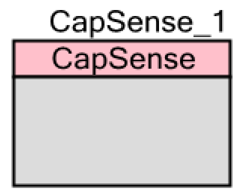
PSoC 4 Capacitive Sensing (CapSense®)

5.0



Features

- Offers best-in-class signal-to-noise ratio (SNR)
- Supports Self-Capacitance (CSD) and Mutual-Capacitance (CSX) sensing methods
- Features SmartSense™ auto-tuning technology for CSD sensing to avoid complex manual tuning process
- Supports various Widgets, such as Buttons, Matrix Buttons, Sliders, Touchpads, and Proximity Sensors
- Provides ultra-low power consumption and liquid tolerant capacitive sensing technology
- Contains integrated graphical Tuner GUI tool for real-time tuning, testing, and debugging
- Provides superior immunity against external noise and low radiated emission.
- Offers best-in-class liquid tolerance
- Contains built-in self-test (BIST) library for implementing Class-B requirements for CapSense
- Supports one-finger and two-finger gestures

Note This PSoC 4 CapSense v5.X Component (and any version that follows) is a new Component and it is **not** backward-compatible with CapSense_CSD v2.X or older. If you are working on a project with an older Component, Cypress recommends backing it up before replacing the Component in your schematic. The C code written for CapSense_CSD_P4 is not compatible with the new Component. Refer to the *Migration Guide* section in this datasheet for details on how the two Components are different.

General Description

CapSense is a Cypress capacitive sensing solution. Capacitive sensing can be used in a variety of applications and products where conventional mechanical buttons can be replaced with sleek human interfaces to transform the way users interact with electronic systems. These include home appliances, automotive, IoT, and industrial applications. CapSense supports multiple interfaces (widgets) using both CSX and CSD sensing methods, with robust performance.

This CapSense Component solution includes a configuration wizard to create and configure CapSense widgets, API to control the Component from the application firmware, and a *CapSense Tuner* application for tuning, testing, and debugging for easy and smooth design of human interfaces on customer products. This datasheet includes the following sections:

- *Quick Start* – Helps you quickly configure the Component to create a simple demo.
- *Component Configuration Parameters* – Contains descriptions of the Component's parameters in the configuration wizard.
- *Application Programming Interface* – Provides descriptions of the API in the firmware library, as well as descriptions of the data structures (Register map) used by the firmware library.
- *CapSense Tuner* – Contains descriptions of the user-interface controls in the Tuner application.
- *Electrical Characteristics* – Provides the Component performance specifications and other details such as certification specifications.
- *Migration Guide* – Helps to manually transition designs from CapSense_CSD_P4 v2.X or older versions to CapSense v5.X.

Note Important information such as the CapSense-technology overview, appropriate Cypress device for the design, CapSense system and sensor design guidelines, as well as different interfaces and tuning guidelines necessary for a successful design of a CapSense system is available in the *Getting Started with CapSense®* document and the product-specific *CapSense design guide*. Cypress highly recommends starting with these documents. They can be found on the Cypress web site at *www.cypress.com*. For details about application notes, code examples, and kits, see the *References* section in this datasheet.

When to Use a CapSense Component

CapSense has become a popular technology to replace conventional mechanical- and optical-based user interfaces. There are fewer parts involved, which saves cost and increases reliability, with no wear-and-tear. The main advantages of CapSense compared with other solutions include:

- robust performance in harsh environmental conditions
- rejection of a wide range of external noise sources

Use CapSense for:

- Touch and gesture detection for various interfaces
- Proximity detection for innovative user experiences and low-power optimization
- Replacement for IR-based proximity detection which is sensitive to skin and colors
- Contactless liquid level sensing in a variety of applications
- Touch free operations in hazardous materials

Limitations

This Component supports all CapSense-enabled devices in the PSoC 4 family of devices, including:

- Third-generation CapSense: PSoC 4000, PSoC 4100, PSoC 4200, PSoC 4100M, PSoC 4200M, PSoC 4200L, PSoC 4100 BLE, PSoC 4200 BLE, and PRoC BLE.
- Fourth-generation CapSense: PSoC 4000S, PSoC 4100S, PSoC 4100S Plus, and PSoC Analog Coprocessor.

However, some features are restricted:

- This version of the Component supports gesture detection on one widget at a time.
- The second hardware CSD block is not supported in PSoC 4100M / PSoC 4200M devices.
- The *CSX sensing method* is not supported in PSoC 4100 devices.

Note Component operation is dependent on a high-frequency (system clock) input to the block. Changing the clock frequency during run-time will impact Component operation, and the Component may not operate as expected.

Quick Start

This section will help you create a PSoC Creator project with a *Linear Slider* interface using the *CSD sensing method*. After creating the project, refer to the *Tuning Quick Start with EzI2C* section for information on how to to monitor sensor performance using the *CapSense Tuner*.

Note The *CY8CKIT-042 PSoC® 4 Pioneer Kit* with PSoC 4200 devices include a built-in linear slider.

As needed, refer to the following documents for more information about PSoC Creator:

- *Quick Start Guide*
- *PSoC Creator Help*

Step 1: Create Design in PSoC Creator

Create a project using PSoC Creator and select the desired CapSense-enabled PSoC 4 device from the drop-down menu in the New Project wizard.

Step 2: Place and Configure CapSense Component

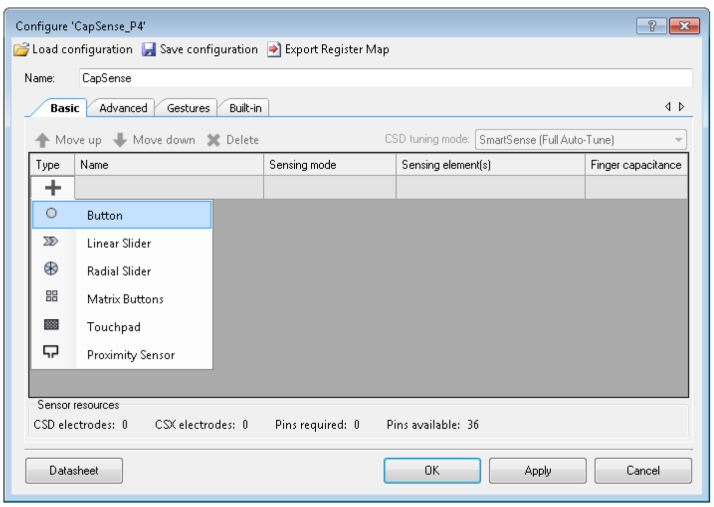
Drag and drop the CapSense Component from the Component Catalog onto the design to add the CapSense functionality to the project.

Double-click on the dropped Component in the schematic to open the Configure dialog.

The *Component Configuration Parameters* are arranged over the multiple tabs and sub-tabs.

Basic Tab

1. Use this tab to select the *Widget Type*, *Sensing mode*, and a number of *Widget Sensing element(s)* required for the design.
2. Type the desired Component name (in this case: CapSense for the code in *Step 4* to work).
3. Click '+' and select the Widget Type required from the drop-down list. This Component offers six different types of widgets.



4. Add the *Linear Slider* widget.

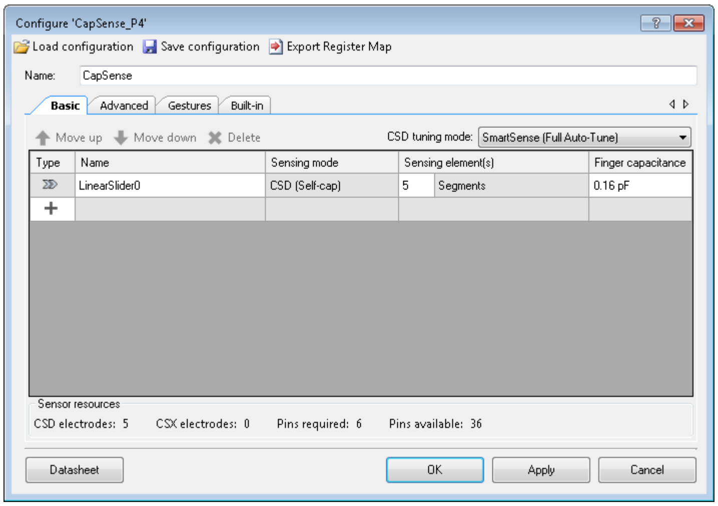
   Note Each widget consumes a specific set of port pins from the device. The number of *Pins required* should always be less than or equal to *Pins availabl* in the selected device to successfully build a project.

5. Use the *CSD tuning mode* pull-down menu to select one of the following options:
   - *SmartSense (Full Auto-Tune)* – With full auto-tuning mode, the majority of configuration parameters in the *Advanced Tab* are automatically set by the SmartSense algorithm.

- *SmartSense (Hardware parameters only)*
- *Manual* tuning

Note SmartSense auto-tuning is available for widgets using the *CSD sensing method* only. Widgets that use CSX mode must be configured manually. This example uses *SmartSense (Full Auto-Tune)* tuning mode.

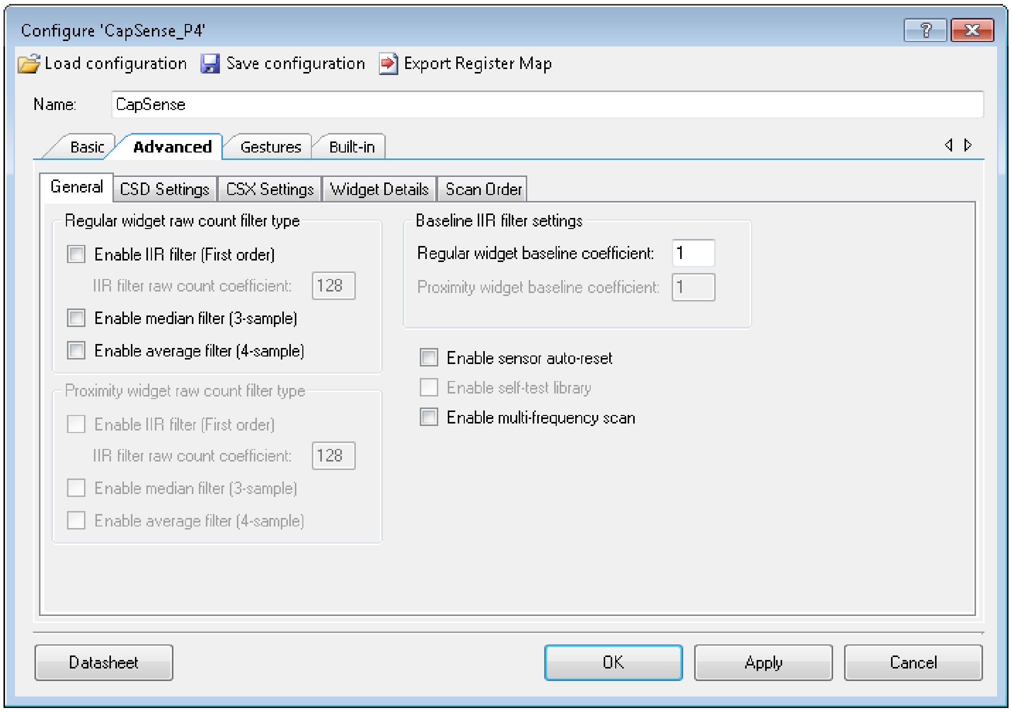
The **Basic** tab contains a table with the following columns:



- *Widget Type* – Shows the selected widget type.
- *Widget Name* – Changes the name of each widget if required (In this example, default name LinearSlider0 is used).
- *Sensing mode* – Selects mode for each widget. This Component supports both Self-cap and Mutual-cap sensing methods for the *Button, Matrix Buttons* and *Touchpad* widgets. (In this example, the default (CSD) sensing mode is used).
- *Widget Sensing element*(s) – Selects a number of sensing elements for each widget. The number of sensing elements is configurable as the application requires (In this example, the default value of 5 is used).
- *Finger capacitance* – Selects Finger capacitance between 0.1pF and 1pF in *SmartSense (Full Auto-Tune)* tuning mode and between 0.02pF to 20.48pF in *SmartSense (Hardware parameters only)* tuning mode to get 50-count signal. Note that this parameter is available for the CSD (Self-cap) *Sensing mode* when *SmartSense (Full Auto-Tune)* mode is enabled.

Advanced Tab

Use this tab to configure parameters required for an extensive level of manual tuning. This tab has multiple sub-tabs used to systematically arrange parameters. Refer to the *Component Configuration Parameters* section for details of these parameters.



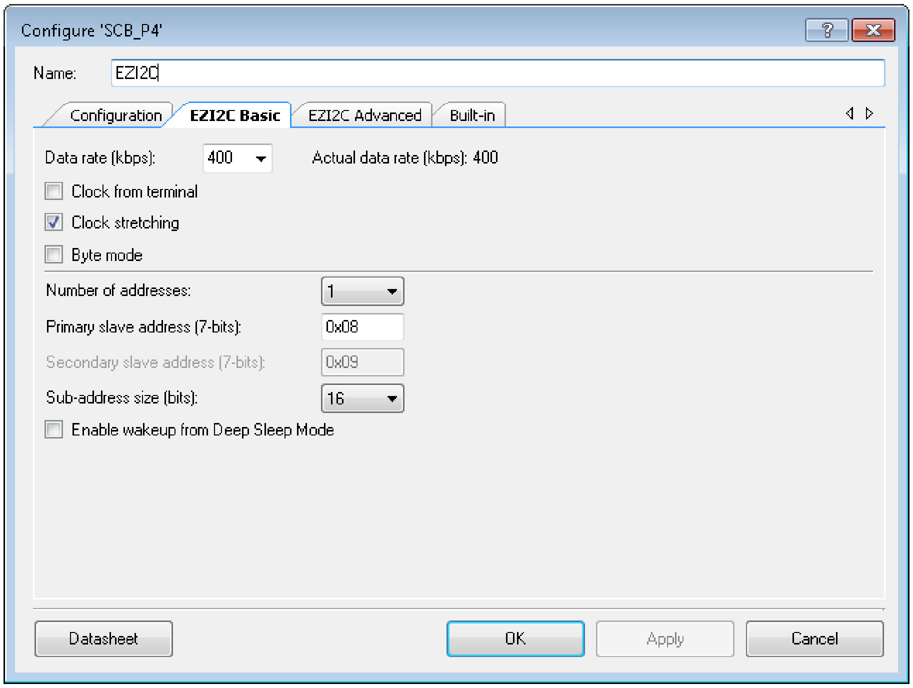
The sub-tabs contain:

- *General* – The parameters common for all widgets in the Component.
- *CSD Settings* – The parameters common for all CSD widgets.
- *CSX Settings* – The parameters common for all CSX widgets.
- *Widget Details* – The parameters specific for each widget and sensing element.
- *Scan Order* – No editable content. It provides scan time for sensors.

Step 3: Place and Configure EZI2C Component

1. Drag an EZI2C Slave (SCB mode) Component from the Component Catalog onto the schematic to add an I^2C communication interface to the project. This I^2C slave interface is required for the Tuner GUI to monitor the Component parameters in real time.

2. Double-click the EZI2C Component in the schematic to open the Configure dialog and set the following parameters:



   - Type the desired Component name (in this case: EZI2C).
   - Set Data Rate (kbps) to 400.
   - Set Number of Addresses to 1.
   - Set Primary Slave Address (7-bits) to 0x08.
   - Set Sub-Address Size (bits) to 16 bits.

3. Click **OK** to close the GUI and save changes.

Step 4: Write Application Code

Copy the following code into the *main.c* file:

```
#include "project.h"

int main()
{
    __enable_irq();                             /* Enable global interrupts. */

    EZI2C_Start();                              /* Start EZI2C Component */
    /*
    *  Set up communication and initialize data buffer to CapSense data structure
    *  to use Tuner application
    */
    EZI2C_EzI2CSetBuffer1(sizeof(CapSense_dsRam),
                          sizeof(CapSense_dsRam),
                     (uint8_t *)&(CapSense_dsRam));

    CapSense_Start();                           /* Initialize Component */
    CapSense_ScanAllWidgets();                  /* Scan all widgets */

    for(;;)
    {
        /* Do this only when a scan is done */
        if(CapSense_NOT_BUSY == CapSense_IsBusy())
        {
            CapSense_ProcessAllWidgets();       /* Process all widgets */
            CapSense_RunTuner();                /* To sync with Tuner application */
            if (CapSense_IsAnyWidgetActive())   /* Scan result verification */
            {
                /* add custom tasks to execute when touch detected */
            }

            CapSense_ScanAllWidgets();          /* Start next scan */
        }
    }
}
```

Note The provided example shows the simplest way of using the Component.

Step 5: Assign Pins in Pin Editor

Open the Pin Editor and assign physical pins for all CapSense sensors and I²C pins.

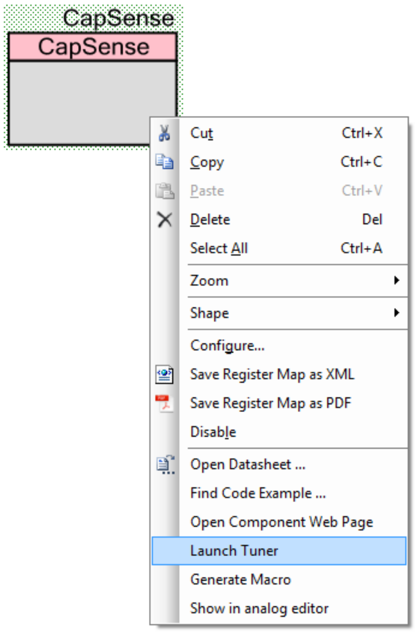
If you are using a Cypress kit, refer to the kit user guide to select the USB-I2C bridge pin. This bridge firmware enables the I²C communication between the PSoC and the Tuner application across the USB. Alternatively, you can use a MiniProg3 debugger/programmer kit as the USB-I2C Bridge.

Step 6: Build Design and Program PSoC Device

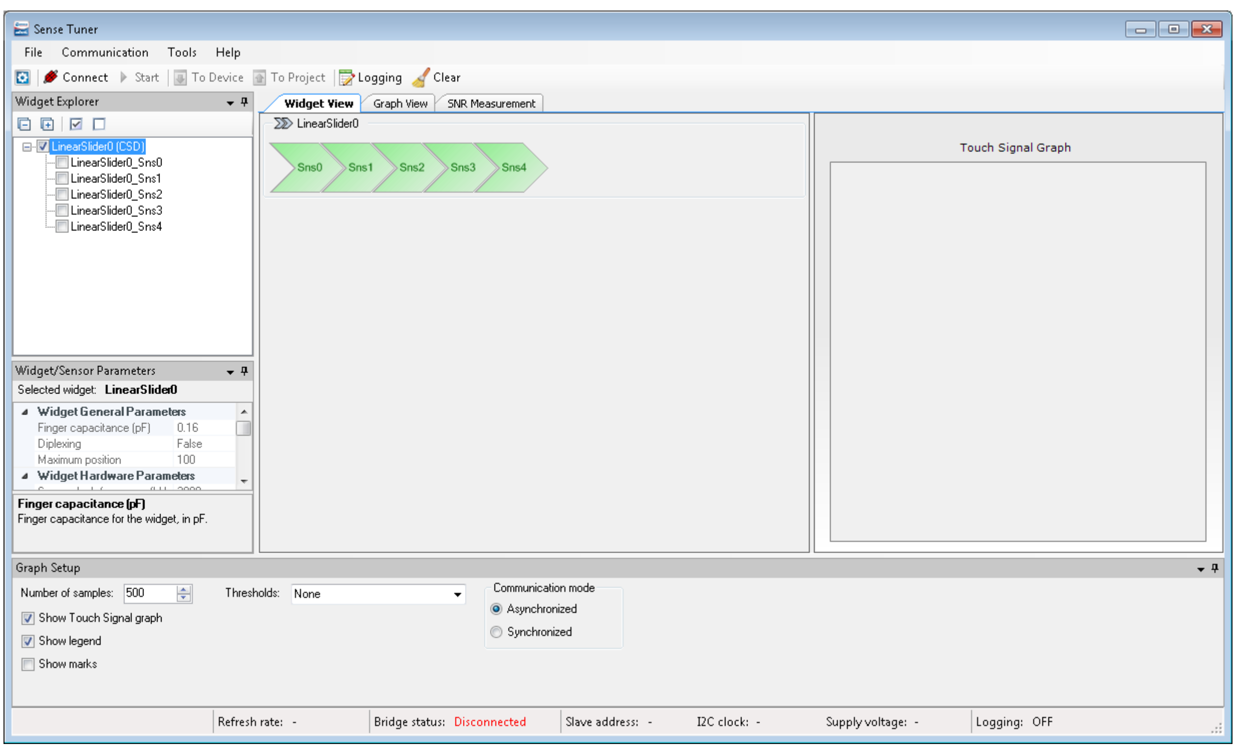
Select ***Program*** from the ***Debug*** menu to download the hex file into the device. This will also perform a build if needed.

Step 7: Launch Tuner Application

Right-click the CapSense Component in the schematic and select **Launch Tuner** from the context menu.



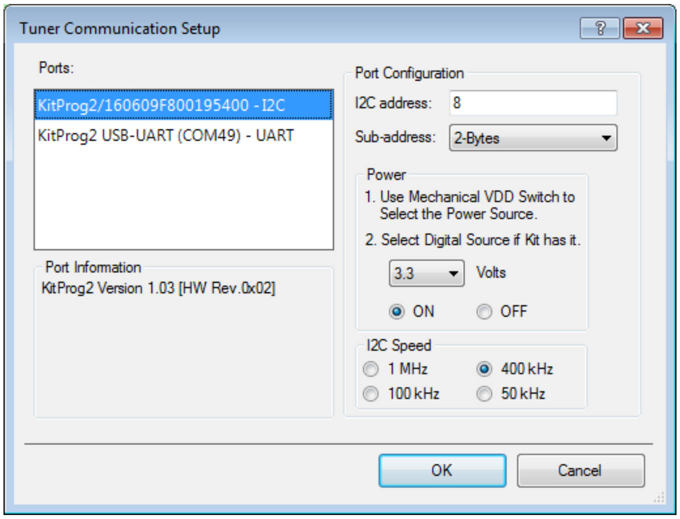
The *CapSense Tuner* application opens as shown. Note that the 5-element slider, called LinearSlider0, appears in the Widget View panel automatically.



Step 8: Configure Communication Parameters

To establish communication between the Tuner and a target device, configure the Tuner communication parameters to match the I^2C Component parameters.

1. Open the Tuner Communication Setup dialog from PSoC Creator by selecting *Tools > Tuner Communication Setup…*



2. Select the appropriate I^2C communication device which is KitProg2 (or MiniProg3) and set the following parameters:
   - **I2C Address**: 8 (or the address set in the EzI2C Component configuration wizard).
   - **Sub-address**: 2 bytes.
   - **I2C Speed**: 400 kHz (or the speed set in the Component configuration wizard).

   Note The I2C address, Sub-address, and I2C speed fields in the Tuner Communication Setup dialog must be identical to the Primary slave address, Sub-address size, and Data Rate parameters in the EZI2C Component Configure dialog (see *Step 3*). Sub-address must be set to 2-Bytes in both places.

Step 9: Start Communication

1. Click **Connect** to establish a connection and then **Start** to extract data.
2. Select the Synchronized control in the Graph Setup Pane. This ensures the Tuner only collects data when CapSense is not scanning. Refer to *Graph Setup Pane* for details about synchronized operation.

The *Status Bar* shows the communication bridge connection status and communication refresh rate. You can see the status of the LinearSlider0 widget in the *Widget View* and signals for each of the five sensors in the *Graph View*. Touch the sensors on the kit to observe the CapSense operation.

Refer to the *CapSense Tuner* section for more details.

Input / Output Connections

This section describes the various input and output connections for the CapSense Component. These do not appear as connectable terminals on the Component symbol but these terminals can be assigned to the port pins in the PSoC Creator Pin Editor. The Pin Editor provides guidelines on the recommended pins for each terminal and does not allow an invalid pin assignment.

Name [1]	I/O Type	Description
C_{mod}[2]	Analog	External modulator capacitor. Mandatory for operation of the CSD sensing method and required only if CSD sensing is used. The recommended value is 2.2nF/5V/X7R or an NP0 capacitor.
$C_{int}A$ [2]	Analog	Integration capacitors. Mandatory for operation of the CSX sensing method and required only if the CSX sensing is used. The recommended value is 470pF/5V/X7R or NP0 capacitors.
$C_{int}B$ [2]	Analog	
C_{sh} [2]	Analog	Shield tank capacitor. Used for an improved shield electrode driver when the CSD sensing is used. This capacitor is optional. The recommended value is 10nF/5V/X7R or an NP0 capacitor.
Shield	Analog	Shield electrode. Reduces the effect of the parasitic capacitance (Cp) of the sensor in the CSD sensing method. The number of shields depends on the user selection in the Component configuration wizard.
Sns	Analog	Sensors of CSD widgets. The number of sensors depends on the CSD widgets selected.
Tx	Digital Output	Transmitter electrodes of CSX widgets. The number of sensors depends on the CSX widgets selected.
Rx	Analog	Receiver electrodes of CSX widgets. The number of sensors depends on the CSX widgets selected.

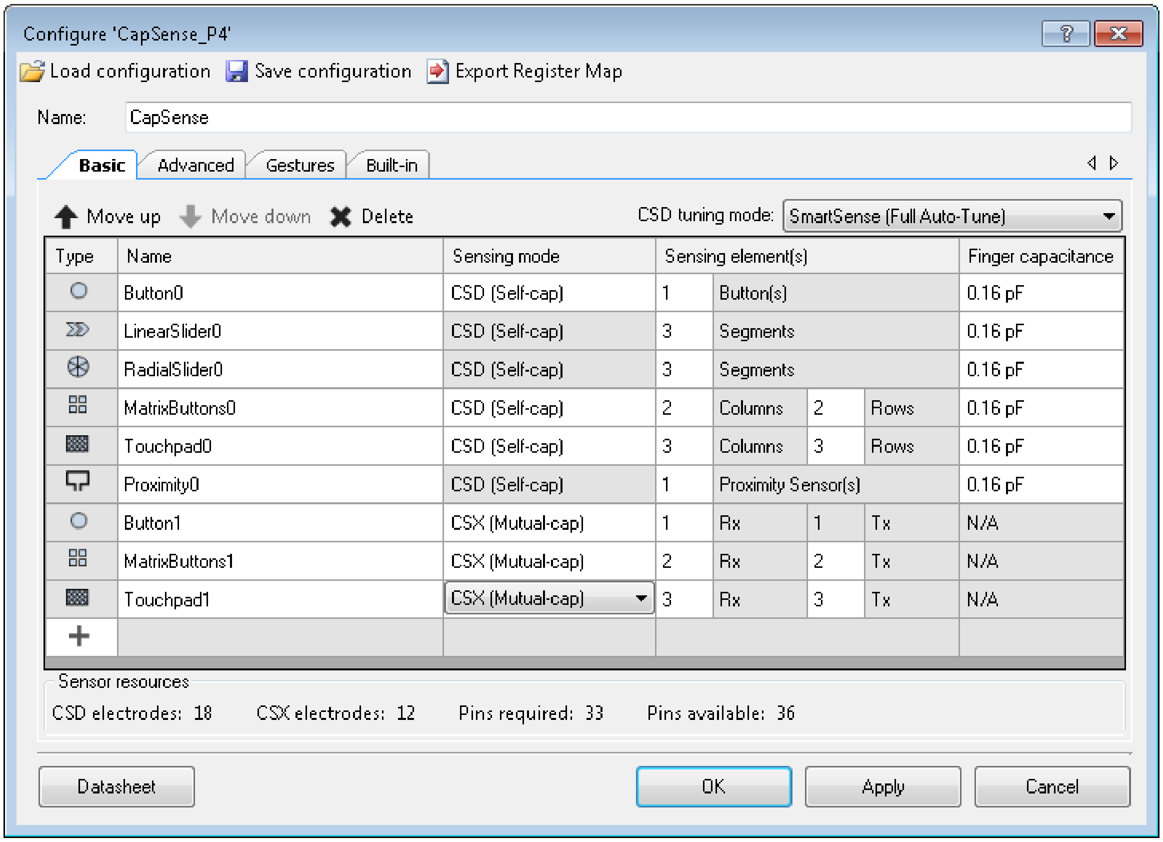
1 No input/output terminals described in the table appear on the Component symbol in the Schematic Editor.

2 The applied rules of restricted placement depend on devices used. For details, refer to the device datasheet or PSoC Creator Pin Editor.

Component Configuration Parameters

This section describes the configurable parameters in the Component Configure dialog. This section does not provide design and tuning guidelines. For complete guidelines on the CapSense system design and CapSense tuning, refer to the *Getting Started with CapSense®* document and the product-specific *CapSense design guide*.

Drag a Component onto the design canvas and double-click to open the dialog.

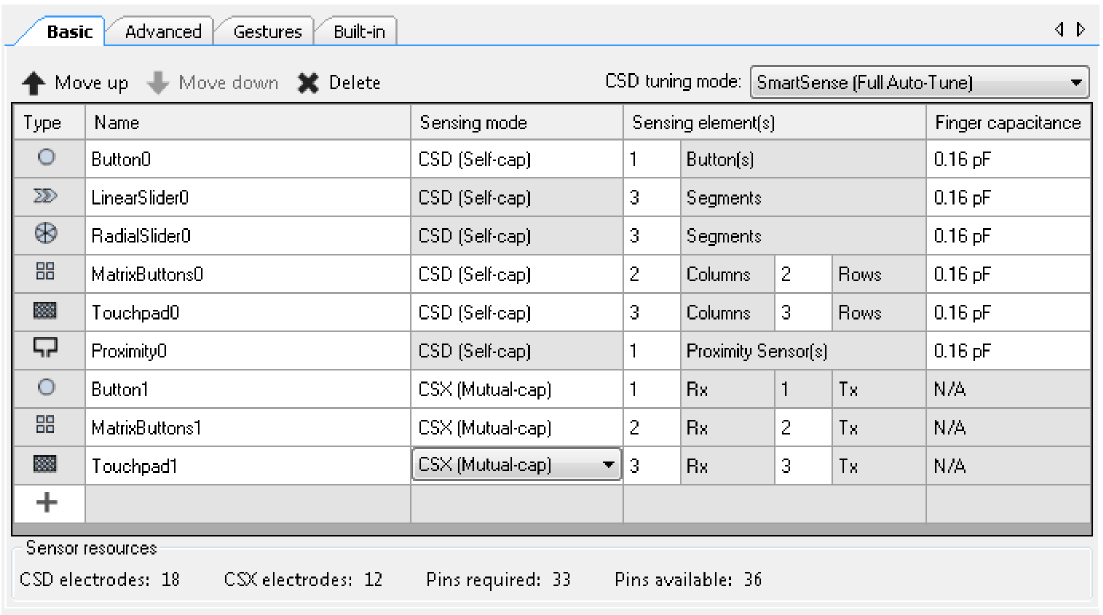


Common Controls

- **Load configuration** – Open (load) a previously saved configuration (XML) file for the CapSense Component.
- **Save configuration** – Save the current Component configuration into a (XML) file.
- **Export Register Map** – The CapSense Component firmware library uses a data structure (known as Register map) to store the configurable parameters, various outputs and signals of the Component. The Export Register Map button creates an explanation for registers and bit fields of the register map in a PDF or XML file that serves as a reference for development.

Basic Tab

The **Basic** tab defines the high-level Component configuration. Use this tab to add various *Widget Type* and assign *Sensing mode*, *Widget Sensing element*(s)*(s)* and *Finger capacitance* for each widget.



The following table provides descriptions of the various **Basic** tab parameters:

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>CSD tuning mode</td><td>
Tuning is a process of finding appropriate values for configurable parameters (Hardware parameters and Threshold parameters) for proper functionality and optimized performance of the CapSense system.
SmartSense Auto-tuning is an algorithm embedded in the Component that automatically finds the optimum values for configurable parameters, based on the hardware properties of the capacitive sensors, therefore avoids the manual tuning process by the user.
Configurable parameters that affect the operation of the sensing hardware are called Hardware parameters. Parameters that affect the operation of the touch-detection firmware algorithm are called Threshold parameters.
This parameter is a drop-down menu to select the tuning mode for CSD widgets only.

SmartSense (Full Auto-Tune) – This is the quickest way to tune a design. Most hardware and threshold parameters are automatically tuned by the Component and the GUI displays them as Set by SmartSense mode. In this mode, the following parameters are automatically tuned:

CSD Settings tab: Enable common sense clock, Enable IDAC auto-calibration, Sense clock frequency
Widget Details tab: The CSD-related parameters of the Widget Hardware Parameters and Widget Threshold Parameters groups
Widget Details tab: the Compensation IDAC value parameter if Enable compensation IDAC is set.

SmartSense (Hardware parameters only) – The Hardware parameters are automatically set by the Component. Threshold parameters are set manually. This mode consumes less memory and less CPU processing time. This consumes lower average power. In this mode, the following parameters are automatically tuned:

CSD Settings tab: Enable common sense clock, Enable IDAC auto-calibration, Sense clock frequency
Widget Details tab: The CSD-related parameters of the Widget Hardware Parameters group
Widget Details tab: Compensation IDAC value parameter if Enable compensation IDAC is set.

Manual –SmartSense auto-tuning is disabled. The Widget Hardware Parameters and Widget Threshold Parameters must be tuned manually. The is the lowest memory and CPU process-time consumption mode.

SmartSense Auto-tuning (both Full Auto-Tune and Hardware parameters only) supports the IDAC sourcing configuration only.
If the SmartSense (Full Auto-Tune) is enabled, then Enable multi-frequency scan cannot be enabled.
Also, if SmartSense (Full Auto-Tune) is enabled, the Enable self-test library cannot be enabled.
SmartSense auto-tuning requires the Modulator clock frequency to be set at 6000 kHz or higher for Fourth-generation CapSense and 3000 kHz or higher for Third-generation CapSense.
SmartSense operating conditions (see Performance Characteristics):

Sensor capacitance Cp range 5 pF to 61 pF
Maximum external series resistance on a sensor Rext < 1.1 kOhm

</td></tr>
</table>

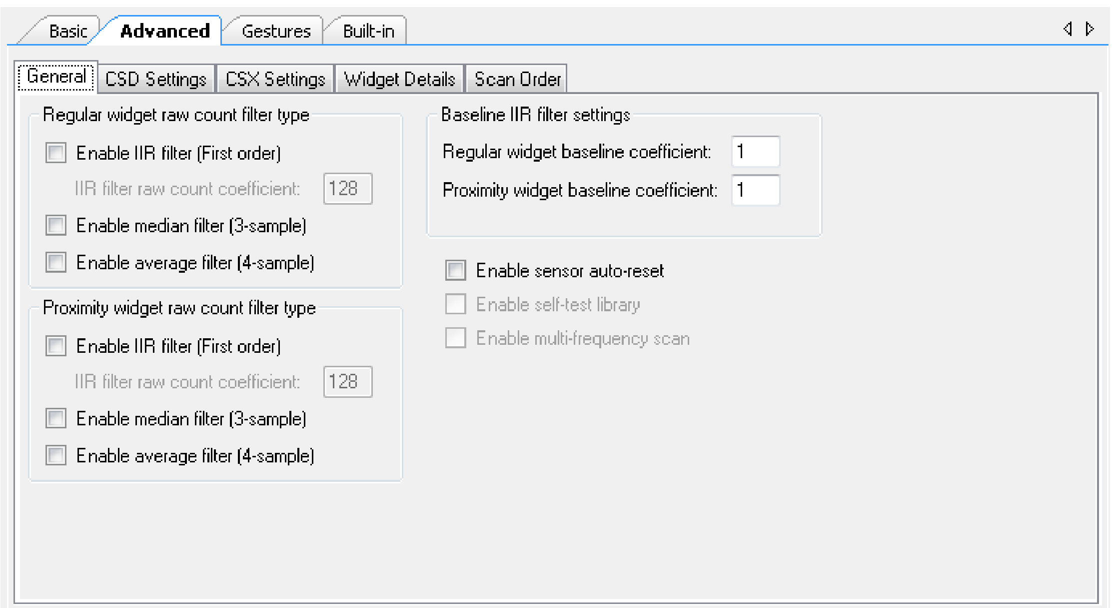
Name	Description
Widget Type	A widget is one sensor or a group of sensors that perform a specific user-interface function. The following describe the widgets types: ▪ **Button** – One or more sensors. Each sensor in the widget can detect the presence or absence (i.e., only two states) of a finger on the sensor. ▪ **Linear Slider** – More than one sensor arranged in a specific order to detect the presence and movement of a finger on a linear axis. If a finger is present, the Linear Slider detects the physical position (single axis position) of the finger. ▪ **Radial Slider** – More than one sensor arranged in a circular order to detect the presence and radial movement of a finger. If a finger is present, the Radial Slider detects the physical position of the finger. ▪ **Matrix Buttons** – Two or more sensors arranged in a specific horizontal and vertical order to detect the presence or absence of a finger on the intersections of vertically and horizontally arranged sensors. If M and N are numbers of sensors in the horizontal and vertical axis, respectively, the total of the M x N intersection positions can detect a finger touch. When using the *CSD sensing method*, a simultaneous finger touch on more than one intersection is invalid and produces invalid results. This limitation does not apply when using the *CSX sensing method* and all intersections can detect a valid touch simultaneously. ▪ **Touchpad** – Multiple sensors arranged in the specific horizontal and vertical order to detect the presence or absence of a human finger. If a finger is present, the widget will detect the physical position (both X and Y axis position) of the touch. More than one simultaneous touch in the *CSD sensing method* is invalid. The *CSX sensing method* supports detection of up to 3 simultaneous finger touches. ▪ **Proximity Sensor** – One or more sensors. Each sensor in the widget can detect the proximity of conductive objects, such as a human hand or finger to the sensors. The proximity sensor has two thresholds: o *Proximity threshold –* To detect an approaching hand or finger. o *Touch threshold –* To detect a finger touch on the sensor.
Widget Name	A widget name can be defined to aid in referring to a specific widget in a design. A widget name does not affect functionality or performance. A widget name is used throughout source code to generate macro values and data structure variables. A maximum of 16 alphanumeric characters (the first letter must be an alphabetic character) is acceptable for a widget name.
Sensing mode	The parameter to select the sensing mode for each widget: ▪ **CSD sensing method (Capacitive Sigma Delta) –** A Cypress patented method of performing self-capacitance measurements. All widget types support CSD sensing. ▪ **CSX sensing method –** A Cypress patented method of performing mutual-capacitance measurements. Only buttons, matrix buttons, and touchpad widgets support CSX sensing.

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>Widget Sensing element(s)</td><td>A sensing element refers to the Component terminals assigned to port pins to connect to physical sensors on a user-interface panel (such as a pad or layer on a PCB, ITO, or FPCB).
The following element numbers are supported by the CSD sensing method:
▪ Button – Supports 1 to 32 sensors within a widget.
▪ Linear Slider – Supports 3 to 32 segments within a widget.
▪ Radial Slider – Supports 3 to 32 segments within a widget.
▪ Matrix Buttons – Support 2 to 16 rows and columns. The number of total intersections (sensors) is equal to that of rows x columns, limited to the maximum of 32.
▪ Touchpad – Supports 3 to 16 rows and columns.
▪ Proximity – Supports 1 to 16 sensors within a widget.
The following element numbers are supported by the CSX sensing method:
▪ Button – 1 to 32 Rx electrodes (for 1 to 32 sensors) and Tx is fixed to 1.
▪ Matrix Buttons – 2 to 16 Tx and Rx. The total intersections (node) number is equal to Tx × Rx limited to the maximum of 32.
▪ Touchpad – Supports 3 to 16 Tx and Rx. The total intersections (node) number is equal to Tx × Rx. The maximum number of nodes is 256.</td></tr>
<tr><td>Finger capacitance</td><td>Finger capacitance is defined as capacitance introduced by a user touch on the sensors. This parameter is used to indicate how a sensitive CSD widget is tuned by the SmartSense Auto-tuning algorithm.
The supported Finger capacitance range:
▪ SmartSense (Full Auto-Tune) mode – 0.1 pF to 1 pF with a 0.02-pF step.
▪ SmartSense (Hardware parameters only) mode – 0.02 pF to 20.48 pF on the exponential scale.
CapSense sensor sensitivity is inversely proportional to a finger capacitance value. A smaller value of finger capacitance provides higher sensitivity for a sensor. To detect a user touch on a thick overlay (4-mm plastic overlay), finger capacitance is set to a small value (e.g., 0.1pF). For a sensor with a thin overlay or no overlay, the 0.1pF finger capacitance setting makes the sensor too sensitive and may cause false touches. For robust operation, it is important to set the appropriate finger capacitance value by considering the sensor size and overlay thickness of the design. Refer to the CapSense design guide for more information.</td></tr>
<tr><td>Move up / Move down</td><td>Moves the selected widget up or down by one on the list. It defines the widget scanning order.
Note Moving a widget may break a pin assignment, which requires repairing the assignment in the Pin Editor.</td></tr>
<tr><td>Delete</td><td>Deletes the selected widget from the list.
Note Deleting a widget may break a pin assignment, which requires repairing the assignment in the Pin Editor.</td></tr>
<tr><td>CSD electrodes</td><td>Indicates the total number of electrodes (port pins) used by the CSD widgets, including the Cmod, Csh and Shield electrodes.</td></tr>
<tr><td>CSX electrodes</td><td>Indicates the total number of electrodes (port pins) used by the CSX widgets, including the CintA and CintB capacitors.</td></tr>
</table>

Name	Description
Pins required	Indicates the total number of port pins required for the design. This does not include port pins used by other Components in the project or SWD pins in Debug mode. The number of **Pins required** must always be less than or equal to that of *Pins availabl* for a project to build successfully. **Pins required** includes the number of CSD and CSX electrodes, *Cmod*, *Csh, Shield*, *CintA* ,and *CintB* electrodes.
Pins available	Indicates the total number of port pins available for the selected device.

Advanced Tab

This tab provides advanced configuration parameters. In *SmartSense Auto-tuning,* most of the advanced parameters are automatically tuned by the algorithm and the user does not need to set values for these parameters. When the manual tuning mode is selected, this tab allows the user to control and configure all the Component parameters.



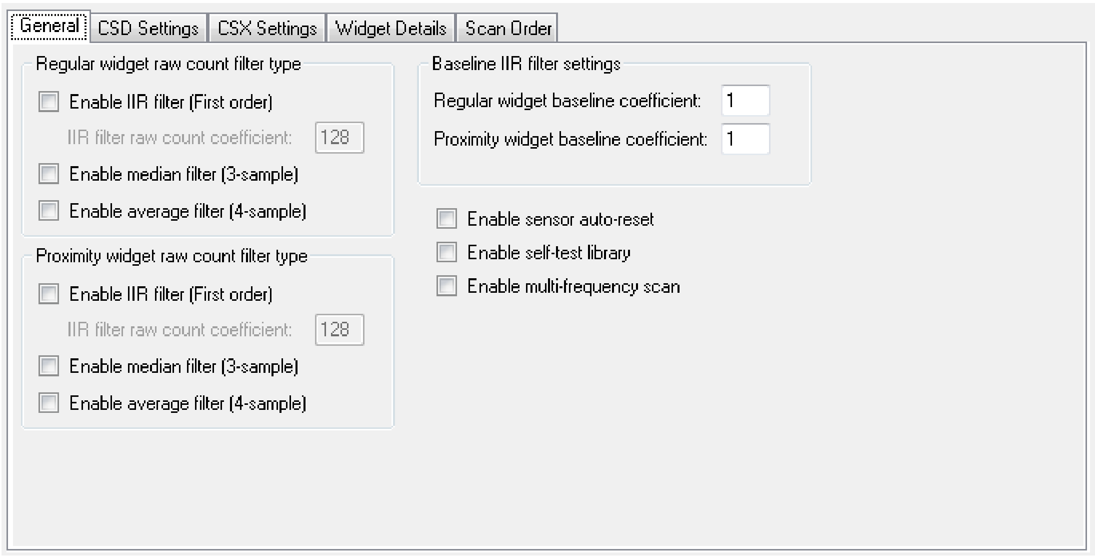
The parameters in the Advanced tab are arranged in the follwing sub-tabs.

- *General* – Contains parameters common for all widgets respective of the sensing method used for the widgets.
- *CSD Settings* – Contains parameters common for all widgets using the CSD sensing method. This tab is relevant only if one or more widget use the CSD sensing method.
- *CSX Settings* – Contains parameters common for all widgets using the CSX sensing method. This tab is relevant only if one or more widget use the CSX sensing method.
- *Widget Details* – Contains parameters specific to widgets and/or sensors.

- *Scan Order* – Provides information such as scan time for each sensor and total scan time for all sensors.

General Sub-Tab

Contains parameters common for all widgets respective of *Sensing mode* used for widgets.



These parameters are described in the following sections:

Regular widget raw count filter type

The Regular widget raw count filter type applies to raw counts of sensors belonging to non-proximity widgets. These parameters can be enabled only when one or more non-proximity widgets are added to the **Basic** tab. The filter algorithm is executed when any processing function is called by the application layer. When enabled, each filter consumes RAM to store a previous raw count (filter history). If multiple filters are enabled, the total filter history correspondingly increases so that the size of the total filter history is equal to a sum of all enabled filter histories.

Name	Description
Enable IIR filter (First order)	Enables the infinite-impulse response filter (See equation below) with a step response similar to an RC low-pass filter, thereby passing the low-frequency signals (finger touch responses). $Output = \frac{N}{K} \times input + \frac{(K-N)}{K} \times previousOutput$ where: *K* is always 256. *N* is the IIR filter raw count coefficient selectable from 1 to 128 in the customizer. A lower N (set in the *IIR filter raw count coefficient* parameter) results in lower noise, but slows down the response. This filter eliminates high-frequency noise. Consumes 2 bytes of RAM per each sensor to store a previous raw count (filter history).
IIR filter raw count coefficient	The coefficient (N) of IIR filter for raw counts is explained in the *Enable IIR filter (First order)* parameter. The range of valid values: 1-128
Enable median filter (3-sample)	Enables a non-linear filter that takes three of most recent samples and computes the median value. This filter eliminates spike noise typically caused by motors and switching power supplies. Consumes 4 bytes of RAM per each sensor to store a previous raw count (filter history).
Enable average filter (4-sample)	The finite impulse response filter (no feedback) with equally weighted coefficients. It takes four of most recent samples and computes their average. Eliminates periodic noise (e.g. noise from AC mains). Consumes 6 bytes of RAM per each sensor to store a previous raw count (filter history).

Note If the *Enable multi-frequency scan* parameter is enabled, the memory consumption of filters increases by three times.

Note If multiple filters are enabled, the execution order is as follows:

1. Median filter
2. IIR filter
3. Average filter

Proximity widget raw count filter type

The proximity widget raw count filter applies to raw counts of sensors belonging to the proximity widgets. These parameters can be enabled only when one or more proximity widgets are added on the *Basic tab*.

Parameter Name	Description
Enable IIR filter (First order)	The design of these parameters is the same as the *Regular widget raw count filter* type parameters. The *Proximity* sensors require high-noise reduction. These dedicated parameters allow for setting the proximity filter configuration and behavior differently compared to other widgets.
IIR filter raw count coefficient	
Enable median filter (3-sample)	
Enable average filter (2-sample)	

Baseline filter settings

Baseline filter settings are applied to all sensors baselines. However, filter coefficients for the proximity and regulator widgets can be controlled independently from each other.

The design baseline IIR filter is the same as the raw count *Enable IIR filter (First order)* parameter, but filter coefficients can be separate for both baseline and raw count filters to produce a different roll-off. The baseline filter is applied to a filtered raw count (if the widget raw count filters are enabled).

Name	Description
Regular widget baseline coefficient	Baseline IIR filter coefficient selection for sensors in non-proximity widgets. The range of valid values: 1-255.
Proximity widget baseline coefficient	The design of these parameters is the same as the *Regular widget baseline coefficient*, but with a dedicated parameter allows controlling the baseline update-rate of the proximity sensors differently compared to other widgets.

General settings

General settings are applicable to the whole Component behavior.

Name	Description
Enable sensor auto-reset	When enabled, the baseline is always updated. When disabled, the baseline is updated only when the difference between the baseline and raw count is less than the noise threshold. When enabled, this feature prevents sensors from permanently turning on when the raw count accidentally rises due to a large power supply voltage fluctuation or other spurious conditions.

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>Enable self-test library</td><td>
The Component provides the Built-In Self-Test (BIST) library to support Class B (IEC-60730), safety integrity-level compliant design such as white goods and automotive, and design for manufacturing testing. The library includes a set of tests for board validation, as well as Component configuration and operation. Enable the feature to get these advantages. Include the safety functions for risk-reduction, validate boards at manufacturing, and verify the Component operation in run-time.
The provided tests are classified into two categories:

HW Tests – To confirm the CSD block and sensor hardware (external to chip) are functional:

Chip analog routing verification
Pin faults checking
PCB-trace opens / shorts checking
External capacitors and sensors capacitance measurement
VDDA measurement.

FW Tests – To confirm the integrity of data used for decision making on the sensor status:

Component global and widget specific configuration verification
Sensor baseline duplication
Sensor raw count and baseline are in the specified range

The application layer is responsible for running each test at start and run-time as required by the product requirements.
The high-level function CapSense_RunSelfTest() executes a set of tests based on an enable-mask input. This function allows running all tests or only the selected tests. The return status contains a PASS/FAIL bit for each test. Also, a set of low-level functions allows executing tests specific to a widget and a sensor. The execution time of each test is less than 10 ms at PeriClk = 12 MHz when low-level functions are used. Refer to the Application Programming Interface section for more details.
Note Use CapSense_SetParam() to update the CapSense Data Structure parameters. Any other method invalids the CRC.
Note If SmartSense (Full Auto-Tune) is enabled, the self-test library cannot be enabled.
</td></tr>
</table>

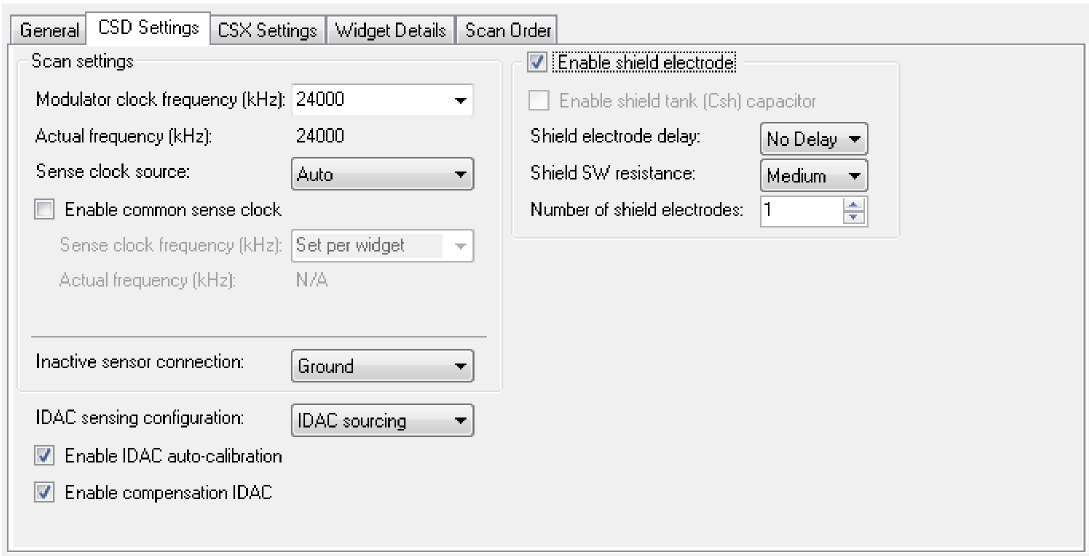
<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>Enable multi-frequency scan</td><td>
The multi-frequency scan performs a triple-sensor scan with different frequencies. Then, it chooses a median sensor difference-count for further processing. Enable the feature for robust and reliable operation in the presence of external noise at a certain sensor scan frequency.
When the multi-frequency scan is enabled, each sensor is scanned three times with three different sensor frequencies. The Component changes the IMO frequency of the device during a triple scan. The frequency of the scan is called a channel. The base channel (zero channel) is the nominal IMO frequency. Based on the device limitations, the second and the third channels frequencies are: +5% and +10% or -5% and +5% or -5% and -10%. When a sensor scan is complete, the nominal IMO frequency is configured back. The Component finishes sensor scanning after all the three frequency scans have been performed. The Component tracks the raw count and baseline for a sensor separately for each frequency channel, then calculates three difference counts. Finally, it chooses the optimal difference count by applying the median filter to the calculated difference counts.
If Enable compensation IDAC is enabled, then each sensor has three IDAC values corresponding to each scan channel.
If any of the raw count filters is enabled (Regular widget raw count filter type or Proximity widget raw count filter type), it is applied to the three sensor raw counts and their filter history separately.
The multi-frequency scan algorithm is common for the CSX and CSD sensing methods. The multi-frequency scan and SmartSense (Full Auto-Tune) features are mutually exclusive. I.e. if the multi-frequency scan is enabled, it is not possible to enable SmartSense (Full Auto-Tune) or vice-versa.
For the CSX widgets, the Tx clock frequency is set to 300 kHz for the Third-generation CapSense devices and 1MHz for Fourth-generation CapSense devices.
Side effects:

Increased flash and RAM usage. Refer to the Memory Usage section for details.
Increased the sensor scan duration by three times and partially processing time.
The multi-frequency scan changes the IMO clock. All Components which reuse IMO for critical time-dependent operations will be affected by the CapSense Component. For example, the communication-oriented Component.

</td></tr>
</table>

CSD Settings Sub-Tab

This sub-tab contains parameters common for all widgets using the *CSD sensing method*. It is relevant only if at least one widget uses the CSD sensing method.



These parameters are described in the following table:

Name	Description
Modulator clock frequency	Selects the modulator clock frequency used for the *CSD sensing method*. It is the operating frequency of the CSD block. The minimum value is 1000 kHz. The maximum value is device-dependent as follows: ▪ PSoC 4000: 16000 kHz or equal or HFCLK, whichever is lower. ▪ PSoC 4100/PSoC 4200: 24000 kHz or HFCLK/2, whichever is lower. ▪ PSoC 4000S/PSoC 4100S/PSoC 4100S Plus/PSoC Analog Coprocessor: 48000 kHz or HFCLK, whichever is lower. ▪ Other devices (PSoC 4200 BLE/PRoC BLE/PSoC 4200M/PSoC 4200L): 24000 kHz or HFCLK, whichever is lower. Enter any value between the min and max limits based on the availability of the clock divider, the next valid lower value is selected by the Component, and the actual frequency is shown in the read-only label below the drop-down list. The default value is the highest modulator clock. A higher modulator clock-frequency reduces the sensor scan time. This results in lower average power consumption and reduces the noise in the raw counts. Cypress recommends using the highest possible frequency. *SmartSense Auto-tuning* requires the *Modulator clock frequency* to be set at 6000 kHz or higher for *Fourth-generation CapSense* and 3000 kHz or higher for *Third-generation CapSense*.

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>Sense clock source</td><td>
Sense clock frequency is derived from the Modulator clock frequency using a clock-divider and is used to sample the sensor. Both the clock source and clock frequency are configurable.
The Spread Spectrum Clock (SSC) provides a dithering clock source with a center frequency equal to the frequency set in the Sense clock frequency parameter. The PRS clock source spreads the clock using the pseudo-random sequencer and the Direct source disables both SSC and PRS sources and uses a fixed-frequency clock.
Both PRS and SSC reduce the radiated noise by spreading the clock and improve the immunity against external noise. Using a higher number of bits of SSC and PRS lowers the radiation and increases the immunity against external noise.
The following sources are available:

Direct – PRS and SSC are disabled and a fixed clock is used.
PRS8 – The clock spreads using PRS to Modulator Clock / 256.
PRS12 – The clock spreads using PRS to Modulator Clock / 4096.
Auto – The Component automatically selects optimal SSC, PRS or Direct sources individually for each widget. The Auto is the recommended sense clock source selection.

In addition to the listed above options, the following sense-clock sources are available as follows:

Fourth-generation CapSense: SSC6, SSC7, SSC9 and SSC10 – The clock spreads using a range of 6 bits to 10 bits of the sense-clock divider respectively.

The following rules and recommendations for the SSC selection:

The ratio between the Modulator clock frequency and Sense clock frequency must be greater than or equal to 20.
20% of the ratio between the Modulator clock frequency and Sense clock frequency should be greater or equal to the SSC frequency range = 32. It allows varying the ratio between the Modulator and Sense clock frequencies to 32 different clocks evenly spaced over +/- 10% from the center frequency.
$$160 \le \frac{ModClk}{SnsClk}$$
Where ModClk is the Modulator clock frequency and SnsClk is Sense clock frequency.
At least one full-spread spectrum polynomial should end during the scan time:
$$\frac{2^N - 1}{ModClk} \ge \frac{2^{SSCN} - 1}{SnsClk}$$
where N is the Scan resolution, SSCN is the number of bits used for SSC (6, 7, 9 and 10 for Fourth-generation CapSense),
ModClk is Modulator clock frequency and SnsClk is Sense clock frequency.
The number of sub-conversions for the widget should be an integer multiple of the SSC polynomial selected. For example, if SSC6 is selected, the number of the sub-conversion should be multiple of (2^{SSC6}-1) = 63.

</td></tr>
</table>

Name	Description
Sense clock source (cont.)	The recommendation for the PRS selection: ■ At least one full PRS polynomial should finish during the scan time: $\frac{2^N - 1}{ModClk} \geq \frac{2^{PRSN} - 1}{SnsClk}$ where *N* is the *Scan resolution*, *PRSN* is the number of bits used for PRS (8 and 12), *ModClk* is the *Modulator clock frequency* and *SnsClk* is the average *Sense clock frequency*
Enable common sense clock	When selected, all CSD widgets share the same sense clock at a frequency specified in the *Sense clock frequency (kHz)* parameter. Otherwise, *Sense clock frequency* can be entered separately for each CSD widget in the *Widget Details* tab. Using a common sense clock for all CSD widgets results in lower power consumption and optimized memory usage. However, if the sensor parasitic capacitance significantly differs for each widget, then a common sense clock may not produce the optimal performance. To enable SmartSense Auto-tuning, disable this parameter, because SmartSense will set a Sense clock for each widget based on the sensor properties for the optimal performance.
Sense clock frequency	Sets the CSD Sense clock frequency. The minimum value is 45 kHz. The maximum value depends on the selected device: ■ PSoC 4100 / PSoC 4200: 12000 kHz or MODCLK/2, whichever is lower (MODCLK is CSD *Modulator clock frequency*). ■ PSoC 4000S / PSoC 4100S / PSoC 4100S Plus / PSoC Analog Coprocessor: 6000 kHz or HFCLK/2, whichever is lower. ■ Other devices: 12000 kHz or HFCLK/2, whichever is lower. Enter any value between the min and max limits, basing on the clock divider availability, the next valid lower value is selected by the Component, and the actual frequency appears in the read-only label below the drop-down list. When SmartSense is selected in *CSD tuning mode*, the Sense Clock frequency is automatically set by the Component to an optimal value by following the 2*5*R*C rule (refer to *CapSense design guide* for more information on this rule) and this control is grayed out. When *Enable common sense clock* is unselected, the Sense clock frequency can be set individually for each widget in the *Widget Details* tab, and this control is grayed out. **Note** If the PeriClk frequency or *Modulator clock frequency* changes, the Component automatically recalculates the next closest Sense clock frequency value to a possible one.
Inactive sensor connection	Selects the state of the sensor when it is not scanned: ■ **Ground (default)** – Inactive sensors are connected to ground. ■ **High-Z** – Inactive sensors are floating (not connected to GND or Shield). ■ **Shield** - Inactive sensors are connected to Shield. This option is available only if the *Enable shield electrode* check box is set. **Ground** is the recommended selection for this parameter when water tolerance is not required for the design. Select **Shield** when the design needs water tolerance or to reduce the sensor parasitic capacitance in the design.

Name	Description
IDAC sensing configuration	Selects the type of IDAC switching: ▪ **IDAC sourcing (default)** – Sources current into the modulator capacitor (*Cmod*). The analog switches are configured to alternate between the *Cmod* and GND. IDAC Sourcing is recommended for most designs because of the better signal-to-noise ratio ▪ **IDAC sinking** – Sinks current from the modulator capacitor (*Cmod*). The analog switches are configured to alternate between V_{DD} and *Cmod*.
Enable IDAC auto-calibration	When enabled, values of the CSD widget IDACs are automatically set by the Component. Select the Enable IDAC Auto-calibration parameter for robust operation. The SmartSense Auto-tuning parameter can be enabled only when the Enable IDAC auto-calibration is selected.
Enable compensation IDAC	The compensation IDAC is used to compensate for sensor parasitic capacitance to improve performance. Enabling the compensation IDAC is recommended unless one IDAC is required for general purpose (other than CapSense) in the project.
Enable shield electrode	The shield electrode is used to reduce the sensor parasitic capacitance, enable water-tolerant CapSense designs and enhance the detection range for the *Proximity* sensors. When the shield electrode is disabled, configurable parameters associated with the shield electrode are hidden.
Enable shield tank (Csh) capacitor	The shield tank capacitor is used to increase the drive capacity of the shield electrode driver. It should be enabled when the shield electrode capacitance is higher than 100 pF. The recommended value for a shield tank capacitor is 10nF/5V/X7R or an NP0 capacitor. The shield tank capacitor is not supported in configurations that include both CSD and CSX sensing-based widgets.
Csh initialization source	Selects the initialization source for the shield tank electrode, when *Enable shield tank (Csh) capacitor* is enabled. The two options are available: ▪ **Vref** – Precharge the shield tank by connecting VREF to the Csh capacitor. ▪ **IO Buffer** – Precharge the shield tank by connecting the VDD supply to the Csh capacitor and turning it off using the feedback system when the Csh voltage reaches Vref. This option is available only when the Csh capacitor is assigned to one of the dedicated Csh pins (refer to the device datasheet for pin details) and these dedicated pins are available for the Csh when the *CSX sensing method* is not used in the project. The recommended source of precharge is the IO buffer. **Note** This parameter is available for Third-generation CapSense only.

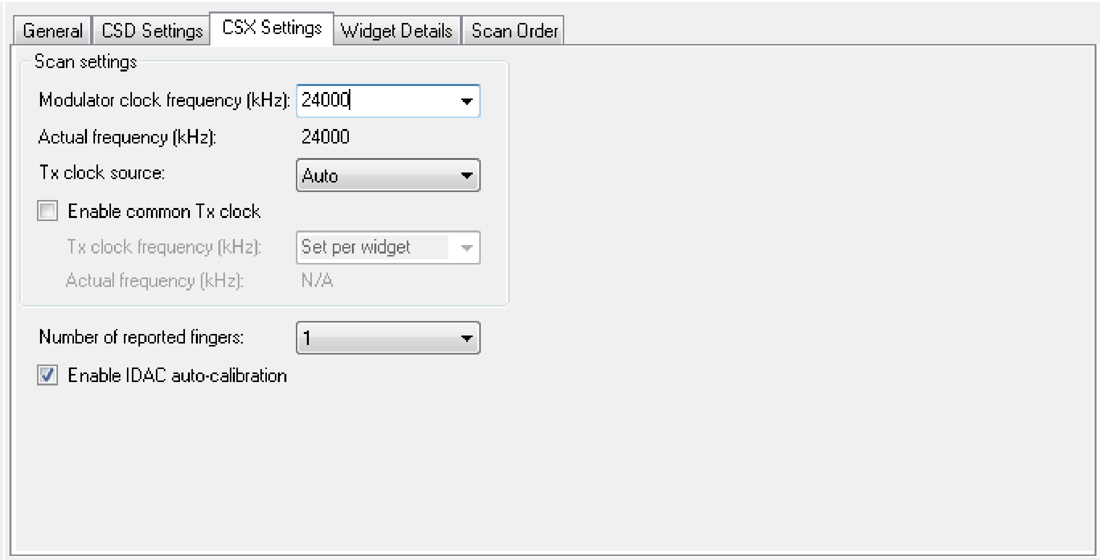
Name	Description

Shield electrode delay	Configures the delay between the sensor signal and the shield electrode signal for compensation of the delay added by signal routing. The following options are available for selection:
	▪ *Third-generation CapSense*
	  ○ **No Delay**
	  ○ **10 ns**
	  ○ **50 ns**.
	▪ *Fourth-generation CapSense*:
	  ○ **No Delay**
	  ○ **5 ns**
	  ○ **10 ns**
	  ○ **20 ns**.
	Most designs work with the **No delay** option and it is the recommended value.

Shield SW resistance	Selects the resistance of switches used to drive the shield electrode. The four options: ▪ **Low** ▪ **Medium (default)** ▪ **High** ▪ **Low EMI** **Note** This parameter is available for *Fourth-generation CapSense* only.
Number of shield electrodes	Selects the number of shield electrodes required in the design. Most designs work with one dedicated shield electrode, but some designs require multiple dedicated shield electrodes to ease the PCB layout routing or to minimize the PCB area used for the shield layer. The minimum value is 0 (i.e. shield signal could be routed to sensors using the *Inactive sensor connection* parameter) and the maximum value is equal to the total number of CapSense-enabled port pins available for the selected device.

CSX Settings Sub-tab

The parameters in this sub-tab apply to all widgets that use the *CSX sensing method.* If no widget uses the CSX sensing method, the configuration parameters in this sub-tab are grayed out and become not configurable.



These parameters are described in the following table:

Name	Description
Modulator clock frequency	Selects the modulator clock frequency used for the *CSX sensing method*. It is the operating frequency of the CSD block. The minimum value is 1000 kHz. The maximum value is device-dependent as follows: ■ PSoC 4000: 16000 kHz or equal or HFCLK, whichever is lower. ■ PSoC 4100/PSoC 4200: 24000 kHz or HFCLK/2, whichever is lower. ■ PSoC 4000S/PSoC 4100S/PSoC 4100S Plus/PSoC Analog Coprocessor: 48000 kHz or HFCLK, whichever is lower. ■ Other devices (PSoC 4200 BLE/PRoC BLE/PSoC 4200M/PSoC 4200L): 24000 kHz or HFCLK, whichever is lower. Enter any value between the min and max limits, basing on the availability of the clock divider, the next valid lower value is selected by the Component, and the actual frequency appears in the read-only label below the drop-down list. A higher modulator clock-frequency reduces the sensor scan time, results in lower power, and reduces the noise in raw counts. Cypress recommends using the highest possible frequency.

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>Tx clock source</td><td>
The Tx clock frequency derives from the Modulator clock frequency using a clock-divider and is used to sample the sensor. Both the type of the clock source and the clock frequency are configurable in Fourth-generation CapSense devices, in Third-generation CapSense, Direct clock source is used and not configurable.
The Spread Spectrum Clock (SSC) provides a dithering clock source with a center frequency equal to the frequency set in the Tx clock frequency parameter and the Direct source disables the SSC source and uses a fixed frequency clock. The SSC reduces the radiated noise by spreading the clock and improves the immunity against external noise. Using a higher number of bits of SSC lowers the radiation and increases the immunity against external noise.
The following clock sources are available:

Direct – SSC is disabled and a fixed clock is used.
Auto – The Component automatically selects optimal SSC or Direct sources individually for each widget. Auto is the recommended Sense clock source selection.

In addition to the listed above options, the following sense-clock sources are available as follows:

Fourth-generation CapSense: SSC6, SSC7, SSC9 and SSC10 – The clock spreads using a range of 6 bits to 10 bits of the sense-clock divider respectively.

The rules and recommendations for the SSC selection:

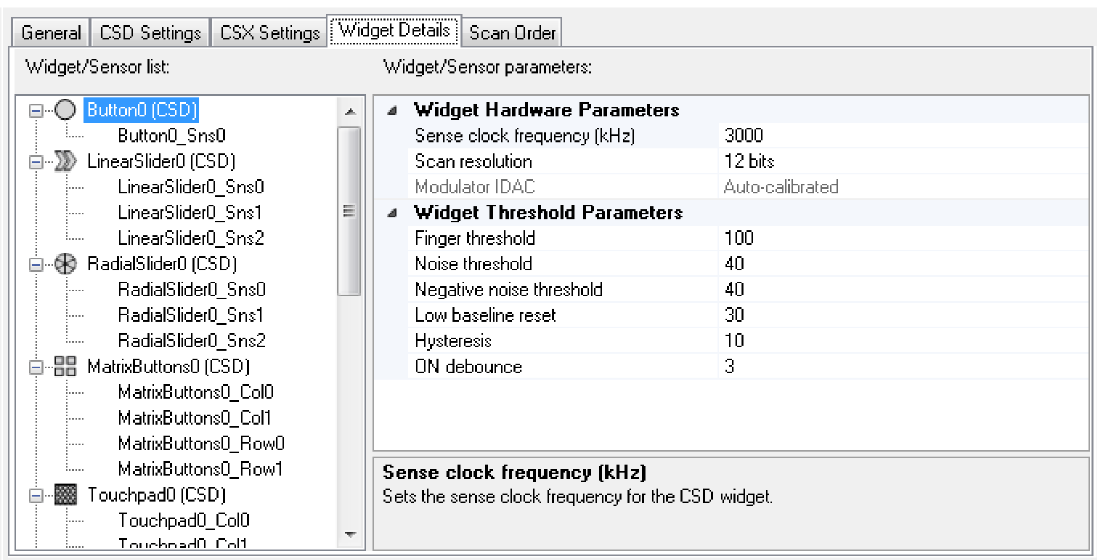
The ratio between the Modulator clock frequency and Tx clock frequency must be greater than or equal to 20.
20% of the ratio between the Modulator clock frequency and Tx clock frequency should be greater or equal to the SSC frequency range = 32. It allows varying the ratio between the Modulator and Tx clock frequencies to 32 different clocks evenly spaced over +/-10% from the center frequency.
$$160 \le \frac{ModClk}{TxClk}$$
where ModClk is the Modulator clock frequency and TxClk is Tx clock frequency.
It is recommended that at least one full-spread spectrum polynomial should end during the scan time.
$$\frac{N_{Sub}}{ModClk} \ge \frac{2^{SSCN}-1}{TxClk}$$
where N_{Sub} is the Number of sub-conversions, SSCN is the number of bits used for SSC (6, 7, 9 and 10), ModClk is the Modulator clock frequency and TxClk is the Tx clock frequency.

It is recommended that Number of sub-conversions for the widget should be an integer multiple of the SSC polynomial selected. For example, if SSC6 is selected, the number of sub-conversion should be multiple of (2^{SSC6}-1) = 63.
</td></tr>
<tr><td>Enable common Tx clock</td><td>
When selected, all CSX widgets share the same Tx clock with the frequency specified in the Tx clock frequency (kHz) parameter. Otherwise, the Tx clock frequency is entered separately for each CSX widget in the Widget Details tab.
Using the common Tx clock for all CSX widgets results in lower power consumption and optimized memory usage and it is the recommended setting for the CSX widgets. But, in rare cases, if the electrode properties capacitance is significantly different for each widget, a common Tx clock may not produce the optimal performance.
</td></tr>
</table>

Name	Description
Tx clock frequency	Sets the Tx clock frequency. The minimum value is 45 kHz for all device families. The maximum value depends on the selected device as follows: ▪ *Fourth-generation CapSense*: 3000 kHz. ▪ *Third-generation CapSense*: 300 kHz. Set any value between the min and max limits, basing on the clock divider availability, the next valid lower value is selected by the Component, and the actual frequency appears in the read-only label below the drop-down list. The highest Tx clock frequency produces the maximum signal and is the recommended setting. When *Enable common Tx clock* is unselected, the Tx clock frequency is set individually for each widget in the *Widget Details* tab, and this control is grayed out. **Note** If the PeriClk frequency or *Modulator clock frequency* is changed, the Component automatically recalculates the next closest Tx clock frequency value to a possible one.
Number of reported fingers	Sets the number of reported fingers for a CSX Touchpad widget only. The available options are from 1 to 3.
Enable IDAC auto-calibration	When enabled, IDAC values are automatically set by the Component. It is recommended to select the Enable IDAC auto-calibration for robust operation.

Widget Details Sub-tab

This sub-tab contains parameters specific to each widget and sensor. These parameters must be set when *SmartSense (Full Auto-Tune)* is not enabled. The parameters are unique for each widget type.



These parameters are described in the following table:

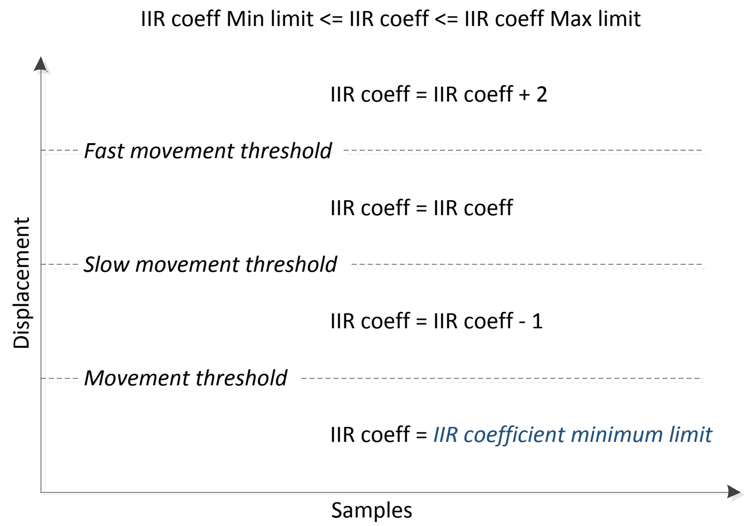
Name	Description
Widget General Parameters	
Diplexing	Enabling Diplexing allows doubling the slider physical touch sensing area by using a specific duplexing sensor pattern and without using additional port pins and sensors in the Component.
Maximum position	Represents the maximum Centroid position for the slider. A touch on the slider would produce a position value from 0 to the maximum position-value set. **No Touch** would produce 0xFFFF.
Maximum X-axis position	Represents the maximum column (X-axis) Centroid position and row (Y-axis) Centroid positions for a touchpad. A touch on the touchpad would produce a position value from 0 to the maximum position set. **No Touch** would produce 0xFFFF.
Maximum Y-axis position	
Widget Hardware Parameters **Note** All Widget Hardware parameters for CSD widgets are automatically set when *SmartSense (Full Auto-Tune)* is selected in the *CSD tuning mode*.	
Sense clock frequency	This parameter is identical to the *Sense clock frequency* parameter in the *CSD Settings* tab. When *Enable common sense clock* is unselected in the *CSD Settings* tab, a sense-clock frequency for each widget is set here.
Row sense clock frequency	These parameters are identical to the *Sense clock frequency* parameter, and are used to set a sense-clock frequency for row and column sensors of the *Matrix Buttons* and *Touchpad* widgets.
Column sense clock frequency	
Tx clock frequency	This parameter is identical to the *Tx clock frequency* parameter in the *CSX Settings* tab. When *Enable common Tx clock* is unselected in the *CSX Settings* tab, a Tx clock frequency for each widget is set here.
Scan resolution	Selects the scan resolution of CSD widgets (resolution of capacitance to digital conversion). Acceptable values are from 6 to 16 bits.
Number of sub-conversions	Selects the number of sub-conversions in the *CSX sensing method*. $N_{Sub} < \frac{2^{16} \bullet TxClk}{ModClk}$ where, *ModClk* is the CSX *Modulator clock frequency* *TxClk* is the *Tx clock frequency* N_{Sub} is the value of this parameter
Modulator IDAC	Sets the modulator IDAC value for the CSD Button, Slider, or Proximity widget. The value of this parameter is automatically set when *Enable IDAC auto-calibration* is selected in the *CSD Settings* tab.
Row modulator IDAC	Sets a separate modulator IDAC value for the row and column sensors of the CSD *Matrix Buttons* and *Touchpad widget*. These parameters values are automatically set when *Enable IDAC auto-calibration* is checked in the *CSD Settings* tab.
Column modulator IDAC	

Name	Description
Widget Threshold Parameters **Note** All the threshold parameters for the CSD widgets are automatically set when *SmartSense (Full Auto-Tune)* is selected in the *CSD tuning mode* parameter.	
Finger threshold	The finger threshold parameter is used along with the hysteresis parameter to determine the sensor state as follows: ▪ ON – Signal > (Finger Threshold + Hysteresis) ▪ OFF – Signal ≤ (Finger Threshold – Hysteresis). Note that "Signal" in the above equations refers to: Difference Count = Raw Count – Baseline. It is recommended to set the Finger threshold parameter value equal to the 80% of the touch signal. The Finger Threshold parameter is not available for the *Proximity* widget. Instead, Proximity has two thresholds: ▪ *Proximity threshold* ▪ *Touch threshold*
Noise threshold	Sets a raw count limit below which a raw count is considered as noise. When a raw count is above the Noise Threshold, a difference count is produced and the baseline is updated only if *Enable sensor auto-reset* is selected. In other words, the baseline remains constant as long as the raw count is above the baseline + noise threshold. This prevents the baseline from following raw counts during a finger touch detection event. It is recommended to set the noise threshold parameter value equal to 2x noise in the raw count or the 40% of the signal.
Negative noise threshold	Sets a raw count limit below which the baseline is not updated for the number of samples specified by the *Low baseline reset* parameter. The negative noise threshold ensures that the baseline does not fall low because of any high-amplitude repeated negative-noise spikes on a raw count caused by different noise sources such as ESD events. It is recommended to set the negative noise threshold parameter value equal to the *Noise threshold* parameter value.
Low baseline reset	This parameter is used along with the *Negative noise threshold* parameter. It counts the number of abnormally low raw counts required to reset the baseline. If a finger is placed on the sensor during a device startup, the baseline gets initialized to a high raw count value at a startup. When the finger is removed, the raw count falls to a lower value. In this case, the baseline should track low raw counts. The Low Baseline Reset parameter helps handle this event. It resets the baseline to a low raw count value when the number of low samples reaches the low-baseline reset number. **Note** After a finger is removed from the sensor, the sensor will not respond to finger touches for low baseline-reset time. The recommended value is 30, which works for most designs.

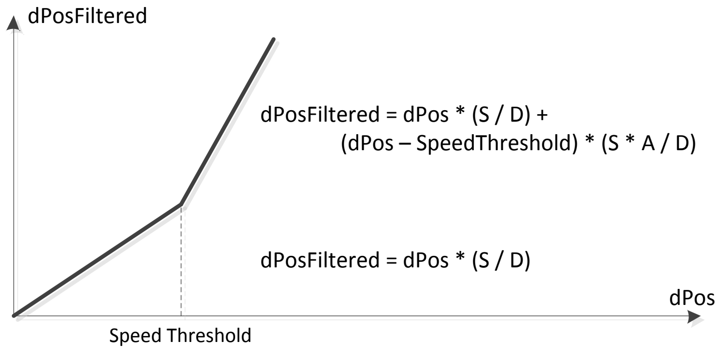
Name	Description
Hysteresis	The hysteresis parameter is used along with the *Finger threshold* parameter (*Proximity threshold* and *Touch threshold* for Proximity sensor) to determine the sensor state. The hysteresis provides immunity against noisy transitions of the sensor state. See the description of the *Finger threshold* parameter for details. The recommend value for the hysteresis is the 10% *Finger threshold*.
ON debounce	Selects a number of consecutive CapSense scans during which a sensor must be active to generate an ON state from the Component. Debounce ensures that high-frequency, high-amplitude noise does not cause false detection ▪ Buttons/Matrix buttons/Proximity – An ON status is reported only when the sensor is touched for a consecutive debounce number of samples. ▪ Sliders/Touchpads – The position status is reported only when any of the sensors is touched for a consecutive debounce number of samples. The recommended value for the Debounce parameter is 3 for reliable sensor status detection.
Proximity threshold Touch threshold	The design of these parameters is the same as for the *Finger threshold* parameters. The proximity sensor requires a higher noise reduction, and supports two levels of detection: ▪ The proximity level to detect an approaching hand or finger ▪ The touch level to detect a finger touch on the sensor similarly to other Widget Type sensors Note that for valid operation, the Proximity threshold must be higher than the Touch threshold. The threshold parameters such as *Hysteresis* and *ON debounce* are applicable to both detection levels.
Velocity	Defines the maximum speed of a finger movement in terms of the squared distance of the touchpad resolution. The parameter is applicable for a multi-touch touchpad (CSX Touchpad) only. If the detected position of the next scan is further than the defined squared distance, then this touch is considered as a separate touch with a new touch ID.
Position Filter Parameters These parameters enable firmware filters on a centroid position to reduce noise. These filters are available for Slider and Touchpad widgets only. If multiple filters are enabled, the execution order corresponds to the listed below and the total RAM consumption increases so that the size of the total filter history is equal to a sum of all enabled filter histories.	
Median filter	Enables a non-linear filter that takes three of most recent samples and computes the median value. This filter eliminates the spikes noise typically caused by motors and switching power supplies. Consumes 4 bytes of RAM per each position (filter history).

Name	Description
IIR filter	Enables the infinite-impulse response filter (see equation below) with a step response. $Output = \frac{N}{K} \times Input + \frac{(K-N)}{K} \times prevOutput$ where: *K* is always 256; *N* is the IIR filter raw count coefficient selectable from 1 to 255 in the customizer. A lower N (set in the *IIR filter coefficient* parameter) results in lower noise, but slows down the response. This filter eliminates high-frequency noise. Consumes 2 bytes of RAM per each position (filter history).
IIR filter coefficient	The coefficient (N) of the IIR filter for a position as explained in the *IIR filter* parameter. The range of valid values: 1-255.
Adaptive IIR filter	Enables the Adaptive IIR filter. It is the IIR filter that changes its IIR coefficient according to the speed of the finger movement. This is done to smooth the fast movement of the finger and at the same time control properly the position movement. The filter coefficients are automatically adjusted by the adaptive algorithm with the speed of the finger movement. If the finger moves slowly, the IIR coefficient decreases; if the finger moves fast, the IIR coefficient increases from the existing value. Consumes 3 bytes of RAM per each position (filter history). When this filter is enabled, the *Adaptive IIR Filter Parameters* are available for configuration. The adaptive IIR filter is available for gesture-enabled part numbers.
Average filter	Enables the finite-impulse response filter (no feedback) with equally weighted coefficients. It takes two of most recent samples and computes their average. Eliminates periodic noise (e.g. noise from AC mains). Consumes 2 bytes of RAM per each position (filter history).
Jitter filter	This filter eliminates the noise in the position data that toggles between the two most recent values. If the most recent position value is greater than the previous one, the current position is decremented by 1; if it is less, the current position is incremented by 1. The filter is most effective at low noise. Consumes 2 bytes of RAM per each position (filter history).
Ballistic multiplier	Enables the Ballistic multiplier filter used to provide better user experience of the pointer movement. Fast movement will move the cursor by more pixels. Consumes 16 bytes of RAM when enabled. **Note** The Ballistic multiplier filter can be enabled for only one CSD Touchpad widget. The Ballistic multiplier filter is available for gesture-enabled part numbers. The Ballistic multiplier filter depends on the scanning refresh rate.

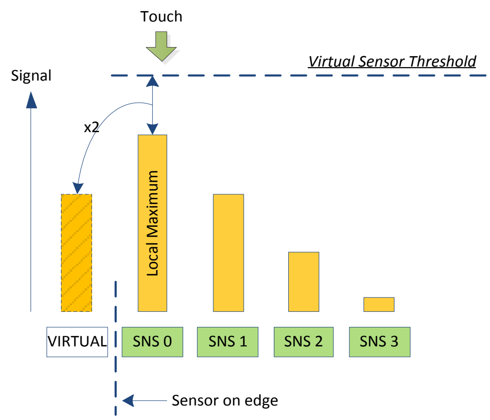
Name	Description
Adaptive IIR Filter Parameters These parameters are available when the *Adaptive IIR filter* is enabled.	

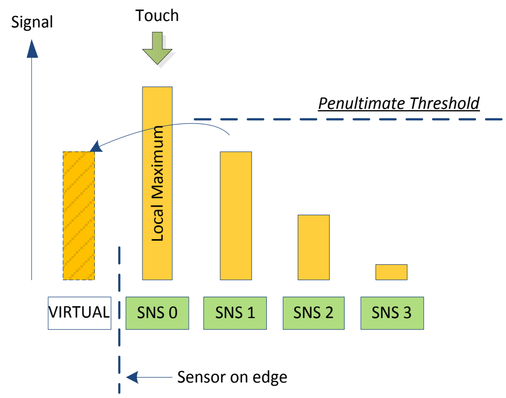


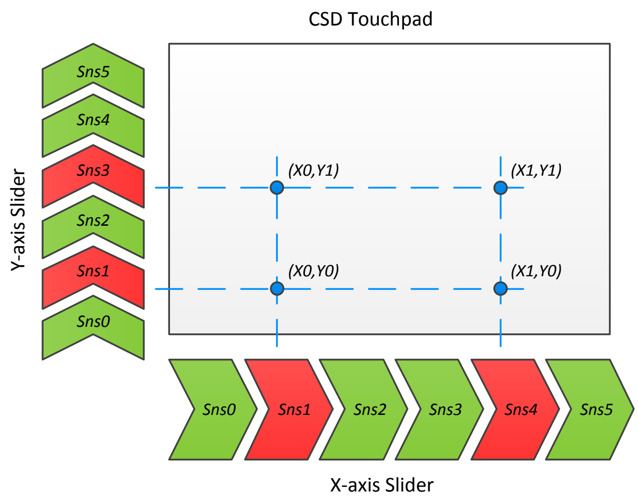
Name	Description
Position movement threshold	Defines the position threshold below which a position displacement is ignored or considered as no movement. If the position displacement is within the threshold limit, the IIR coefficient equals the *IIR coefficient minimum limit* and filtering affects a position intensively.
Position slow movement threshold	Defines the position threshold below which (and above *Position movement threshold*) a position displacement (the difference between the current and previous position) is considered as slow movement. If the position displacement is within the threshold limits, the IIR filter coefficient decreases during each new scan. So, the filter impact on the position becomes less intensive.
Position fast movement threshold	Defines the position threshold above which a position displacement is considered as fast movement. If the position displacement is above the threshold limit, the IIR filter impact on the position becomes more intensive during each new scan as the filter coefficient increases.
IIR coefficient maximum limit	Defines the maximum limit of the IIR coefficient when the finger moves fast. The fast movement event is defined by the *Position fast movement threshold*.
IIR coefficient minimum limit	Defines the minimum limit of the IIR coefficient when the finger moves slowly. The slow movement event is defined by the *Position slow movement threshold*.
IIR coefficient divisor	This parameter acts as the scale factor for the filter IIR coefficient. $$Output = \frac{Coeff}{Divisor} \times Input + \frac{Divisor - Coeff}{Divisor} \times previous\,Output$$ where: *Input*, *Output*, and *Previous Output* are the touch positions; *Coeff* is the automatically adjusted IIR filter coefficient; *Divisor* is the IIR coefficient divisor (this parameter).

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td colspan="2">Ballistic Multiplier Parameters
These parameters are available when the Ballistic multiplier is enabled.
The simplified diagram of the Ballistic Multiplier filter operation:


where,
dPos is an input position displacement either in the X axis or Y axis,
dPosFiltered is the filtered displacement;
SpeedThreshold is either the X-axis speed threshold or Y-axis speed threshold;
A is the Acceleration coefficient;
S is the Speed coefficient;
D is the Divisor value.</td></tr>
<tr><td>Acceleration coefficient</td><td>Defines the value at which the position movement needs to be interpolated when the movement is classified as fast movement. The reported position displacement is multiplied by this parameter.</td></tr>
<tr><td>Speed coefficient</td><td>Defines the value at which the position movement is interpolated when the movement is classified as slow movement. The reported position displacement is multiplied by this parameter.</td></tr>
<tr><td>Divisor value</td><td>Defines the divisor value used to create a fraction for the acceleration and speed coefficients. The interpolated position coordinates are divided by the value of this parameter.</td></tr>
<tr><td>X-axis speed threshold</td><td>Defines the threshold to distinguish fast and slow movement on the X axis. If the X-axis position displacement reported between two consecutive scans exceeds this threshold, then it is considered as fast movement, otherwise as slow movement.</td></tr>
<tr><td>Y-axis speed threshold</td><td>Defines the threshold to distinguish fast and slow movement on the Y axis. If the Y-axis position displacement reported between two consecutive scans exceeds this threshold, then it is considered as fast movement, otherwise as slow movement.</td></tr>
<tr><td colspan="2">Centroid Parameters
Centroid parameters are available for the CSD Touchpad widgets only.</td></tr>
<tr><td>Centroid type</td><td>Selects a sensor matrix size for centroid calculation. The 5x5 centroid (also known as Advanced Centroid) provides benefits such as Two finger detection, Edge correction, and improved accuracy.
If Advanced Centroid is selected, the below parameters are configured as well.</td></tr>
</table>

Name	Description
Cross coupling position threshold	Defines the cross coupling threshold. This value is subtracted from the sensor signal used for centroid position calculation to improve the accuracy. The threshold should be equal to a sensor signal when a finger is near the sensor but is not touching the sensor. This can be determined by slowly dragging the finger across the panel and finding the inflection point of the difference counts at the base of the curve. The difference value at this point is the Cross-coupling threshold. The default value is 5.
Edge correction	This feature is available if the *Centroid type* is configured to 5x5. When enabled, a matrix of centroid calculation is updated with virtual sensors on the edges of a touchpad. It improves the accuracy of the reported position on the edges. When enabled, two more parameters must be configured: *Virtual sensor threshold* and *Penultimate threshold*.
Virtual sensor threshold	This parameter is applicable only if *Edge correction* is enabled and it is used to calculate a signal (difference count) for a virtual sensor used for the edge correction algorithm. A touch position on a slider or touchpad is calculated using a signal from the local-maxima sensor and its neighboring sensors. A touch on the edge sensor of a slider or touchpad does not accurately report a position because the edge sensor lacks signal from one side of neighboring sensors of the local-maxima sensor.  If the *Edge correction* is enabled, the algorithm adds a virtual neighbor sensor to correct the deviation in the reported position. The Virtual sensor signal is defined by the Virtual sensor threshold: $DiffCount_{VIRTUAL} = (Threshold_{VIRTUAL} - DiffCount_{SNS0}) \times 2$ where: $DiffCount_{VIRTUAL}$ is the virtual sensor difference count; $Threshold_{VIRTUAL}$ is the virtual sensor threshold; $DiffCount_{SNS0}$ is the sensor 0 difference count. The conditions for a virtual sensor (and *Edge correction* algorithm) to be applied: ▪ Local-maxima detected on the edge sensor ▪ Difference count from the penultimate sensor less than the Penultimate threshold.

Name	Description
Penultimate threshold	This parameter is applicable only if the Edge correction is enabled and it works along with the Virtual sensor threshold parameter. This parameter defines the threshold of penultimate sensor signal. If the signal from penultimate sensor is below the Penultimate threshold, the edge correction algorithm is applied to the centroid calculation. The conditions for the edge correction to be applied: ▪ Local-maxima detected on the edge sensor ▪ The difference count of the penultimate sensor (SNS 1 in the figure below) less than the Penultimate threshold. 

Name	Description
Two finger detection	Enables the detection of the second finger on a CSD touchpad. In general, a CSD touchpad can detect only one true touch position. A CSD touchpad widget consists of two Linear Sliders and each slider reports the X and Y coordinates of a finger touch. If there are two touches on the touchpad, there are four possible touch positions as shown in the figure below. The two of these touches are real touches and two are known as “ghost” touches. There is no possibility to differentiate between ghost and real touches in a CSD widget (to get true multi-touch performance, use the CSX Touchpad widget).  But, if this feature is enabled, the CSD touchpad can report up to two touches, mainly to be used in conjunction with two-finger gestures where real and ghost touches do not need to be fully differentiated. It is available for the CSD touchpad only when the *Centroid type* is configured to 5x5. The Advanced centroid (*Centroid type* is 5x5) uses the 3x3 centroid matrix when detects two touches.
Sensor parameters	
Compensation IDAC value	Sets the Compensation IDAC value for each CSD sensor when *Enable compensation IDAC* is selected on the *CSD Settings* tab. If the *CSD tuning mode* is set to SmartSense Auto-tuning or *Enable IDAC auto-calibration* is selected on the *CSD Settings* tab, the value of this parameter is set equal to the Modulator IDAC value at a device power-up for the maximum performance from the sensor. Select the *Enable IDAC auto-calibration* for robust operation.
IDAC Values	Sets the IDAC value for each CSX sensor/node, a lower IDAC value without saturating raw counts provides better performance for sensor/nodes. When *Enable IDAC auto-calibration* is selected on the *CSX Settings* tab, the value of this parameter is automatically set to the lowest possible value at a device power-up for better performance. It is recommended to select *Enable IDAC auto-calibration* for robust operation.
Selected pins	Selects a port pin for the sensor (CSD sensing) and electrode (CSX sensing). The available options use a dedicated pin for a sensor or reuse one or more pins from any other sensor in the Component. Reusing the pins of any other sensor from any widgets helps create a ganged sensor.

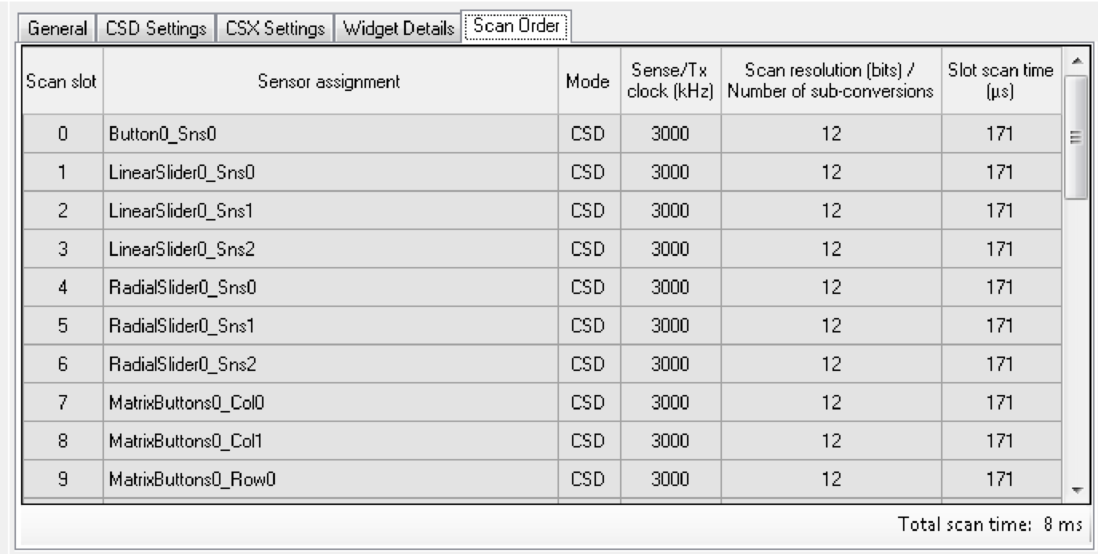
The following table shows which Widget / Sensor parameters belong to a given widget type.

Parameters	Widget Type								
	CSD Widget						CSX Widget		
	Button	Linear Slider	Radial Slider	Matrix Buttons	Touch pad	Proximity	Button	Matrix Buttons	Touch pad
Widget General									
Diplexing		√							
Maximum position		√	√						
Maximum X-axis position					√				√
Maximum Y-axis position					√				√
Widget Hardware									
Sense clock frequency	√	√	√			√			
Row sense clock frequency				√	√				
Column sense clock frequency				√	√				
Tx clock frequency							√	√	√
Scan resolution	√	√	√	√	√	√			
Number of sub-conversions							√	√	√
Modulator IDAC	√	√	√			√			
Row modulator IDAC				√	√				
Column modulator IDAC				√	√				
Widget Threshold									
Finger threshold	√	√	√	√	√		√	√	√
Noise threshold	√	√	√	√	√	√	√	√	√
Negative noise threshold	√	√	√	√	√	√	√	√	√
Low baseline reset	√	√	√	√	√	√	√	√	√
Hysteresis	√	√	√	√	√	√	√	√	√
ON debounce	√	√	√	√	√	√	√	√	√
Proximity threshold						√			
Touch threshold						√			
Velocity									√
Sensor Parameters									
Compensation IDAC value	√	√	√	√	√	√			
IDAC Values							√	√	√
Selected pins	√			√		√	√	√	√
Position Filter Parameters									
Median filter		√	√		√				√
IIR filter		√	√		√				√
IIR filter coefficient		√	√		√				√
Adaptive IIR filter		√	√		√				√
Average filter		√	√		√				√
Jitter filter		√	√		√				√
Ballistic multiplier		√	√		√				√

Parameters	Widget Type								
	CSD Widget						CSX Widget		
	Button	Linear Slider	Radial Slider	Matrix Buttons	Touch pad	Proximity	Button	Matrix Buttons	Touch pad
Adaptive IIR Filter Parameters									
Position movement threshold		√	√		√				√
Position slow movement threshold		√	√		√				√
Position fast movement threshold		√	√		√				√
IIR coefficient maximum limit		√	√		√				√
IIR coefficient minimum limit		√	√		√				√
IIR coefficient divisor		√	√		√				√
Ballistic Multiplier Parameters									
Acceleration coefficient					√				
Speed coefficient					√				
Divisor value					√				
X-axis speed threshold					√				
Y-axis speed threshold					√				
Centroid Parameters									
Centroid type					√				
Cross-coupling position threshold					√				
Edge correction					√				
Virtual sensor threshold					√				
Penultimate threshold					√				
Two finger detection					√				

Scan Order Sub-Tab

This tab provides the **Scan time** for each sensor in the Component and **Total scan time** required to scan all the sensors in the Component.



Scan slot	Sensor assignment	Mode	Sense/Tx clock (kHz)	Scan resolution (bits) / Number of sub-conversions	Slot scan time (µs)
0	Button0_Sns0	CSD	3000	12	171
1	LinearSlider0_Sns0	CSD	3000	12	171
2	LinearSlider0_Sns1	CSD	3000	12	171
3	LinearSlider0_Sns2	CSD	3000	12	171
4	RadialSlider0_Sns0	CSD	3000	12	171
5	RadialSlider0_Sns1	CSD	3000	12	171
6	RadialSlider0_Sns2	CSD	3000	12	171
7	MatrixButtons0_Col0	CSD	3000	12	171
8	MatrixButtons0_Col1	CSD	3000	12	171
9	MatrixButtons0_Row0	CSD	3000	12	171

Total scan time: 8 ms

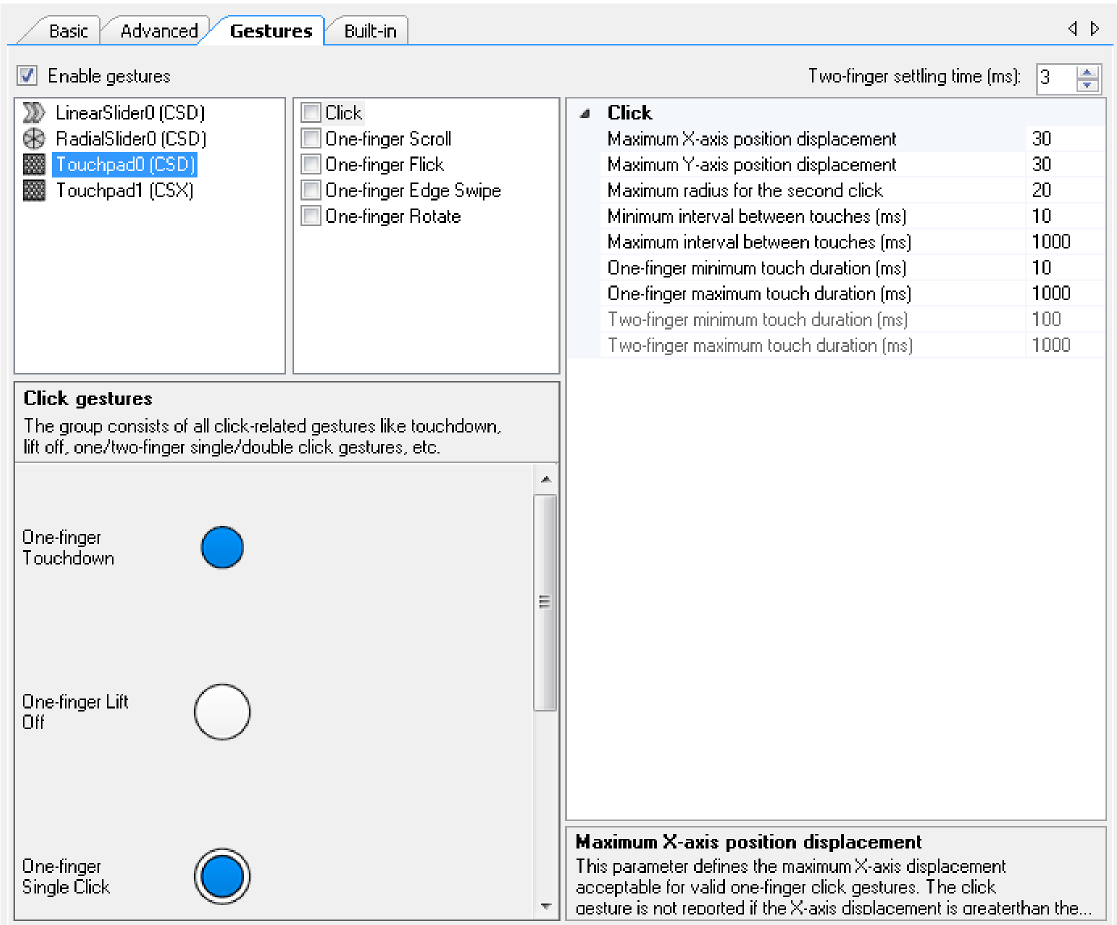
This **Scan Order** tab provides hardware scan duration for each sensor and total hardware scan duration for all the sensors in the component. The actual duration to complete a scan is sum of duration of hardware scan, duration of initialization prior a scan and duration of firmware execution. Therefore, it is recommended to measure the time (from start of scan function to end of CapSense processing function) on hardware for accurate scan time information.

Note If *SmartSense Auto-tuning* mode is enabled for CSD Widgets, the scan time information is not available in this tab as tuning parameters are identified by auto-tuning algorithm during execution. Use the Tuner GUI to read the parameter from device which provides actual scan time for sensor when is SmartSense enabled.

Gestures Tab

The **Gestures** tab provides gesture-related configuration parameters. It is available for gesture-supported part numbers only. If gestures are enabled, all gesture parameters are systematically arranged by widgets / gesture groups.

Note This version of the Component supports gesture detection on one widget at a time.



1. Click on a widget (in the left pane) to display all groups of gestures supported on the selected widget.
2. Use the check boxes (in the middle pane) to enable specific gesture groups or a combination of gesture groups for the selected widget.
3. Click on a gesture group in the middle pane to display the parameters associated with the selected gesture group.
4. Configure the parameters for each gesture groups in the right pane.

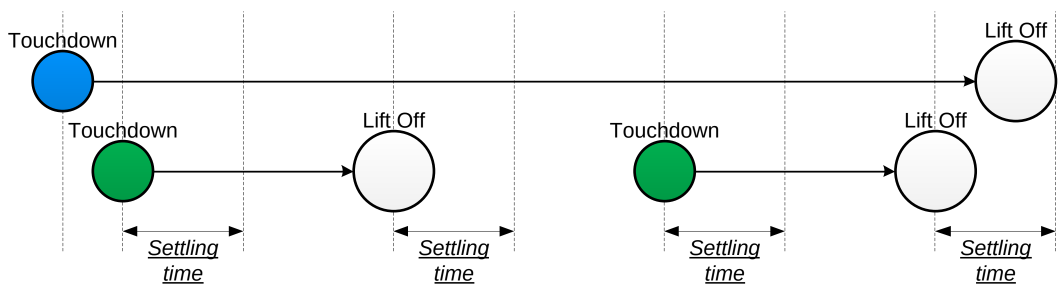
Note The Flick gestures and One-finger Scroll gestures cannot be enabled simultaneously.

Gesture groups include: Click, One-finger Scroll, Two-finger Scroll, Two-finger Zoom, One-finger Edge Swipe, One-finger Flick, One-finger Rotate. The table below shows the gesture groups supported in each widget type.

Widget Type	Gesture Groups						
	Click	One-finger Scroll	Two-finger Scroll	One-finger Flick	One-finger Edge Swipe	Two-finger Zoom	One-finger Rotate
Button							
Linear Slider	√	√		√			
Radial Slider	√						
Matrix Buttons							
Touchpad	√	√	√	√	√	√	√
Proximity							

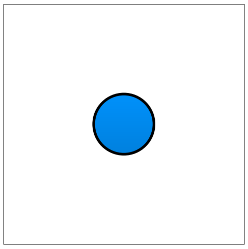
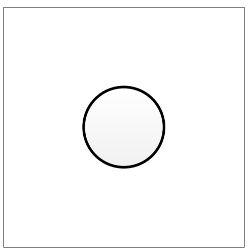
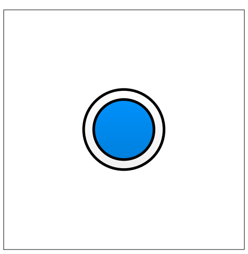
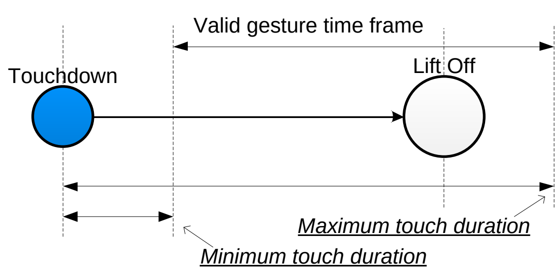
General Gesture Parameters

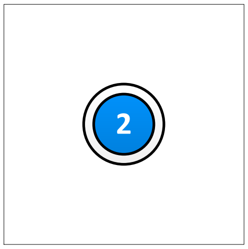
Contains the parameters common for gestures.

Name	Description
Enable gestures	Master enable for gestures feature.
Two-finger settling time (ms)	Sets a delay threshold that to be met before gestures are computed. This parameter helps avoid spurious gestures being reported during transient conditions. The parameter is applied for the following conditions. ▪ 1 touch → 2 touches ▪ 2 touches → 1 touch ▪ No touch → 2 touches.  Example: A false one-finger click may be reported during a two-finger click gesture, if the user lifts the fingers non-simultaneously (2 touches → 1 touch → no touch). Two-finger settling time can help avoid false reporting.

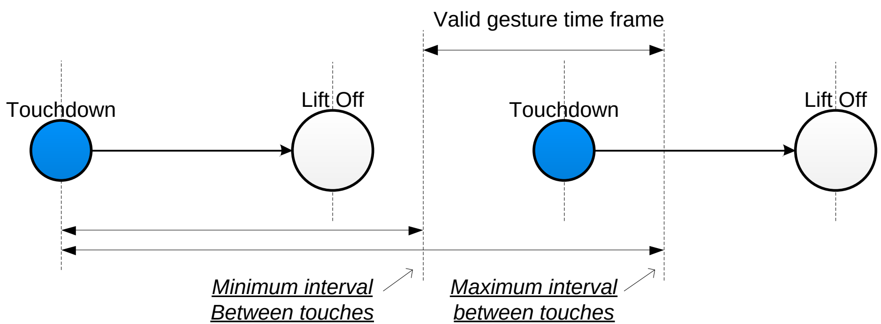
Click Group

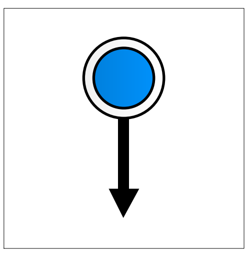
This group delivers the following gestures:

Touchdown 	A simple touch on a widget is reported as a Touchdown event.
Lift Off 	Removal of a finger from a widget reported as a Lift Off event. If the Lift Off event triggers another higher-level Gesture, then the Lift Off event is not reported.
One-finger Single Click 	One-finger single click gesture is a combination of a Touchdown and Lift Off events with the conditions to be met: ▪ A touchdown event is followed by a Lift Off event. ▪ The touch duration (duration between touchdown and lift off) must be greater than *One-finger minimum touch duration* and less than *One-finger maximum touch duration*. ▪ For a touchpad, position displacements in the X and Y axis between the Touchdown and Lift Off events must be within the click displacement limits (i.e. *Maximum X-axis position displacement* and *Maximum Y-axis position displacement*). ▪ For a slider, position displacements between the Touchdown and Lift Off events must be within the *Maximum position displacement*. 

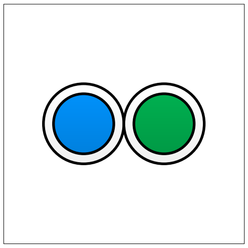
<table>
<tr>
<td>One-finger Double Click
</td>
<td>
A One-finger double click gesture is a combination of two sequential one-finger single click gestures under specific conditions:

Both clicks in the sequence must meet one-finger single click conditions.
The touch duration between the two touchdown events must be within the Minimum interval between touches and Maximum interval between touches timeout limits.
For a touchpad, the distance between two clicks must not exceed the Maximum radius for the second click.
For a slider, the distance between two clicks must not exceed the Maximum position displacement.

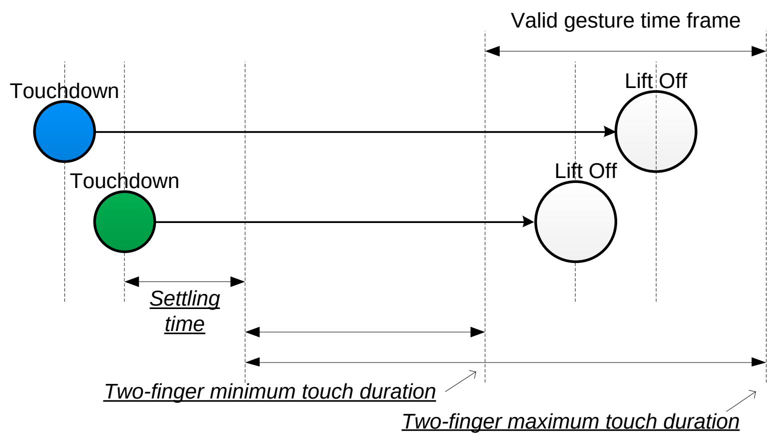


</td>
</tr>
<tr>
<td>One-finger Click and Drag
</td>
<td>
This gesture is a one-finger click and then a hold, followed by a drag. A typical use case is while moving items on the screen from one point to another. It is triggered when the finger movement follows this sequence: Touchdown → Lift Off → Touchdown → Drag

Gesture triggering condition: A one-finger click gesture and a subsequent touchdown were detected within the Minimum interval between touches and Maximum interval between touches timeout limits and within Maximum radius for the second click (for Touchpads) or Maximum displacement for second click (for Sliders). Then the finger exceeds the Maximum X-axis position displacement and Maximum Y-axis position displacement (for Touchpads) or Maximum position displacement (for Sliders) from a drag touchdown.
</td>
</tr>
</table>

<table>
<tr>
<td>Two-finger Click
</td>
<td>A Two-finger single click gesture is a combination of a Touchdown and Lift Off events with under specific conditions:

Two simultaneous finger touches (touchdown and lift off) should be detected.
The duration between the second finger touchdown and lift off events of both fingers must be within the Two-finger minimum touch duration and Two-finger maximum touch duration timeout limits. The duration counting starts when the settling time elapsed for the second finger touchdown event.
For a touchpad, a position displacement in the X and Y axes between a touchdown and lift off events must be less than the click displacement limits (i.e. Maximum X-axis position displacement and Maximum Y-axis position displacement).
For a slider, a position displacement between the touchdown and lift off events must be less than the Maximum position displacement.



</td>
</tr>
</table>

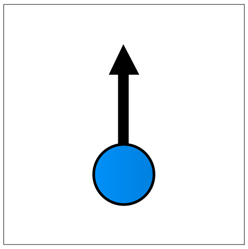
The following table shows the One-finger / Two-finger Click Group parameters:

Name	Description
Maximum X-axis position displacement	Defines the maximum X-axis displacement acceptable between a touchdown and lift off events for a valid one-finger click gesture on touchpad widget. The click gesture is not reported if the X-axis displacement is greater than the parameter value. Available for a Touchpad widget.
Maximum Y-axis position displacement	Defines the maximum Y-axis displacement acceptable between a touchdown and lift off events for a valid one-finger click gesture on a touchpad widget. The click gesture is not reported if the Y-axis displacement is greater than the parameter value. Available for a Touchpad widget.
Maximum position displacement	Defines the maximum displacement acceptable between a touchdown and lift off events for a valid one-finger click gesture on a slider widget. The click gesture is not reported if the position displacement is greater than the parameter value. Available for a Slider widget.

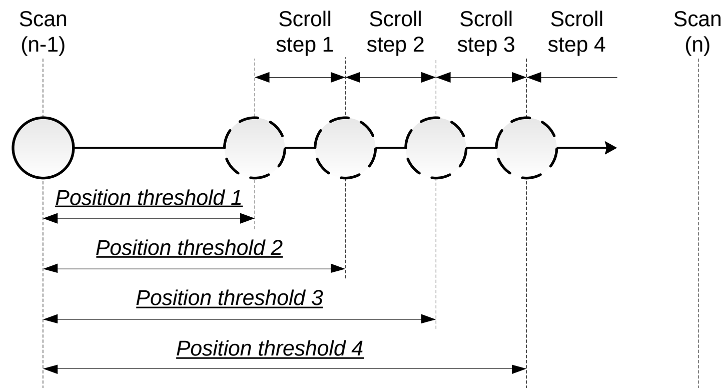
Name	Description
One-finger minimum touch duration (ms)	A duration between a touchdown and lift off events in a one-finger click must be greater than the minimum limit specified by the parameter for a one-finger click gesture to be valid. If the second click occurs within the *Minimum interval between touches*, no double click or click gesture is reported. Use this parameter to filter out a quick double click and short single click motions.
One-finger maximum touch duration (ms)	A duration between a touchdown and lift off events in a one-finger click must be less than the maximum limit specified by the parameter for a one-finger click gesture to be valid. If the finger remains on the widget for longer than this value, no click event is reported. This parameter also sets the maximum duration of how long each click of a one-finger double click can remain on the widget. If the first-click touch or second-click touch remains on the widget for longer than this value, the double click is not reported.
Maximum radius for the second click	Defines the maximum displacement (the center is the position of the first touch) that the second click in a one-finger double click can extend on a touchpad widget. If the second click occurs outside this radius limit, the double click is not reported. In this case, a Click and Drag gesture may be reported if the gesture sequence meets the conditions for the Click and Drag gesture.
Maximum displacement for second click	This parameter defines the maximum displacement (the center is the position of the first touch) that the second click in a one-finger double click can extend on a slider widget. If the second click occurs outside this displacement limit, the double click is not reported. In this case, a Click and Drag gesture may be reported if the gesture sequence meets the conditions for the Click and Drag gesture.
Minimum interval between touches (ms)	This parameter defines the minimum duration between two sequential clicks for a double click to be considered valid. If the second click occurs within the duration specified by this parameter, no click or double click gesture is reported. Use this parameter to filter out quick double-click motions.
Maximum interval between touches (ms)	This parameter defines the maximum duration allowed between two sequential touchdowns for a double click to be considered valid and reported. If the second touchdown occurs outside the duration specified by this parameter, no double click gesture is reported. Use this parameter to filter out slow double-click motions.
Two-finger minimum touch duration (ms)	This parameter defines the minimum duration between the second touchdown and the first lift off events in a two-finger click gesture to be considered valid. Use this parameter to filter out a quick two-finger click gesture.
Two-finger maximum touch duration (ms)	This parameter defines the minimum duration between the first touchdown and the second lift off events in a two-finger click gesture to be considered valid. Use this parameter to filter out a slow two-finger click gesture.

One-finger Scroll Group

This group delivers the following gestures:

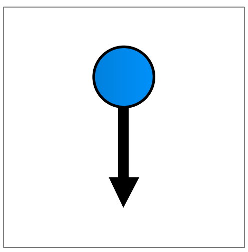
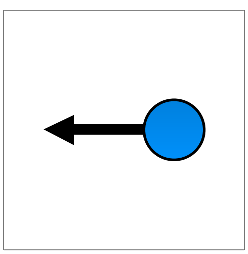
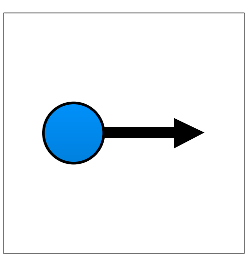
<table>
<tr>
<td>One-finger
Scroll Up
</td>
<td rowspan="4">
A One-finger Scroll gesture is a combination of a touchdown followed by a displacement in a specific direction under specific conditions:

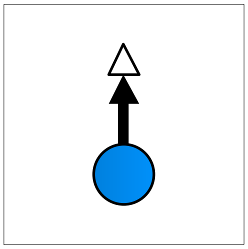
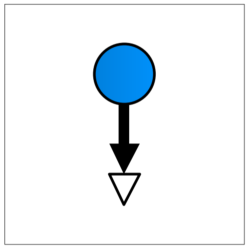
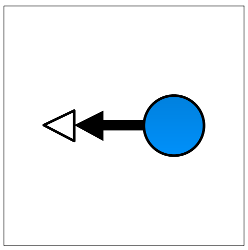
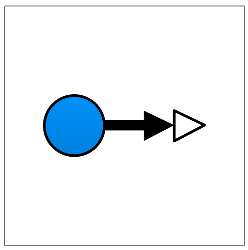
For a touchpad, the position displacement between two consecutive scans must exceed the X-axis position threshold N or Y-axis position threshold N.
For a slider, the position displacement between two consecutive scans must exceed the Position threshold N.
The Debounce number of a scroll gesture in the same direction is already detected.



Notes

If the displacement exceeds the position threshold between 2 consecutives scans, the corresponding scroll-step number (Scroll step N) is reported.
There are four levels of thresholds: If the displacement between two scans is greater than Position Threshold 1 and less than Position Threshold 2, then Scroll Step 1 is reported, and so on.
Scrolls in the four directions are detected and reported: Up, Down, Right, and Left.
The debounce logic ensures that the direction avoids incorrection results.

</td>
</tr>
<tr><td>One-finger
Scroll Down
</td></tr>
<tr><td>One-finger
Scroll Left
</td></tr>
<tr><td>One-finger
Scroll Right
</td></tr>
</table>

<table>
<tr>
<td>One-finger Scroll Inertial Up
</td>
<td>One-finger Scroll Inertial Down
</td>
<td rowspan="2">A one-finger inertial scroll gesture is reported for the specific duration after a one-finger inertial scroll gesture is followed by a lift off. A typical use case is scrolling through the pages.
The conditions for an inertial scroll gesture:
▪ A lift off is detected immediately after the scroll.
▪ For a touchpad, the position displacement between two consecutive scans must exceed the X-axis position inertial threshold or Y-axis position inertial threshold.
▪ For a slider, the position displacement between consecutive scans must exceed the Position Inertial Threshold.
Note
▪ If an inertial scroll is detected, the reported scroll value decays through the value selected by the Count level.</td>
</tr>
<tr>
<td>One-finger Scroll Inertial Left
</td>
<td>One-finger Scroll Inertial Right
</td>
</tr>
</table>

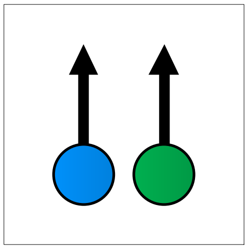
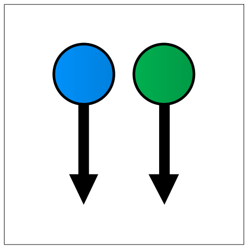
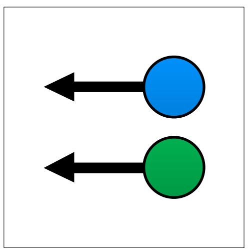
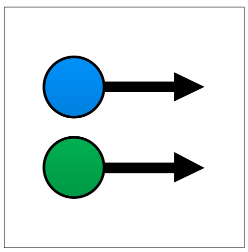
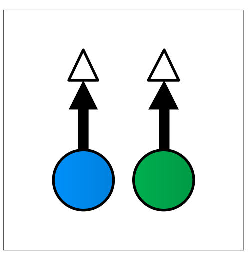
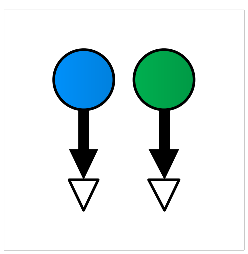
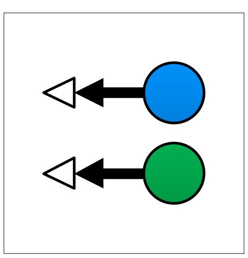
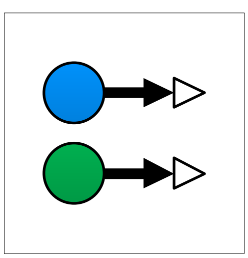
The following table shows the One-finger Scroll Group parameters:

Name	Description
X-axis position threshold N	Defines the minimum X-axis displacement to be detected on a touchpad between two consecutive scans for a one-finger scroll to be valid. The reported scroll number (*Scroll step N*) corresponds to the exceeded threshold N.
Y-axis position threshold N	Defines the minimum Y-axis displacement to be detected on a touchpad between two consecutive scans for a one-finger scroll to be valid. The reported scroll number (*Scroll step N*) corresponds to the exceeded threshold N.
Position threshold N	Defines the minimum displacement to be detected on a slider between two consecutive scans for a one-finger scroll to be valid. The reported scroll number (*Scroll step N*) corresponds to the exceeded threshold N.
Scroll step N	Defines the number of scrolls to be reported when the displacement between two consecutive scans exceeds the corresponding threshold N (*X-axis position threshold N* or *Y-axis position threshold N* for a Touchpad widget and *Position threshold N* for a Slider widget).
Debounce	Sets the number of similar, sequential scroll counts that to be detected prior to the scroll is considered valid. A widget must detect scroll counts, at the minimum of (Debounce + 1) times in the same direction to be considered as a scroll in that direction.
X-axis position inertial threshold	Defines the minimum displacement that to be detected on the X axis of a touchpad for a one-finger scroll gesture followed by a lift off event to be considered as a valid inertial scroll. Use this parameter to avoid accidental scroll triggers when fingers are removed from a touchpad after a scroll gesture.

Name	Description
Y-axis position inertial threshold	This parameter defines the minimum displacement that to be detected on the Y axis of a touchpad for a one-finger scroll gesture followed by a lift off event to be considered as a valid inertial scroll. Use this parameter to avoid accidental scroll triggers when fingers are removed from a touchpad after a scroll gesture.
Position Inertial Threshold	This parameter defines the minimum displacement that to be detected on a slider for a one-finger scroll gesture followed by a lift off event to be considered as a valid inertial scroll. Use this parameter to avoid accidental scroll triggers when fingers are removed from a slider after a scroll gesture.
Count level	This parameter selects the inertial scroll decay rate. The options are High and Low: ▪ **Low (default)** – Uses a 32-byte array for inertial scroll implementation, reports a few inertial scrolls. ▪ **High** – Uses a 64-byte array for inertial scroll implementation, reports more inertial scrolls.

Two-finger Scroll Group

This group delivers the following gestures:

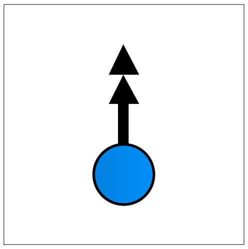
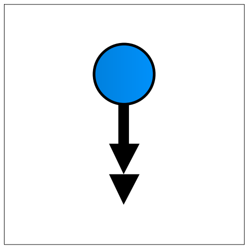
Two-finger Scroll Up 	Two-finger Scroll Down 	The design of a two-finger scroll gesture is the same as of a one-finger scroll gesture, except for the conditions below. ▪ The conditions of a one-finger scroll are met. ▪ There must be two simultaneous finger touches detected on a widget for a scroll to be considered as a two-finger scroll. ▪ The displacement of both finger touches must be on same direction for a two-finger scroll to be valid.
Two-finger Scroll Left 	Two-finger Scroll Right 	
Two-finger Scroll Inertial Up 	Two-finger Scroll Inertial Down 	
Two-finger Scroll Inertial Left 	Two-finger Scroll Inertial Right 	

The following table shows the Two-finger Scroll Group parameters:

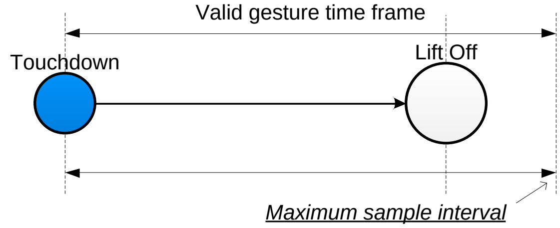
Name	Description
X-axis position threshold N	This parameter defines the minimum X-axis displacement that to be detected on a touchpad between two consecutive scans for a two-finger scroll to be valid. The reported scroll number (*Scroll step N*) corresponds to the threshold N exceeded by the displacement.
Y-axis position threshold N	This parameter defines the minimum Y-axis displacement that to be detected on a touchpad between two consecutive scans for a two-finger scroll to be valid. The reported scroll number (*Scroll step N*) corresponds to the threshold N exceeded by the displacement.
Position threshold N	This parameter defines the minimum displacement that to be detected on a slider between two consecutive scans for a two-finger scroll to be valid. The reported scroll number (*Scroll step N*) corresponds to the threshold N exceeded by the displacement.
Scroll step N	This parameter defines the number of scrolls that to be reported when a finger displacement between two consecutive scans exceeds the corresponding threshold N (*X-axis position threshold N* or *Y-axis position threshold N* for a Touchpad widget or *Position threshold N* for a Slider widget).
Debounce	Sets the number of similar, sequential scroll counts to be detected prior to a scroll is considered valid. A widget must detect scroll counts, the minimum of (Debounce + 1) times in the same direction to be considered as a scroll in that direction.
X-axis position inertial threshold	This parameter defines the minimum displacement that to be detected on the X axis of a touchpad for a two-finger scroll gesture followed by a lift off event to be considered as a valid inertial scroll. Use this parameter to avoid accidental scroll triggers when fingers are removed from a touchpad after a scroll gesture.
Y-axis position inertial threshold	This parameter defines the minimum displacement that to be detected on the Y axis of a touchpad for a two-finger scroll gesture followed by a lift off event to be considered as a valid inertial scroll. Use this parameter to avoid accidental scroll triggers when fingers are removed from a touchpad after a scroll gesture.
Position Inertial Threshold	This parameter defines the minimum displacement that to be detected on a slider for a two-finger scroll gesture followed by a lift off event to be considered as a valid inertial scroll. Use this parameter to avoid accidental scroll triggers when fingers are removed from a slider after a scroll gesture.
Count level	This parameter selects the inertial scroll decay rate. The options are High and Low: ▪ **Low (default)** – Uses a 32-byte array for inertial scroll implementation, reports a few inertial scrolls. ▪ **High** – Uses a 64-byte array for inertial scroll implementation, reports more inertial scrolls.

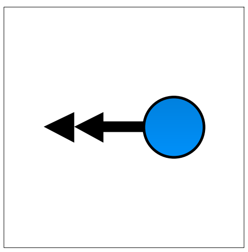
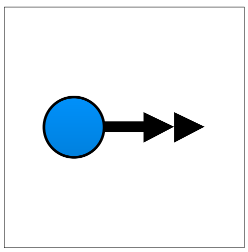
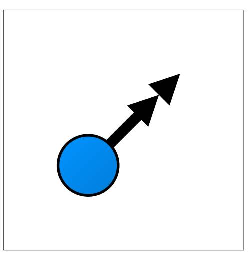
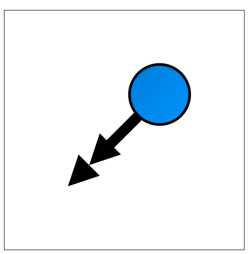
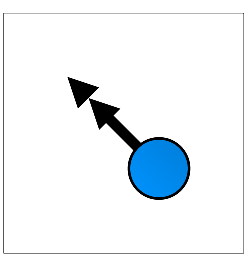
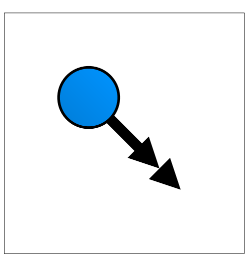
One-finger Flick Group

This group delivers the following gestures:

<table>
<tr>
<td>One-finger Flick Up
</td>
<td>One-finger Flick Down
</td>
<td rowspan="4">
A flick gesture is a combination of a touchdown followed by a high-speed displacement and a lift off event.
A flick gesture starts at a touchdown and ends and reported at a lift off event. The conditions for a flick gesture.

For a touchpad, the displacement must exceed the X-axis position threshold or Y-axis position threshold.
For a slider, the displacement must exceed the Position threshold.
The duration between a touchdown and lift off events must be less than the Maximum sample interval.



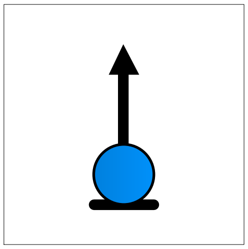
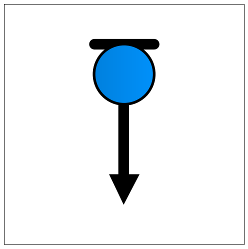
Note
The flick gesture is detected in 8 directions:
- Up
- Down
- Left
- Right
- Up-Right
- Down-Left
- Up-Left
- Down-Right
</td>
</tr>
<tr>
<td>One-finger Flick Left
</td>
<td>One-finger Flick Right
</td>
</tr>
<tr>
<td>One-finger Flick Up-Right
</td>
<td>One-finger Flick Down-Left
</td>
</tr>
<tr>
<td>One-finger Flick Up-Left
</td>
<td>One-finger Flick Down-Right
</td>
</tr>
</table>

The following table shows the One-finger Flick Group parameters:

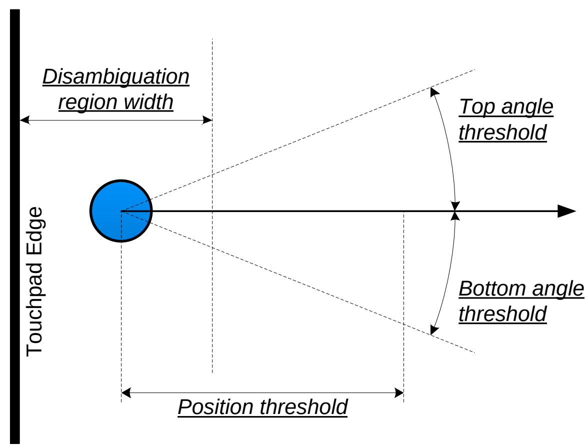
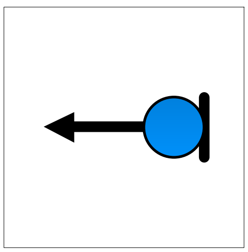
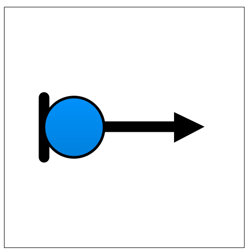
Name	Description
X-axis position threshold	Defines the minimum displacement that to be detected on the X-axis of a touchpad between two consecutive scans for a one-finger flick to be valid.
Y-axis position threshold	Defines the minimum displacement that to be detected on the Y-axis of a touchpad between two consecutive scans for a one-finger flick to be valid.
Position threshold	Defines the minimum displacement to be detected on a slider between two consecutive scans for a one-finger flick to be valid.
Maximum sample interval (ms)	Defines the maximum duration of how long a flick gesture is searched after a touchdown event. A position displacement and lift off event must happen within the duration defined by this parameter for a flick to be valid.

One-finger Edge Swipe Group

This group delivers the following gestures:

<table>
<tr>
<td>One-finger Edge Swipe Up
</td>
<td>One-finger Edge Swipe Down
</td>
<td rowspan="2">An edge swipe gesture is a combination of a touchdown on an edge followed by a displacement towards the center.

The conditions for an edge swipe gesture:

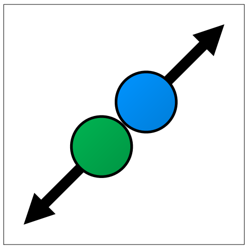
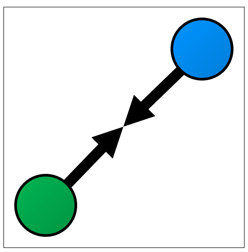
▪ A touchdown event must occur in the edge area defined by the Disambiguation region width.
▪ A finger displacement must occur from the edge towards the center within the angular threshold Top angle threshold and Bottom angle threshold.
▪ The displacement must exceed the Position threshold within the Detection time duration.

</td>
</tr>
<tr>
<td>One-finger Edge Swipe Left
</td>
<td>One-finger Edge Swipe Right
</td>
</tr>
</table>

The following table shows the One-finger Edge Swipe Group parameters:

Name	Description
Disambiguation region width	Defines the maximum edge area where a touchdown must be detected for an edge swipe to be reported.
Position threshold	Defines the minimum displacement to be detected from an edge to the center for an edge swipe to be reported.
Detection time (ms)	Defines the maximum duration within which an edge swipe must occur to be reported. The displacement must exceed the *Position threshold* within the duration defined by this parameter for the edge swipe to be reported.
Timeout interval	Defines the time interval for which all other gestures will be ignored after the of a one-finger edge swipe gesture.
Top angle threshold (degree)	Defines the maximum angles (in degrees) that the displacement path of a finger can subtend at the point of a touch-down, near the edge. Degree 1 means that the user can do gestures only on a single line.
Bottom angle threshold (degree)	

Two-finger Zoom Group

This group delivers the following gestures:

<table>
<tr><td>Two-finger Zoom In
</td><td rowspan="2">A two-finger zoom gesture is reported when two touches move towards each other (Zoom Out) or move away from each other (Zoom In).
The conditions for a zoom gesture:
An increase or decrease in distance between two-finger touch positions in X must exceed the X-axis position threshold or the Y axis must exceed the Y-axis position threshold.
The Debounce number of a Zoom In or Zoom Out gesture must be sequentially detected for a Zoom gesture to be reported.
A scroll to the zoom debounce number of a zoom gestures must be sequentially detected for a Zoom gesture to be reported. If a Zoom gesture occurred after a scroll, the gesture is reported and there was no lift off event between the scroll and Zoom gestures.</td></tr>
<tr><td>Two-finger Zoom Out
</td></tr>
</table>

The following table shows the Two-finger Zoom Group parameters:

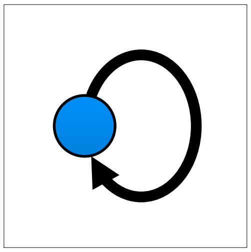
Name	Description
X-axis position threshold	This parameter defines the minimum displacement that to be detected on the X-axis of a touchpad between two consecutive scans for a two-finger Zoom to be reported.
Y-axis position threshold	This parameter defines the minimum displacement that to be detected on the Y-axis of a touchpad between two consecutive scans for a two-finger Zoom to be reported.
Debounce	This parameter defines the number of sequential zoom gestures in a particular direction (in or out) that to be detected before a zoom gesture is deemed valid.
Scroll to zoom debounce	If a scroll was detected and then a zoom is detected without a lift off event (i.e. without removing fingers from a touchpad), the first few zoom gestures specified by this parameter are ignored before reporting a zoom gesture.

One-finger Rotate Group

This group delivers the following gestures:

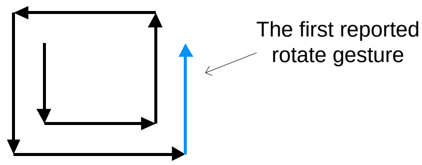
<table>
<tr>
<td>One-finger
Rotate CW

(Clockwise)

</td>
<td rowspan="2">A one-finger rotate gesture is reported when a circular displacement is detected. The decoding algorithm uses four directions to identify a circular displacement. A displacement in all four directions must be in the succession order to report a rotate gesture. The rotation direction can be clockwise or counter-clockwise.

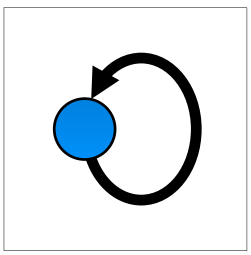
The conditions for a zoom gesture:

▪ A displacement in the four directions (UP, DOWN, RIGHT and LEFT) in the succession order must be detected for a rotate gesture to be reported.
▪ At least one and a half circular displacement must be reported for a rotate gesture to be reported.
▪ A detected scroll count must be less than the Debounce value.
▪ Finger movement must exceed the displacement thresholds X-axis position threshold N and Y-axis position threshold N defined in One-finger Scroll Group.



To determine a four-direction value, a motion must be present. The motion of a touch object must exceed the displacement threshold belonging to One-finger Scroll Group: X-axis position threshold N and Y-axis position threshold N (where N = 1).</td>
</tr>
<tr>
<td>One-finger
Rotate CCW

(Counter-clockwise)

</td>
</tr>
</table>

The following table shows the One-finger Rotate Group parameter:

Name	Description
Debounce	This parameter sets the number of sequential scroll counts in a particular direction to deem a rotate gesture invalid. For example, if the Debounce value is set to 20, then the touch cannot continue in the same direction for 20 scroll counts and still have a valid rotate gesture. After this threshold, the reported gesture stops being a rotate gesture. If this parameter is set to 0, then the Debounce is disabled. All rotate gestures will be considered valid and no scroll gestures will be detected until the rotate condition is broken.

Application Programming Interface

The Application Programming Interface (API) routines allow controlling and executing specific tasks using the Component firmware. The following sections list and describe each function and dependency.

Note The CapSense_P4 v5.X firmware API is very different compared with the third-generation API of the CapSense_CSD_P4 Component (v2.60 and before). In addition to the new CSX features, the API has been optimized to reduce power consumption and user code complexity. As a result, applications that run on the older Component will require significant changes to the firmware if you change the design to use the new Component. Refer to the *Step-7: API Comparison* section for details on migrating your firmware to new CapSense API.

The compilers the CapSense firmware library supports:

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

To use the IAR Embedded Workbench, refer to the PSoC Creator Help > Integrating into 3rd Party IDEs section.

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

PSoC Creator\psoc\content\CyComponentLibrary\CyComponentLibrary.cylib\ CapSense_P4_**vX_XX**\PSoC4\

(Replace vX_XX with the Component version)

By default, the instance name of the Component is “CapSense_1” for the first instance of a Component in a given design. It can be renamed to any unique text that follows the syntactic rules for identifiers. The instance name is prefixed to every function, variable, and constant name. For readability, this section assumes “CapSense” as the instance name.

CapSense High-Level APIs

Description

High-level APIs represent the highest abstraction layer of the component APIs. These APIs perform tasks such as scanning, data processing, data reporting and tuning interfaces. When performing a task, different initialization is required based on a the sensing method or type of widgets is automatically handled by these APIs, therefore these APIs are sensing methods, features and widget type agnostics.

All the tasks required to implement a sensing system can be fulfilled by the high-level APIs. But, there is a set of CapSense Low-Level APIs which provides access to lower level and specific tasks. If a design require access to low-level tasks, these APIs can be used. The functions related to a given sensing methods are not available if the corresponding method is disabled.

Functions

- cystatus CapSense_Start(void)
  Initializes the Component hardware and firmware modules. This function is called by the application program prior to calling any other function of the Component.
- cystatus CapSense_Stop(void)
  Stops the Component operation.
- cystatus CapSense_Resume(void)
  Resumes the Component operation if the CapSense_Stop() function was called previously.
- cystatus CapSense_ProcessAllWidgets(void)
  Performs full data processing of all enabled widgets.
- cystatus CapSense_ProcessWidget(uint32 widgetId)
  Performs full data processing of the specified widget if it is enabled.
- void CapSense_Sleep(void)
  Prepares the Component for deep sleep.
- void CapSense_Wakeup(void)
  Resumes the Component after deep sleep power mode.
- uint32 CapSense_DecodeWidgetGestures(uint32 widgetId)
  Decodes all enabled gestures for the specified widget and returns the gesture code.
- void CapSense_IncrementGestureTimestamp(void)
  Increases the timestamp register for the predefined timestamp interval.
- void CapSense_SetGestureTimestamp(uint32 timestampValue)
  Rewrites the timestamp register by the specified value.
- uint32 CapSense_RunSelfTest(uint32 testEnMask)
  Runs built-in self-tests specified by the test enable mask.
- cystatus CapSense_SetupWidget(uint32 widgetId)
  Performs the initialization required to scan the specified widget.
- cystatus CapSense_Scan(void)
  Initiates scanning of all the sensors in the widget initialized by CapSense_SetupWidget(), if no scan is in progress.
- cystatus CapSense_ScanAllWidgets(void)
  Initializes the first enabled widget and scanning of all the sensors in the widget, then the same process is repeated for all the widgets in the Component, i.e. scanning of all the widgets in the Component.
- uint32 CapSense_IsBusy(void)

Returns the current status of the Component (Scan is completed or Scan is in progress).

- uint32 CapSense_IsAnyWidgetActive(void)
Reports if any widget has detected a touch.
- uint32 CapSense_IsWidgetActive(uint32 widgetId)
Reports if the specified widget detects a touch on any of its sensors.
- uint32 CapSense_IsSensorActive(uint32 widgetId, uint32 sensorId)
Reports if the specified sensor in the widget detects a touch.
- uint32 CapSense_IsProximitySensorActive(uint32 widgetId, uint32 proxId)
Reports the finger detection status of the specified proximity widget/sensor.
- uint32 CapSense_IsMatrixButtonsActive(uint32 widgetId)
Reports the status of the specified matrix button widget.
- uint32 CapSense_GetCentroidPos(uint32 widgetId)
Reports the centroid position for the specified slider widget.
- uint32 CapSense_GetXYCoordinates(uint32 widgetId)
Reports the X/Y position detected for the specified touchpad widget.
- uint32 CapSense_RunTuner(void)
Establishes synchronized communication with the Tuner application.

Function Documentation

cystatus CapSense_Start (void)

This function initializes the Component hardware and firmware modules and is called by the application program prior to calling any other API of the Component. When this function is called, the following tasks are executed as part of the initialization process:

1. Initialize the registers of the Data Structure variable CapSense_dsRam based on the user selection in the Component configuration wizard.
2. Configure the hardware to perform capacitive sensing.
3. If SmartSense Auto-tuning is selected for the CSD Tuning mode in the Basic tab, the auto-tuning algorithm is executed to set the optimal values for the hardware parameters of the widgets/sensors.
    1. Calibrate the sensors and find the optimal values for IDACs of each widget / sensor, if the Enable IDAC auto-calibration is enabled in the CSD Setting or CSX Setting tabs.
4. Perform scanning for all the sensors and initialize the baseline history.
5. If the firmware filters are enabled in the Advanced General tab, the filter histories are also initialized.

Any next call of this API repeats an initialization process except for data structure initialization. Therefore, it is possible to change the Component configuration from the application program by writing registers to the data structure and calling this function again. This is also done inside the CapSense_RunTuner() function when a restart command is received.

When the Component operation is stopped by the CapSense_Stop() function, the CapSense_Start() function repeats an initialization process including data structure initialization.

Returns:

Returns the status of the initialization process. If CYRET_SUCCESS is not received, some of the initialization fails and the Component may not operate as expected.

Go to the top of the CapSense High-Level APIs section.

cystatus CapSense_Stop (void)

This function stops the Component operation, no sensor scanning can be executed when the Component is stopped. Once stopped, the hardware block may be reconfigured by the application program for any other special usage. The Component operation can be resumed by calling the CapSense_Resume() function or the Component can be reset by calling the CapSense_Start() function.

This function is called when no scanning is in progress. I.e. CapSense_IsBusy() returns a non-busy status.

Returns:

Returns the status of the stop process. If CYRET_SUCCESS is not received, the stop process fails and retries may be required.

Go to the top of the CapSense High-Level APIs section.

cystatus CapSense_Resume (void)

This function resumes the Component operation if the operation is stopped previously by the CapSense_Stop() function. The following tasks are executed as part of the operation resume process:

1. Reset all the Widgets/Sensors statuses.
2. Configure the hardware to perform capacitive sensing.

Returns:

Returns the status of the resume process. If CYRET_SUCCESS is not received, the resume process fails and retries may be required.

Go to the top of the CapSense High-Level APIs section.

cystatus CapSense_ProcessAllWidgets (void)

This function performs all data processes for all enabled widgets in the Component. The following tasks are executed as part of processing all the widgets:

1. Apply raw count filters to the raw counts, if they are enabled in the customizer.
2. Update the thresholds if the SmartSense Full Auto-Tuning is enabled in the customizer.
3. Update the baselines and difference counts for all the sensors.
4. Update the sensor and widget status (on/off), update the centroid for the sliders and the X/Y position for the touchpads.

This function is called by an application program only after all the enabled widgets (and sensors) in the Component is scanned. Calling this function multiple times without sensor scanning causes unexpected behavior.

The disabled widgets are not processed by this function. To disable/enable a widget, set the appropriate values in the CapSense_WDGT_ENABLE<RegisterNumber>_PARAM_ID register using the CapSense_SetParam() function.

If the Ballistic multiplier filter is enabled the Timestamp must be updated before calling this function using the CapSense_IncrementGestureTimestamp() function.

If the Self-test library is enabled, this function executes the baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Returns:

Returns the status of the processing operation. If CYRET_SUCCESS is not received, the processing fails and retries may be required.

Go to the top of the CapSense High-Level APIs section.

cystatus CapSense_ProcessWidget (uint32 *widgetId*)

This function performs exactly the same tasks as CapSense_ProcessAllWidgets(), but only for a specified widget. This function can be used along with the CapSense_SetupWidget() and CapSense_Scan() functions to scan and process data for a specific widget. This function is called only after all the sensors in the widgets are scanned. A disabled widget is not processed by this function.

A pipeline scan method (i.e. during scanning of a widget perform processing of the previously scanned widget) can be implemented using this function and it may reduce the total execution time, increase the refresh rate and decrease the average power consumption.

If the Ballistic multiplier filter is enabled the Timestamp must be updated before calling this function using the CapSense_IncrementGestureTimestamp() function.

If the Self-test library is enabled, this function executes the baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID

Returns:

Returns the status of the widget processing:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_PARAM - The input parameter is invalid.
- CYRET_INVALID_STATE - The specified widget is disabled.
- CYRET_BAD_DATA - The processing is failed.

Go to the top of the CapSense High-Level APIs section.

void CapSense_Sleep (void)

Currently this function is empty and exists as a place for future updates, this function will be used to prepare the Component to enter deep sleep.

Go to the top of the CapSense High-Level APIs section.

void CapSense_Wakeup (void)

Resumes the Component after deep sleep power mode. This function is used to resume the Component after exiting deep sleep.

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_DecodeWidgetGestures (uint32 *widgetId*)

This function decodes all the enabled gestures on a specific widget and returns a code for the detected gesture. Refer to the Gesture tab section for more details on supported Gestures.

This function is called only after scan and data processing are completed for the specified widget.

The Timestamp must be updated before calling this function using the CapSense_IncrementGestureTimestamp() function.

Parameters:

widgetId	Specifies the ID number of the widget to decode the gesture. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Returns:

Returns the status of the gesture detection or the detected gesture code:

- CapSense_NON_VALID_PARAMETER
- CapSense_NO_GESTURE
- CapSense_UNRECOGNIZED_GESTURE
- CapSense_ONE_FINGER_TOUCHDOWN
- CapSense_ONE_FINGER_LIFT_OFF
- CapSense_ONE_FINGER_SINGLE_CLICK
- CapSense_ONE_FINGER_DOUBLE_CLICK
- CapSense_ONE_FINGER_CLICK_AND_DRAG
- CapSense_ONE_FINGER_SCROLL_UP
- CapSense_ONE_FINGER_SCROLL_DOWN
- CapSense_ONE_FINGER_SCROLL_RIGHT
- CapSense_ONE_FINGER_SCROLL_LEFT
- CapSense_ONE_FINGER_SCROLL_INERTIAL_UP

- CapSense_ONE_FINGER_SCROLL_INERTIAL_DOWN
- CapSense_ONE_FINGER_SCROLL_INERTIAL_RIGHT
- CapSense_ONE_FINGER_SCROLL_INERTIAL_LEFT
- CapSense_ONE_FINGER_FLICK_UP
- CapSense_ONE_FINGER_FLICK_DOWN
- CapSense_ONE_FINGER_FLICK_RIGHT
- CapSense_ONE_FINGER_FLICK_LEFT
- CapSense_ONE_FINGER_FLICK_UP_RIGHT
- CapSense_ONE_FINGER_FLICK_DOWN_RIGHT
- CapSense_ONE_FINGER_FLICK_DOWN_LEFT
- CapSense_ONE_FINGER_FLICK_UP_LEFT
- CapSense_ONE_FINGER_EDGE_SWIPE_UP
- CapSense_ONE_FINGER_EDGE_SWIPE_DOWN
- CapSense_ONE_FINGER_EDGE_SWIPE_RIGTH
- CapSense_ONE_FINGER_EDGE_SWIPE_LEFT
- CapSense_ONE_FINGER_ROTATE_CW
- CapSense_ONE_FINGER_ROTATE_CCW
- CapSense_TWO_FINGER_SINGLE_CLICK
- CapSense_TWO_FINGER_SCROLL_UP
- CapSense_TWO_FINGER_SCROLL_DOWN
- CapSense_TWO_FINGER_SCROLL_RIGHT
- CapSense_TWO_FINGER_SCROLL_LEFT
- CapSense_TWO_FINGER_SCROLL_INERTIAL_UP
- CapSense_TWO_FINGER_SCROLL_INERTIAL_DOWN
- CapSense_TWO_FINGER_SCROLL_INERTIAL_RIGHT
- CapSense_TWO_FINGER_SCROLL_INERTIAL_LEFT
- CapSense_TWO_FINGER_ZOOM_IN
- CapSense_TWO_FINGER_ZOOM_OUT

Go to the top of the CapSense High-Level APIs section.

void CapSense_IncrementGestureTimestamp (void)

This function increments the Component timestamp (CapSense_TIMESTAMP_VALUE register) by the interval specified in the CapSense_TIMESTAMP_INTERVAL_VALUE register. The unit for both registers is millisecond and default value of CapSense_TIMESTAMP_INTERVAL_VALUE is 1.

It is the application layer responsibility to periodically call this function or register a periodic callback to this function to keep the Component timestamp updated and operational, which is vital for the operation of Gesture and Ballistic multiplier features.

The Component timestamp can be updated in one of the three methods:

- Register a periodic callback for the CapSense_IncrementGestureTimestamp() function.
- Periodically call the CapSense_IncrementGestureTimestamp() function by application layer.
- Directly modify the timestamp using the CapSense_SetGestureTimestamp() function.

The interval at which this function is called should match with interval defined in CapSense_TIMESTAMP_INTERVAL_VALUE register. Either the register value can be updated to match the callback interval or the callback can be made at interval set in the register.

If a timestamp is available from another source or from host controller, application layer may choose to periodically update the Component timestamp by using CapSense_SetGestureTimestamp() function instead of registering a callback.

Go to the top of the CapSense High-Level APIs section.

void CapSense_SetGestureTimestamp (uint32 *timestampValue*)

This function writes the specified value into the Component timestamp (i.e. CapSense_TIMESTAMP_VALUE register).

If a timestamp is available from another source or from host controller, application layer may choose to periodically update the Component timestamp by using this function instead of registering a callback.

It is not recommended to modify the Component timestamp arbitrarily or simultaneously use with the CapSense_IncrementGestureTimestamp() function.

Parameters:

timestampValue	Specifies the timestamp value (in ms).

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_RunSelfTest (uint32 *testEnMask*)

The function performs various self-tests on all the enabled widgets and sensors in the Component. The required set of tests can be selected using the bit-mask in testEnMask parameter.

Use CapSense_TST_RUN_SELF_TEST_MASK to execute all the self-tests or any combination of the masks (defined in testEnMask parameter) to specify the test list.

To execute a single-element test (i.e. for one widget or sensor), the following functions available:

- CapSense_CheckGlobalCRC()
- CapSense_CheckWidgetCRC()
- CapSense_CheckBaselineDuplication()
- CapSense_CheckSensorShort()
- CapSense_CheckSns2SnsShort()
- CapSense_GetSensorCapacitance()
- CapSense_GetShieldCapacitance()
- CapSense_GetExtCapCapacitance()
- CapSense_GetVdda()

Refer to these functions for detail information on the corresponding test.

Parameters:

testEnMask	Specifies the tests to be executed. Each bit corresponds to one test. It is possible to launch the function with any combination of the available tests. • CapSense_TST_GLOBAL_CRC - Verifies the RAM structure CRC of global parameters • CapSense_TST_WDGT_CRC - Verifies the RAM widget structure CRC for all the widgets • CapSense_TST_BSLN_DUPLICATION - Verifies the baseline consistency of all the sensors (inverse copy) • CapSense_TST_SNS_SHORT - Checks all the sensors for a short to GND or VDD • CapSense_TST_SNS2SNS_SHORT - Checks all the sensors for a short to other sensors • CapSense_TST_SNS_CAP - Measures all the sensors capacitance • CapSense_TST_SH_CAP - Measures the shield capacitance • CapSense_TST_EXTERNAL_CAP - Measures the capacitance of the available external capacitors • CapSense_TST_VDDA - Measures the Vdda voltage

	• CapSense_TST_RUN_SELF_TEST_MASK - Executes all available tests.

Returns:

Returns a bit-mask with a status of execution of the specified tests:

- CY_RET_SUCCESS - All the tests passed.
- CapSense_TST_NOT_EXECUTED - The previously triggered scanning is not completed.
- CapSense_TST_BAD_PARAM - A non-defined test was requested in the testEnMask parameter.
- The bit-mask of the failed tests.

Go to the top of the CapSense High-Level APIs section.

cystatus CapSense_SetupWidget (uint32 *widgetId*)

This function prepares the Component to scan all the sensors in the specified widget by executing the following tasks:

1. Re-initialize the hardware if it is not configured to perform the sensing method used by the specified widget, this happens only if multiple sensing methods are used in the Component.
2. Initialize the hardware with specific sensing configuration (e.g. sensor clock, scan resolution) used by the widget.
3. Disconnect all previously connected electrodes, if the electrodes connected by the CapSense_CSDSetupWidgetExt(), CapSense_CSXSetupWidgetExt(), CapSense_CSDConnectSns() functions and not disconnected.

This function does not start sensor scanning, the CapSense_Scan() function must be called to start the scan sensors in the widget. If this function is called more than once, it does not break the Component operation, but only the last initialized widget is in effect.

Parameters:

widgetId	Specifies the ID number of the widget to be initialized for scanning. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Returns:

Returns the status of the widget setting up operation:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_PARAM - The widget is invalid or if the specified widget is disabled
- CYRET_INVALID_STATE - The previous scanning is not completed and the hardware block is busy.
- CYRET_UNKNOWN - An unknown sensing method is used by the widget or any other spurious error occurred.

Go to the top of the CapSense High-Level APIs section.

cystatus CapSense_Scan (void)

This function is called only after the CapSense_SetupWidget() function is called to start the scanning of the sensors in the widget. The status of a sensor scan must be checked using the CapSense_IsBusy() API prior to starting a next scan or setting up another widget.

Returns:

Returns the status of the scan initiation operation:

- CYRET_SUCCESS - Scanning is successfully started.
- CYRET_INVALID_STATE - The previous scanning is not completed and the hardware block is busy.
- CYRET_UNKNOWN - An unknown sensing method is used by the widget.

Go to the top of the CapSense High-Level APIs section.

cystatus CapSense_ScanAllWidgets (void)

This function initializes a widget and scans all the sensors in the widget, and then repeats the same for all the widgets in the Component. The tasks of the CapSense_SetupWidget() and CapSense_Scan() functions are executed by these functions. The status of a sensor scan must be checked using the CapSense_IsBusy() API prior to starting a next scan or setting up another widget.

Returns:

Returns the status of the operation:

- CYRET_SUCCESS - Scanning is successfully started.
- CYRET_BAD_PARAM - All the widgets are disabled.
- CYRET_INVALID_STATE - The previous scanning is not completed and the HW block is busy.
- CYRET_UNKNOWN - There are unknown errors.

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_IsBusy (void)

This function returns a status of the hardware block whether a scan is currently in progress or not. If the Component is busy, no new scan or setup widgets is made. The critical section (i.e. disable global interrupt) is recommended for the application when the device transitions from the active mode to sleep or deep sleep modes.

Returns:

Returns the current status of the Component:

- CapSense_NOT_BUSY - No scan is in progress and a next scan can be initiated.
- CapSense_SW_STS_BUSY - The previous scanning is not completed and the hardware block is busy.

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_IsAnyWidgetActive (void)

This function reports if any widget has detected a touch or not by extracting information from the wdgtStatus registers (CapSense_WDGT_STATUS<X>_VALUE). This function does not process a widget but extracts processed results from the Data Structure.

Returns:

Returns the touch detection status of all the widgets:

- Zero - No touch is detected in all the widgets or sensors.
- Non-zero - At least one widget or sensor detected a touch.

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_IsWidgetActive (uint32 *widgetId*)

This function reports if the specified widget has detected a touch or not by extracting information from the wdgtStatus registers (CapSense_WDGT_STATUS<X>_VALUE). This function does not process the widget but extracts processed results from the Data Structure.

Parameters:

widgetId	Specifies the ID number of the widget to get its status. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Returns:

Returns the touch detection status of the specified widgets:

- Zero - No touch is detected in the specified widget or a wrong widgetId is specified.
- Non-zero if at least one sensor of the specified widget is active, i.e. a touch is detected.

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_IsSensorActive (uint32 *widgetId*, uint32 *sensorId*)

This function reports if the specified sensor in the widget has detected a touch or not by extracting information from the wdgtStatus registers (CapSense_WDGT_STATUS<X>_VALUE). This function does not process the widget or sensor but extracts processed results from the Data Structure.

For proximity sensors, this function returns the proximity detection status. To get the finger touch status of proximity sensors, use the CapSense_IsProximitySensorActive() function.

Parameters:

widgetId	Specifies the ID number of the widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget to get its touch detection status. A macro for the sensor ID within the specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.

Returns:

Returns the touch detection status of the specified sensor / widget:

- Zero if no touch is detected in the specified sensor / widget or a wrong widget ID / sensor ID is specified.
- Non-zero if the specified sensor is active i.e. touch is detected. If the specific sensor belongs to a proximity widget, the proximity detection status is returned.

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_IsProximitySensorActive (uint32 *widgetId*, uint32 *proxId*)

This function reports if the specified proximity sensor has detected a touch or not by extracting information from the wdgtStatus registers (CapSense_SNS_STATUS<WidgetId>_VALUE). This function is used only with proximity sensor widgets. This function does not process the widget but extracts processed results from the Data Structure.

Parameters:

widgetId	Specifies the ID number of the proximity widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID
proxId	Specifies the ID number of the proximity sensor within the proximity widget to get its touch detection status. A macro for the proximity ID within a specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID

Returns:

Returns the status of the specified sensor of the proximity widget. Zero indicates that no touch is detected in the specified sensor / widget or a wrong widgetId / proxId is specified.

- Bits [31..2] are reserved.
- Bit [1] indicates that a touch is detected.
- Bit [0] indicates that a proximity is detected.

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_IsMatrixButtonsActive (uint32 *widgetId*)

This function reports if the specified matrix widget has detected a touch or not by extracting information from the wdgtStatus registers (CapSense_WDGT_STATUS<X>_VALUE for the CSD widgets and CapSense_SNS_STATUS<WidgetId>_VALUE for CSX widget). In addition, the function provides details of the active sensor including active rows/columns for the CSD widgets. This function is used only with the matrix button widgets. This function does not process the widget but extracts processed results from the Data Structure.

Parameters:

widgetId	Specifies the ID number of the matrix button widget to check the status of its sensors. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID

Returns:

Returns the touch detection status of the sensors in the specified matrix buttons widget. Zero indicates that no touch is detected in the specified widget or a wrong widgetId is specified.

1. For the matrix buttons widgets with the CSD sensing mode:
   - Bit [31] if set, indicates that one or more sensors in the widget detected a touch.
   - Bits [30..24] are reserved
   - Bits [23..16] indicate the logical sensor number of the sensor that detected a touch. If more than one sensor detected a touch for the CSD widget, no status is reported because more than one touch is invalid for the CSD matrix buttons widgets.
   - Bits [15..8] indicate the active row number.
   - Bits [7..0] indicate the active column number.
2. For the matrix buttons widgets with the CSX widgets, each bit (31..0) corresponds to the TX/RX intersection.

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_GetCentroidPos (uint32 *widgetId*)

This function reports the centroid value of a specified radial or linear slider widget by extracting information from the wdgtStatus registers (CapSense_<WidgetName>_POSITION<X>_VALUE). This function is used only with radial or linear slider widgets. This function does not process the widget but extracts processed results from the Data Structure.

Parameters:

widgetId	Specifies the ID number of a slider widget to get the centroid of the detected touch. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID

Returns:

Returns the centroid position of a specified slider widget:

- The centroid position if a touch is detected.
- CapSense_SLIDER_NO_TOUCH - No touch is detected or a wrong widgetId is specified.

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_GetXYCoordinates (uint32 *widgetId*)

This function reports a touch position (X and Y coordinates) value of a specified touchpad widget by extracting information from the wdgtStatus registers (CapSense_<WidgetName>_POS_Y_VALUE). This function should be used only with the touchpad widgets. This function does not process the widget but extracts processed results from the Data Structure.

Parameters:

widgetId	Specifies the ID number of a touchpad widget to get the X/Y position of a detected touch. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Returns:

Returns the touch position of a specified touchpad widget:

1. If a touch is detected:
   - Bits [31..16] indicate the Y coordinate.

- Bits [15..0] indicate the X coordinate.

2. If no touch is detected or a wrong widgetId is specified:
   - CapSense_TOUCHPAD_NO_TOUCH.

Go to the top of the CapSense High-Level APIs section.

uint32 CapSense_RunTuner (void)

This function is used to establish synchronized communication between the CapSense Component and Tuner application (or other host controllers). This function is called periodically in the application program loop to serve the Tuner application (or host controller) requests and commands. In most cases, the best place to call this function is after processing and before next scanning.

If this function is absent in the application program, then communication is asynchronous and the following disadvantages are applicable:

- The raw counts displayed in the tuner may be filtered and/or unfiltered. As a result, noise and SNR measurements will not be accurate.
- The Tuner tool may read the sensor data such as raw counts from a scan multiple times, as a result, noise and SNR measurement will not be accurate.
- The Tuner tool and host controller should not change the Component parameters via the tuner interface. Changing the Component parameters via the tuner interface in the async mode will result in Component abnormal behavior.

Note that calling this function is not mandatory for the application, but required only to synchronize the communication with the host controller or tuner application.

Returns:

In some cases, the application program may need to know if the Component was re-initialized. The return indicates if a restart command was executed or not:

- CapSense_STATUS_RESTART_DONE - Based on a received command, the Component was restarted.
- CapSense_STATUS_RESTART_NONE - No restart was executed by this function.

Go to the top of the CapSense High-Level APIs section.

CapSense Low-Level APIs

Description

The low-level APIs represent the lower layer of abstraction in support of high-level APIs. These APIs also enable implementation of special case designs requiring performance optimization and non-typical functionalities.

The functions which contain CSD or CSX in the name are specified for that sensing method appropriately and should be used only with dedicated widgets having that mode. All other functions are general to all sensing methods, some of the APIs detect the sensing method used by the widget and executes tasks as appropriate.

Functions

- cystatus CapSense_ProcessWidgetExt(uint32 widgetId, uint32 mode)
  Performs customized data processing on the selected widget.
- cystatus CapSense_ProcessSensorExt(uint32 widgetId, uint32 sensorId, uint32 mode)
  Performs customized data processing on the selected widget's sensor.
- cystatus CapSense_UpdateAllBaselines(void)
  Updates the baseline for all the sensors in all the widgets.

- cystatus CapSense_UpdateWidgetBaseline(uint32 widgetId)
  Updates the baselines for all the sensors in a widget specified by the input parameter.
- cystatus CapSense_UpdateSensorBaseline(uint32 widgetId, uint32 sensorId)
  Updates the baseline for a sensor in a widget specified by the input parameters.
- void CapSense_InitializeAllBaselines(void)
  Initializes (or re-initializes) the baselines of all the sensors of all the widgets.
- void CapSense_InitializeWidgetBaseline(uint32 widgetId)
  Initializes (or re-initializes) the baselines of all the sensors in a widget specified by the input parameter.
- void CapSense_InitializeSensorBaseline(uint32 widgetId, uint32 sensorId)
  Initializes (or re-initializes) the baseline of a sensor in a widget specified by the input parameters.
- void CapSense_InitializeAllFilters(void)
  Initializes (or re-initializes) the raw count filter history of all the sensors of all the widgets.
- void CapSense_InitializeWidgetFilter(uint32 widgetId)
  Initializes (or re-initializes) the raw count filter history of all the sensors in a widget specified by the input parameter.
- uint32 CapSense_CheckGlobalCRC(void)
  Checks the stored CRC of the CapSense_RAM_STRUCT data structure.
- uint32 CapSense_CheckWidgetCRC(uint32 widgetId)
  Checks the stored CRC of the CapSense_RAM_WD_BASE_STRUCT data structure of the specified widget.
- uint32 CapSense_CheckBaselineDuplication(uint32 widgetId, uint32 sensorId)
  Checks that the baseline of the specified widget/sensor is not corrupted by comparing it with a baseline inverse copy.
- uint32 CapSense_CheckBaselineRawcountRange(uint32 widgetId, uint32 sensorId, CapSense_BSLN_RAW_RANGE_STRUCT*ranges)
  Checks that raw count and baseline of the specified widget/sensor are within the specified range.
- uint32 CapSense_CheckSensorShort(uint32 widgetId, uint32 sensorId)
  Checks the specified widget/sensor for shorts to GND or VDD.
- uint32 CapSense_CheckSns2SnsShort(uint32 widgetId, uint32 sensorId)
  Checks the specified widget/sensor for shorts to any other CapSense sensors.
- uint32 CapSense_GetSensorCapacitance(uint32 widgetId, uint32 sensorId)
  Measures the specified widget/sensor capacitance.
- uint32 CapSense_GetShieldCapacitance(void)
  Measures the shield electrode capacitance.
- uint32 CapSense_GetExtCapCapacitance(uint32 extCapId)
  Measures the capacitance of the specified external capacitor.
- uint16 CapSense_GetVdda(void)
  Measures and returns the VDDA voltage.
- void CapSense_SetPinState(uint32 widgetId, uint32 sensorElement, uint32 state)
  Sets the state (drive mode and output state) of the port pin used by a sensor. The possible states are GND, Shield, High-Z, Tx or Rx, Sensor. If the sensor specified in the input parameter is a ganged sensor, then the state of all pins associated with the ganged sensor is updated.
- cystatus CapSense_CalibrateWidget(uint32 widgetId)
  Calibrates the IDACs for all the sensors in the specified widget to the default target, this function detects the sensing method used by the widget prior to calibration.
- cystatus CapSense_CalibrateAllWidgets(void)

Calibrates the IDACs for all the widgets in the Component to the default target, this function detects the sensing method used by the widgets prior to calibration.

- void CapSense_CSDSetupWidget(uint32 widgetId)
  Performs hardware and firmware initialization required for scanning sensors in a specific widget using the CSD sensing method. This function requires using the CapSense_CSDScan() function to start scanning.
- void CapSense_CSDSetupWidgetExt(uint32 widgetId, uint32 sensorId)
  Performs extended initialization for the CSD widget and also performs initialization required for a specific sensor in the widget. This function requires using the CapSense_CSDScanExt() function to initiate a scan.
- void CapSense_CSDScan(void)
  This function initiates a scan for the sensors of the widget initialized by the CapSense_CSDSetupWidget() function.
- void CapSense_CSDScanExt(void)
  Starts the CSD conversion on the preconfigured sensor. This function requires using the CapSense_CSDSetupWidgetExt() function to set up the a widget.
- cystatus CapSense_CSDCalibrateWidget(uint32 widgetId, uint32 target)
  Executes the IDAC calibration for all the sensors in the widget specified in the input.
- void CapSense_CSDConnectSns (CapSense_FLASH_IO_STRUCTconst *snsAddrPtr)
  Connects a port pin used by the sensor to the AMUX bus of the sensing HW block.
- void CapSense_CSDDisconnectSns (CapSense_FLASH_IO_STRUCTconst *snsAddrPtr)
  Disconnects a sensor port pin from the sensing HW block and the AMUX bus. Sets the default state of the un-scanned sensor.
- void CapSense_CSXSetupWidget(uint32 widgetId)
  Performs hardware and firmware initialization required for scanning sensors in a specific widget using the CSX sensing method. This function requires using the CapSense_CSXScan() function to start scanning.
- void CapSense_CSXSetupWidgetExt(uint32 widgetId, uint32 sensorId)
  Performs extended initialization for the CSX widget and also performs initialization required for a specific sensor in the widget. This function requires using the CapSense_CSXScan() function to initiate a scan.
- void CapSense_CSXScan(void)
  This function initiates a scan for the sensors of the widget initialized by the CapSense_CSXSetupWidget() function.
- void CapSense_CSXScanExt(void)
  Starts the CSX conversion on the preconfigured sensor. This function requires using the CapSense_CSXSetupWidgetExt() function to set up a widget.
- void CapSense_CSXCalibrateWidget(uint32 widgetId, uint16 target)
  Calibrates the raw count values of all the sensors/nodes in a CSX widget.
- void CapSense_CSXConnectTx (CapSense_FLASH_IO_STRUCTconst *txPtr)
  Connects a Tx electrode to the CSX scanning hardware.
- void CapSense_CSXConnectRx (CapSense_FLASH_IO_STRUCTconst *rxPtr)
  Connects an Rx electrode to the CSX scanning hardware.
- void CapSense_CSXDisconnectTx (CapSense_FLASH_IO_STRUCTconst *txPtr)
  Disconnects a Tx electrode from the CSX scanning hardware.
- void CapSense_CSXDisconnectRx (CapSense_FLASH_IO_STRUCTconst *rxPtr)
  Disconnects an Rx electrode from the CSX scanning hardware.
- cystatus CapSense_GetParam(uint32 paramId, uint32 *value)
  Gets the specified parameter value from the Data Structure.
- cystatus CapSense_SetParam(uint32 paramId, uint32 value)

Sets a new value for the specified parameter in the Data Structure.

Function Documentation

cystatus CapSense_ProcessWidgetExt (uint32 *widgetId*, uint32 *mode*)

This function performs data processes for the specified widget specified by the mode parameter. The execution order of the requested operations is from LSB to MSB of the mode parameter. For a different order, this API can be called multiple times with the required mode parameter.

This function can be used with any of the available scan functions. This function is called only after all the sensors in the specified widget are scanned. Calling this function multiple times with the same mode without sensor scanning causes unexpected behavior. This function ignores the value of the wdgtEnable register. The CapSense_PROCESS_CALC_NOISE and CapSense_PROCESS_THRESHOLDS flags are supported by the CSD sensing method only when Auto-tuning mode is enabled. The pipeline scan method (i.e. during scanning of a widget, processing of a previously scanned widget is performed) can be implemented using this function and it may reduce the total scan/process time, increase the refresh rate and decrease the power consumption.

If the Ballistic multiplier filter is enabled the Timestamp must be updated before calling this function using the CapSense_IncrementGestureTimestamp() function.

If the Self-test library is enabled, this function executes the baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
mode	Specifies the type of widget processing to be executed for the specified widget: 1. Bits [31..6] - Reserved. 2. Bits [5..0] - CapSense_PROCESS_ALL - Execute all the tasks. 3. Bit [5] - CapSense_PROCESS_STATUS - Update the status (on/off, centroid position). 4. Bit [4] - CapSense_PROCESS_THRESHOLDS - Update the thresholds (only in CSD auto-tuning mode). 5. Bit [3] - CapSense_PROCESS_CALC_NOISE - Calculate the noise (only in CSD auto-tuning mode). 1. Bit [2] - CapSense_PROCESS_DIFFCOUNTS - Update the difference counts. 6. Bit [1] - CapSense_PROCESS_BASELINE - Update the baselines. 7. Bit [0] - CapSense_PROCESS_FILTER - Run the firmware filters.

Returns:

Returns the status of the widget processing operation:

- CYRET_SUCCESS - The processing is successfully performed.
- CYRET_BAD_PARAM - The input parameter is invalid.
- CYRET_BAD_DATA - The processing is failed.

Go to the top of the CapSense Low-Level APIs section.

cystatus CapSense_ProcessSensorExt (uint32 *widgetId*, uint32 *sensorId*, uint32 *mode*)

This function performs data processes for the specified sensor specified by the mode parameter. The execution order of the requested operations is from LSB to MSB of the mode parameter. For a different order, this function can be called multiple times with the required mode parameter.

This function can be used with any of the available scan functions. This function is called only after a specified sensor in the widget is scanned. Calling this function multiple times with the same mode without sensor scanning causes unexpected behavior. This function ignores the value of the wdgtEnable register.

The CapSense_PROCESS_CALC_NOISE and CapSense_PROCESS_THRESHOLDS flags are supported by the CSD sensing method only when Auto-tuning mode is enabled.

The pipeline scan method (i.e. during scanning of a sensor, processing of a previously scanned sensor is performed) can be implemented using this function and it may reduce the total scan/process time, increase the refresh rate and decrease the power consumption.

If the Self-test library is enabled, this function executes the baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Parameters:

widgetId	Specifies the ID number of the widget to process one of its sensors. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget to process it. A macro for the sensor ID within a specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.
mode	Specifies the type of the sensor processing that needs to be executed for the specified sensor: 1. Bits [31..5] - Reserved. 2. Bits [4..0] - CapSense_PROCESS_ALL - Executes all the tasks. 3. Bit [4] - CapSense_PROCESS_THRESHOLDS - Updates the thresholds (only in auto-tuning mode). 4. Bit [3] - CapSense_PROCESS_CALC_NOISE - Calculates the noise (only in auto-tuning mode). 1. Bit [2] - CapSense_PROCESS_DIFFCOUNTS - Updates the difference count. 5. Bit [1] - CapSense_PROCESS_BASELINE - Updates the baseline. 6. Bit [0] - CapSense_PROCESS_FILTER - Runs the firmware filters.

Returns:

Returns the status of the sensor process operation:

- CYRET_SUCCESS - The processing is successfully performed.
- CYRET_BAD_PARAM - The input parameter is invalid.
- CYRET_BAD_DATA - The processing is failed.

Go to the top of the CapSense Low-Level APIs section.

cystatus CapSense_UpdateAllBaselines (void)

Updates the baseline for all the sensors in all the widgets. Baseline updating is a part of data processing performed by the process functions. So, no need to call this function except a specific process flow is implemented.

This function ignores the value of the wdgtEnable register. Multiple calling of this function (or any other function with a baseline updating task) without scanning leads to unexpected behavior.

If the Self-test library is enabled, this function executes the baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Returns:

Returns the status of the update baseline operation of all the widgets:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_DATA - The baseline processing failed.

Go to the top of the CapSense Low-Level APIs section.

cystatus CapSense_UpdateWidgetBaseline (uint32 *widgetId*)

This function performs exactly the same tasks as CapSense_UpdateAllBaselines() but only for a specified widget.

This function ignores the value of the wdgtEnable register. Multiple calling of this function (or any other function with a baseline updating task) without scanning leads to unexpected behavior.

If the Self-test library is enabled, this function executes the baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Parameters:

widgetId	Specifies the ID number of the widget to update the baseline of all the sensors in the widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Returns:

Returns the status of the specified widget update baseline operation:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_DATA - The baseline processing is failed.

Go to the top of the CapSense Low-Level APIs section.

cystatus CapSense_UpdateSensorBaseline (uint32 *widgetId*, uint32 *sensorId*)

This function performs exactly the same tasks as CapSense_UpdateAllBaselines() and CapSense_UpdateWidgetBaseline() but only for a specified sensor.

This function ignores the value of the wdgtEnable register. Multiple calling of this function (or any other function with a baseline updating task) without scanning leads to unexpected behavior.

If the Self-test library is enabled, this function executes the baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Parameters:

widgetId	Specifies the ID number of the widget to update the baseline of the sensor specified by the sensorId argument. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget to update its baseline. A macro for the sensor ID within a specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.

Returns:

Returns the status of the specified sensor update baseline operation:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_DATA - The baseline processing failed.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_InitializeAllBaselines (void)

Initializes the baseline for all the sensors of all the widgets. Also, this function can be used to re-initialize baselines. CapSense_Start() calls this API as part of CapSense operation initialization.

If any raw count filter is enabled, make sure the raw count filter history is initialized as well using one of these functions:

- CapSense_InitializeAllFilters().
- CapSense_InitializeWidgetFilter().

Go to the top of the CapSense Low-Level APIs section.

void CapSense_InitializeWidgetBaseline (uint32 *widgetId*)

Initializes (or re-initializes) the baseline for all the sensors of the specified widget.

If any raw count filter is enabled, make sure the raw count filter history is initialized as well using one of these functions:

- CapSense_InitializeAllFilters().
- CapSense_InitializeWidgetFilter().

Parameters:

widgetId	Specifies the ID number of a widget to initialize the baseline of all the sensors in the widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_InitializeSensorBaseline (uint32 *widgetId*, uint32 *sensorId*)

Initializes (or re-initializes) the baseline for a specified sensor within a specified widget.

Parameters:

widgetId	Specifies the ID number of a widget to initialize the baseline of the sensor in the widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget to initialize its baseline. A macro for the sensor ID within a specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_InitializeAllFilters (void)

Initializes the raw count filter history for all the sensors of all the widgets. Also, this function can be used to re-initialize baselines. CapSense_Start() calls this API as part of CapSense operation initialization.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_InitializeWidgetFilter (uint32 *widgetId*)

Initializes (or re-initializes) the raw count filter history of all the sensors in a widget specified by the input parameter.

Parameters:

widgetId	Specifies the ID number of a widget to initialize the filter history of all the sensors in the widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Go to the top of the CapSense Low-Level APIs section.

uint32 CapSense_CheckGlobalCRC (void)

This function validates the data integrity of the CapSense_RAM_STRUCT data structure by calculating the CRC and comparing it with the stored CRC value (i.e. CapSense_GLB_CRC_VALUE).

If the stored and calculated CRC values differ, the calculated CRC is stored to the CapSense_GLB_CRC_CALC_VALUE register and the CapSense_TST_GLOBAL_CRC bit is set in the CapSense_TEST_RESULT_MASK_VALUE register. The function never clears the CapSense_TST_GLOBAL_CRC bit.

It is recommended to use the CapSense_SetParam() function to change a value of CapSense_RAM_STRUCT data structure register/elements as CRC is updated by the CapSense_SetParam() function.

This test also can be initiated by using CapSense_RunSelfTest() function with the CapSense_TST_GLOBAL_CRC mask input.

Returns:

Returns a status of the executed test:

- CY_RET_SUCCESS - The stored CRC matches the calculated CRC
- CapSense_TST_GLOBAL_CRC - The stored CRC is wrong.

Go to the top of the CapSense Low-Level APIs section.

uint32 CapSense_CheckWidgetCRC (uint32 *widgetId*)

This function validates the data integrity of the CapSense_RAM_WD_BASE_STRUCT data structure of the specified widget by calculating the CRC and comparing it with the stored CRC value (i.e. CapSense_<WidgetName>_CRC_VALUE).

If the stored and calculated CRC values differ:

1. The calculated CRC is stored to the CapSense_WDGT_CRC_CALC_VALUE register
2. The widget ID is stored to the CapSense_WDGT_CRC_ID_VALUE register
3. The CapSense_TST_WDGT_CRC bit is set in the CapSense_TEST_RESULT_MASK_VALUE register.

The function never clears the CapSense_TST_WDGT_CRC bit. If the CapSense_TST_WDGT_CRC bit is set, the CapSense_WDGT_CRC_CALC_VALUE and CapSense_WDGT_CRC_ID_VALUE registers are not updated.

It is recommended to use the CapSense_SetParam() function to change a value of CapSense_RAM_WD_BASE_STRUCT data structure register/elements as the CRC is updated by CapSense_SetParam() function.

This test can be initiated by CapSense_RunSelfTest() function with the CapSense_TST_WDGT_CRC mask as an input.

The function updates the wdgtWorking register CapSense_WDGT_WORKING<Number>_VALUE by clearing the widget-corresponding bit. Those non-working widgets are skipped by the high-level API. Restoring a widget to its working state should be done by the application level.

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Returns:

Returns a status of the test execution:

- CY_RET_SUCCESS - The stored CRC matches the calculated CRC.
- CapSense_TST_WDGT_CRC - The widget CRC is wrong.
- CapSense_TST_BAD_PARAM - The input parameter is invalid.

Go to the top of the CapSense Low-Level APIs section.

uint32 CapSense_CheckBaselineDuplication (uint32 *widgetId*, uint32 *sensorId*)

This function validates the integrity of baseline of sensor by comparing the conformity of the baseline and its inversion.

If the baseline does not match its inverse copy:

1. The widget ID is stored to the CapSense_INV_BSLN_WDGT_ID_VALUE register

2. The sensor ID is stored to the CapSense_INV_BSLN_SNS_ID_VALUE register
3. The CapSense_TST_BSLN_DUPLICATION bit is set in the CapSense_TEST_RESULT_MASK_VALUE register.

The function never clears the CapSense_TST_BSLN_DUPLICATION bit. If the CapSense_TST_BSLN_DUPLICATION bit is set, the CapSense_INV_BSLN_WDGT_ID_VALUE and CapSense_INV_BSLN_SNS_ID_VALUE registers are not updated.

It is possible to execute a test for all the widgets using CapSense_RunSelfTest() function with the CapSense_TST_BSLN_DUPLICATION mask. In this case, the CapSense_INV_BSLN_WDGT_ID_VALUE and CapSense_INV_BSLN_SNS_ID_VALUE registers contain the widget and sensor ID of the first detected fail.

The function updates the wdgtWorking register CapSense_WDGT_WORKING<Number>_VALUE by clearing the widget-corresponding bit. Those non-working widgets are skipped by the high-level API. Restoring a widget to its working state should be done by the application level.

The test is integrated into the CapSense Component. All CapSense processing functions like CapSense_ProcessAllWidgets() or CapSense_UpdateSensorBaseline() automatically verify the baseline value before using it and update its inverse copy after processing. If fail is detected during a baseline update a CYRET_BAD_DATA result is returned. The baseline initialization functions do not verify the baseline and update the baseline inverse copy.

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget. A macro for the sensor ID within the specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.

Returns:

Returns the status of the test execution:

- CY_RET_SUCCESS - The baseline matches its inverse copy.
- CapSense_TST_BSLN_DUPLICATION - The test failed.
- CapSense_TST_BAD_PARAM - The input parameters are invalid.

Go to the top of the CapSense Low-Level APIs section.

uint32 CapSense_CheckBaselineRawcountRange (uint32 *widgetId*, uint32 *sensorId*, CapSense_BSLN_RAW_RANGE_STRUCT* *ranges*)

The baseline and raw count shall be within specific range (based on calibration target) for good units. The function checks whether or not the baseline and raw count are within the limits defined by the user in the ranges function argument. If baseline or raw count are out of limits this function sets the CapSense_TST_BSLN_RAW_OUT_RANGE bit in the CapSense_TEST_RESULT_MASK_VALUE register.

Unlike other tests, this test does not update CapSense_WDGT_WORKING<Number>_VALUE register and is not available in the CapSense_RunSelfTest() function.

Use this function to verify the uniformity of sensors, for example, at mass-production or during an operation phase.

Parameters:

widgetId	Specifies the ID number of the widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget. A macro for the sensor ID within the specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.

**ranges*	Specifies the pointer to the CapSense_BSLN_RAW_RANGE_STRUCT structure with valid ranges for the raw count and baseline.

Returns:

Returns a status of the test execution:

- CY_RET_SUCCESS - The raw count and baseline are within the specified range
- CapSense_TST_BSLN_RAW_OUT_RANGE - The test failed and baseline or raw count or both are out of the specified limit.
- CapSense_TST_BAD_PARAM - The input parameters are invalid.

Go to the top of the CapSense Low-Level APIs section.

uint32 CapSense_CheckSensorShort (uint32 *widgetId*, uint32 *sensorId*)

The function performs test to check for a short to GND or VDD on the specified sensor. The resistance of electrical short must be less than 1100 Ohm including series resistors on sensor for the short to be detected. The CapSense_GetSensorCapacitance() function can be used to check an electrical short with resistance higher than 1100 Ohm or when the specified ganged sensor consists of two or more electrodes.

This function performs the following tasks:

- If a short is detected, the widget ID is stored to the CapSense_SHORTED_WDGT_ID_VALUE register.
- If a short is detected, the sensor ID is stored to the CapSense_SHORTED_SNS_ID_VALUE register.
- If a short is detected, the CapSense_TST_SNS_SHORT bit is set in the CapSense_TEST_RESULT_MASK_VALUE register.
- If a short is detected, the bit corresponding to the specified widget (CapSense_WDGT_WORKING<Number>_VALUE) is cleared in the wdgtWorking register to indicate fault with the widget. Once the bit is cleared, the widget is treated as non-working widget by the high-level functions and further processing are skipped. Restoring the bit corresponding the widget can be done by application layer to restore the operation of the high-level functions.
- If CapSense_TST_SNS_SHORT is already set due to previously detected fault on any of the sensor, the CapSense_TST_SNS_SHORT register is not cleared by this function and CapSense_SHORTED_WDGT_ID_VALUE and CapSense_SHORTED_SNS_ID_VALUE registers are not updated. For this reason, remember to read details of defective sensor and clear CapSense_TST_SNS_SHORT prior to calling this function on the same or different sensor.

This function performs the test on one specific sensor. The CapSense_RunSelfTest() function with the CapSense_TST_SNS_SHORT mask, performs the short test on all the widgets and sensors in the Component. In this case, CapSense_SHORTED_WDGT_ID_VALUE and CapSense_SHORTED_SNS_ID_VALUE registers stores the widget and sensor ID of the first faulty sensor.

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget. A macro for the sensor ID within the specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.

Returns:

Returns a status of the test execution:

- CY_RET_SUCCESS - The sensor of the widget does not have a short to VDD or GND and is in working condition.
- CapSense_TST_SNS_SHORT - A short is detected on the specified sensor.
- CapSense_TST_BAD_PARAM - The input parameters are invalid.

Go to the top of the CapSense Low-Level APIs section.

uint32 CapSense_CheckSns2SnsShort (uint32 *widgetId*, uint32 *sensorId*)

The function performs the test on the specified sensor to check for a short to other sensors in the Component. The resistance of an electrical short must be less than 1100 Ohm including the series resistors on the sensor for the short to be detected. The CapSense_GetSensorCapacitance() function can be used to check the electrical short with the resistance higher than 1100 Ohm or when the specified ganged sensor consists of two or more electrodes.

This function performs the following tasks:

- If a short is detected, widget ID is stored to the CapSense_P2P_WDGT_ID_VALUE register.
- If a short is detected, sensor ID is stored to the CapSense_P2P_SNS_ID_VALUE register.
- If a short is detected, CapSense_TST_SNS2SNS_SHORT bit is set in the CapSense_TEST_RESULT_MASK_VALUE register.
- If a short is detected, the bit corresponding to widget (CapSense_WDGT_WORKING<Number>_VALUE) is cleared in the wdgtWorking register to indicate fault with widget. Once the bit is cleared, the widget is treated as a non-working widget by the high-level functions and further processing are skipped. Restoring the bit corresponding to the widget can be done by application layer to restore the operation of high-level functions.
- If CapSense_TST_SNS2SNS_SHORT is already set due to the previously detected fault on any of the sensor, the CapSense_TST_SNS2SNS_SHORT register is not cleared by this function and CapSense_P2P_WDGT_ID_VALUE and CapSense_P2P_SNS_ID_VALUE registers are not updated. For this reason, remember to read details of defective sensor and clear CapSense_TST_SNS2SNS_SHORT prior to calling this function on the same or different sensor.

This function performs the test on one specific sensor. The CapSense_RunSelfTest() function with the CapSense_TST_SNS2SNS_SHORT mask performs the short test on all the widgets and sensors in the Component. In this case, CapSense_P2P_WDGT_ID_VALUE and CapSense_P2P_SNS_ID_VALUE registers store the widget and sensor ID of the first faulty sensor.

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget. A macro for the sensor ID within the specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.

Returns:

Returns a status of the test execution:

- CY_RET_SUCCESS - The sensor is not shorted to any other sensor is in working condition.
- CapSense_TST_SNS2SNS_SHORT - A short is detected with one or more sensors in the Component.
- CapSense_TST_BAD_PARAM - The input parameters are invalid.

Go to the top of the CapSense Low-Level APIs section.

uint32 CapSense_GetSensorCapacitance (uint32 *widgetId*, uint32 *sensorId*)

The function measures capacitance of the specified sensor and returns the result, alternatively the result is stored in the Component data structure.

For CSD sensors, the capacitance of the specified sensor is measured. For CSX sensors, the capacitance the both Rx and Tx electrodes of the sensor is measured. For ganged sensor, total capacitance of all electrodes associated with the sensor is measured. The capacitance measurement result is independent on the Component or sensor tuning parameters and neither tuning parameter nor the capacitance measurement function creates interference to other.

While measuring capacitance of a CSX sensor electrode, all the non-measured electrodes and CSD shield electrodes (if enabled) are set to active low (GND). While measuring capacitance of a CSD sensor electrode, all the CSX sensor electrodes are set to active low (GND) and all CSD sensor electrodes are set to the state defined by the inactive sensor state parameter in the Component CSD Setting tab of the customizer. If the shield electrode is enabled, it is enabled during CSD sensor capacitance measurement.

The measurable capacitance range using this function is from 5pF to 255pF. If a returned value is 255, the sensor capacitance can be higher.

The measured capacitance is stored in the CapSense_RAM_SNS_CP_STRUCT structure. The CapSense_<WidgetName>_PTR2SNS_CP_VALUE register contains a pointer to the array of the specified widget with the sensor capacitance.

This test can be executed for all the sensors at once using the CapSense_RunSelfTest() function along with the CapSense_TST_SNS_CAP mask.

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget. A macro for the sensor ID within the specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.

Returns:

Returns a result of the test execution:

- Bits [7..0] The capacitance (in pF) of the CSD electrode or the capacitance of Rx electrode of CSX sensor.
- Bits [15..8] The capacitance (in pF) of Tx electrode of CSX sensor.
- Bit [30] CapSense_TST_BAD_PARAM The input parameters are invalid.

Go to the top of the CapSense Low-Level APIs section.

uint32 CapSense_GetShieldCapacitance (void)

The function measures capacitance of the shield electrode and returns the result, alternatively the result is stored in CapSense_SHIELD_CAP_VALUE of data structure. If the shield consists of several electrodes, total capacitance of all shield electrodes is reported.

While measuring capacitance of shield electrode, the sensor states are inherited from the Component configuration. All the CSX electrodes are set to active low (GND) and all the CSD electrodes are set to state defined by the inactive sensor state parameter in the Component CSD Setting tab of the customizer.

The measurable capacitance range using this function is from 5pF to 255pF.

This test can be executed for all the sensors at once using the CapSense_RunSelfTest() function with the CapSense_TST_SH_CAP mask.

Returns:

The shield electrode capacitance (in pF)

Go to the top of the CapSense Low-Level APIs section.

uint32 CapSense_GetExtCapCapacitance (uint32 *extCapId*)

The function measures the capacitance of the specified external capacitor such as Cmod and returns the result, alternatively the result is stored in the CapSense_EXT_CAP<EXT_CAP_ID>_VALUE register in data structure.

The measurable capacitance range using this function is from 200pF to 60,000pF with measurement accuracy of 10%.

This test can be executed for all the external capacitors at once using the CapSense_RunSelfTest() function with the CapSense_TST_EXTERNAL_CAP mask.

Parameters:

extCapId	Specifies the ID number of the external capacitor to be measured:

	• CapSense_TST_CMOD_ID - Cmod capacitor • CapSense_TST_CSH_ID - Csh capacitor • CapSense_TST_CINTA_ID - CintA capacitor • CapSense_TST_CINTB_ID - CintB capacitor

Returns:

Returns a status of the test execution:

- The capacitance (in pF) of the specified external capacitor
- CapSense_TST_BAD_PARAM if the input parameter is invalid.

Go to the top of the CapSense Low-Level APIs section.

uint16 CapSense_GetVdda (void)

This function measures voltage on VDDA terminal of the chip and returns the result, alternatively the result is stores in the CapSense_VDDA_VOLTAGE_VALUE register of data structure.

Returns:

The VDDA voltage in mV.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_SetPinState (uint32 *widgetId*, uint32 *sensorElement*, uint32 *state*)

This function sets a specified state for a specified sensor element. For the CSD widgets, sensor element is a sensor ID, for the CSX widgets, it is either an Rx or Tx electrode ID. If the specified sensor is a ganged sensor, then the specified state is set for all the electrodes belong to the sensor. This function must not be called while the Component is in the busy state.

This function accepts the CapSense_SHIELD and CapSense_SENSOR states as an input only if there is at least one CSD widget. Similarly, this function accepts the CapSense_TX_PIN and CapSense_RX_PIN states as an input only if there is at least one CSX widget in the project.

Calling this function directly from the application layer is not recommended. This function is used to implement only the custom-specific use cases. Functions that perform a setup and scan of a sensor/widget automatically set the required pin states. They ignore changes in the design made by the CapSense_SetPinState() function. This function neither check wdgtIndex nor sensorElement for the correctness.

Parameters:

widgetId	Specifies the ID of the widget to change the pin state of the specified sensor. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorElement	Specifies the ID of the sensor element within the widget to change its pin state. For the CSD widgets, sensorElement is the sensor ID and can be found in the CapSense Configuration header file defined as • CapSense_<WidgetName>_SNS<SensorNumber>_ID. For the CSX widgets, sensorElement is defined either as Rx ID or Tx ID. The first Rx in a widget corresponds to sensorElement = 0, the second Rx in a widget corresponds to sensorElement = 1, and so on. The last Tx in a widget corresponds to sensorElement = (RxNum + TxNum). Macros for Rx and Tx IDs can be found in the CapSense Configuration header file defined as: • CapSense_<WidgetName>_RX<RXNumber>_ID • CapSense_<WidgetName>_TX<TXNumber>_ID.
state	Specifies the state of the sensor to be set: 1. CapSense_GROUND - The pin is connected to the ground.

	2. CapSense_HIGHZ - The drive mode of the pin is set to High-Z Analog. 3. CapSense_SHIELD - The shield signal is routed to the pin (available only if CSD sensing method with shield electrode is enabled). 4. CapSense_SENSOR - The pin is connected to the scanning bus (available only if CSD sensing method is enabled). 5. CapSense_TX_PIN - The Tx signal is routed to the sensor (available only if CSX sensing method is enabled). 6. CapSense_RX_PIN - The pin is connected to the scanning bus (available only if CSX sensing method is enabled).

Go to the top of the CapSense Low-Level APIs section.

cystatus CapSense_CalibrateWidget (uint32 *widgetId*)

This function performs exactly the same tasks as CapSense_CalibrateAllWidgets, but only for a specified widget. This function detects the sensing method used by the widgets and uses the Enable compensation IDAC parameter.

This function is available when the CSD and/or CSX Enable IDAC auto-calibration parameter is enabled.

Parameters:

widgetId	Specifies the ID number of the widget to calibrate its raw count. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Returns:

Returns the status of the specified widget calibration:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_PARAM - The input parameter is invalid.
- CYRET_BAD_DATA - The calibration failed and the Component may not operate as expected.

Go to the top of the CapSense Low-Level APIs section.

cystatus CapSense_CalibrateAllWidgets (void)

Calibrates the IDACs for all the widgets in the Component to the default target value. This function detects the sensing method used by the widgets and regards the Enable compensation IDAC parameter.

This function is available when the CSD and/or CSX Enable IDAC auto-calibration parameter is enabled.

Returns:

Returns the status of the calibration process:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_DATA - The calibration failed and the Component may not operate as expected.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSDSetupWidget (uint32 *widgetId*)

This function initializes the specific widget common parameters to perform the CSD scanning. The initialization includes setting up a Modulator and Sense clock frequency and scanning resolution.

This function does not connect any specific sensors to the scanning hardware, neither does it start a scanning process. The CapSense_CSDScan() API must be called after initializing the widget to start scanning.

This function is called when no scanning is in progress. I.e. CapSense_IsBusy() returns a non-busy status.

This function is called by the CapSense_SetupWidget() API if the given widget uses the CSD sensing method.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time or pipeline scanning for example).

Parameters:

widgetId	Specifies the ID number of the widget to perform hardware and firmware initialization required for scanning sensors in the specific widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSDSetupWidgetExt (uint32 *widgetId*, uint32 *sensorId*)

This function does the same as CapSense_CSDSetupWidget() and also does the following tasks:

1. Connects the first sensor of the widget.
2. Configures the IDAC value.
3. Initializes an interrupt callback function to initialize a scan of the next sensors in a widget.

Once this function is called to initialize a widget and a sensor, the CapSense_CSDScanExt() function is called to scan the sensor.

This function is called when no scanning is in progress. I.e. CapSense_IsBusy() returns a non-busy status.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time or pipeline scanning for example).

Parameters:

widgetId	Specifies the ID number of the widget to perform hardware and firmware initialization required for scanning the specific sensor in the specific widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget to perform hardware and firmware initialization required for scanning a specific sensor in a specific widget. A macro for the sensor ID within a specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSDScan (void)

This function performs scanning of all the sensors in the widget configured by the CapSense_CSDSetupWidget() function. It does the following tasks:

1. Connects the first sensor of the widget.
2. Configures the IDAC value.
3. Initializes the interrupt callback function to initialize a scan of the next sensors in a widget.
4. Starts scanning for the first sensor in the widget.

This function is called by the CapSense_Scan() API if the given widget uses the CSD sensing method.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time or pipeline scanning for example).

This function is called when no scanning is in progress. I.e. CapSense_IsBusy() returns a non-busy status. The widget must be preconfigured by the CapSense_CSDSetupWidget() function if any other widget was previously scanned or any other type of the scan functions was used.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSDScanExt (void)

This function performs single scanning of one sensor in the widget configured by the CapSense_CSDSetupWidgetExt() function. It does the following tasks:

1. Sets the busy flag in the CapSense_dsRam structure.

2. Performs the clock-phase alignment of the sense and modulator clocks.
3. Performs the Cmod pre-charging.
4. Starts single scanning.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time or pipeline scanning for example). This function is called when no scanning is in progress. I.e. CapSense_IsBusy() returns a non-busy status.

The sensor must be preconfigured by using the CapSense_CSDSetupWidgetExt() API prior to calling this function. The sensor remains ready for a next scan if a previous scan was triggered by using the CapSense_CSDScanExt() function. In this case, calling CapSense_CSDSetupWidgetExt() is not required every time before the CapSense_CSDScanExt() function. If a previous scan was triggered in any other way - CapSense_Scan(), CapSense_ScanAllWidgets() or CapSense_RunTuner() - (see the CapSense_RunTuner() function description for more details), the sensor must be preconfigured again by using the CapSense_CSDSetupWidgetExt() API prior to calling the CapSense_CSDScanExt() function.

If disconnection of the sensors is required after calling CapSense_CSDScanExt(), the CapSense_CSDDisconnectSns() function can be used.

Go to the top of the CapSense Low-Level APIs section.

cystatus CapSense_CSDCalibrateWidget (uint32 *widgetId*, uint32 *target*)

Performs a successive approximation search algorithm to find appropriate IDAC values for sensors in the specified widget that provides the raw count to the level specified by the target parameter.

Calibration is always performed in the single IDAC mode and if the dual IDAC mode (Enable compensation IDAC is enabled) is configured, the IDAC values are re-calculated to match the raw count target. If a widget consists of two or more elements (buttons, slider segments, etc.), then calibration is performed by the element with the highest sensor capacitance.

Calibration fails if the achieved raw count is outside of the +/-10% range of the target.

This function is available when the CSD Enable IDAC auto-calibration parameter is enabled or the SmartSense auto-tuning mode is configured.

Parameters:

widgetId	Specifies the ID number of the CSD widget to calibrate its raw count. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
target	Specifies the calibration target in percentages of the maximum raw count.

Returns:

Returns the status of the specified widget calibration:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_PARAM - The input parameter is invalid.
- CYRET_BAD_DATA - The calibration failed and the Component may not operate as expected.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSDConnectSns (CapSense_FLASH_IO_STRUCTconst * *snsAddrPtr*)

Connects a port pin used by the sensor to the AMUX bus of the sensing HW block while a sensor is being scanned. The function ignores the fact if the sensor is a ganged sensor and connects only a specified pin.

Scanning should be completed before calling this API.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases. Functions that perform a setup and scan of a sensor/widget, automatically set the required pin states and perform the sensor connection. They do not take into account changes in the design made by the CapSense_CSDConnectSns() function.

Parameters:

snsAddrPtr	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a sensor which to be connected to the sensing HW block.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSDDisconnectSns (CapSense_FLASH_IO_STRUCTconst * *snsAddrPtr*)

This function works identically to CapSense_CSDConnectSns() except it disconnects the specified port-pin used by the sensor.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases. Functions that perform a setup and scan of sensor/widget automatically set the required pin states and perform the sensor connection. They ignore changes in the design made by the CapSense_CSDDisconnectSns() function.

Parameters:

snsAddrPtr	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a sensor which should be disconnected from the sensing HW block.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSXSetupWidget (uint32 *widgetId*)

This function initializes the widgets specific common parameters to perform the CSX scanning. The initialization includes the following:

1. The CSD_CONFIG register.
2. The IDAC register.
3. The Sense clock frequency
4. The phase alignment of the sense and modulator clocks.

This function does not connect any specific sensors to the scanning hardware and neither does it start a scanning process. The CapSense_CSXScan() function must be called after initializing the widget to start scanning.

This function is called when no scanning is in progress. I.e. CapSense_IsBusy() returns a non-busy status.

This function is called by the CapSense_SetupWidget() API if the given widget uses the CSX sensing method.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time or pipeline scanning for example).

Parameters:

widgetId	Specifies the ID number of the widget to perform hardware and firmware initialization required for scanning sensors in the specific widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSXSetupWidgetExt (uint32 *widgetId*, uint32 *sensorId*)

This function does the same tasks as CapSense_CSXSetupWidget() and also connects a sensor in the widget for scanning. Once this function is called to initialize a widget and a sensor, the CapSense_CSXScanExt() function must be called to scan the sensor.

This function is called when no scanning is in progress. I.e. CapSense_IsBusy() returns a non-busy status.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time or pipeline scanning for example).

Parameters:

widgetId	Specifies the ID number of the widget to perform hardware and firmware initialization required for scanning a specific sensor in a

	specific widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget to perform hardware and firmware initialization required for scanning a specific sensor in a specific widget. A macro for the sensor ID within a specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSXScan (void)

This function performs scanning of all the sensors in the widget configured by the CapSense_CSXSetupWidget() function. It does the following tasks:

1. Connects the first sensor of the widget.
2. Initializes an interrupt callback function to initialize a scan of the next sensors in a widget.
3. Starts scanning for the first sensor in the widget.

This function is called by the CapSense_Scan() API if the given widget uses the CSX sensing method.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time or pipeline scanning for example).

This function is called when no scanning is in progress. I.e. CapSense_IsBusy() returns a non-busy status. The widget must be preconfigured by the CapSense_CSXSetupWidget() function if any other widget was previously scanned or any other type of scan functions were used.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSXScanExt (void)

This function performs single scanning of one sensor in the widget configured by the CapSense_CSXSetupWidgetExt() function. It does the following tasks:

1. Sets a busy flag in the CapSense_dsRam structure.
2. Configures the Tx clock frequency.
3. Configures the Modulator clock frequency.
4. Configures the IDAC value.
5. Starts single scanning.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time or pipeline scanning for example). This function is called when no scanning is in progress. I.e. CapSense_IsBusy() returns a non-busy status.

The sensor must be preconfigured by using the CapSense_CSXSetupWidgetExt() API prior to calling this function. The sensor remains ready for the next scan if a previous scan was triggered by using the CapSense_CSXScanExt() function. In this case, calling CapSense_CSXSetupWidgetExt() is not required every time before the CapSense_CSXScanExt() function. If a previous scan was triggered in any other way - CapSense_Scan(), CapSense_ScanAllWidgets() or CapSense_RunTuner() - (see the CapSense_RunTuner() function description for more details), the sensor must be preconfigured again by using the CapSense_CSXSetupWidgetExt() API prior to calling the CapSense_CSXScanExt() function.

If disconnection of the sensors is required after calling CapSense_CSXScanExt(), the CapSense_CSXDisconnectTx() and CapSense_CSXDisconnectRx() APIs can be used.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSXCalibrateWidget (uint32 *widgetId*, uint16 *target*)

Performs a successive approximation search algorithm to find appropriate IDAC values for sensors in the specified widget that provides a raw count to the level specified by the target parameter.

This function is available when the CSX Enable IDAC auto-calibration parameter is enabled.

Parameters:

widgetId	Specifies the ID number of the CSX widget to calibrate its raw count. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
target	Specifies the calibration target in percentages of the maximum raw count.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSXConnectTx (CapSense_FLASH_IO_STRUCTconst * *txPtr*)

This function connects a port pin (Tx electrode) to the CSD_SENSE signal. It is assumed that drive mode of the port pin is already set to STRONG in the HSIOM_PORT_SELx register.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time when there is only one port pin for an electrode for example).

Parameters:

txPtr	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a sensor to be connected to the sensing HW block as a Tx pin.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSXConnectRx (CapSense_FLASH_IO_STRUCTconst * *rxPtr*)

This function connects a port pin (Rx electrode) to AMUXBUS-A and sets drive mode of the port pin to High-Z in the GPIO_PRT_PCx register.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time when there is only one port pin for an electrode for example).

Parameters:

rxPtr	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a sensor to be connected to the sensing HW block as an Rx pin.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSXDisconnectTx (CapSense_FLASH_IO_STRUCTconst * *txPtr*)

This function disconnects a port pin (Tx electrode) from the CSD_SENSE signal and configures the port pin to the strong drive mode. It is assumed that the data register (GPIO_PRTx_DR) of the port pin is already 0.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time when there is only one port pin for an electrode for example).

Parameters:

txPtr	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a Tx pin sensor to be disconnected from the sensing HW block.

Go to the top of the CapSense Low-Level APIs section.

void CapSense_CSXDisconnectRx (CapSense_FLASH_IO_STRUCTconst * *rxPtr*)

This function disconnects a port pin (Rx electrode) from AMUXBUS_A and configures the port pin to the strong drive mode. It is assumed that the data register (GPIO_PRTx_DR) of the port pin is already 0.

Calling this function directly from the application layer is not recommended. This function is used to implement only the user's specific use cases (for faster execution time when there is only one port pin for an electrode for example).

Parameters:

rxPtr	Specifies the pointer to the FLASH_IO_STRUCT object belonging to an

	Rx pin sensor to be disconnected from the sensing HW block.

Go to the top of the CapSense Low-Level APIs section.

cystatus CapSense_GetParam (uint32 *paramId*, uint32 * *value*)

This function gets the value of the specified parameter by the paramId argument. The paramId for each register is available in the CapSense RegisterMap header file as CapSense_<ParameterName>_PARAM_ID. The paramId is a special enumerated value generated by the customizer. The format of paramId is as follows:

1. [byte 3 byte 2 byte 1 byte 0]
2. [TTWFCCCC UIIIIIII MMMMMMMM LLLLLLLL]
3. T - encodes the parameter type:
   - 01b: uint8
   - 10b: uint16
   - 11b: uint32
4. W - indicates whether the parameter is writable:
   - 0: ReadOnly
   - 1: Read/Write
5. C - 4 bit CRC (X^3 + 1) of the whole paramId word, the C bits are filled with 0s when the CRC is calculated.
6. U - indicates if the parameter affects the RAM Widget Object CRC.
7. I - specifies that the widgetId parameter belongs to
8. M,L - the parameter offset MSB and LSB accordingly in:
   - Flash Data Structure if W bit is 0.
   - RAM Data Structure if W bit is 1.

Refer to the Data Structure section for details of the data structure organization and examples of its register access.

Parameters:

paramId	Specifies the ID of parameter to get its value. A macro for the parameter ID can be found in the CapSense RegisterMap header file defined as CapSense_<ParameterName>_PARAM_ID.
value	The pointer to a variable to be updated with the obtained value.

Returns:

Returns the status of the operation:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_PARAM - The input parameter is invalid.

Go to the top of the CapSense Low-Level APIs section.

cystatus CapSense_SetParam (uint32 *paramId*, uint32 *value*)

This function sets the value of the specified parameter by the paramId argument. The paramId for each register is available in the CapSense RegisterMap header file as CapSense_<ParameterName>_PARAM_ID. The paramId is a special enumerated value generated by the customizer. The format of paramId is as follows:

1. [byte 3 byte 2 byte 1 byte 0]
2. [TTWFCCCC UIIIIIII MMMMMMMM LLLLLLLL]
3. T - encodes the parameter type:
   - 01b: uint8
   - 10b: uint16
   - 11b: uint32
4. W - indicates whether the parameter is writable:
   - 0: ReadOnly
   - 1: Read/Write
5. C - 4 bit CRC (X^3 + 1) of the whole paramId word, the C bits are filled with 0s when the CRC is calculated.
6. U - indicates if the parameter affects the RAM Widget Object CRC.

7. I - specifies that the widgetId parameter belongs to
8. M,L - the parameter offset MSB and LSB accordingly in:
   - Flash Data Structure if W bit is 0.
   - RAM Data Structure if W bit is 1.

Refer to the Data Structure section for details of the data structure organization and examples of its register access.

This function writes specified value into the desired register without other registers update. It is application layer responsibility to keep all the data structure registers aligned. Repeated call of CapSense_Start() function helps aligning dependent register values.

Parameters:

paramId	Specifies the ID of parameter to set its value. A macro for the parameter ID can be found in the CapSense RegisterMap header file defined as CapSense_<ParameterName>_PARAM_ID.
value	Specifies the new parameter's value.

Returns:

Returns the status of the operation:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_PARAM - The input parameter is invalid.

Go to the top of the CapSense Low-Level APIs section.

Interrupt Service Routine

Description

The CapSense component uses an interrupt that triggers after the end of each sensor scan.

After scanning is complete, the ISR copies the measured sensor raw data to the Data Structure. If the scanning queue is not empty, the ISR starts the next sensor scanning.

The Component implementation avoids using critical sections in the code. In an unavoidable situation, the critical section is used and the code is optimized for the shortest execution time.

The CapSense component does not alter or affect the priority of other interrupts in the system.

These API should not be used in the application layer.

Functions

- CY_ISR(CapSense_CSDPostSingleScan)
  This is an internal ISR function for the single-sensor scanning implementation.
- CY_ISR(CapSense_CSDPostMultiScan)
  This is an internal ISR function for the multiple-sensor scanning implementation.
- CY_ISR(CapSense_CSDPostMultiScanGanged)
  This is an internal ISR function for the multiple-sensor scanning implementation for ganged sensors.
- CY_ISR(CapSense_CSXScanISR)
  This is an internal ISR function to handle the CSX sensing method operation.

Function Documentation

CY_ISR (CapSense_CSDPostSingleScan)

This ISR handler is triggered when the user calls the CapSense_CSDScanExt() function.

The following tasks are performed for Third-generation HW block:

1. Disable the CSD interrupt.
2. Read the Counter register and update the data structure with raw data.
3. Connect the Vref buffer to the AMUX bus.
4. Update the Scan Counter.
5. Reset the BUSY flag.
6. Enable the CSD interrupt.

The following tasks are performed for Fourth-generation HW block:

1. Check if the raw data is not noisy.
2. Read the Counter register and update the data structure with raw data.
3. Configure and start the scan for the next frequency if the multi-frequency is enabled.
4. Update the Scan Counter.
5. Reset the BUSY flag.
6. Enable the CSD interrupt.

The ISR handler changes the IMO and initializes scanning for the next frequency channels when multi-frequency scanning is enabled.

This function has two Macro Callbacks that allow calling the user code from macros specified in Component's generated code. Refer to the Macro Callbacks section of the PSoC Creator User Guide for details.

Go to the top of the Interrupt Service Routine section.

CY_ISR (CapSense_CSDPostMultiScan)

This ISR handler is triggered when the user calls the CapSense_Scan() or CapSense_ScanAllWidgets() APIs.

The following tasks are performed:

1. Disable the CSD interrupt.
2. Read the Counter register and update the data structure with raw data.
3. Connect the Vref buffer to the AMUX bus.
4. Disable the CSD block (after the widget has been scanned).
5. Update the Scan Counter.
6. Reset the BUSY flag.
7. Enable the CSD interrupt.

The ISR handler initializes scanning for the previous sensor when the widget has more than one sensor. The ISR handler initializes scanning for the next widget when the CapSense_ScanAllWidgets() APIs are called and the project has more than one widget. The ISR handler changes the IMO and initializes scanning for the next frequency channels when multi-frequency scanning is enabled.

This function has two Macro Callbacks that allow calling the user code from macros specified in Component's generated code. Refer to the Macro Callbacks section of the PSoC Creator User Guide for details.

Go to the top of the Interrupt Service Routine section.

CY_ISR (CapSense_CSDPostMultiScanGanged)

This ISR handler is triggered when the user calls the CapSense_Scan() API for a ganged sensor or the CapSense_ScanAllWidgets() API in the project with ganged sensors.

The following tasks are performed:

1. Disable the CSD interrupt.
2. Read the Counter register and update the data structure with raw data.
3. Connect the Vref buffer to the AMUX bus.
4. Disable the CSD block (after the widget has been scanned).
5. Update the Scan Counter.
6. Reset the BUSY flag.
7. Enable the CSD interrupt.

The ISR handler initializes scanning for the previous sensor when the widget has more than one sensor. The ISR handler initializes scanning for the next widget when the CapSense_ScanAllWidgets() APIs are called and the project has more than one widget. The ISR handler changes the IMO and initializes scanning for the next frequency channels when multi-frequency scanning is enabled.

This function has two Macro Callbacks that allow calling the user code from macros specified in Component's generated code. Refer to the Macro Callbacks section of the PSoC Creator User Guide for details.

Go to the top of the Interrupt Service Routine section.

CY_ISR (CapSense_CSXScanISR)

This handler covers the following functionality:

- Read the result of the measurement and store it into the corresponding register of the data structure.
- If the Noise Metric functionality is enabled, then check the number of bad conversions and repeat the scan of the current sensor of the number of bad conversions is greater than the Noise Metric Threshold.
- Initiate the scan of the next sensor for multiple sensor scanning mode.
- Update the Status register in the data structure.
- Switch the HW block to the default state if scanning of all the sensors is completed.

Go to the top of the Interrupt Service Routine section.

Macro Callbacks

Macro callbacks allow the user to execute the code from the API files automatically generated by PSoC Creator. Refer to the PSoC Creator Help and Component Author Guide for more details.

In order to add the code to the macro callback present in the component's generated source files, perform the following:

- Define a macro to signal the presence of a callback (in cyapicallbacks.h). This will "uncomment" the function call from the component's source code.
- Write the function declaration (in cyapicallbacks.h) using the name provided in the table. This will make this function visible to all the project files.
- Write the function implementation (in any user file).

CapSense Macro Callbacks

Macro Callback Function Name	Associated Macro	Description
CapSense_EntryCallback	CapSense_ENTRY_CALLBACK	Used at the beginning of the CapSense interrupt handler to perform additional application-specific actions
CapSense_ExitCallback	CapSense_EXIT_CALLBACK	Used at the end of the CapSense interrupt handler to perform additional application-specific actions
CapSense_StartSampleCallback(uint8 CapSense_widgetId, uint8 CapSense_sensorId)	CapSense_START_SAMPLE_CALLBACK	Used before each sensor scan triggering and deliver the current widget / sensor Id

Global Variables

Description

The section documents the CapSense component related global Variables.

The CapSense component stores the component configuration and scanning data in the data structure. Refer to the Data Structure section for details of organization of the data structure.

Variables

- CapSense_RAM_STRUCT CapSense_dsRam

Variable Documentation

CapSense_RAM_STRUCTCapSense_dsRam

The variable that contains the CapSense configuration, settings and scanning results. CapSense_dsRam represents RAM Data Structure.

API Constants

Description

The section documents the CapSense component related API Constants.

Variables

- const CapSense_FLASH_STRUCT CapSense_dsFlash
- const CapSense_FLASH_IO_STRUCT CapSense_ioList[CapSense_TOTAL_ELECTRODES]
- const CapSense_SHIELD_IO_STRUCT CapSense_shieldIoList[CapSense_CSD_TOTAL_SHIELD_COUNT]

Variable Documentation

const CapSense_FLASH_STRUCTCapSense_dsFlash

Constant for the FLASH Data Structure

const CapSense_FLASH_IO_STRUCTCapSense_ioList[CapSense_TOTAL_ELECTRODES]

The array of the pointers to the electrode specific register.

const CapSense_SHIELD_IO_STRUCTCapSense_shieldIoList[CapSense_CSD_TOTAL_SHIELD_COUNT]

The array of the pointers to the shield electrode specific register.

Data Structure

Description

This section provides the list of structures/registers available in the component.

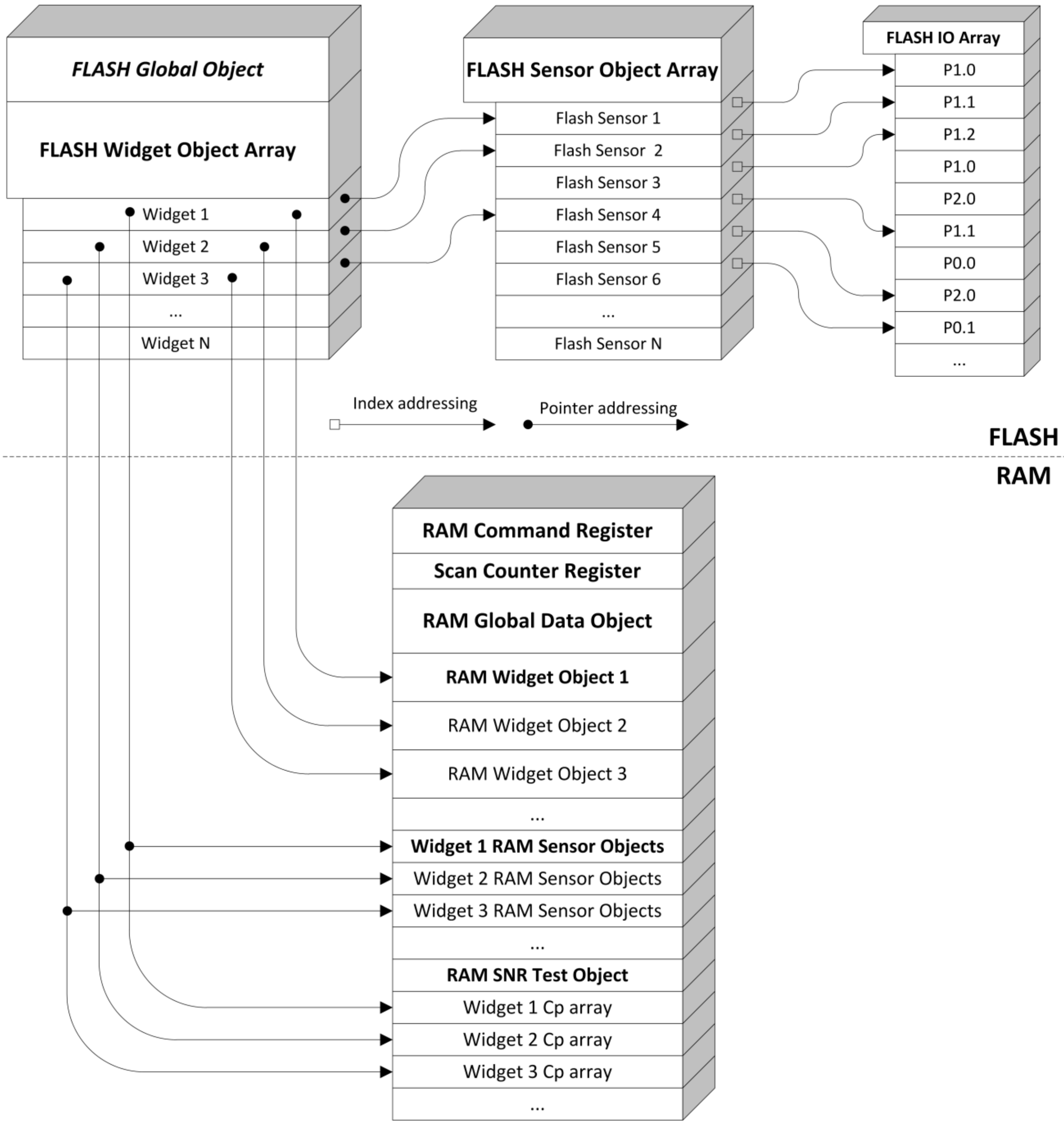
The key responsibilities of Data Structure are as follows:

- The Data Structure is the only data container in the component.
- It serves as storage for the configuration and the output data.

- All other component FW part as well as an application layer and Tuner SW use the data structure for the communication and data exchange.

The CapSense Data Structure organizes configuration parameters, input and output data shared among different FW IP modules within the component. It also organizes input and output data presented at the Tuner interface (the tuner register map) into a globally accessible data structure. CapSense Data Structure is only a data container.

The Data Structure is a composite of several smaller structures (for global data, widget data, sensor data, and pin data). Furthermore, the data is split between RAM and Flash to achieve a reasonable balance between resources consumption and configuration / tuning flexibility at runtime and compile time. A graphical representation of CapSense Data Structure is shown below:



Note that figure above shows a sample representation and documents the high-level design of the data structure, it may not include all the parameters and elements in each object.

CapSense Data Structure does not perform error checking on the data written to CapSense Data Structure. It is the responsibility of application layer to ensure register map rule are not violated while modifying the value of data field in CapSense Data Structure.

The CapSense Data Structure parameter fields and their offset address is specific to an application, and it is based on component configuration used for the project. A user readable representation of the Data Structure specific to the component configuration is the component register map. The Register map file available from the Customizer GUI and it describes offsets and data/bit fields for each static (Flash) and dynamic (RAM) parameters of the component.

The embedded CapSense_RegisterMap header file list all registers of data structure with the following:

```
#define CapSense_<RegisterName>_VALUE       (<Direct Register Access Macro>)
#define CapSense_<RegisterName>_OFFSET      (<Register Offset Within Data Structure (RAM or Flash)>)
#define CapSense_<RegisterName>_SIZE        (<Register Size in Bytes>)
#define CapSense_<RegisterName>_PARAM_ID    (<ParamId for Getter/Setter functions>)
```

To access CapSense Data Structure registers you have the following options:

1. Direct Access

The access to registers is performed through the Data Structure variable CapSense_dsRam and constants CapSense_dsFlash from application program.

Example of access to the Raw Count register of third sensor of Button0 widget:

```
rawCount = CapSense_dsRam.snsList.button0[CapSense_BUTTON0_SNS2_ID].raw[0];
```

Corresponding macro to access register value is defined in the CapSense_RegisterMap header file:

```
rawCount = CapSense_BUTTON0_SNS2_RAW0_VALUE;
```

2. Getter/Setter Access

The access to registers from application program is performed by using two functions:

```
cystatus CapSense_GetParam(uint32 paramId, uint32 *value)
cystatus CapSense_SetParam(uint32 paramId, uint32 value)
```

The value of paramId argument for each register can be found in CapSense_RegisterMap header file.

Example of access to the Raw Count register of third sensor of Button0 widget:

```
CapSense_GetParam(CapSense_BUTTON0_SNS2_RAW0_PARAM_ID, &rawCount);
```

You can also write to a register if it is writable (writing new finger threshold value to Button0 widget):

```
CapSense_SetParam(CapSense_BUTTON0_FINGER_TH_PARAM_ID, fingerThreshold);
```

3. Offset Access

The access to registers is performed by host through the I2C communication by reading / writing registers based on their offset.

Example of access to the Raw Count register of third sensor of Button0 widget: Setting up communication data buffer to CapSense data structure to be exposed to I2C master at primary slave address request once at initialization an application program:

```
EZI2C_Start();
EZI2C_EzI2CSetBuffer1(sizeof(CapSense_dsRam), sizeof(CapSense_dsRam),
                      (uint8 *)&CapSense_dsRam);
```

Now host can read (write) the whole CapSense Data Structure and get the specified register value by register offset macro available in CapSense_RegisterMap header file:

```
rawCount = *(uint16 *)(I2C_buffer1Ptr + CapSense_BUTTON0_SNS2_RAW0_OFFSET);
```

The current example is applicable to 2-byte registers only. Depends on register size defined CapSense_RegisterMap header file by corresponding macros (CapSense_BUTTON0_SNS2_RAW0_SIZE) specific logic should be added to read 4-byte, 2-byte and 1-byte registers.

Data Structures

- struct ADAPTIVE_FILTER_CONFIG_STRUCT
  Declares Adaptive Filter configuration parameters.
- struct ADVANCED_CENTROID_POSITION_STRUCT
  Declares Advanced Centroid position structure.
- struct ADVANCED_CENTROID_TOUCH_STRUCT
  Declares Advanced Centroid touch structure.
- struct SMARTSENSE_CSD_NOISE_ENVELOPE_STRUCT
  Declares Noise envelope data structure for CSD widgets when SmartSense is enabled.
- struct CapSense_RAM_WD_BASE_STRUCT
  Declares common widget RAM parameters.
- struct CapSense_RAM_WD_BUTTON_STRUCT
  Declares RAM parameters for the CSD Button.
- struct CapSense_RAM_WD_SLIDER_STRUCT
  Declares RAM parameters for the Slider.
- struct CapSense_RAM_WD_CSD_MATRIX_STRUCT
  Declares RAM parameters for the CSD Matrix Buttons.
- struct CapSense_RAM_WD_CSD_TOUCHPAD_STRUCT
  Declares RAM parameters for the CSD Touchpad.
- struct CapSense_RAM_WD_PROXIMITY_STRUCT
  Declares RAM parameters for the CSD Proximity.
- struct CapSense_RAM_WD_CSX_MATRIX_STRUCT
  Declares RAM parameters for the CSX Matrix Buttons.
- struct CapSense_RAM_WD_LIST_STRUCT
  Declares RAM structure with all defined widgets.
- struct CapSense_RAM_SNS_STRUCT
  Declares RAM structure for sensors.
- struct CapSense_RAM_SNS_LIST_STRUCT
  Declares RAM structure with all defined sensors.
- struct CapSense_RAM_STRUCT
  Declares the top-level RAM Data Structure.
- struct CapSense_FLASH_IO_STRUCT
  Declares the Flash IO object.
- struct CapSense_FLASH_SNS_STRUCT
  Declares the Flash Electrode object.
- struct CapSense_FLASH_SNS_LIST_STRUCT
  Declares the structure with all Flash electrode objects.
- struct CapSense_FLASH_WD_STRUCT
  Declares Flash widget object.
- struct CapSense_FLASH_STRUCT
  Declares top-level Flash Data Structure.
- struct CapSense_SHIELD_IO_STRUCT
  Declares the Flash IO structure for Shield electrodes.
- struct CapSense_BSLN_RAW_RANGE_STRUCT

Defines the structure for test of baseline and raw count limits which will be determined by user for every sensor grounding on the manufacturing specific data.

- struct CapSense_TMG_CONFIG_STRUCT
  Gesture configuration structure.
- struct CapSense_TMG_BALLISTIC_MULT
  Ballistic multiplier configuration structure.

Data Structure Documentation

struct ADAPTIVE_FILTER_CONFIG_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint8	maxK	Maximum filter coefficient
uint8	minK	Minimum filter coefficient
uint8	noMovTh	No-movement threshold
uint8	littleMovTh	Little movement threshold
uint8	largeMovTh	Large movement threshold
uint8	divVal	Divisor value

struct ADVANCED_CENTROID_POSITION_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	x	X position
uint16	y	Y position
uint16	zX	Z value of X axis
uint16	zY	Z value of Y axis

struct ADVANCED_CENTROID_TOUCH_STRUCT

Go to the top of the Data Structures section.

Data Fields:

ADVANCED_CENTROID_POSITION_STRUCT	pos[ADVANCED_CENTROID_MAX_TOUCHES]	Array of position structure
uint8	touchNum	Number of touches

struct SMARTSENSE_CSD_NOISE_ENVELOPE_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	param0	Parameter 0 configuration
uint16	param1	Parameter 1 configuration
uint16	param2	Parameter 2 configuration
uint16	param3	Parameter 3 configuration
uint16	param4	Parameter 4 configuration
uint8	param5	Parameter 5 configuration
uint8	param6	Parameter 6 configuration

struct CapSense_RAM_WD_BASE_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
CapSense_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
CapSense_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widgets. For the CSD Touchpad and Matrix Button widgets, sets the current of the modulation IDAC for the column sensors.
uint8	rowIdacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the row sensors for the CSD Touchpad and Matrix Button widgets. Not used for the CSX widgets.
uint16	snsClk	Specifies the sense clock divider. Present only if individual clock dividers are enabled. Specifies the sense clock divider for the Column sensors for the Matrix Buttons and Touchpad widgets. Sets Tx clock divider for CSX Widgets.
uint16	rowSnsClk	For the Matrix Buttons and Touchpad widgets specifies the sense clock divider for the row sensors. Present only if individual clock dividers are enabled.
uint8	snsClkSource	Register for internal use
uint8	rowSnsClkSource	Register for internal use
uint16	fingerCap	Widget Finger capacitance parameter. Available only if the SmartSense is enabled. Not used for the CSX Widgets.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
uint8	gestureId	Keeps either current gesture detection status or detected gesture code.
uint8	scrollCnt	The scroll count of the last detected scroll gesture.
int16	posXDelta	The filtered by Ballistic Multiplier X-displacement between current and previous touch.
int16	posYDelta	The filtered by Ballistic Multiplier Y-displacement between current and previous touch.

struct CapSense_RAM_WD_BUTTON_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
CapSense_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
CapSense_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widgets. For the CSD Touchpad and Matrix Button widgets, sets the current of the modulation IDAC for the column sensors.
uint8	rowIdacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the row sensors for the CSD Touchpad and Matrix Button widgets. Not used for the CSX widgets.
uint16	snsClk	Specifies the sense clock divider. Present only if individual clock dividers are enabled. Specifies the sense clock divider for the Column sensors for the Matrix Buttons and Touchpad widgets. Sets Tx clock divider for CSX Widgets.
uint16	rowSnsClk	For the Matrix Buttons and Touchpad widgets specifies the sense clock divider for the row sensors. Present only if individual clock dividers are enabled.
uint8	snsClkSource	Register for internal use
uint8	rowSnsClkSource	Register for internal use
uint16	fingerCap	Widget Finger capacitance parameter. Available only if the SmartSense is enabled. Not used for the CSX Widgets.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
uint8	gestureId	Keeps either current gesture detection status or detected gesture code.
uint8	scrollCnt	The scroll count of the last detected scroll gesture.
int16	posXDelta	The filtered by Ballistic Multiplier X-displacement between current and previous touch.
int16	posYDelta	The filtered by Ballistic Multiplier Y-displacement between current and previous touch.

struct CapSense_RAM_WD_SLIDER_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
CapSense_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
CapSense_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widgets. For the CSD Touchpad and Matrix Button widgets, sets the current of the modulation IDAC for the column sensors.
uint8	rowIdacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the row sensors for the CSD Touchpad and Matrix Button widgets. Not used for the CSX widgets.
uint16	snsClk	Specifies the sense clock divider. Present only if individual clock dividers are enabled. Specifies the sense clock divider for the Column sensors for the Matrix Buttons and Touchpad widgets. Sets Tx clock divider for CSX Widgets.
uint16	rowSnsClk	For the Matrix Buttons and Touchpad widgets specifies the sense clock divider for the row sensors. Present only if individual clock dividers are enabled.
uint8	snsClkSource	Register for internal use
uint8	rowSnsClkSource	Register for internal use
uint16	fingerCap	Widget Finger capacitance parameter. Available only if the SmartSense is enabled. Not used for the CSX Widgets.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
uint8	gestureId	Keeps either current gesture detection status or detected gesture code.
uint8	scrollCnt	The scroll count of the last detected scroll gesture.
uint16	position[CapSense_NUM_CENTROIDS]	Reports the widget position.
int16	posXDelta	The filtered by Ballistic Multiplier X-displacement between current and previous touch.
int16	posYDelta	The filtered by Ballistic Multiplier Y-

		displacement between current and previous touch.
ADAPTIVE_FILTER_CONFIG_STRUCT	aiirConfig	Keeps the configuration of position adaptive filter.

struct CapSense_RAM_WD_CSD_MATRIX_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
CapSense_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
CapSense_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widgets. For the CSD Touchpad and Matrix Button widgets, sets the current of the modulation IDAC for the column sensors.
uint8	rowIdacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the row sensors for the CSD Touchpad and Matrix Button widgets. Not used for the CSX widgets.
uint16	snsClk	Specifies the sense clock divider. Present only if individual clock dividers are enabled. Specifies the sense clock divider for the Column sensors for the Matrix Buttons and Touchpad widgets. Sets Tx clock divider for CSX Widgets.
uint16	rowSnsClk	For the Matrix Buttons and Touchpad widgets specifies the sense clock divider for the row sensors. Present only if individual clock dividers are enabled.
uint8	snsClkSource	Register for internal use
uint8	rowSnsClkSource	Register for internal use
uint16	fingerCap	Widget Finger capacitance parameter. Available only if the SmartSense is enabled. Not used for the CSX Widgets.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
uint8	gestureId	Keeps either current gesture detection status or detected gesture code.
uint8	scrollCnt	The scroll count of the last detected scroll

		gesture.
uint8	posCol	The active column sensor. From 0 to ColNumber - 1.
uint8	posRow	The active row sensor. From 0 to RowNumber - 1.
uint8	posSnsId	The active button ID. From 0 to RowNumber*ColNumber - 1.
int16	posXDelta	The filtered by Ballistic Multiplier X-displacement between current and previous touch.
int16	posYDelta	The filtered by Ballistic Multiplier Y-displacement between current and previous touch.

struct CapSense_RAM_WD_CSD_TOUCHPAD_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
CapSense_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
CapSense_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widgets. For the CSD Touchpad and Matrix Button widgets, sets the current of the modulation IDAC for the column sensors.
uint8	rowIdacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the row sensors for the CSD Touchpad and Matrix Button widgets. Not used for the CSX widgets.
uint16	snsClk	Specifies the sense clock divider. Present only if individual clock dividers are enabled. Specifies the sense clock divider for the Column sensors for the Matrix Buttons and Touchpad widgets. Sets Tx clock divider for CSX Widgets.
uint16	rowSnsClk	For the Matrix Buttons and Touchpad widgets specifies the sense clock divider for the row sensors. Present only if individual clock dividers are enabled.
uint8	snsClkSource	Register for internal use
uint8	rowSnsClkSource	Register for internal use
uint16	fingerCap	Widget Finger capacitance parameter.

		Available only if the SmartSense is enabled. Not used for the CSX Widgets.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
uint8	gestureId	Keeps either current gesture detection status or detected gesture code.
uint8	scrollCnt	The scroll count of the last detected scroll gesture.
uint16	posX	The X coordinate.
uint16	posY	The Y coordinate.
ADVANCED_CENTROID_TOUCH_STRUCT	position	The touch information about detected fingers.
int16	posXDelta	The filtered by Ballistic Multiplier X-displacement between current and previous touch.
int16	posYDelta	The filtered by Ballistic Multiplier Y-displacement between current and previous touch.
uint16	edgeVirtualSensorTh	The virtual sensor parameter that defines its signal calculation.
uint16	edgePenultimateTh	The threshold for determining when virtual sensor signal is calculated.
uint8	crossCouplingPosTh	The sensors cross coupling threshold
ADAPTIVE_FILTER_CONFIG_STRUCT	aiirConfig	Keeps the configuration of position adaptive filter.

struct CapSense_RAM_WD_PROXIMITY_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
CapSense_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
CapSense_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widgets. For the CSD Touchpad and Matrix Button widgets, sets the current of the modulation IDAC for the column sensors.

uint8	rowIdacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the row sensors for the CSD Touchpad and Matrix Button widgets. Not used for the CSX widgets.
uint16	snsClk	Specifies the sense clock divider. Present only if individual clock dividers are enabled. Specifies the sense clock divider for the Column sensors for the Matrix Buttons and Touchpad widgets. Sets Tx clock divider for CSX Widgets.
uint16	rowSnsClk	For the Matrix Buttons and Touchpad widgets specifies the sense clock divider for the row sensors. Present only if individual clock dividers are enabled.
uint8	snsClkSource	Register for internal use
uint8	rowSnsClkSource	Register for internal use
uint16	fingerCap	Widget Finger capacitance parameter. Available only if the SmartSense is enabled. Not used for the CSX Widgets.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
uint8	gestureId	Keeps either current gesture detection status or detected gesture code.
uint8	scrollCnt	The scroll count of the last detected scroll gesture.
int16	posXDelta	The filtered by Ballistic Multiplier X-displacement between current and previous touch.
int16	posYDelta	The filtered by Ballistic Multiplier Y-displacement between current and previous touch.
CapSense_THRESHOLD_TYPE	proxTouchTh	The proximity touch threshold.

struct CapSense_RAM_WD_CSX_MATRIX_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
CapSense_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
CapSense_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.

uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widgets. For the CSD Touchpad and Matrix Button widgets, sets the current of the modulation IDAC for the column sensors.
uint8	rowIdacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the row sensors for the CSD Touchpad and Matrix Button widgets. Not used for the CSX widgets.
uint16	snsClk	Specifies the sense clock divider. Present only if individual clock dividers are enabled. Specifies the sense clock divider for the Column sensors for the Matrix Buttons and Touchpad widgets. Sets Tx clock divider for CSX Widgets.
uint16	rowSnsClk	For the Matrix Buttons and Touchpad widgets specifies the sense clock divider for the row sensors. Present only if individual clock dividers are enabled.
uint8	snsClkSource	Register for internal use
uint8	rowSnsClkSource	Register for internal use
uint16	fingerCap	Widget Finger capacitance parameter. Available only if the SmartSense is enabled. Not used for the CSX Widgets.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
uint8	gestureId	Keeps either current gesture detection status or detected gesture code.
uint8	scrollCnt	The scroll count of the last detected scroll gesture.
int16	posXDelta	The filtered by Ballistic Multiplier X-displacement between current and previous touch.
int16	posYDelta	The filtered by Ballistic Multiplier Y-displacement between current and previous touch.

struct CapSense_RAM_WD_LIST_STRUCT

Go to the top of the Data Structures section.

Data Fields:

CapSense_RAM_WD_BUTTON_STRUCT	button0	Button0 widget RAM structure
CapSense_RAM_WD_SLIDER_STRUCT	linearslider0	LinearSlider0 widget RAM structure
CapSense_RAM_WD_SLIDER_STRUCT	radialslider0	RadialSlider0 widget RAM structure
CapSense_RAM_WD_C	matrixbuttons0	MatrixButtons0 widget RAM structure

SD_MATRIX_STRUCT		
CapSense_RAM_WD_CSD_TOUCHPAD_STRUCT	touchpad0	Touchpad0 widget RAM structure
CapSense_RAM_WD_PROXIMITY_STRUCT	proximity0	Proximity0 widget RAM structure
CapSense_RAM_WD_BUTTON_STRUCT	button1	Button1 widget RAM structure
CapSense_RAM_WD_CSX_MATRIX_STRUCT	matrixbuttons1	MatrixButtons1 widget RAM structure

struct CapSense_RAM_SNS_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	raw[CapSense_NUM_SCAN_FREQS]	The sensor raw counts.
uint16	bsln[CapSense_NUM_SCAN_FREQS]	The sensor baseline.
uint8	bslnExt[CapSense_NUM_SCAN_FREQS]	For the bucket baseline algorithm holds the bucket state, For the IIR baseline keeps LSB of the baseline value.
CapSense_THRESHOLD_TYPE	diff	Sensor differences.
CapSense_LOW_BSLN_RST_TYPE	negBslnRstCnt[CapSense_NUM_SCAN_FREQS]	The baseline reset counter for the low baseline reset function.
uint8	idacComp[CapSense_NUM_SCAN_FREQS]	The compensation IDAC value or the balancing IDAC value.

struct CapSense_RAM_SNS_LIST_STRUCT

Go to the top of the Data Structures section.

Data Fields:

CapSense_RAM_SNS_STRUCT	button0[CapSense_BUTTON0_NUM_SENSORS]	Button0 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	linearslider0[CapSense_LINEARSLIDER0_NUM_SENSORS]	LinearSlider0 sensors RAM structures array
CapSense_RAM_SNS_	radialslider0[CapSense_RADIALSLIDER0_	RadialSlider0 sensors RAM structures array

STRUCT	NUM_SENSORS]	
CapSense_RAM_SNS_STRUCT	matrixbuttons0[CapSense_MATRIXBUTTONS0_NUM_COLS+CapSense_MATRIXBUTTONS0_NUM_ROWS]	MatrixButtons0 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	touchpad0[CapSense_TOUCHPAD0_NUM_COLS+CapSense_TOUCHPAD0_NUM_ROWS]	Touchpad0 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	proximity0[CapSense_PROXIMITY0_NUM_SENSORS]	Proximity0 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	button1[CapSense_BUTTON1_NUM_SENSORS]	Button1 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	matrixbuttons1[(CapSense_MATRIXBUTTONS1_NUM_RX)*(CapSense_MATRIXBUTTONS1_NUM_TX)]	MatrixButtons1 sensors RAM structures array

struct CapSense_RAM_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	configId	16-bit CRC calculated by the customizer for the component configuration. Used by the Tuner application to identify if the FW corresponds to the specific user configuration.
uint16	deviceId	Used by the Tuner application to identify device-specific configuration.
uint16	hwClock	Used by the Tuner application to identify the system clock frequency.
uint16	tunerCmd	Tuner Command Register. Used for the communication between the Tuner GUI and the component.
uint16	scanCounter	This counter gets incremented after each scan.
volatile uint32	status	Status information: Current Widget, Scan active, Error code.
uint32	wdgtEnable[CapSense_WDGT_STATUS_WORDS]	The bitmask that sets which Widgets are enabled and scanned, each bit corresponds to one widget.
uint32	wdgtStatus[CapSense_WDGT_STATUS_WORDS]	The bitmask that reports activated Widgets (widgets that detect a touch signal above the threshold), each bit corresponds to one widget.
CapSense_SNS_STS_TYPE	snsStatus[CapSense_TOTAL_WIDGETS]	For Buttons, Sliders, Matrix Buttons and CSD Touchpad each bit reports status of the individual sensor of the widget: 1 - active (above the finger threshold); 0 - inactive; For

		the CSD Touchpad and CSD Matrix Buttons, the column sensors occupy the least significant bits. For the Proximity widget, each sensor uses two bits with the following meaning: 00 - Not active; 01 - Proximity detected (signal above finger threshold); 11 - A finger touch detected (signal above the touch threshold); For the CSX Touchpad Widget, this register provides a number of the detected touches. The array size is equal to the total number of widgets. The size of the array element depends on the max number of sensors per widget used in the current design. It could be 1, 2 or 4 bytes.
uint16	csd0Config	The configuration register for global parameters of the SENSE_HW0 block.
uint8	modCsdClk	The modulator clock divider for the CSD widgets.
uint8	modCsxClk	The modulator clock divider for the CSX widgets.
uint16	snsCsdClk	The global sense clock divider for the CSD widgets.
uint16	snsCsxClk	Global sense clock divider for the CSX widgets.
CapSense_RAM_WD_LIST_STRUCT	wdgtList	RAM Widget Objects.
CapSense_RAM_SNS_LIST_STRUCT	snsList	RAM Sensor Objects.
CapSense_TMG_CONFIG_STRUCT	gestures	The configuration data for gestures detection.
CapSense_TMG_BALLISTIC_MULT	ballisticConfig	The configuration data for position ballistic filter.
uint32	timestampInterval	The timestamp interval used at increasing the timestamp.
uint32	timestamp	The current timestamp.
uint8	snrTestWidgetId	The selected widget ID.
uint8	snrTestSensorId	The selected sensor ID.
uint16	snrTestScanCounter	The scan counter.
uint16	snrTestRawCount[CapSense_NUM_SCAN_FREQS]	The sensor raw counts.

struct CapSense_FLASH_IO_STRUCT

Go to the top of the Data Structures section.

Data Fields:

reg32 *	hsiomPtr	Pointer to the HSIOM configuration register of the IO.

reg32 *	pcPtr	Pointer to the port configuration register of the IO.
reg32 *	drPtr	Pointer to the port data register of the IO.
reg32 *	psPtr	Pointer to the pin state data register of the IO.
uint32	hsiomMask	IO mask in the HSIOM configuration register.
uint32	mask	IO mask in the DR and PS registers.
uint8	hsiomShift	Position of the IO configuration bits in the HSIOM register.
uint8	drShift	Position of the IO configuration bits in the DR and PS registers.
uint8	shift	Position of the IO configuration bits in the PC register.

struct CapSense_FLASH_SNS_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	firstPinId	Index of the first IO in the Flash IO Object Array.
uint8	numPins	Total number of IOs in this sensor.
uint8	type	Sensor type:

struct CapSense_FLASH_SNS_LIST_STRUCT

Go to the top of the Data Structures section.

Data Fields:

CapSense_FLASH_SNS_STRUCT	proximity0[CapSense_PROXIMITY0_NUM_SENSORS]	Proximity0 FLASH electrodes array

struct CapSense_FLASH_WD_STRUCT

Go to the top of the Data Structures section.

Data Fields:

void const *	ptr2SnsFlash	Points to the array of the FLASH Sensor Objects or FLASH IO Objects that belong to this widget. Sensing block uses this pointer to access and configure IOs for the scanning. Bit #2 in WD_STATIC_CONFIG field indicates the type of array: 1 - Sensor Object; 0 - IO Object.
void *	ptr2WdgtRam	Points to the Widget Object in RAM. Sensing block uses it to access scan parameters. Processing uses it to access threshold and widget specific data.
CapSense_RAM_SNS_STRUCT*	ptr2SnsRam	Points to the array of Sensor Objects in RAM. The sensing and processing blocks use it to access the scan data.
void *	ptr2FltrHistory	Points to the array of the Filter History Objects in RAM that belongs to this widget.
uint8 *	ptr2DebounceArr	Points to the array of the debounce counters. The size of the debounce counter is 8 bits. These arrays are not part of the data structure.
uint32	staticConfig	Miscellaneous configuration flags.

uint16	totalNumSns	The total number of sensors. For CSD widgets: WD_NUM_ROWS + WD_NUM_COLS. For CSX widgets: WD_NUM_ROWS * WD_NUM_COLS.
uint8	wdgtType	Specifies one of the following widget types: WD_BUTTON_E, WD_LINEAR_SLIDER_E, WD_RADIAL_SLIDER_E, WD_MATRIX_BUTTON_E, WD_TOUCHPAD_E, WD_PROXIMITY_E
uint8	senseMethod	Specifies the widget sensing method that could be either WD_CSD_SENSE_METHOD or WD_CSX_SENSE_METHOD
uint8	numCols	For CSD Button and Proximity Widgets, the number of sensors. For CSD Slider Widget, the number of segments. For CSD Touchpad and Matrix Button, the number of the column sensors. For CSX Button, Touchpad and Matrix Button, the number of the Rx electrodes.
uint8	numRows	For CSD Touchpad and Matrix Buttons, the number of the row sensors. For the CSX Button, the number of the Tx electrodes (constant 1u). For CSX Touchpad and Matrix Button, the number of the Tx electrodes.
uint16	xResolution	Sliders: The Linear/Angular resolution. Touchpad: The X-Axis resolution.
uint16	yResolution	Touchpad: The Y-Axis resolution.
uint32	xCentroidMultiplier	The pre-calculated X resolution centroid multiplier used for the X-axis position calculation. Calculated as follows: RADIAL: (WD_X_RESOLUTION * 256) / WD_NUM_COLS; LINEAR and TOUCHPAD: (WD_X_RESOLUTION * 256) / (WD_NUM_COLS - CONFIG); where CONFIG is 0 or 1 depends on CentroidMultiplerMethod parameter
uint32	yCentroidMultiplier	The pre-calculated Y resolution centroid multiplier used for the Y-axis position calculation. Calculated as follows: (WD_Y_RESOLUTION * 256) / (WD_NUM_ROWS - CONFIG); where CONFIG is 0 or 1 depends on CentroidMultiplerMethod parameter
SMARTSENSE_CSD_NOISE_ENVELOPE_STRUCT*	ptr2NoiseEnvlp	The pointer to the array with the sensor noise envelope data. Set to the valid value only for the CSD widgets. For the CSX widgets this pointer is set to NULL. The pointed array is not part of the data structure.
void *	ptr2PosHistory	The pointer to the RAM position history object. This parameter is used for the Sliders and CSD touchpads that have enabled the median position filter.
uint8	iirFilterCoeff	The position IIR filter coefficient.

struct CapSense_FLASH_STRUCT

Go to the top of the Data Structures section.

Data Fields:

CapSense_FLASH_WD_STRUCT	wdgtArray[CapSense_TOTAL_WIDGETS]	Array of flash widget objects
CapSense_FLASH_SNS_LIST_STRUCT	eltdList	Structure with all Ganged Flash electrode objects

struct CapSense_SHIELD_IO_STRUCT

Go to the top of the Data Structures section.

Data Fields:

reg32 *	hsiomPtr	The pointer to the HSIOM configuration register of the IO.
reg32 *	pcPtr	The pointer to the port configuration register of the IO.
reg32 *	drPtr	The pointer to the port data register of the IO.
uint32	hsiomMask	The IO mask in the HSIOM configuration register.
uint8	hsiomShift	The position of the IO configuration bits in the HSIOM register.
uint8	drShift	The position of the IO configuration bits in the DR and PS registers.
uint8	shift	The position of the IO configuration bits in the PC register.

struct CapSense_BSLN_RAW_RANGE_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	bslnHiLim	Upper limit of a sensor baseline.
uint16	bslnLoLim	Lower limit of a sensor baseline.
uint16	rawHiLim	Upper limit of a sensor raw count.
uint16	rawLoLim	Lower limit of a sensor raw count.

struct CapSense_TMG_CONFIG_STRUCT

Go to the top of the Data Structures section.

Data Fields:

volatile uint8_t	size	The size of the CapSense_TMG_CONFIG_STRUCT in bytes.
volatile uint8_t	panActiveDistanceX	Sets the minimum active step distance in the X dimension that has to be exceeded before a motion is considered active. The distance is measured in the resolution units. The range is 1 to 255.
volatile uint8_t	panActiveDistanceY	Sets the minimum active step distance in the Y dimension that has to be exceeded before a motion is considered active

volatile uint8_t	zoomActiveDistanceX	This parameter sets the minimum active step distance in the X dimension that has to be exceeded before a motion is considered an active Zoom (in or out)
volatile uint8_t	zoomActiveDistanceY	This parameter sets the minimum active step distance in the Y dimension that has to be exceeded before a motion is considered an active Zoom (in or out)
volatile uint8_t	flickActiveDistanceX	This parameter sets the minimum active step distance in the X dimension that has to be exceeded before a motion is considered Flick gesture
volatile uint8_t	flickActiveDistanceY	This parameter sets the minimum active step distance in the Y dimension that has to be exceeded before a motion is considered Flick gesture
volatile uint8_t	stScrollThreshold1X	This is a distance in the X-axis that finger(s) should pass between 2 consecutive scans to activate One-finger Scroll gesture
volatile uint8_t	stScrollThreshold2X	This is a distance in the X-axis that finger(s) should pass between 2 consecutive scans to activate 1-finger scroll gesture
volatile uint8_t	stScrollThreshold3X	This is a distance in the X-axis that finger(s) should pass between 2 consecutive scans to activate One-finger Scroll gesture
volatile uint8_t	stScrollThreshold4X	This is a distance in the X-axis that finger(s) should pass between 2 consecutive scans to activate One-finger Scroll gesture
volatile uint8_t	stScrollThreshold1Y	This is a distance in the Y-axis that finger(s) should pass between 2 consecutive scans to activate One-finger Scroll gesture
volatile uint8_t	stScrollThreshold2Y	This is a distance in the Y-axis that finger(s) should pass between 2 consecutive scans to activate One-finger Scroll gesture
volatile uint8_t	stScrollThreshold3Y	This is a distance in the Y-axis that finger(s) should pass between 2 consecutive scans to activate One-finger Scroll gesture
volatile uint8_t	stScrollThreshold4Y	This is a distance in the Y-axis that finger(s) should pass between 2 consecutive scans to activate One-finger Scroll gesture
volatile uint8_t	stScrollStep1	This is a number of scrolls that is reported if Scroll gesture is detected and the distance passed between 2 consecutive scans is: • (stScrollThreshold1X <= distance < stScrollThreshold2X) - for X-axis; • (stScrollThreshold1Y <= distance < stScrollThreshold2Y) - for Y-axis;
volatile uint8_t	stScrollStep2	This is a number of scrolls that is reported if Scroll gesture is detected and the distance passed between 2 consecutive scans is: • (stScrollThreshold2X <= distance <

		stScrollThreshold3X) - for X-axis; • (stScrollThreshold2Y <= distance < stScrollThreshold3Y) - for Y-axis;
volatile uint8_t	stScrollStep3	This is a number of scrolls that is reported if Scroll gesture is detected and the distance passed between 2 consecutive scans is: • (stScrollThreshold3X <= distance < stScrollThreshold4X) - for X-axis; • (stScrollThreshold3Y <= distance < stScrollThreshold4Y) - for Y-axis;
volatile uint8_t	stScrollStep4	This is a number of scrolls that is reported if Scroll gesture is detected and the distance passed between 2 consecutive scans is: • (stScrollThreshold4X <= distance) - for X-axis; • (stScrollThreshold4Y <= distance) - for Y-axis;
volatile uint8_t	stScrollDebounce	This parameter sets the number of similar, sequential One-finger Scroll gestures that should be performed before the One-finger Scroll gesture is considered valid. This parameter is for the One-finger Scroll gestures.
volatile uint8_t	dtScrollThreshold1X	This is a distance in the X-axis that finger(s) should pass between 2 consecutive scans to activate Two-finger Scroll gesture. The following number of scrolls will be reported in this case: dtScrollStep1.
volatile uint8_t	dtScrollThreshold2X	This is a distance in the X-axis that finger(s) should pass between 2 consecutive scans to activate Two-finger Scroll gesture. The following number of scrolls will be reported in this case: dtScrollStep2.
volatile uint8_t	dtScrollThreshold3X	This is a distance in the X-axis that finger(s) should pass between 2 consecutive scans to activate Two-finger Scroll gesture. The following number of scrolls will be reported in this case: dtScrollStep3.
volatile uint8_t	dtScrollThreshold4X	This is a distance in the X-axis that finger(s) should pass between 2 consecutive scans to activate Two-finger Scroll gesture. The following number of scrolls will be reported in this case: dtScrollStep4.
volatile uint8_t	dtScrollThreshold1Y	This is a distance in the Y-axis that finger(s) should pass between 2 consecutive scans to activate Two-finger Scroll gesture. The following number of scrolls will be reported in this case: dtScrollStep1.

volatile uint8_t	dtScrollThreshold2Y	This is a distance in the Y-axis that finger(s) should pass between 2 consecutive scans to activate Two-finger Scroll gesture. The following number of scrolls will be reported in this case: dtScrollStep2.
volatile uint8_t	dtScrollThreshold3Y	This is a distance in the Y-axis that finger(s) should pass between 2 consecutive scans to activate Two-finger Scroll gesture. The following number of scrolls will be reported in this case: dtScrollStep3.
volatile uint8_t	dtScrollThreshold4Y	This is a distance in the Y-axis that finger(s) should pass between 2 consecutive scans to activate Two-finger Scroll gesture. The following number of scrolls will be reported in this case: dtScrollStep4.
volatile uint8_t	dtScrollStep1	This is a number of scrolls that is reported if Scroll gesture is detected and the distance passed between 2 consecutive scans is: • (dtScrollThreshold1X <= distance < dtScrollThreshold2X) - for X-axis; • (dtScrollThreshold1Y <= distance < dtScrollThreshold2Y) - for Y-axis;
volatile uint8_t	dtScrollStep2	This is a number of scrolls that is reported if Scroll gesture is detected and the distance passed between 2 consecutive scans is: • (dtScrollThreshold2X <= distance < dtScrollThreshold3X) - for X-axis; • (dtScrollThreshold2Y <= distance < dtScrollThreshold3Y) - for Y-axis;
volatile uint8_t	dtScrollStep3	This is a number of scrolls that is reported if Scroll gesture is detected and the distance passed between 2 consecutive scans is: • (dtScrollThreshold3X <= distance < dtScrollThreshold4X) - for X-axis; • (dtScrollThreshold3Y <= distance < dtScrollThreshold4Y) - for Y-axis;
volatile uint8_t	dtScrollStep4	This is a number of scrolls that is reported if Scroll gesture is detected and the distance passed between 2 consecutive scans is: • (dtScrollThreshold4X <= distance) - for X-axis; • (dtScrollThreshold4Y <= distance) - for Y-axis;
volatile uint8_t	dtScrollDebounce	This parameter sets the number of similar, sequential Two-finger Scroll gestures that should be performed before the Two-finger

		Scroll gesture is considered valid. This parameter is for the Two-finger Scroll gestures.
volatile uint8_t	dtScrollToZoomDebounce	This parameter sets the number of Zoom gestures that will be ignored after a Two-finger Scroll gesture is observed. This is used to filter out Zoom gestures that inevitably occur during a transition from the Two-finger Scroll.
volatile uint8_t	stInScrActiveDistanceX	This parameter sets the number of pixels in X direction that has to be exceeded before a Lift Off event to trigger the Two-finger Inertial Scroll. A high value indicates that a bigger distance should be passed to activate a Two-finger Inertial Scroll gesture.
volatile uint8_t	stInScrActiveDistanceY	This parameter sets the number of pixels in Y direction that has to be exceeded before a Lift Off event to trigger the Two-finger Inertial Scroll. A high value indicates that a bigger distance should be passed to activate a Two-finger Inertial Scroll gesture.
volatile uint8_t	stInScrCountLevel	This use can select Low or High levels of the One-finger Inertial count. The decayCount decays through a 64-byte array or a 32-byte array. A low Inertial Scroll count level selects a 32-byte array and sends a few Inertial scrolls. High = 1. Low = 0.
volatile uint8_t	dtInScrActiveDistanceX	This parameter sets the number of pixels in X direction that has to be exceeded before a Lift Off event to trigger the Two-finger Inertial Scroll. A high value indicates that a bigger distance should be passed to activate a Two-finger Inertial Scroll gesture.
volatile uint8_t	dtInScrActiveDistanceY	This parameter sets the number of pixels in Y direction that has to be exceeded before a Lift Off event to trigger the Two-finger Inertial Scroll. A high value indicates that a bigger distance should be passed to activate a Two-finger Inertial Scroll gesture.
volatile uint8_t	dtInScrCountLevel	This use can select Low or High levels of the Two-finger Inertial count. The decayCount decays through a 64-byte array or a 32-byte array. A low Two-finger Inertial Scroll count level selects a 32-byte array and sends a few Inertial scrolls. High = 1; Low = 0;
volatile uint8_t	edgeSwipeActiveDistance	This parameter sets the minimum active step distance (in pixels) from the point of a Touchdown, near the edge, that has to be exceeded before the gesture is triggered. The path covered by the finger should not exceed the top angle threshold (topAngleThreshold) and the bottom angle threshold (bottomAngleThreshold).
volatile uint8_t	topAngleThreshold	This parameter defines the maximum angle (in degrees) that the path of a finger can subtend

		on the point of a Touch Down, near the edge. A 1 degree angle means that the user can do gestures only on a single line.
volatile uint8_t	bottomAngleThreshold	This parameter defines the maximum angle (in degrees) that the path of a finger can subtend on the point of a Touchdown, near the edge. A 1 degree angle means that the user can do gestures only on a single line.
volatile uint8_t	widthOfDisambiguation	This parameter sets the edge area for the Edge Swipe gestures. A valid Edge Swipe gesture should start within the width of the disambiguation region. Increasing this parameter makes it easier for the user to find the edge, but it reduces the useful area of the trackpad.
volatile uint8_t	STPanDebounce	This parameter sets the number of similar, sequential pan gestures that should be performed before the pan motion is considered valid. This parameter is for the One-finger Pan motions.
volatile uint8_t	DTPanDebounce	This parameter sets the number of similar, sequential pan gestures that should be performed before the pan motion is considered valid. This parameter is for the Two-finger Pan motions.
volatile uint8_t	DTZoomDebounce	This parameter sets the number of sequential Zoom gestures in a particular direction (in or out) that has to be observed before the Zoom gesture is deemed valid. The default is 2. For example, for a Zoom in action, three Zoom in gestures must be observed in sequence before reporting the action to the caller.
volatile uint8_t	DTPanToZoomDebounce	This parameter sets the number of Zoom gestures that will be ignored after a Two-finger Pan gesture is observed. This is used to filter out Zoom gestures that inevitably occur during a transition from the Two-finger Pan. If you set this parameter to 0 you will observe debounced Zoom gestures right after Two-finger Pan gestures.
volatile uint8_t	rotateDebounce	This parameter sets the number of sequential Pan gestures in a particular direction that have to be observed before the Rotate gesture is deemed invalid. For example, if this parameter is set to 20 and you are performing a Rotate action, then the touch cannot continue in the same direction for 20 Pan counts and still have a valid Rotate gesture. After this threshold is reached, the reported gesture causes to be a Rotate and the corresponding Pan gesture is reported.
volatile uint8_t	completedDebounce	Determines the number of motion gestures that must be detected before a subsequent gesture is considered as a completed gesture;

		for example, a debounce of 2 requires three consecutive gestures.
volatile uint8_t	doubleClickRadius	This parameter sets the maximum radius in resolution units that the second Click in a Double Click sequence can extend. If the second Click occurs outside this radius, the Double Click sequence is discarded.
volatile uint8_t	clickRadiusX	These parameters set the maximum X-axis displacement for Click gestures (One-finger Click, Two-finger Click and constituents of One-finger Double Click).
volatile uint8_t	clickRadiusY	These parameters set the maximum Y-axis displacement for Click gestures (One-finger Click, Two-finger Click and constituents of One-finger Double Click).
volatile uint16_t	settlingTimeout	This parameter sets the minimum duration of how long to wait prior to decoding when touches switch from a single-touch to dual-touch or vice versa. The time is measured in milliseconds.
volatile uint16_t	resolutionX	Resolution X axis.
volatile uint16_t	resolutionY	Resolution Y axis.
volatile uint16_t	flickSampleTime	This is the maximum time window that will be searched for the flick (in milliseconds).
volatile uint16_t	edgeSwipeTimeout	This is the maximum time window that will be searched for the flick (in milliseconds).
volatile uint16_t	DTClickTimeoutMax	This parameter sets the maximum time during which two touches can be on the panel before being disqualified as a Two-finger Click event. The time is measured in milliseconds.
volatile uint16_t	DTClickTimeoutMin	This parameter sets the minimum duration that two touches need to be on the panel before a Two-finger Click event is registered. This filters very rapid dual-touch clicks. This helps applications define very deliberate dual-touch click events. This parameter should be set lower than the dual-touch maximum click timeout parameter.
volatile uint16_t	STClickTimeoutMax	This parameter sets the maximum duration that a touch has to be on the panel to consider this gesture as a One-finger Single Click. If the touch is placed on the panel for longer than this value, CapSense_TMG_NO_GESTURE event is sent.
volatile uint16_t	STClickTimeoutMin	This parameter sets the minimum duration that a Click can stay on the panel to qualify as a One-finger Click. This can be used by applications to set how deliberately a Single Click operation must be performed. This helps filter out noisy events or very rapid clicks which are usually performed inadvertently. This parameter should be set lower than the One-

		finger max click timeout parameter.
volatile uint16_t	STDoubleClickTimeoutMax	This parameter is the maximum allowable time between the release times of two sequential clicks in order the motion is be considered a Double Click.
volatile uint16_t	STDoubleClickTimeoutMin	This parameter sets the minimum duration between the release times of two sequential clicks in order the motion is considered a Double Click.
volatile uint8_t	groupMask	This parameter keeps masks for the 4 gesture groups. The four most significant bits are used. Each bit represents a group. The most significant bit is associated with 4-th group. This parameter is used to enable/disable reporting for groups. When a mask is set to 0, reporting is disabled for the corresponding group.
volatile uint8_t	group1Start	Gesture mask group internal parameter
volatile uint8_t	group1End	Gesture mask group internal parameter
volatile uint8_t	group2Start	Gesture mask group internal parameter
volatile uint8_t	group2End	Gesture mask group internal parameter
volatile uint8_t	group3Start	Gesture mask group internal parameter
volatile uint8_t	group3End	Gesture mask group internal parameter
volatile uint8_t	group4Start	Gesture mask group internal parameter
volatile uint8_t	group4End	Gesture mask group internal parameter

struct CapSense_TMG_BALLISTIC_MULT

Go to the top of the Data Structures section.

Data Fields:

uint8_t	touchNumber	Number of detected fingers (0, 1 or 2)
uint8_t	accelCoeff	Acceleration Coefficient
uint8_t	speedCoeff	Speed Coefficient
uint8_t	divisorValue	Divisor Value
uint8_t	speedThresholdX	Speed Threshold X
uint8_t	speedThresholdY	Speed Threshold Y

Memory Usage

The Component Flash and RAM memory usage varies significantly depending on the compiler, device, number of APIs called by the application program and Component configuration. The table below provides the total memory usage of firmware for a given Component configuration.

The measurements were done with an associated compiler configured in the 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.

PSoC 4 (GCC)

The following Component configuration is used to represent the memory usage:

Configuration	Memory Consumption	
	Flash	SRAM
Configuration #1: CSX Matrix Button – One widget with 4 Rx and 8 Tx.		
Configuration #1	< 4800	< 500
Configuration #1 + *Enable multi-frequency scan* is enabled	< 5200	< 1000
Configuration #2: CSX Touchpad – One widget with 9 Rx and 4 Tx.		
Configuration #2	< 7100	< 800
Configuration #2 + *Enable multi-frequency scan* is enabled	< 7500	< 1350
Configuration #3: CSD Buttons – Three widgets with 4, 3 and 3 sensors in each widget, and Manual tuning mode is selected.		
Configuration #3	< 5500	< 300
Configuration #3 + *Enable multi-frequency scan* is enabled	< 6000	< 450
Configuration #3 + *Enable self-test library* is enabled	< 10000	< 350
Configuration #3 + *SmartSense (Full Auto-Tune)* mode is selected	< 6600	< 400
Configuration #3 + All firmware raw count filters enabled. The following parameters are used to enable filters: *Enable IIR filter (First order)*, *Enable average filter (4-sample)* and *Enable median filter (3-sample)*.	< 6100	< 400

Note The configurations consist of the default customizer configuration except where noted. The default customizer configuration includes:

- All filters disabled. The *Enable IIR filter (First order)*, *Enable average filter (4-sample)* and *Enable median filter (3-sample)* parameters are disabled.
- The *Enable compensation IDAC* parameter is enabled.
- The *Enable IDAC auto-calibration* parameter is enabled.

CapSense Tuner

The CapSense Component provides a graphical-based Tuner application for debugging and tuning the CapSense system.

To make the Tuner application work, a communication Component is added to the project and then the Component register map is exposed to the Tuner application. The CapSense Tuner application works with the EZI2C and UART Communication Components.

To edit the parameters, use the Tuner application and apply the new settings to the device using the **To Device** button. You can do this when using *Manual* or *SmartSense (Hardware parameters only)* modes for tuning.

- To edit the threshold parameters, use *SmartSense (Hardware parameters only)* mode.
- To edit all the parameters, use *Manual* mode.
- When *SmartSense (Full Auto-Tune)* is selected for CSD tuning mode, the user has the Read only access parameters (except the **Finger capacitance** parameter).

The **To Device** button is available when the *Synchronized* control in the *Graph Setup Pane* is enabled and any parameter in the Tuner is changed. The *Synchronized* control can be enabled when the FW flow regularly calls the CapSense_RunTuner() function. If this function is not present in the application code, then *Synchronized* communication mode is disabled.

This section describes the parameters used in the Tuner UI interface. For details of the tuning and system design guidelines, refer to the *Getting Started with CapSense®* document and the product-specific *CapSense design guide*.

Tuning Quick Start with EzI2C

Refer to the *Quick Start* section for tuning with the EzI2C interface.

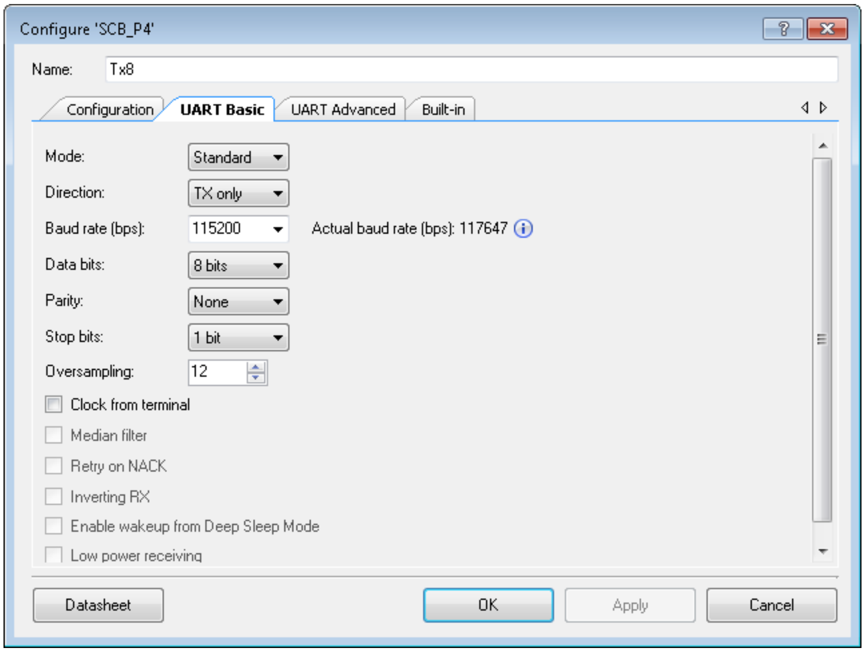
Tuning Quick Start with UART

The following steps to show how to set up CapSense tuning across a UART communication channel.

Step 1: Place and Configure UART (SCB) Component

1. Drag a UART (SCB) Component from the Component Catalog onto the schematic to add a UART communication interface to the project. This UART interface is required for the Tuner GUI to monitor the Component parameters in real time.
2. Double-click the UART (SCB) Component.

3. In the **UART Basic** tab, set the parameters as shown:



- Type the desired Component name (in this case: Tx8).
- Set Direction to TX only. The CapSense Tuner allows only monitoring data received from a device and does not support Synchronized Communication mode.
- Set the Data Rate (bps) to 115200.
- Set the Data Width to 8 bits.

4. Click **OK** to close the GUI and save changes.

Step 2: Assign Tx Pin in Pin Editor

Open the Pin Editor and assign a physical pin to \Tx8:tx\.

If you are using a Cypress kit, refer to the kit user guide for the pin selections. This bridge firmware enables the UART communication between the PSoC and the Tuner application across the USB. You can also use a MiniProg3 debugger/programmer kit as the USB-UART Bridge.

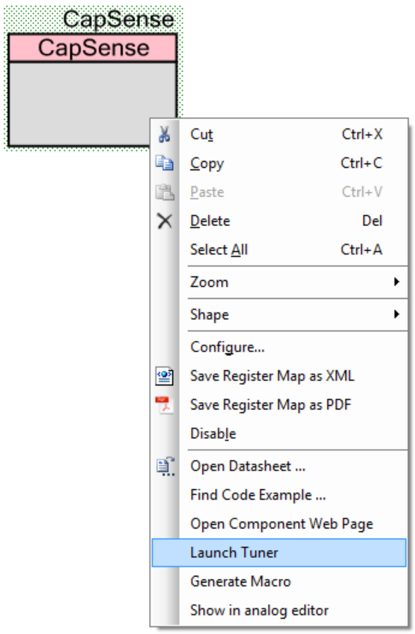
Step 3: Modify Application Code

Replace your *main.c* file from *Step 4* in the *Quick Start* section with the following code:

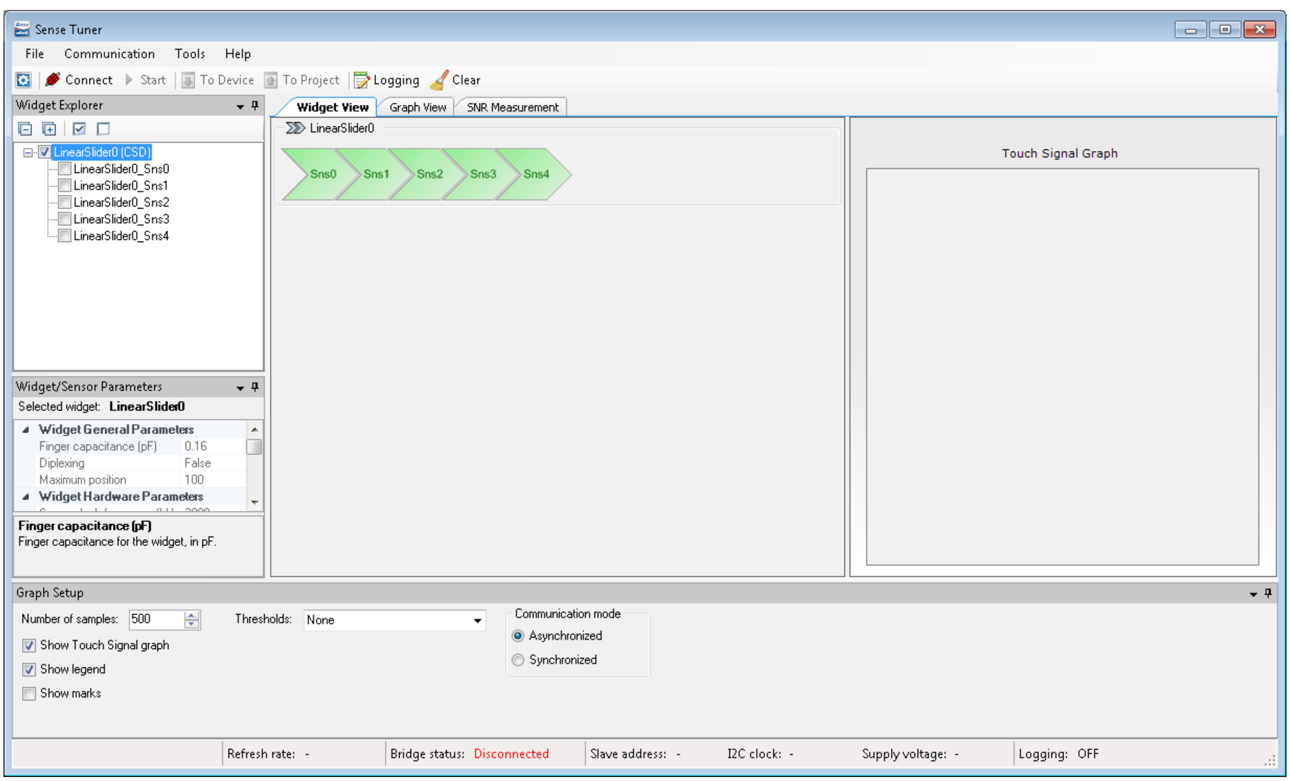
```
#include "project.h"

uint8 header[] = {0x0Du, 0x0Au};
uint8 tail[] = {0x00u, 0xFFu, 0xFFu};

int main()
{
    __enable_irq();                             /* Enable global interrupts. */

    Tx8_Start();                                /* Start UART SCB Component */

    CapSense_Start();                           /* Initialize Component */
    CapSense_ScanAllWidgets();                  /* Scan all widgets */

    for(;;)
    {
        /* Do this only when a scan is done */
        if(CapSense_NOT_BUSY == CapSense_IsBusy())
        {
            CapSense_ProcessAllWidgets();       /* Process all widgets */

            /* Send packet header */
            Tx8_SpiUartPutArray((uint8 *)(&header), sizeof(header));
            /* Send packet with CapSense data */
            Tx8_SpiUartPutArray((uint8 *)(&CapSense_dsRam), sizeof(CapSense_dsRam));
            /* Send packet tail */
            Tx8_SpiUartPutArray((uint8 *)(&tail), sizeof(tail));

            if (CapSense_IsAnyWidgetActive())   /* Scan result verification */
            {
                /* add custom tasks to execute when touch detected */
            }

            CapSense_ScanAllWidgets();          /* Start next scan */
        }
    }
}
```

Step 4: Launch Tuner Application

Right-click the CapSense Component in the schematic and select **Launch Tuner** from the context menu.



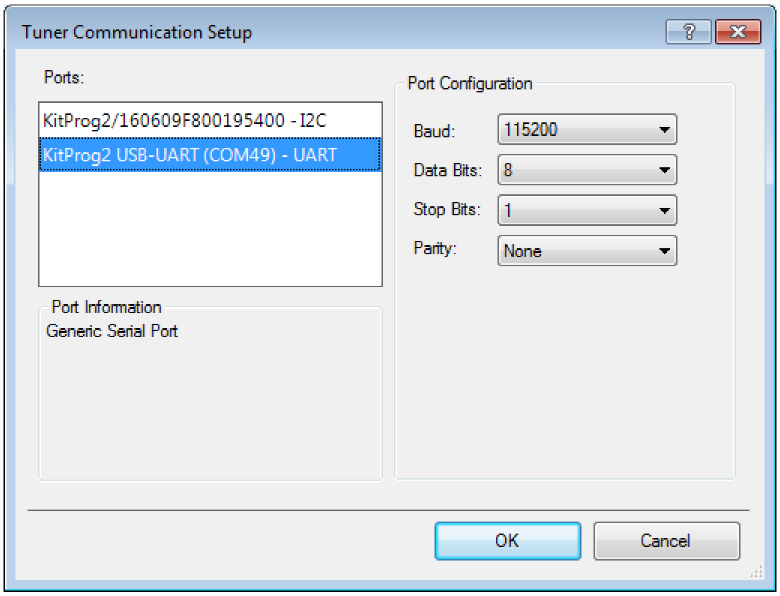
The *CapSense Tuner* application opens as shown. Note that the 5-element slider, called LinearSlider0, appears in the Widget View panel automatically.



Step 5: Configure Communication Parameters

To establish communication between the Tuner and a target device, configure the Tuner communication parameters to match those of the UART SCB Component.

1. Open the Tuner Communication Setup dialog from PSoC Creator by selecting *Tools > Tuner Communication Setup…*



2. Select the appropriate UART communication device which is KitProg2 (or MiniProg3) and set the following parameters:
   - **Baud**: 115200
   - **Data Bits**: 8
   - **Stop Bits**: 1
   - **Parity**: None

   Note The parameters in the Tuner Communication Setup must be identical to the parameters in the UART SCB Component Configure dialog (see *Tuning Quick Start with UART*).

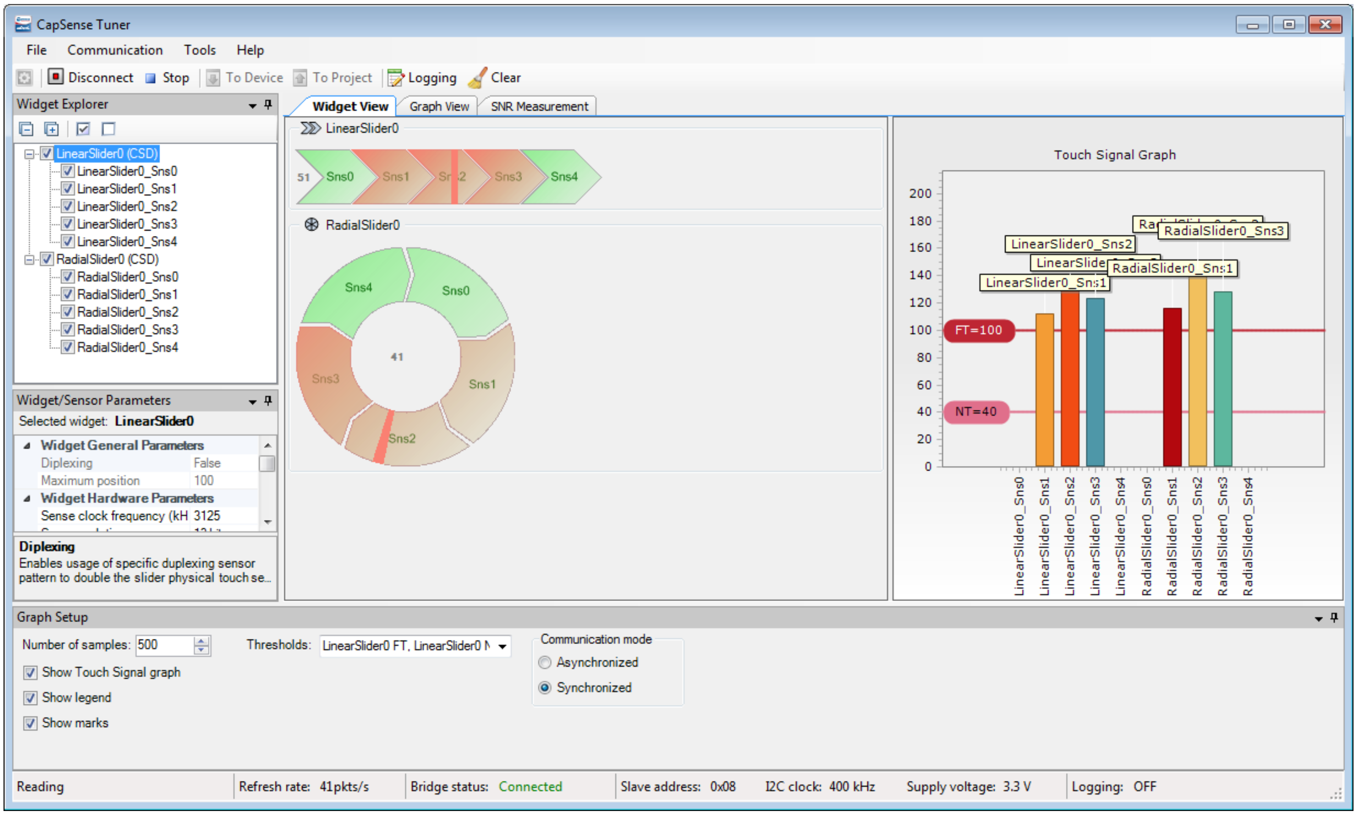
Step 6: Start Communication

Click *Connect* to establish connection and then *Start*s to extract data.

The *Synchronized* control in the *Graph Setup Pane* is grayed out and is not available with the UART communication. The Tuner is not able to write any data into a device. Refer to *Graph Setup Pane* for details of the synchronized operation.

The *Status Bar* shows a communication bridge connection status and communication refresh rate. The status of the LinearSlider0 widget appears in the *Widget View* and signals for each of the five sensors – in the *Graph View*. Touch the sensors on the kit to observe the CapSense operation.

General Interface



The application consists of the following tabs:

- *Widget View* – Displays the widgets, their touch status and the touch signal bar graph.
- *Graph View* – Displays the sensor data charts.
- *SNR Measurement* – Provides the SNR measurement functionality.
- *Touchpad View* – Displays the touchpad heatmap.
- *Gesture View* – Displays the Gesture operation.

Menus

The main menu commands to control and navigate the Tuner:

- **File > Apply to Device (Ctrl + D)** – Commits the current values of the widget/sensor parameters to the device. This item becomes active if a value of any configuration parameter from the Tuner application is changed (i.e. if the parameter values in the Tuner and the device are different). This is an indication that the changed parameter values need to be applied to the device.

- **File > Apply to Project (Ctrl + S)** – Commits the current values of widget / sensor parameters to the CapSense Component instance. The changes are applied after the Tuner is closed and the Customizer is opened. Refer to the *Procedure to Save Tuner* Parameters section for details of merging parameters to a project.
- **File > Save Graph… (Ctrl + Shift + S)** – Opens the dialog to save the current graph as a PNG image. The saved graph depends on the currently selected view: it is *Touch Signal Graph* for *Widget View* (only when shown), a combined graph with Sensor Data, Sensor Signal and Status for Graph View, and SNR Raw counts graph for the SNR Measurement View.
- **File > Exit (Alt+F4)** – Asks to save changes if there are any and closes the Tuner. Changes are saved to the PSoC Creator project (merged back by the customizer).
- **Communication > Connect (F4)** – Connects to the device via a communication channel selected in the Tuner Communication Setup dialog. When the channel was not previously selected, the Tuner Communication dialog will open.
- **Communication > Disconnect (Shift+F4)** – Closes the communication channel with the connected device.
- **Communication > Start (F5)** – Starts reading data from the device.

  If communication does not start and the dialog *"Checksum mismatch for the data stored…"* or *"There was an error reading data…"* appears the following reasons are possible:

  - The invalid configuration of the communication channel (Slave address / Data rate / Sub-address size)
  - The invalid data buffer exposed via the communication protocol (not *CapSense_dsRam* / wrong header-tail of packet at UART communication)
  - The latest customizer parameters modification was not programmed into the device.
  - Edits performed in the customizer during a tuning session: the Tuner must be closed and opened again after the customizer update.
  - The Tuner is opened for the wrong project.
- **Communication > Stop (Shift+F5)** – Stops reading data from the device.
- **Tools > Tuner Communication Setup… (F10)** – Opens the configuration dialog to set up a communication channel with the device.
- **Tools > Options** – Opens the configuration dialog to set up different Tuner preferences.
- **Help > Help Contents (F1)** – Opens the CapSense Component datasheet.

Toolbar

Contains frequently used buttons that duplicate the main menu items:

-  – Duplicates the **Tools > Tuner Communication Setup** menu item
-  – Duplicates the **Communication > Connect** menu item
-  – Duplicates the **Communication > Disconnect** menu item
-  – Duplicates the **Communication > Start** menu item
-  – Duplicates the **Communication > Stop** menu item
-  – Duplicates the **File > Apply to Device** menu item
-  – Duplicates the **File > Apply to Project** menu item
-  – Starts data logging into a specified file
-  – Stops data logging
-  – Clears the Tuner graphs.

Status Bar

The status bar displays information related to the communication state between the Tuner and the device:

- **Current operation mode of Tuner** – Either **Reading** (when Tuner is reading from the device), **Writing** (when the Write operation is in progress), or empty (idle – no operation performed).
- **Refresh rate** – A count of read samples performed per second. The count depends on multiple factors: the selected communication channel, communication speed, and amount of time needed to perform a single scan.
- **Bridge status** – Either **Connected**, when the communication channel is active, or **Disconnected** otherwise.
- **Slave address** [I2C specific] – The address of the I2C slave configured for the current communication channel.
- **I2C clock** [I2C specific] – The data rate used by the I2C communication channel.
- **Supply voltage** – The supply voltage.
- **Logging** – Either **ON** (when the data logging to a file in progress) or **OFF** otherwise.

Widget Explorer Pane

The Widget explorer pane contains a tree of widgets and sensors used in the CapSense project. The Widget nodes can be expanded/collapsed to show/hide widget's sensor nodes. It is possible to check/uncheck individual widgets and sensors. The Widget checked status affects its visibility in the *Widget View*, while the sensor checked status is controlling the visibility of the sensor raw count / baseline / signal / status graph series in the Graph View and signals in the *Touch Signal Graph* on the *Widget View*.

Selection of a widget or sensor in the *Widget Explorer Pane* updates the selection in the *Widget/Sensor Parameters Pane.* Selecting multiple widget or sensor nodes allows editing multiple parameters simultaneously. For example, you can edit the Finger Threshold parameter for all widgets simultaneously.

Note For the CSX widgets, the sensor tree displays individual nodes (Rx0_Tx0, Rx0_Tx1 …), contrary to the customizer where the CSX electrodes are displayed (Rx0, Rx1 … Tx0, Tx1 ...).

The toolbar at the top of the widget explorer provides easy access to commonly used functions: buttons ⊞ ⊟ can be used to expand/collapse all sensor nodes simultaneously, and ☑ ☐ to check/uncheck all widgets and sensors.

Widget/Sensor Parameters Pane

The Widget/Sensor parameters pane displays the parameters of the widget or sensor selected in the Widget Explorer tree. The grid is similar to the grid on the *Widget Details* tab in the CapSense customizer. The main difference is that some parameters are available for modification in the customizer, but not in the Tuner. This pane includes the following parameters:

- **Widget General Parameters** – Cannot be modified from the Tuner because corresponding parameter values reside in the Flash widget structures that cannot be modified at runtime.
- **Widget Hardware Parameters** – Cannot be modified for the CSD widgets when *CSD tuning mode* is set to *SmartSense (Full Auto-Tune)* or SmartSense Hardware in the CapSense Configure dialog. In *Manual* tuning mode (for both CSD and CSX widgets), any change to *Widget Hardware Parameters* requires hardware re-initialization. This can be performed only if the Tuner communicates with the device in Synchronizedmode.
- **Widget Threshold Parameters** – Cannot be modified for the CSD widgets when *CSD tuning mode* is set to *SmartSense (Full Auto-Tune)* in the customizer. In *Manual* tuning mode (for both CSD and CSX widgets), the threshold parameters are always writable (Synchronized mode is not required). The exception is the *ON debounce* parameter that also requires a Component restart (in the same way as the hardware parameters).

- **Sensor Parameters** – Sensor-specific parameters. The Tuner application displays only *IDAC Values* or/and *Compensation IDAC value*. The parameter is not present for the CSD widget when *Enable compensation IDAC* is disabled on the customizer *CSD Settings* tab. When CSD *Enable IDAC auto-calibration* or/and CSX *Enable IDAC auto-calibration* is enabled, the parameter is Read-only and displays the IDAC value as calibrated by the Component firmware. When auto-calibration is disabled, the IDAC value entered in the Configure dialog is shown. If the Tuner is in **Synchronized** mode, you can edit the value and apply it to the device.

- **Filter Parameters** and **Centroid Parameters** – Cannot be modified at run-time from the Tuner, because unlike the other parameters, these parameter values reside in the Flash widget structures that cannot be modified at run-time.

- **Gesture Parameters** – *Synchronized* communication mode must be selected to update the Gesture parameters during run-time from the Tuner application.

Graph Setup Pane

The Graph Setup pane provides quick access to different Tuner configuration options that affect the Tuner graphs display.

- **Number of samples** – Defines the total amount of data samples shown on a single graph.

- **Show legend** – Displays the sensor series descriptions (with names and colors) in graphs when checked (Sensor Data/Sensor Signal/Status graphs in the *Graph View* and a *Touch Signal Graph* in the *Widget View*).

- **Show marks** – When checked, the sensor names appear as marks over the signal bars in the *Touch Signal Graph*.

- **Show Touch Signal graph** – When checked, a *Touch Signal Graph* appears.

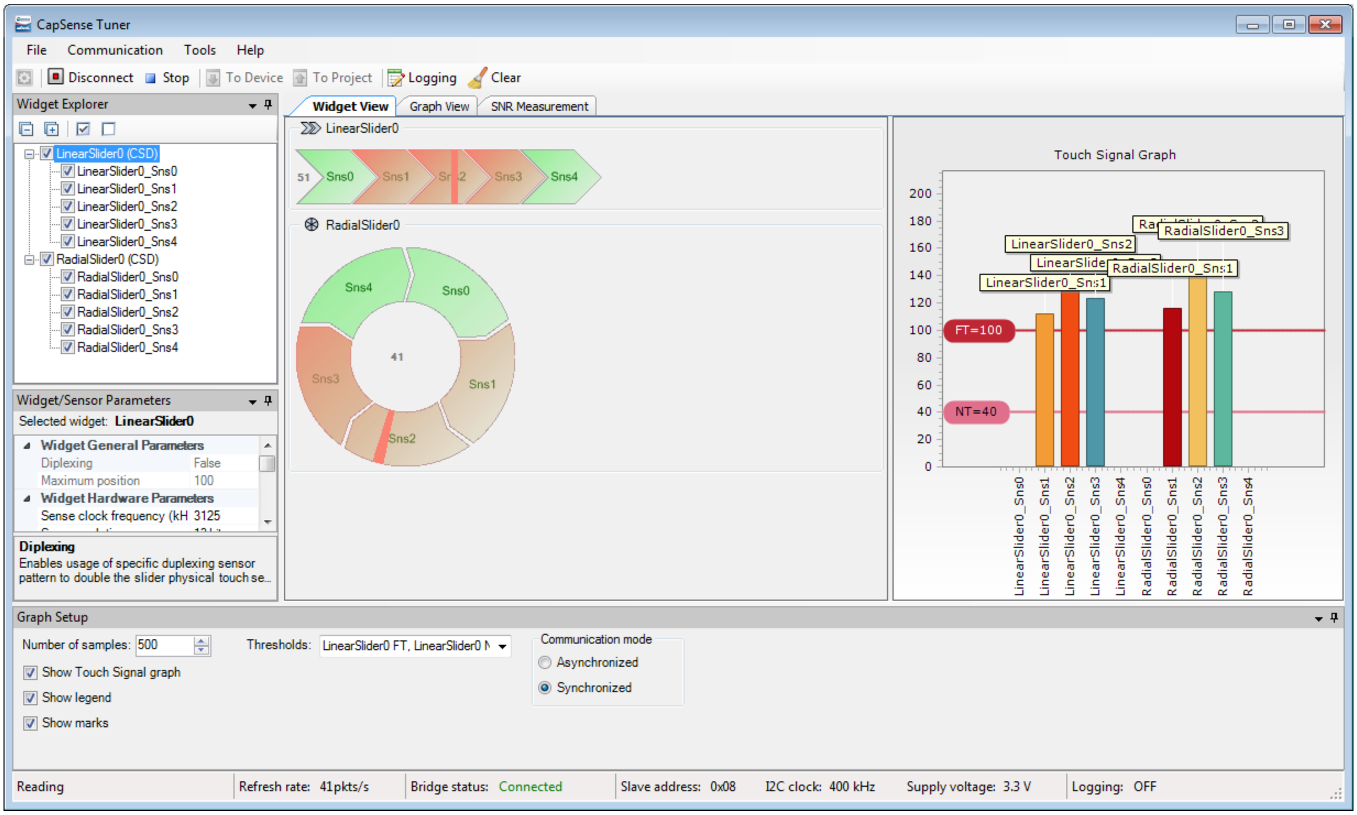
- **Thresholds** – A drop-down menu with checkboxes to enable the threshold visualization in the *Touch Signal Graph* and a Sensor Signal graph in the *Graph View* tab.

- **Communication mode** – Selects Tuner communication mode with a device. Two options are available (when the EZI2C Component is used):

  - **Synchronized** – This communication mode is available when a FW loop periodically calls a corresponding Tuner function: CapSense_RunTuner(). When Synchronized Communication mode is selected, the CapSense Tuner manages an execution flow by suspending scanning during the Read operation. Before starting data reading, the Tuner sends a **OneScan** command to the device. The device performs one cycle of scanning and the second call of CapSense_RunTuner() hangs the FW flow until a new command is received. The Tuner reads all the needed data and sends a **OneScan** command again.

- **Asynchronized** – When selected, the Tuner reads data asynchronously to sensor scanning. Because reading data by the CapSense Tuner and data processing happen asynchronously, the CapSense Tuner may read the updated data only partially. For example, the device updates only the first sensor data and the second sensor is not updated yet. At this moment, the CapSense Tuner is reading the data. As a result, the second sensor data is not processed.

Widget View



Provides a visual representation of all widgets selected in the *Widget Explorer Pane*. If a widget consists of more than one sensor, individual sensors may be selected to be highlighted in the *Widget Explorer Pane* and *Widget/Sensor Parameters Pane.*

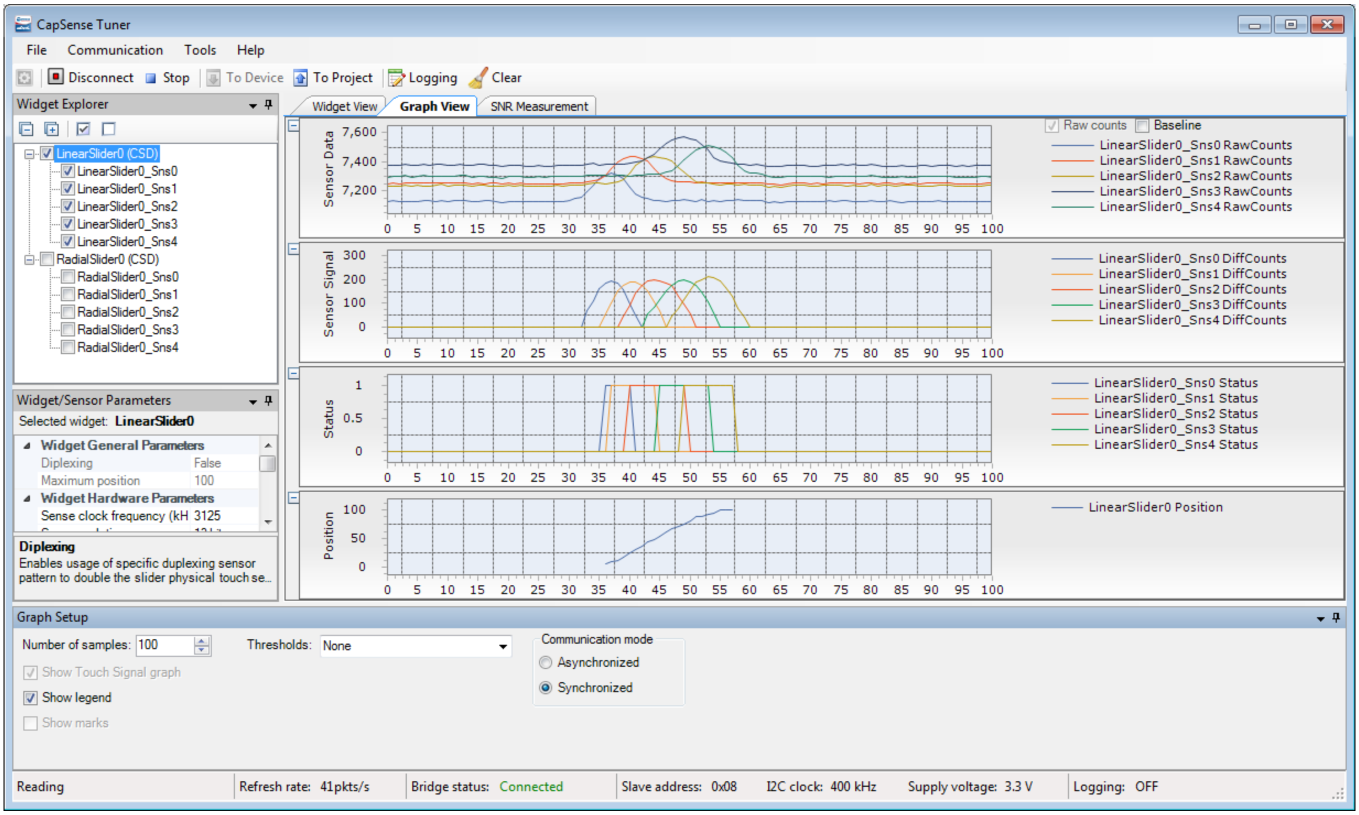
The Widget sensors are highlighted red when the device reports their touch status as active.

Some additional features are available depending on the widget type:

Touch Signal Graph

The Widget view also displays Touch Signal Graph when the "Display Touch Signal graph" checkbox is checked in the *Graph Setup Pane*. This graph contains a touch signal level for each sensor selected in the *Widget Explorer Pane*.

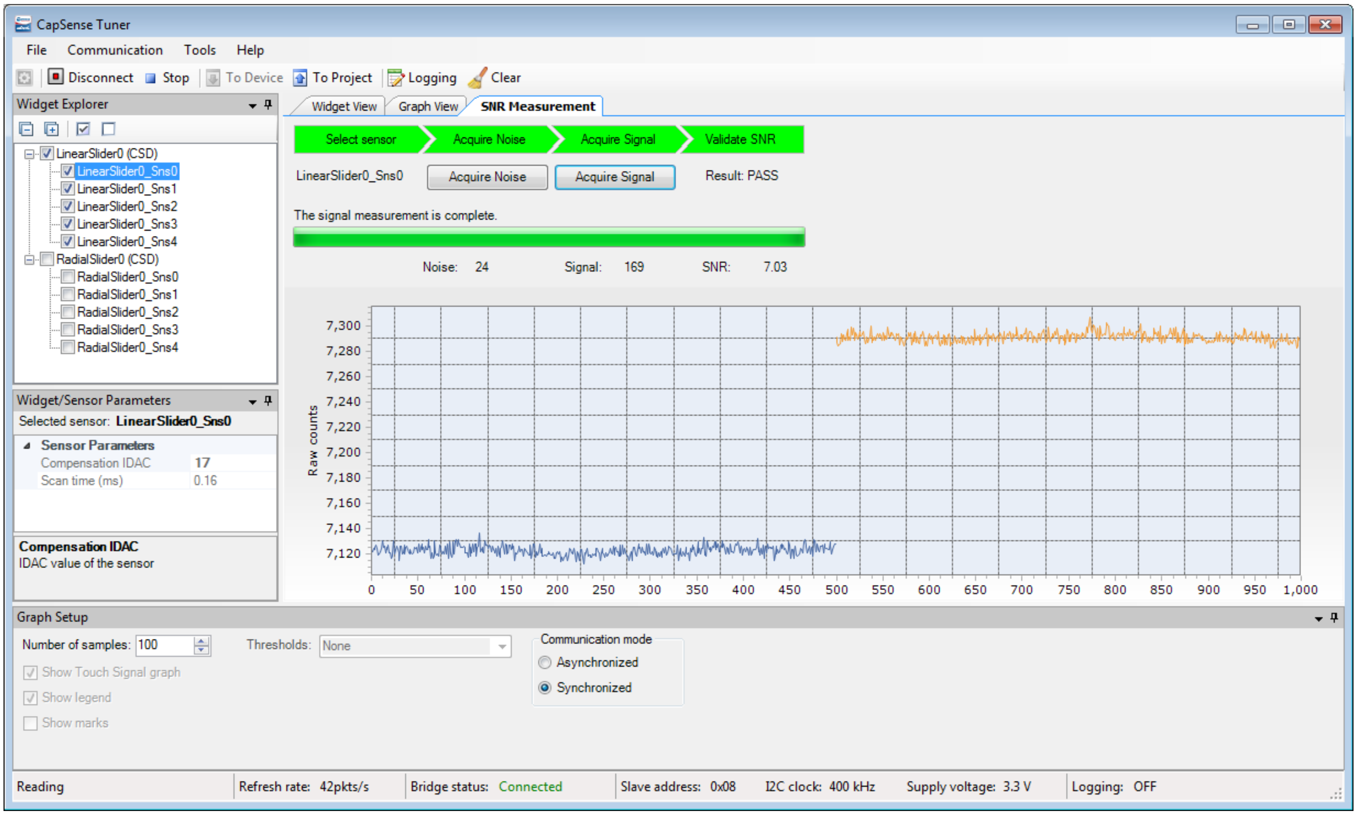
Graph View



Displays graphs for selected sensors in the *Widget Explorer Pane*. The following charts are available:

- **Sensor Data graph** – Displays raw counts and baseline. Use the checkboxes on the right to select the series to be displayed:
  - Raw counts and baseline
  - Raw counts
  - Baseline
- **Sensor Signal graph** – Displays a signal difference.
- **Status graph** – Displays the sensor status (Touch/No Touch). For proximity sensors, it also shows the proximity status (at 50% of the status axis) along with the touch status (at 100% of the axis).
- **Position graph** – Displays touch positions for the *Linear Slider*, *Radial Slider* and *Touchpad* widgets.

SNR Measurement



The **SNR Measurement** tab allows measuring a SNR (Signal-to-Noise Ratio) for individual sensors.

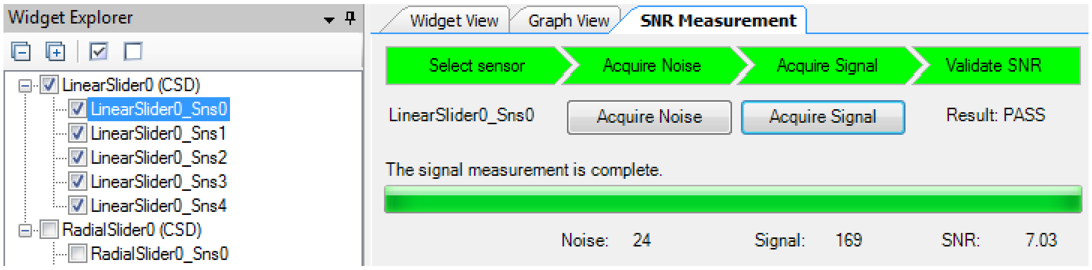
The tab provides UI to acquire noise and signal samples separately and then calculates a SNR basing on the captured data. The obtained value is then validated by a comparison with the required minimum (5 by default, can be configured in the *Tuner Configuration Options*).

Typical Flow of SNR Measurement

1. Connect to the device and start communication (by pressing **Connect**, then **Start** on the toolbar).
2. Switch to the **SNR Measurement** tab.
3. Select a sensor in the *Widget Explorer Pane* located on the left of the **SNR Measurement** tab.
4. Make sure no touch is present on the selected sensor.
5. Press **Acquire Noise**, and wait for the required count of noise samples to be collected.
6. Observe the Noise label is updated with the calculated noise average value.
7. Put a finger on the selected sensor.
8. Press **Acquire Signal**, and wait for the required count of signal samples to be collected.

9. Observe the Signal label is updated with the calculated signal average value
10. Observe the SNR label is updated with the SNR (signal-to-noise ratio).

Description of SNR Measurement GUI



At the top of the **SNR measurement** tab, there is a bar with the status labels. Each label status is defined by its background color:

- **Select sensor** – Green when there is a sensor selected; gray otherwise.
- **Acquire noise** – Green when noise samples are already collected for the selected sensor; gray otherwise.
- **Acquire signal** – Green when signal samples are already collected for the selected sensor; gray otherwise.
- **Validate SNR** – Green when both noise and signal samples are collected, and the SNR is above the valid limit; red when the SNR is below the valid limit, and gray when either noise or signal are not yet collected.

Below the top status labels bar, there are the following controls:

- **Sensor name** – The label selected in the *Widget Explorer Pane* or None (if no sensor selected).
- **Acquire Noise** – This button is disabled when the sensor is not selected or communication is not started. When acquiring noise is in progress, the button can be used to abort the operation.
- **Acquire Signal** – This button is disabled when the sensor is not selected, communication is not started, or noise samples are not yet collected for the selected sensor. When acquiring signal is in progress, the button can be used to abort the operation.
- **Result** – This label shows either N/A (when the SNR cannot be calculated due to noise/signal samples not collected yet), PASS (when the SNR is above the required limit), or FAIL (when the SNR is below the required limit).

Below the controls, there is the status label that displays the current status message and the progress bar that displays the progress of the current operation.

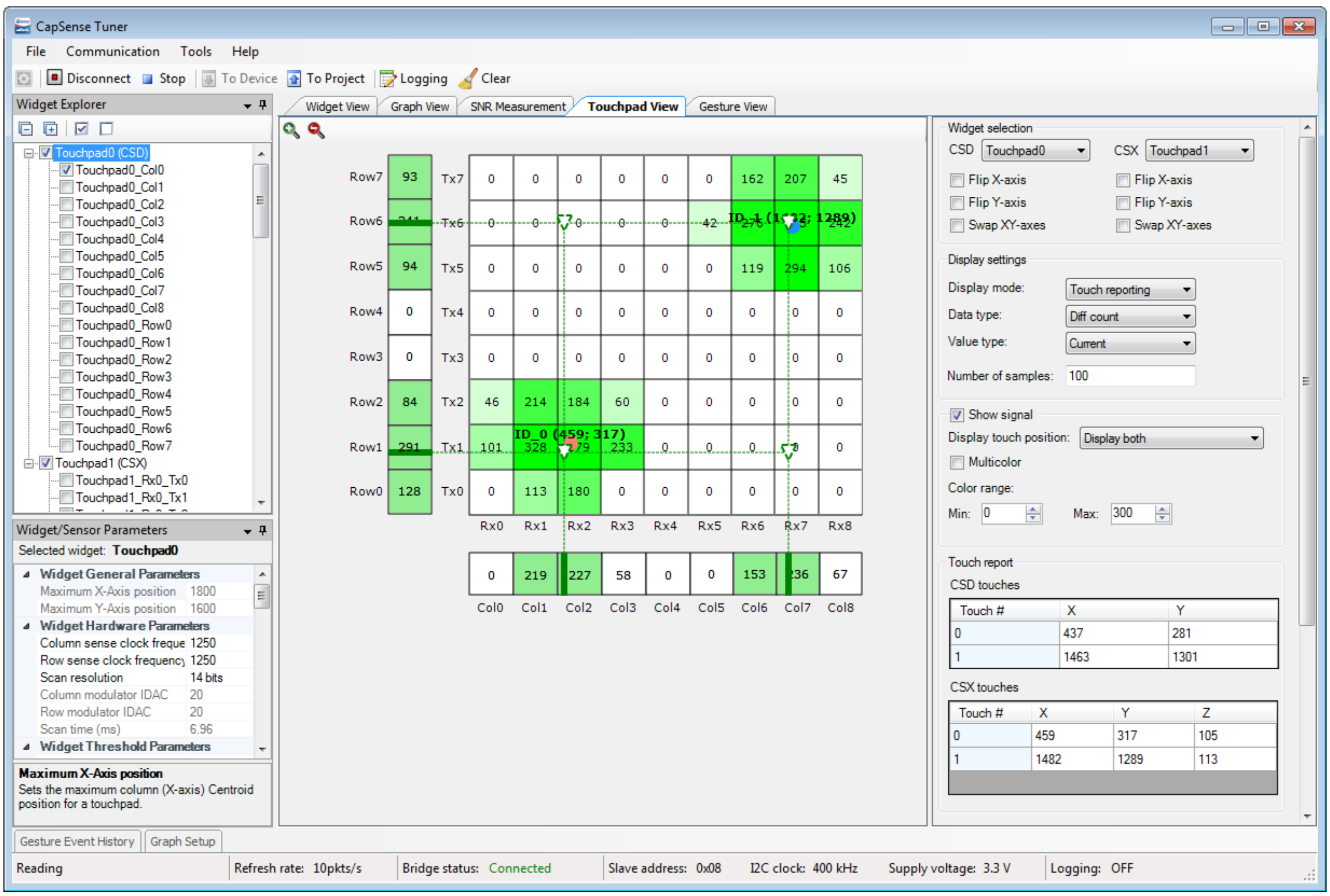
Below the status label, there are the following controls:

- **Noise** – The label that shows the noise average value calculated during the last noise measurement for the selected sensor, or N/A if no noise measurement is performed yet.
- **Signal** – The label that shows the signal average value calculated during the last signal measurement for the selected sensor, or N/A if no signal measurement is performed yet.
- **SNR** – The label that shows the calculated SNR value. This is the result of the Signal/Noise division rounded up to 2 decimal points. When a SNR cannot be calculated, N/A is displayed instead.

Pressing **Clear** on the *Toolbar* clears the graph and collected data to calculate a SNR.

Touchpad View

This tab provides a visual representation of signals and positions of a selected touchpad widget in the heatmap form. Only one CSD and one CSX touchpad can be displayed at a time.



The following options are available:

Widget Selection

Consists of the configuration options for mapping the customer touchpad configuration to the identical representation in the heatmap:

- **CSD combo box** – Selects any CSD touchpad displayed in the heatmap. The CSD combo box is grayed out if the CSD touchpad does not exist in the user design.
- **CSX combo box** – Selects any CSX touchpad displayed in the heatmap. The CSX combo box is grayed out if the CSX touchpad does not exist in the user design.
- **Flip X-axis** – Flips the displayed X-axis correspondingly to the CSD or/and CSX touchpad.
- **Flip Y-axis** – Flips the displayed Y-axis correspondingly to the CSD or/and CSX touchpad.
- **Swap XY-axes** – Swaps the X- and Y-axes for the desired touchpad.

Display settings

Manages heatmap data that to be displayed. These options are available for a CSX touchpad only.

- **Display mode** – The drop-down menu with 3 options for the display format:
  - **Touch reporting** – Shows the current detected touches only.
  - **Line drawing** – Joins the previous and current touches in a continuous line.
  - **Touch Traces** – Plots all the reported touches as dots.
- **Data type** – The drop-down menu to select the signal type to be displayed: Diff count, Raw count, Baseline.
- **Value type** – The drop-down menu to select the type of a value to be displayed: Current, Max hold, Min hold, Max-Min and Average.
- **Number of samples** – Defines a length of history of data for the **Line Drawing**, **Touch Traces**, **Max hold**, **Min hold**, **Max-Min** and **Average** options.

Show signal

Enables displaying data for each sensor if selected. Otherwise, it displays only touches. This option is applicable for the CSX touchpad only.

- **Display touch position** – Defines positions from which the touchpad is displayed. The three options:
  - Display only CSX

  - Display only CSD
  - Display both

- **Multicolor** – When the checked heatmap uses the rainbow color palette to display sensor signals. Otherwise, a monochrome color is used.
- **Color range** – Defines a range of sensor signals within which the color gradient is applied. If a sensor signal is outside of the range, then a sensor color is either minimum or maximum out of the available color palette.

Touch report

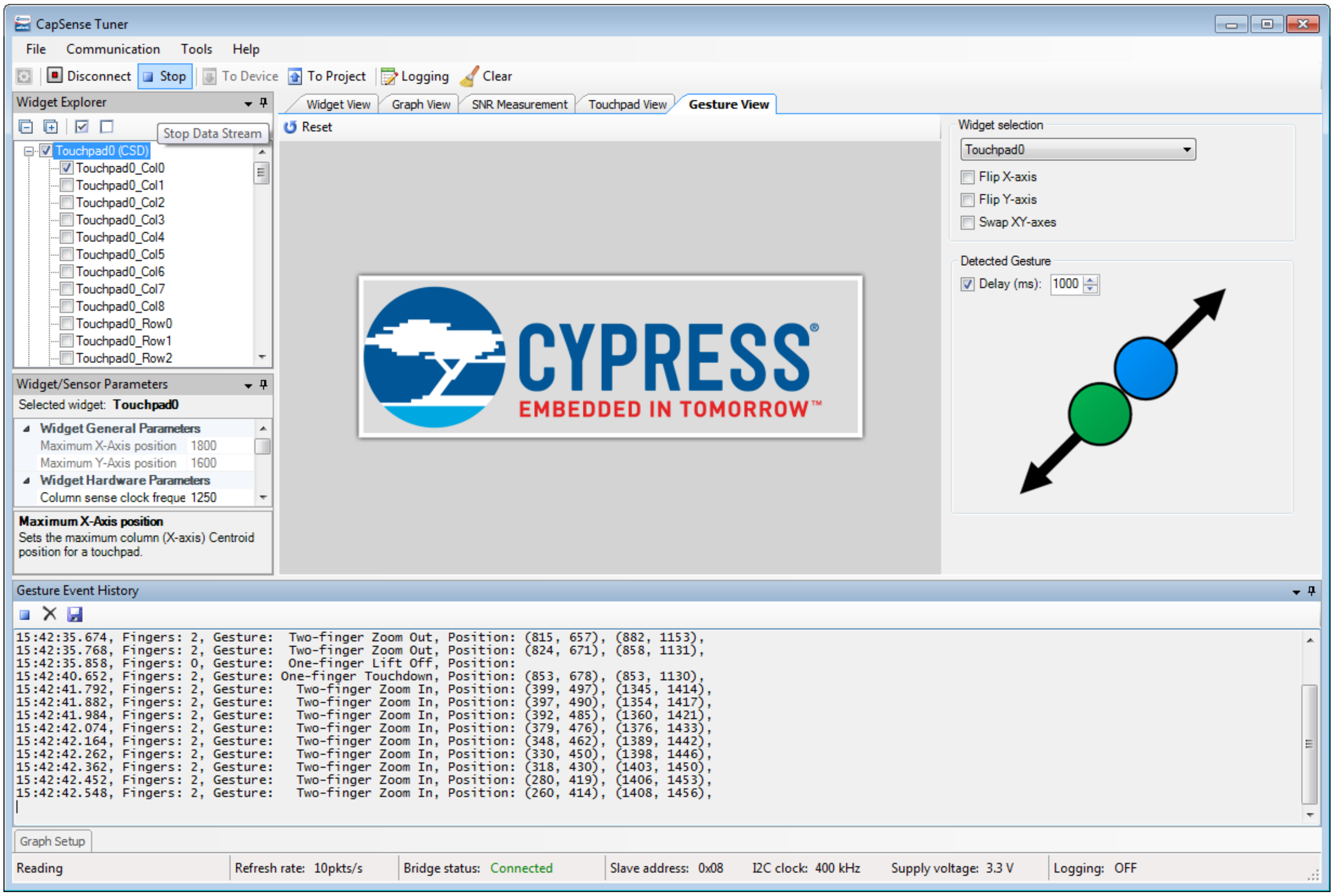
- CSD touches table – Displays the current X and Y touch position of the CSD touchpad configured in **CSD combo box*.*** If the CSD touchpad is neither configured nor touch-detected, the touch table is empty. When *Two finger* detection is enabled for a CSD touchpad, then two touch positions are reported.
- CSX touches table – Displays the X, Y, Z values of the detected touches of the CSX touchpad configured in **CSX combo box*.*** If the CSX touchpad is neither configured nor touch-detected, the touch table is empty. The Component supports simultaneous detection up to three touches for a CSX touchpad touch, so the touch table displays all the detected touches.

Detected gesture

If the selected touchpad in *CSD combo box* or *CSX combo box* has enabled gestures, then this pane displays an image of a detected gesture.

Gesture View

This tab provides a visual representation of gestures. This tab can display gestures from one widget at a time.



Note Use of *Synchronized* communication mode or UART communication is recommended for Gesture validation, to make sure no gesture events such as a touchdown or lift off is missed during communication.

Widget Selection

Allows selecting a widget and controls that the display in the Tuner matches the hardware orientation.

- **Combo box** – Selects the widget with Gesture enabled to display the Gesture from the selected widget on this pane.
- **Flip X-axis** – Flips the direction of the X-axis.
- **Flip Y-axis** – Flips the direction of the Y-axis.
- **Swap XY-axes** – Swaps the X- and Y-axes for the selected widget.

Detected Gesture

Provides visual indication for a detected Gesture.

If the delay check box is enabled, a Gesture picture is displayed for the specified time-interval. If disabled, the last reported gesture picture is displayed until a new Gesture is reported.

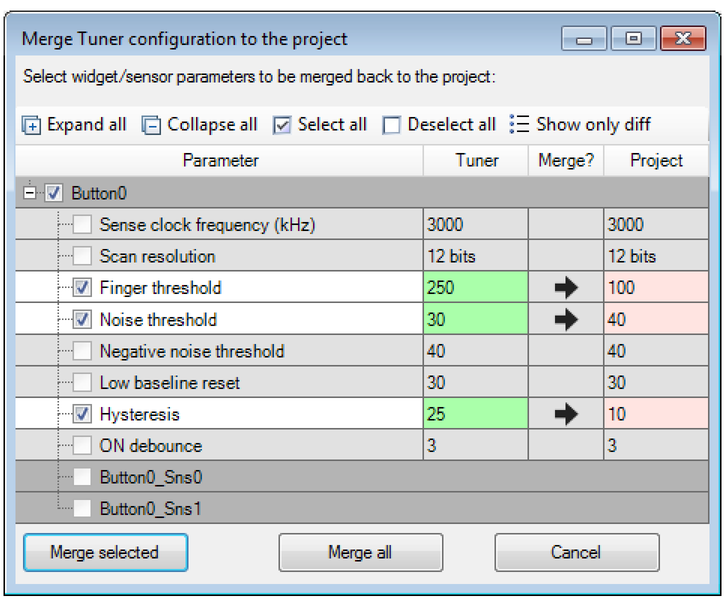
If a spurious condition or Gesture is reported, the following image is displayed.

Procedure to Save Tuner Parameters

Changes to widget / sensor parameters made in the Tuner GUI are not automatically updated to the PSoC Creator project, unless specifically saved. Use the following steps to save the updated tuning parameters to project:

1. If any parameter is changed during the tuning process in the Tuner GUI, the **Apply to Project** button is active. Click this button to apply the new parameters to the project and follow the instructions.
2. Close the Tuner GUI.
3. Open the Component Configure dialog.

   The following dialog asks to merge the Tuner configuration updates back to the customizer:



4. Click the **Merge all** or **Merge selected** buttons to apply the Tuner's changed parameters to the project. Click **Cancel** to leave the Component parameters unchanged.

   Note Some parameters can be changed by the device at run-time when one of the following features is enabled:

   - *SmartSense Auto-tuning*
   - *CSD Enable IDAC auto-calibration*
   - *CSX Enable IDAC auto-calibration*

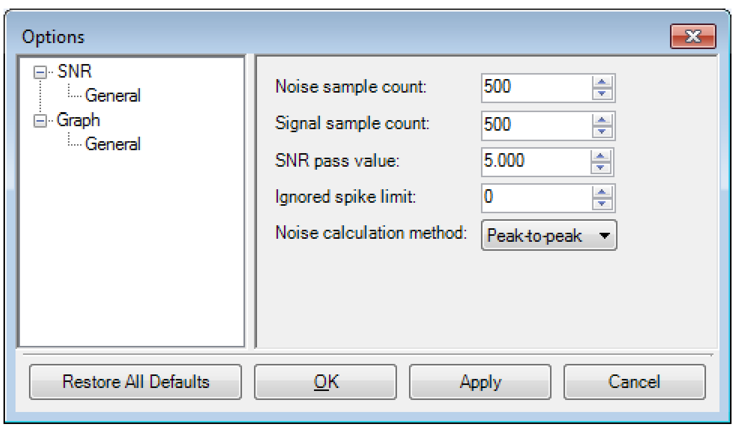
   The Tuner automatically picks up the changed parameters from a device. Clicking **To Project** merges these parameters to the Component and later they can be used as a starting point for manual calibration or tuning.

5. Save the new Component settings and build the project.

Tuner Configuration Options

The Tuner application allows setting different configuration options with the Options dialog. Settings are applied on per-project basis and divided into groups:

SNR Options

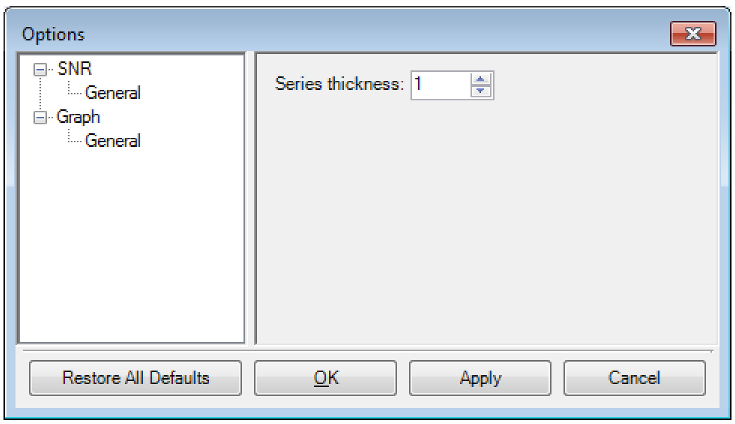


- **Noise sample count** – The count of samples to acquire during the noise measurement operation.
- **Signal sample count** – The count of samples to acquire during the signal measurement operation.
- **SNR pass value** – The minimal acceptable value of the SNR.
- **Ignore spike limit** – Ignores a specified number of the highest and the lowest spikes at noise / signal calculation. That is, if you specify number 3, then three upper and lower

three raw counts are ignored separately for the noise calculation and for the signal calculation.

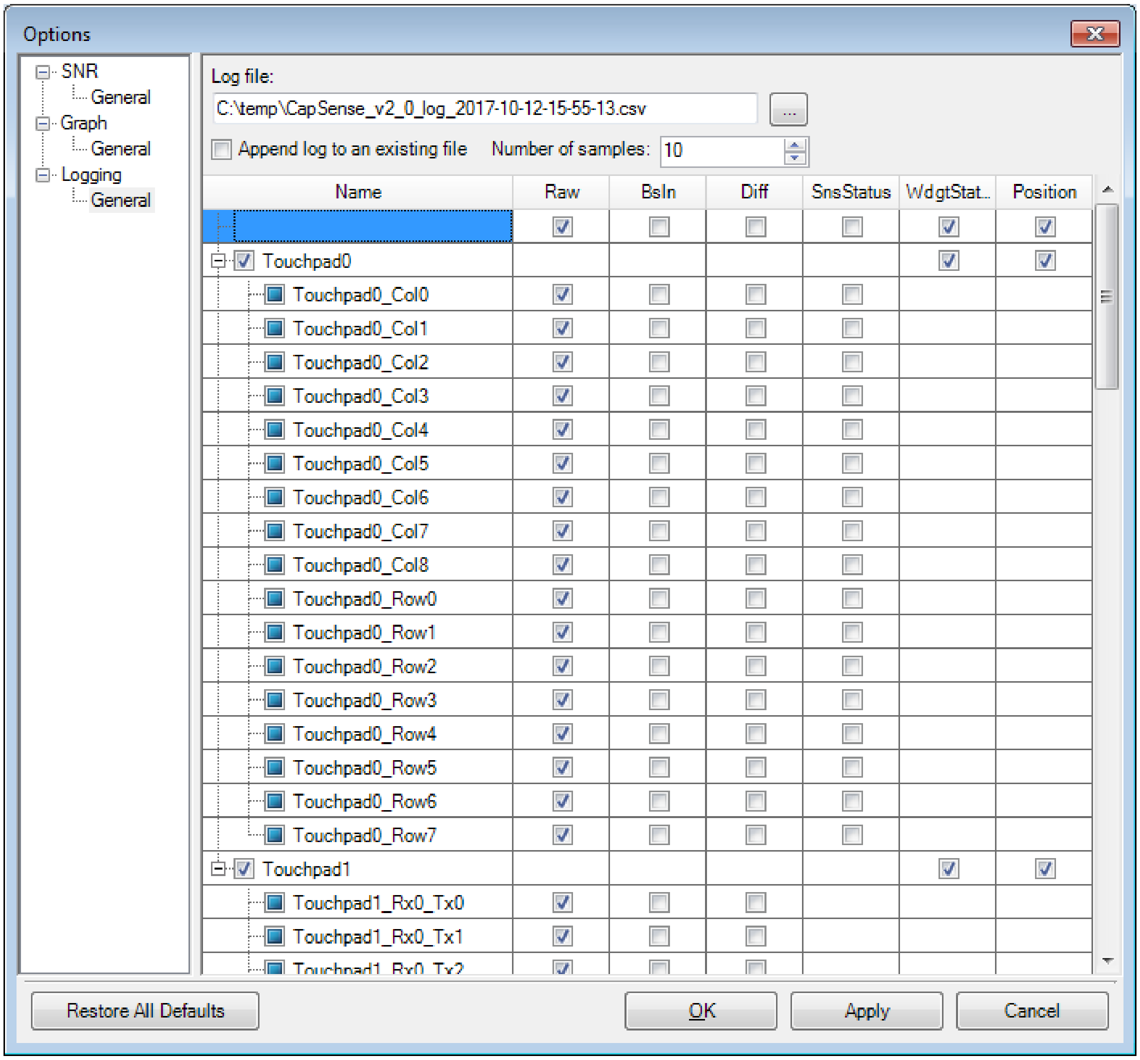
- **Noise calculation method** – Allows selecting the method to calculate the noise average. The following methods are available for selection:
  - **Peak-to-peak** (by default) – Calculates noise as a difference between the maximum and minimum value collected during the noise measurement.
  - **RMS** – Calculates noise as a root mean-square of all samples collected during the noise measurement.

Graph options



- **Series thickness** – Allows specifying the thickness of lines drawn on the graphs.

Data Log Options



- **Log File** – Selects the file for information to be stored and its location.
- **Append log to an existing file** – When checked, the selected file is never over-written and defined file is expanded with new data, otherwise it is overwritten.
- **Number of samples** – Defines a log session duration in samples.
- **Data configuration checkbox table** – Defines data that to be collected into a log file.

MISRA Compliance Report

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

- project deviations – applicable for all PSoC Creator Components
- specific deviations – applicable only for this Component

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

The CapSense_P4 Component has the following specific deviations:

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
8.8	R	An external object or function shall be declared in only one file.	Some arrays are generated based on the Component configuration and these arrays are declared locally in the .c source files where they are used instead of in .h include files.
11.4	A	A cast should not be performed between a pointer to object type and a different pointer to object type.	Pointers are used to allow many types of widgets and sensors. The architecture is designed to allow indexing a specific pointer.
12.13	A	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	These violations are reported for the GCC ARM optimized form of the “for” loop that have the following syntax: for(index = COUNT; index --> 0u;) It is used to improve performance.
13.7	R	The result of this logical operation is always 'true' (1)	This violation exists in the Gestures module only. It allows you to enable different sets of gestures. Since some of the gestures are interconnected, in some configurations, the result of the IF statement is always true.
14.2	R	All non-null statements shall either have at least one side effect however executed, or cause the control flow to change.	These violations are caused by expressions suppressing the C-compiler warnings about the unused function parameters. The CapSense Component has many different configurations. Some of them do not use specific function parameters. To avoid the complier's warning, the following code is used: (void)paramName.
16.7	A	A pointer parameter in a function prototype should be declared as the pointer to const if the pointer is not used to modify the addressed object.	Mostly all data processing for variety configuration, widgets and data types is required to pass the pointers as an argument. The architecture and design are intended for this casting.

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	Pointers are used to allow many types of widgets and sensors. The architecture is designed to allow indexing a specific pointer.
18.4	R	Unions shall not be used.	There are two general cases in the code where this rule is violated. 1. <INSTANCE_NAME>_PTR_FILTER_VARIANT definition and usage. This union is used to simplify the pointer arithmetic with the Filter History Objects. Widgets may have two kinds of Filter History: Regular History Object and Proximity History Object. The mentioned union defines three different pointers: void, RegularObjPtr, and ProximityObjPtr. 2. APIs use unions to simplify operation with pointers on the parameters. The union defines four pointers: void*, uint8*, uint16*, and uint32*. In all cases, the pointers are verified for proper alignment before usage.
19.7	A	A function should be used in preference to a function-like macro.	Simple function-like macros are used to decrease execution time in time critical functions.

This Component has the following embedded Components: PSoC 4 Current Digital to Analog Converter (IDAC_P4).

Refer to the corresponding Component *datasheet* for information on their MISRA compliance and specific deviations.

Component Debug Window

PSoC Creator allows you to view debug information about Components in your design. Each Component window lists the memory and registers for the instance. For detailed hardware registers descriptions, refer to the appropriate device technical reference manual.

Note Component debug window is available for *Fourth-generation CapSense* only.

To open the Component Debug window:

1. Make sure the debugger is running or in the break mode.
2. Choose **Windows > Components…** from the **Debug** menu.
3. In the Component Window Selector dialog, select the Component instances to view and click **OK**.

The selected Component Debug window(s) will open within the debugger framework. Refer to the "Component Debug Window" topic in the PSoC Creator Help for more information.

Resources

The CapSense Component consumes one CSD (CapSense Sigma-Delta) block, two Analog Mux buses, two IDACs and one port pin for each ADC channel, sensors, Tx and Rx electrodes configured to use a dedicated pin in the *Widget Details* tab.

Note If a design contains several components, which requires some resources (analog mux bus), and the resource utilization triggers a conflict, PSoC Creator generates a build error:

```
Unable to find a solution for the analog routing.
```

One IDAC and one analog mux bus are not consumed (and available for general purpose use) when:

- Only the ADC is configured and both CSD and CSX sensing methods are disabled.
- The Enable compensation IDAC is unselected in the *CSD Settings* tab, Shield is disabled, and ADC is disabled.

Additionally, the following may be consumed:

- UDB resources (1 macro cell) are consumed with the PSoC 4200, PSoC 4200M, PSoC 4200L and PSoC 4200 BLE device families.
- UDB resources (6 macro cell, 2 status cells and 1 control cell) are consumed only when *CSX sensing method* is used in the *Basic Tab* along with PSoC 4200 devices.
- An additional analog mux bus is consumed with a shield electrode enabled in the *CSD Settings* tab.
- One 7-bit IDAC in the CSD block is not consumed (and available for general purpose use) when the Enable compensation IDAC is unselected in the *CSD Settings* tab.

References

General References

- *Cypress Semiconductor web site*
- *PSoC 4 Device datasheets*

Application Notes

Cypress provides a number of application notes describing how PSoC can be integrated into your design. You can access them at the *Cypress Application Notes web page*. Examples that relate to CapSense include:

- *AN64846* – Getting Started with CapSense®
- *AN72362* – Reducing Radiated Emissions in Automotive CapSense® Applications
- *AN85951* – PSoC® 4 CapSense® Design Guide
- *AN92239* – Proximity Sensing with CapSense®

Code Examples

PSoC Creator provides access to code examples in the Find Code Example 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 Code Example" topic in the PSoC Creator Help for more information.

There are also numerous code examples that include schematics and code examples available online at the *Cypress Code Examples web page*. The examples that use this Component include:

- *CE210289 - PSoC®4 CapSense® Linear Slider*
- *CE210291 - PSoC® 4 CapSense® One Button*
- *CE210290 - PSoC® 4 CapSense® Low-Power Ganged Sensor*
- *CE210311 - CapSense® ADC Sequential*

Development Kit Boards

Cypress provides a number of development kits. You can access them at the *Cypress Development Kit web page*. Mentioned Code Examples uses the following development kits:

- *CY8CKIT-040 PSoC® 4000 Pioneer Kit*
- *CY8CKIT-042-BLE Bluetooth® Low Energy Pioneer Kit*
- *CY8CKIT-042 PSoC® 4 Pioneer Kit*
- *CY8CKIT-044 PSoC® 4 M-Series Pioneer Kit*
- *CY8CKIT-046 PSoC® 4 L-Series Pioneer Kit*

- *CY8CKIT-041 PSoC® 4 S-Series Pioneer Kit*

Electrical Characteristics

Specifications are valid for +25° C, VDD 3.3 V, Cmod = 2.2 nF, Csh = 10 nF, and CintA = CintB = 470 pF except where noted.

Note Final characterization data for PSoC 4100S Plus and PSoC Analog Coprocessor devices is not available at this time. Once the data is available, the Component datasheet will be updated on the Cypress web site.

Performance Characteristics

Parameter	Condition	Typical	Units
Sensor Calibration level (Applicable for sensor with highest Cp within a Widget)	Cp = 5 to 45 pF (Single IDAC mode)	85% of full scale ±5 %	-
	Cp = 5 to 45 pF (Dual IDAC mode)	85% of full scale ±10 %	-
Touch signal accuracy The touch signal is the difference between measured raw counts with and without a finger present on a sensor (difference count).		Not less than 10% of sensor sensitivity.	-
Supported Sensor Cp range		Min: 5. Max: 45	pF
SNR (Noise Floor) The simple ratio of (Signal/Noise) is called the CapSense SNR. It is usually simplified to [(Finger Signal/Noise): 1]	Cp < 35 pF Single IDAC: Finger capacitance >= 0.2 pF Dual IDAC: Finger capacitance >= 0.1 pF	> 5:1	-
	Cp < 45 pF Single IDAC: Finger capacitance >= 0.2 pF Dual IDAC: Finger capacitance >= 0.1 pF	> 4:1	-
Supply (VDD) ripple	VDD > 3.3 V, Finger capacitance = 0.1 pF, VDD ripple +/-50 mV	< 30% of noise	
	VDD < 2 V, internally regulated mode. Finger capacitance = 0.4 pF, VDD ripple +/-50 mV	< 30% of noise	
	VDD < 2 V, externally regulated mode. Finger capacitance = 0.4 pF, VDD ripple +/-25 mV	< 30% of noise	

Parameter	Condition	Typical	Units
GPIO Sink Current	10 mA per GPIO on multiple pin to sink max current. Device max = 40 mA for *Third-generation CapSense* devices. Device max = 80 mA for PSoC 4000. Device max = 25 mA for *Fourth-generation CapSense* devices.	< 30% of noise	
Tx Output Voltage	Logic High	> Vddd-0.6	V
	Logic Low	< 0.6	V
Voltage Reference (Vref) (for *Third-generation CapSense* devices, *CSD sensing method*, *CSX sensing method*)		1.2	V
Voltage Reference (Vref) (for *Fourth-generation CapSense* devices, *CSD sensing method*)	VDDA < 2.6V	1.2	V
	2.6V <= VDDA < 3.2V	1.477	V
	3.2V <= VDDA < 4.7V	2.021	V
	4.7V <= VDDA	2.743	V
Voltage Reference (Vref) (for *Fourth-generation CapSense* devices, *CSX sensing method*)		1.2	V
Finger-Conducted AC Noise Finger-Conducted AC Noise is the change in the sensor raw count when AC noise is applied on the sensor (injected into the system)	50/60 Hz, noise Vpp = 20 V	< 30%	-
	10 kHz to 1 MHz, noise Vpp = 20 V, Cp < 10 pF	< 30%	-
Interrupt immunity Excessive raw counts noise at asynchronous interrupts used.		< 30%	-
Current Consumption	1 CSD Button Widget (Ganged Sensor, 4 electrodes). Resolution = 9 bits. Each electrode Cp < 10 pF. Shield Electrode = Disabled. SYSCLK = 16 MHz. No I2C traffic (I2C block ON). Report Rate >= 8 Hz. Chip state = DeepSleep (LFT).	< 6 (PSoC 4000)	µA
		< 7 (PSoC 4000S)	µA
	1 CSD Button Widget, 8 Sensors. Resolution = 9 bits.	< 18 (PSoC 4000)	µA

Parameter	Condition	Typical	Units
	Each electrode Cp < 10 pF. Shield Electrode = Disabled. SYSCLK = 16 MHz. No I2C traffic (I2C block ON). Report Rate >= 8 Hz Chip state = DeepSleep (LFT).	< 22 (PSoC 4000S)	µA
	1 CSX Button Widget (1 x 1 electrodes). Num of sub-conversions = 25. SYSCLK = 16 MHz. Overlay >= 1 mm plastic. Button Size <= 10 mm. No I2C traffic (I2C block ON). Report Rate >= 8 Hz. Chip state = DeepSleep (LFT).	< 6 (PSoC 4000)	µA
		< 6 (PSoC 4000S)	µA
	1 CSX Touchpad Widget 32 nodes (9 x 4 electrodes). Num of sub-conversions = 25. SYSCLK = 16 MHz. Overlay => 1 mm plastic. 4.8 x 4.8 mm diamond sensors. 9 mm metal finger. 1 Touch only. Report Rate >= 8 Hz. Chip state = DeepSleep (LFT).	< 150 (PSoC 4000)	µA
		< 200 (PSoC 4000S)	µA

IDAC Characteristic

PSoC 4000S, PSoC 4100S:

Parameter	Description	Min	Typ	Max	Units	Conditions
$IDAC1_{DNL}$	DNL	-1	–	1	LSB	
$IDAC1_{INL}$	INL	-2	–	2	LSB	INL is ±5.5 LSB for VDDA < 2 V
$IDAC2_{DNL}$	DNL	-1	–	1	LSB	
$IDAC2_{INL}$	INL	-2	–	2	LSB	INL is ±5.5 LSB for VDDA < 2 V

PSoC 4100S Plus, PSoC Analog Coprocessor:

Parameter	Description	Min	Typ	Max	Units	Conditions
$IDAC1_{DNL}$	DNL	-1	–	1	LSB	
$IDAC1_{INL}$	INL	-3	–	3	LSB	
$IDAC2_{DNL}$	DNL	-1	–	1	LSB	
$IDAC2_{INL}$	INL	-3	–	3	LSB	

Third-generation CapSense devices:

Parameter	Description	Min	Typ	Max	Units	Conditions
$IDAC1_{DNL}$	DNL for 8-bit resolution	-1	–	1	LSB	
$IDAC1_{INL}$	INL for 8-bit resolution	-3	–	3	LSB	
$IDAC2_{DNL}$	DNL for 7-bit resolution	-1	–	1	LSB	
$IDAC2_{INL}$	INL for 7-bit resolution	-3	–	3	LSB	

DC/AC Specifications

Refer to device-specific datasheet *PSoC 4 Device datasheets* for more details.

Component Errata

This section lists the problems known with the CapSense P4 Component.

Cypress ID	Component Version	Problem	Workaround
248295	3.0 to 5.0	For PSoC 4000 device family the first scan after waking up from deep sleep could produce lower raw count then all the next scans.	Execute a dummy scan after waking up from the deep sleep.
215127	3.0 to 5.0	The Tuner GUI fails to work with two instances of the CapSense Component simultaneously.	Perform tuning of each instance separately.

Component Changes

This section lists the major changes in the Component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
5.0	Added Gesture, Advanced centroid, VDDA measurement to Built-in Self-test (BIST).	Expanded Component functionality.
4.10	New Component version.	Fixed the errata item 287117 for the GetExtCapCapacitance() function.
4.0.a	Edited datasheet.	Added errata item 287117 to document issue with GetExtCapCapacitance() function.
4.0	Added support for PSoC 4100S Plus device family. Renamed ExitCall**B**ack () to ExitCall**b**ack (). Improved the Component.	Fixed issues documented in the following errata items: 242894, 253147, 260781, 232921, and 259648, and removed them from the errata section.
3.10.b	Edited datasheet.	Added several errata items to document the following issues: 215127 260781 232921 259648
3.10.a	Fixed Number of Subconversions equation.	Equation was incorrect.
3.10	Added the following features: ▪ CSX Touchpad support ▪ Self-test library ▪ Multi-frequency scan feature ▪ IDAC sinking mode in Fourth generation CapSense	Expanded functionality. Fixed potential issue with Auto mode. Documented potential issue with Inactive sensor connection to shield.
3.0.b	Edited datasheet.	Added Component Errata section to document potential issue with Auto mode.
3.0.a	Removed empty CapSense_SaveConfig() and CapSense_RestoreConfig() APIs	No usage of these API is expected in future.
	Renamed CapSense_IsProximityTouchActive() to CapSense_IsProximitySensorActive() without functionality change	Providing a meaningful name and being consistent with other APIs
	Changed Sensitivity parameter to Finger Capacitance	Providing a meaningful parameter with intuitive usage
	Added IDAC sensing configuration parameter with IDAC sinking mode	Expanded functionality
	Edited datasheet.	Final characterization data for PSoC 4000S, PSoC 4100S and PSoC Analog Coprocessor devices is not available at this time. Once the data is available, the Component datasheet will be updated on the Cypress web site.

Version	Description of Changes	Reason for Changes / Impact
3.0	The initial version of new Component implementation. This version is not backward compatible with the previous versions. See *Migration Guide* for more information.	Improved implementation of the CapSense Component with PSoC 4 devices.

Migration Guide

CapSense P4 v5.X is a new Component, **not** backward-compatible with CapSense CSD P4 v2.X. So, a design that uses version 2.X requires manual migration to version v5.X to benefit from the new features and enhanced performance.

CapSense P4 is a completely new Component. Projects using CapSense CSD P4 v2.60 (and prior versions) cannot be automatically updated to the new Component. You must back up your project, replace the old Component with CapSense_P4, and set up the parameters as described below. Note that the firmware API is very different in the new Component and it is highly recommended that you read section *Step-7: API Comparison* in order to make changes in your firmware. It is highly recommended that all the new design must start with CapSense P4 v5.X, and the design that requires the features of CapSense P4 v5.X, such as mutual-cap sensing or low power, must be manually migrated. The existing designs in production or minor revisions of the existing product may use version 2.X, however, no further enhancements are planned on that version.

This section provides the guidelines migration to CapSense P4 v5.X. In general, the migration requires the following steps:

- *Step-1: Add Widget / Sensor*
- *Step-2: Parameters: Enable firmware filters*
- *Step-3: Parameters: CSD Settings*
- *Step-4: Parameters: Widgets Details*
- *Step-5: Scan Order*
- *Step-6: Pinout*
- *Step-7: API Comparison*

Differences in supported features

The table below shows the difference in the features supported by the v2.60 and v5.X Components.

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Comments
Gesture	Added in CapSense_P4 v5.X.	
Generic widget type	Planned for a future version of the Component.	Use the Button widget as a replacement.
Guard Sensor	Replaced with the Button widget.	Use the Button widget type to create a guard sensor in the design.

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Comments
Jitter filter for Raw Counts	Not supported.	The Jitter filter is supported only for slider/touchpad positions; this filter is not very effective for noise suppression in raw counts, so use IIR, Median or average filters instead.
Widget Resolution (8-bit)	Not supported.	Only the 16-bit widget resolution is supported.
Modulator clock frequency for each sensor	Not supported.	The modulator clock frequency is set for the whole Component for optimized performance.
IDAC range (8x)	Not supported.	CapSense designs do not require the 8x mode. In order to make the tuning simple, the 8x mode is removed.

Note If a device has more than one HW CSD block, different Component versions should not be used (i.e. do not place v3.0 in HW CSD block1 and v4.0 in HW CSD block2), such configuration is not guaranteed to be functional.

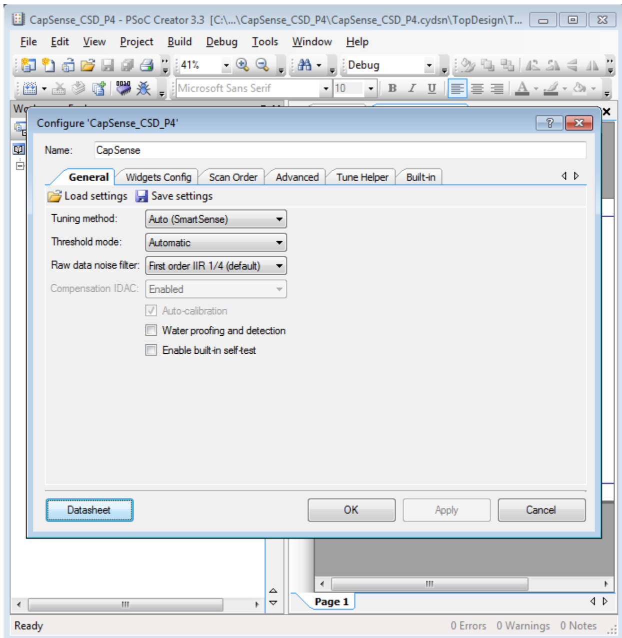
Step-1: Add Widget / Sensor

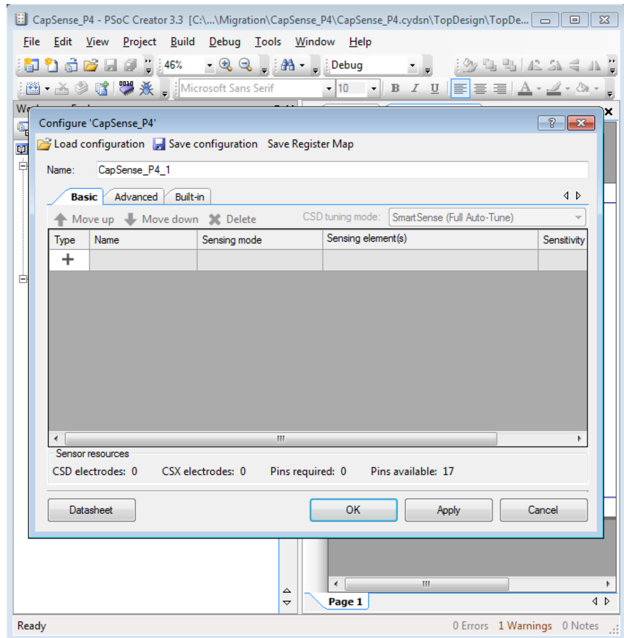
In v2.60, the **Widget Config** tab is used to create and configure the widgets, and in v5.x The the *Basic tab* is used to create widgets, and the *Widget Details* sub-tab under the *Advanced tab* is used to configure the widget's parameters.

Behavior of all widgets is the same between both versions, except for the Button and Generic widgets. In CapSense v5.X, up to 32 sensors can be created under one Button widget, configure and scan all those sensors as a group of sensors called the widget. If the design is based on v2.60 and has multiple button sensors, consider creating several widgets and multiple sensors in the widgets for optimized performance.

1. Add the widgets in the *Basic tab*, and select a number of sensors or segments of those widgets.

2. Select *CSD sensing method* for all widgets for *Sensing mode*.





3. The tuning mode selection in the new Component is updated.

   Note that there is only one selection in the new Component for the tuning mode instead of two parameters ("Tuning method" and "Threshold mode") in the old Component. The able below shows equivalent tuning modes in CapSense v5.X.

CapSense CSD P4 (v2.60)		CapSense P4 (v5.X)	Comments
Tuning Method	**Threshold Mode**	***CSD tuning mode***	
Auto (SmartSense)	Automatic	*SmartSense (Full Auto-Tune)*	Both *Widget Hardware Parameters* and *Widget Threshold Parameters* are auto-tuned. SmartSense in CapSense CSD P4 v2.60 enables the Compensation IDAC automatically. SmartSense in CapSense P4 v5.X is more flexible and allows operating with a disabled Compensation IDAC. To properly migrate, enable the *Enable compensation IDAC* parameter (enabled by default).
Auto (SmartSense)	Flexible	*SmartSense Hardware*	*Widget Hardware Parameters* are auto-tuned and *Widget Threshold Parameters* can be set by users in the customizer.
Manual with run-time tuning	N/A	*Manual*	Set both *Widget Hardware Parameters* and *Widget Threshold Parameters* manually in the customizer.
Manual	N/A		

4. If SmartSense (Full Auto-Tune) was not used, skip the following step.

   The new Component has enhanced flexibility for *Finger capacitance* selection compared to only 10 selections in v2.60 Component:

   - In the SmartSense (Full Auto-Tune) mode it is 0.1 pF to 1 pF with a 0.02 pF step.
   - In the SmartSense Hardware mode it is 0.02 pF to 20.48 pF on the exponential scale.

   The following table shows equivalent settings in the v5.X Component for 10 selections in the v2.60 Component, set the sensitivity value as indicated in table below. It is also acceptable to select a different user-set value for this parameter to benefit from the CapSense v5.X performance.

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Signal Representation
Sensitivity	*Finger capacitance*	
1	0.1	50 Counts/0.1 pF
2	0.2	50 Counts/0.2 pF
3	0.3	50 Counts/0.3 pF
4	0.4	50 Counts/0.4 pF
5	0.5	50 Counts/0.5 pF
6	0.6	50 Counts/0.6 pF
7	0.7	50 Counts/0.7 pF
8	0.8	50 Counts/0.8 pF
9	0.9	50 Counts/0.9 pF
10	1.0	50 Counts/1.0 pF

5. Compensation IDAC in v2.60: If the compensation IDAC is enabled in the Basic tab of the v2.60 Component, enable the same (*Enable compensation IDAC*) in *CSD Settings* sub-tab of v5.X.
6. Raw data filter in v2.60: If the Raw data noise filter is used in the v2.60 Component, enable the same (*Regular widget raw count filter type*) in *General* sub-tab of the new Component as described in Step-2.
7. Enable Built-in self-test in v2.60: If the self-test is enabled in the v2.60 Component, enable the same (*Enable self-test library*) in *General* sub-tab of the new Component.
8. Water proofing and detection: If water proofing and detection is enabled, enable the shield electrodes (*Enable shield electrode*) in *CSD Settings* sub-tab as described in Step-3, create a Button widget instead of guard sensors, and discard the status reported by widgets/sensors when the guard sensor is active in the application program.

Step-2: Parameters: Enable firmware filters

If the design based on the v2.60 Component met the following conditions, migration of the Component configuration is complete and go to the *Step-7* application programming interface section to continue:

- Used Auto SmartSense tuning mode
- Used Automatic threshold mode
- All firmware filters were disabled
- Water proofing and detection in v2.60 was disabled
- Sensor auto-reset was disabled

The firmware filter feature in v2.60 allows using only one filter in a design and all widget types must use common filter settings. In the CapSense P4 v5.X Component, the filter feature is enhanced by:

- Allowing coexistence of multiple filters simultaneously in a project.
- Both baseline filter and raw count filter coefficients having more configurable options.
- Both baseline filter and raw count filter for proximity and non-proximity sensors can be configured separately because the proximity filters often require different filter configuration as the proximity sensors are usually more affected by noise.

Raw count filters

The table below shows equivalent configurations between the v2.60 and v5.X Components, enable the filters and select coefficients based on the information in the table below:

CapSense CSD P4 (v2.60)	**CapSense P4 (v5.X)**	**Comments**
Raw data noise filter	***Regular widget raw count filter type***	
None	No check box selected.	
Median	Select *Enable median filter (3-sample).*	
Averaging	Select *Enable average filter (4-sample)*	Note that v5.X implements averaging of 4 samples compared to 3 samples in v2.60.
First order IIR 1/2	Select *Enable IIR filter (First order)* and set *IIR filter raw count coefficient* to 128.	Note that v5.X implements more flexibility to set a filter coefficient at 1 from 1 to 128 in steps. This table shows coefficients equivalent to the configuration in the previous Component.
First order IIR 1/4 (default)	Select *Enable IIR filter (First order)* and set *IIR filter raw count coefficient* to 64.	
First order IIR 1/8	Select *Enable IIR filter (First order)* and set *IIR filter raw count coefficient* to 32.	

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Comments
Raw data noise filter	***Regular widget raw count filter type***	
First order IIR 1/16	Select *Enable IIR filter (First order)* and set *IIR filter raw count coefficient* to 16.	
Jitter	Not supported.	The Jitter filter is supported only for the centroid positions with sliders and a touchpad as these filters don't provide any significant benefit to the sensor raw counter filtering.

Make the same for *Proximity widget raw count filter type***.**

Baseline filters:

Set the filter coefficients as listed below for equivalent performance of v2.60 from CapSense v5.X Component:

- *Regular widget baseline coefficient* = 1
- *Proximity widget baseline coefficient* = 1

Enable sensor auto-reset

- Select *Enable sensor auto-reset* if it was enabled in the previous Component.

Step-3: Parameters: CSD Settings

The *CSD Settings* sub-tab contains the parameter common to all the CSD widgets available in the **Advanced** tab of the v2.60 Component. Follow the guidelines in the table below to set the parameters in the table below.

Note The parameters requiring selection of a "frequency" are dependent on the HFCLK settings in the Clock Editor. If HFCLK is changed in the Clock Editor, these parameters may need to be set again.

Parameter in v2.60	Parameter in v5.X (***CSD Settings*** sub-tab)	Comment
Current source	*IDAC sensing configuration*	No change in the functionality between the Component versions.
IDAC range	NA	Not configurable and set to 4x in v5.X as all designs can work with a 4x range.
Analog Switch drive source	*Sense clock source*	The same selection options are available in both versions of the Component, use the same configuration from the previous Component.

Parameter in v2.60	Parameter in v5.X (*CSD Settings* sub-tab)	Comment
Individual frequency settings	*Enable common sense clock*	If an individual frequency setting was enabled in v2.60, unselect the Enable common clock in the v5.X Component.
Sense clock divider	*Sense clock frequency*	v5.X sets a clock in terms of a “frequency” instead of a divider value. If *Enable common sense clock* is selected, set *Sense clock frequency* in the *CSD Settings* tab, if not, then set *Sense clock frequency* under *Widget Details* for each widget.
Modulator clock divider	*Modulator clock frequency*	v5.X sets a clock in terms of a “frequency” instead of a divider value. Set the same modulator clock in the new Component.
Sensor auto-reset	*Enable sensor auto-reset* (In *General* tab)	No change in the functionality between the Component versions.
Widget resolution	NA	The Widget resolution is not configurable and set to 16 bits in v5.X of the Component. If the previous design used the 16 bits, no change is required. If the previous design used the 8 bits, it is automatically moved to the 16 bits in v5.X.
Negative Noise threshold	*Negative noise threshold* (In *Widget Details* tab)	No change in the functionality between the Component versions. However, v5.X sets these parameters separately for each widget.
Low baseline reset	*Low baseline reset* (In *Widget Details* tab)	
Shield	*Enable shield electrode*	No change in the functionality between the Component versions. These parameters are available in the *CSD Settings* tab only when *Enable shield electrode* is selected.
Shield signal delay	*Shield electrode delay*	
Shield tank capacitor	*Enable shield tank (Csh) capacitor*	
Pre-charge settings (shield tank capacitor)	*Csh initialization source*	
Inactive sensor connection	*Inactive sensor connection*	No change in the functionality between the Component versions.
Compensation IDAC (General tab)	*Enable compensation IDAC*	
Auto-calibration (General tab)	*Enable IDAC auto-calibration*	
Guard sensor	NA	As mentioned in *Step-1*, use the button sensor instead of the guard sensor.
NA	*Number of shield electrodes*	The number of dedicated shield electrodes was fixed to 1 in v2.60. But, v5.X allows using more than one dedicated shield electrode.

Step-4: Parameters: Widgets Details

Parameter in v2.60	Parameter in v5.X (*Widget Details* sub-tab)	Comment
Diplexing	Diplexing	No change in the feature.
API resolution	Maximum position	The parameter renamed, no change in the feature.
Row API resolution	Maximum X-axis position	
Column API resolution	Maximum Y-axis position	
Position Filter	*Position Filter*	None disables of the filters in both version of the Component. The *Median filter* and *Jitter filter* functionality has not changed, but the name is updated. For *Average filter,* v2.60 implemented 3 sample average filters and v4.X implements 2 sample average filters. The v4.X supports *IIR filter*(½), instead of IIR filter ½ and ¼ in v2.40.
Sense clock divider (Scan order tab)	*Sense clock frequency*	The v5.X Component allows setting the sense-clock frequency separately for each widget. If a dedicated sense-clock frequency is required for each sensor, create multiple widgets with one sensor each. In addition, Matrix Buttons and Touchpad widgets set separate sense clocks for rows and columns.
	Row sense clock frequency	
	Column sense clock frequency	
Scan Resolution	*Scan resolution*	The parameter behavior is the same between v2.60 and v5.X except the following changes. The v2.60 Component provides one common scan resolution for all the sensors in a widget. There is no separate scan resolution for each sensor in a button widget or no separate scan resolution for rows and columns of the matrix buttons and touchpads. As the best practice, sensors with similar electrical properties should be grouped as a widget, so that no dedicated scan resolution should be required for each sensor, row or column. If a dedicated scan Resolution is required for each sensor, create multiple widgets with one sensor each. Similarly, create two widgets for column and row sensors, but, this is not the recommended design.
Row Scan Resolution		
Column Scan Resolution		
Modulator IDAC (Scan order tab)	*Modulator IDAC*	The v5.X Component sets the modulator IDAC separately for each widget.
	Row modulator IDAC	

Parameter in v2.60	Parameter in v5.X (*Widget Details* sub-tab)	Comment
	Column modulator IDAC	If a dedicated modulator IDAC is required for each sensor, create multiple widgets with one sensor each. In addition, Matrix Buttons**Matrix Buttons** and *Touchpad* widgets set separate modulator IDAC for rows and columns.
Finger Threshold Row Finger Threshold Column Finger Threshold	*Finger threshold* (*Proximity threshold* and *Touch threshold* for Proximity widget)	The parameter behavior is the same between v2.60 and v5.X except the following changes. The v2.60 Component provides one common finger threshold for all sensors in a widget. There is no separate finger threshold for each sensor in a button widget or no separate finger threshold for rows and columns of the matrix buttons and touchpads. As the best practice, sensors with similar electrical properties should be grouped as a widget, so that no dedicated scan resolution shoud be required for each sensor, row or column. If a dedicated finger thresholds is required for each sensor, create multiple widgets with one sensor each. Similarly, create two widgets for column and row sensors, but, this is not the recommended design. For the Proximity widget, the threshold is split into two following thresholds: ▪ *Proximity threshold* to detect an approaching hand or a finger ▪ *Touch threshold* to detect a finger touch on the sensor similarly to other Widget Type sensors.
Noise Threshold Row Noise Threshold Column Noise Threshold	*Noise threshold*	The same rule as *Finger Threshold* applies.
Negative Noise Threshold (Advanced Tab)	*Negative noise threshold*	The parameter behavior is the same between v2.60 and v5.X except the following changes. The v5.X Component setting *Negative noise threshold* separately for each widget, compared to the common value for all widgets in v2.60. Follow the design guide to set values for the negative noise threshold, or set the same value for all widgets for the backward compatibility.
Low baseline reset (Advanced Tab)	*Low baseline reset*	The same rule as *Negative Noise Threshold* applies.
Hysteresis Row Hysteresis	*Hysteresis*	The parameter behavior is the same between v2.60 and v5.X except the following changes.

Parameter in v2.60	Parameter in v5.X (*Widget Details* sub-tab)	Comment
Column Hysteresis		All widgets have a dedicated hysteresis in v5.X, and it is used along with the Finger threshold for finger detection. Follow the design guide to set values for the hysteresis, or set a value to zero for the backward compatibility for the *Linear Slider*, *Radial Slider* and *Touchpad* widgets.
Debounce	*ON debounce*	The parameter behavior is the same between v2.60 and v5.X except the following changes. All widgets have a dedicated ON denounce in v5.X, and it is used along with the Finger threshold for detection finger detection. Follow the design guide to set values for the ON debounce, or set a value to zero for the backward compatibility for the *Linear Slider*, *Radial Slider* and *Touchpad* widgets.
Compensation IDAC (Scan order tab)	*Compensation IDAC value*	The behavior is the same because the v5.X Component sets the compensation IDAC separately for each sensor.
NA	*Selected pins*	Sensors in the *Button*, *Matrix Buttons* and *Proximity* widgets use a dedicated port pin for a sensor or reuses one or more pins from the existing sensors. By reusing the port pins from other sensors, ganged sensors, implementation of CSD and CSX sensing methods on the same port pins can be done.

Step-5: Scan Order

The **Scan Order** tab has no editable contents in v5.X, all the parameters available in the **Scan Order** tab of v2.60 are already configured in the other tabs on v5.X in the steps above.

Step-6: Pinout

Assign the pins in the Pin Editor; this interface is not affected by Component update.

Step-7: API Comparison

The following table lists the APIs whose functionality hasn't changed in v5.X of the Component.

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Comments
IsBusy()	IsBusy()	No major functional changes.
Sleep()	Sleep()	

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Comments
Stop	Stop()	
Wakeup()	Wakeup()	
InitializeAllBaselines()	InitializeAllBaselines()	
InitializeSensorBaseline()	InitializeSensorBaseline()	
ScanEnabledWidgets()	ScanAllWidgets()	
UpdateSensorBaseline()	UpdateSensorBaseline()	
UpdateThresholds()	ProcessSensorExt()	
UpdateWidgetBaseline()	UpdateWidgetBaseline()	

The following table shows list of API in v2.60 and its functional equivalent in v5.X Component.

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Comments
Start()	Start()	Start API in v5.X also initializes the sensor baselines and Tuner interfaces compared to v2.60.
CheckIsAnyWidgetActive()	IsAnyWidgetActive()	Name updated. These APIs return the same output, but note that the APIs in v5.X do not execute the touch detection algorithm every time an API is called, instead it returns the previously identified status by ProcessWidget() APIs.
CheckIsSensorActive()	IsSensorActive()	
CheckIsWidgetActive()	IsWidgetActive()	
GetCentroidPos()	GetCentroidPos()	
GetRadialCentroidPos()		
GetMatrixButtonPos()	IsMatrixButtonsActive()	
GetTouchCentroidPos()	GetXYCoordinates()	
GetBaselineData() GetCompensationIDAC() GetDebounce() GetDiffCountData() GetFingerHysteresis() GetFingerThreshold() GetLowBaseline () GetModulationIDAC() GetScanResolution() GetSenseClkDivider() GetModulatorClkDivider() GetNoiseThreshold() GetSensitivityCoefficient() ReadSensorRaw()	GetParam()	The APIs in v2.60 are used to read the status and output values of the parameter of the Component. In v5.X, these parameter values (or value equivalent parameter in v5.X) can be read using common APIs by passing an appropriate register address as an argument. In addition to the parameters that can be read using these APIs, v5.X provides access to many more other parameters as well as through a register map interface. The register address is defined in the RegisterMap header file. The details of the registers and bit fields of the registers are available in RegisterMap.pdf and RegisterMap.xml files by using the *Export Register Map* feature.

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Comments
SetCompensationIDAC() SetDebounce() SetFingerHysteresis() SetFingerThreshold() SetLowBaselineReset() SetModulatorClkDivider() SetNegativeNoiseThreshold() SetNoiseThreshold() SetScanResolution() SetModulationIDAC() SetSenseClkDivider() SetSensitivity() EnableWidget() DisableWidget()	SetParam()	The APIs in v2.60 are used to write values to the parameter of the Component. In v5.X, these parameter values (or a value-equivalent parameter in v5.X) can be set using common APIs by passing an appropriate register address and value as arguments. In addition to the parameters that can be read using these APIs, v5.X provides access to many more other parameters as well as through a register map interface. The register address is defined in the RegisterMap header file. The details of the registers and bit fields of the registers are available in RegisterMap.pdf and RegisterMap.xml files by using the *Export Register Map* feature.
SetUnscannedSensorState()	SetPinState()	Name updated. v5.X function additionally supports CSX widgets.
InitializeEnabledBaselines()	InitializeAllBaselines() InitializeWidgetBaseline() InitializeSensorBaseline()	The baselines are initialized in Start() API, so this API is discontinued. But, the same functionality can be achieved using one of the three APIs available in v5.X.
UpdateBaselineNoThreshold()	ProcessWidgetExt() ProcessSensorExt()	This API is discontinued, but, the same functionality can be implemented using one of the listed APIs from v5.X
UpdateEnabledBaselines()	ProcessAllWidgets() ProcessWidget() ProcessWidgetExt() ProcessSensorExt() UpdateSensorBaseline() UpdateAllBaselines() UpdateWidgetBaseline()	This API is discontinued, but, the same functionality can be implemented using one of the listed APIs from v5.X.
DisableRawDataFilters() EnableRawDataFilters()	ProcessWidgetExt() ProcessSensorExt()	These v2.60 APIs are discontinued and the filter is enabled in the firmware if it is enabled in the customizer and executed part of the Process Widget APIs. If required to avoid execution of the filter, even if it is enabled in the customizer, use the one of the listed API from v5.X.
ScanSensor()	CSDSetupWidgetExt() CSDScanExt()	This v2.60 API is discounted, but, the same functionality can be achieved using the two APIs in

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Comments
ScanWidget()	CSDSetupWidget() CSDScan()	v5.X (both APIs are needed to implement the functionality). Refer to the code examples, design guides (*References*) or API description (*Application Programming Interface*) to learn how to optimize the system performance using these APIs.
DisableSensor()	CSDDisconnectSns()	Both APIs disconnect the sensor port pin and are set to an inactive state.
EnableSensor()	CSDConnectSns()	Both APIs connect to a sensor port pin AMUX and the sensor is ready for scan.
ClearSensors()	CSDDisconnectSns()	v2.60 API disconnects all the sensors. Call the v5.X API in a loop to disconnect all the sensors for functional equivalence.
Enable()	NA	The Component is enabled and the Tuner and the Component are initialized by the Start API, so this API is discontinued without functional impact.
Init()	NA	The Component is enabled and the Tuner and the Component are initialized by the Start API, so this API is discontinued without functional impact.
TunerStart()	NA	The Component is enabled and the Tuner and the Component are initialized by the Start API, so this API is discontinued without functional impact.
SetScanSlotSettings()	CSDSetupWidget()	The v2.60 API loads the settings for scanning a sensor. v5.X API loads the common parameters for all sensors in the widget.
MeasureCmod()	GetExtCapCapacitance()	Name updated. These APIs return the same output.
MeasureCShieldTank()	GetExtCapCapacitance()	Name updated. These APIs return the same output.
MeasureCShield()	GetShieldCapacitance()	Name updated. These APIs return the same output.
GetSensorCp()	GetSensorCapacitance()	Name updated. These APIs return the same output.
DecodeAllGestures()	DecodeWidgetGestures()	Name updated. v5.X function has only widgetId parameter.

The following table shows the list of discontinued APIs and feature related these APIs are not available in the Component.

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Comments
GetDiffDoubleCentroidPos()	NA	Not supported
GetDoubleTouchCentroidPos()	NA	Not supported
GetScrollCnt()	NA	Not supported
GetWidgetNumber()	NA	Not supported
GetNormalizedDiffCountData()	NA	Not supported
GetNoiseEnvelope()	NA	Not supported
ReadCurrentScanningSensor()	NA	Not supported
GetIDACRange()	NA	Not supported

CapSense CSD P4 (v2.60)	CapSense P4 (v5.X)	Comments
SetIDACRange()		
SetDriveModeAllPins()		
RestoreDriveModeAllPins()		
SaveConfig()		
RestoreConfig()		
SetBaselineData()		
SetDiffCountData()		
WriteSensorRaw()		