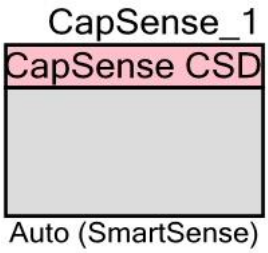
PSoC 4 Capacitive Sensing (CapSense® CSD)

1.10



Features

- Support for user-defined combinations of button, slider, touchpad, and proximity capacitive sensors
- Automatic SmartSense™ tuning or manual tuning with integrated PC GUI
- High immunity to AC power line noise, EMC noise, and power supply voltage changes
- Shield electrode support for reliable operation in the presence of water film or droplets
- Guided sensor and terminal assignments using the CapSense customizer

General Description

Capacitive Sensing, using a Delta-Sigma Modulator (CapSense CSD) component, is a versatile and efficient way to measure capacitance in applications such as touch sense buttons, sliders, touchpad, and proximity detection.

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

- *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.

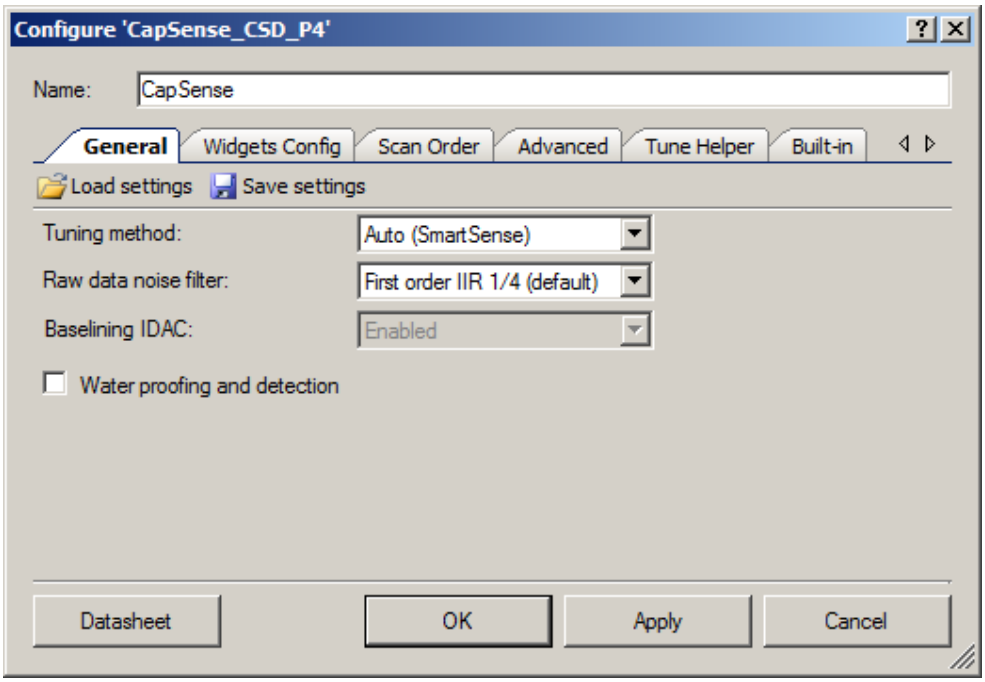
Input/Output Connections

N/A

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 into the Tuner GUI.

Tuning method

This parameter specifies the tuning method. There are three options:

- **Auto (SmartSense)** – Provides automatic tuning of the CapSense CSD component.

  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.

  Important –SmartSense tuning may be used with a I2C communication, which is specified on the **Tuner Helper** tab, to transmit data from the target device to the Tuner GUI.

- **Manual** – Allows you to tune the CapSense CSD component manually using the Tuner GUI.

  To launch the GUI, right-click on the symbol and select **Launch Tuner**. For more information about manual tuning, see the Tuner GUI User Guide section in this datasheet. Manual tuning requires a I2C communication, which is specified on the **Tuner Helper** tab, to transmit data between the target device and the Tuner GUI.

- **None** (default) – Disables tuning.

  All tuning parameters are stored in flash. You should use this option only after all parameters of the CapSense component are tuned and finalized. If you use this option, Tuner will work in read-only mode.

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 Filters in the Functional Description 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.
- **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.
- **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.

Baselining IDAC

This parameter enables the split IDACs mode. This mode provides increasing sensitivity and SNR. The Baselining 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

Water proofing and detection

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

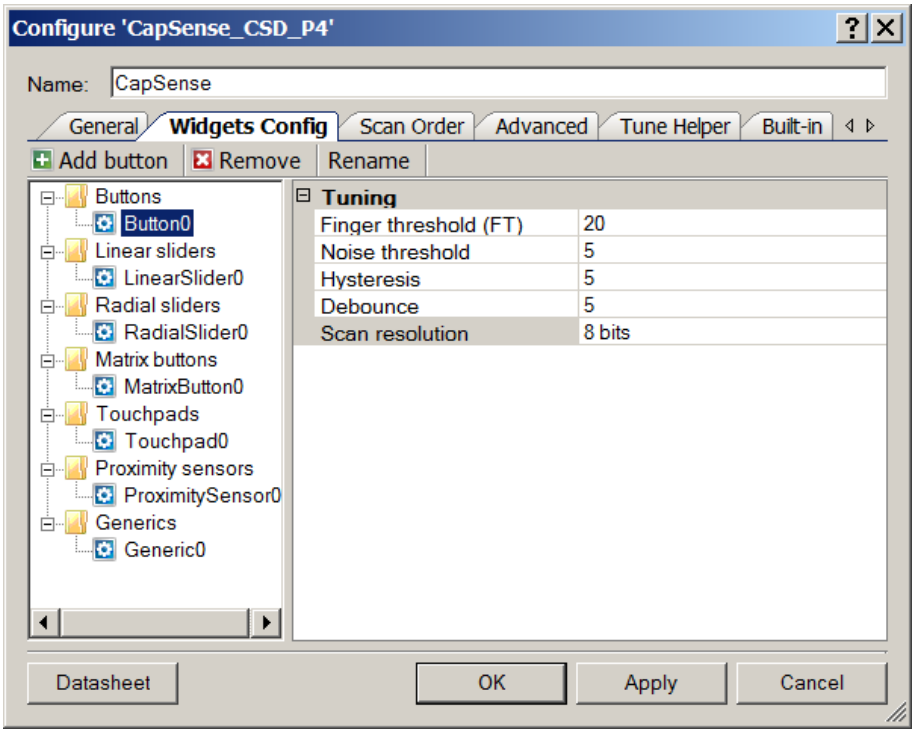
- Enables the Shield output terminal

  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.

Widgets Config Tab



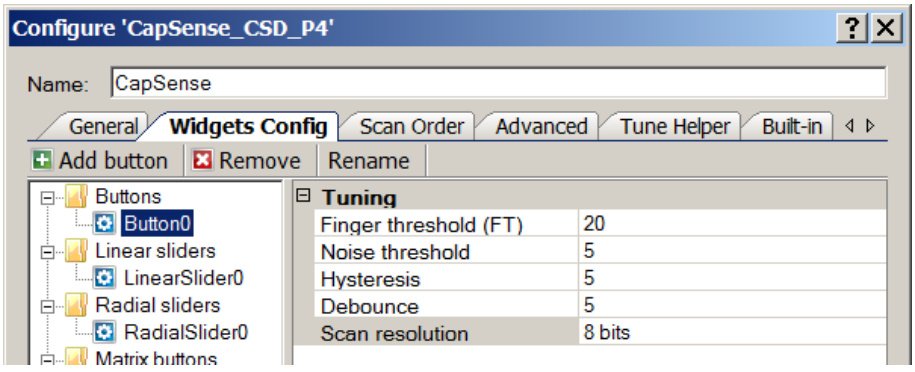
Definitions for various parameters are provided in the Functional Description section.

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 widget** (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.

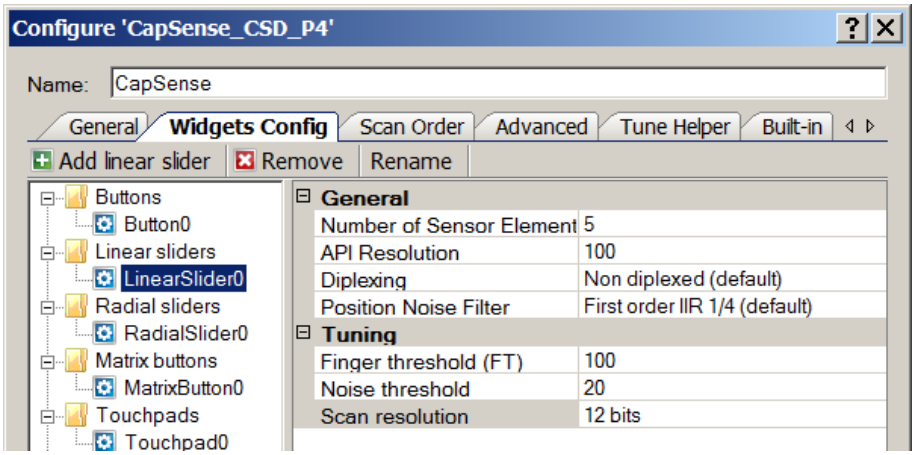
- **Noise Threshold** – Defines sensor noise threshold. Count values above this threshold do not update the baseline. If the noise threshold is too low, sensor and thermal offsets may not be accounted for. This can result in false or missed touches. If the noise threshold is too high, a finger touch 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.

- **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]] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution.

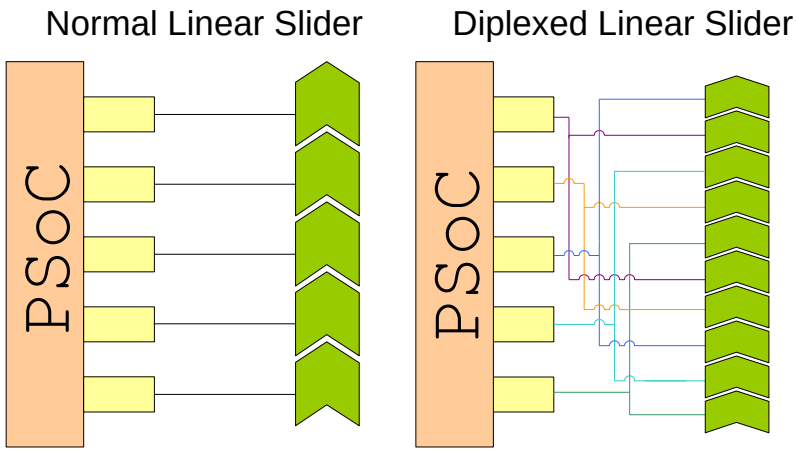
- **Scan Resolution** – Defines the scanning resolution. This parameter affects the scanning time of the sensor within the button widget. The maximum raw count for the scanning resolution for N bits is $2^N - 1$. Increasing the resolution improves sensitivity and the signal-to-noise ratio (SNR) of touch detection but increases scan time. Default value is **10 bits**.

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].
- **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.

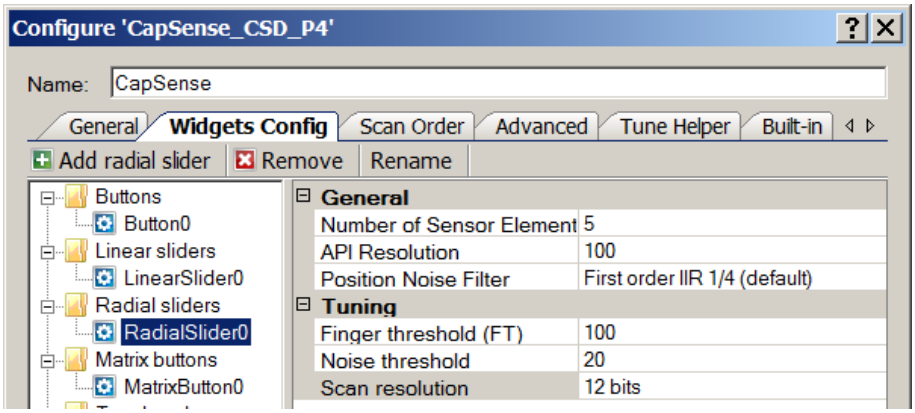


- **Position Noise Filter** – Selects the type of noise filter to perform on position calculations. Only one filter can be applied for a selected widget. Details about the types of filters can be found in Filters in the Functional Description section in this document.
  - ❑ **None**
  - ❑ **Median**
  - ❑ **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**.

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].
- **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 in Filters in the Functional Description section of this datasheet.
  - None
  - Median
  - Averaging
  - First Order IIR 1/2
  - First Order IIR 1/4 (default)
  - Jitter

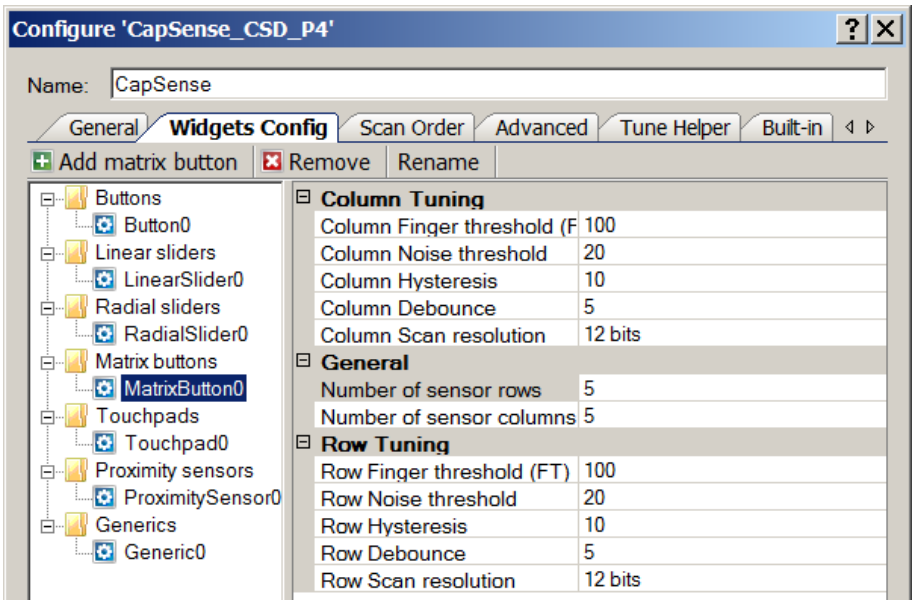
Tuning:

- **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**.

Matrix Buttons



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.

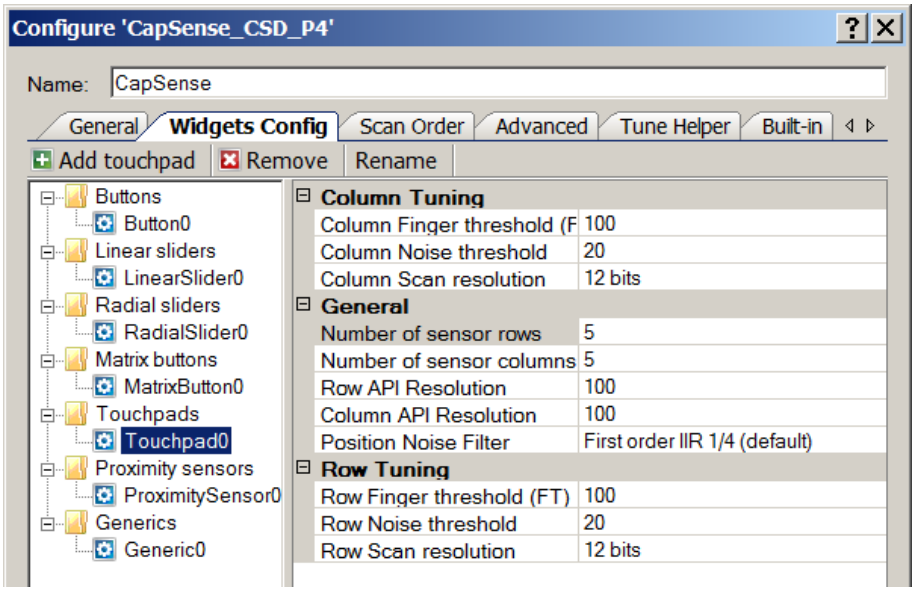
- **Column and Row Debounce** – Adds a debounce counter for detection of the sensor active state transition for matrix buttons column or row. For the sensor to transition from inactive to active, the difference count value must stay above the finger threshold plus hysteresis for the number of samples 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] for 8-bit widget resolution and [1..65535] for 16-bit widget resolution.

- **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**.

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.

Touchpads



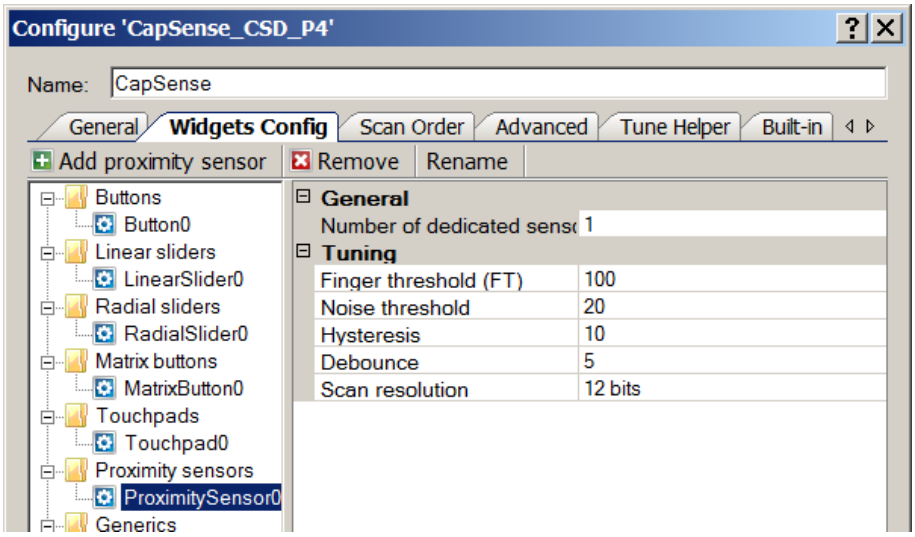
Tuning:

- **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**.

General:

- **Numbers of Sensors Column and Row** – 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.
- **API Resolution Column and Row** – Defines the resolution of the touchpad columns and rows. The finger position values are reported within this range. Valid range of values is [1…255].
- **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 Filters in the Functional Description section in this datasheet.
  - ❑ None
  - ❑ Median
  - ❑ Averaging
  - ❑ First Order IIR 1/2
  - ❑ First Order IIR 1/4 (default)
  - ❑ Jitter

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 *CapSense_EnableWidget()* function to enable proximity widget.

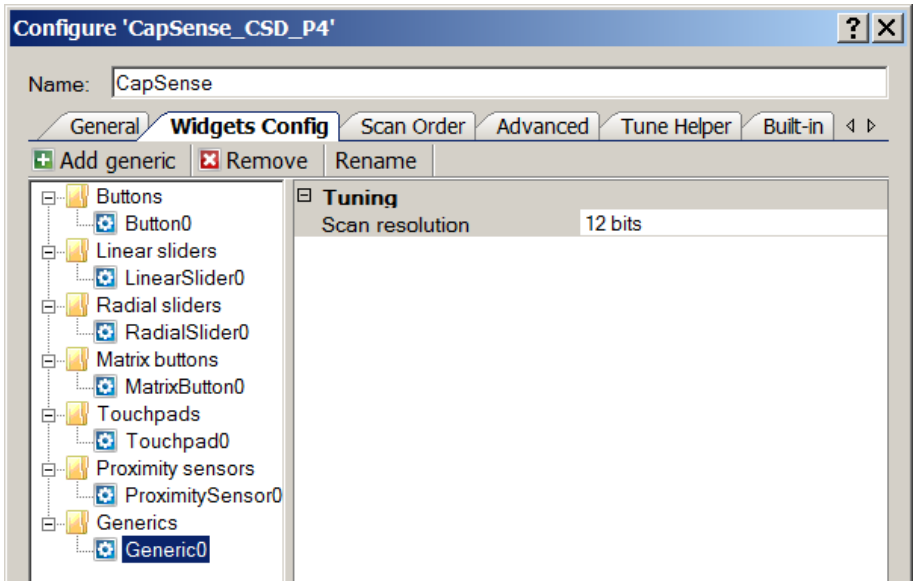
General:

- **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.

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.
- **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.
- **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] 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 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 **10 bits**.

Generics

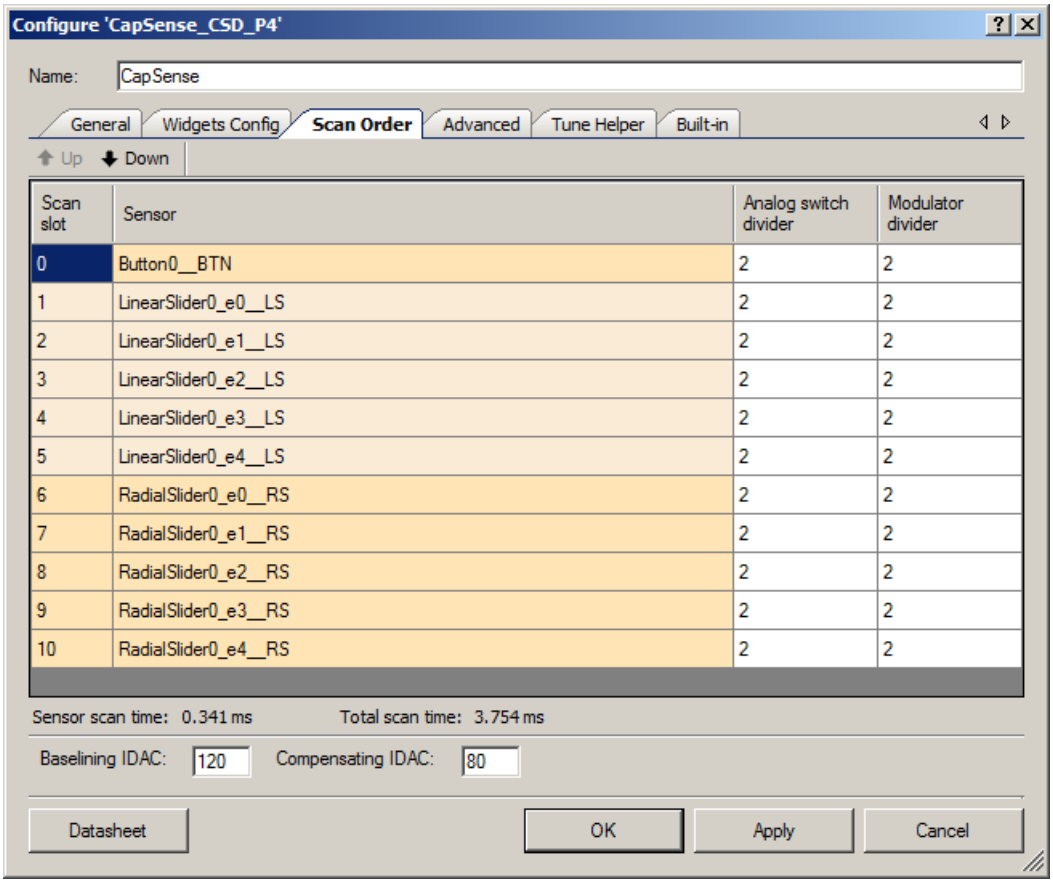


Tuning:

- **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



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 only to generic and proximity widgets).

Analog Switch Divider Column

Specifies the **Analog Switch Divider** value and determines the precharge switch output frequency for scan slot. Valid range of values is [2…255]. Default value is 12.

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

Modulator Divider Column

Specifies the **Modulator Divider** value and determines the modulator input frequency for scan slot. Valid range of values is [2…255]. Default value is 12.

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

Baselining IDAC Value

Specifies the Baselining IDAC value. Valid range is 0 to 250 for 4x range and 0 to 125 for 8x range. Default value is **200**.

Compensating IDACValue

Specifies the Compensating IDAC value. Valid range is 0 to 127. Default value is **80**.

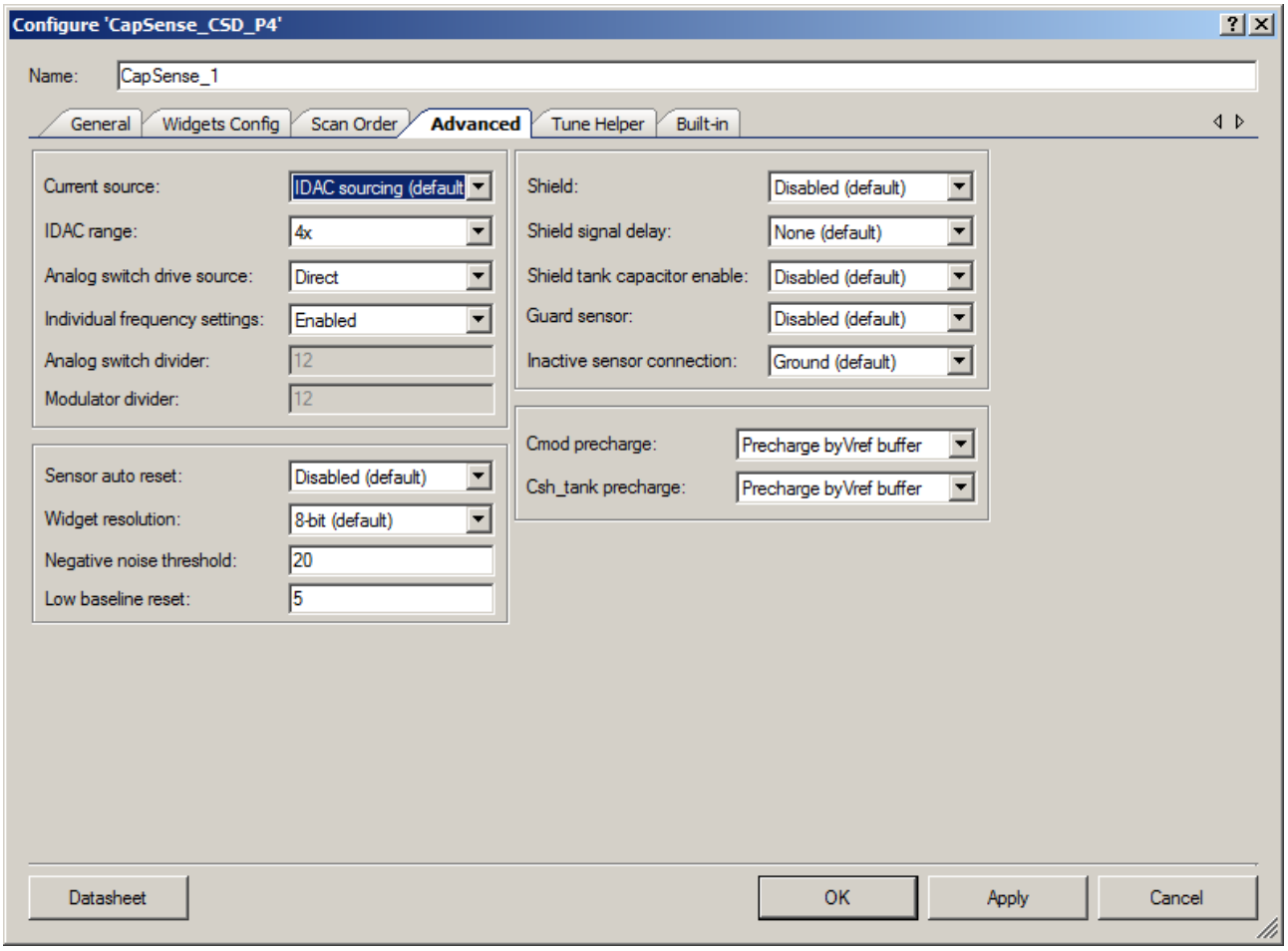
Widget List

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.

Proximity scan sensors can use dedicated proximity sensors, or they can detect proximity from a combination of dedicated sensors, other sensors, or both. 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 is provided on proximity scan sensors to choose one or more sensors to scan to detect proximity.

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 provided.

Advanced Tab



Current Source

CapSense CSD requires 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 of the three methods, but it may require an additional VDAC resource to set the Vref level that the other modes do not require.
- **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**. This parameter is disabled if **Current Source** is set to **External Resistor**. The default is the best choice for almost all CapSense designs. 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 analog switch 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

Individual Frequency Settings

This parameter defines the Analog Switch Divider usage. If enabled, each scan slot uses a dedicated Analog Switch Divider value, otherwise, sensors use only one Analog Switch Divider value.

Analog Switch Divider

This parameter specifies the value of the analog switch divider and determines the precharge switch output frequency. Valid range of values is [2…255]. 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 **Analog Switch 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.

Modulator Divider

This parameter specifies the value of the modulator divider and determines the modulator input frequency. Valid range of values is [2…255]. Default value is **12**.

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

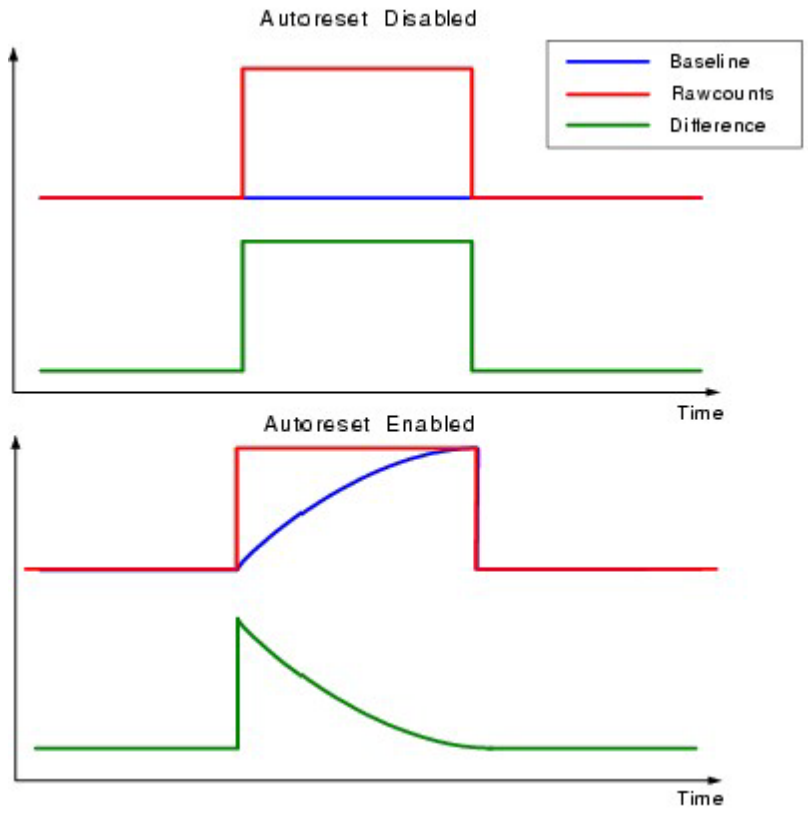
Note The Modulator Divider should be multiple of the Analog Switch Divider due to these dividers are chained. For more details please see CapSense Clocking section.

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. 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.

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 CSD shield is delayed relative to csd_sense.

- None (default)
- 1 cycle
- 2 cycle

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. 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

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**.

Cmod precharge

This parameter specifies precharge source for the Cmod capacitor.

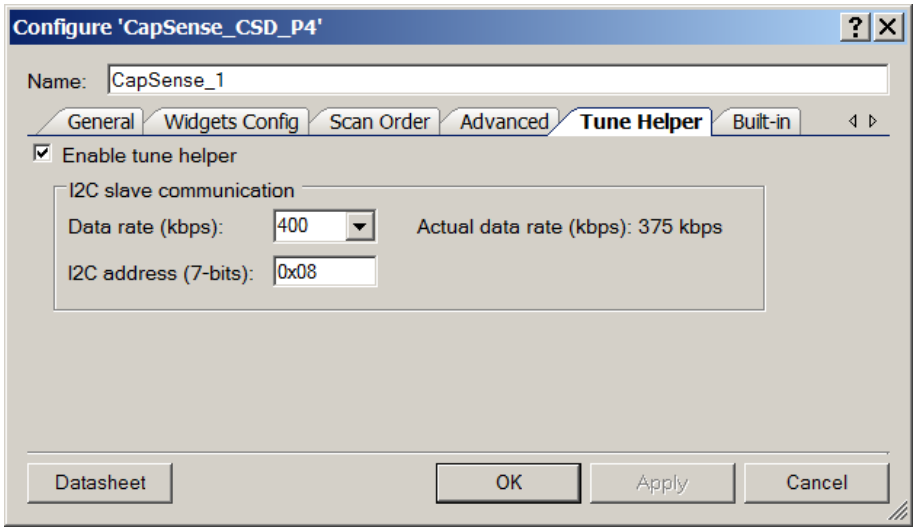
- Precharge by Vref buffer (default)
- Precharge by IO buffer

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.

Data rate

This parameter is used to set the I²C data rate value up to 400 kbps; the actual speed may differ based on available clock speed and divider range. The standard data rates are 50, 100 and 400 (default) kbps. If **Enable Tune Helper** is disabled, this parameter is ignored.

Actual data rate

Actual data rate displays data rate on which communication component will operate with current settings. The selected data rate could be differing from actual data rate. The factors that have effect on actual data rate calculation are: HFCLK clock and accuracy of internal component clock.

I2C address (7-bits)

This is the I²C address that will be recognized by the slave. A slave address between 0 and 127 may be selected; the default is **8**. This address is the 7-bit right-justified slave address and does not include the R/W bit.

The value may be entered as decimal or hexadecimal; for hexadecimal numbers type ‘0x’ before the address. If **Enable Tune Helper** is disabled, this parameter is ignored.

For more information about how to use Tuner GUI, refer to the Tuner GUI User Guide section of this datasheet.

Tuner GUI User Guide

This section includes instructions and information that will help you use the CapSense Tuner.

The CapSense Tuner assists in tuning the CapSense component to the specific environment of the system when in manual tuning mode. It can also display the tuning values (read only) and performance when the component is in SmartSense mode. No tuning is supported when the component is in no tuning mode as all parameters are stored in flash and are read only for minimum SRAM usage.

CapSense Tuning Process

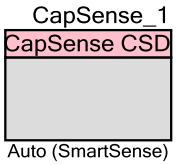
The following is the typical process for using and tuning a CapSense component:

Create a Design in PSoC Creator

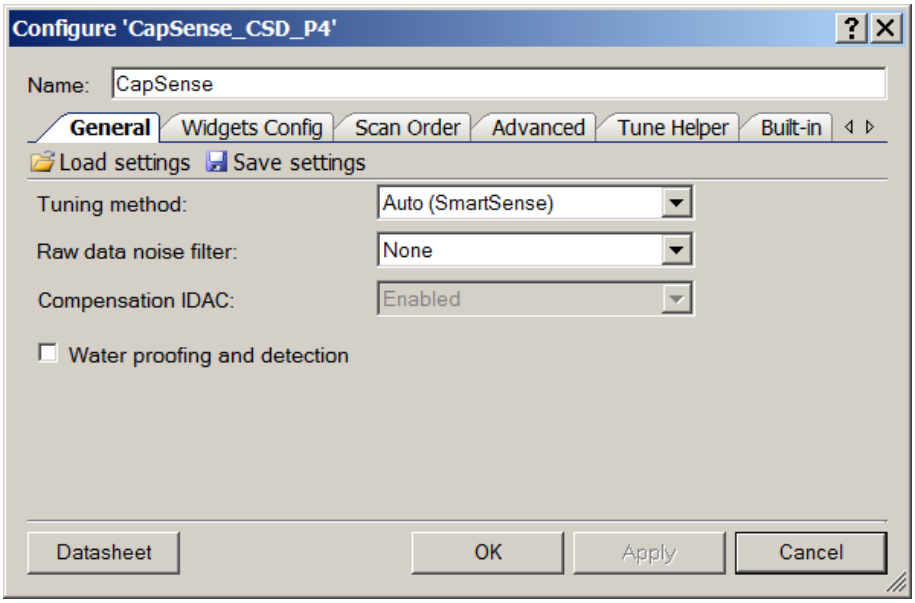
Refer to the PSoC Creator Help as needed.

Place and Configure the CapSense Component

1. Drag a **CapSense_CSD** component from the Component Catalog onto your design.



2. Double-click it to open the **Configure** dialog.
3. Change CapSense CSD parameters as required for your application. Select **Tuning method** as **Manual** or **Auto (SmartSense)**. Click **OK** to close the dialog and save the selected parameters.



Selecting Auto (SmartSense)

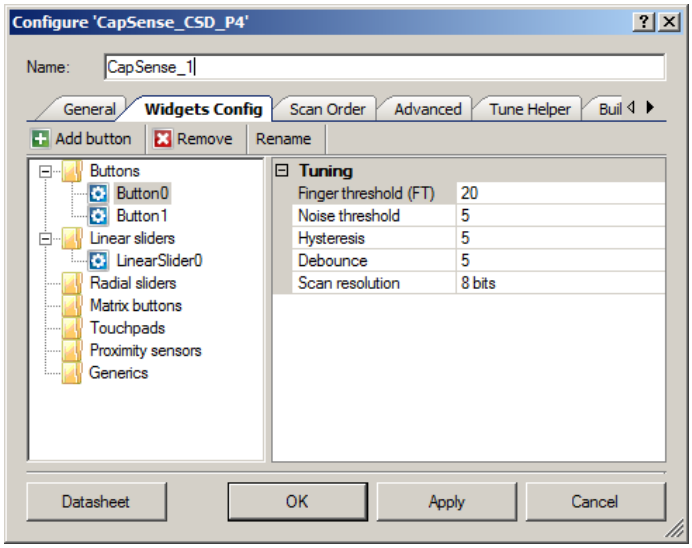
Auto (SmartSense) allows you to tune the CapSense CSD component to the specifics of the system automatically. CapSense CSD parameters are computed at run time by firmware. Additional RAM and CPU time are used in this mode. Auto (SmartSense) eliminates the error-prone and repetitive process of manually tuning the CapSense CSD component parameters to ensure proper system operation. Selecting Auto (SmartSense) tunes the following CSD parameters:

Parameter	Calculation
Finger Threshold	Calculated continuously during sensor scanning.
Noise Threshold	Calculated continuously during sensor scanning.
Baselining IDAC	Calculated once on CapSense CSD startup.
Compensating IDAC	Calculated once on CapSense CSD startup.
Analog Switch Divider	Calculated once on CapSense CSD startup.
Modulator Divider	Calculated once on CapSense CSD startup.

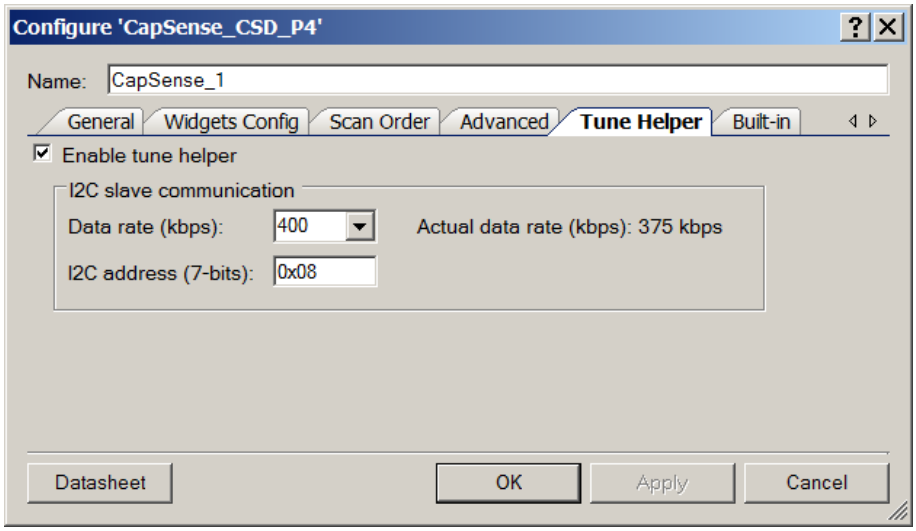
Negative Noise Threshold	Calculated once on CapSense CSD startup.
Low Baseline Reset	Calculated once on CapSense CSD startup.

Configure your CapSense Component

1. Add widgets on the **Widgets Config** tab and configure them.



2. On the **Tune Helper** tab: The **Enable Tune Helper** check box must be selected.



Add Code

- Add Tuner initialization and communication code to the projects *main.c* file. Example *main.c* file:

```
void  main()
{
    CYGlobalIntEnable;
```

```
        CapSense_1_TunerStart();

        /*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.
        */

        while(1)
        {
            CapSense_1_TunerComm();
        }
}
```

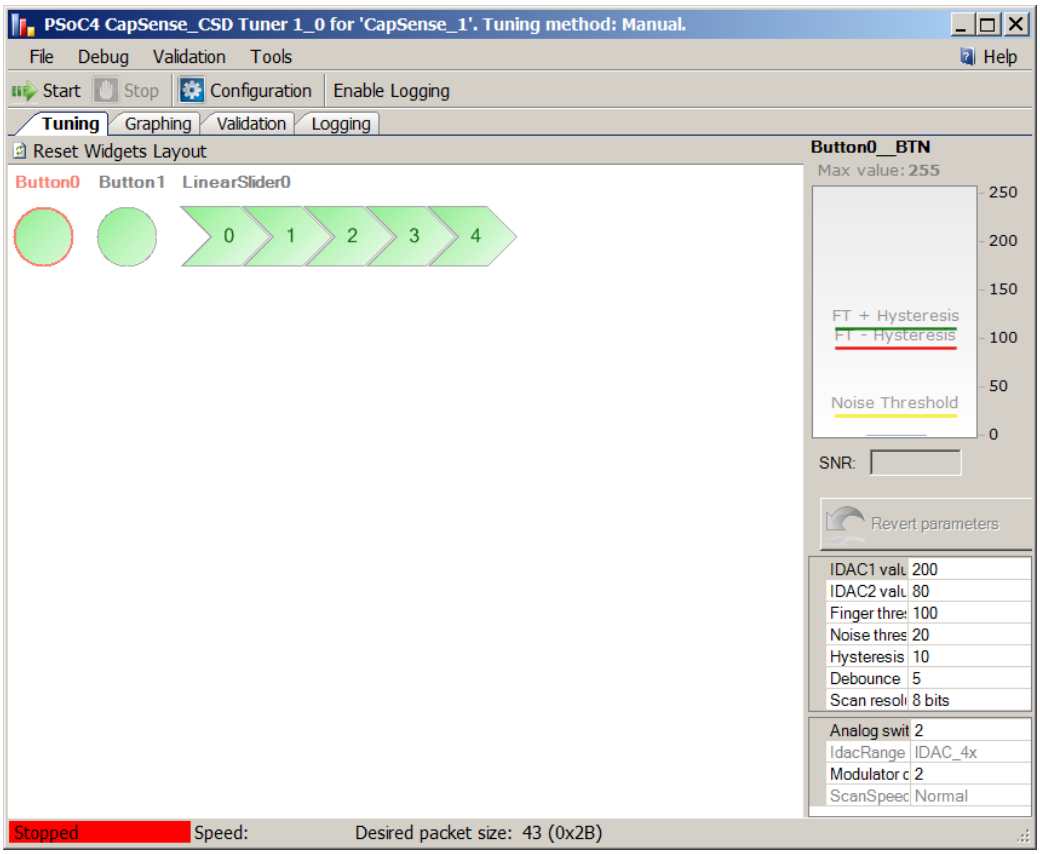
Build the Design and Program the PSoC Device

Refer to PSoC Creator Help as needed.

Launch the Tuner application

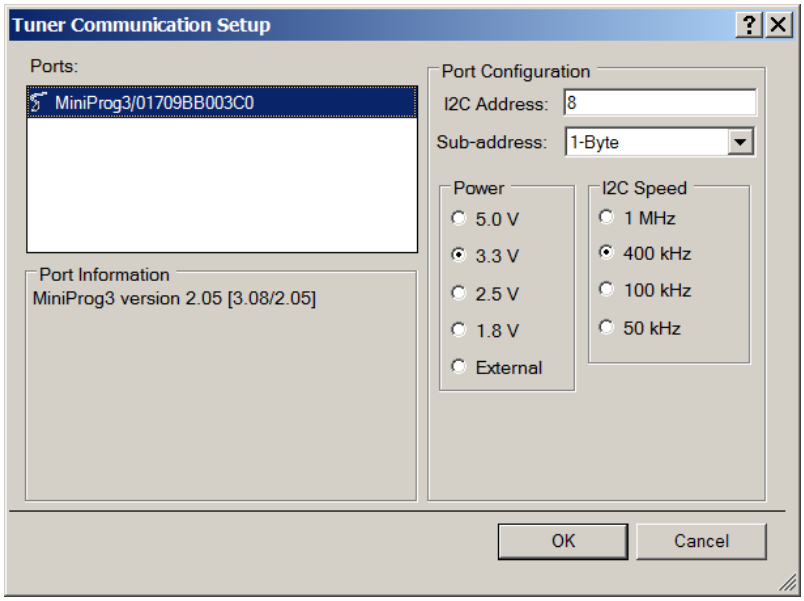
- Right-click the CapSense CSD component icon and select **Launch Tuner** from the context menu.

  The Tuner application opens.



Configure Communication Parameters

1. Click **Configuration** to open the **Tuner Communication** dialog.



2. Set the communication parameters and click **OK**.

Important: Properties must be identical to those in the **Tune Helper** tab: **I2C Speed**, **I2C Address**. **Sub-address** must be set to **1-Byte**.

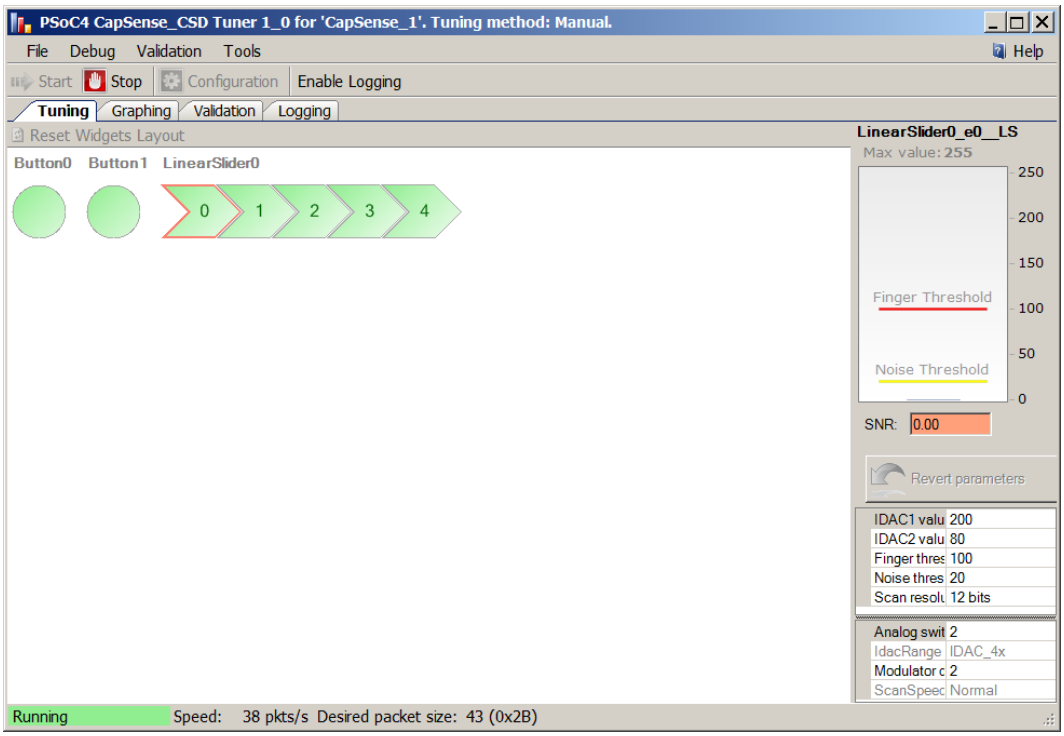
Start Tuning

- Click **Start** on the tuning GUI. All of the CapSense elements start to show their values.

Edit CapSense Parameter Values

- Edit a parameter value for one of the elements, and it is automatically applied after you press the **[Enter]** key or move to another option. The GUI continues to show the scanning data, but

it is now altered based on the application of the updated parameter. Refer to the Tuner GUI Interface section later in this datasheet.



Repeat as Needed

- Repeat steps as needed until tuning is complete and the CapSense component gives reliable touch sensor results.

Close the Tuner application

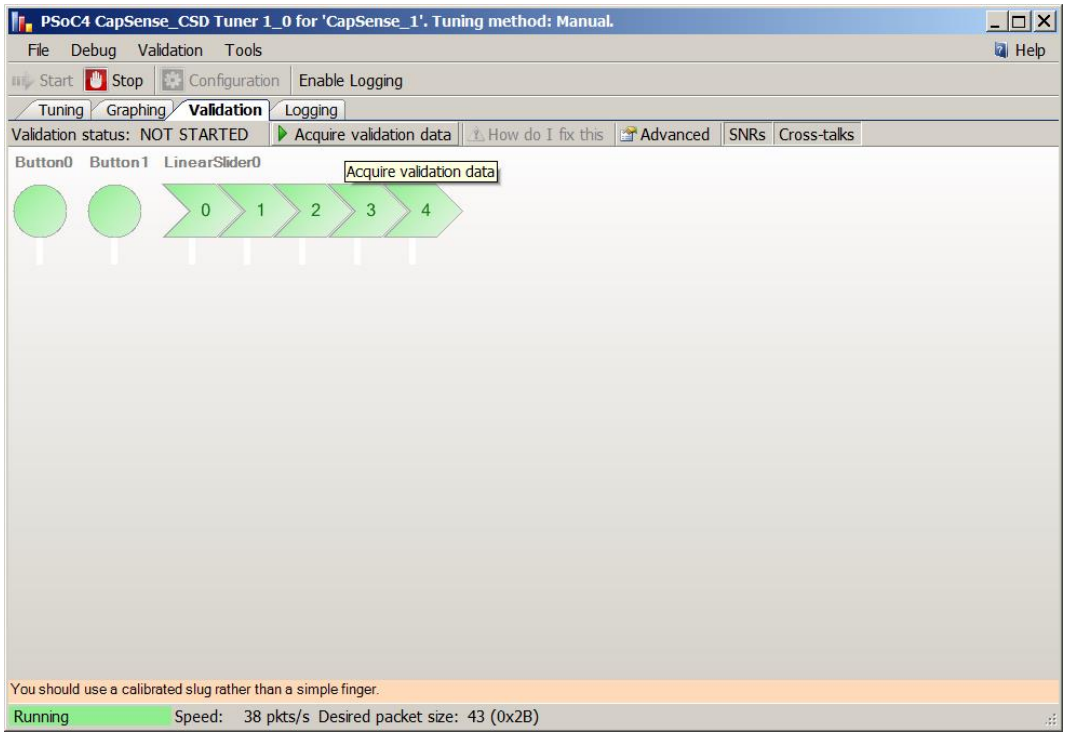
- Click **File->Apply Changes and Close** and the parameters are written back to the CapSense_CSD instance. The Tuner application dialog closes.

CapSense Validation Process

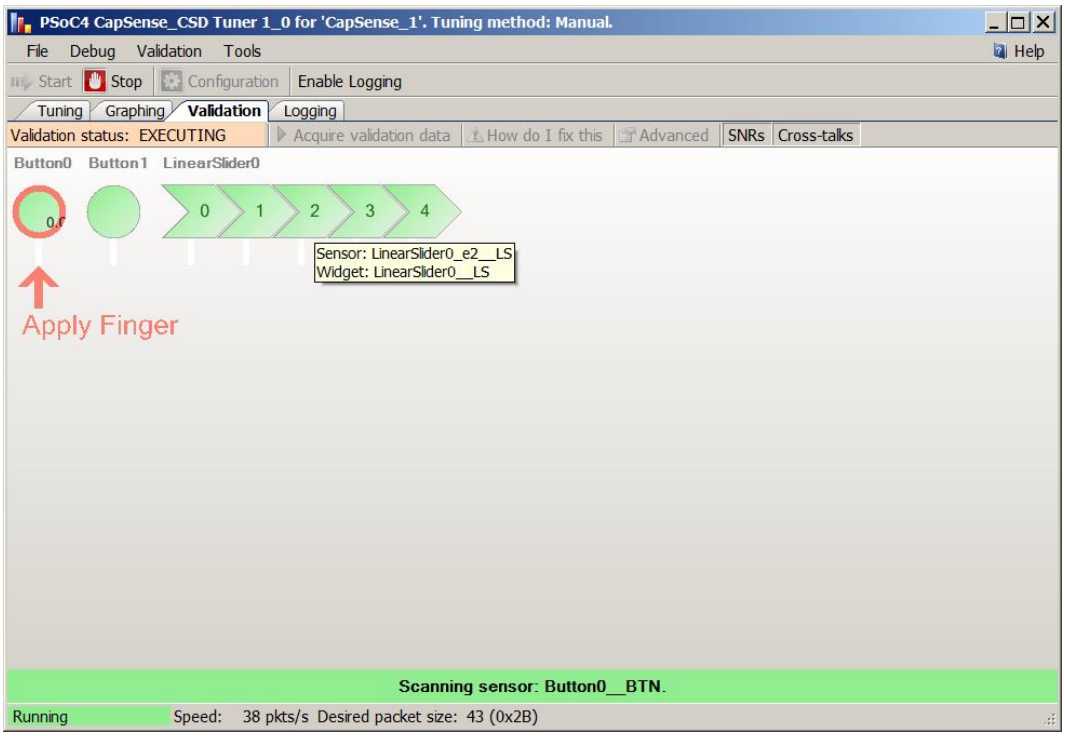
The validation mechanism determines whether the board has been sufficiently tuned. The typical process for using the Tuner Validation feature to validate a CapSense design follows.

Start Validation

1. The Tuner and hardware must be ready before you start the scanning process. See the previous section, CapSense Tuning Process, to prepare the system for scanning.
2. On the **Validation** tab, click **Acquire Validation Data**. Values will begin to appear for all CapSense elements.



Stimulation Sensors



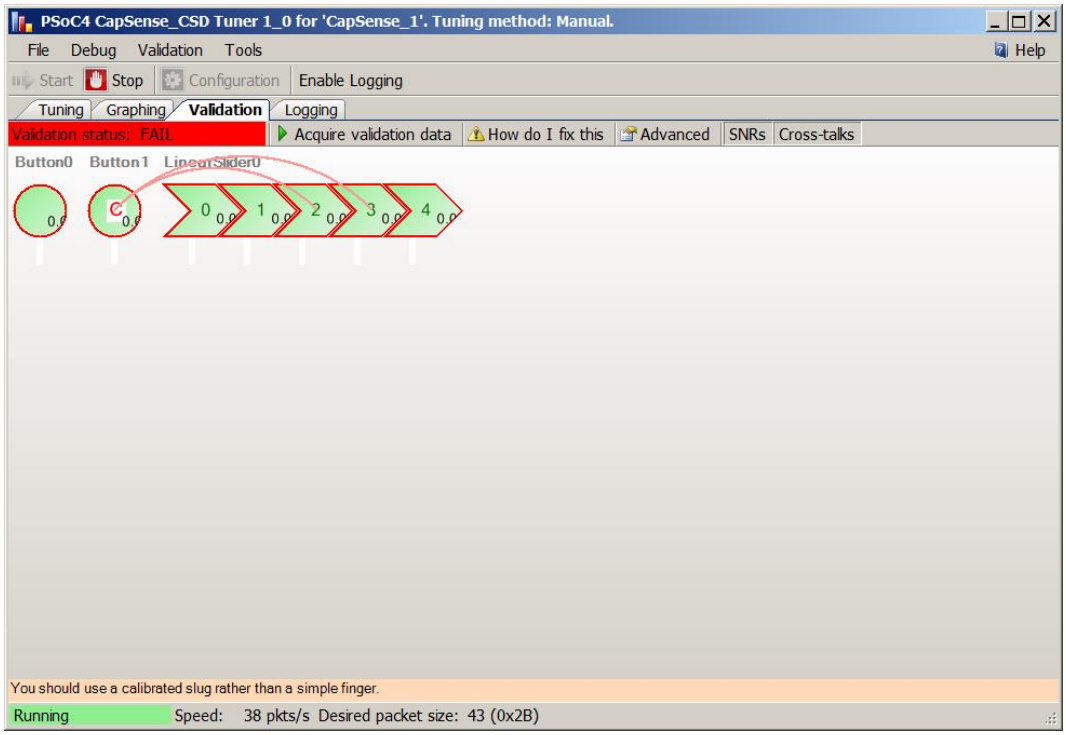
You will be prompted to apply a finger on each sensor. Each time you are prompted to press a CapSense element, a flashing red arrow pointing to the target appears on the layout, with the

text **PRESS HERE**. Text appears beneath the Tuner that will guide you through the validation process.

- To start scanning for the current sensor, press any key on the keyboard.

  It is recommended that you use a calibrated slug instead of a finger press to stimulate the sensors.

Validation Displays



SNR warnings appear as follows:

- **Flashing red** highlights surround any CapSense sensor that has an SNR less than the **Sufficient Value**.
- **Flashing yellow** highlights surround any CapSense sensor that has an SNR between the **Sufficient** and **Optimal Values**.
- **Solid green** highlights surround any CapSense sensor that has an SNR above the **Optimal Value.**

Crosstalk effects warnings appear as follows:

- **Individual Crosstalk Check**. During the validation process, the software monitors all elements other than the one you have been told to stimulate. If an element exhibits difference counts that exceed the **Crosstalk Threshold Percentage** (when not directly stimulated), a

crosstalk warning is generated. This is displayed by a flashing line between the element that exhibits the unwanted counts and the element that was stimulated.

- **Worst Case Crosstalk Check**. As each of the individual crosstalk checks are made, the software keeps a record of each difference count measurement. At the completion of the process, worst-case crosstalk estimates are made.

  For each sensor, a sum appears that is the number of the crosstalk effects equal to the **Worst Case Crosstalk Sensor Count**. The largest crosstalk value is the first element in the sum, the second largest is the second, and so on. For example: if you have the following crosstalk counts (1,5,3,2,4,1,1,0) and the **Worst Case Crosstalk Sensor Count** is 2, then the **Worst Case Crosstalk** computation will be (5 + 4 = 9).

  If this value exceeds the **Worst Case Crosstalk Threshold**, it is flagged with **a flashing "C" character** in the middle of the sensor display.

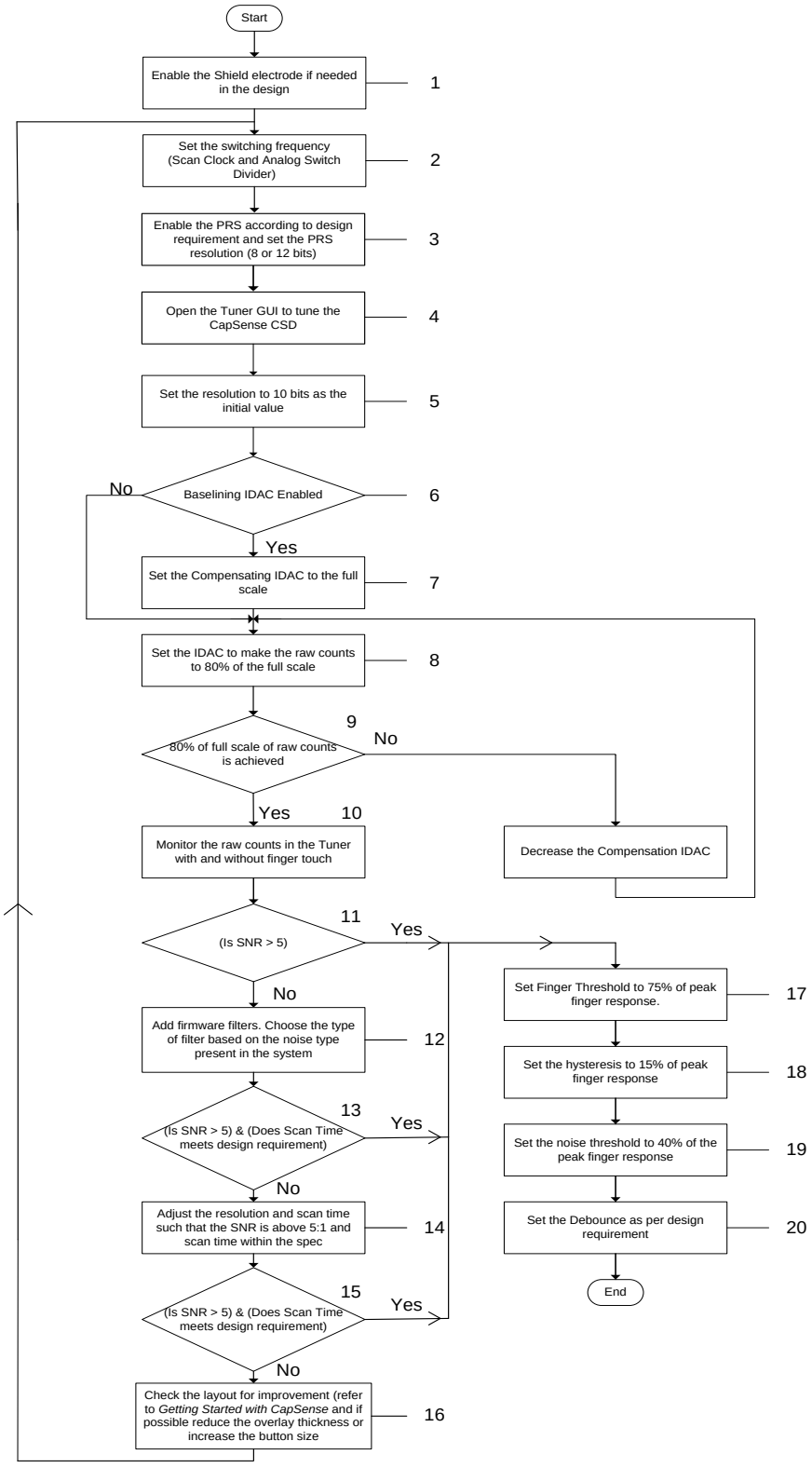
Validation Results

If the validation process uncovers failures, a **Validation Report** will be displayed. This report contains the following information:

- Any SNR values less than the **Optimal Value**
- Any SNR values less than **Sufficient Value**
- Any signals with a worst-case crosstalk failure, and, if so, the crosstalk number

You can also open the Validation Report by clicking the **How do I fix this** button on the **Validation** tab.

Manual Tuning Process



1. The Shield is enabled or disabled depending on the design requirements. A shield is useful in applications where the sensor overlay may become wet (see AN2398), to shield against EMI, or to mitigate excessively high Cp. The shield signal must be connected to an SIO pin if the current mode is set to **IDAC Sourcing** and the reference voltage is **Vref 1.024V**. Otherwise, the shield can be routed to any pin. For more information, see the Shield Electrode section later in this document.
2. The switching frequency of each capacitive sensor should be set such that the sensor completely charges and discharges. **Scan Clock** and **Analog Switch Divider** determine the frequency at which the sensor capacitor is switched. **Scan Clock** is the primary clock source for the CapSense component. The **Analog Switch Divider** (prescaler) divides the scan clock to produce the switching clock used to charge and discharge each sensor. If the sensor is not charging and discharging completely, reduce the switching frequency by increasing the **Analog Switch Divider**.

   To test whether sensors are charging and discharging completely, probe each sensor pin. Note that, while observing the voltage on the sensor capacitor with an oscilloscope probe, the probe capacitance is added to the sensor parasitic capacitance. Using probes in 10x mode reduces the capacitance of the probes. Use an FET input probe, if available. Make sure that the sensor is charging and discharging completely; if not, increase the **Analog Switch Divider** value in the component configuration. Because this parameter cannot be changed in the Tuner GUI, it must be set in the component configuration. Therefore, when this value is changed in the component **Configure** dialog, the project should be built again and programmed to the device.
3. The PRS is enabled by default to reduce the effects of external EMI on CapSense as well as to reduce sensor scan emissions. It is enabled in designs prone to EMI effects. The resolution of the PRS is set to 8 or 12 bits depending on the scan time. For longer scan times, use PRS 12; for shorter scan times, use PRS 8.
4. Open the CapSense Tuner GUI.
5. Set the resolution to 10.

   Increasing the resolution and scan speed give better sensitivity but they increase the scan time. Therefore, there is a tradeoff between scan time and sensitivity.

   The resolution of 10 is a good starting value in the tuning process, although lower resolutions of 8 and 9 can also be used as initial values if the design has thin overlays less than 1 mm.
6. Enable baselining IDAC if needed.
7. I baselining IDAC is enabled then set the Compensating IDAC to the full scale.
8. Change the Baselining IDAC value in the GUI until the raw counts reach 80 percent of the full scale value. The full scale value is 2^Resolution. Note that decreasing the IDAC value increases the raw counts and vice versa. If it is not possible to achieve 80 percent with any of the IDAC values, then you should change the IDAC range in the component configuration. The IDAC range cannot be changed in the Tuner GUI; it should be changed in the component **Configure** dialog. When the value is changed in the **Configure** dialog, you should build the project again and program it to the device.

9. If it is not possible to achieve 80 percent with any of the Baselining IDAC values, then you should decrease the Compensating IDAC value and repeat step 8.
10. Monitor the raw counts with and without a finger present. Note the peak-to-peak noise and peak finger response. Calculate the SNR as,

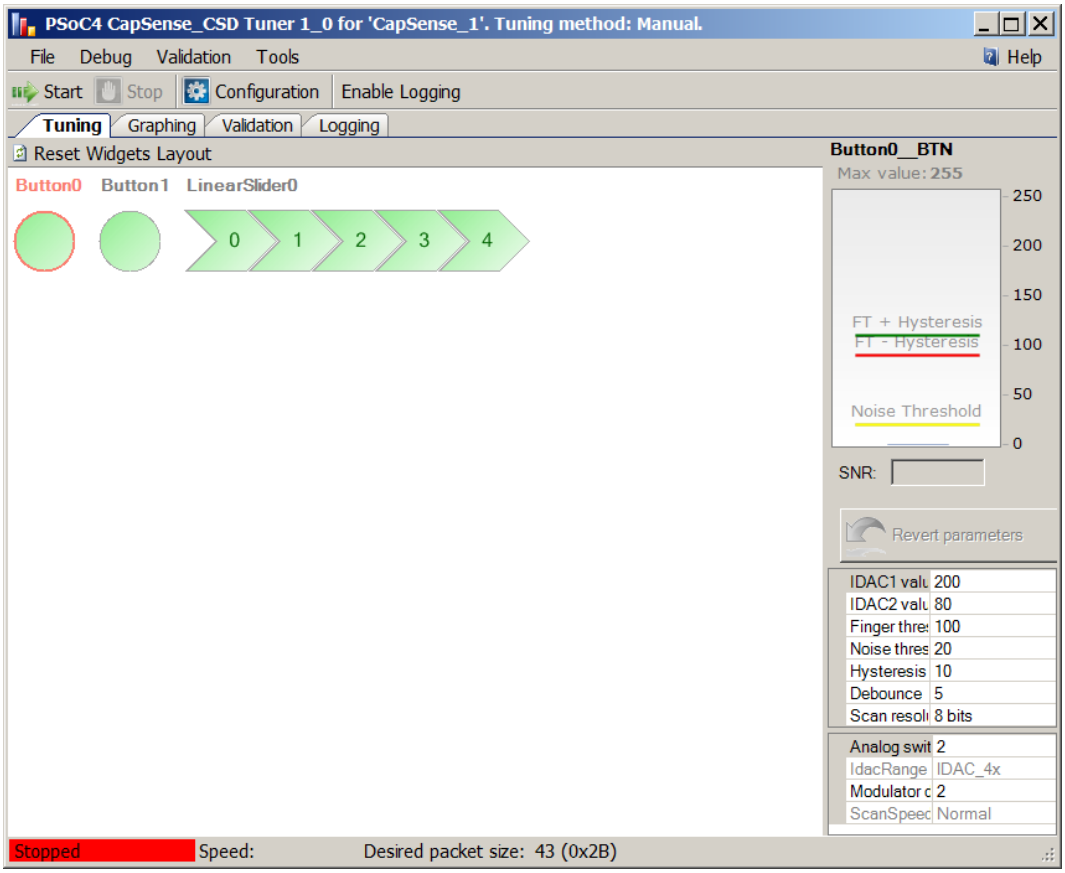
$$\text{SNR} = \frac{\text{Peak Finger Response}}{\text{Peak to Peak Noise when finger is not present}}$$

11. For a good CapSense design, the SNR should be above 5. Check whether the total scan time fits the design. If the SNR requirement is not met, add firmware filters. See the Filters section and select the type of filter that suits the noise present in the system. For most designs, start with the First Order IIR 1/4 filter because it requires minimal SRAM and gives a fast response.
12. Check the SNR as in step 8. Also, check whether the scan time requirement for the design is met. The tuned values inside the GUI are updated in the component when the **OK** button is clicked on the GUI. The scan time approximation is calculated by the component based on the parameter settings. The scan time is shown in the scan order tab. If the design has many sensors that have high resolution and low scan speed then the total scan time for all the sensors will result in a long scan interval for the sensors.

    If the SNR is below 5, increase the resolution, the scan speed, or both. By doing this, the scan time increases. Therefore, the resolution and scan time both should be tuned to achieve the SNR above 5 and keep the scan time below design spec. Recheck the SNR and scan. If you cannot achieve the SNR of 5:1 and keep the scan time within the design spec, look for improvements in the PCB layout or overlay design. Refer to *Getting Started with CapSense* for PCB design guidelines. You can also reduce the overlay thickness or increase the button diameter, which increases the sensitivity.
13. After SNR of 5:1 is achieved, set the following firmware parameters.
    - ❑ The Finger Threshold is the parameter the firmware uses as the threshold to determine whether the sensor is active or not. Set this parameter to 75 percent of the peak finger response.
    - ❑ Set the hysteresis to 15 percent of peak finger response.
    - ❑ Set the noise threshold to 40 percent of the finger response.
    - ❑ Debouncing ensures that high-frequency, high-amplitude noise such as an ESD event does not cause button activation. The debouncing value should be a small number such as 1 or 2, because the spike or high frequency noise that can trigger a false button touch will not be as wide as two scan lengths. In fast scanning designs, the debounce value should be set to a higher value such as 5.

Tuner GUI Interface

General Interface



The top panel buttons are as follows:

- **Start** (or main menu item **Debug > Start**) – Starts reading and displaying data from the chip. Also starts graphing and logging if configured.
- **Stop** (or main menu item **Debug > Stop**) – Stops reading and displaying data from the chip.
- **Configuration** (or main menu item **Debug > Configuration**) – Opens the **Communication Configuration** dialog.
- **Enable Logging** (or main menu item **Debug > Start**) – Enables logging of data received from the device to a log file.

Main Menu:

- **File > Settings > Load Settings from File** – Imports settings from an XML tuning file and loads all data into the Tuner.

- **File > Help** – Opens help file.

  Other items duplicate the functionality of top and bottom panel buttons.

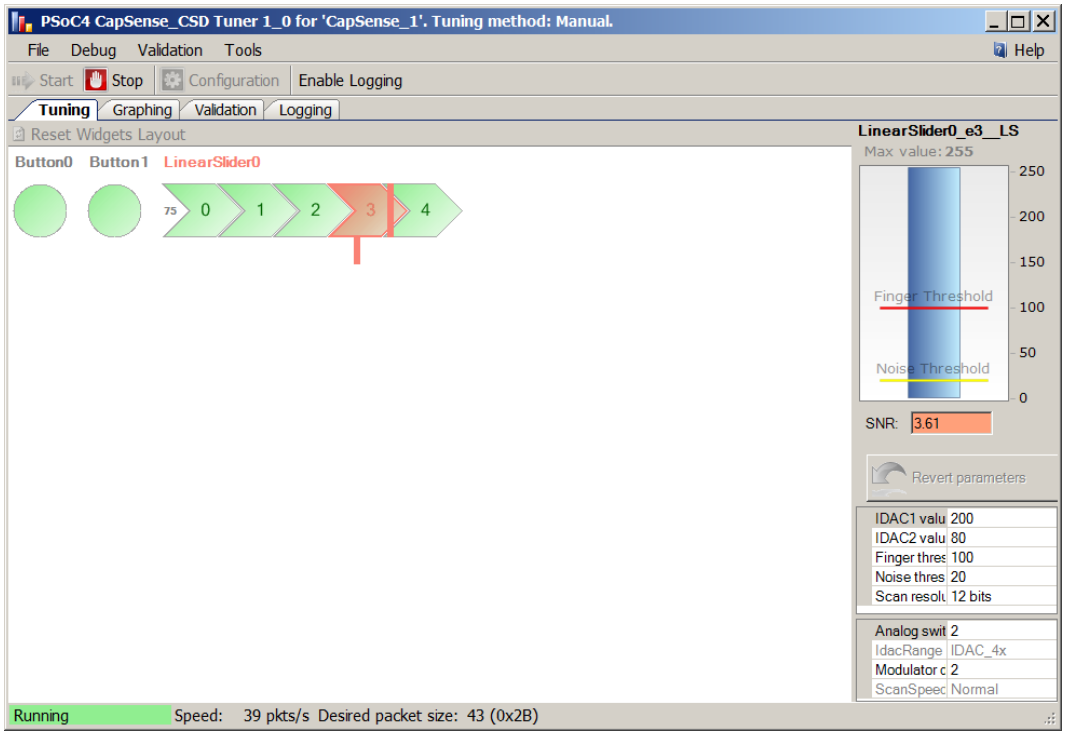
Tabs:

- **Tuning** – Displays all of the component widgets as configured on a workspace. This allows you to arrange the widgets similarly to the way they appear on the physical PCB or enclosure. This tab is used for tuning widget parameters and visualizing widgets data and states.
- **Graphing** – Displays detailed individual widget data on charts.
- **Logging** – Provides logging data functionality and debugging features.

Bottom panel buttons:

- **OK** (or main menu item **File > Apply Changes and Close**) – Commits the current values of parameters to the CapSense component instance and exits the GUI.
- **Cancel** (or main menu item **File > Exit**) – Exits the GUI without committing the values of parameters to the component instance.

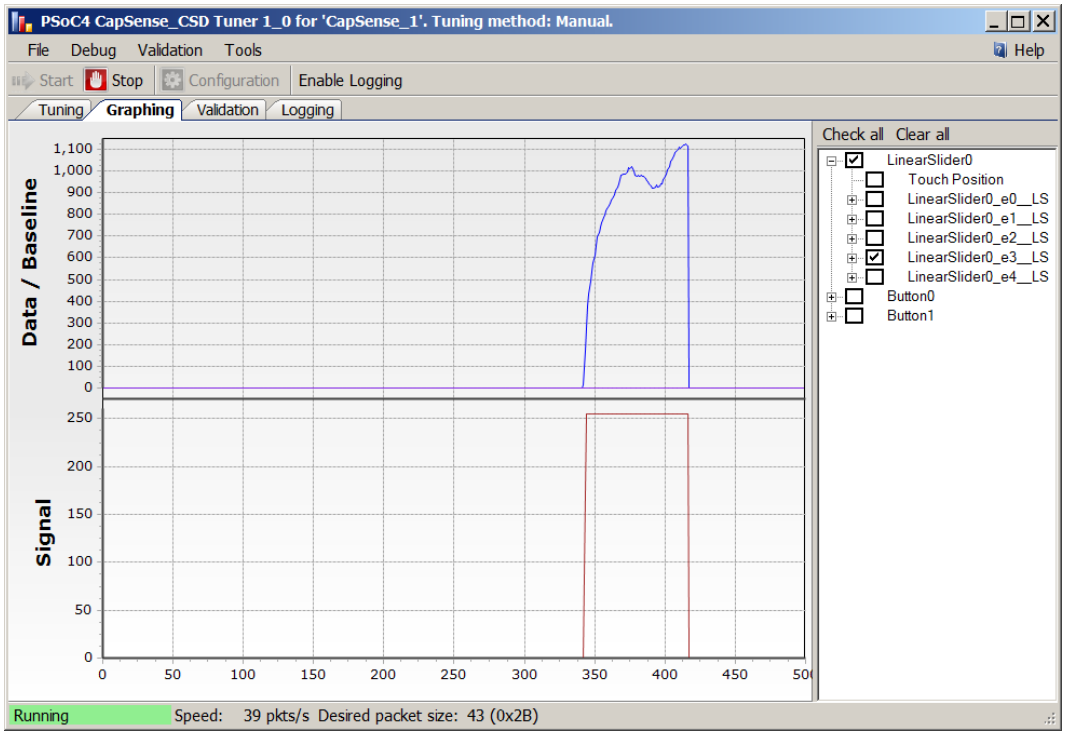
Tuning Tab



- **Widgets schematic** – Contains a graphical representation of all of the configured widgets. If a widget is composed of more than one sensor the individual sensors may be selected for detailed analysis. Every widget is movable within the schematic.
- **Reset Widgets Layout** button – Moves widgets to default positions within the schematic.
- **Bar graph** – Displays signal values for the selected sensor.
  - The maximum scale of the detailed view bar graph can be adjusted by double-clicking on Max Value label. Valid range for 8 bit Widget Resolution is between 1 and 255, default is **255**. Valid range for 16 bit Widget Resolution is between 1 and 32767, default is **32767**.
  - The current finger turn on threshold is displayed as a **green line** across the bar graph.
  - The current finger turn off threshold is displayed as a **red line** across the bar graph.
  - The current noise threshold is displayed as a **yellow line** across the bar graph.
- **SNR** – The signal-to-noise ratio is computed in real time for the selected sensor. SNR values below 5 are poor and colored red, 5 to 10 are marginal and yellow, and greater than 10 is good and colored green. SNR value is calculated based on previously received data.
- **Revert Parameters** button – Resets the parameters to their initial values and sends those values to the chip. Initial values are what were displayed when the GUI was launched.
- **Sensor properties** – Displays the properties for the selected sensor based on the widget type. It is located on the right side panel.
- **General CapSense properties** (read only) – Displays global properties for the CapSense CSD component that cannot be changed at run time. These are for reference only. This information is located on the bottom of the right-side panel.
- **Widget controls context menu** (this functionality applies only to the layout of widget controls in GUI):
  - **Send To Back** – Sends widget control to the back of the view.
  - **Bring To Front** – Brings widget control to the front of the view.
  - **Rotate Clockwise 90** – Rotates widget control 90 degrees clockwise. (Only for Linear Sliders)
  - **Rotate Counter Clockwise 90** – Rotates widget control 90 degrees counter clockwise. (Only for Linear Sliders)
  - **Flip Sensors** – Reverses the order of the sensors. (Only for Linear and Radial Sliders)
  - **Flip Columns Sensors** – Reverses the order of the Columns sensors. (Only for Touchpads and Matrix Buttons)
  - **Flip Row Sensors** – Reverses the order of the Row sensors. (Only for Touchpads and Matrix Buttons)

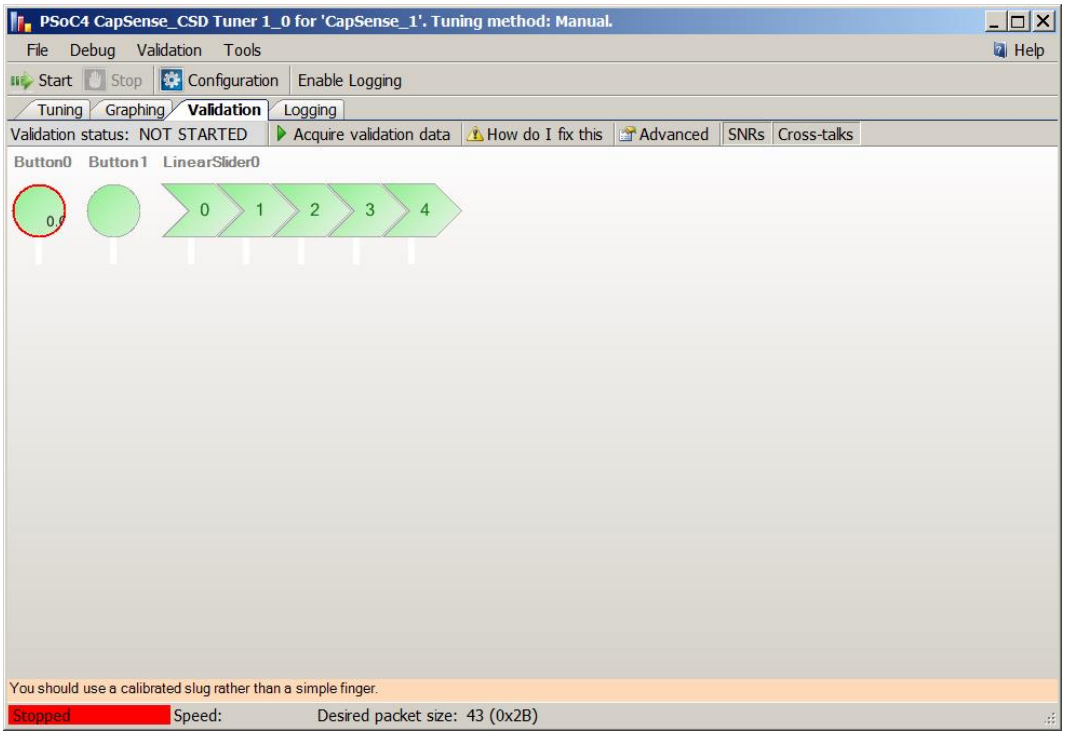
- **Exchange Columns and Rows** – Columns sensors become rows and rows sensors become columns. (Only for Touchpads and Matrix Buttons)

Graphing Tab



- **Chart area** – Displays charts for selected items from the tree view. If you right-click the menu item **Export to .jpg**, you can generate a screenshot of the chart area that is saved as a .jpg file.
- **Tree view** – Gives all combinations of data for widgets and sensors which can be shown on the chart and logged to a file if the logging feature is enabled. The On/Off Status data value can only be logged, it cannot be shown on a chart.

Validation Tab



The **Validation** tab is for diagnostics only. The tab contains the widget layout view, but without the ability to edit the layout. This layout portion is used as a display only.

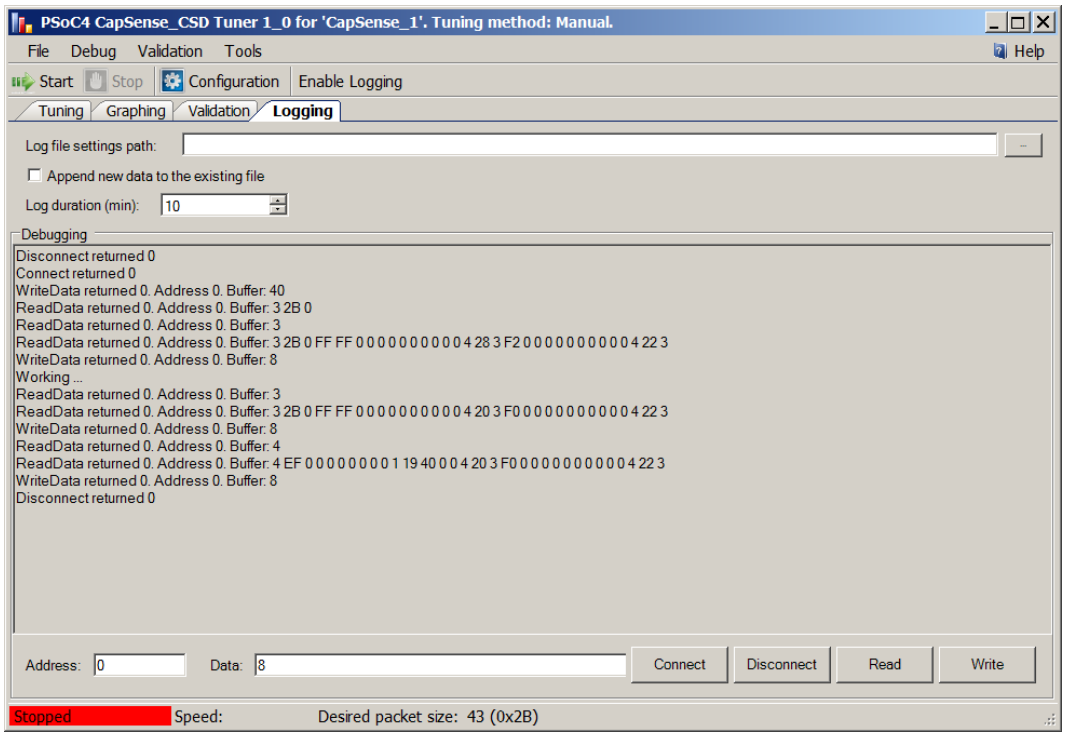
- **Widgets schematic** – Contains a graphical representation of all of the configured widgets.

Top panel controls:

- **Validation Status** label – Shows validation status. It has following messages:
  - **VALIDATION NOT STARTED** – The validation process has not been run since the last time the design was changed.
  - **PASS** – The full validation process has been completed without failures.
  - **FAIL** – The validation process has uncovered failures; a validation report will be displayed.
- **Acquire Validation Data** button (or main menu item **Validation > Acquire Validation Data**) – Starts validation process. This process guides you through a sequence of operations in which you are prompted to apply your finger to each sensor in sequence.
- **How do I fix this** button – Opens a report with a list of suggested fixes for sensors that did not pass validation. This button is available only if the validation process was previously completed and design errors were found.

- **Advanced button** (or main menu item **Validation > Validation Advanced properties**) – Opens the properties window for validation properties (for more information, see Validation Advanced Properties).
- **SNRs** button – In the widget schematic, turns the SNR display on or off (for more information, see Validation Displays).
- **Crosstalks** button – In the widget schematic turns crosstalk display on or off (for more information, see Validation Displays).

Logging Tab



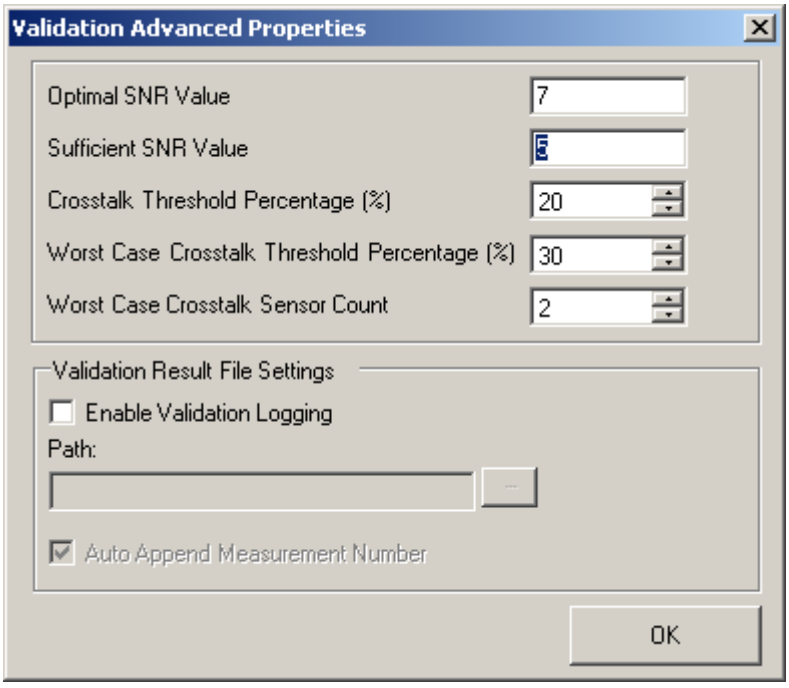
- Data that will be logged is indicated by selecting check boxes on the Tree View of the **Graphing** tab.
- **Path** – Defines log file path (file extension is .csv).
- **Append new data to existing file** check box – If selected, new data is appended to an existing file. If not selected, old data is erased from the file and replaced with the new data.
- **Log duration** – Defines log duration in minutes. Valid range is between 1 and 480; the default is **255**.

Debugging group

This functionality exists only for debugging purposes. It helps you investigate Tuner communication errors.

- Debugging log window – Displays communication commands that the Tuner executes. All communication errors are logged here. If the Tuner was successfully started, only the first few communication commands are logged.
- **Connect** – Connects to the PSoC device.
- **Disconnect** – Disconnects from the PSoC device.
- **Address** – Specifies the PSoC device address.
- **Read** – Reads data from the PSoC device. The address field defines the address in the buffer. The data field defines number of bytes to read.
- **Write** – Writes data to the PSoC device. The address field defines the address in the buffer. The data field defines the data to write.

Validation Advanced Properties



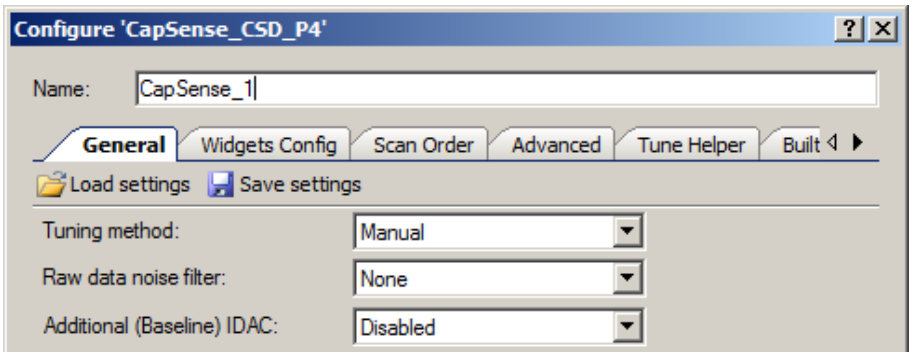
- **Optimal SNR Value** – Defines optimal SNR value. Valid range is between 0 and 100; default is **7**.
- **Sufficient SNR Value** – Defines sufficient SNR value. Valid range is between 0 and 100; default is **5**.

- **Crosstalk Threshold Percentage (%)**– Defines crosstalk threshold value as a percentage of the finger threshold for each sensor. Valid range is between 0 and 100 percent; default is **20**.
- **Worst Case Crosstalk Threshold Percentage (%)** – Defines worst case crosstalk threshold value as a percentage of worst case crosstalk. Valid range is between 0 and 100 percent; default is **30**.
- **Worst Case Crosstalk Sensor Count** – Defines the number of sensors used to compute worst case crosstalk; valid range is between 0 and 100; default is **2**.
- **Enable Validation Logging** – Enables logging of validation data.
- **Path** – Defines log file path for validation data (file name extension is .csv).
- **Auto Append Measurement Number** check box – If selected, after each start of the validation process, the log file name will be incremented (for example “validation001.csv”) and data will be saved in a new file.

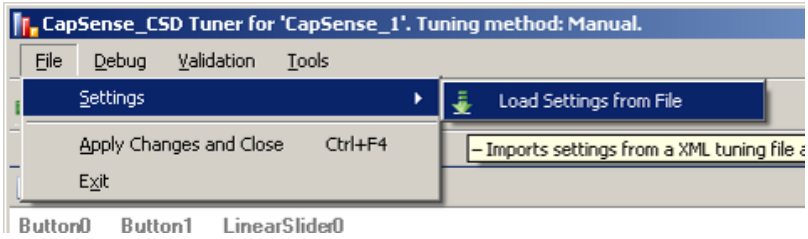
Save/Load Settings Feature

The Tuner GUI can also be opened as standalone application. In this case you must use the Save and Load Settings feature of the CapSense CSD component Tuner GUI.

1. Click the **Save Settings** button in the customizer.



2. In the **Save File** dialog box, specify name of the file and location where it will be saved.
3. Open the Tuner window and click **File > Settings > Load Settings from File**.



4. In the **File Open** dialog box, point to the previously saved file with the component settings. Settings will automatically load into the Tuner.

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.

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.
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 nonretention registers, and resets all sensors to an inactive state.
CapSense_Wakeup()	Restores CapSense configuration and nonretention 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. The CapSense_Start() routine must be called before any other API routines.

Parameters: None

Return Value: None

Side Effects: None

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. To change a pin's drive mode, use the functions described in the section.

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. To change a pin's drive mode, use the functions described in the section

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_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.

void CapSense_ScanSensor(uint8 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: uint8 sensor: Sensor number

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.

uint8 CapSense_IsBusy (void)

Description: Returns the status of sensor scanning.

Parameters: None

Return Value: uint8: Returns the state of scanning. ‘1’ – scanning in progress, ‘0’ – scanning completed.

Side Effects: None

void CapSense_SetScanSlotSettings(uint8 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: uint8 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(uint8 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: uint8 sensor: Sensor number

Return Value: None

Side Effects: None

void CapSense_DisableSensor(uint8 sensor)

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

Parameters: uint8 sensor: Sensor number

Return Value: None

Side Effects: None

uint16 CapSense_ReadSensorRaw(uint8 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: uint8 sensor: Sensor number

Return Value: uint16: Current raw data value

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.

Function	Description
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.
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.

void CapSense_InitializeSensorBaseline(uint8 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: uint8 sensor: Sensor number

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(uint8 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: uint8 sensor: Sensor number

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(uint8 widget)

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

Parameters: uint8 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.

Return Value: None

Side Effects: None

void CapSense_DisableWidget(uint8 widget)

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

Parameters: uint8 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.

Return Value: None

Side Effects: None

uint8 CapSense_CheckIsWidgetActive(uint8 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: uint8 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.

Return Value: uint8: 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.

uint8 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: uint8: 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(uint8 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: uint8 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.

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(uint8 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: uint8 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.

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.

uint8 CapSense_GetTouchCentroidPos(uint8 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.
```

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

Side Effects:

uint8 CapSense_GetMatrixButtonPos(uint8 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.
```

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

Side Effects:

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: None

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: None

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:

- CapSense_SensorRaw [ ]
- CapSense_SensorEnableMask [ ]
- CapSense_portTable[ ] and CapSense_maskTable[ ]
- CapSense_SensorBaseline [ ]
- CapSense_SensorBaselineLow[ ]
- CapSense_SensorSignal [ ]
- CapSense_SensorOnMask[ ]

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()

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()

CapSense_SensorOnMask[ ]

This is a byte array that holds the sensor's on/off state.

CapSense_SensorOnMask[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). CapSense_SensorOnMask[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 is 1 if the sensor is on (active) and 0 if the sensor is off (inactive). The CapSense_SensorOnMask[ ] data is updated by functions:

- CapSense_CheckIsWidgetActive()
- CapSense_CheckIsAnyWidgetActive()

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. These constants can be used as parameters in the following functions:

- CapSense_EnableSensor()
- CapSense_DisableSensor()

The constant names consist of:

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

For example:

```
#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
MB	Matrix Buttons
PROX	Proximity Sensors
GEN	Generic Sensors
GRD	Guard Sensor

Widget Constants

A constant is provided for each widget. These constants can be used as parameters in the following functions:

- CapSense_CheckIsWidgetActive()
- CapSense_EnableWidget() and CapSense_DisableWidget()
- CapSense_GetCentroidPos()
- CapSense_GetRadialCentroidPos()
- CapSense_GetTouchCentroidPos()

The constants consist of:

Instance name + *Widget Name* + *Widget Type*

For example:

```
#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. 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.

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 – CapSense_cPort – Pin Assignment

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

CapSense_cCmod_Port – Pin Assignment

One side of the external modulator capacitor (C_{MOD}) should be connected to a physical pin and the other to GND. The C_{MOD} can be connected to P4[2] **pin**.

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.

Functional Description

Definitions

Sensor

One CapSense element connected to PSoC via one pin. A sensor is a conductive element on a substrate. Examples of sensors include: Copper on FR4, Copper on Flex, Silver ink on PET, ITO on glass.

Scan Time

A scan time is a period of time that the CapSense module is scanning one or more capacitive sensors. Multiple sensors can be combined in a given scan sensor to enable modes such as proximity sensing.

CapSense Widget

A CapSense widget is built from one or more scan sensors to provide higher-level functionality. Some examples of CapSense Widgets include buttons, sliders, radial sliders, touchpads, matrix buttons, and proximity sensors.

FingerThreshold

This value is used to determine if a finger is present on the sensor.

NoiseThreshold

Determines the level of noise in the capacitive scan. The baseline algorithm filters the noise in order to track voltage and temperature variations in the sensor baseline value.

Debounce

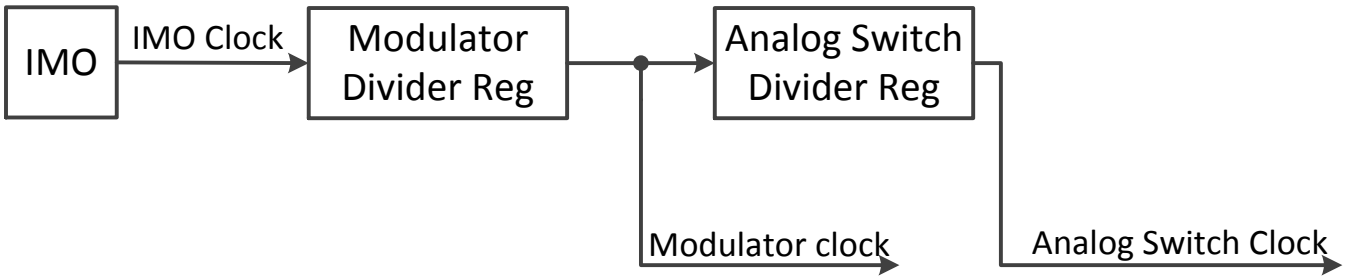
Adds a debounce counter to the sensor active transition. For the sensor to transition from inactive to active, the difference count value must stay above the finger threshold plus hysteresis for the number of samples specified. This is necessary to filter out high-amplitude and frequency noise.

Hysteresis

Sets the hysteresis value used with the finger threshold. If hysteresis is desired, the sensor will not be considered "On" or "Active" until the count value exceeds the finger threshold plus the hysteresis value. The sensor will not be considered "Off" or "Inactive" until the measured count value drops below the finger threshold minus the hysteresis value.

CapSense Clocking

The CapSense clocking tree is showed on figure below.



The Modulator clock is formed by dividing the IMO Clock by the Modulator Divider.

The Analog Switch Clock is formed by dividing the Modulator Clock by the Analog Switch Divider.

For example, if you configure the Analog Switch Divider value to 8 and the Modulator Divider value to 4, then the Modulator Divider Reg will be configured to dividing by 4 and Analog Switch Divider Reg will configured to dividing by 2.

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.

$$N_{Cent} = \frac{n_{i-1}(i-1) + n_i i + n_{i+1}(i+1)}{n_{i-1} + n_1 + n_{i+1}}$$

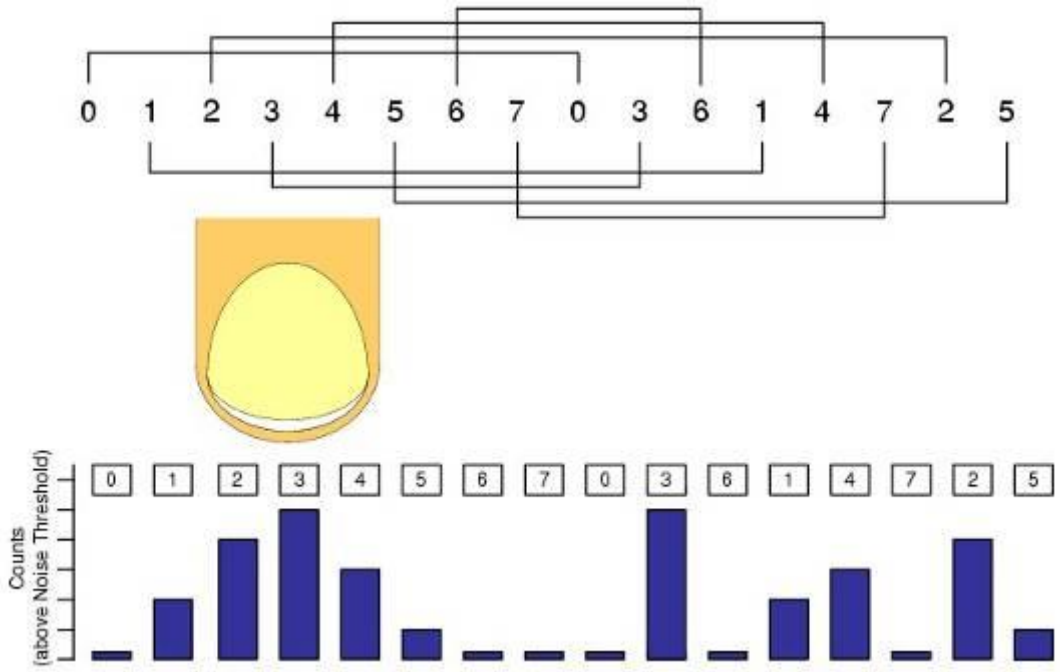
The calculated value is typically fractional. In order to report the centroid to a specific resolution, for example a range of 0 to 100 for 12 sensors, the centroid value is multiplied by a scalar. It is more efficient to combine the interpolation and scaling operations into a single calculation and 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 index 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

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

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:

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

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

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

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

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

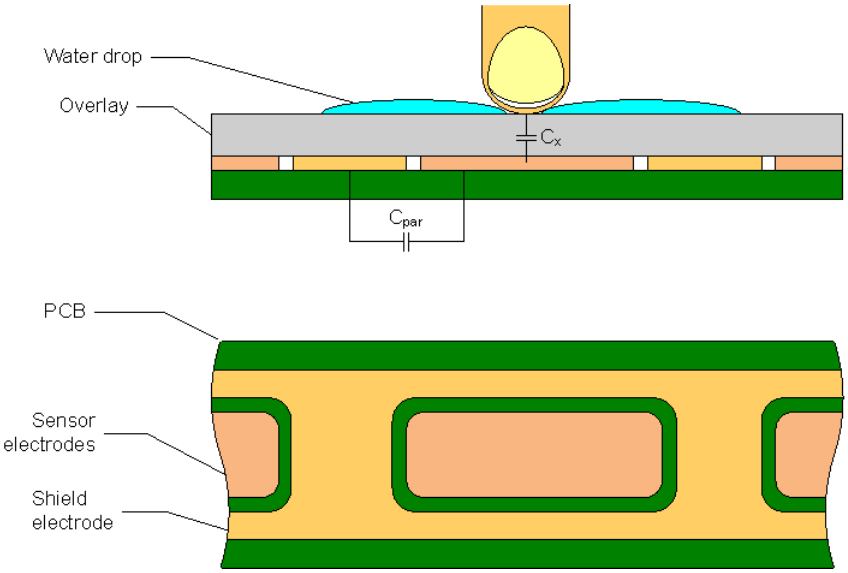


Figure 2 illustrates one possible layout configuration for the button's shield electrode. The shield electrode is especially useful for transparent ITO touchpad devices, where it blocks the LCD drive electrode's noise and reduces stray capacitance at the same time.

In this example, the button is covered by a shielding electrode plane. As an alternative, the shielding electrode can be located on the opposite PCB layer, including the plane under the button. A hatch pattern is recommended in this case, with a fill ratio of about 30 to 40 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.

Resources

The CapSense CSD utilizes the following digital and analog 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
Baselining IDAC disabled	1	0
Baselining IDAC enabled	1	1
SmartSense	1	1

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 4 (GCC)	
	Flash Bytes	SRAM Bytes
Tuning method: Auto (SmartSense) Current source: IDAC sourcing Raw data filter : Averaging Baselining IDAC: Enabled Waterproof: Disabled	2042	812
Tuning method: None Current source: IDAC sinking Raw data filter : None Baselining IDAC: Enabled Waterproof: Disabled	1904	278
Tuning method: None Current source: IDAC sourcing Raw data filter : None Baselining IDAC: Enabled Waterproof: Disabled	1904	278

Configuration	PSoC 4 (GCC)	
	Flash Bytes	SRAM Bytes
Tuning method: None Current source: IDAC sourcing Raw data filter : None Baselining IDAC: Enabled Waterproof: Enabled	1956	287
Tuning method: None Current source: IDAC sourcing Raw data filter : None Baselining IDAC: Disabled Waterproof: Enabled	1752	287

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
VCSD	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	Capacitance range of 9 to 32 pF, 0.1pF sensitivity

Component Changes

Version	Description of Changes	Reason for Changes / Impact
1.10.a	Minor datasheet edit.	
1.10	The scan time is 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, 2013-2016. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC (“Cypress”). This document, including any software or firmware included or referenced in this document (“Software”), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress’s patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage (“Unintended Uses”). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.

