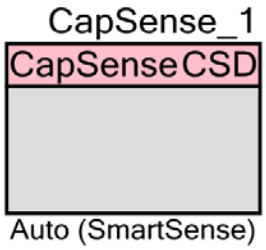
PSoC 4 Capacitive Sensing (CapSense® CSD)

2.20

Features



- Best-In-Class SNR performance
  - Superior noise-immunity performance against conducted and radiated external noise
  - Ultra-low radiated emissions
  - CapSense button support: Overlay thickness of up to 15 mm for glass and 5 mm for plastic
- SmartSense™ auto-tuning
  - Sets and maintains optimal sensor performance during run time
  - Eliminates manual tuning during development and production
- Advanced user interface features: Water tolerance
  - Shield electrode support for reliable operation in the presence of water droplets
  - Guard sensor to prevent false touches under the water or flowing water
- Support for user-defined combinations of button, linear slider, radial slider, touchpad and proximity capacitive sensors
- Easy to use Application Programming Interface (API) for fast proto-typing
- Integrated PC-based GUI for tuning in manual tuning mode (Refer to the *PSoC® 4 CapSense® Tuning Guide.*)

Note This document refers to PSoC 4 devices throughout. References to PSoC 4 should be interpreted to mean PSoC 4 and PSoC 4 BLE (Bluetooth Low Energy) devices. This component also supports the PRoC BLE device.

General Description

Capacitive sensing using a Delta-Sigma Modulator (CapSense CSD) is a versatile and efficient way to measure capacitance and detect finger touches in user interface panel applications such as capacitive touch buttons, sliders, touchpads, touch screens, and proximity sensors.

Read the following documents along with this datasheet. They can be found on the Cypress Semiconductor web site at www.cypress.com:

- *Getting Started with CapSense*
- *PSoC 4 CapSense Design Guide*

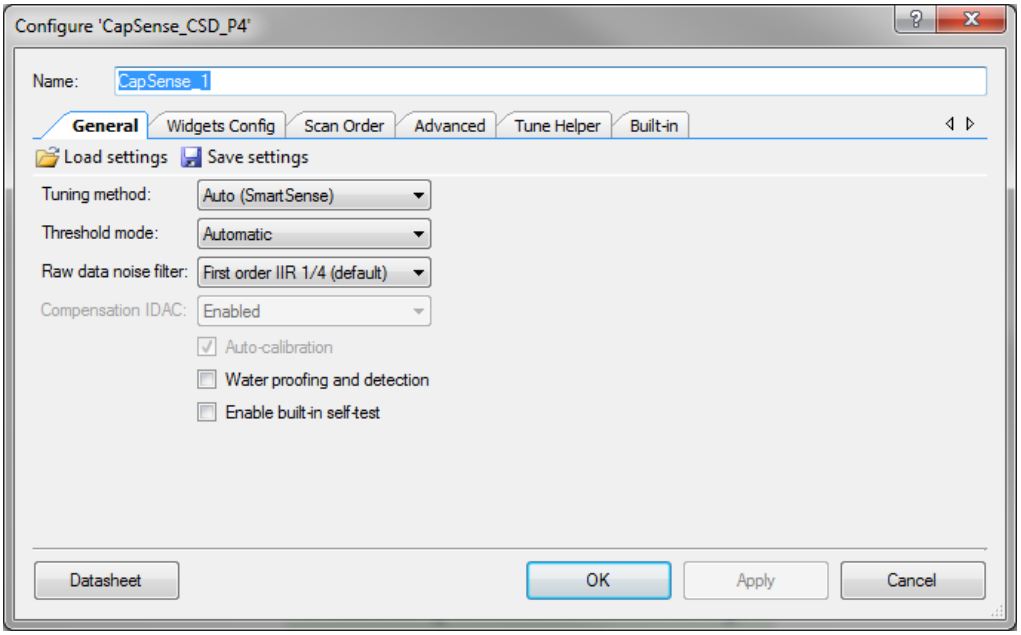
When to Use a CapSense Component

Capacitance sensing systems can be used in many applications in place of conventional buttons, switches, and other controls; even in applications that are exposed to rain or water. Such applications include automotive, outdoor equipment, ATMs, public access systems, portable devices such as cell phones and PDAs, and kitchen and bathroom applications.

Component Parameters

Drag a CapSense CSD component onto your design and double-click it to open the Configure dialog. This dialog has several tabs to guide you through the process of setting up the CapSense CSD component.

General Tab



Load Settings/Save Settings

Save Settings is used to save all settings and tuning data configured for a component. This allows quick duplication in a new project. **Load Settings** is used to load previously saved settings.

The stored settings can also be used to import settings and tuning data.

Tuning method

This parameter specifies the tuning method. Tuning consists of selecting optimal parameters for a given hardware configuration.

There are three options:

- **Auto (SmartSense)** – This option provides automatic tuning of the CapSense CSD component in supported range of Parasitic Capacitance (Cp) from 5 pF to 55 pF.

  This is the recommended tuning method for all designs. Firmware algorithms determine the best tuning parameters continuously at run time. Additional RAM and CPU resources are required in this mode. Use **Tuning method** “Manual with Run-Time Tuning” or “Manual” if specific tuning is required (strict control of scan time or if Cp is higher than 55 pF).

  Important SmartSense tuning may be used with I^2C communication, which is specified on the **Tune Helper** tab, to transmit data from the target device to the Tuner GUI.

- **Manual with Run-Time Tuning** – This option allows you to manually tune the CapSense CSD component using the Tuner GUI during run-time. Run-time tuning can be done using the Tuner GUI or using the API to change tuning parameters. Tuning parameters are stored in RAM.

  To launch the Tuner GUI, right-click on the symbol and select **Launch Tuner**. For more information about manual tuning, see the *PSoC® 4 CapSense® Tuning Guide*. Manual tuning requires I^2C communication, which is specified on the **Tune Helper** tab, to transmit data between the target device and the Tuner GUI.

- **Manual** – This option disables tuning.

  Setting to **Manual** (disabling run-time tuning) does not allow run-time tuning of the component, and all possible tuning parameters are stored in Flash.

Threshold mode

This parameter specifies the threshold mode when the **Tuning method** parameter is set to “Auto (SmartSense).” This parameter is not available when either manual option is selected. In manual tuning mode all thresholds are set manually.

There are two options:

- **Automatic (default)** – In this mode, the SmartSense algorithm automatically calculates and sets all sensor threshold values.
- **Flexible** – The flexible threshold is implemented by the component. In this case, the component accepts "Finger Threshold" for each widget and sets other threshold parameters based on the finger threshold:
  - lowBaselineReset = 30
  - hysteresis = 12.5 % of finger threshold
  - Noise Threshold = 50% of finger threshold
  - Negative Noise Threshold = 50% of finger threshold

Raw Data Noise Filter

This parameter selects the raw data filter. Only one filter can be selected and it is applied to all sensors. You should use a filter to reduce the effect of noise during sensor scans. Details about the types of filters can be found in the Filters section in this document.

- **None** – No filter is provided. No filter firmware or SRAM variable overhead is incurred.
- **Median** – Sorts the last three sensor values in order and returns the middle value.
- **Averaging** – Returns the simple average of the last three sensor values.
- **First Order IIR 1/2** – Returns one-half of the most current sensor value added to one-half of the previous filter value. IIR filters require the lowest firmware and SRAM overhead of all of the filter types.
- **First Order IIR 1/4** (default) – Returns one-fourth of the most current sensor value added to three-fourths of the previous filter value.
- **First Order IIR 1/8** – Returns one-eighth of the most current sensor value added to seven-eighths of the previous filter value.
- **First Order IIR 1/16** – Returns one-sixteenth of the most current sensor value added to fifteen-sixteenths of the previous filter value.
- **Jitter** – If the most current sensor value is greater than the last sensor value, the previous filter value is incremented by 1; if it is less, the value is decremented.

Compensation IDAC

This parameter enables the split IDACs mode. This mode provides increasing sensitivity and SNR. The **Compensation IDAC** is connected to the amuxbus full time during CapSense operation and is intended to compensate for the sensor's parasitic capacitance.

- Disabled (default)
- Enabled

Note The **Compensation IDAC** parameter is always enabled for the Auto (SmartSense) Tuning method.

Auto-calibration check box

Enables or disables IDAC auto-calibration for manual Tuning method options. Default: Disabled.

Water proofing and detection

This feature configures the CapSense CSD component to support water proofing (disabled by default). This feature enables the Shield electrode. This feature sets the following parameters:

- Enables the Shield output terminal in the PSoC Creator Design-Wide Resources Pin Editor

  Note Not recommended to use the shield electrode with SmartSense tuning mode.

- Adds a Guard widget

  Note If you do not want the Guard widget with water proofing, you can remove it on the **Advanced** tab.

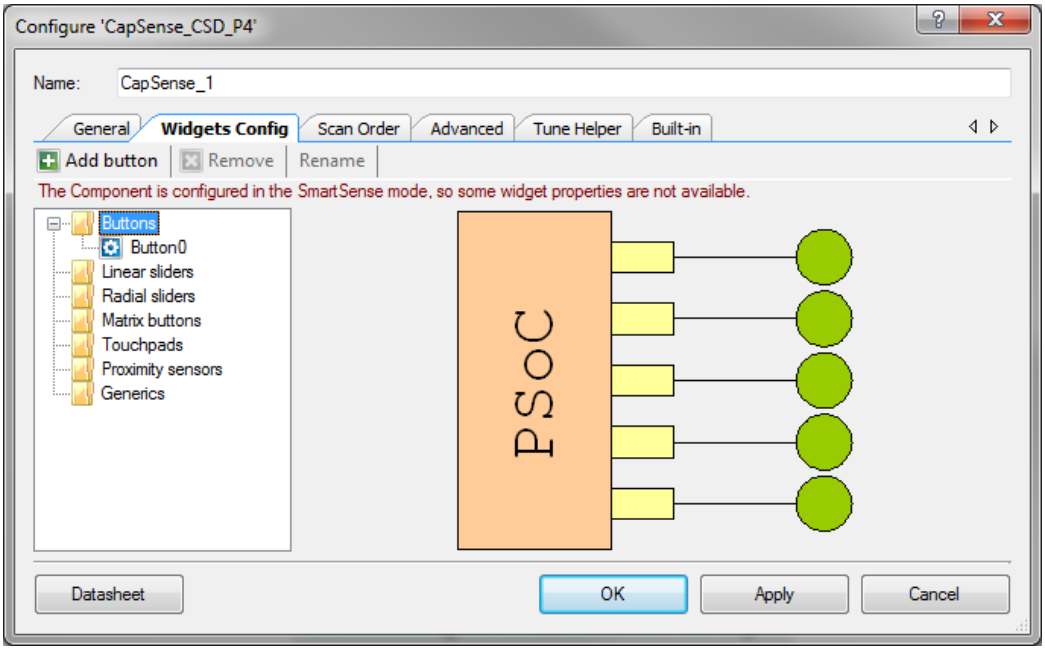
Enable BIST

This parameter enables the Built In Self Test (BIST) APIs that allow Cp and Cmod measuring. For SmartSense to operate correctly, the following must hold true:

- Cmod = 2.2 nF
- Sensor Cp < 55 pF

  Note If Cp > 55 pF, you can use the Manual Tuning method option and tune the sensors based on the higher sensor Cp, such that the Sense Clock Frequency meets the 5RC time constant.

Widgets Config Tab



Definitions for various parameters are provided in the Functional Description section.

Toolbar

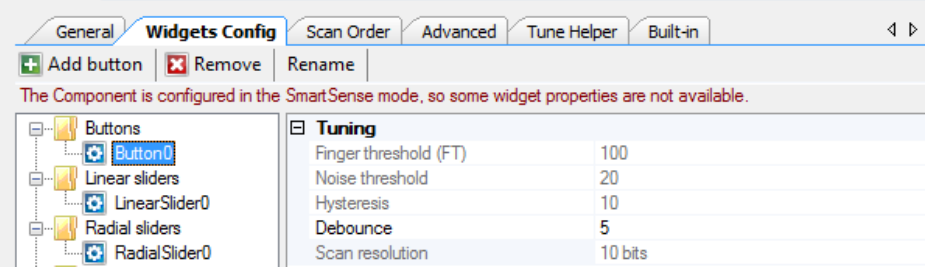
The toolbar contains the following commands:

- **Add widget** (hot key - Insert) – Adds the selected type of widget to the tree. The widget types are:
  - **Buttons** – A button detects a finger press on a single sensor and provides a single mechanical button replacement.
  - **Linear Sliders** – A linear slider provides an integer value based on interpolating the location of a finger press on a small number of sensors.
  - **Radial Sliders** – A radial slider is similar to a linear slider except that the sensors are placed in a circle.
  - **Matrix Buttons** – A matrix button detects a finger press at the intersection formed by a row sensor and column sensor. Matrix buttons provide an efficient method of scanning a large number of buttons.
  - **Touchpads** – A touchpad returns the X and Y coordinates of a finger press within the touchpad area. A touchpad is made of multiple row and column sensors.
  - **Proximity Sensors** – A proximity sensor is optimized to detect the presence of a finger, hand, or other large object at a large distance from the sensor. This avoids the need for an actual touch.

- **Generic Sensors** – A generic sensor provides raw data from a single sensor. This allows you to create unique or advanced sensors not otherwise possible with processed outputs of the other sensor types.

- **Remove** (hot key - Delete) – Removes the selected widget from the tree.
- **Rename** (hot key – F2) – Opens a dialog to change the selected widget name. You can also double-click a widget to open the dialog.

Buttons



Tuning:

- **Finger Threshold** – Defines sensor active threshold resulting in increased or decreased sensitivity to touches. When the sensor scan value is greater than this threshold the button is reported as touched. Default value is **100**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution.

  Finger Threshold + **Hysteresis** cannot be more than 254 for 8-bit widget resolution and 65534 for 16-bit widget resolution.

- **Noise Threshold** – Defines sensor noise threshold. Count values above this threshold do not update the baseline. If the noise threshold is too low, sensor and thermal offsets may not be accounted for. This can result in false or missed touches. If the noise threshold is too high, a finger touch may be interpreted as noise and artificially increase the baseline resulting in missed finger touches. Default value is **20**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution.

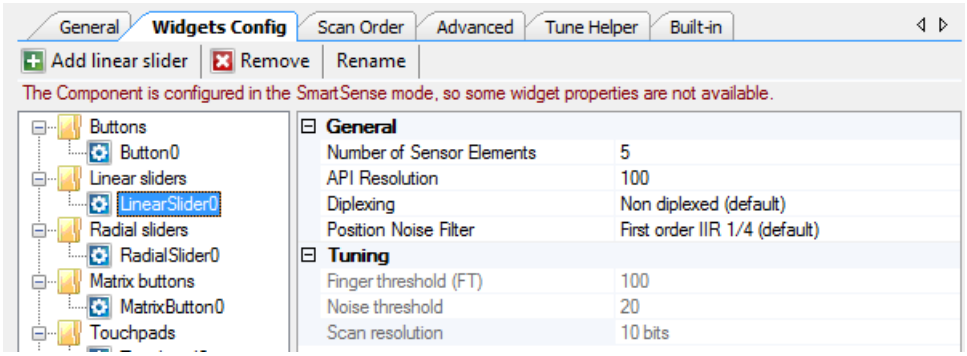
- **Hysteresis** – Adds differential hysteresis for sensor active state transitions. If the sensor is inactive, the difference count must overcome the finger threshold plus hysteresis. If the sensor is active, the difference count must go below the finger threshold minus hysteresis. Hysteresis helps to ensure that low-amplitude sensor noise and small finger moves do not cause cycling of the button state. Default value is **10**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution. **Finger Threshold** + **Hysteresis** cannot be more than 254 for 8-bit widget resolution and 65534 for 16-bit widget resolution.

- **Debounce** – Adds a debounce counter to detect the sensor active state 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. Default value is **5**. Debounce ensures that high-frequency high-amplitude noise does not cause false detection of a pressed button. Valid range of values is [1…255].
- **Scan Resolution** – Defines the scanning resolution. This parameter affects the scanning time of the sensor within the button widget. The maximum raw count for the scanning resolution for N bits is $2^N - 1$. Increasing the resolution improves sensitivity and the signal-to-noise ratio (SNR) of touch detection but increases scan time. Default value is **10 bits**. Valid range of values is [6…16].

Note These parameters (except for Finger Threshold) are not available for SmartSense mode and are automatically set by the SmartSense algorithm. For Manual mode, the following values are recommended:

- Finger Threshold = 80% of signal
- Noise Threshold = Negative Noise Threshold = 50% of Finger Threshold (Advanced tab)
- Hysteresis = 12.5% of Finger Threshold
- Debounce = 3
- Low Baseline Reset = 30 (Advanced Tab)

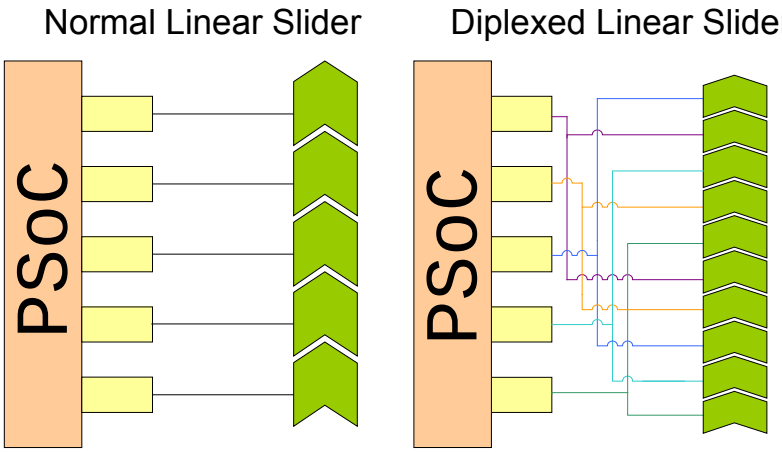
Linear Sliders



General:

- **Numbers of Sensor Elements** – Defines the number of elements within the slider. A good ratio of API resolution to sensor elements is 20:1. Increasing the ratio of API resolution to sensor elements too much can result in increased noise on the calculated finger position. Valid range of values is [2…32]. Default value is **5** elements.

- **API Resolution** – Defines the slider resolution. The position value will be changed within this range. Valid range of values is [1…255]. Default value is 100.
- **Diplexing – Non diplexed** (default) or **Diplexed**. Diplexing allows two slider sensors to share a single device pin, which reduces the total number of pins required for a given number of slider sensors. Minimum number of sensor elements for a diplexed slider is 5.



- **Position Noise Filter** – Selects the type of noise filter to perform on position calculations. Only one filter can be applied for a selected widget. Details about the types of filters can be found in the Filters section in this document.
  - **None**
  - **Median**
  - **Averaging**
  - **First Order IIR 1/2**
  - **First Order IIR 1/4 (default**
  - **Jitter**

Tuning:

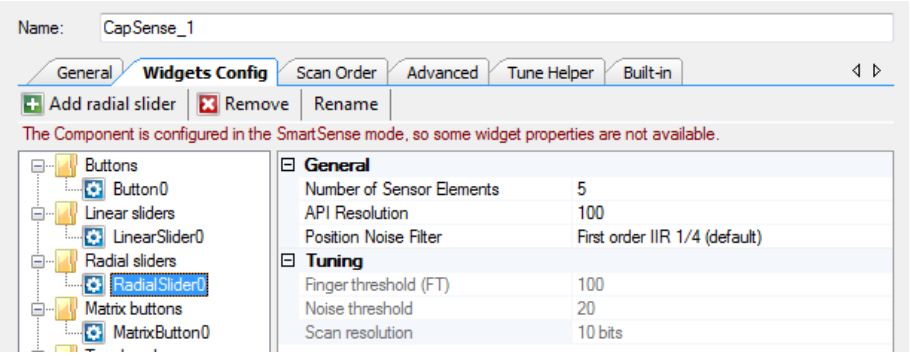
- **Finger Threshold** – Defines sensor active threshold resulting in increased or decreased sensitivity to touches. When the sensor scan value is greater than this threshold the button is reported as touched. Default value is **100**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution.
- **Noise Threshold** – Defines the sensor noise threshold for slider elements. Count values above this threshold do not update the baseline. If the noise threshold is too low, sensor and thermal offsets may not be accounted for. This can result in false or missed touches. If the noise threshold is too high, a finger touch may be interpreted as noise and artificially increase the baseline resulting in centroid location calculation errors. Count values below this threshold are not counted in the calculation of the centroid. Default value is **20**. Valid

range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution.

- **Scan Resolution** – Defines the scanning resolution. This parameter affects the scanning time of all sensors within the linear slider widget. The maximum raw count for scanning resolution for N bits is $2^N - 1$. Increasing the resolution improves sensitivity and the SNR of touch detection but increases scan time. Default value is **10 bits**. Valid range of values is [6…16].

Note The **Noise Threshold** and **Scan Resolution** parameters are not available for SmartSense mode and are automatically set by the SmartSense algorithm.

Radial Slider



General:

- **Numbers of Sensor Elements** – Defines the number of elements within the slider. A good ratio of API resolution to sensor elements is 20:1. Increasing the ratio of API resolution to sensor elements too much can result in increased noise on the resolution calculation. Valid range of values is [2…32]. Default value is **5** elements.
- **API Resolution** – Defines the resolution of the slider. The position value will be changed within this range. Valid range of values is [1…255]. Default value is 100.
- **Position Noise Filter** – Selects the type of noise filter to perform on position calculations. Only one filter may be applied for a selected widget. Details about the types of filters can be found the Filters section of this datasheet.
  - None
  - Median
  - Averaging
  - First Order IIR 1/2
  - First Order IIR 1/4 (default)
  - Jitter

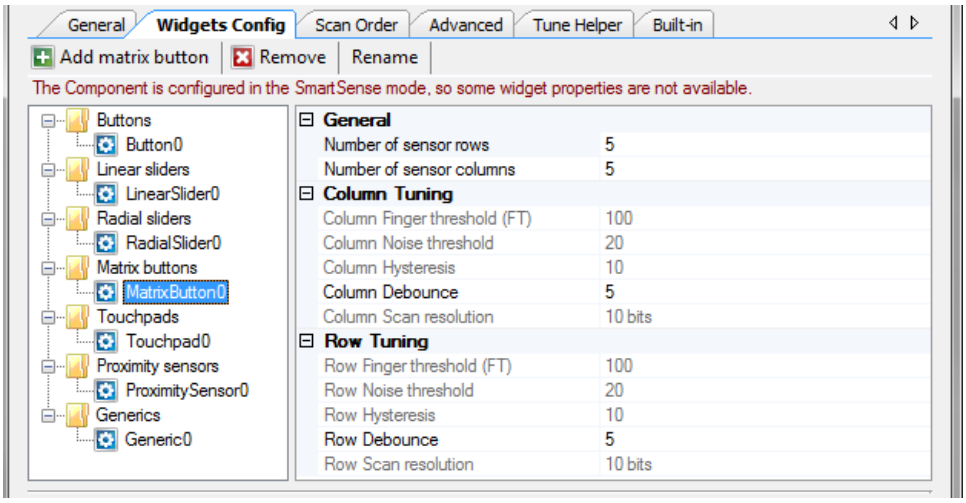
Tuning:

- **Finger Threshold** – Defines the sensor active threshold resulting in increased or decreased sensitivity to touches. When the sensor scan value is greater than this threshold the button is reported as touched. Default value is **100**.
- **Noise Threshold** – Defines the sensor noise threshold for slider elements. Count values above this threshold do not update the baseline. If the noise threshold is too low, sensor and thermal offsets may not be accounted. This can result in false or missed touches. If the noise threshold is too high, a finger touch may be interpreted as noise and artificially increase the baseline resulting in centroid location calculation errors. Count values below this threshold are not counted in the calculation of the centroid. Default value is **20**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution.
- **Scan Resolution** – Defines the scanning resolution. This parameter affects the scanning time of all sensors within a radial slider widget. The maximum raw count for scanning resolution for N bits is $2^N - 1$. Increasing the resolution improves sensitivity and the SNR of touch detection but increases scan time. Default value is **10 bits**. Valid range of values is [6…16].

Note The **Noise Threshold** and **Scan Resolution** parameters are not available for SmartSense mode and are automatically set by the SmartSense algorithm.

Note Position Noise Averaging and IIR filters are not recommended for the Radial Sliders because such filters use the previous data for updating the current one. This can cause a false position calculation when a finger is moving from the last to first slider segment.

Matrix Buttons



General:

- **Number of sensor columns and rows** – Defines the number of columns and rows that form the matrix. Valid range of values is [2…32]. Default value is **5** elements for both columns and rows.

Tuning:

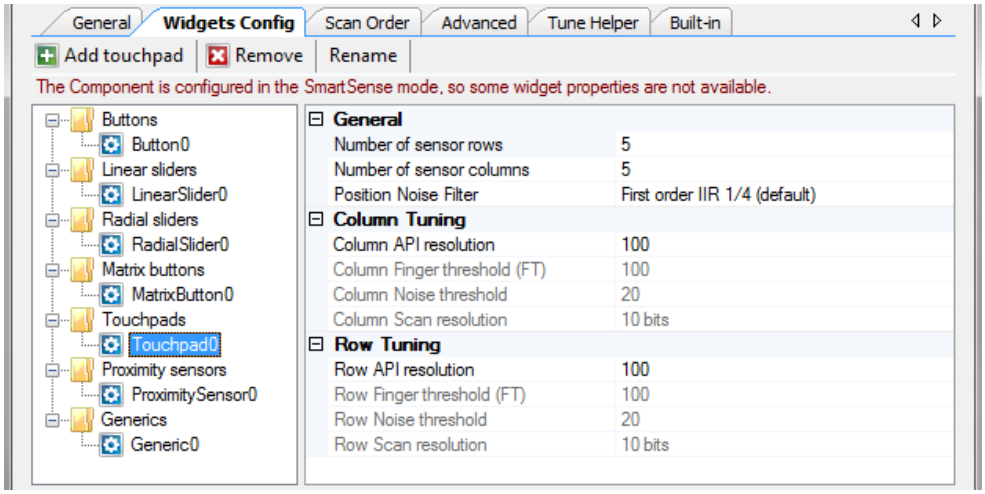
- **Column and Row Finger Threshold** – Defines the sensor active threshold for matrix button columns and rows resulting in increased or decreased sensitivity to touches. When the sensor scan value is greater than this threshold the button is reported as touched. Default value is **100**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution. **Finger Threshold** + **Hysteresis** cannot be more than 254 for 8-bit widget resolution and 65534 for 16-bit widget resolution.
- **Column and Row Noise Threshold** – Defines the sensor noise threshold for matrix button columns and rows. Count values above this threshold do not update the baseline. If the noise threshold is too low, sensor and thermal offsets may not be accounted for. This can result in false or missed touches. If the noise threshold is too high, a finger touch may be interpreted as noise and artificially increase the baseline. This can result in missed finger touches. Default value is **20**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution.
- **Column and Row Hysteresis** – Adds differential hysteresis for sensor active state transitions for matrix button columns and rows. 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. Hysteresis helps to ensure that low-amplitude sensor noise and small finger moves do not cause cycling of the button state. Default value is **10**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16 bit-widget resolution. **Finger Threshold** + **Hysteresis** cannot be more than 254 for 8-bit widget resolution and 65534 for 16-bit widget resolution.
- **Column and Row Debounce** – Adds a debounce counter for detection of the sensor active state transition for matrix buttons column or row. 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. Default value is **5**. Debounce ensures that high-frequency high-amplitude noise does not cause false detection of a pressed button. Valid range of values is [1…255].
- **Column and Row Scan Resolution** – Defines the scanning resolution of matrix button columns and rows. This parameter affects the scanning time of all sensors within a column or row of a matrix button widget. The maximum raw count for scanning resolution for N bits is $2^N - 1$. Increasing the resolution improves sensitivity and the SNR of touch detection but increases scan time. The column and row scanning resolutions should be

the same to get the same sensitivity level. Default value is **10 bits**. Valid range of values is [6…16].

Note The **Noise Threshold**, **Hysteresis**, **Debounce**, and **Scan Resolution** parameters are not available for SmartSense mode and are automatically set by the SmartSense algorithm. For Manual mode, the following values are recommended:

- Finger Threshold = 80% of signal
- Noise Threshold = Negative Noise Threshold = 50% of Finger Threshold(Advanced Tab)
- Hysteresis = 12.5% of Finger Threshold
- Debounce = 3
- Low Baseline Reset = 30 (Advanced Tab)

Touchpads



General:

- **Numbers of sensor columns and rows** – Defines the number of columns and rows that form the touchpad. Valid range of values is [2…32]. Default value is **5** elements for both the column and row.
- **Position Noise Filter** – Adds noise filter to position calculations. Only one filter may be applied for a selected widget. Details on the types of filters can be found in the Filters section in this datasheet.
  - None
  - Median
  - Averaging

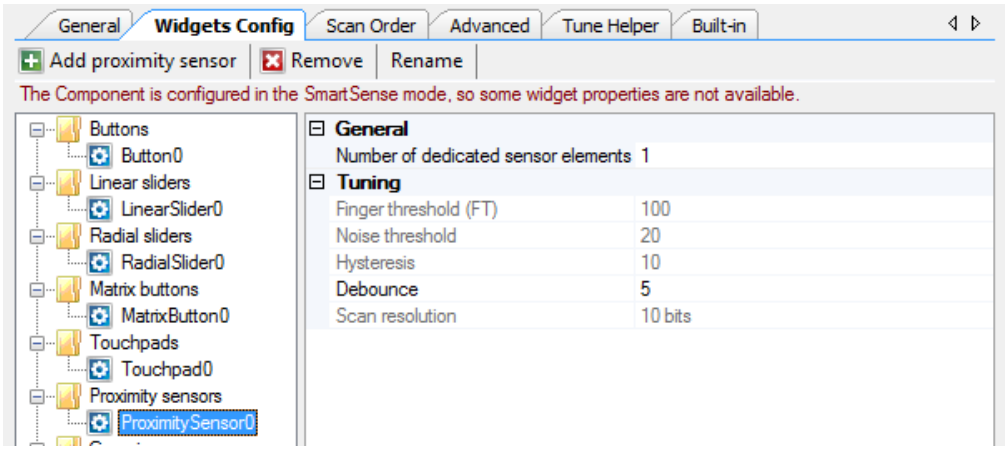
- First Order IIR 1/2
- First Order IIR 1/4 (default)
- Jitter

Tuning:

- **Column and Row API Resolution**– Defines the resolution of the touchpad columns and rows. The finger position values are reported within this range. Default value is **100**. Valid range of values is [1…255].
- **Column and Row Finger Threshold** – Defines the sensor active threshold for touchpad columns and rows resulting in increased or decreased sensitivity to touches. When the sensor scan value is greater than this threshold the touchpad reports the touch position. Default value is **100**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16 bit-widget resolution.
- **Column and Row Noise Threshold** – Defines the sensor noise threshold for touchpad columns and rows. Count values above this threshold do not update the baseline. Count values below this threshold are not counted in the calculation of the centroid location. If the noise threshold is too low sensor and thermal offsets may not be accounted for. This can result in false or missed touches. If the noise threshold is too high a finger touch may be interpreted as noise and artificially increase the baseline. This can result in centroid calculation errors. Default value is **20**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16 bit-widget resolution.
- **Column and Row Scan Resolution** – Defines the scanning resolution of touchpad columns and rows. This parameter affects the scanning time of all sensors within a column or row of a touchpad widget. The maximum raw count for scanning resolution for N bits is $2^N - 1$. Increasing the resolution improves sensitivity and the SNR of touch detection but increases scan time. The column and row scanning resolution should be equal to get the same sensitivity level. Default value is **10 bits**. Valid range of values is [6…16].

Note The **Noise Threshold** and **Scan Resolution** parameters are not available for SmartSense mode and are automatically set by SmartSense algorithm.

Proximity Sensors



Note All widgets are enabled by default except proximity widgets. Proximity widgets must be manually enabled in API as their long scan time is incompatible with the fast response required of other widget types. Use the CapSense_EnableWidget() function to enable proximity widgets. See How to use the proximity sensors for more information about proximity sensors.

General:

- **Number of Dedicated Sensor Elements** – Selects the number of dedicated proximity sensors. These sensor elements are in addition to all of the other sensors used for other Widgets. Any Widget sensors may be used individually or connected together in parallel to create proximity sensors.
  - **0** – The proximity sensor only scans one or more existing sensors to determine proximity. No new sensors are allocated for this widget.
  - **1** (default) – Number of dedicated proximity sensors in the system. All dedicated sensors form one complex proximity sensor and are scanned with common parameters.

Tuning:

- **Finger Threshold** – Defines the sensor active threshold resulting in increased or decreased sensitivity to the proximity of a touch. When the sensor scan value is greater than this threshold the proximity sensor is reported as touched. Default value is **100**. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16 bit-widget resolution. **Finger Threshold** + **Hysteresis** cannot be more than 254 for 8-bit widget resolution and 65534 for 16-bit widget resolution.
- **Noise Threshold** – Defines the sensor noise threshold. Count values above this threshold do not update the baseline. If the noise threshold is too low, sensor and thermal offsets may not be accounted for. This can result in false or missed proximity touches. If the noise threshold is too high, a figure touch may be interpreted as noise and artificially increase the baseline. This can result in missed finger touches. Valid range of values is

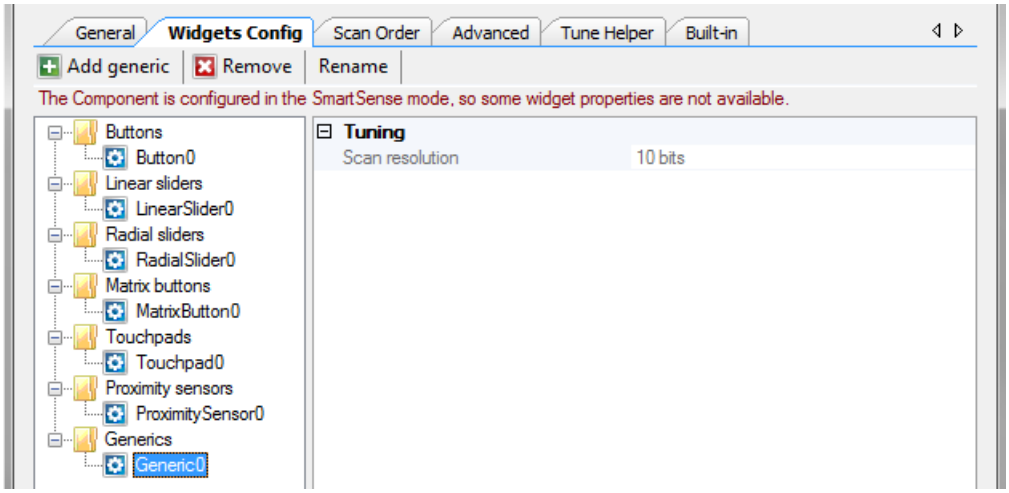
[1…255] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution. Default value is 20.

- **Hysteresis** – Adds differential hysteresis for the sensor active state transition. 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. Hysteresis helps to ensure that low amplitude sensor noise and small finger or body moves do not cause cycling of the proximity sensor state. Valid range of values is [1…255] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution. Default value is 10.
- **Debounce** – Adds a debounce counter to detect the sensor active state 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. Debounce ensures that high-frequency high-amplitude noise does not cause false detection of a proximity event. Valid range of values is [1…255]. Default value is 5.
- **Scan Resolution** – Defines the scanning resolution. This parameter affects the scanning time of a proximity widget. The maximum raw count for scanning resolution for N bits is $2^N - 1$. Increasing the resolution improves sensitivity and the SNR of touch detection but increases scan time. It is best to use a higher resolution for proximity detection than what is used for a typical button to increase detection range. Default value is **16 bits**. Valid range of values is [6…16].

Note The **Noise Threshold**, **Hysteresis**, **Debounce**, and **Scan Resolution** parameters are not available for SmartSense mode and are automatically set by the SmartSense algorithm. For Manual mode, the following values are recommended:

- Finger Threshold = 80% of signal
- Noise Threshold = Negative Noise Threshold = 50% of Finger Threshold(Advanced Tab)
- Hysteresis = 12.5% of Finger Threshold
- Debounce = 3
- Low Baseline Reset = 30 (Advanced Tab)

Generics

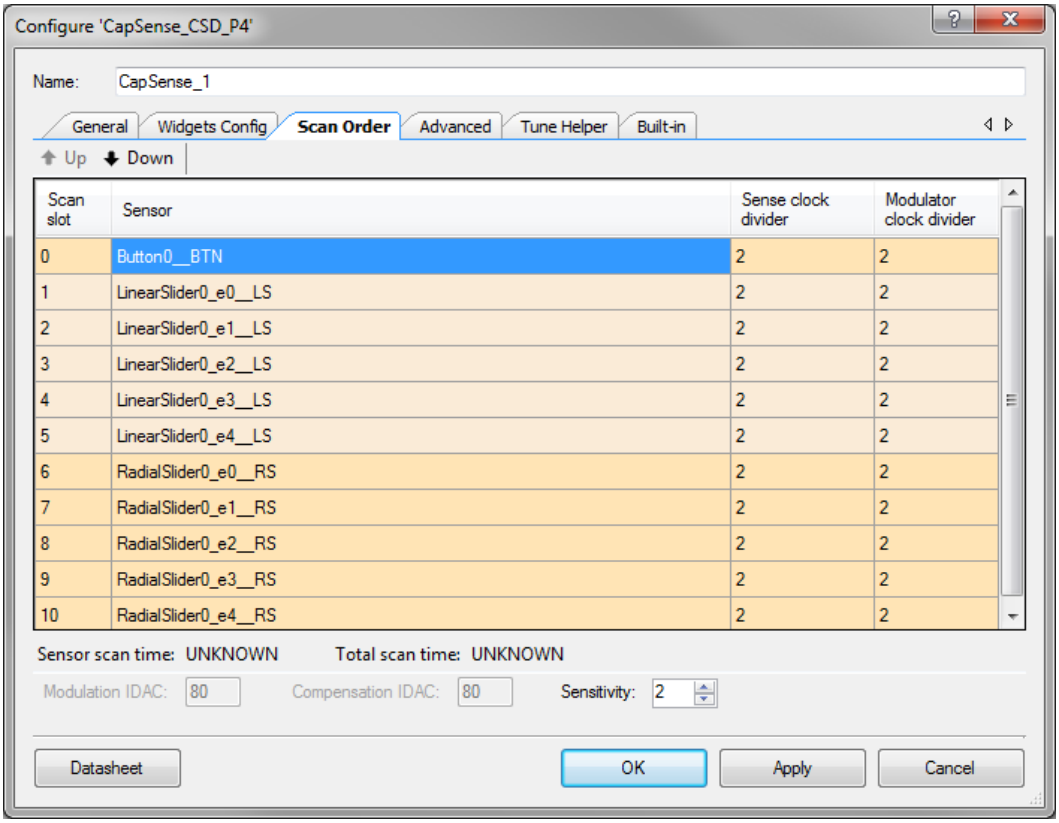


Tuning:

- **Scan Resolution** – Defines the scanning resolution. This parameter affects the scanning time of a generic widget. The maximum raw count for scanning resolution for N bits is $2^N - 1$. Increasing the resolution improves sensitivity and the SNR of touch detection but increases scan time. Default value is **10 bits**.

Only one tuning option is available for a generic widget because all high-level handling is left to you to support CapSense sensors and algorithms that do not fit into any of the predefined widgets.

Scan Order Tab



Note Scan order does not affect the performance; the default scan order is good enough for most applications.

Toolbar

The toolbar contains the following commands:

- **Up/Down** (hot key - Add/Subtract) – Moves the selected widget up or down in the data grid. The whole widget is selected if one or more of its elements are selected.

Note You should reassign pins if the scanning order changes.

Note A proximity sensor is excluded from the scanning process by default. Its scan must be started manually at run time because it is typically not scanned at the same time as the other sensors.

Additional Hot Keys:

- **Ctrl + A** – Select all sensors.
- **Delete** – Remove all sensors from the complex sensor (applies to generic and proximity widgets).

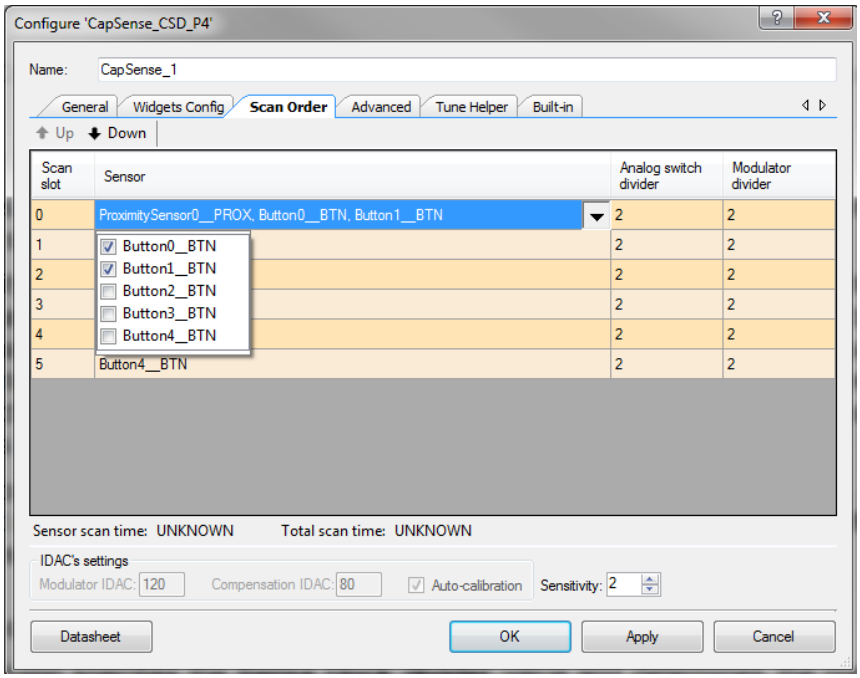
Widget List

Widgets are listed in alternating gray and orange rows in the table. All sensors associated with a widget share the same color to highlight different widget elements.

Complex sensors

Proximity scan sensors can use dedicated proximity sensors, or they can detect proximity from a combination of dedicated sensors, other sensors, or both. Such complex sensors form a Sensor Scan Slot, where all dedicated sensors have the same parameters during scanning.

For example, the board may have a trace that goes all the way around an array of buttons and the proximity sensor may be made up of the trace and all of the buttons in the array. All of these sensors are scanned at the same time to detect proximity. A drop-down list is provided on proximity scan sensors to choose one or more dedicated sensors to scan to detect proximity. These sensors can be assigned to the complex proximity sensor using check boxes opposite each sensor in the drop down list.



Like proximity sensors, generic sensors can also consist of multiple sensors. A generic sensor can get data from a dedicated sensor, any other existing sensor, or from multiple sensors. Select the sensors with the drop down list provided.

Sense clock divider

This column specifies the **Sense clock divider** value and determines the precharge switch output frequency for scan slot. The clock frequency on the sensor pin equals the HFCLK frequency divided by the Sense Clock divider value. Valid range of values is [2…255] for PSoC 4100/PSoC 4200/ PSoC 4100M/PSoC 4200M devices and [1…255] for PSoC 4000 devices. Default value is 2.

This column is hidden if the **Individual frequency setting** is disabled (on the **Advanced** tab).

The Sense Clock divider is the most critical Hardware parameter for properly tuning a Capsense design. It depends on the selected HFCLK (IMO), and the Cp of the sensor(s) being scanned. The following shows the recommended Sense Clock Divider settings based on these parameters:

Cp, pF	PSoC 4000			PSoC 4100/PSoC 4200/ PSoC 4100M/PSoC 4200M [1]		
	12 MHz	6 MHz	3 MHz	48 MHz	24 MHz	12 MHz
<15	1	1 [2]	1 [2]	2	2 [2]	2 [2]
16-34	2	1	1 [2]	4	2	2 [2]
35-60	4	2	1	8	4	2

Modulator clock divider

This column specifies the **Modulator clock divider** value and determines the modulator input frequency for scan slot. The Modulator Clock frequency equals the HFCLK frequency divided by the Modulator Clock divider value. Valid range of values is [2…255] for PSoC 4100/PSoC 4200/ PSoC 4100M/PSoC 4200M devices and [1…255] for PSoC 4000 devices. Default value is 2.

This column is hidden if the **Individual frequency setting** is disabled (on the **Advanced** tab).

Details of the clock configuration can be found in CapSense Clocking in the Functional Description section.

Sensor scan time and Total scan time labels

The Sensor scan time label shows hardware scan time for selected sensor:

$(2^{resolution}-1)$ / Modulator Clock

Total scan time is sum of scan time of all sensors.

Note These labels show scan times that do not include processing time.

In Auto (Smartsense) tuning mode, the scan time is not shown. It depends on the resolution, which is set automatically in Auto (Smartsense) tuning mode. The Sensor Scan Time and resolution values in Auto (Smartsense) tuning mode are given in the Sensor Scan Time section.

---

[1] In PSoC 4100/PSoC 4200 devices, the Sense Clock also depends on the Modulator Clock Divider because these dividers are chained. Data is provided for Modulator Clock Divider = 2. For more details, refer to the CapSense Clocking section

[2] This combination of the Sense Clock and Cp is not recommended because the switching frequency will be too low to give good performance. For this Cp we recommend the HFCLK frequency is increased.

Modulation IDAC

This field specifies the Modulation IDAC value. Valid range is 0 to 255 (0 to 250 for PSoC 4100/PSoC 4200 devices) for 4x range and 0 to 125 for 8x range. Default value is **80**. Details of the IDACs configuration can be found in CapSense Analog System in the Functional Description section in this datasheet.

Compensation IDAC

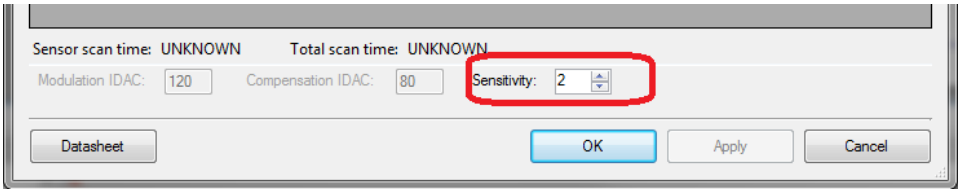
This field specifies the Compensation IDAC value. Valid range is 0 to 127. Default value is **80**.

Note The Sense Clock Divider, Modulator Clock Divider, Compensation IDAC, and Modulation IDAC parameters are not available in SmartSense mode. Refer to the *PSoC® 4 CapSense® Tuning Guide* for additional Tuning details in SmartSense and Manual modes.

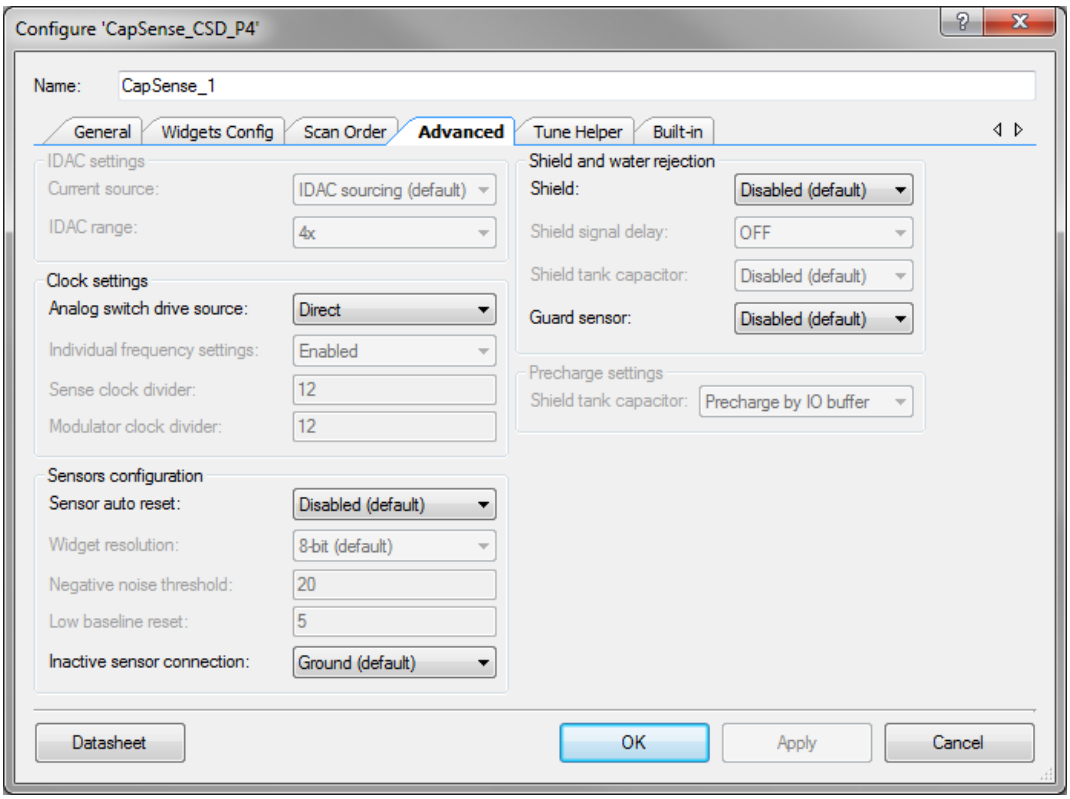
Sensitivity

The **Sensitivity** parameter in SmartSense mode represents the nominal change in Cs (sensor capacitance) required to activate a sensor. The valid range of values is [1…10], which corresponds to sensitivity levels: 0.1, 0.2, 0.3, and 1 pF. The default value is 2. The recommended range is 0.1-0.4 pF. Sensitivity sets the overall sensitivity of the sensors to account for the different thicknesses of overlay material. Thicker material should use a lower sensitivity value.

The **Sensitivity** parameter is available for Auto (Smartsense) tuning mode only:



Advanced Tab



Current Source

The CapSense CSD componentrequires a precision current source for detecting touch on the sensors. **IDAC Sinking** and **IDAC Sourcing** require the use of IDAC on the PSoC device.

- **IDAC Sourcing** (default) – The IDAC sources the current into the modulation capacitor C_{MOD}. The analog switches are configured to alternate between the modulation capacitor C_{MOD} and GND, providing a sink for the current. **IDAC Sourcing** is recommended for most designs because it provides the greatest signal-to-noise ratio.
- **IDAC Sinking** – The IDAC sinks current from the modulation capacitor C_{MOD}. The analog switches are configured to alternate between V_{DD} and the modulation capacitor C_{MOD} providing a source for the current. This works well in most designs, although SNR is generally not as high as the **IDAC Sourcing** mode.

IDAC range

This parameter specifies the IDAC range of the **Current Source**. The lower and higher current ranges are generally only used with non-touch-capacitive based sensors.

- 4x (default)
- 8x

Analog Switch Drive Source

This parameter specifies the source of the **Sense Clock Divider**, which determines the rate at which the sensors are switched to and from the modulation capacitor C_{MOD}.

- Direct (default)
- PRS-8b
- PRS-12b
- PRS-Auto

Note Refer to the *PSoC® 4 CapSense® Tuning Guide*, CapSense Tuning Process section to determine when you could use Direct clock or PRS.

Individual Frequency Settings

This parameter defines the **Sense Clock Divider** usage. If enabled, each scan slot uses a dedicated Sense Clock Divider value (set in **Scan Order** tab). Otherwise, sensors use only one **Sense Clock Divider** value and **Modulator Clock Divider** value that are set below this parameter. Individual Frequency Settings are recommended to be enabled if the parasitic capacitances of the sensors are not similar.

Sense Clock Divider

This parameter specifies the value of the **Sense Clock Divider** and determines the precharge switch output frequency. Valid range of values is [2…255] for PSoC 4100/PSoC 4200/ PSoC 4100M/PSoC 4200M devices and [1…255] for PSoC 4000 devices. Default value is **12**.

This feature is unavailable if **Individual Frequency Settings** are **enabled**.

The sensors are continuously switched to and from the modulation capacitor C_{MOD} at the speed of the precharge clock. The **Sense Clock Divider** divides the CapSense CSD clock to generate the precharge clock. When the divider value is decreased, the sensors are switched faster and the raw counts increase and vice versa.

Details of the clock configuration can be found in the CapSense Clocking section in this datasheet.

Modulator Clock Divider

This parameter specifies the value of the **Modulator Clock Divider** and determines the modulator input frequency. Valid range of values is [2…255] for PSoC 4100/PSoC 4200/ PSoC 4100M/PSoC 4200M devices and [1…255] for PSoC 4000 devices. Default value is **12**.

When the divider value is decreased, the scan time is decreased and vice versa.

This feature is unavailable if **Individual Frequency Settings** are **enabled**.

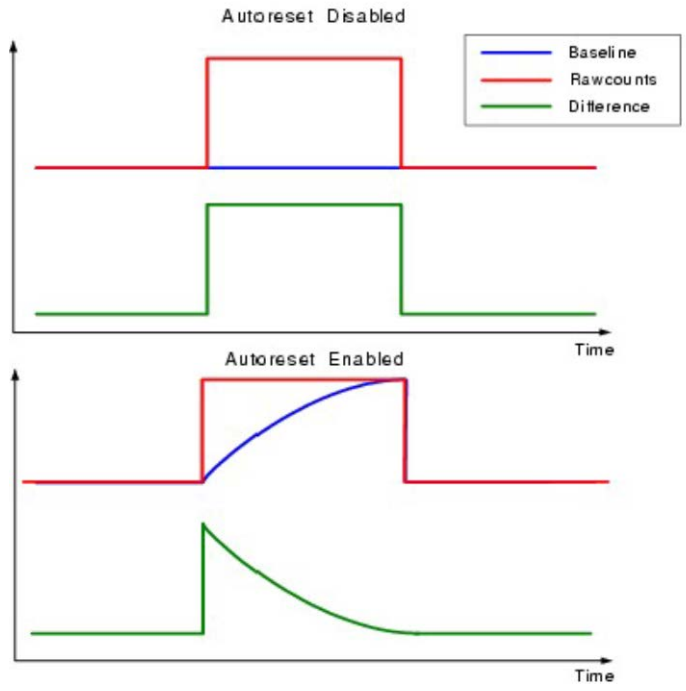
Note In PSoC 4100/PSoC 4200 devices, the **Modulator Clock Divider** should be a multiple of the **Sense Clock Divider** since these dividers are chained. For more details, refer to the CapSense Clocking section.

Sense Clock Divider and **Modulator Clock Divider** are not available in SmartSense mode. Refer to the *PSoC® 4 CapSense® Tuning Guide* for additional Tuning details in the SmartSense and Manual modes.

Sensor Auto Reset

This parameter enables auto reset, which causes the baseline to always update regardless of whether the difference counts are above or below the noise threshold. When auto reset is disabled, the baseline only updates when difference counts are within the plus/minus noise threshold (the noise threshold is mirrored). 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.

- **Enabled** – Auto reset ensures that the baseline is always updated, avoiding missed button presses and stuck buttons, but limits the maximum length of time a button will report as pressed. This setting limits the maximum time duration of the sensor (typical values are 5 to 10 seconds), 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.
- **Disabled** (default) – Abnormal system conditions can cause the baseline to stop updating by continuously exceeding the noise threshold. This can result in missed button presses or stuck buttons. The benefit is that a button can continue to report its pressed state indefinitely. You may need to provide an application-dependent method of determining stuck or unresponsive buttons.



Widget Resolution

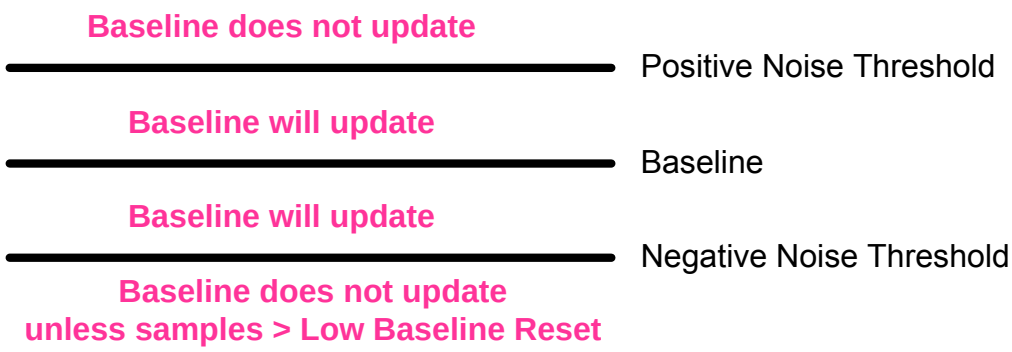
This parameter specifies the signal resolution that the widget reports. 8 bits (1 byte) is the default option and should be used for the vast majority of applications. If widget values exceed the 8-bit range, the system is too sensitive and should be tuned to move the nominal value to approximately mid range (~128). Slider and Touchpad widgets that require high accuracy can benefit from 16-bit resolution. 16-bit resolution increases linearity by avoiding rounding errors possible with 8 bits but at the expense of additional SRAM usage of two bytes per sensor.

- 8-bit (1 byte) – default
- 16-bit (2 bytes)

Negative Noise Threshold

This parameter specifies the negative difference between the raw count and baseline levels for baseline resetting to the raw count level. If raw counts are below this level, the baseline will not reset unless the **Low Baseline Reset** parameter limit is reached. In that case, the baseline will reset. Refer to the following figure, which shows the relationship between the noise thresholds and baseline reset. A good starting point for Negative Noise Threshold is to use the same value as Noise Threshold.

Valid range of values is [5…255]. Default value is 20.



Low Baseline Reset

This parameter defines the number of samples with raw counts less than baseline needed to make the baseline snap down to the raw count level. Valid range of values is [1…255]. Default value is 5.

Inactive Sensor Connection

This parameter defines the default sensor connection for all sensors not being actively scanned.

- **Ground** (default) – Use this for the vast majority of applications as it reduces noise on the actively scanned sensors.
- **Hi-Z Analog** – Leaves the inactive sensors at Hi-Z.
- **Shield** – Provides the shield waveform to all unscanned sensors. The amplitude of the shield signal is equal to the amplitude of the signal on the scanned sensor. Provides increased water proofing and lower noise when used with the shield electrode. This feature is unavailable if **Shield** is **disabled**.

  Note Inactive Sensor Connection changes to Shield when the Shield is set to Enabled.

Shield

This parameter specifies if the shield electrode output, which is used to remove the effects of water droplets and water films, is enabled or disabled. For more information about shield electrode usage, see the Shield Electrode section.

- Disabled (default)
- Enabled

Shield signal delay

This parameter specifies the number of HFCLK cycles that the CapSense shield is delayed relative to the signal on the sensor pin.

- None (default)
- 1 cycle
- 2 cycle

Note For correct shield operation, the shield signal should be in phase with the signal on the sensor.

Shield tank capacitor enable

This parameter specifies whether pin for the off-chip Ctank capacitor connection, in parallel with shield capacitance, is enabled. This capacitor is intended to increase the shield capacitance. Shield tank capacitor helps to reduce phase difference between the shield and sensor clocks in case the shield Cp is really high. Also Ctank capacitor needs to be enabled when either Cmod precharge or Csh_tank precharge are configured as “Precharge by IO buffer”.

- Disabled (default)
- Enabled

Guard Sensor

This parameter enables the guard sensor, which helps detect water drops in an application that requires water proofing. This feature is enabled automatically if **Water Proofing and detection** (under the **General** tab) is selected. For more information about the Guard sensor, see the Functional Description section of this datasheet.

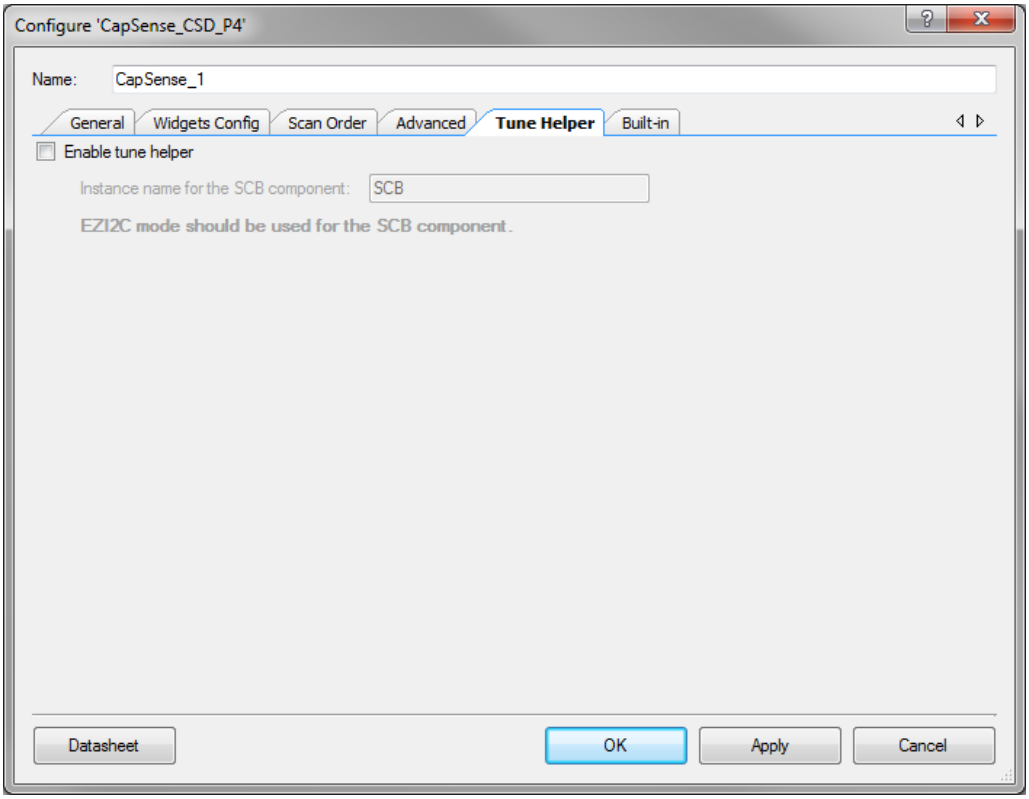
- Disabled (default)
- Enabled

Csh_tank precharge

This parameter specifies Vref source for driving the shield electrode.

- Precharge by Vref buffer (default)
- Precharge by IO buffer

Tune Helper Tab



Enable Tune Helper

This parameter adds functions to support easier communication with the Tuner GUI. Select this feature if you are going to use the Tuner GUI. If this option is not selected, the communication functions are still provided but do nothing. Therefore, when tuning is complete or the tuning method is changed you do not need to remove these functions. Disabled by default.

EzI2C component instance name

This parameter defines the instance name for the EZI2C component in your design to be used for communication with the Tuner GUI.

For more information about how to use Tuner GUI, refer to the *PSoC® 4 CapSense® Tuning Guide*.

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table provides an overview of each function. The subsequent sections cover each function in more detail.

Component can be used in IDEs that support the following compilers:

- ARM GCC compiler
- ARM MDK compiler
- ARM RealView compiler
- IAR C/C++ compiler

Note If using the IAR Embedded Workbench, set the path to the static library. This library is located in the following PSoC Creator installation directory:

PSoC Creator\psoc\content\CyComponentLibrary\CyComponentLibrary.cylib\CortexM0\IAR

By default, PSoC Creator assigns the instance name “CapSense_1” to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “CapSense.”

General APIs

These are the general CapSense API functions that place the component into operation or halt operation:

Function	Description
CapSense_Start()	Preferred method to start the component. Initializes registers and enables active mode power template bits of the subcomponents used within CapSense. In Smartsense tuning mode the API adjusts the parameters such as Sense Clock Divider, IDACs and resolution based on the calculated parasitic capacitances.
CapSense_Stop()	Disables component interrupts, and calls CapSense_ClearSensors() to reset all sensors to an inactive state.
CapSense_Sleep()	Prepares the component for the device entering a low-power mode. Disables Active mode power template bits of the sub components used within CapSense, saves non-retention registers, and resets all sensors to an inactive state.
CapSense_Wakeup()	Restores CapSense configuration and non-retention register values after the device wake from a low power mode sleep mode.
CapSense_Init()	Initializes the default CapSense configuration provided with the customizer.
CapSense_Enable()	Enables the Active mode power template bits of the subcomponents used within CapSense.
CapSense_SaveConfig()	Saves the configuration of CapSense.
CapSense_RestoreConfig()	Restores CapSense configuration.

void CapSense_Start(void)

Description: This is the preferred method to begin component operation. CapSense_Start() calls the CapSense_Init() function, and then calls the CapSense_Enable() function. Initializes registers and starts the CSD method of the CapSense component. Resets all sensors to an inactive state. Enables interrupts for sensors scanning. When SmartSense tuning mode is selected, the tuning procedure is applied for all sensors. In Smartsense tuning mode the API adjusts the parameters such as Sense Clock Divider, IDACs and resolution based on the calculated parasitic capacitances. The CapSense_Start() routine must be called before any other API routines.

Parameters: None

Return Value: None

Side Effects: Global interrupts (CyGlobalIntEnable;) must be enabled before CapSense_Start() if the Auto (Smartsense) Tuning method or Auto-calibration is selected.

void CapSense_Stop(void)

Description: Stops the sensor scanning, disables component interrupts, and resets all sensors to an inactive state. Disables Active mode power template bits for the subcomponents used within CapSense.

Parameters: None

Return Value: None

Side Effects: This function should be called after all scanning is completed.

void CapSense_Sleep(void)

Description: This is the preferred method to prepare the component for device low-power modes. Disables Active mode power template bits for the subcomponents used within CapSense. Calls CapSense_SaveConfig() function to save customer configuration of CapSense and resets all sensors to an inactive state.

Parameters: None

Return Value: None

Side Effects: This function should be called after scans are completed.

This function does not put pins used by CapSense component into lowest power consumption state.

void CapSense_Wakeup(void)

Description: Restores the CapSense configuration. Restores the enabled state of the component by setting Active mode power template bits for the subcomponents used within CapSense.

Parameters: None

Return Value: None

Side Effects: This function does not restore pins used by the CapSense component to the state they were before.

void CapSense_Init(void)

Description: Initializes the default CapSense configuration provided by the customizer that defines component operation. Resets all sensors to an inactive state.

Parameters: None

Return Value: None

Side Effects: None

void CapSense_Enable(void)

Description: Enables Active mode power template bits for the subcomponents used within CapSense.

Parameters: None

Return Value: None

Side Effects: None

void CapSense_SaveConfig(void)

Description: Saves the configuration of CapSense. Resets all sensors to an inactive state.

Parameters: None

Return Value: None

Side Effects: This function should be called after scanning is complete.

This function does not put pins used by CapSense component into lowest power consumption state.

void CapSense_RestoreConfig(void)

Description: Restores CapSense configuration.

Parameters: None

Return Value: None

Side Effects: This function should be called after scanning is complete.

This function does not restore pins used by the CapSense component to the state they were in before.

Scanning Specific APIs

These API functions are used to implement CapSense sensor scanning.

Function	Description
CapSense_ScanSensor()	Sets scan settings and starts scanning a sensor or group of combined sensors.
CapSense_ScanWidget()	Sets scan settings and starts scanning a widget.
CapSense_ScanEnabledWidgets()	The preferred scanning method. Scans all of the enabled widgets.
CapSense_IsBusy()	Returns the status of sensor scanning.
CapSense_SetScanSlotSettings()	Sets the scan settings of the selected scan slot (sensor).
CapSense_ClearSensors()	Resets all sensors to the nonsampling state.
CapSense_EnableSensor()	Configures the selected sensor to be scanned during the next scanning cycle.
CapSense_DisableSensor()	Disables the selected sensor so it is not scanned in the next scanning cycle.
CapSense_ReadSensorRaw()	Returns sensor raw data from the CapSense_SensorResult[ ] array.
CapSense_ReadCurrentScanningSensor()	Returns scanning sensor number when sensor scan is in progress.

void CapSense_ScanSensor(uint32 sensor)

Description: Sets scan settings and starts scanning a sensor. After scanning is complete, the ISR copies the measured sensor raw data to the global raw sensor array. Use of the ISR ensures this function is non-blocking. Each sensor has a unique number within the sensor array. This number is assigned by the CapSense customizer in sequence.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: None

Side Effects: None

void CapSense_ScanWidget (uint32 widget)

Description: Sets scan settings and starts scanning a widget.

Parameters: uint32 widget: Widget number. For every widget there are defines in this format:

```
#define CapSense_"widget_name"__"widget type"      "Widget number"
```

Example:

```
#define CapSense_TOUCHPAD0__TP                     5
```

All widget names are upper case. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: None

Side Effects: None

void CapSense_ScanEnabledWidgets(void)

Description: This is the preferred method to scan all of the enabled widgets. Starts scanning a sensor within the enabled widgets. The ISR continues scanning sensors until all enabled widgets are scanned. Use of the ISR ensures this function is non-blocking.

All widgets are enabled by default except proximity widgets. Proximity widgets must be manually enabled as their long scan time is incompatible with the fast response required of other widget types.

Parameters: None

Return Value: None

Side Effects: If no widgets are enabled the function call has no effect.

uint32 CapSense_IsBusy (void)

Description: Returns the status of sensor scanning.

Parameters: None

Return Value: uint32: Returns the state of scanning. ‘1’ – scanning in progress, ‘0’ – scanning completed.

Side Effects: None

void CapSense_SetScanSlotSettings(uint32 slot)

Description: Sets the scan settings provided in the customizer or wizard of the selected scan slot (sensor). The scan settings provide an IDAC value for every sensor, as well as resolution. The resolution is the same for all sensors within a widget.

Parameters: uint32 slot: Scan slot number

Return Value: None

Side Effects: None

void CapSense_ClearSensors(void)

Description: Resets all sensors to the nonsampling state by sequentially disconnecting all sensors from the Analog MUX Bus and connecting them to the inactive state.

Parameters: None

Return Value: None

Side Effects: None

void CapSense_EnableSensor(uint32 sensor)

Description: Configures the selected sensor to be scanned during the next measurement cycle. The corresponding pins are set to Analog HI-Z mode and connected to the Analog Mux Bus. This also affects the comparator output.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: None

Side Effects: None

void CapSense_DisableSensor(uint32 sensor)

Description: Disables the selected sensor. The corresponding pins are disconnected from the Analog Mux Bus and put into the inactive state.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: None

Side Effects: None

uint16 CapSense_ReadSensorRaw(uint32 sensor)

Description: Returns sensor raw data from the global CapSense_SensorResult[ ] array. Each scan sensor has a unique number within the sensor array. This number is assigned by the CapSense customizer in sequence. Raw data can be used to perform calculations outside of the CapSense provided framework.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint16: Current raw data value

Side Effects: None

uint32 CapSense_ReadCurrentScanningSensor(void)

Description: This API returns the sensor ID of the sensor which is being scanned currently. The API returns 0xFFFFFFFF when no sensor is being scanned.

Parameters: None

Return Value: uint32: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Side Effects: None

High-Level APIs

These API functions are used to work with raw data for sensor widgets. The raw data is retrieved from scanned sensors and converted to on/off for buttons, position for sliders, or X and Y coordinates for touchpads.

Function	Description
CapSense_InitializeSensorBaseline()	Loads the CapSense_sensorBaseline[sensor] array element with an initial value by scanning the selected sensor.
CapSense_InitializeEnabledBaselines()	Loads the CapSense_sensorBaseline[] array with initial values by scanning enabled sensors only. This function is available only for two-channel designs.
CapSense_InitializeAllBaselines()	Loads the CapSense_sensorBaseline[] array with initial values by scanning all sensors.
CapSense_UpdateSensorBaseline()	The historical count value, calculated independently for each sensor, is called the sensor's baseline. This baseline updated uses a low-pass filter with k = 256.
CapSense_UpdateEnabledBaselines()	Checks the CapSense_sensorEnableMask[] array and calls the CapSense_UpdateSensorBaseline() function to update the baselines for enabled sensors.
CapSense_EnableWidget()	Enables all sensor elements in a widget for the scanning process.

Function	Description
CapSense_DisableWidget()	Disables all sensor elements in a widget from the scanning process.
CapSense_CheckIsWidgetActive()	Compares the selected of widget to the CapSense_Signal[] array to determine if it has a finger press.
CapSense_CheckIsAnyWidgetActive()	Uses the CapSense_CheckIsWidgetActive() function to find if any widget of the CapSense CSD component is in active state.
CapSense_GetCentroidPos()	Checks the CapSense_sensorSignal[] array for a finger press in a linear slider and returns the position.
CapSense_GetRadialCentroidPos()	Checks the CapSense_sensorSignal[] array for a finger press in a radial slider widget and returns the position.
CapSense_GetTouchCentroidPos()	If a finger is present, this function calculates the X and Y position of the finger by calculating the centroids within the touchpad.
CapSense_GetMatrixButtonPos()	If a finger is present, this function calculates the row and column position of the finger on the matrix buttons.
CapSense_CheckIsSensorActive()	Returns true if sensor is active.
CapSense_GetBaselineData()	Reads sensor baseline.
CapSense_GetDiffCountData()	Returns difference count data.
CapSense_GetNormalizedDiffCountData()	Returns normalized difference count data.
CapSense_GetNoiseThreshold()	Returns the noise threshold value.
CapSense_GetNegativeNoiseThreshold()	Returns the negative noise threshold value.
CapSense_GetNoiseEnvelope()	Returns the measured noise envelope value.
CapSense_GetFingerThreshold()	Returns finger threshold value.
CapSense_GetFingerHysteresis()	Returns Hysteresis value.
CapSense_WriteSensorRaw()	Writes the raw count value.
CapSense_SetBaselineData()	Writes the baseline value.
CapSense_SetSensitivity()	Sets the sensitivity value.
CapSense_GetSensitivityCoefficient()	Returns the K coefficient.
Capsense_SetDebounce()	Sets the debounce value.
Capsense_GetDebounce()	Returns the debounce value.
CapSense_SetFingerHysteresis()	Sets the hysteresis value sensors.
CapSense_SetNoiseThreshold()	Sets the Noise Threshold value.
CapSense_SetNegativeNoiseThreshold()	Sets the Negative Noise Threshold value.
CapSense_SetLowBaselineReset()	Sets the low baseline reset threshold value.
CapSense_GetLowBaselineReset()	Returns the low baseline reset threshold value.

Function	Description
CapSense_SetFingerThreshold()	Sets the finger threshold value.
CapSense_SetDiffCountData()	Sets difference counts data.
CapSense_GetWidgetNumber()	Returns the widget number for the sensor.
CapSense_UpdateThresholds()	Updates the Thresholds.
CapSense_UpdateBaselineNoThreshold()	Updates sensor Baseline without updating the Thresholds.
CapSense_SetIDACRange()	Sets the IDAC range.
CapSense_GetIDACRange()	Returns the IDAC range.
CapSense_SetModulationIDAC()	Sets value for modulation IDAC.
CapSense_GetModulationIDAC()	Returns value for modulation IDAC.
CapSense_SetCompensationIDAC()	Sets value of compensation IDAC.
CapSense_GetCompensationIDAC()	Returns value of compensation IDAC.
CapSense_SetSenseClkDivider()	Sets value of sense clock divider.
CapSense_GetSenseClkDivider()	Returns value of sense clock divider.
CapSense_SetModulatorClkDivider()	Sets value of modulator sample clock divider.
CapSense_GetModulatorClkDivider()	Returns value of modulator sample clock divider.
CapSense_SetScanResolution()	Sets value of sensor scan resolution.
CapSense_GetScanResolution()	Returns value of sensor scan resolution.
CapSense_SetDriveModeAllPins()	Sets the drive mode of port pins.
CapSense_RestoreDriveModeAllPins()	Restore the drive for all CapSense port pins to original state.
CapSense_SetUnscannedSensorState()	Sets the state for un-scanned sensors.
CapSense_UpdateWidgetBaseline()	Updates the baselines for enabled sensors that belong to a widget.
CapSense_EnableRawDataFilters()	Enables the rawdata filters for the sensor signals.
CapSense_DisableRawDataFilters()	Disables the rawdata filters for the sensor signals.

void CapSense_InitializeSensorBaseline(uint32 sensor)

Description: Loads the CapSense_sensorBaseline[sensor] array element with an initial value by scanning the selected sensor. The raw count value is copied into the baseline array for each sensor. The raw data filters are initialized if enabled.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: None

Side Effects: None

void CapSense_InitializeEnabledBaselines(void)

Description: Scans all enabled widgets. The raw count values are copied into the CapSense_sensorBaseline[ ] array for all sensors enabled in scanning process. Initializes CapSense_sensorBaseline[ ] with zero values for sensors disabled from the scanning process. The raw data filters are initialized if enabled.

Parameters: None

Return Value: None

Side Effects: None

void CapSense_InitializeAllBaselines(void)

Description: Uses the CapSense_InitializeSensorBaseline() function to load the CapSense_sensorBaseline[ ] array with initial values by scanning all sensors. The raw count values are copied into the baseline array for all sensors. The raw data filters are initialized if enabled.

Parameters: None

Return Value: None

Side Effects: None

void CapSense_UpdateSensorBaseline(uint32 sensor)

Description: The sensor's baseline is a historical count value, calculated independently for each sensor. Updates the CapSense_sensorBaseline[sensor] array element using a low-pass filter with k = 256. The function calculates the difference count by subtracting the previous baseline from the current raw count value and stores it in CapSense_sensorSignal[sensor].

If the auto reset option is enabled, the baseline updates independent of the noise threshold.

If the auto reset option is disabled, the baseline stops updating if the signal is greater than the noise threshold and resets the baseline when the signal is less than the minus noise threshold.

Raw data filters are applied to the values if enabled before baseline calculation.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: None

Side Effects: None

void CapSense_UpdateEnabledBaselines(void)

Description: Checks the CapSense_sensorEnableMask [] array and calls the CapSense_UpdateSensorBaseline() function to update the baselines for all enabled sensors.

Parameters: None

Return Value: None

Side Effects: None

void CapSense_EnableWidget(uint32 widget)

Description: Enables the selected widget sensors to be part of the scanning process.

Parameters: uint32 widget: Widget number. For every widget there are defines in this format:

```
#define CapSense_"widget_name"__"widget type"          5
```

Example:

```
#define CapSense_MY_VOLUME1__LS                        5
#define CapSense_MY_UP__BNT                            6
```

All widget names are upper case. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: None

Side Effects: None

void CapSense_DisableWidget(uint32 widget)

Description: Disables the selected widget sensors from the scanning process.

Parameters: uint32 widget: Widget number. For every widget there are defines in this format:

```
#define CapSense_"widget_name"__"widget type"          5
```

Example:

```
#define CapSense_MY_VOLUME1__RS                        5
#define CapSense_MY_UP__MB                             6
```

All widget names are upper case. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: None

Side Effects: None

uint32 CapSense_CheckIsWidgetActive(uint32 widget)

Description: Compares the selected sensor CapSense_Signal[ ] array value to its finger threshold. Hysteresis and debounce are considered. If the sensor is active, the threshold is lowered by the hysteresis amount. If it is inactive, the threshold is increased by the hysteresis amount. If the active threshold is met, the debounce counter increments by one until reaching the sensor active transition, at which point this API sets the widget as active. This function also updates the sensor's bit in the CapSense_sensorOnMask[ ] array.

The touchpad and matrix buttons widgets need to have active sensor within column and row to return widget active status.

Parameters: uint32 widget: Widget number. For every widget there are defines in this format:

```
#define CapSense_"widget_name"__"widget type"          5
```

Example:

```
#define CapSense_MY_VOLUME1__LS                        5
```

All widget names are upper case. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint32: Widget sensor state. 1 if one or more sensors within the widget are active, 0 if all sensors within the widget are inactive.

Side Effects: This function also updates values in CapSense_sensorOnMask[ ] for all sensors belonging to the widget. The debounce counter is also modified on every call when there is a transition to the active state.

uint32 CapSense_CheckIsAnyWidgetActive(void)

Description: Compares all sensors of the CapSense_Signal[ ] array to their finger threshold. Calls Capsense_CheckIsWidgetActive() for each widget so that the CapSense_sensorOnMask[ ] array is up to date after calling this function.

Parameters: None

Return Value: uint32: 1 if any widget is active, 0 no widgets are active.

Side Effects: Has the same side effects as the CapSense_CheckIsWidgetActive() function but for all sensors.

uint16 CapSense_GetCentroidPos(uint32 widget)

Description: Checks the CapSense_Signal[ ] array for a finger press within a linear slider. The finger position is calculated to the API resolution specified in the CapSense customizer. A position filter is applied to the result if enabled. This function is available only if a linear slider widget is defined by the CapSense customizer.

Parameters: uint32 widget: Widget number. For every linear slider widget there are defines in this format:

```
#define CapSense_"widget_name"__LS          5
```

Example:

```
#define CapSense_MY_VOLUME1__LS             5
```

All widget names are upper case. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint16: Position value of the linear slider

Side Effects: If any sensors within the slider widget are active, the function returns values from zero to the API resolution value set in the CapSense customizer. If no sensors are active, the function returns 0xFFFF. If an error occurs during execution of the centroid/diplexing algorithm, the function returns 0xFFFF.

There are no checks of widget argument provided to this function. An incorrect widget value causes unexpected position calculations.

Note If noise counts on the slider segments are greater than the noise threshold, this subroutine may generate a false finger press result. The noise threshold should be set carefully (high enough above the noise level) so that noise will not generate a false finger press.

uint16 CapSense_GetRadialCentroidPos(uint32 widget)

Description: Checks the CapSense_Signal[ ] array for a finger press within a radial slider. The finger position is calculated to the API resolution specified in the CapSense customizer. A position filter is applied to the result if enabled. This function is available only if a radial slider widget is defined by the CapSense customizer.

Parameters: uint32 widget: Widget number. For every radial slider widget there are defines in this format:

```
#define CapSense_"widget_name"__RS          5
```

Example:

```
#define CapSense_MY_VOLUME2__RS             5
```

All widget names are upper case. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint16: Position value of the radial slider.

Side Effects: If any sensors within the slider widget are active, the function returns values from zero to the API resolution value set in the CapSense customizer. If no sensors are active, the function returns 0xFFFF.

There are no checks of widget type argument provided to this function. An incorrect widget value causes unexpected position calculations.

Note If noise counts on the slider segments are greater than the noise threshold, this subroutine may generate a false finger press result. The noise threshold should be set carefully (high enough above the noise level) so that noise will not generate a false finger press.

uint32 CapSense_GetTouchCentroidPos(uint32 widget, uint16* pos)

Description: If a finger is present on touchpad, this function calculates the X and Y position of the finger by calculating the centroids within the touchpad sensors. The X and Y positions are calculated to the API resolutions set in the CapSense customizer. Returns a '1' if a finger is on the touchpad. A position filter is applied to the result if enabled. This function is available only if a touchpad is defined by the CapSense customizer.

Parameters: uint8 widget: Widget number. For every touchpad widget there are defines in this format:

```
#define CapSense_"widget_name"__TP          5
```

Example:

```
#define CapSense_MY_TOUCH1__TP              5
```

All widget names are upper case.

(uint16* pos): pointer to an array of two uint16, where touch postion will be stored:

pos[0] - X position;

pos[1] - Y position.

All widget names are upper case. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint32: 1 if finger is on the touchpad, 0 if not.

Side Effects: None

uint32 CapSense_GetMatrixButtonPos(uint32 widget, uint8* pos)

Description: If a finger is present on matrix buttons, this function calculates the row and column position of the finger. Returns a '1' if a finger is on the matrix buttons. This function is available only if a matrix buttons are defined by the CapSense customizer.

Parameters: uint8 widget: Widget number. For every matrix buttons widget there are defines in this format:

```
#define CapSense_"widget_name"__MB          5
```

Example:

```
#define CapSense_MY_TOUCH1__MB              5
```

All widget names are upper case.

(uint8* pos): pointer to an array of two uint8, where touch postion will be stored:

pos[0] - column position;

pos[1] - row position.

All widget names are upper case. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint8: 1 if finger is on the touchpad, 0 if not.

Side Effects: None

uint32 CapSense_CheckIsSensorActive(uint32 sensor)

Description: Compares the selected Sensor of the CapSense_sensorSignal[ ] array to its finger threshold. Hysteresis and Debounce are taken into account. The Hysteresis value is added or subtracted from the finger threshold based on whether the Sensor is currently active. If the Sensor is active, the threshold is lowered by the hysteresis amount. If it is inactive, the threshold is raised by the hysteresis amount. The Debounce counter added to the Sensor active transition. This function also updates the Sensor's bit in the CapSense_sensorOnMask[ ] array.

Parameters: uint32 – sensor: Scan Sensor Number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint32: Scan Sensor state 1 if active, 0 if inactive

Side Effects: Updates the Sensor's bit in the CapSense_sensorOnMask[ ] array

uint16 CapSense_GetBaselineData(uint32 sensor)

Description: This is a function to read sensor baseline from component.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint16: This API returns baseline value of the sensor indicated by argument.

Side Effects: None

uint16 CapSense_GetDiffCountData(uint32 sensor)

Description: This API returns difference count data.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: unit16 : This API returns difference count value of the sensor indicated by argument.

Side Effects: None

uint16 CapSense_GetNormalizedDiffCountData(uint32 sensor)

Description: This API returns normalized difference count data.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: unit16: This API returns normalized difference count value of the sensor indicated by argument.

Side Effects: None

uint8 CapSense_GetNoiseThreshold(uint32 widget)

Description: This API returns the noise threshold value.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint8: This API returns the noise threshold of the widget indicated by argument.

Side Effects: None

uint8 CapSense_GetNegativeNoiseThreshold(uint32 widget)

Description: This API returns the negative noise threshold value.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint8: This API returns the negative noise threshold of the widget indicated by argument.

Side Effects: None

uint16 CapSense_GetNoiseEnvelope(uint32 sensor)

Description: This API returns the measured noise envelope value. The min value for this API is 1 and it never returns 0 as noise.

This API is available only when SmartSense (Auto-tune) is enabled.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint16: This API shall return the noise envelope value of the sensor indicated by argument.

Side Effects: None

uint8/uint16 CapSense_GetFingerThreshold(uint32 widget)

Description: This API returns finger threshold value.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint8/uint16: This API returns the finger threshold of the widget indicated by argument.

Side Effects: None

uint8 CapSense_GetFingerHysteresis(uint32 widget)

Description: This API returns Hysteresis value.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint8: This API returns the Hysteresis of the widget indicated by argument.

Side Effects: None

void CapSense_WriteSensorRaw(uint32 sensor, uint16 data)

Description: This API has two arguments, sensor number and raw count value. This API writes the raw count value passed as argument to the sensor raw count array.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

uint16 data: Sensor raw count.

Return Value: None

Side Effects: None

void CapSense_SetBaselineData(uint32 sensor, uint16 data)

Description: This API has two arguments, sensor number and baseline value.

This API writes the data value passed as argument to the sensor baseline array.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

uint16 data: Sensor baseline.

Return Value: None

Side Effects: None

void CapSense_SetSensitivity(uint32 sensor, uint32 data)

Description: This API sets the sensitivity value for the sensor. The sensitivity value is used during the auto-tuning algorithm executed as part of CapSense_Start API.

This API is called by application layer prior to calling CapSense_Start API. Calling this API after execution of CapSense_Start API has no effect.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

uint32 data: Sensitivity of the sensor. Possible values include:

1 – 0.1pF sensitivity

2 – 0.2pF sensitivity

3 – 0.3pFsensitivity

4 – 0.4pF sensitivity

5 – 0.5pF sensitivity

6 – 0.6pF sensitivity

7 – 0.7pFsensitivity

8 – 0.8pF sensitivity

9 – 0.9pF sensitivity

10 – 1.0pF sensitivity

All other values, set sensitivity to 1.0pF.

Return Value: None

Side Effects: None

uint32 CapSense_GetSensitivityCoefficient(uint32 sensor)

Description: This API returns the K coefficient for the appropriate sensor.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint32: K value for the appropriate sensor

Side Effects: None

void Capsense_SetDebounce(uint32 widget, uint8 value)

Description: This API sets the debounce value. This API affects all the sensors in the widget.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

uint8 value: Debounce value.

Return Value: None

Side Effects: None

uint8 Capsense_GetDebounce(uint32 widget)

Description: This API returns the debounce value.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint8: returns the debounce value.

Side Effects: None

void CapSense_SetFingerHysteresis(uint32 widget, uint8 value)

Description: This API sets the hysteresis value sensors. This API affects all the sensors in the widget.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

uint8 value: hysteresis value.

Return Value: None

Side Effects: None

void CapSense_SetNoiseThreshold(uint32 widget, uint8 value)

Description: This API sets the Noise Threshold value for all sensors in the widget.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

uint8 value: Noise Threshold value.

Return Value: None

Side Effects: None

void CapSense_SetNegativeNoiseThreshold(uint32 widget, uint8 value)

Description: This API sets the Negative Noise Threshold value for a widget. This API affects all the sensors in the widget.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

uint8 value: Negative Noise Threshold value.

Return Value: None

Side Effects: None

void CapSense_SetLowBaselineReset(uint32 sensor, uint8 value)

Description: This API sets the low baseline reset threshold value a sensor.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

uint8 value: low baseline reset threshold value.

Return Value: None

Side Effects: None

uint8 CapSense_GetLowBaselineReset(uint32 sensor)

Description: This API returns the low baseline reset threshold value a sensor.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint8: return low baseline reset threshold value.

Side Effects: None

void CapSense_SetFingerThreshold(uint32 widget, uint8/16 value)

Description: This API sets the finger threshold value for a Widget.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

uint8/16 value: Finger threshold value for the Widget.

Return Value: None

Side Effects: None

void CapSense_SetDiffCountData(uint32 sensor, uint16/uint8 value)

Description: This API sets difference counts data for each sensor.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

uint16/uint8 value: difference counts data.

Return Value: None

Side Effects: None

uint32 CapSense_GetWidgetNumber(uint32 sensor)

Description: This API returns the widget number for the sensor.

Parameters: uint32 sensor: Sensor number. The value of Sensor number can be from 0 to N. The value N can be 0 to total number of sensor-1. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint32: returns the widget number of sensor. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Side Effects: None

void CapSense_UpdateThresholds(uint32 sensor)

Description: This API calculates the threshold parameters for the given sensor and updates the parameter to the respective arrays/variables that store threshold parameter for each sensor when SmartSense is enabled. There are two possible methods to calculate the threshold values as mentioned below

When automatic threshold is enabled, this API shall calculate the threshold parameters based on measured noise envelope of the sensor. In this mode, API shall calculate finger threshold for the given sensor along with other thresholds.

When automatic threshold is disabled, this API shall not calculate the finger threshold. The finger threshold shall be set by the application firmware. All other thresholds shall be calculated by this API based on the finger threshold value set by the caller. In this mode, the API expects caller to set appropriate finger threshold values prior to calling this API.

This API is applicable for all types of sensors.

Parameters: uint32 sensor: Sensor number. The value of Sensor number can be from 0 to N. The value N can be 0 to total number of sensor-1. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: None

Side Effects: None

void CapSense_UpdateBaselineNoThreshold(uint32 sensor)

Description: This API updates the baseline of the given sensor. This API does not calculate or modify the threshold parameter associated with given sensor.

Parameters: uint32 sensor: Sensor number. The value of Sensor number can be from 0 to N. The value N can be 0 to total number of sensor-1. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: None

Side Effects: Sensor baseline variable is updated.

void CapSense_SetIDACRange(uint32 iDacRange)

Description: Sets the IDAC range to 4x (1.2uA/bit) or 8x (2.4uA/bit) mode. The IDAC range is common for all sensors and common for modulation and compensation IDACs.

Parameters: uint32 iDacRange: represents value for IDAC range

0 - IDAC range set to 4x (1.2uA/bit)

1 or >1 - IDAC range set to 8x (2.4uA/bit)

Return Value: None

Side Effects: None

uint32 CapSense_GetIDACRange(void)

Description: Returns value that indicates the IDAC range used by the component to scan sensors. The IDAC range is common for all sensors.

Parameters: None

Return Value: uint32 iDacRange: represents value for IDAC range

0 - IDAC range set to 4x (1.2uA/bit)

1 or >1 - IDAC range set to 8x (2.4uA/bit)

Side Effects: None

void CapSense_SetModulationIDAC(uint32 sensor, uint32 modIdacValue)

Description: Sets value for modulation IDAC for a sensor.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

uint32 modIdacValue: represents the modulation IDAC data register value.

Return Value: None

Side Effects: None

uint32 CapSense_GetModulationIDAC(uint32 sensor)

Description: Returns value of modulation IDAC for a sensor.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint32 returns the modulation IDAC data register value.

Side Effects: None

void CapSense_SetCompensationIDAC(uint32 sensor, uint32 compIdacValue)

Description: Sets value of compensation IDAC for a sensor.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

uint32 compIdacValue: represents the compensation IDAC data register value.

Return Value: None

Side Effects: None

uint32 CapSense_GetCompensationIDAC(uint32 sensor)

Description: Returns value of compensation IDAC for a sensor.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint32: returns the compensation IDAC data register value.

Side Effects: None

void CapSense_SetSenseClkDivider(uint32 sensor, uint32 senseClk)

Description: Sets value of sense clock divider for a sensor.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

uint32 senseClk: represents the sense clock value.

Note In PSoC 4100/PSoC 4200 devices, the Sense Clock also depends on the Modulator Clock divider because these dividers are chained. This means that the Sense Clock divider input connects to the Modulator Clock output. The Sense Clock divider value should take into account the Modulator Clock Divider value. The customizer adjusts the Sense Clock value automatically to take into account the modulator output clock. For example, if you set the Modulator Clock divider to 8 and the Sense Clock divider to 8, the CapSense_GetSenseClkDivider API returns 1. Refer to the CapSense Clocking section for chained clocks' details.

Return Value: None

Side Effects: None

uint32 CapSense_GetSenseClkDivider(uint32 sensor)

Description: Returns value of sense clock divider for a sensor.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint32: returns sense clock divider for a sensor.

Note In PSoC 4100/PSoC 4200 devices, the Sense Clock also depends on the Modulator Clock divider because these dividers are chained. This means that the Sense Clock divider input connects to the Modulator Clock output. The Sense Clock divider value should take into account the Modulator Clock Divider value. The customizer adjusts the Sense Clock value automatically to take into account the modulator output clock. For example, if you set the Modulator Clock divider to 8 and the Sense Clock divider to 8, the CapSense_GetSenseClkDivider API returns 1. Refer to the CapSense Clocking section for chained clocks' details.

Side Effects: None

void CapSense_SetModulatorClkDivider(uint32 sensor, uint32 modulatorClk)

Description: Sets value of modulator sample clock divider for a sensor.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

uint32 – modulatorClk: represents the modulator sample clock value.

Return Value: None

Side Effects: None

uint32 CapSense_GetModulatorClkDivider(uint32 sensor)

Description: Returns value of modulator sample clock divider for a sensor.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint32: returns modulator sample clock divider for a sensor.

Side Effects: None

void CapSense_SetScanResolution(uint32 widget, uint32 resolution)

Description: Sets value of sensor scan resolution for a widget.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

uint32 resolution: represents the resolution value. The following defines available in the *CapSense.h* file should be used:

- CapSense_RESOLUTION_6_BITS
- CapSense_RESOLUTION_7_BITS
- CapSense_RESOLUTION_8_BITS
- CapSense_RESOLUTION_9_BITS
- CapSense_RESOLUTION_10_BITS
- CapSense_RESOLUTION_11_BITS
- CapSense_RESOLUTION_12_BITS
- CapSense_RESOLUTION_13_BITS
- CapSense_RESOLUTION_14_BITS
- CapSense_RESOLUTION_15_BITS
- CapSense_RESOLUTION_16_BITS

Return Value: None

Side Effects: None

uint32 CapSense_GetScanResolution(uint32 widget)

Description: Return value of resolution for a widget.

Parameters: uint32 widget: Widget number. The *Capsense_CSHL.h* file contains defines for the widget numbers. See the Widget Constants section for details.

Return Value: uint32: returns resolution for a widget. The return value corresponds to the defines available in the *CapSense.h* file:

- CapSense_RESOLUTION_6_BITS
- CapSense_RESOLUTION_7_BITS
- CapSense_RESOLUTION_8_BITS
- CapSense_RESOLUTION_9_BITS
- CapSense_RESOLUTION_10_BITS
- CapSense_RESOLUTION_11_BITS
- CapSense_RESOLUTION_12_BITS
- CapSense_RESOLUTION_13_BITS
- CapSense_RESOLUTION_14_BITS
- CapSense_RESOLUTION_15_BITS
- CapSense_RESOLUTION_16_BITS

Side Effects: None

void CapSense_SetDriveModeAllPins(uint32 driveMode)

Description: This API sets the drive mode of port pins used by CapSense component (sensors, guard, shield, shield tank and Cmod) to drive mode specified by the argument.

Parameters: uint32 driveMode: parameter that indicates the drive mode.

Values:

- CY_SYS_PINS_DM_ALG_HIZ - High Impedance Analog
- CY_SYS_PINS_DM_DIG_HIZ - High Impedance Digital
- CY_SYS_PINS_DM_RES_UP - Resistive Pull Up
- CY_SYS_PINS_DM_RES_DWN - Resistive Pull Down
- CY_SYS_PINS_DM_OD_LO - Open Drain, Drives Low
- CY_SYS_PINS_DM_OD_HI - Open Drain, Drives High
- CY_SYS_PINS_DM_STRONG - Strong Drive
- CY_SYS_PINS_DM_RES_UPDWN - Resistive Pull Up/Down

Return Value: None

Side Effects: This API shall be called only after CapSense component is stopped.

void CapSense_RestoreDriveModeAllPins(void)

Description: This API restores the drive for all CapSense port pins to original state. This APIs is compliment of CapSense_SetDriveModeAllPins API.

Parameters: None

Return Value: None

Side Effects: None

void CapSense_SetUnscannedSensorState(uint32 sensor, uint32 sensorState)

Description: This API sets the state for un-scanned sensors. It is possible to set state to Ground, High-Z or shield electrode. The un-scanned sensor can be connected to shield electrode only if shield is enabled. If case of shield is disabled and this API is called with parameter indicating shield state, the un-scanned sensor shall be connected to Ground.

Parameters: uint32 sensor: this parameter indicates the Sensor ID. The Capsense.h file contains defines for the sensor numbers. See the Sensor Constants section for details.

uint32 sensorState: this parameter indicates un-scanned sensor state.

Return Value: None

Side Effects: This API shall be called only after CapSense component is stopped.

void CapSense_UpdateWidgetBaseline (uint32 widget)

Description: The sensor's baseline is a historical count value, calculated independently for each sensor in the widget. It updates the CapSense_sensorBaseline[sensor] array element using a low-pass filter with k = 256. The function calculates the difference count by subtracting the previous baseline from the current raw count value and stores it in CapSense_sensorSignal[sensor] for sensor numbers that belong to the widget.

If the auto reset option is enabled, the baseline updates independent of the noise threshold.

If the auto reset option is disabled, the baseline stops updating if the signal is greater than the noise threshold and resets the baseline when the signal is less than the minus noise threshold.

Parameters: uint32 widget: widget number

Return Value: None

Side Effects: Updates the CapSense_sensorBaseline[ ] array.

void CapSense_EnableRawDataFilters(void)

Description: This API enables the rawdata filters for the sensor signals.

Parameters: None

Return Value: None

Side Effects: None

void CapSense_DisableRawDataFilters(void)

Description: This API disables the rawdata filters for the sensor signals.

Parameters: None

Return Value: None

Side Effects: None

Tuner Helper APIs

These API functions are used to work with the Tuner GUI.

Function	Description
CapSense_TunerStart()	Initializes CapSense CSD and internal communication components, initializes baselines and starts the sensor scanning loop.
CapSense_TunerComm()	Execute communication between the Tuner GUI.

void CapSense_TunerStart(void)

Description: Initializes CapSense CSD and internal communication components.

All widgets are enabled by default except proximity widgets. Proximity widgets must be manually enabled as their long scan time is incompatible with the fast response required of other widget types.

Parameters: None

Return Value: None

Side Effects: Global interrupts (CyGlobalIntEnable;) must be enabled before CapSense_TunerStart() if the Auto (Smartsense) Tuning method or Auto-calibration is selected.

void CapSense_TunerComm(void)

Description: Executes communication functions with Tuner GUI.

- Manual mode: Transfers sensor scanning and widget processing results to the Tuner GUI from the CapSense CSD component. Reads new parameters from Tuner GUI and apply them to the CapSense CSD component.
- Auto (SmartSense): Executes communication functions with Tuner GUI. Transfer sensor scanning and widget processing results to Tuner GUI. The auto tuning parameters also transfer to Tuner GUI. Tuner GUI parameters are not transferred back to the CapSense CSD component.

This function is blocking and waits while the Tuner GUI modifies CapSense CSD component buffers to allow new data.

Parameters: None

Return Value: None

Side Effects: This API does not allow the code to proceed and will not return until a successful connection has been made with the Tuner GUI.

Built-in Self Test APIs

These API functions are used to check the correct Hardware Setup such as Cmod, parasitic capacitance, shield electrode and external shield tank capacitor capacitance.

Function	Description
CapSense_GetSensorCp()	Returns the parasitic capacitance of sensor.
CapSense_MeasureCmod()	Measures the CMOD external capacitor value in pF.
CapSense_MeasureCShield()	Measures the capacitance value of shield electrode.
CapSense_MeasureCShieldTank()	Measures the capacitance value of external shield tank capacitor.

uint32 CapSense_GetSensorCp(uint32 sensor)

Description: This API returns the Cp (parasitic capacitance) of sensor in pF (pico farads).

The supported range is 5-100 pF. The precision of API is +/-10%.

Note This API can be used to measure a capacitance of the external capacitors. In this case the parasitic capacitance should be subtracted from the total result.

Parameters: uint32 sensor: Sensor number. The *Capsense.h* file contains defines for the sensor numbers. See the Sensor Constants section for details.

Return Value: uint32: This API returns Sensor parasitic capacitance (Cp) of the sensor indicated as argument. The unit of sensor Cp value is pico-farads.

Side Effects: Sensor scan should be complete before using this API.

uint32 CapSense_MeasureCmod(void)

Description: This API measures the CMOD external capacitor value in pF.

Parameters: None

Return Value: uint32: returns measured CMOD in pico-farads.

Side Effects: Component should be stopped before calling this API.

uint32 CapSense_MeasureCShield(void)

Description: This API implements method to measure the capacitance value of shield electrode. When this APIs is called, it returns the shield electrode capacitance in pico-farads.

Parameters: None

Return Value: uint32: returns measured capacitance of shield electrode in pico-farads.

Side Effects: None.

uint32 CapSense_MeasureCShieldTank(void)

Description: This API implements method to measure the capacitance value of external shield tank capacitor. When this APIs is called, it returns the shield tank capacitance in pico-farads.

Parameters: None

Return Value: uint32: returns measured capacitance of shield tank capacitor in pico-farads.

Side Effects: Component should be stopped before calling this API.

Data Structures

The API functions use several global arrays for processing sensor and widget data. You should not alter these arrays manually. These values can be viewed for debugging and tuning purposes. For example, you can use a charting tool to display the contents of the arrays. The global arrays are:

Array	Description
CapSense_sensorRaw[]	This array contains the raw data for each sensor. The array size is equal to the total number of sensors (CapSense_TOTAL_SENSOR_COUNT). The CapSense_sensorRaw [ ] data is updated by these functions: • CapSense_ScanSensor() • CapSense_ScanEnabledWidgets() • CapSense_InitializeSensorBaseline() • CapSense_InitializeAllBaselines() • CapSense_UpdateEnabledBaselines()

Array	Description
CapSense_sensorEnableMask[]	This is a byte array that holds the sensor scanning state CapSense_sensorEnableMask [0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). CapSense_sensorEnableMask[1] contains the masked bits for sensors 8 through 15 (if needed), and so on. This byte array holds as many elements as are necessary to contain the total number of sensors. The value of a bit specifies if a sensor is scanned by the CapSense_ScanEnabledWidgets() function call: 1 – sensor is scanned , 0 – sensor is not scanned. The CapSense_sensorEnableMask[ ] data is changed by functions: • CapSense_EnabledWidget() • CapSense_DisableWidget() • The CapSense_sensorEnableMask[ ] data is used by function: • CapSense_ScanEnabledWidgets()
CapSense_portTable[] and CapSense_maskTable[]	These arrays contain port and pin masks for every sensor to specify what pin the sensor is connected to. • Port – Defines the port number that pin belongs to. • Mask – Defines pin number within the port.
CapSense_sensorBaselineLow[]	This array holds the fractional byte of baseline data of each sensor used in the low pass filter for baseline update. The array's size is equal to the total number of sensors. The CapSense_sensorBaselineLow[ ] array is updated by these functions: • CapSense_InitializeSensorBaseline() • CapSense_InitializeAllBaselines() • CapSense_UpdateSensorBaseline() • CapSense_UpdateEnabledBaselines()
CapSense_sensorBaseline[]	This array holds the baseline data of each sensor. The array's size is equal to the total number of sensors. The CapSense_sensorBaseline[ ] array is updated by these functions: • CapSense_InitializeSensorBaseline() • CapSense_InitializeAllBaselines() • CapSense_UpdateSensorBaseline() • CapSense_UpdateEnabledBaselines().
CapSense_sensorSignal[]	This array holds the sensor signal count computed by subtracting the previous baseline from the current raw count of each sensor. The array size is equal to the total number of sensors. The Widget Resolution parameter defines the resolution of this array as 1 byte or 2 bytes. The CapSense_sensorSignal[ ] array is updated by these functions: • CapSense_InitializeSensorBaseline() • CapSense_InitializeAllBaselines() • CapSense_UpdateSensorBaseline() • CapSense_UpdateEnabledBaselines().

Array	Description
CapSense_sensorOnMask[]	This is a uint8 array that holds the sensor 'on' or 'off' state (for buttons, matrix buttons or sliders). CapSense_sensorOnMask[0] contains the masked bits for sensor 0 through 7 (sensor 0 is bit 0, sensor1 is bit 1). CapSense_sensorOnMask[1] contains the masked bits for sensor 8 through 15 (if they are needed), and so on. This uint8 array contains as many elements as are necessary to contain all placed sensor. The value of a bit is 1 if the sensor is on and 0 if the sensor is off.
CapSense_ModulatorIDAC[]	This array contains an 8-bit IDAC value for every sensor. The array size is equal to the total number of sensors.
CapSense_CompensationIDAC[]	This array contains a 7-bit IDAC value for every sensor. The array size is equal to the total number of sensors.
CapSense_senseClkDividerVal[]	This array contains the Sense Clock dividers for every sensor. An array is generated only if the Individual frequency settings are enabled in the Customizer.
CapSense_sampleClkDividerVal[]	This array contains the Modulator Clock dividers for every sensor. An array is generated only if the Individual frequency settings are enabled in the Customizer.
CapSense_rawFilterData1[]	This array is used to store previous samples of any enabled raw data filter. The CapSense_rawFilterData1[ ] data is updated by this function: • CapSense_UpdateSensorBaseline()
CapSense_rawFilterData2[]	This array is used to store previous samples of enabled raw data filter. It is required only for median or average filters (these filters also use CapSense_rawFilterData1 array to store previous samples). The CapSense_rawFilterData2[ ] data is updated by this function: • CapSense_UpdateSensorBaseline()
CapSense_lowBaselineResetCnt[]	The elements of this array are used as the counter to decide if baseline reset should be done for each of the scanned sensors. The counter increments if the difference signal is negative and above the CapSense_NEGATIVE_NOISE_THRESHOLD. When the counter reaches the CapSense_LOW_BASELINE_RESET value, the baseline for that sensor will be re-initialized and counter set to zero. The CapSense_lowBaselineResetCnt[ ] data is updated by this function: • CapSense_UpdateSensorBaseline()
CapSense_fingerThreshold[]	This array contains the level of signal for each sensor that determines if a finger is present on the sensor.
CapSense_noiseThreshold[]	This array contains the level of signal for each sensor that determines the level of noise in the capacitive scan. Noise below the threshold is used to update the sensors baseline. Noise above the threshold is not used to update the baseline.
CapSense_hysteresis[]	This array contains hysteresis values for each widget. The CapSense_debounceCounter[] data is updated by this function: • CapSense_CalculateThresholds()

Array	Description
CapSense_debounce[]	This array holds the debounce value for each Widget's debounce feature. The value is set for widgets that have this parameter. These widgets are buttons, matrix buttons, proximity, and guard sensor. All other widgets do not have a debounce parameter and use the last element of this array with value 0 (0 means no debounce). The CapSense_debounce[] array is used for initialization of the CapSense_debounceCounter[] array.
CapSense_debounceCounter[]	This array holds the current debounce counter of a sensor. The counter is decremented if the sensor is active (sensor signal is above the finger threshold plus hysteresis). When it reaches 1, the sensor ON mask (CapSense_sensorOnMask) will be set and the counter value reset to the default value from CapSense_debounce[] array. The same occurs when the sensor goes inactive (touch release) and the sensor signal is below the finger threshold minus hysteresis. This functionality is implemented in CapSense_CheckIsSensorActive() function. The CapSense_debounceCounter[] data is updated by these functions: • CapSense_BaseInit() • CapSense_CheckIsSensorActive()

Constants

The following constants are defined. Some of the constants are defined conditionally and will only be present if needed for the current configuration.

- CapSense_TOTAL_SENSOR_COUNT – Defines the total number of sensors within the CapSense CSD component.

Sensor Constants

A constant is provided for each sensor. Any function that takes sensor as an argument can use the constants. For example, these APIs take sensor as an argument:

ScanSensor(), ReadSensorRaw(), CheckIsSensorActive(), InitializeSensorBaseline(), UpdateSensorBaseline(), GetBaselineData(), GetDiffCountData(), GetNormalizedDiffCountData(), GetNoiseEnvelope(), WriteSensorRaw(), SetBaselineData(), SetSensitivity(), GetSensitivityCoefficient(), SetLowBaselineReset(), GetLowBaselineReset().

The constant names consist of:

Instance name + "_SENSOR" + Widget Name + element + "#element number" + "__" + Widget Type

These constants are contained in the generated code (Capsense.h). The names are forced to upper case.

For example:

```
/* Define Sensors */
#define CapSense_SENSOR_TP1_ROW0__TP    0
#define CapSense_SENSOR_TP1_ROW1__TP    1
#define CapSense_SENSOR_TP1_COL0__TP    2
#define CapSense_SENSOR_TP1_COL0__TP    3
#define CapSense_SENSOR_LS0_E0__LS      5
#define CapSense_SENSOR_LS0_E1__LS      6
#define CapSense_SENSOR_PROX1__PROX     7
```

- Widget Name – The user-defined name of the widget (must be a valid C style identifier). The widget name must be unique within the CapSense CSD component. All Widget Names are upper case.
- Element Number – The element number only exists for widgets that have multiple elements, such as radial sliders. For touchpads and matrix buttons, the element number consists of the word ‘Col’ or ‘Row’ and its number (for example: Col0, Col1, Row0, Row1). For linear and radial sliders, the element number consists of the character ‘e’ and its number (for example: e0, e1, e2, e3).
- Widget Type – There are several widget types:

Alias	Description
BTN	Buttons
LS	Linear Sliders
RS	Radial Sliders
TP	Touchpads and Trackpad
MB	Matrix Buttons
PROX	Proximity Sensors
GEN	Generic Sensors
GRD	Guard Sensor

Widget Constants

A constant is provided for each widget. Any function that takes widget as an argument can use the constants. For example, these APIs take widget as an argument:

CapSense_CheckIsWidgetActive(), CapSense_EnableWidget(), CapSense_DisableWidget(), CapSense_GetCentroidPos(), CapSense_GetRadialCentroidPos(), CapSense_GetTouchCentroidPos(), ScanWidget(), GetMatrixButtonPos(),

GetNoiseThreshold(), GetNegativeNoiseThreshold(), GetFingerThreshold(), GetFingerHysteresis(), SetDebounce(), GetDebounce(), SetFingerHysteresis(), SetNoiseThreshold(), SetNegativeNoiseThreshold(), SetFingerThreshold().

The constants consist of:

Instance name + Widget Name + Widget Type

These constants are contained in the generated code (Capsense_CSHL.h). The names are forced to upper case.

For example:

```
/* Widgets constants definition */
#define CapSense_UP__BTN        0
#define CapSense_DOWN__BTN      1
#define CapSense_VOLUME__SL     2
#define CapSense_TOUCHPAD__TP   3
```

Sample Firmware Source Code

PSoC Creator provides numerous example projects that include schematics and example code in the Find Example Project dialog (**File > Example Project…**). For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable only for this component

This section provides information on component-specific deviations. Project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

The CapSense CSD component has the following specific deviation:

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Justification of Violation(s)
8.8	R	An external object or function shall be declared in one and only one file.	Some arrays are generated based on the component configuration and these arrays are declared locally in the .c source files where they are used instead of in .h include files.

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Justification of Violation(s)
11.4	A	A cast should not be performed between a pointer to object type and a different pointer to object type.	In the component tuner helper, pointers to component structures are cast to 8-bit data pointers and then passed to an I2C API for transmission. The I2C component only transmits streams of bytes, so this cast is required.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	The component has several functions that take pointer arguments. The arguments are intended to be passed arrays of data and they are accessed using array indexing.
19.7	A	A function should be used in preference to a function-like macro.	Function-like macros are used to improve performance.

API Memory Usage

The component memory usage varies significantly, depending on the compiler, device, number of APIs used, and component configuration. This table shows the memory use for all APIs available in the given component configuration.

The measurements were done with an associated compiler configured in release mode with optimization set for size. For a specific design, the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 4000		PSoC 4100/ PSoC 4200		PSoC 4100M/ PSoC 4200M	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Widgets: 5-buttons Tuning method: Manual Compensation IDAC: Enabled Auto Calibration: Enabled Raw data filter: First Order IIR ¼ BIST: Enabled Precharging mode: PRS_Auto Individual frequency settings: Enabled Widget Resolution: 8-bit	3980	107	3944	107	4036	107

Configuration	PSoC 4000		PSoC 4100/ PSoC 4200		PSoC 4100M/ PSoC 4200M	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Widgets: 5-segment linear slider Tuning method: Manual Compensation IDAC: Enabled Auto Calibration: Enabled Raw data filter: First Order IIR ¼ Position noise filter: First Order IIR ¼ BIST: Enabled Precharging mode: PRS_Auto Individual frequency settings: Enabled Widget Resolution: 8-bit	4082	108	3982	108	4134	108
Widgets: 5-buttons, 5-segment linear slider Tuning method: Manual Compensation IDAC: Enabled Auto Calibration: Enabled Raw data filter: First Order IIR ¼ Position noise filter: Jitter BIST: Enabled Precharging mode: PRS_Auto Individual frequency settings: Enabled Widget Resolution: 8-bit	4568	201	4442	201	4637	201
Widgets: 4x4 Matrix Button Tuning method: Manual Compensation IDAC: Enabled Auto Calibration: Enabled Raw data filter: First Order IIR ¼ BIST: Enabled Precharging mode: PRS_Auto Individual frequency settings: Enabled Widget Resolution: 8-bit	4012	164	3952	164	4068	164

Configuration	PSoC 4000		PSoC 4100/ PSoC 4200		PSoC 4100M/ PSoC 4200M	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Widgets: 8x8 Touchpad Tuning method: Manual Compensation IDAC: Enabled Auto Calibration: Enabled Raw data filter: First Order IIR ¼ Position noise filter: First Order IIR ¼ BIST: Enabled Precharging mode: PRS_Auto Individual frequency settings: Enabled Widget Resolution: 8-bit	4622	299	4490	299	4682	299

Pin Assignments

The CapSense customizer generates a pin alias name for each of the CapSense sensors and support signals. These aliases are used to assign sensors and signals to physical pins on the device. Assign CapSense CSD component sensors and signals to pins in the Pin Editor tab of the Design Wide Resources file view.

Sensor Pins

Aliases are provided to associate sensor names with widget types and widget names in the CapSense customizer.

The aliases for sensors are:

Widget Name + Element Number + "__" + Widget Type

Cmod Pin

One side of the external modulator capacitor (C_{MOD}) should be connected to a physical pin and the other to GND. In PSoC 4100/PSoC 4200 devices, the C_{MOD} can be connected to P4[2] pin. In PSoC 4000 devices, the C_{MOD} can be connected to P0[4] pin. In PSoC 4100M/PSoC 4200M devices, the C_{MOD} can be connected to P4[2] or P5[0] pin.

Recommended C_{MOD} value is 2.2 nF.

Shield Pin

Shield alias can be assigned to any available pin.

Cshield_tank Pin

In PSoC 4100/PSoC 4200 devices, the Cshield_tank can be connected to P4[3] pin. In PSoC 4000 devices, the Cshield_tank can be connected to P0[2] pin. In PSoC 4100M/ PSoC 4200M devices, the Cshield_tank can be connected to 4[3] or 5[1] pin.

Functional Description

Definitions

Sensor

A sensor is a conductive element on a substrate whose capacitance increase with a touch; the conductive element is connected to one pin of PSoC.

Examples of sensors include: Copper pad on PCB connected to PSoC, Copper or silver on Flex PCB connected to PSoC, Silver ink on PET connected to PSoC, ITO on glass connected to PSoC.

CapSense Widget

A CapSense widget is one sensor or group of sensors which has similar properties used to construct functionality.

Some examples of CapSense Widgets include button widget or proximity widgets which usually has only one sensor to detect touch or no-touch status. Linear slider, radial slider, touchpads and matrix buttons widgets are examples for widget constructed by group of sensors which has similar properties.

Scan Time

Scan time is a period of time that the CapSense component is scanning one capacitive sensor.

In **Manual Mode**, the Sensor Scan Time depends on resolution and modulator clock:

Scan Time (ms) = (2^N-1)*ModDiv / clockInKHz,

where:

- N – resolution
- ModDiv – Modulator Clock Divider
- clockInKHz – HFCLK clock in KHz

Note Values shown here may differ from those estimated by the customizer scan time because of the approximation of the setup and preprocessing time made by the customizer.

In **Auto (Smartsense) Tuning Mode**, the Sensor Scan Time depends on Parasitic Capacitance (Cp) and Sensitivity.

The following table shows Scanning Time in µs versus Sensitivity and Parasitic Capacitance for HFCLK = 24 MHz.

Parasitic Capacitance, pF	Sensitivity			
	1	2	3	4
10	410	237	237	153
15	750	410	237	237
20	750	410	410	237
25	2800	1440	750	750
30	2800	1440	750	750
35	2800	1440	750	750
40	2800	1440	1440	750
45	2800	1440	1440	750
50	5600	2800	1440	1440

The following table shows Resolution versus Sensitivity and Parasitic Capacitance for HFCLK = 24 MHz.

Parasitic Capacitance, pF	Sensitivity			
	1	2	3	4
10	12	11	11	10
15	13	12	11	11

Parasitic Capacitance, pF	Sensitivity			
	1	2	3	4
20	13	12	12	11
25	14	13	12	12
30	14	13	12	12
35	14	13	12	12
40	14	13	13	12
45	14	13	13	12
50	15	14	13	13

Note Scan time is an estimate based on the following settings: CPU Clock = 24 MHz, number of channels = 1. The Scanning time was measured as the time interval of one sensor scan. This time includes sensor setup time, sample conversion interval, and data processing time. These values can be used to estimate scanning speed for other clock rates and additional sensors by scaling the provided values linearly.

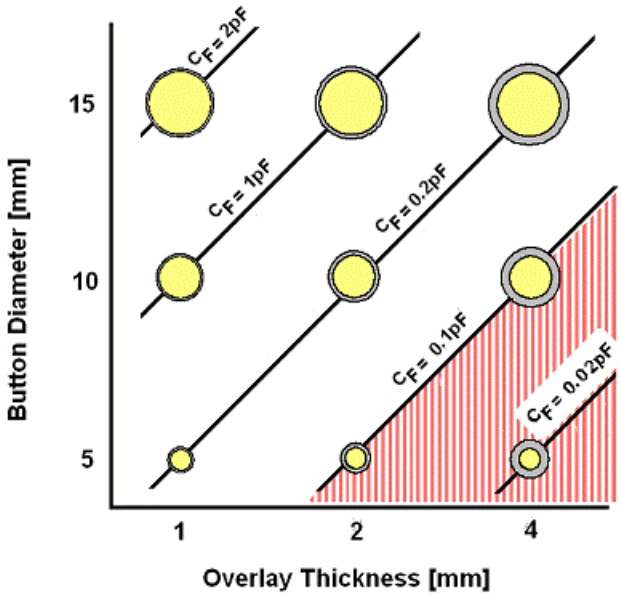
Scan Resolution

This parameter defines maximum raw count (full scale range) for scanning which equals to 2^N-1, where N - scanning resolution. Raising the resolution raises sensitivity, SNR, and noise immunity at the expense of scan time.

Table below provides recommended Scan Resolution settings based on Cp and the finger capacitance Cf. Cf is the change in capacitance of a sensor when a finger is placed on the sensor. Cf depends on overlay thickness, sensor size, and proximity of the sensor to other large conductors.

Cp (pF)	Cf = 0.1pF	Cf = 0.2pF	Cf = 0.4pF	Cf = 0.8pF
<6	12	11	10	9
7-12	13	12	11	10
13-24	14	13	12	11
25-48	15	14	13	12
>49	16	15	14	13

The following figure provides Cf values as a function of overlay thickness and circular sensor diameter.



Sensor Scan Slot

A sensor scan slot is a period of time that the CapSense module is scanning one or more combined capacitive sensors. Multiple sensors can be combined in a given scan slot to enable features such as ganged proximity sensing. This means that a proximity sensor can be a complex sensor that can be configured in the **Scan Order** tab by selecting certain other sensors. These sensors will be a part of the complex proximity sensor and will have the common parameters when this complex sensor is being scanned.

To reduce term confusion, a sensor scan slot only refers to the period of time a sensor is scanned, not to the sensor itself.

The Complex sensors section describes how to configure the complex sensor.

Raw Count

The CapSense component measures the capacitance of the sensor and provides the result in a digital form called Raw Count. The value of Raw Count increases as sensor capacitance increases.

Baseline

The raw count values of a sensor vary gradually due to changes in the environment such as temperature and humidity. These gradual variations are compensated for with the baseline values. The baseline keeps track of gradual changes in raw count using a software algorithm. It is a low-pass filter that is less sensitive to sudden changes in the raw count. The baseline values provide the reference level for computing the difference counts.

Difference Count

The difference count is the difference between the raw count and the baseline of the sensor. Usually, the difference count is zero when the sensor is untouched. When the sensor is touched, it causes the raw count to increase, and results in a difference count value.

Sensor State

The state of a sensor is represented as 1 if the button is ON (touched) and 0 if the button is OFF (untouched). The ON state is a.k.a active state and OFF state is a.k.a inactive state.

Finger Threshold

This value is used to determine if a finger is present on the sensor. The CapSense component uses the Finger Threshold parameter to judge the active/inactive state of a sensor. If the Difference Count value of a sensor is greater than the Finger Threshold value, the sensor is judged as active.

Note This definition assumes that the hysteresis level is set to 0 and Debounce is set to 1.

Hysteresis

The Hysteresis parameter is used in conjunction with the finger threshold to determine sensor state. The touch state turns ON once the difference count is higher than the Finger threshold + Hysteresis. The touch state stays on until the difference counts is reduces below Finger threshold - Hysteresis.

This prevents the touch / no touch state machine from reporting ON and OFF due to noise when the difference counts very close to Finger Thershold.

Debounce

Debounce parameter adds a counter to the sensor transition from OFF to ON. For the sensor to transition from OFF to ON, the difference count value must stay above the finger threshold + hysteresis level for the number of samples specified as Debounce.

Noise Threshold

For individual sensors, the Noise Threshold parameter sets the upper raw count limit for updating the baseline value. For slider sensors, it sets the lower limit for difference count to be considered for centroid calculation.

Negative Noise Threshold

The Negative Noise Threshold parameter acts as a negative difference count threshold. If the raw count is below the baseline minus the negative noise threshold for the number of samples specified by the Low Baseline Reset parameter, the baseline is reset to the current raw count value.

Low Baseline Reset

The Low Baseline Reset parameter works together with the Negative Noise Threshold parameter. It counts the number of abnormally low samples required to reset the baseline. It is used to correct the finger-on-at-startup condition.

Sensors Autoreset

This parameter determines whether the baseline is updated at all times, or only when the difference counts are below the noise threshold.

When Sensors Autoreset is enabled, the baseline is updated all the times. These limits the maximum time duration of the sensor can report an ON state when sensor is touched continuously for long time (typical values are 5 to 10 seconds), but prevents the sensors from permanently reporting ON state when the raw count accidentally rises without anything touching the sensor. This sudden rise can be caused by an electrical damage in the system, unacceptable operation like metal object accidently falled on front panel etc.

When Sensors Autoreset is disabled, the baseline is updated only when the difference counts are below the noise. This makes sensor to report ON state as long as sensor is touched.

Parasitic Capacitance (C_p)

The parasitic capacitance is the residual capacitance of sensor. It is the capacitance of sensor measured without a finger touch on the sensor

The parasitic capacitance of a sensor influenced by various things such as: PCB layout, dielectric constand of PCB material, PCB thickness, overlay material and overlay thickness etc. Environmental conditions such as temperature may also impact dielectric constant of PCB material which will indirectly affect the sensor parasitic capacitance.

Finger Capacitance (C_f)

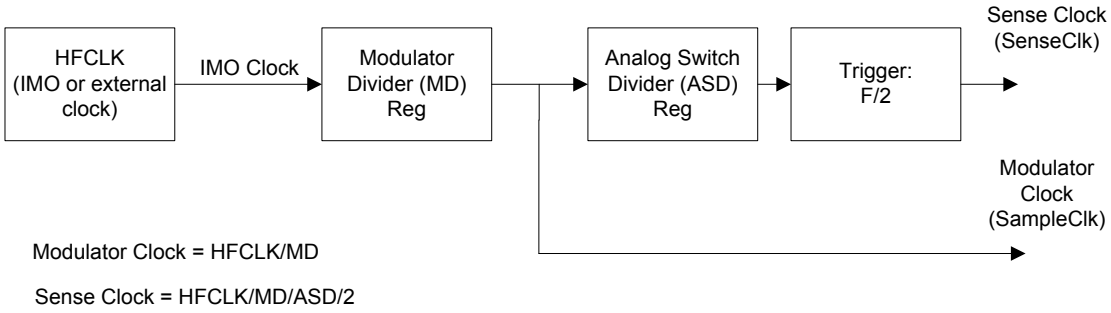
The finger capacitance is the capacitance attributed to the addition of the finger to the sensor.

CapSense Clocking

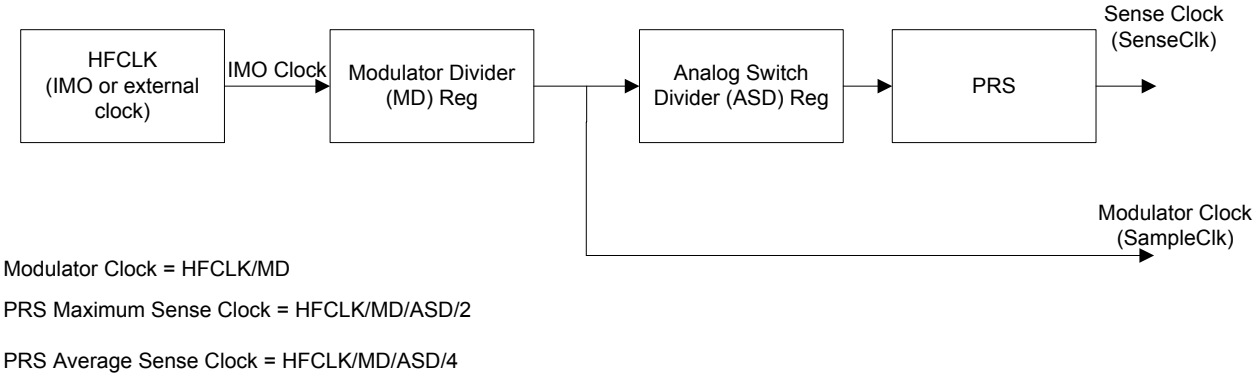
PSoC 4100/PSoC 4200

Clocks for PSoC 4100/PSoC 4200 devices are chained. The following figure shows the CapSense clocking tree for PSoC 4100/PSoC 4200.

Clocks for Direct Clock Mode in PSoC 4100/PSoC 4200:



Clocks for PRS Clock Mode in PSoC 4100/PSoC 4200:

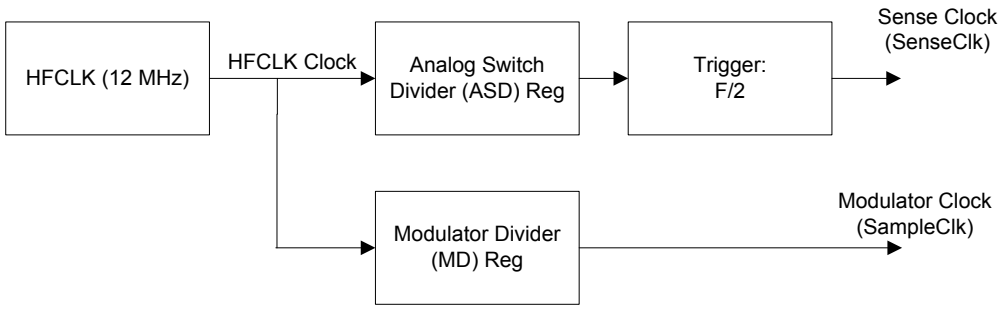


The Modulator clock is formed by dividing the HFCLK Clock by the Modulator Clock Divider. The Sense Clock is formed by dividing the Modulator Clock by the Sense Clock Divider. For example, if you configure the Sense Clock Divider value to 8 and the Modulator Clock Divider value to 4, then the Modulator Clock Divider Reg will be configured to dividing by 4 and the Sense Clock Divider Reg will configured to dividing by 2.

PSoC 4000/PSoC 4100M/PSoC 4200M

Clocks for PSoC 4000/PSoC 4100M/PSoC 4200M devices are not chained. The following figure shows the CapSense clocking tree for PSoC 4000/PSoC 4100M/PSoC 4200M.

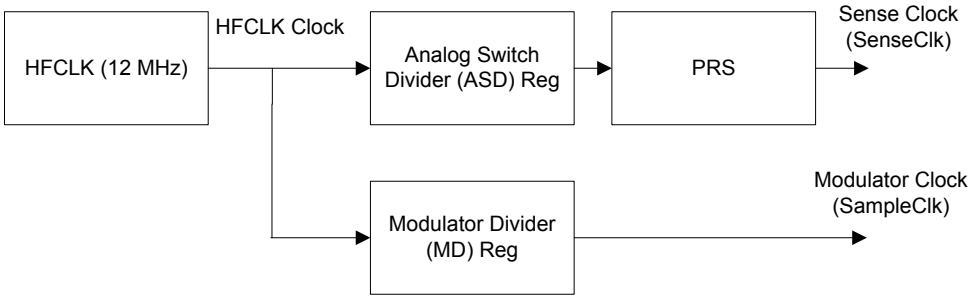
Clocks for Direct Clock Mode in PSoC 4000/PSoC 4100M/PSoC 4200M:



Modulator Clock = HFCLK/MD

Sense Clock = HFCLK/ASD/2

Clocks for PRS Clock Mode in PSoC 4000/PSoC 4100M/PSoC 4200M:



Modulator Clock = HFCLK/MD

PRS Maximum Sense Clock = HFCLK/ASD/2

PRS Average Sense Clock = HFCLK/ASD/4

CapSense Analog System

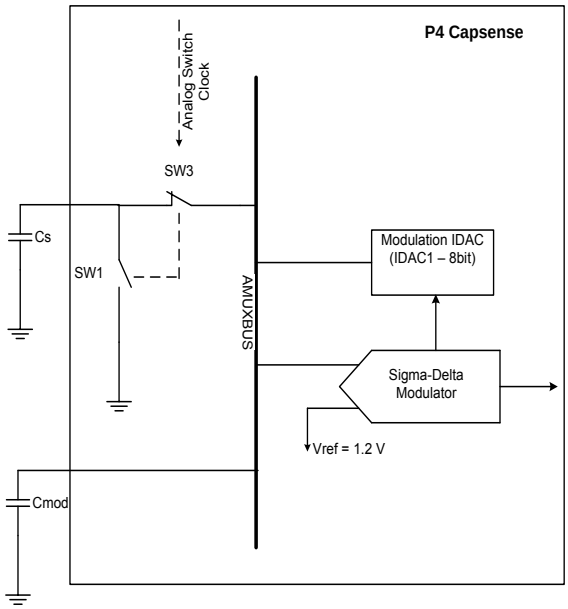
CapSense Analog System consists on Sigma Delta Modulator, Analog MUX bus, Modulation IDAC (IDAC1 – 8 bit, Main IDAC) and Compensation IDAC (IDAC2 – 7 bit, Second IDAC).

In Single IDAC mode (Compensation IDAC is disabled on the general tab of Customizer) and the component uses only Main IDAC (IDAC1 – 8 bit). In this case Main IDAC is configured as variable (controlled by modulator output).

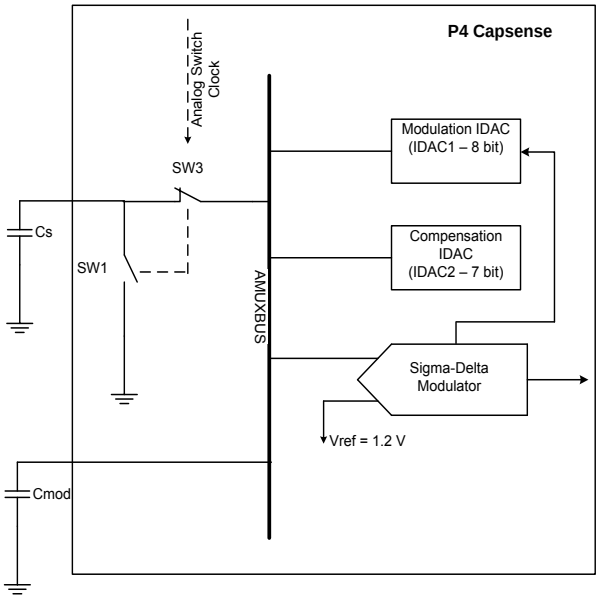
In Split IDAC mode (Compensation IDAC is enabled on general tab of Customizer) the component uses both IDACs (8-bit Main IDAC and 7-bit Second IDAC).

In this case Main IDAC (8-bit) is called Modulation IDAC because it is configured as Variable IDAC and Second IDAC (7-bit) is called Compensation because it is configured as fixed IDAC.

CapSense Analog System in Single IDAC mode



CapSense Analog System in Split IDAC mode



API Resolution – Interpolation and Scaling

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

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

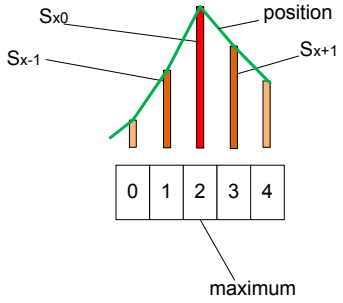
CapSense_GetCentroid (CapSense_CalcCentroid) function in the PSoC4 (for Linear Slider)

$$position = \left(\frac{S_{x+1} - S_{x-1}}{S_{x-1} + S_{x0} + S_{x+1}} + maximum \right) * (Resolution / (n-1))$$

Resolution – API Resolution set in the Customiser,
n – Number of sensor elements in the Customiser.

maximum: Index of maximum element within centroid.

S_i – different counts (with subtracted Noise Threshold value) near by the maximum position:



Example 1:
We have linear centroid of 5 elements with resolution = 100. Noise threshold = 2.
CapSense_sensorSignal= [0, 0, 100, 200, 100].
maximum = 3;
Then position = ((98-98)/(98+108+98) + 3)*100/(5-1) = 75.

Example 2:
We have linear centroid of 5 elements with resolution = 100. Noise threshold = 20.
CapSense_sensorSignal= [0, 10, 100, 210, 180].
maximum = 3;
Then position = ((160-80)/(80+190+160) + 3)*100/(5-1) = 79.65 = 80 ← Rounded

Note1 for Radial Slider:

$$position = \left(\frac{S_{x+1} - S_{x-1}}{S_{x-1} + S_{x0} + S_{x+1}} + maximum \right) * (Resolution / n)$$

if position < 0 then

$$position = \left(\frac{S_{x+1} - S_{x-1}}{S_{x-1} + S_{x0} + S_{x+1}} + maximum + n \right) * (Resolution / n)$$

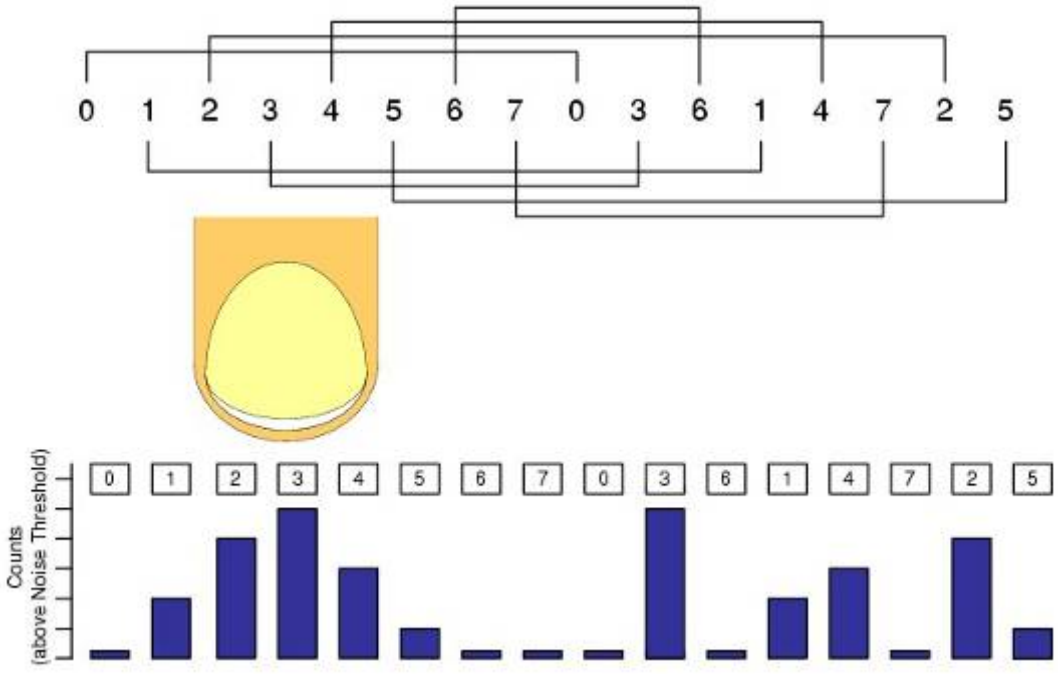
Note2 for Radial Slider:
For Radial Slider the algorithm takes to the account the first and last slider segments.
For example if CapSense_sensorSignal= [30, 0, 0, 40, 180] the position in the Radial Slider is calculated for x0; x3 and x4 elements. But in the Linear Slider the position is calculated for x3 and x4 elements only.

The calculated value is typically fractional. In order to report the centroid to a specific resolution, for example a range of 0 to 100 for 12 sensors, the centroid value is multiplied by a 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 CapSense CSD customizer.

Diplexing

In a diplexed slider, each PSoC sensor connection in the 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 you assigning the port pin using the CapSense customizer. The second (or upper) half of the physical sensor locations is automatically mapped by an algorithm in the customizer and listed in an include file. The order is established so that adjacent sensor actuation in one half does not result in adjacent sensor actuation in the other half. Be careful to determine this order and map it onto the printed circuit board.

Figure 1. Diplexing



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 number table is automatically generated by the CapSense customizer when you select diplexing and is included in the following table for your reference.

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
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

Interrupt Service Routines

The CapSense component uses an interrupt that triggers after the end of each sensor scan. Sub routine is provided where you can add your own code if required. The stub routine is generated in the *CapSense_INT.c* file the first time the project is built. Your code must be added between the provided comment tags in order to be preserved between builds.

Filters

Several filters are provided in the CapSense component: median, averaging, first order IIR and jitter. The filters can be used with both raw sensor data to reduce sensor noise and with position data of sliders and touchpad to reduce position noise.

Median Filter

The median filter looks at the three most recent samples and reports the median value. The median is calculated by sorting the three samples and taking the middle value. This filter is used to remove short noise spikes and generates a delay of one sample. This filter is generally not recommended because of the delay and RAM use. Enabling this filter consumes 4 bytes of RAM for each sensor(raw) and Widget(position). It is disabled by default.

Averaging Filter

The averaging filter looks at the three most recent samples of position and reports the simple average value. It is used to remove short noise spikes and generates a delay of one sample. This filter is generally not recommended because of the delay and RAM use. Enabling this filter consumes 4 bytes of RAM for each sensor(raw) and Widget(position). It is disabled by default.

First Order IIR Filter

The first order IIR filter is the recommended filter for both raw and sensor filters because it requires the smallest amount of SRAM and provides a fast response. The IIR filter scales the most recent sensor or position data and adds it to a scaled version of the previous filter output. Enabling this filter consumes and 2 bytes of RAM for each sensor(raw) and Widget(position). The IIR1/4 is enabled by default for both raw and position filters.

1st-Order IIR filters:

$$\mathrm{IIR}\,1/2 = 1/2\,\mathrm{previous} + 1/2\,\mathrm{current}$$

$$\mathrm{IIR}\,1/4 = 3/4\,\mathrm{previous} + 1/4\,\mathrm{current}$$

$$\mathrm{IIR}\,1/8 = 7/8\,\mathrm{previous} + 1/8\,\mathrm{current}$$

$$\mathrm{IIR}\,1/16 = 15/16\,\mathrm{previous} + 1/16\,\mathrm{current}$$

Jitter Filter

This filter eliminates noise in the raw sensor or position data that toggles between two values (jitter). If the most current sensor value is greater than the last sensor value, the previous filter value is incremented by 1; if it is less, it is decremented. This is most effective when applied to data that contains noise of four LSBs peak-to-peak or less and when a slow response is acceptable, which is useful for some position sensors. Enabling this filter consumes two bytes of RAM for each sensor(raw) and Widget(position). It is disabled by default.

Water Influence on CapSense System

The water drop and finger influence on CapSense are similar. However, water drop influence on the whole surface of the sensing area differs from a finger influence.

There are several variants of water influence on the CapSense surface:

- Forming of thin stripes or streams of water on the device surface.
- Separate drops of water.
- Stream of water covering all or a large portion of the device surface, when the device is being washed or dipped.

Salts or minerals that the water contains make it conductive. Moreover, the greater their concentration, the more conductive the water is. Soapy water, sea water, and mineral water are liquids that influence the CapSense unfavorably. These liquids emulate a finger touch on the device surface, which can cause faulty device performance.

Waterproofing and Detection

This feature configures the CapSense CSD component to suppress water influence on the CapSense system. This feature sets the following parameters:

- Enables a Shield electrode to be used to compensate for the water drops' influence on the sensor at the hardware level.

Shield Electrode

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

In some applications it is useful to select the shield electrode signal and its placement relative to the sensing electrodes such that increasing the coupling between these electrodes caused by

moisture causes a negative touch change of the sensing electrode capacitance measurement. This simplifies the high-level software API work by suppressing false touches caused by moisture. The CapSense CSD component supports separate outputs for the shield electrode to simplify PCB routing.

Figure 2. Possible Shield Electrode PCB Layout

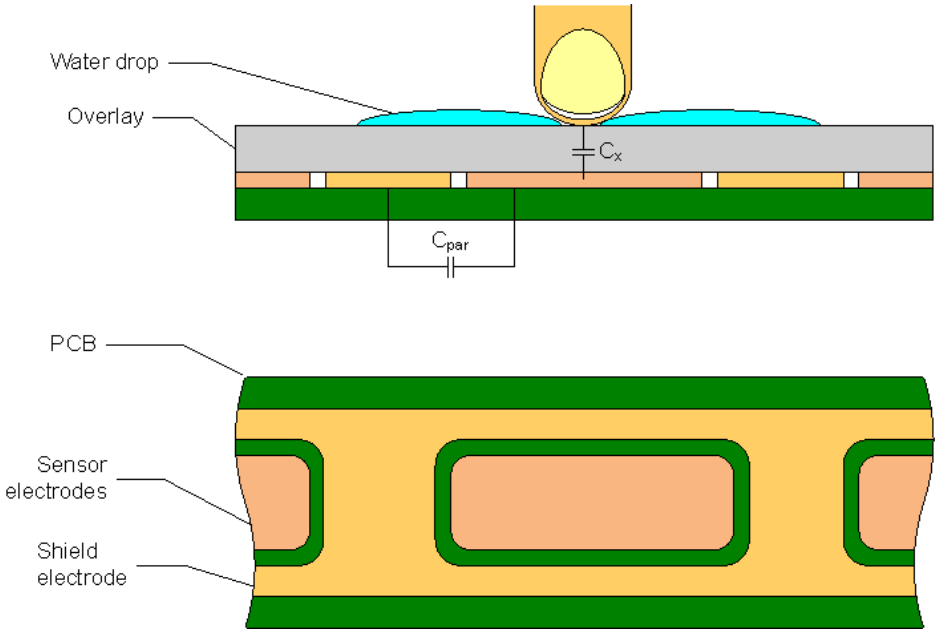


Figure 2 illustrates one possible layout configuration for the button's shield electrode. The shield electrode is especially useful for transparent ITO touchpad devices, where it blocks the LCD drive electrode's noise and reduces stray capacitance at the same time.

In this example, the button is surrounded by a shielding electrode plane. As an alternative, the shielding electrode can be located on the opposite PCB layer, including the plane under the button. A hatch pattern is recommended in this case, with a fill ratio of about 30 to 40 percent. No additional ground plane is required in this case.

When water drops are located between the shield and sensing electrodes, the parasitic capacitance (C_{PAR}) is increased and modulator current can be reduced.

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

How to use the proximity sensors

Proximity sensors detect the presence of a hand in the three-dimensional space around the sensor. However, the actual output of the proximity sensor is an ON/OFF state similar to a CapSense button. The ON/OFF state of the proximity sensor can be detected using the CapSense_CheckIsSensorActive() or CapSense_CheckIsWidgetActive() API.

Proximity sensing can detect a hand at a distance of several centimeters to tens of centimeters depending on the sensor construction. To increase the detected distance, the diameter of the proximity sensor loop should be increased also. In practice, a well-configured proximity sensor has a scan resolution of 16 bits and it requires a scan time much more than one for the normal sensors. Because of the long scan time, the proximity widgets are excluded from the scanning process by default. Use the CapSense_EnableWidget() function to enable the proximity widgets.

The CapSense_GetDiffCountData() API can be used to read the sensor signal level on the proximity sensor. The Customizer provides the #defines for the proximity widget/sensor numbers that are contained in the *Capsense_CSHL.h* and *Capsense.h* files. See the Widget Constants and Sensor Constants sections for details.

You can also implement a proximity sensor by ganging other sensors together. This is accomplished by combining multiple sensor pads into one large sensor using firmware. The disadvantage of this method is high parasitic capacitance. See the Complex sensors section of this document for details.

Resources

Digital Resources

Configuration	Resource Type	
	CSD Fixed Blocks	Interrupts
All Configurations	1	1

Analog Resources

Configuration	Resource Type	
	8-bit CapSense IDACs	7-bit CapSense IDACs
Compensation IDAC disabled	1	0
Compensation IDAC enabled	1	1
SmartSense	1	1

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ T_A ≤ 85 °C and T_J ≤ 100 °C, except where noted.
Specifications are valid for 1.71 V to 5.5 V, except where noted.

DC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
V_{CSD}	Voltage range of operation	1.71	–	5.5	V	

AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
SNR	Ratio of counts of finger to noise	5	–	–	Ratio	1) Capacitance range of 9 to 35 pF, 0.1 pF sensitivity. 2) Capacitance range of 9 to 45 pF, 0.2 pF sensitivity.
IDAC1	DNL for 8-bit resolution	-1	–	1	LSB	
IDAC1	INL for 8-bit resolution	-3	–	3	LSB	
IDAC2	DNL for 7-bit resolution	-1	–	1	LSB	
IDAC2	INL for 7-bit resolution	-3	–	3	LSB	
IDAC1_CRT1	Output current of Idac1 (8-bits) in High range	–	612	–	µA	
IDAC1_CRT2	Output current of Idac1(8-bits) in Low range	–	306	–	µA	
IDAC2_CRT1	Output current of Idac2 (7-bits) in High range	–	305	–	µA	
IDAC2_CRT2	Output current of Idac2 (7-bits) in Low range	–	153	–	µA	

Component Changes

Version	Description of Changes	Reason for Changes / Impact
2.20	Added support for PSoC 4100M/PSoC 4200M devices. Removed the Cmod precharge in **Advanced** Tab. Precharge by Vref buffer is set to default.	New devices and features.
2.10.a	Datasheet edits.	Added default values for some parameters and clarified device support. Clarified that CapSense_TunerComm() API is a blocking call. Added new parameters to AC Specifications Added CapSense_EnableRawDataFilters and CapSense_DisableRawDataFilters APIs.
2.10	The "Shield tank capacitor" field in the Customizer is set to the "Disabled" state when shield is disabled. The "Shield signal delay" is set to "None (default)" and greyed out in the Customizer when shield is disabled. The "Shield Tank capacitor enable" is set to "Disabled (default)" and greyed out in the Customizer when shield is disabled. The Precharge setting of Shield tank capacitor is greyed out in the Customizer. Additional explanation of how to use proximity is added to the datasheet. Scan time values and resolutions are provided in the datasheet. Build Error when CSD is configured for Generic Widget only is fixed. Tuner is updated to show the actual IDAC values in the Manual tuning mode when Auto Calibration option is enabled. Sensitivity parameter on the Scan Order tab is greyed out in the Customizer for Manual Tuning.	New devices and features.

Version	Description of Changes	Reason for Changes / Impact
2.0	Added support for PSoC 4000 devices. Tuning and scanning algorithms were updated. Changed names for Tuning Modes and IDACs in the dialog: • Baselining IDAC was renamed into Modulation IDAC and it is always 8 bit; • Compensating IDAC was renamed into Compensation IDAC and it is 7 bit; • None Tuning Method was renamed into Manual one; • Manual Tuning Method was renamed into Manual with run-time tuning; • CapSense_idac1Settings array was renamed into CapSense_modulationIDAC one; • CapSense_idac2Settings array was renamed into CapSense_compensationIDAC one; Added new APIs for parameters setting/reading. Added BIST support. Added Autocalibration support for manual mode.	New devices. Better performance. Improved usability.
1.11	Several global array names and descriptions were changed and a few non-descript global arrays were added.	
	Added MISRA Compliance section.	This component was not verified for MISRA-C:2004 coding guidelines compliance.
1.10	The scan time was optimized.	
1.0.a	Updated link to PSoC 4 CapSense Design Guide, and various edits to the datasheet	
1.0	Initial version.	

© Cypress Semiconductor Corporation, 2015. 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® and CapSense® are registered trademarks, and SmartSense™, PSoC Creator™, and Programmable System-on-Chip™ are trademarks 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.