod and Rb product is out of range |
| CSD2X_RIGHT_CMOD_SHORTED_TO_GND | 0x00 | Right Cmod shorted to GND |
| CSD2X_RIGHT_CMOD_SHORTED_TO_VDD | 0x01 | Right Cmod shorted to VDD |
| CSD2X_RIGHT_RB_SHORTED_TO_GND | 0x02 | Right Rb shorted to GND |
| CSD2X_RIGHT_RB_SHORTED_TO_VDD | 0x03 | Right Rb shorted to VDD |
| CSD2X_RIGHT_CMODRB_WITHIN_40 | 0x04 | Right Cmod and Rb product is within +40% |
| CSD2X_RIGHT_CMODRB_WITHOUT_40 | 0x05 | Right Cmod and Rb product is out of range |

**Note**

1. Disable the background scanning parameter before using FMEA_Cmod_Rb_Check.

2.If Rb is less than 6 k ohms, then the 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 the 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.

## 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 usage 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(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 the buttons pressing
}
}
}
```

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

```
 //--------------------------------------------------------------------------
// Sample C code for the 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.** 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();
    //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_Cmod_Check() != 4)
        // Uncomment this line for Single Channel with IDAC configuration
    {
        // Add user code here to alarm Cmod issues
        // CSD2X_wCmod_Val
    }
        // if (CSD2X_bFMEA_Cmod_Rb_Check() != 4)
        // Uncomment this line for Single Channel with Rb configuration
    {
        // 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
    }
}
```

## Configuration Registers

Table 4. Block CapSense, Register: CS_CR0

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Value | 0 | 0 | CSD2X_ PRSCLK | 0 | 1 | 0 | 0 | EN |

Table 5. Block CapSense, Register: CS_CR1

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

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

Table 6. Block CapSense, Register: CS_CR2

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

Table 7. Block CapSense, Register: CS_CR3

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

Table 8. Block CapSense, Register: CS_CNTH

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

Table 9. Block CapSense, Register: CS_CNTL

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

Table 10. Block CapSense, Register: PRS_CR

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

Table 11. Block Timer, Register: PT1_CFG

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

Table 12. Block Timer, Register: PT1_DATA0

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

Table 13. Block Timer, Register: PT1_DATA1

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

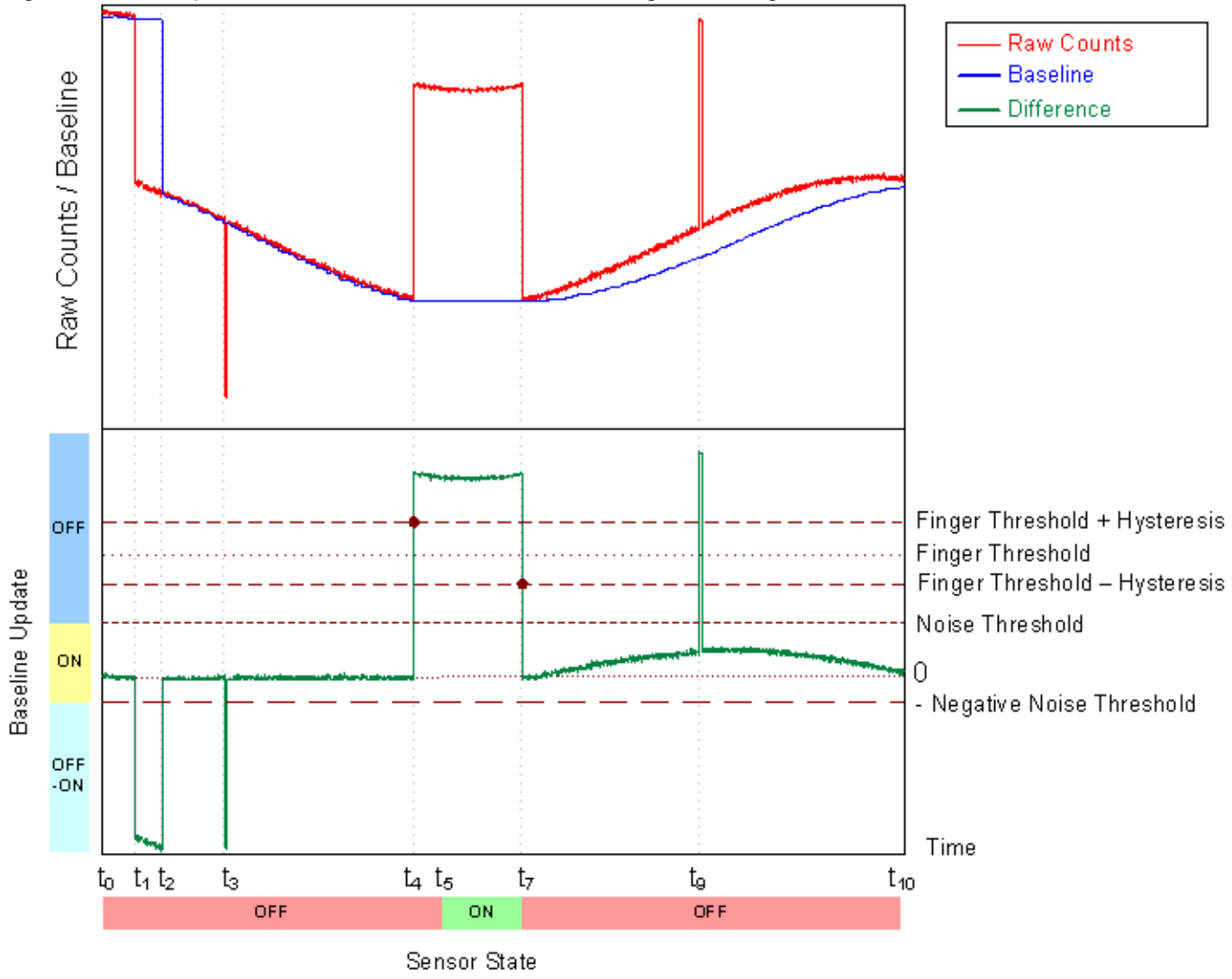
# Appendix

The following sections contain information beyond what is usually included in user module datasheets. The detailed information was developed by Cypress engineers to help you successfully design CapSense applications. 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 UM 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 17. 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.

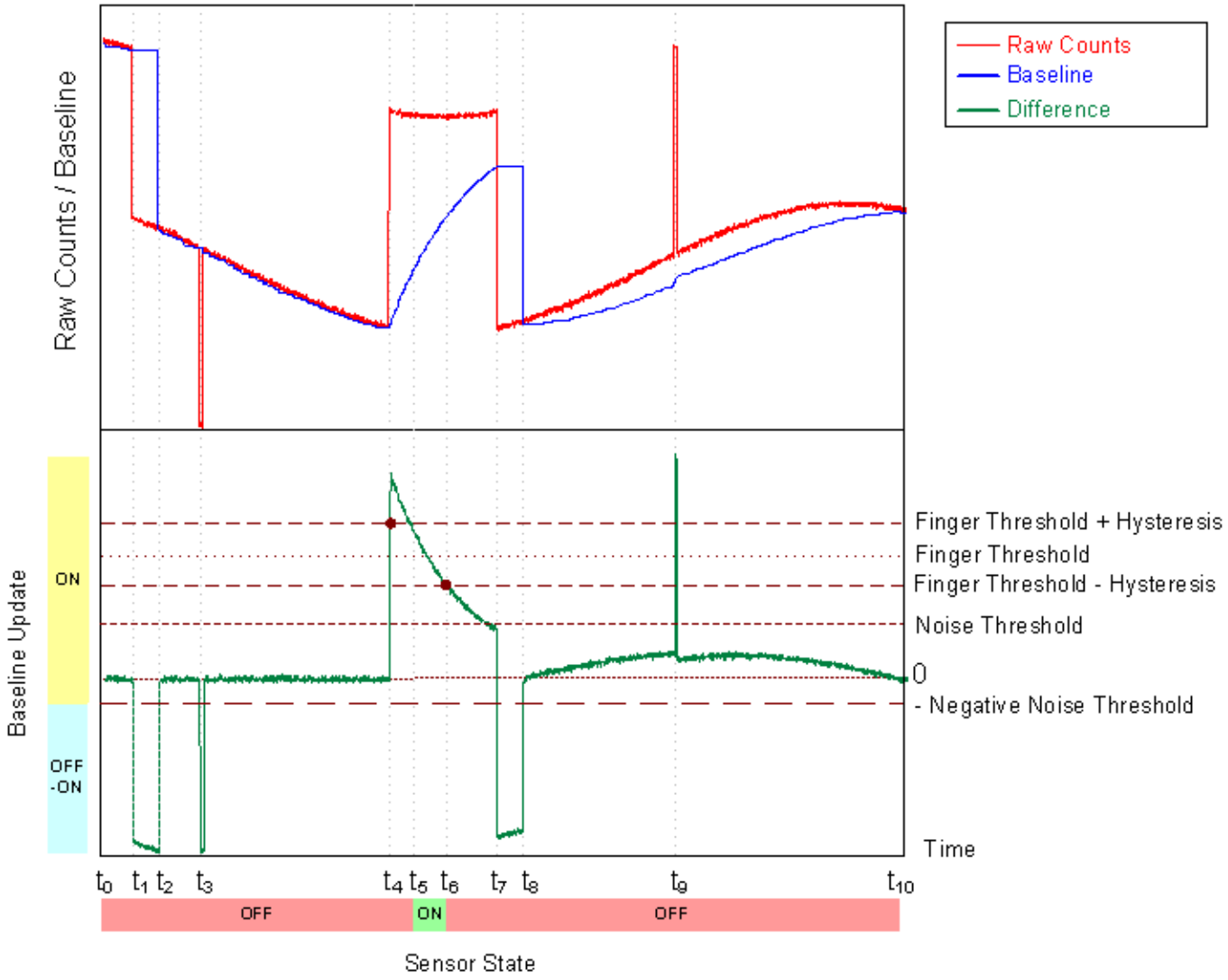
At $t_1$ the raw count drops sharply and the negative difference exceeds the NegativeNoiseThreshold. This situation can happen if the device is powered on when a finger is on the sensor and then the finger is removed after a period of time. At this time, the baseline update mechanism is frozen and an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

Figure 18. 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 scanning of the previous pair is complete. 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 does not 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 sensor positions are assigned in the GUI according to sensor placement from left top to right bottom. To ensure the efficiency of your dual channel scanning, 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 |
| 2.1 | DHA | 1. Removed the following parameters from CSD2x Wizard:<br>IDAC Value, Finger Threshold, Reference Value, Scan Speed, Scanning Resolution<br><br>2. Disabled option to set individual per-sensor scanning parameters in Wizard. This saves RAM and ROM space.<br><br>3. Added the following new parameters to User Module parameters window:<br>Finger Threshold, Reference Value (Left/Right), IDAC Value (Compensation/Left/Right), Scanning Speed, Resolution. These parameters are applied to all sensors at startup.<br><br>4. Increased Reference Value (Left/Right) parameter resolution to 5 bits.<br><br>5. Added the following new API functions for consistency with other CapSense User Modules:<br>SetScanMode (Left/Right), SetPrescaler (Left/Right), SetRefSource (Left/Right), SetRefValue (Left/Right), SetDACValue (Left/Right)<br><br>6. Stored all scanning parameters in RAM. This allows setting different scanning parameters for individual sensors at runtime.<br><br>7. Auto Calibration (if enabled) is now done automatically at startup. There is no need to call CSD2X_Calibrate API manually.<br><br>8. Updated IDAC Value input range.<br><br>9. Updated descriptions of Reference Value and Autocalibration parameters.<br><br>10. Changed parameter order in PSoC Designer.<br><br>11. Added references to datasheet sections.<br><br>12. Added SetRefSource API to datasheet.<br><br>13. Updated the descriptions of the following API:<br>CSD2X_DisableSensor, CSD2X_Start, CSD2X_SetScanMode<br><br>14. Changed MUM Configurations description.<br><br>15. Added support for new part numbers.<br><br>16. Added ranges for LowBaselineReset, NegativeNoiseThreshold, and Modular Capacitor Pin parameters.<br><br>17. Fixed range issue for Finger Threshold parameter.<br><br>18. Added prototypes for SetLeftRefSource and SetRightRefSource APIs.<br><br>19. Fixed other defects and cleaned up user module source code. |

| Version | Originator | Description |
|---|---|---|
| 2.20 | DHA | Added DRC warning message for unassigned sensors. |
| 2.30 | DHA | 1. Changed RAM_EPILOGUE to RAM_USE_CLASS_3 in Start() API.<br><br>2. Moved DiplexTables and Order_Table_Left/Right tables into the AREA lit.<br><br>3. Added wizard help button and file. |
| 2.40 | DHA | 1. Changed the CSD2X_Order_Table_Left to the CSD2X_Order_Table_Right in the CSD2X_InitializeSensorBaseline() API function implementation to fix baseline initialization.<br><br>2. Added "call `@INSTANCE_NAME`_ScanAllSensors" in CSD2X_Start() API function.<br><br>3. Added exports for the CSD2x_baDACCodeBaselineL and CSD2x_baDACCodeBaselineR IDAC tables.<br><br>4. Updated descriptions of the CSD2X_wGetPortPinLeft() and CSD2X_wGetPortPinRight() API functions.<br><br>5. Added new "GOO[4]" and "GOO[5]" selection options in the "Feedback Resistor Pin" parameter description.<br><br>6. Added more information about the sensor scanning time dependence on the prescaler value.<br><br>7. Set the ACOL[1:0] bitfield initialization value in the CSDx_CR1 register to 01b.<br><br>8. Added the RAM_SETPAGE_CUR >`@INSTANCE_NAME`_bBitMask in the SetIdacRange() API function. |
| 2.50 | DHA | Corrected the ACOL[1:0] bitfield initialization values in the CSDx_CR1 register for Double Channel with IDAC configuration. |

| Version | Originator | Description |
|---|---|---|
| 2.60 | DHA | 1. Updated area declarations to support Imagecraft optimization.<br><br>2. Added tables with information about scanning time in this user module datasheet.<br><br>3. Added symbolic names for the Resolution parameter in this user module datasheet.<br><br>4. Updated the Calibrate API to prevent IDAC compensation usage when the IDAC compensation parameter is set to 0.<br><br>5. Renamed SetLeftRefValue and SetRightRefValue APIs to SetLeftDACValue and vSetRightDACValue in the CSD2X_SGL.asm to address issues in IDAC configuration.<br><br>6. Added SetDACRange API support for old projects.<br><br>7. Updated the descriptions of the Feedback Resistor Pin and the SetScanMode() API function in this user module datasheet.<br><br>8. Updated the resolution range calculation for Slider and Radial Slider in the user module wizard.<br><br>9. Updated user module wizard help. Added a description of the slider resolution parameter min/max values. |
| 2.70 | DHA | 1. Corrected resolution value calculation in UM wizard to address the error after change in diplexing.<br><br>2. Changed calibration resolution from 9 bits to 12 bits in CSD_Start API. |
| 3.00 | DHA | 1. Added FMEA and Background Scanning functionality to user module.<br><br>2. Added slider labels in user module wizard.<br><br>3. Explained limitations of dynamic reconfiguration in the user module datasheet. |
| 3.10 | HPHA | 1. Added Resume() function to User Module API.<br><br>2. Implemented precharge in the CSD2X_ScanSensor function to eliminate noise when the project has long interrupts.<br><br>3. Fixed problem with saving information for sliders.<br><br>4. Updated baseline algorithm to check for negative difference counts.<br><br>5. Added build error message when user attempts to build project without first calling the user module wizard.<br><br>6. Optimized Start User Module function code.<br><br>7. Updated UM Wizard sliders setting algorithm to take into account free pins.<br><br>8. CSD counter reset bit initialization to eliminate incorrect results after first scan.<br><br>9. Added User Code section in User Module *.inc file to provide ability to customize 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 © 2009-2014 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

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

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

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

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