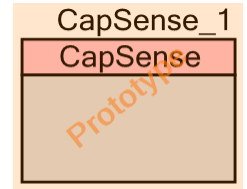


PSoC 6 Capacitive Sensing (CapSense®)

1.0

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
- Supports single-slope ADC



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_ADC 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, APIs 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.

PRELIMINARY

- *Application Programming Interface* – Provides descriptions of all APIs in the firmware library, as well as descriptions of all data structures (Register map) used by the firmware library.
- *CapSense Tuner* – Contains descriptions of all user-interface controls in the tuner application.
- *DC and AC Electrical Characteristics* – Provides the Component performance specifications and other details such as certification specifications.

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 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 are that it provides robust performance in harsh environmental conditions and rejects a wide range of external noise sources.

Use CapSense for following:

- 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 6 family of devices. However, some features are restricted in specific devices:

- The *CapSense Tuner* does not support ADC functionality. This feature will be added in a future Component version.
- The built-in self-test (BIST) library is not available and will be added in a future Component version.

PRELIMINARY



- The Multi-frequency scanning is not available and will be added in a future Component version.

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 Mode](#). In order to monitor performance of the sensor using the [CapSense Tuner](#), refer to the [Tuning Quick Start with EzI2C](#) section once the basic Linear Slider project has been created. Note that the CY8CKIT-062-BLE *PSoC® 6 Pioneer Kit* with PSoC 6 devices has a linear slider built in.

As needed, refer to the following documents for more information about PSoC Creator available from the Help menu):

- [Quick Start Guide](#)
- [PSoC Creator Help](#)

Step-1: Create a Design in PSoC Creator

Create a project using PSoC Creator and select the desired CapSense-enabled PSoC 6 device from the drop-down menu in the New Project wizard.

Step-2: Place and Configure the 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.

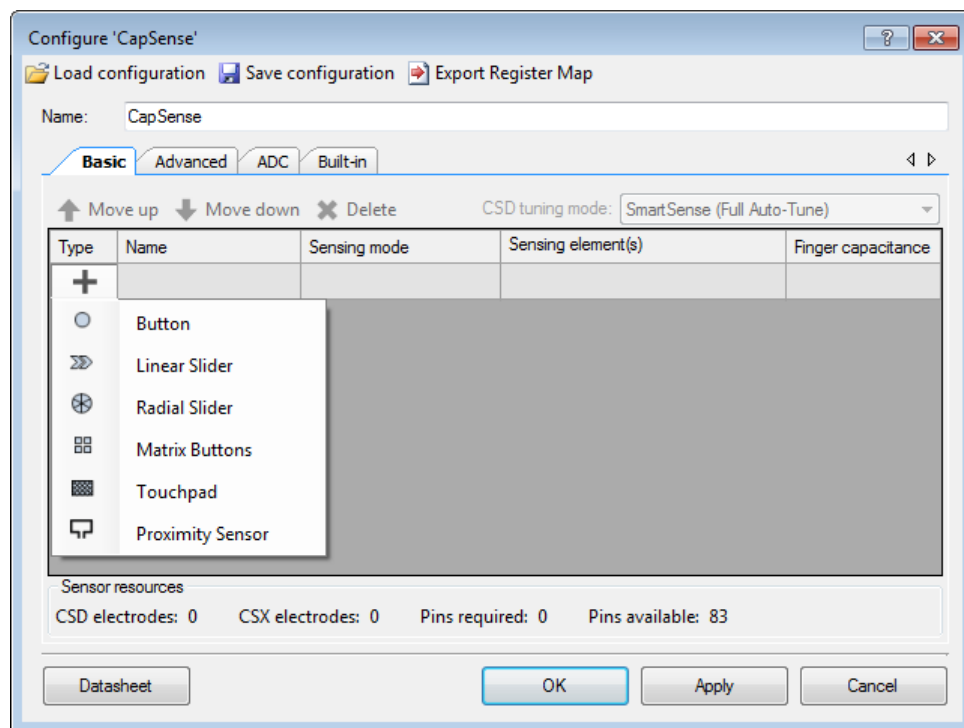
The [Component Configuration Parameters](#) are arranged over the multiple tabs and sub-tabs.

Basic tab

Use this tab to select the [Widget Type](#), [Sensing Mode](#), and a number of [Widget Sensing Element\(s\)](#) required for the design.

Type the desired Component name (in this case: *CapSense* for the code in [Step-3](#) to work).

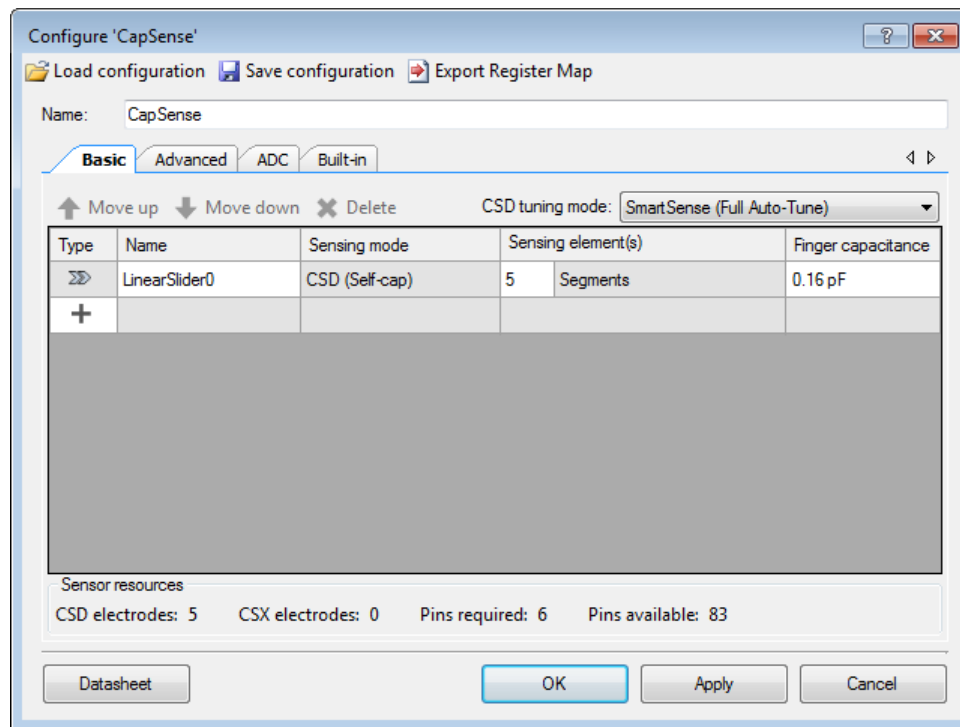
Click '+' and select the *Widget Type* required from the drop-down list. This Component offers six different types of widgets.



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 available* in the selected device to successfully build a project.

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 a 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 to meet the application requirement (In this example, default values 5 is used).
- *Finger capacitance* – Selects Finger capacitance between 0.1pF and 1pF in the *SmartSense (Full Auto-Tune)* tuning mode and between 0.02pF to 20.48pF in the *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 Auto-tuning* mode is enabled.

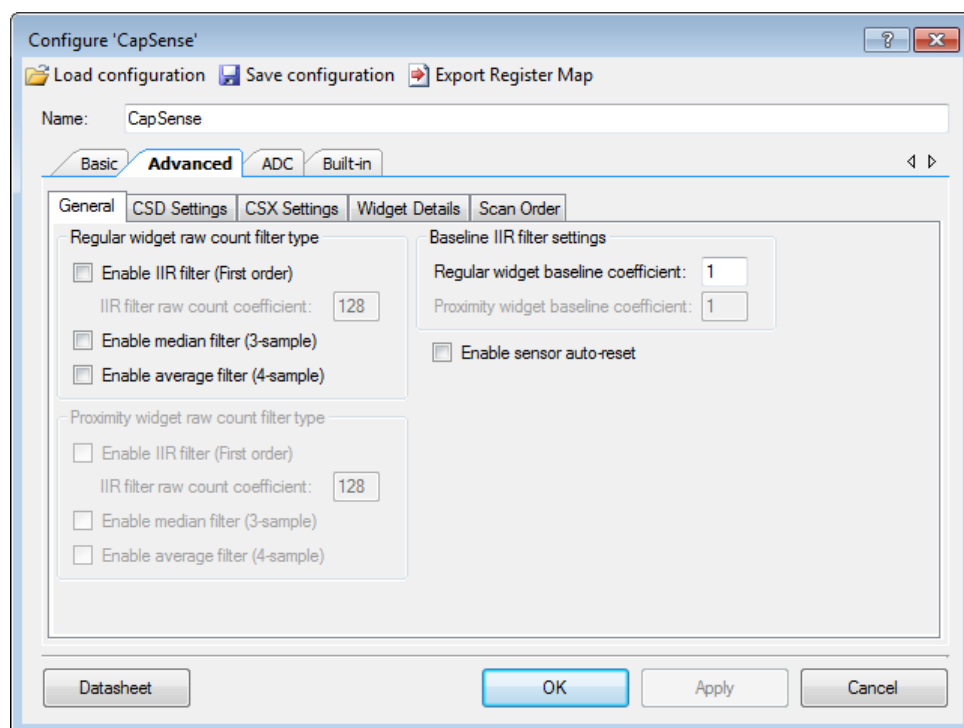
Use the *CSD tuning mode* pull-down (above the table) to select one of the following options:

- *SmartSense (Full Auto-Tune)* – With the 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 Mode* only. Widgets using the CSX mode must be configured manually. This example uses the *SmartSense (Full Auto-Tune)* tuning mode.

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.



- *General* – This sub-tab contains the parameters common for all widgets in the Component.
- *CSD Settings* – This sub-tab contains all the parameters common for all CSD widgets.
- *CSX Settings* – This sub-tab contains all the parameters common for all CSX widgets.

PRELIMINARY



- **Widget Details** – This sub-tab contains all the parameters specific for each widget and sensing element.
- **Scan Order** – This sub-tab has no editable content and provides the scan time for sensors.

Step-3: Write Application Code

Copy the following code into *main_cm4.c* file:

```
#include <project.h>

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

    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 */
            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-4: Assign Pins in Pin Editor

Double-click the Design-Wide Resources Pin Editor (in the Workspace Explorer) and assign physical pins for all CapSense sensors. If you are using a Cypress kit, refer to the kit user guide for pin selections for that hardware.

Step-5: Build Design and Program PSoC Device

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

Input / Output Connections

This section describes the various input and output connections for the CapSense Component. These are not exposed as connectable terminals on the Component symbol but these terminals can be assigned to the port pins in the **Pins** tab of the Design-Wide Resources setting of PSoC Creator. The **Pin Editor** provides the 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 _{intA} ^[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 _{intB} ^[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 (C _p) 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 is based on the user selection of the CSD widgets.
Tx	Digital Output	Transmitter electrodes of CSX widgets. The number of sensors is based on the user selection of the CSX widgets.
Rx	Analog	Receiver electrodes of CSX widgets. The number of sensors is based on the user selection of the CSX widgets.
AdcInput	Analog	ADC voltage inputs. The number of inputs is set by the Component parameter.

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

² The restricted placement rules apply dependent on devices used. Refer to the device datasheet or PSoC Creator Pin Editor.

Component Configuration Parameters

This section provides a brief description of all 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 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 can be used 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)* and *Finger capacitance* for each widget.

Configure 'CapSense'

Load configuration Save configuration Export Register Map

Name: CapSense

Basic Advanced ADC Built-in

Move up Move down Delete CSD tuning mode: SmartSense (Full Auto-Tune)

Type	Name	Sensing mode	Sensing element(s)			Finger capacitance
○	Button0	CSD (Self-cap)	1	Button(s)		0.16 pF
↔	LinearSlider0	CSD (Self-cap)	5	Segments		0.16 pF
⊗	RadialSlider0	CSD (Self-cap)	5	Segments		0.16 pF
⌘	MatrixButtons0	CSD (Self-cap)	2	Columns	2 Rows	0.16 pF
⌘	Touchpad0	CSD (Self-cap)	6	Columns	6 Rows	0.16 pF
📶	Proximity0	CSD (Self-cap)	1	Proximity Sensor(s)		0.16 pF
○	Button1	CSX (Mutual-cap)	1	Rx	1 Tx	N/A
⌘	MatrixButtons1	CSX (Mutual-cap)	2	Rx	2 Tx	N/A
⌘	Touchpad1	CSX (Mutual-cap)	6	Rx	6 Tx	N/A
+						

Sensor resources
CSD electrodes: 28 CSX electrodes: 18 Pins required: 49 Pins available: 83

Datasheet OK Apply Cancel

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

Name	Description
CSD tuning mode	<p>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.</p> <p>The 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.</p> <p>Configurable parameters that affect the operation of the sensing hardware are called Hardware parameters and parameters that affect the operation of the touch detection firmware algorithm are called Threshold parameters.</p> <p>This parameter is a drop-down box to select the tuning mode for CSD widgets only.</p> <ul style="list-style-type: none"> ▪ SmartSense (Full Auto-Tune) – This is the quickest way to tune a design. Mostly all the parameters (hardware and threshold parameters) are automatically tuned by the Component and Customizer GUI displays them as the <i>Set by SmartSense</i> mode. In this mode, the following parameters are automatically tuned: <ul style="list-style-type: none"> ○ The <i>CSD Settings</i> tab: <i>Enable common Sense clock</i>, <i>Enable IDAC auto-calibration</i> and <i>Sense clock frequency</i> ○ The <i>Widget Details</i> tab: All the CSD-related parameters of the <i>Widget Hardware Parameters</i> and <i>Widget Threshold Parameters</i> groups. ○ The <i>Widget Details</i> tab: the <i>Compensation IDAC value</i> parameter if <i>Enable compensation IDAC</i> is set. ▪ SmartSense (Hardware parameters only) – the Hardware parameters are automatically set by the Component, all the Threshold parameters can be manually set by the user. This mode consumes less memory and less CPU processing time, therefore leads to lower average power. In this mode, the following parameters are automatically tuned: <ul style="list-style-type: none"> ○ The <i>CSD Settings</i> tab: <i>Enable common Sense clock</i>, <i>Enable IDAC auto-calibration</i> and <i>Sense clock frequency</i> ○ The <i>Widget Details</i> tab: All the CSD-related parameters of <i>Widget Hardware Parameters</i> group. ○ The <i>Widget Details</i> tab: <i>Compensation IDAC value</i> parameter if <i>Enable compensation IDAC</i> is set. ▪ Manual – SmartSense auto-tuning is disabled, all the <i>Widget Hardware Parameters</i> and <i>Widget Threshold Parameters</i> must be manually tuned. The lowest memory and CPU process time consumption. <p>The SmartSense Auto-tuning (both Full Auto-Tune and Hardware parameters only) supports the <i>IDAC Sourcing</i> configuration only.</p> <p><i>SmartSense Auto-tuning</i> requires <i>Modulator clock frequency</i> is set to 6000 kHz or higher.</p>

Name	Description
Widget Type	<p>A widget is one or a group of sensors that perform a specific user-interface functionality.</p> <ul style="list-style-type: none"> ▪ Button is a widget consisting of 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 is a widget consisting of more than one sensor arranged in a specific fashion 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 is a widget consisting of more than one sensor arranged in a circular fashion 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 is a widget consisting of two or more sensors, each arranged in a specific horizontal and vertical order to detect the presence or absence of a finger on 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 is a widget consisting of multiple sensors arranged in a 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 is a widget consisting of 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: <ul style="list-style-type: none"> ○ Proximity threshold: To detect an approaching hand or finger. ○ Touch threshold: To detect a finger touch on the sensor.
Widget Name	<p>A widget name can be defined to aid in referring to a specific widget in the design. A widget name does not have any effect on functionality or performance and a widget name is used throughout the 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.</p>
Sensing Mode	<p>The parameter to select the sensing mode for each widget:</p> <ul style="list-style-type: none"> ▪ CSD sensing method (Capacitive Sigma Delta) is a Cypress patented method of performing self-capacitance measurements. All widget types support CSD sensing. ▪ CSX sensing method is a Cypress patented method of performing mutual-capacitance measurements; only buttons, matrix buttons, and touchpad widgets support CSX sensing.

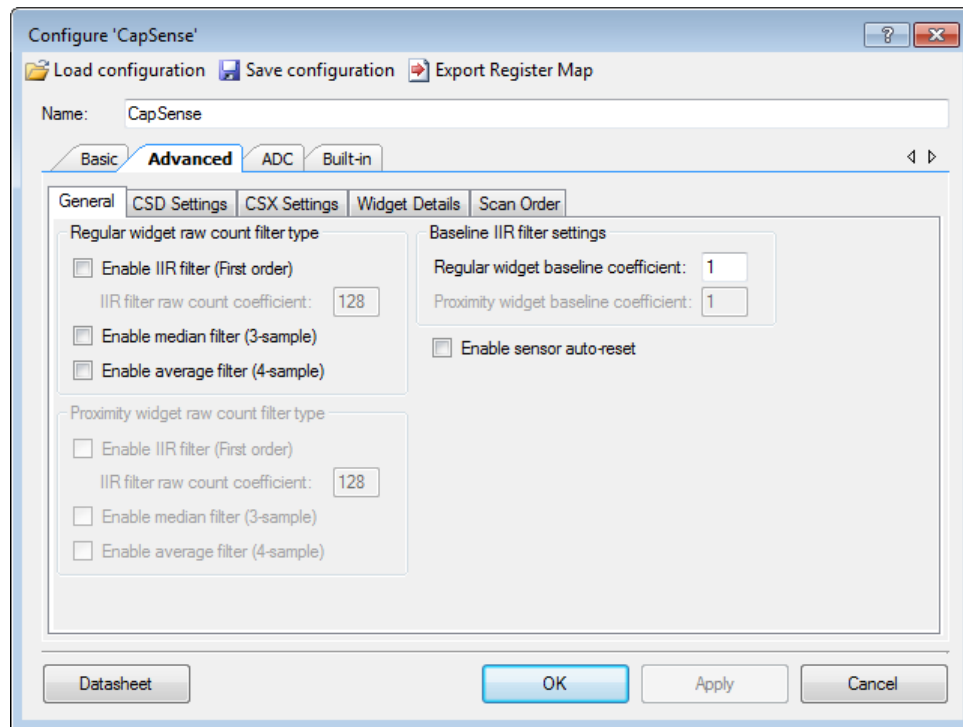
PRELIMINARY



Name	Description
Widget Sensing Element(s)	<p>The sensing element refers to Component terminals assigned to port pins to connect to physical sensors on the user-interface panel (such as a pad or layer on a PCB, ITO, or FPCB).</p> <p>The following element numbers are supported by the <i>CSD sensing method</i>:</p> <ul style="list-style-type: none"> ▪ <i>Button</i> supports 1 to 32 sensors within a widget. ▪ <i>Linear Slider</i> supports 3 to 32 segments within a widget. ▪ <i>Radial Slider</i> supports 3 to 32 segments within a widget. ▪ <i>Matrix Buttons</i> 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. ▪ <i>Touchpad</i> supports 3 to 16 rows and columns. ▪ <i>Proximity</i> supports 1 to 16 sensors within a widget. <p>The following element numbers are supported by the <i>CSX sensing method</i>:</p> <ul style="list-style-type: none"> ▪ <i>Button</i> – 1 to 32 Rx electrodes (for 1 to 32 sensors) and Tx is fixed to 1. ▪ <i>Matrix Buttons</i> – 2 to 16 Tx and Rx. The total intersections (node) number is equal to Tx x Rx, limited to the maximum of 32. ▪ <i>Touchpad</i> supports 3 to 16 Tx and Rx. The total intersections (node) number is equal to Tx x Rx. The maximum number of nodes is 256.
Finger capacitance	<p>Finger capacitance is defined as capacitance introduced by the producing signal not less than 50 counts. This parameter is used to indicate how a sensitive CSD widget should be tuned by the <i>SmartSense Auto-tuning</i> algorithm.</p> <p>The supported Finger capacitance range:</p> <ul style="list-style-type: none"> ▪ In the <i>SmartSense (Full Auto-Tune)</i> mode it is 0.1 pF to 1 pF with a 0.02-pF step. ▪ In the <i>SmartSense (Hardware parameters only)</i> mode it is 0.02 pF to 20.48 pF on the exponential scale.
Move up / Move down	<p>Moves the selected widget up or down by one on the list. It defines the widget scanning order.</p> <p>Note Widget deleting may break a pin assignment and you will need to repair the assignment in the Pin Editor.</p>
Delete	<p>Deletes the selected widget from the list.</p> <p>Note Widget deleting may break a pin assignment and you will need to repair the assignment in the Pin Editor.</p>
CSD electrodes	<p>Information: Indicates the total number of electrodes (port pins) used by the CSD widgets, including the <i>Cmod</i>, <i>Csh</i> and <i>Shield</i> electrodes.</p>
CSX electrodes	<p>Information: Indicates the total number of electrodes (port pins) used by the CSX widgets, including the <i>CintA</i> and <i>CintB</i> capacitors.</p>
Pins required	<p>Information: 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 the debug mode. The number of Pins required should always be less than or equal to that of <i>Pins available</i> for a project to build successfully.</p>
Pins available	<p>Information: Indicates the total number of port pins available for the selected device.</p>

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 by the *Manual* tuning process. When the manual tuning mode is selected, the **Advanced** tab allows the user to control and configure all the Component parameters.



The parameters in the **Advanced** tab are systematically arranged in the five sub-tabs.

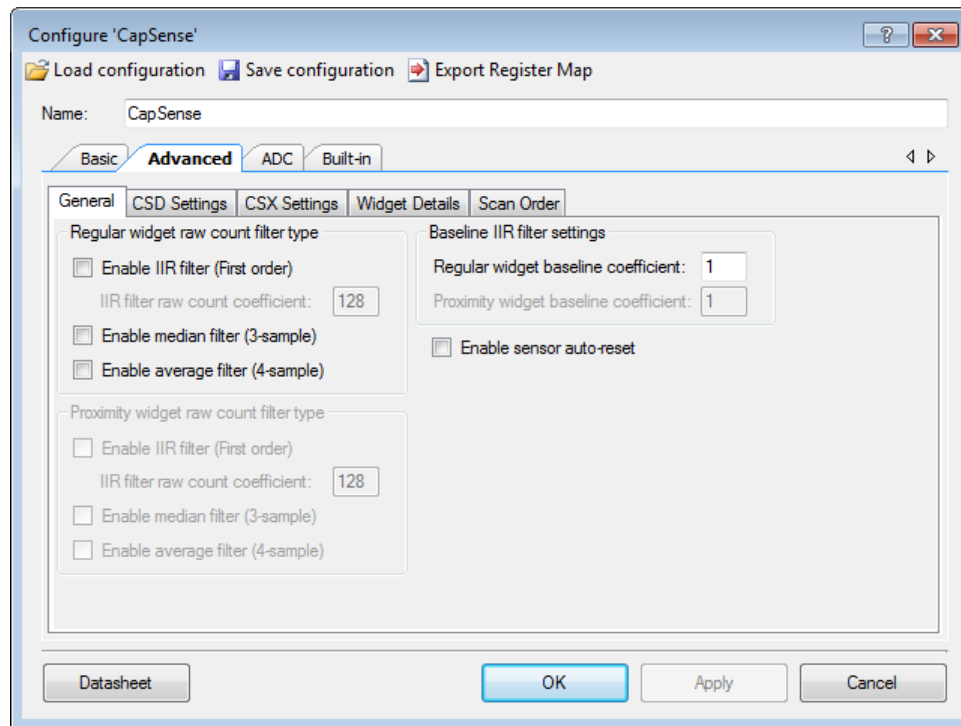
- **General** – Contains all parameters common for all widgets irrespective of the sensing method used for the widgets.
- **CSD Settings** – Contains all parameters common for all widgets using the CSD sensing method. This tab is relevant only if one or more widget use the CSD sensing method.
- **CSX Settings** – Contains all 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.

PRELIMINARY



General Sub-tab

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



This **Sub-tab** contains 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. So these parameters can be enabled only when one or more non-proximity widgets are added on the **Basic** tab. The filter algorithm is executed when any processing API 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 is correspondingly increased 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)	<p>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).</p> $Output = \frac{N}{K} \times input + \frac{(K - N)}{K} \times previous \quad Output$ <p>Where: <i>K</i> is always 256, <i>N</i> is the IIR filter raw count coefficient selectable from 1 to 128 in the customizer. A lower <i>N</i> (set in 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).</p>
IIR filter raw count coefficient	<p>The coefficient (<i>N</i>) of IIR filter for raw counts as explained in the Enable IIR filter (First order) parameter. The range of valid values: 1-128.</p>
Enable median filter (3-sample)	<p>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).</p>
Enable average filter (4-sample)	<p>The finite-impulse response filter (no feedback) with equally weighted coefficients. It takes four of most recent samples and computes their average. Eliminates the periodic noise (e.g. noise from AC mains). Consumes 6 bytes of RAM per each sensor to store a previous raw count (filter history).</p>

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

- Median filter
- IIR filter
- Average filter

However, the Component provides the ability to change the order using a low-level processing API. Refer to [Application Programming Interface](#) for details.

Proximity widget raw count filter type

The proximity widget raw count filter applies to raw counts of sensors belonging to the proximity widgets, so 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 <i>Regular widget raw count filter type</i> parameters. The <i>Proximity</i> 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, the 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. However, the filter coefficients can be separate for both baseline filter and raw count filters to produce a different roll-off. The baseline filter is applied to the 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 <i>Regular widget baseline</i> coefficient, but with a dedicated parameter allows controlling the baseline update-rate of the proximity sensors differently compared to other widgets.

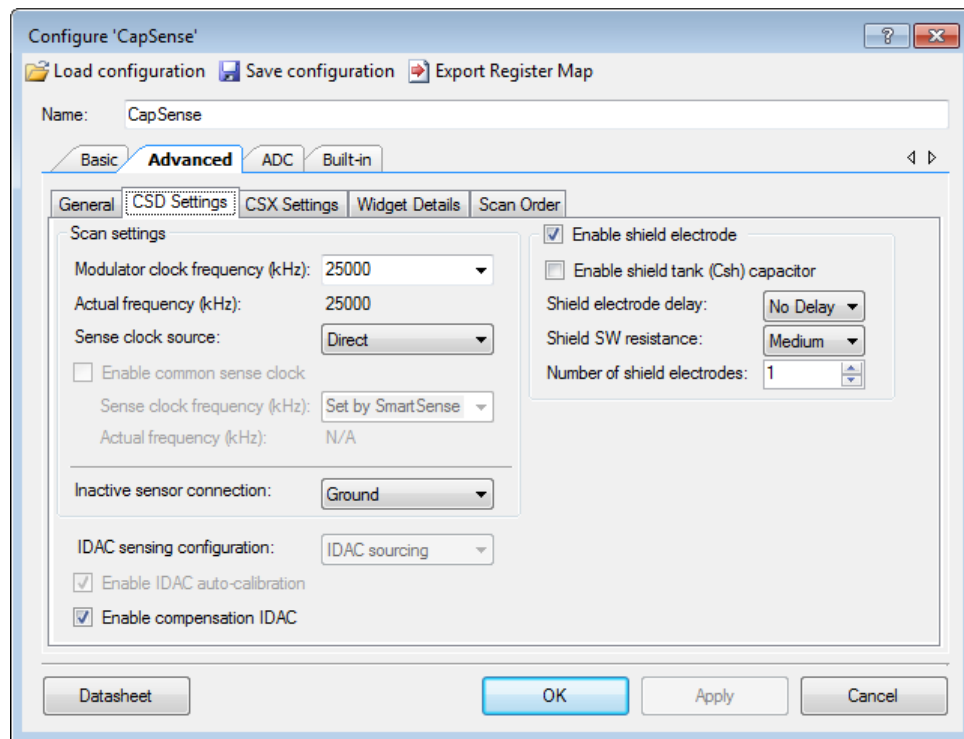
General settings

The general settings are applicable to the whole Component behavior.

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

CSD Settings Sub-tab

Contains all parameters common for all widgets using the *CSD sensing method*. This **Sub-tab** is relevant only if one or more widgets use the CSD sensing method.



This **Sub-tab** contains the following parameters:

Name	Description
Modulator clock frequency	<p>Selects the modulator clock frequency used for the <i>CSD sensing method</i>. The minimum value is 1000 kHz. The maximum value is 50000 kHz or HFCLK, whichever is lower.</p> <p>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.</p> <p>The default value is the highest modulator clock. A higher modulator clock-frequency reduces the sensor scan time, therefore, results in lower power and reduces the noise in the raw counts, so it is recommended to use the highest possible frequency.</p> <p><i>SmartSense Auto-tuning</i> requires <i>Modulator clock frequency</i> is set to 6000 kHz or higher.</p>

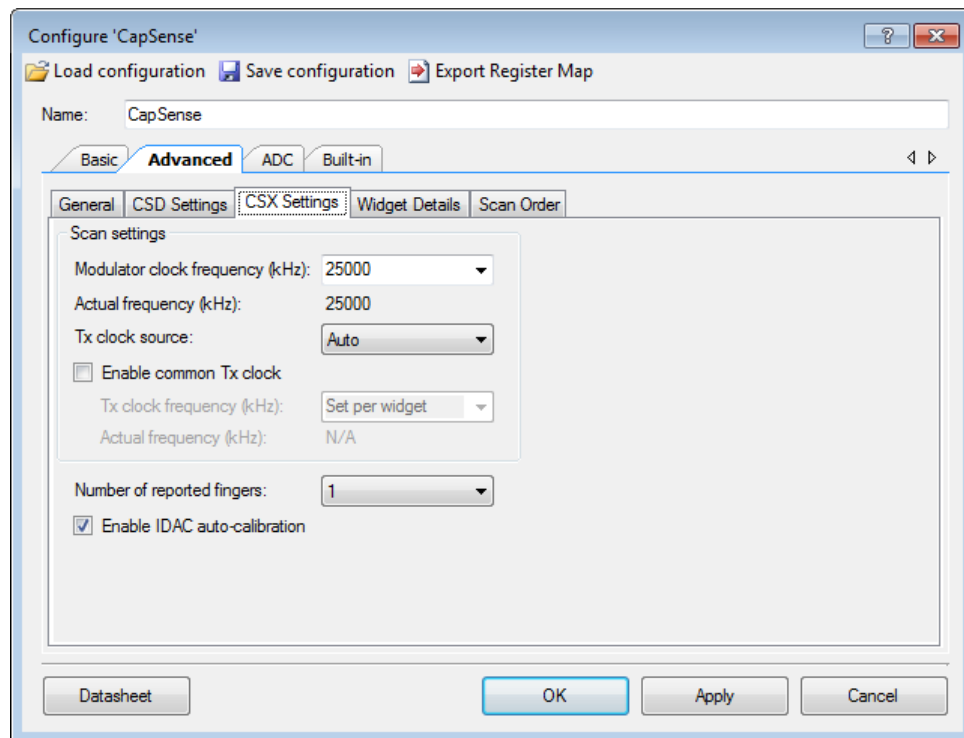
Name	Description
Sense clock source	<p><i>Sense clock frequency</i> is used to sample the input sensor. Both the type of the clock source and the clock frequency are configurable. The Spread Spectrum Clock (SSC) provides a dithering clock source with a center frequency equal to the frequency set in the <i>Sense clock frequency</i> parameter, PRS clock source spreads the clock using pseudo-random sequencer and 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, increasing the number of bits, lowering the radiation and increasing the immunity against external noise.</p> <p><i>Sense clock frequency</i> is derived from <i>Modulator clock frequency</i> using a Sense clock-divider. The following sources are available:</p> <ul style="list-style-type: none"> ▪ <i>Direct</i> – PRS and SSC are disabled and a fixed clock is used. ▪ <i>PRS8</i> – The clock spreads using PRS to Modulator Clock / 256. ▪ <i>PRS12</i> – The clock spreads using PRS to Modulator Clock / 4096. ▪ <i>SSC6, SSC7, SSC9 and SSC10</i> – The clock spreads using from 6 bits to 10 bits of the sense-clock divider respectively. ▪ <i>Auto</i> – The Component automatically selects the optimal SSC, PRS or Direct sources individually for each widget. <p>The Auto is the recommended sense-clock source selection.</p> <p>The following rules should be adhered at SSC selection:</p> <ul style="list-style-type: none"> ▪ The ratio between <i>Modulator clock frequency</i> and <i>Sense clock frequency</i> should be not less than 20. ▪ At least one full-spread spectrum polynomial should finish during scan time. ▪ SSC should be selected so that 2^{SSCn} should be less than or equal to 10% of the ratio between <i>Modulator clock frequency</i> and <i>Sense clock frequency</i>. ▪ The number of conversions in a sample should be an integer number of the repeat period of the SSC. For example, for SSC6, $(2^{SSC6}-1)*N = 63*N$ conversions should be included in a sample. <p>The following rules should be adhered at PRS selection:</p> <ul style="list-style-type: none"> ▪ At least one full PRS polynomial should finish during scan time.
Enable common Sense clock	<p>When selected, all CSD widgets share the same sense clock at a frequency specified in the <i>Sense clock frequency (kHz)</i> parameter, otherwise <i>Sense clock frequency</i> can be entered separately for each CSD widget in the <i>Widget Details</i> tab.</p> <p>Using a common sense clock for all CSD widgets results in lower power consumption and optimized memory usage, but, if the sensor parasitic capacitance is significantly different for each widget, then a common Sense clock may not produce the optimal performance.</p> <p>To enable <i>SmartSense Auto-tuning</i>, this parameter should be unselected because SmartSense will set a Sense clock for each widget based on the sensor properties for the optimal performance.</p>

Name	Description
Sense clock frequency	<p>Sets the CSD Sense clock frequency. The minimum value is 45 kHz. The maximum value is 6000 kHz or <i>Modulator clock frequency</i>/4, whichever is lower.</p> <p>Enter any value between the min and max limits, based on 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.</p> <p>When SmartSense is selected in <i>CSD tuning mode</i>, the Sense Clock frequency is automatically set by the Component to the optimal value by following the $(2 \times 5 \times R \times C)$ rule and this control is grayed out.</p> <p>When <i>Enable common Sense clock</i> is unselected, the Sense clock frequency can be set individually for each widget in the <i>Widget Details</i> tab, and this control is grayed out.</p> <p>Note If the HFCLK or <i>Modulator clock frequency</i> is changed, the Component automatically recalculates a next closest Sense clock frequency value to a possible one.</p>
Inactive sensor connection	<p>Selects the state of the sensor when not being scanned.</p> <ul style="list-style-type: none"> ▪ Ground (default) – All inactive sensors are connected to ground. ▪ High-Z – All inactive sensors are floating (not connected to GND or Shield). ▪ Shield - All inactive sensors are connected to Shield. This option is available only if the <i>Enable shield electrode</i> check box is set. <p>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 sensor parasitic capacitance reduction in a design.</p>
IDAC sensing configuration	<p>Selects the type of IDAC switching:</p> <ul style="list-style-type: none"> ▪ IDAC Sourcing (default) – Sources current into the modulator capacitor (<i>Cmod</i>). The analog switches are configured to alternate between the <i>Cmod</i> 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 (<i>Cmod</i>). The analog switches are configured to alternate between V_{DD} and <i>Cmod</i>. The IDAC sinking provides better robustness to some tests.
Enable IDAC auto-calibration	<p>When enabled, values of the CSD widget IDACs are automatically set by the Component. It is recommended to select the Enable IDAC auto-calibration parameter for robust operation. The <i>SmartSense Auto-tuning</i> parameter can be enabled only when Enable IDAC auto-calibration is selected.</p>
Enable compensation IDAC	<p>Enabling the compensation IDAC is recommended unless one IDAC is required for other purpose in the project. The compensation IDAC is used to compensate for parasitic capacitance of the sensor to improve the performance.</p>
Enable shield electrode	<p>The shield electrode is used to reduce the sensor parasitic capacitance, enable water-tolerant CapSense designs and enhance the detection range for the <i>Proximity</i> sensors. When the shield electrode is disabled, all configurable parameters associated with the shield electrode are hidden.</p>
Enable shield tank (Csh) capacitor	<p>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.</p> <p>The Shield tank capacitor is not supported in a configuration which includes both CSD- and CSX-sensing based widgets.</p>

Name	Description
Shield electrode delay	<p>Configures the delay between the sensor signal and shield electrode signal for phase alignment. The following options are available for selection:</p> <ul style="list-style-type: none"> ▪ No Delay (default) ▪ 5 ns ▪ 10 ns ▪ 20 ns <p>Most designs work with the No Delay option and it is the recommended value.</p>
Shield SW resistance	<p>Selects the resistance of switches used to drive the shield electrode when an internal shield drive is used. The four options are available:</p> <ul style="list-style-type: none"> ▪ Low (400 Ohm, typ) ▪ Medium (default) (1300 Ohm, typ) ▪ High (11000 Ohm, typ) ▪ Low EMI (600 Ohm, typ)
Number of shield electrodes	<p>Select the number of shield electrodes required in the design.</p> <p>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.</p> <p>The minimum value is 0 (i.e. shield signal could be routed to sensors using the <i>Inactive sensor connection</i> parameter) and the maximum value is equal to the total number of CapSense-enabled port pins available for the selected device.</p>

CSX Settings Sub-tab

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



This sub-tab contains the following parameters:

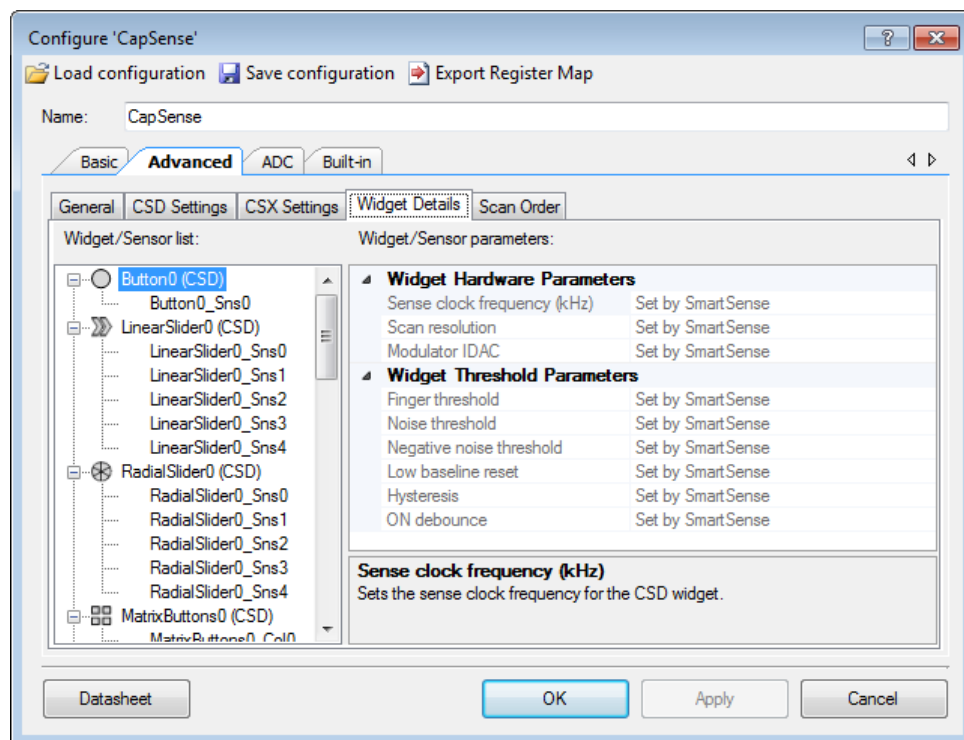
Name	Description
Modulator clock frequency	<p>Selects the modulator clock-frequency used for the <i>CSX sensing method</i>. The minimum value is 1000 kHz. The maximum value is 50000 kHz or HFCLK, whichever is lower.</p> <p>Enter any value between the min and max limits, based on the availability of the 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.</p> <p>A higher modulator clock-frequency reduces the sensor scan time, therefore, results in lower power and reduces the noise in the raw counts, so it is recommended to use the highest possible frequency.</p>

Name	Description
Tx clock source	<p><i>Tx clock frequency</i> is used to sample the input sensor. Both the type of the clock source and the clock frequency are configurable.</p> <p>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, increasing the number of bits, lowering the radiation and increasing the immunity against external noise.</p> <p><i>Tx clock frequency</i> is derived from <i>Modulator clock frequency</i> using a sense clock divider.</p> <p>The following clock sources are available:</p> <ul style="list-style-type: none"> ▪ <i>Direct</i> – SSC is disabled and a fixed clock is used. ▪ <i>SSC6, SSC7, SSC9 and SSC10</i> – The clock spreads using from 6 bits to 10 bits of the sense-clock divider respectively. ▪ <i>Auto</i> – The Component automatically selects the optimal SSC or Direct sources individually for each widget. <p>Auto is the recommended Sense clock source selection.</p> <p>The following rules should be followed at SSC selection:</p> <ul style="list-style-type: none"> ▪ The ratio between <i>Modulator clock frequency</i> and <i>Tx clock frequency</i> should be not less than 20. ▪ At least one full-spread spectrum polynomial should finish during scan time. ▪ SSC should be selected such that 2^{SSCn} should be less than or equal to 10% of the ratio between <i>Modulator clock frequency</i> and <i>Tx clock frequency</i>. <p>The number of conversions in a sample should be an integer number of the repeat period of the SSC. For example for SSC6, $(2^{SSC6}-1)*N = 63*N$ conversions should be included in a sample.</p>
Enable common Tx clock	<p>When selected, all CSX widgets share the same Tx clock with the frequency specified in the <i>Tx clock frequency</i> (kHz) parameter, otherwise <i>Tx clock frequency</i> can be entered separately for each CSX widget in the <i>Widget Details</i> tab.</p> <p>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. However, in rare cases, if the electrode properties capacitance is significantly different for each widget, a common Tx clock may not produce the optimal performance.</p>
Tx clock frequency	<p>Sets the CSX Tx clock frequency. The minimum value is 45 kHz for all device families. The maximum value is 3000 kHz.</p> <p>Enter any value between the min and max limits, based on 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.</p> <p>The highest Tx clock frequency produces the maximum signal and is the recommended setting.</p> <p>When <i>Enable common Tx clock</i> is unselected, the Tx Clock frequency can be set individually for each widget in the <i>Widget Details</i> tab, and this control is grayed out.</p> <p>Note If the HFCLK or <i>Modulator clock frequency</i> is changed, the Component automatically recalculates the next closest Tx clock frequency value to a possible one.</p>
Number of reported fingers	<p>Sets the number of reported fingers for a CSX Touchpad widget only. The available options are from 1 to 3.</p>

Name	Description
Enable IDAC auto-calibration	When enabled, values of CSX widget IDACs 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 Auto-tuning* is not enabled. The parameters are unique for each widget type.



This sub-tab contains the following parameters:

Name	Description
Widget General Parameters	
Diplexing	Enabling Diplexing allows doubling the slider physical touch sensing area by using a specific diplexing 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. A No Touch would produce 0xFFFF.
Maximum X-axis position	

PRELIMINARY



Name	Description
Maximum Y-axis position	Represents the maximum column (X-axis) Centroid position and row (Y-axis) Centroid positions for touchpad. A touch on the touchpad would produce a position value from 0 to the maximum position set. A No Touch would produce 0xFFFF.
Position filter	<p>Enables the specific filter on a Centroid position value to reduce noise due to varying finger touches.</p> <ul style="list-style-type: none"> ▪ None (default) - No filter is implemented ▪ IIR Filter - Enables the infinite-impulse response filter (See equation below) with a step response similar to an RC low-pass filter. $Output = \frac{N}{K} \times input + \frac{(K - N)}{K} \times previous \quad Output$ <p>Where: K = 256, N = 128.</p> $Output = (input + previous \quad Output) / 2$ <p>Consumes 2 bytes of RAM per each position (filter history).</p> ▪ Median Filter (3 sample) - 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). ▪ Average Filter (2 sample) – Enables the finite-impulse response filter (no feedback) with equally weighted coefficients. It takes two of most recent samples and computes their average. Eliminates the 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. The filter is most effective at low noise. Consumes 2 bytes of RAM per each position (filter history).
Widget Hardware Parameters Note All the Widget Hardware parameters for the CSD widgets are automatically set when <i>SmartSense Auto-tuning</i> is selected in the <i>CSD tuning mode</i> .	
Sense clock frequency	This parameter is identical to the <i>Sense clock frequency</i> parameter in <i>CSD Settings</i> tab. When <i>Enable common Sense clock</i> is unselected in the <i>CSD Settings</i> tab, a sense clock frequency for each widget is set here.
Row sense clock frequency	These parameters are identical to the <i>Sense clock frequency</i> parameter, and are used to set the sense clock frequency for row and column sensors of <i>Matrix Buttons</i> and <i>Touchpad</i> widgets.
Column sense clock frequency	
Tx clock frequency	This parameter is identical to <i>Tx clock frequency</i> parameter in the <i>CSX Settings</i> tab. When <i>Enable common Tx clock</i> is unselected in <i>CSX Settings</i> tab, a Tx clock frequency for each widget is set here.

Name	Description
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	<p>Selects the number of sub-conversions in the <i>CSX sensing method</i>. The number of sub-conversion should meet the following equation:</p> $N_{Sub} < \frac{2^{16} \cdot TxClk}{ModClk}$ <p>where, <i>ModClk</i> = CSX <i>Modulator clock frequency</i> <i>TxClk</i> = <i>Tx clock frequency</i> <i>N_{Sub}</i> = the value of this parameter.</p>
Modulator IDAC	<p>Sets the modulator IDAC value for the CSD Button, Slider, or Proximity widget.</p> <p>The value of this parameter is automatically set when <i>Enable IDAC auto-calibration</i> is selected in the <i>CSD Settings</i> tab.</p>
Row modulator IDAC	Sets a separate modulator IDAC value for the row and column sensors of the CSD <i>Matrix Buttons</i> and <i>Touchpad</i> widget.
Column modulator IDAC	Values of these parameters are automatically set when <i>Enable IDAC auto-calibration</i> is checked in the <i>CSD Settings</i> tab.
Widget Threshold Parameters Note All the threshold parameters for the CSD widgets are automatically set when <i>SmartSense (Full Auto-Tune)</i> is selected in the <i>CSD tuning mode</i> parameter.	
Finger threshold	<p>The finger threshold parameter is used along with the hysteresis parameter to determine the sensor state as follows:</p> <ul style="list-style-type: none"> ON: Signal > (Finger Threshold + Hysteresis) OFF: Signal ≤ (Finger Threshold – Hysteresis). <p>Note that “Signal” in the above equations refers to: Difference Count = Raw Count – Baseline.</p> <p>It is recommended to set the Finger threshold parameter value to be equal to the 80% of the touch signal.</p> <p>The Finger Threshold parameter is not available for the <i>Proximity</i> widget. Instead, Proximity has two thresholds:</p> <ul style="list-style-type: none"> <i>Proximity threshold</i> <i>Touch threshold</i>

PRELIMINARY



Name	Description
Noise threshold	<p>The noise threshold parameter sets the raw count limit. Raw count below the limit is considered as noise, when the raw count is above the Noise Threshold difference count is produced and the baseline is updated only if <i>Enable sensor auto-reset</i> 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 the raw counts during a finger touch detection event).</p> <p>It is recommended to set the noise threshold parameter value to be equal to 2x noise in the raw count or 40% of signal.</p> <p>For the <i>Linear Slider</i>, <i>Radial Slider</i> and <i>Touchpad</i> widgets, the centroid position is calculated by subtracting the noise threshold from the signals. It provides a more accurate position.</p>
Negative noise threshold	<p>The negative noise threshold parameter sets the raw count limit below which the baseline is not updated for the number of samples specified by the <i>Low baseline reset</i> parameter.</p> <p>The negative noise threshold ensures that the baseline does not fall low because of any high-amplitude repeated negative-noise spikes on the raw count caused by different noise sources such as ESD events.</p> <p>It is recommended to set the negative noise threshold parameter value to be equal to the <i>Noise threshold</i> parameter value.</p>
Low baseline reset	<p>This parameter is used along with the <i>Negative noise threshold</i> parameter. It counts the number of abnormally low raw counts required to reset the baseline.</p> <p>If a finger is placed on the sensor during a device startup, the baseline gets initialized to the high raw count value at a startup. When the finger is removed, raw counts fall to a lower value. In this case, the baseline should track the low raw counts. The Low Baseline Reset parameter helps to handle this event. It resets the baseline to the low raw count value when the number of low samples reaches the low-baseline reset number. Note that in this case, once a finger is removed from the sensor, the sensor will not respond to finger touches for low baseline-reset time.</p> <p>The recommended value is 30 which works for most designs.</p>
Hysteresis	<p>The hysteresis parameter is used along with the <i>Finger threshold</i> parameter (<i>Proximity threshold</i> and <i>Touch threshold</i> for Proximity sensor) to determine the sensor state. The hysteresis provides immunity against noisy transitions of the sensor state.</p> <p>See the description of <i>Finger threshold</i> parameter for details.</p> <p>The recommend value for the hysteresis is the 10% <i>Finger threshold</i>.</p>
ON debounce	<p>This parameter 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</p> <ul style="list-style-type: none"> 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. <p>The recommended value for the Debounce parameter is 3 for reliable sensor status detection.</p>
Proximity threshold	<p>The design of these parameters is the same as for the <i>Finger threshold</i> parameters. The proximity sensor requires a higher noise reduction, and supports two levels of detection:</p>

Name	Description
Touch threshold	<ul style="list-style-type: none"> The proximity level to detect an approaching hand or finger. The touch level to detect a finger touch on the sensor similarly to other <i>Widget Type</i> sensors. <p>Note that for valid operation, the Proximity threshold should be higher than the Touch threshold.</p> <p>The threshold parameters such as <i>Hysteresis</i> and <i>ON debounce</i> are applicable to both detection levels.</p>
Velocity	Defines the maximum speed of a finger movement in terms of 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.
Sensor parameters	
Compensation IDAC value	<p>Sets the Compensation IDAC value for each CSD sensor when <i>Enable compensation IDAC</i> is selected on <i>CSD Settings</i> tab. If <i>CSD tuning mode</i> is set to <i>SmartSense Auto-tuning</i> or <i>Enable IDAC auto-calibration</i> is selected on the <i>CSD Settings</i> 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.</p> <p>It is recommended to select <i>Enable IDAC auto-calibration</i> for robust operation.</p>
IDAC Values	<p>Sets the IDAC value for each CSX sensor/node, a lower value of IDAC without saturating the raw counts provides better performance for sensor/nodes.</p> <p>When <i>Enable IDAC auto-calibration</i> is selected on the <i>CSX Settings</i> tab, the value of this parameter is automatically set to the lowest possible value at a device power-up for better performance.</p> <p>It is recommended to select <i>Enable IDAC auto-calibration</i> for robust operation.</p>
Selected pins	Selects a port pin for the sensor (CSD sensing) and electrode (CSX sensing). The available options are using a dedicated pin for a sensor or reusing one or more pins from any other sensor in the Component. Reusing the pins of any other sensor from any widgets helps to create a ganged sensor.

PRELIMINARY



The following table shows which parameters belong to a given widget:

Widget Type		Parameters																									
		Widget General					Widget Hardware								Widget Threshold							Sensor					
		Diplexing	Maximum position	Maximum X-axis position	Maximum Y-axis position	Position filter	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	Finger threshold	Noise threshold	Negative noise threshold	Low baseline reset	Hysteresis	ON debounce	Proximity threshold	Touch threshold	Velocity	Compensation IDAC value	IDAC Values	Selected pins
CSD Widget	Button						✓				✓		✓			✓	✓	✓	✓	✓	✓				✓		✓
	Linear Slider	✓	✓			✓	✓				✓		✓			✓	✓	✓	✓	✓	✓				✓		
	Radial Slider		✓			✓	✓				✓		✓			✓	✓	✓	✓	✓	✓				✓		
	Matrix Buttons							✓	✓		✓			✓	✓	✓	✓	✓	✓	✓	✓				✓		✓
	Touchpad			✓	✓	✓		✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓				✓		
	Proximity						✓				✓		✓				✓	✓	✓	✓	✓	✓	✓		✓		✓
CSX Widget	Button									✓		✓				✓	✓	✓	✓	✓	✓					✓	✓
	Matrix Buttons									✓		✓				✓	✓	✓	✓	✓	✓					✓	✓
	Touchpad			✓	✓	✓				✓		✓				✓	✓	✓	✓	✓	✓			✓		✓	✓

Scan Order Sub-tab

This tab provides Total time required to scan all the sensors (does not include the data processing execution time) and scan time for each sensor slot.

Configure 'CapSense'

Load configuration Save configuration Export Register Map

Name: CapSense

Basic **Advanced** ADC Built-in

General CSD Settings CSX Settings Widget Details **Scan Order**

Scan slot	Sensor assignment	Mode	Sense/Tx clock (kHz)	Scan resolution (bits) / Number of sub-conversions	Slot scan time (μs)
0	Button0_Sns0	CSD	3125	12	164
1	LinearSlider0_Sns0	CSD	3125	12	164
2	LinearSlider0_Sns1	CSD	3125	12	164
3	LinearSlider0_Sns2	CSD	3125	12	164
4	LinearSlider0_Sns3	CSD	3125	12	164
5	LinearSlider0_Sns4	CSD	3125	12	164
6	RadialSlider0_Sns0	CSD	3125	12	164
7	RadialSlider0_Sns1	CSD	3125	12	164
8	RadialSlider0_Sns2	CSD	3125	12	164

Total scan time: 6 ms

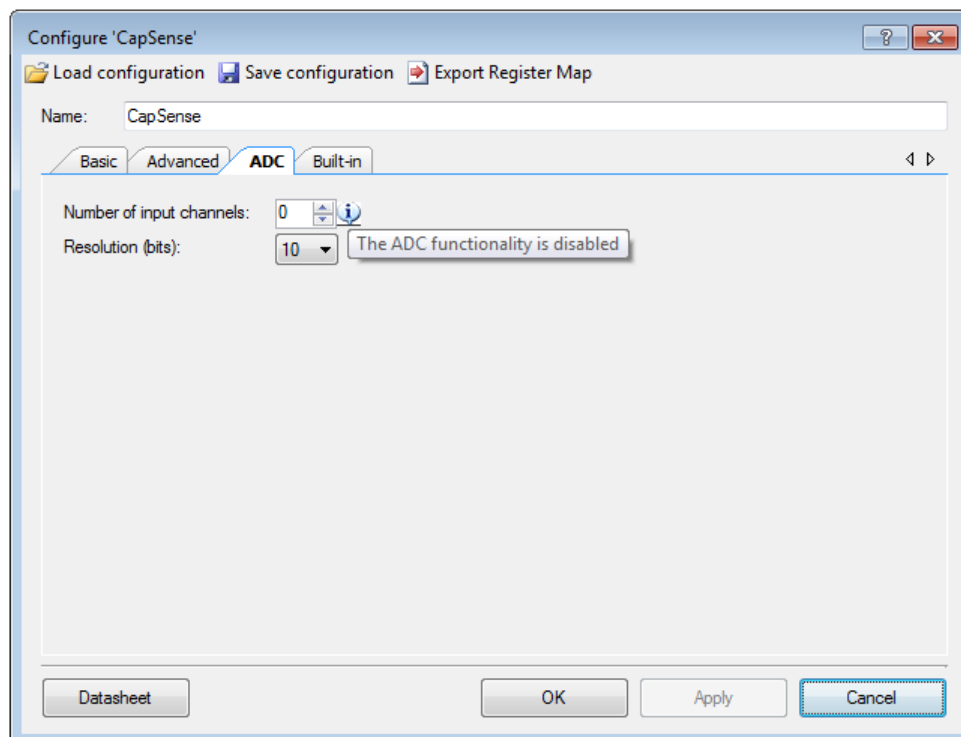
Datasheet OK Apply Cancel

PRELIMINARY



ADC Tab

The parameters in this tab apply to the ADC functionality only.



Name	Description
Number of input channels	Increment/decrement this value to specify the total input channels for the ADC. The range of valid values is 0-10. It is possible to set the number of ADC input channels to 0 to disable the ADC functionality.
Resolution (bits)	This drop-down is used to select the ADC resolution. The possible options are: <ul style="list-style-type: none"> 8 bits 10 bits

Application Programming Interface

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

The CapSense firmware library supports the following compilers:

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

In order to use the IAR Embedded Workbench refer to:

- PSoC Creator menu Help / Documentation / PSoC Creator User Guide
Section: Export a Design to a 3rd Party IDE > Exporting a Design to IAR IDE

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 API function, variable, and constant name. For readability, this section assumes “CapSense” as the instance name.

PRELIMINARY



CapSense High-Level APIs

Description

The 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 automatically handled by these APIs (based on a sensing method (CSD or CSX) or type of widgets). Therefore, these APIs are sensing-methods-agnostic, feature-agnostic, and widget-type-agnostic.

All the tasks required to implement a capacitive user interface can be fulfilled by the high-level APIs. But, there is a set of [CapSense Low-Level APIs](#) which provide access to lower level and specific tasks. If a design requires access to the low-level tasks, these APIs can be used.

The functions related to the CSD / CSX sensing methods are not available if the corresponding method is disabled.

Functions

- `cy_status 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.
- `cy_status CapSense_Stop(void)`
Stops the component operation.
- `cy_status CapSense_Resume(void)`
Resumes the component operation if the [CapSense_Stop\(\)](#) function was called previously.
- `cy_status CapSense_ProcessAllWidgets(void)`
Performs full data processing of all enabled widgets.
- `cy_status 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 sleep.
- `cy_en_syspm_status_t CapSense_DeepSleepCallback(void *base, void *content, cy_en_syspm_callback_mode_t mode)`
Handles Active to DeepSleep power mode transition for the CapSense component.
- `cy_en_syspm_status_t CapSense_EnterLowPowerCallback(void *base, void *content, cy_en_syspm_callback_mode_t mode)`
Handles Active to Low Power Active (LPActive) power mode transition for the CapSense component.
- `cy_en_syspm_status_t CapSense_ExitLowPowerCallback(void *base, void *content, cy_en_syspm_callback_mode_t mode)`
Handles Low Power Active (LPActive) to Active power mode transition for the CapSense component.
- `cy_status CapSense_SetupWidget(uint32 widgetId)`
Performs the initialization required to scan the specified widget.
- `cy_status CapSense_Scan(void)`
Initiates scanning of all the sensors in the widget initialized by [CapSense_SetupWidget\(\)](#), if no scan is in progress.
- `cy_status 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.



PRELIMINARY

- 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_IsMatrixButtonsActive](#)(uint32 widgetId)
Reports the status of the specified matrix button widget.
- uint32 [CapSense_IsProximitySensorActive](#)(uint32 widgetId, uint32 proxId)
Reports the finger detection status of the specified proximity widget/sensor.
- 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

cy_status 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.
4. 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.
5. Perform scanning for all the sensors and initialize the baseline history.
6. 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 CY_RET_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.

cy_status 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 CY_RET_SUCCESS is not received, the stop process fails and retries may be required.

Go to the top of the [CapSense High-Level APIs](#) section.

cy_status 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 CY_RET_SUCCESS is not received, the resume process fails and retries may be required.

Go to the top of the [CapSense High-Level APIs](#) section.

cy_status 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.

Disabled widgets are not processed. To disable/enable a widget, set the appropriate values in the CapSense_WDGT_ENABLE<RegisterNumber>_PARAM_ID register using the [CapSense_SetParam\(\)](#) function. This function is called only after all the sensors in the component are scanned. Calling this function multiple times without sensor scanning causes 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 processing operation. If CY_RET_SUCCESS is not received, the processing fails and retries may be required.

Go to the top of the [CapSense High-Level APIs](#) section.

cy_status 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.

The 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 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:

<i>widgetId</i>	Specifies the ID number of the widget to be processed. A macro for the widget ID can
-----------------	--



PRELIMINARY

	be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID
--	--

Returns:

Returns the status of the widget processing:

- CY_RET_SUCCESS - The operation is successfully completed
- CY_RET_BAD_PARAM - The input parameter is invalid
- CY_RET_INVALID_STATE - The specified widget is disabled
- CY_RET_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 sleep. This function shall be used to resume the component after exiting deep sleep.

Go to the top of the [CapSense High-Level APIs](#) section.

**cy_en_syspm_status_t CapSense_DeepSleepCallback (void * *base*, void * *content*,
cy_en_syspm_callback_mode_t *mode*)**

This API is registered with system power mode using Cy_SysPm_RegisterCallback() function with CY_SYSPM_DEEPSLEEP type. After registration, this API is called by Cy_SysPm_DeepSleep() to prepare or to check a status of the CapSense component prior the core entering into DeepSleep power mode.

Calling this function directly from the application layer is not recommended.

When this API is called with the mode parameter set to CY_SYSPM_CHECK_READY, the function returns CY_SYSPM_SUCCESS if no scanning is in progress, otherwise CY_SYSPM_FAIL is returned that means that device cannot enter into DeepSleep power mode without finishing the current scan that is in progress (transition to DeepSleep power mode during the scan can disrupt a sensor scan result and produce an unexpected behaviour after wakeup).

Parameters:

<i>base</i>	Not used parameter.
<i>content</i>	Not used parameter.
<i>mode</i>	Refer to the description of the cy_en_syspm_callback_mode_t type.

Returns:

Returns the status of the operation requested by the mode parameter: CY_SYSPM_SUCCESS -

DeepSleep power mode can be entered. CY_SYSPM_FAIL - DeepSleep power mode cannot be entered.

Go to the top of the [CapSense High-Level APIs](#) section.

**cy_en_syspm_status_t CapSense_EnterLowPowerCallback (void * *base*, void * *content*,
cy_en_syspm_callback_mode_t *mode*)**

This API is registered with system power mode using Cy_SysPm_RegisterCallback() function with CY_SYSPM_ENTER_LP_MODE type. After registration, this API is called by Cy_SysPm_EnterLpMode() to prepare or to check a status of the CapSense component prior the core entering into LPActive power mode.

Calling this function directly from the application layer is not recommended.

When this API is called with the mode parameter set to CY_SYSPM_CHECK_READY, the function returns CY_SYSPM_SUCCESS if no scanning is in progress, otherwise CY_SYSPM_FAIL is returned that means that

device cannot enter into LPAActive power mode without finishing the current scan that is in progress (transition to LPAActive power mode during the scan can disrupt a sensor scan result and produce an unexpected behaviour).

Parameters:

<i>base</i>	Not used parameter.
<i>content</i>	Not used parameter.
<i>mode</i>	Refer to the description of the <code>cy_en_syspm_callback_mode_t</code> type.

Returns:

Returns the status of the operation requested by the mode parameter: CY_SYSPM_SUCCESS - LPAActive power mode can be entered. CY_SYSPM_FAIL - LPAActive power mode cannot be entered.

Go to the top of the [CapSense High-Level APIs](#) section.

cy_en_syspm_status_t CapSense_ExitLowPowerCallback (void * *base*, void * *content*, cy_en_syspm_callback_mode_t *mode*)

This API is registered with system power mode using `Cy_SysPm_RegisterCallback()` function with CY_SYSPM_EXIT_LP_MODE type. After registration, this API is called by `Cy_SysPm_ExitLpMode()` to prepare or to check a status of the CapSense component prior the core exiting from LPAActive power mode.

Calling this function directly from the application layer is not recommended.

When this API is called with the mode parameter set to CY_SYSPM_CHECK_READY, the function returns CY_SYSPM_SUCCESS if no scanning is in progress, otherwise CY_SYSPM_FAIL is returned that means that device cannot exit LPAActive power mode without finishing the current scan that is in progress (transition from LPAActive to Active power mode during the scan can disrupt a sensor scan result and produce an unexpected behaviour).

Parameters:

<i>base</i>	Not used parameter.
<i>content</i>	Not used parameter.
<i>mode</i>	Refer to the description of the <code>cy_en_syspm_callback_mode_t</code> type.

Returns:

Returns the status of the operation requested by the mode parameter: CY_SYSPM_SUCCESS - Active power mode can be entered. CY_SYSPM_FAIL - Active power mode cannot be entered.

Go to the top of the [CapSense High-Level APIs](#) section.

cy_status 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 the CSD and CSX 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\(\)](#) or [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:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
-----------------	--



PRELIMINARY

Returns:

Returns the status of the widget setting up operation:

- CY_RET_SUCCESS - The operation is successfully completed.
- CY_RET_BAD_PARAM - The widget is invalid or if the specified widget is disabled
- CY_RET_INVALID_STATE - The previous scanning is not completed and the hardware block is busy.
- CY_RET_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.

cy_status 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:

- CY_RET_SUCCESS - Scanning is successfully started.
- CY_RET_INVALID_STATE - The previous scanning is not completed and the hardware block is busy.
- CY_RET_UNKNOWN - An unknown sensing method is used by the widget.

Go to the top of the [CapSense High-Level APIs](#) section.

cy_status 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:

- CY_RET_SUCCESS - Scanning is successfully started.
- CY_RET_BAD_PARAM - All the widgets are disabled.
- CY_RET_INVALID_STATE - The previous scanning is not completed and the HW block is busy.
- CY_RET_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:

PRELIMINARY



- 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:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
-----------------	---

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:

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

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_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:

<i>widgetId</i>	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
-----------------	--



PRELIMINARY

	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.

- 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.
- 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_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:

<i>widgetId</i>	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
<i>proxId</i>	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_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:

<i>widgetId</i>	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:

<i>widgetId</i>	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 the high-level APIs. The low-level APIs also enable implementation of special case designs requiring performance optimization and non-typical functionality.

The functions that contain CSD or CSX in the name are specified for the CSD or CSX sensing methods appropriately and are used only with dedicated CSD or CSX widgets. The rest of the functions are not specific to the CSD or CSX sensing methods, some of the APIs detect the sensing method used by the widget and executes tasks appropriately.

The functions related to the CSD / CSX sensing methods are not available if the corresponding method is disabled.

Functions

- cy_status [CapSense_ProcessWidgetExt](#)(uint32 widgetId, uint32 mode)
Performs customized data processing on the selected widget.
- cy_status [CapSense_ProcessSensorExt](#)(uint32 widgetId, uint32 sensorId, uint32 mode)
Performs customized data processing on the selected widget's sensor.
- cy_status [CapSense_UpdateAllBaselines](#)(void)
Updates the baseline for all the sensors in all the widgets.
- cy_status [CapSense_UpdateWidgetBaseline](#)(uint32 widgetId)
Updates the baselines for all the sensors in a widget specified by the input parameter.
- cy_status [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_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.
- cy_status [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.
- cy_status [CapSense_CalibrateAllWidgets](#)(void)
Calibrates the IDACs for all the widgets in the component to default target value, 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.
- cy_status [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_STRUCT](#)const *snsAddrPtr)
Connects a port pin used by the sensor to the AMUX bus of the CapSense block.
- void [CapSense_CSDDisconnectSns](#) ([CapSense_FLASH_IO_STRUCT](#)const *snsAddrPtr)
Disconnects a sensor port pin from the CapSense 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_STRUCT](#)const *txPtr)
Connects a TX electrode to the CSX scanning hardware.
- void [CapSense_CSXConnectRx](#) ([CapSense_FLASH_IO_STRUCT](#)const *rxPtr)
Connects an RX electrode to the CSX scanning hardware.
- void [CapSense_CSXDisconnectTx](#) ([CapSense_FLASH_IO_STRUCT](#)const *txPtr)
Disconnects a TX electrode from the CSX scanning hardware.
- void [CapSense_CSXDisconnectRx](#) ([CapSense_FLASH_IO_STRUCT](#)const *rxPtr)
Disconnects an RX electrode from the CSX scanning hardware.
- cy_status [CapSense_GetParam](#)(uint32 paramId, uint32 *value)
Gets the specified parameter value from the [Data Structure](#).
- cy_status [CapSense_SetParam](#)(uint32 paramId, uint32 value)
Sets a new value for the specified parameter in the [Data Structure](#).

Function Documentation

cy_status 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.



PRELIMINARY

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

Parameters:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
<i>mode</i>	Specifies the type of widget processing to be executed for the specified widget: <ol style="list-style-type: none"> 1. Bits [31..6] - Reserved. 2. Bits [5..0] - <code>CapSense_PROCESS_ALL</code> - Execute all the tasks. 3. Bit [5] - <code>CapSense_PROCESS_STATUS</code> - Update the status (on/off, centroid position). 4. Bit [4] - <code>CapSense_PROCESS_THRESHOLDS</code> - Update the thresholds (only in CSD auto-tuning mode). 5. Bit [3] - <code>CapSense_PROCESS_CALC_NOISE</code> - Calculate the noise (only in CSD auto-tuning mode). 6. Bit [2] - <code>CapSense_PROCESS_DIFFCOUNTS</code> - Update the difference counts. 7. Bit [1] - <code>CapSense_PROCESS_BASELINE</code> - Update the baselines. 8. Bit [0] - <code>CapSense_PROCESS_FILTER</code> - Run the firmware filters.

Returns:

Returns the status of the widget processing operation:

- `CY_RET_SUCCESS` - The processing is successfully performed.
- `CY_RET_BAD_PARAM` - The input parameter is invalid.
- `CY_RET_BAD_DATA` - The processing failed.

Go to the top of the [CapSense Low-Level APIs](#) section.

cy_status 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:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
<i>sensorId</i>	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 <code>CapSense_<WidgetName>_SNS<SensorNumber>_ID</code> .
<i>mode</i>	Specifies the type of the sensor processing that needs to be executed for the specified sensor: <ol style="list-style-type: none"> 1. Bits [31..5] - Reserved 2. Bits [4..0] - <code>CapSense_PROCESS_ALL</code> - Executes all the tasks 3. Bit [4] - <code>CapSense_PROCESS_THRESHOLDS</code> - Updates the thresholds (only in auto-tuning mode) 4. Bit [3] - <code>CapSense_PROCESS_CALC_NOISE</code> - Calculates the noise (only in auto-tuning mode) 5. Bit [2] - <code>CapSense_PROCESS_DIFFCOUNTS</code> - Updates the difference count 6. Bit [1] - <code>CapSense_PROCESS_BASELINE</code> - Updates the baseline 7. Bit [0] - <code>CapSense_PROCESS_FILTER</code> - Runs the firmware filters

Returns:

Returns the status of the sensor process operation:

- `CY_RET_SUCCESS` - The processing is successfully performed.
- `CY_RET_BAD_PARAM` - The input parameter is invalid.
- `CY_RET_BAD_DATA` - The processing failed.

Go to the top of the [CapSense Low-Level APIs](#) section.

cy_status 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:

- `CY_RET_SUCCESS` - The operation is successfully completed.
- `CY_RET_BAD_DATA` - The baseline processing failed.

Go to the top of the [CapSense Low-Level APIs](#) section.

cy_status 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:

<i>widgetId</i>	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
-----------------	---

**PRELIMINARY**

	header file defined as CapSense_<WidgetName>_WDGT_ID.
--	---

Returns:

Returns the status of the specified widget update baseline operation:

- CY_RET_SUCCESS - The operation is successfully completed.
- CY_RET_BAD_DATA - The baseline processing failed.

Go to the top of the [CapSense Low-Level APIs](#) section.

cy_status 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:

<i>widgetId</i>	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.
<i>sensorId</i>	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:

- CY_RET_SUCCESS - The operation is successfully completed.
- CY_RET_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.

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.

Parameters:

<i>widgetId</i>	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:

<i>widgetId</i>	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.
<i>sensorId</i>	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

PRELIMINARY



	Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.
--	---

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 number, for the CSX widgets, it is either an RX or TX. If the sensor element is a ganged sensor, then the specified state is also set for all ganged pins of this sensor. Scanning must be completed before calling this API.

The CapSense_SHIELD and CapSense_SENSOR states are not allowed if there is no CSD widget configured in the user's project. The CapSense_TX_PIN and CapSense_RX_PIN states are not allowed if there is no CSX widget configured in the user's project.

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. They ignore changes in the design made by the [CapSense_SetPinState\(\)](#) function. This function neither check wdgtIndex nor sensorElement for the correctness.

Parameters:

<i>widgetId</i>	Specifies the ID number 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.
<i>sensorElement</i>	Specifies the ID number 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: <ul style="list-style-type: none"> CapSense_<WidgetName>_RX<RXNumber>_ID CapSense_<WidgetName>_TX<TXNumber>_ID.
<i>state</i>	Specifies the state of the sensor to be set: <ol style="list-style-type: none"> 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 (only in CSD sensing method when shield electrode is enabled). 4. CapSense_SENSOR - The pin is connected to the scanning bus (only in CSD sensing method). 5. CapSense_TX_PIN - The TX signal is routed to the sensor (only in CSX sensing method). 6. CapSense_RX_PIN - The pin is connected to the scanning bus (only in CSX sensing method).

Go to the top of the [CapSense Low-Level APIs](#) section.

cy_status 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.



PRELIMINARY

Parameters:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
-----------------	--

Returns:

Returns the status of the specified widget calibration:

- `CY_RET_SUCCESS` - The operation is successfully completed.
- `CY_RET_BAD_PARAM` - The input parameter is invalid.
- `CY_RET_BAD_DATA` - The calibration failed and the component may not operate as expected.

Go to the top of the [CapSense Low-Level APIs](#) section.

cy_status 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:

- `CY_RET_SUCCESS` - The operation is successfully completed.
- `CY_RET_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:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
-----------------	--

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:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
<i>sensorId</i>	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 <code>CapSense_<WidgetName>_SNS<SensorNumber>_ID</code> .

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.

cy_status 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.

**PRELIMINARY**

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:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
<i>target</i>	Specifies the calibration target in percentages of the maximum raw count.

Returns:

Returns the status of the specified widget calibration:

- `CY_RET_SUCCESS` - The operation is successfully completed.
- `CY_RET_BAD_PARAM` - The input parameter is invalid.
- `CY_RET_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_STRUCT](#)const * *snsAddrPtr*)

Connects a port pin used by the sensor to the AMUX bus of the CapSense 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:

<i>snsAddrPtr</i>	Specifies the pointer to the <code>FLASH_IO_STRUCT</code> object belonging to a sensor which to be connected to the CapSense block.
-------------------	---

Go to the top of the [CapSense Low-Level APIs](#) section.

void CapSense_CSDDisconnectSns ([CapSense_FLASH_IO_STRUCT](#)const * *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:

<i>snsAddrPtr</i>	Specifies the pointer to the <code>FLASH_IO_STRUCT</code> object belonging to a sensor which should be disconnected from the CapSense 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:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
-----------------	--

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:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
<i>sensorId</i>	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 <code>CapSense_<WidgetName>_SNS<SensorNumber>_ID</code> .

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:

<i>widgetId</i>	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 <code>CapSense_<WidgetName>_WDGT_ID</code> .
<i>target</i>	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_STRUCT const * 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:

<i>txPtr</i>	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a sensor to be connected to the CapSense block as a Tx pin.
--------------	--

Go to the top of the [CapSense Low-Level APIs](#) section.

void CapSense_CSXConnectRx (CapSense_FLASH_IO_STRUCT const * 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:

<i>rxPtr</i>	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a sensor to be connected to the CapSense block as an Rx pin.
--------------	---

Go to the top of the [CapSense Low-Level APIs](#) section.

void CapSense_CSXDisconnectTx ([CapSense_FLASH_IO_STRUCT](#) const * *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:

<i>txPtr</i>	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a Tx pin sensor to be disconnected from the CapSense block.
--------------	--

Go to the top of the [CapSense Low-Level APIs](#) section.

void CapSense_CSXDisconnectRx ([CapSense_FLASH_IO_STRUCT](#) const * *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:

<i>rxPtr</i>	Specifies the pointer to the FLASH_IO_STRUCT object belonging to an Rx pin sensor to be disconnected from the CapSense block.
--------------	---

Go to the top of the [CapSense Low-Level APIs](#) section.

cy_status 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:

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

**PRELIMINARY**

Parameters:

<i>paramId</i>	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.
<i>value</i>	The pointer to a variable to be updated with the obtained value.

Returns:

Returns the status of the operation:

- CY_RET_SUCCESS - The operation is successfully completed.
- CY_RET_BAD_PARAM - The input parameter is invalid.

Go to the top of the [CapSense Low-Level APIs](#) section.

cy_status 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. [TTWFC CCC 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:

<i>paramId</i>	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.
<i>value</i>	Specifies the new parameter's value.

Returns:

Returns the status of the operation:

- CY_RET_SUCCESS - The operation is successfully completed.
- CY_RET_BAD_PARAM - The input parameter is invalid.

Go to the top of the [CapSense Low-Level APIs](#) section.

ADC Application Public Interface

Description

The ADC application public interface represents the abstraction layer of the ADC function. The ADC public interface is exposed to the user to implement the ADC function.

If ADC is not configured then ADC-related functions are not available.

Functions

- `cy_status CapSense_AdcStartConvert(uint8 chId)`
Initializes the hardware and initiates an analog-to-digital conversion on the selected input channel.
- `uint8 CapSense_AdclsBusy(void)`
The function returns the status of the ADC's operation.
- `uint16 CapSense_AdcReadResult_mVolts(uint8 chId)`
This is a blocking API. It initiates a conversion, waits for completion and returns the result.
- `uint16 CapSense_AdcGetResult_mVolts(uint8 chId)`
This API does not perform an ADC conversion and returns the last valid result for the specified channel.
- `cy_status CapSense_AdcCalibrate(void)`
Performs calibration of the ADC module.
- `void CapSense_AdcStop(void)`
Disables the hardware sub-blocks that are in use while in the ADC mode, and frees the routing.
- `void CapSense_AdcResume(void)`
Resumes the ADC operation after a stop call.

Function Documentation

`cy_status CapSense_AdcStartConvert (uint8 chId)`

Initializes the hardware and initiates an analog-to-digital conversion on the selected input channel. This API only initiates a conversion and does not wait for the conversion to be completed, therefore the [CapSense_AdclsBusy\(\)](#) API must be used to check the status and ensure that the conversion is complete prior to reading the result, starting a new conversion with the same or a different channel or reconfiguring the hardware for different functionality.

Parameters:

<code>chId</code>	The ID of the channel to be converted.
-------------------	--

Returns:

The function returns `cystatus` of its operation.

- `CY_RET_SUCCESS` - A conversion has started.
- `CY_RET_LOCKED` - The hardware is already in-use by a previously initialized conversion or other functionality. No new conversion is started by this API.
- `CY_RET_BAD_PARAM` - An invalid channel Id. No conversion is started.

Go to the top of the [ADC Application Public Interface](#) section.

`uint8 CapSense_AdclsBusy (void)`

The function returns the status of the ADC's operation. A new conversion or calibration must not be started unless the ADC is in the IDLE state.



PRELIMINARY

Returns:

The function returns the status of the ADC's operation.

- CapSense_AdcSTATUS_IDLE - The ADC is not busy, a new conversion can be initiated.
- CapSense_AdcSTATUS_CONVERTING - A previously initiated conversion is in progress.
- CapSense_AdcSTATUS_CALIBPH1 - The ADC is in the first phase (of 3) of calibration.
- CapSense_AdcSTATUS_CALIBPH2 - The ADC is in the second phase (of 3) of calibration.
- CapSense_AdcSTATUS_CALIBPH3 - The ADC is in the third phase (of 3) of calibration.
- CapSense_AdcSTATUS_OVERFLOW - The most recent measurement caused an overflow. The root cause of the overflow may be the previous calibration values being invalid or the VDDA setting in cydwr and hardware do not match. Perform re-calibration or set the appropriate VDDA value in cydwr to avoid this error condition.

Go to the top of the [ADC Application Public Interface](#) section.

uint16 CapSense_AdcReadResult_mVolts (uint8 chId)

This is a blocking API. Internally, it starts a conversion using [CapSense_AdcStartConvert\(\)](#), checks the status using [CapSense_AdcIsBusy\(\)](#), waits until the conversion is completed and returns the result.

Parameters:

<i>chId</i>	The ID of the channel to be measured
-------------	--------------------------------------

Returns:

The function returns voltage in millivolts or CapSense_AdcVALUE_BAD_RESULT if:

- chId is invalid
- The ADC conversion is not started
- The ADC conversion watch-dog triggered.

Go to the top of the [ADC Application Public Interface](#) section.

uint16 CapSense_AdcGetResult_mVolts (uint8 chId)

Returns the last valid result from the data structure for the specified channel. This function can be used to read a previous result of any channel even if the ADC is busy or a conversion is in progress. However, it is highly recommended not to use this function with a channel that is in an active conversion.

Parameters:

<i>chId</i>	The ID of the channel to be measured
-------------	--------------------------------------

Returns:

The function returns a voltage in millivolts or CapSense_AdcVALUE_BAD_CHAN_ID if chId is invalid.

Go to the top of the [ADC Application Public Interface](#) section.

cy_status CapSense_AdcCalibrate (void)

Performs calibration for the ADC to identify the appropriate hardware configuration to produce accurate results. It is recommended to run the calibration periodically (for example every 10 seconds) for accuracy and compensations.

Returns:

The function returns cystatus of its operation.

- CY_RET_SUCCESS - The block is configured for the ADC use.
- CY_RET_LOCKED - The hardware is already in-use by a previously initialized conversion or other functionality. No new conversion is started by this API.

Go to the top of the [ADC Application Public Interface](#) section.

void CapSense_AdcStop (void)

This function stops the component operation, no ADC conversion can be initiated when the component is stopped. Once stopped, the hardware block may be reconfigured by the application program for any other

special usage. The ADC operation can be resumed by calling the [CapSense_AdcResume\(\)](#) function or the component can be reset by calling the [CapSense_Start\(\)](#) function. This function is called when no ADC conversion is in progress.

Go to the top of the [ADC Application Public Interface](#) section.

void CapSense_AdcResume (void)

Resumes the ADC operation if the operation is stopped previously by the [CapSense_AdcStop\(\)](#) API.

Go to the top of the [ADC Application Public Interface](#) 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 a critical sections in 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 APIs are not used in the application layer.

Functions

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

Function Documentation

void CapSense_AdcIntrHandler (void)

This ISR is triggered after a measurement completes or during the calibration phases.

To use the entry or exit callbacks, define `CapSense_ADC_[ENTRY|EXIT]_CALLBACK` and define the corresponding function, `CapSense_Adc[Entry|Exit]Callback()`.

Go to the top of the [Interrupt Service Routine](#) section.

void CapSense_CSDPostSingleScan (void)

This ISR handler is triggered when the user calls the [CapSense_CSDScanExt\(\)](#) function.

The following tasks are performed:

1. Check if the raw data is not noisy.
2. Read the Counter register and update the data structure with raw data.



PRELIMINARY

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.

void CapSense_CSDPostMultiScan (void)

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.

void CapSense_CSDPostMultiScanGanged (void)

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.

void CapSense_CSXScanISR (void)

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 if 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 CSDv2 HW IP 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 custom code from the API files automatically generated by PSoC Creator. Refer to the PSoC Creator Help and Component Author Guide for the more details.

To add 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 header file). This will “uncomment” the function call from the component's source code.
- Write the function declaration using the provided in the table name (in cyapicallbacks header file). 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.

CapSense Adc Macro Callbacks

Macro Callback Function Name	Associated Macro	Description
CapSense_AdcEntryCallback	CapSense_ADC_ENTRY_CALLBACK	Used at the beginning of the ADC interrupt handler to perform additional application-specific actions.
CapSense_AdcExitCallback	CapSense_ADC_EXIT_CALLBACK	Used at the end of the ADC interrupt handler to perform additional application-specific actions.

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_STRUCT](#) [CapSense_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_FLASH_IO_STRUCT](#) [CapSense_shieldIoList](#)[CapSense_CSD_TOTAL_SHIELD_COUNT]
- const [CapSense_FLASH_IO_STRUCT](#) [CapSense_adcIoList](#)[CapSense_ADC_TOTAL_CHANNELS]

Variable Documentation

const [CapSense_FLASH_STRUCT](#) [CapSense_dsFlash](#)

Constant for the FLASH Data Structure

const [CapSense_FLASH_IO_STRUCT](#) [CapSense_ioList](#)[CapSense_TOTAL_ELECTRODES]

The array of the pointers to the electrode specific register.

const [CapSense_FLASH_IO_STRUCT](#) [CapSense_shieldIoList](#)[CapSense_CSD_TOTAL_SHIELD_COUNT]

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

const [CapSense_FLASH_IO_STRUCT](#) [CapSense_adcIoList](#)[CapSense_ADC_TOTAL_CHANNELS]

The array of the pointers to the ADC input channels specific register.

Data Structure

Description

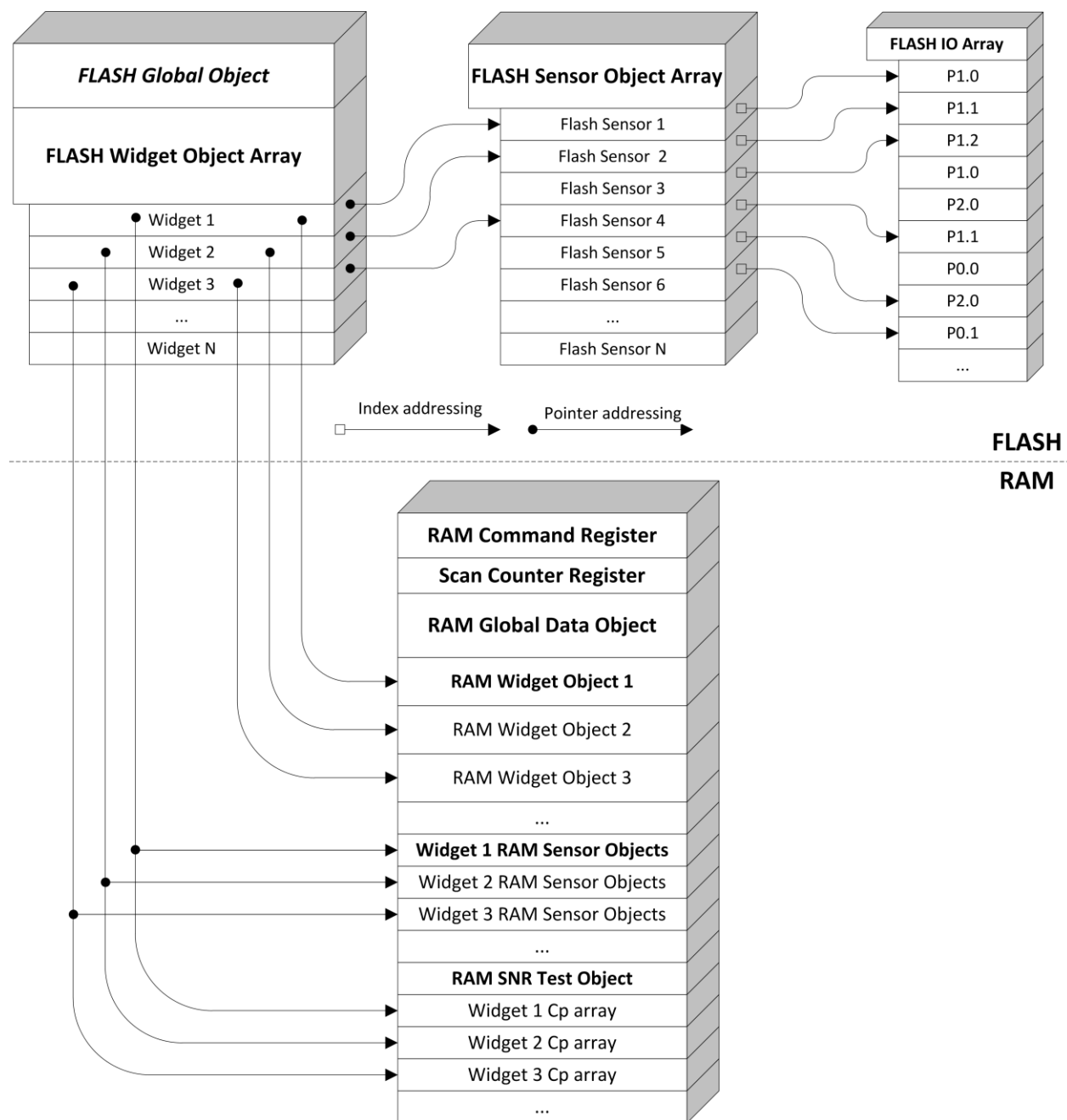
This section provides the list of structures/registers available in the component.

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>)
```

PRELIMINARY



```
#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) 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 [CapSense_RAM_WD_BASE_STRUCT](#)
Declare common widget RAM parameters.
- struct [CapSense_RAM_WD_BUTTON_STRUCT](#)
Declare RAM parameters for the CSD Button.
- struct [CapSense_RAM_WD_SLIDER_STRUCT](#)
Declare RAM parameters for the Slider.



PRELIMINARY

- struct [CapSense_RAM_WD_CSD_MATRIX_STRUCT](#)
Declare RAM parameters for the CSD Matrix Buttons.
- struct [CapSense_RAM_WD_CSD_TOUCHPAD_STRUCT](#)
Declare RAM parameters for the CSD Touchpad.
- struct [CapSense_RAM_WD_PROXIMITY_STRUCT](#)
Declare RAM parameters for the Proximity.
- struct [CapSense_RAM_WD_CSX_MATRIX_STRUCT](#)
Declare 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.

Data Structure Documentation

struct CapSense_RAM_WD_BASE_STRUCT

Go to the top of the [Data Structures](#) section.

Data Fields:

uint16	crc	CRC for the whole Widget Object in RAM (not only the common part)
uint16	resolution	Provides scan resolution for the CSD Widgets. Provides number of the sub-conversions for the CSX Widgets.
CapSense_TH RESHOLD_TY PE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger threshold. OFF to ON.
CapSense_LO W_BSLN_RST_	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the

PRELIMINARY



TYPE		Negative Noise Threshold to trigger a baseline reset.
uint8	bslnCoeff	The widget baseline filter coefficient N (for IIR 2 to 8) or baseline update threshold (for bucket method 1 to 255)
uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the CSD widgets. For the CSD Touchpad and Matrix Button widgets sets the current of the modulation IDAC for the column sensors. Not used for the CSX widgets.
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 the 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	offDebounce	The Widget Debounce for a lift-off event. ON to OFF.

struct CapSense_RAM_WD_BUTTON_STRUCTGo to the top of the [Data Structures](#) section.**Data Fields:**

uint16	crc	CRC for the whole Widget Object in RAM (not only the common part)
uint16	resolution	Provides scan resolution for the CSD Widgets. Provides number of the sub-conversions for the CSX Widgets.
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 threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger 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	bslnCoeff	The widget baseline filter coefficient N (for IIR 2 to 8) or baseline update threshold (for bucket method 1 to 255)
uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the CSD widgets. For the CSD Touchpad and Matrix Button widgets sets the current of the modulation IDAC for the column sensors. Not used for the CSX widgets.
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.

**PRELIMINARY**

	UM_SCAN_FREQS]	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 the 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	offDebounce	The Widget Debounce for a lift-off event. ON to OFF.

struct CapSense_RAM_WD_SLIDER_STRUCTGo to the top of the [Data Structures](#) section.**Data Fields:**

uint16	crc	CRC for the whole Widget Object in RAM (not only the common part)
uint16	resolution	Provides scan resolution for the CSD Widgets. Provides number of the sub-conversions for the CSX Widgets.
CapSense_TH RESHOLD_TY PE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger threshold. OFF to ON.
CapSense_LO W_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	bslnCoeff	The widget baseline filter coefficient N (for IIR 2 to 8) or baseline update threshold (for bucket method 1 to 255)
uint8	idacMod[CapSense_NUM _SCAN_FREQS]	Sets the current of the modulation IDAC for the CSD widgets. For the CSD Touchpad and Matrix Button widgets sets the current of the modulation IDAC for the column sensors. Not used for the CSX widgets.
uint8	rowIdacMod[CapSense_N UM_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 the CSX Widgets.
uint16	rowSnsClk	For the Matrix Buttons and Touchpad widgets specifies

PRELIMINARY

		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	offDebounce	The Widget Debounce for a lift-off event. ON to OFF.
uint16	position[CapSense_NUM_CENTROIDS]	Reports the widget position.

struct CapSense_RAM_WD_CSD_MATRIX_STRUCTGo to the top of the [Data Structures](#) section.**Data Fields:**

uint16	crc	CRC for the whole Widget Object in RAM (not only the common part)
uint16	resolution	Provides scan resolution for the CSD Widgets. Provides number of the sub-conversions for the CSX Widgets.
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 threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger 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	bslnCoeff	The widget baseline filter coefficient N (for IIR 2 to 8) or baseline update threshold (for bucket method 1 to 255)
uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the CSD widgets. For the CSD Touchpad and Matrix Button widgets sets the current of the modulation IDAC for the column sensors. Not used for the CSX widgets.
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 the 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	offDebounce	The Widget Debounce for a lift-off event. ON to OFF.
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.

struct CapSense_RAM_WD_CSD_TOUCHPAD_STRUCT

Go to the top of the [Data Structures](#) section.

Data Fields:

uint16	crc	CRC for the whole Widget Object in RAM (not only the common part)
uint16	resolution	Provides scan resolution for the CSD Widgets. Provides number of the sub-conversions for the CSX Widgets.
CapSense_TH RESHOLD_TY PE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger threshold. OFF to ON.
CapSense_LO W_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	bslnCoeff	The widget baseline filter coefficient N (for IIR 2 to 8) or baseline update threshold (for bucket method 1 to 255)
uint8	idacMod[CapSense_NUM _SCAN_FREQS]	Sets the current of the modulation IDAC for the CSD widgets. For the CSD Touchpad and Matrix Button widgets sets the current of the modulation IDAC for the column sensors. Not used for the CSX widgets.
uint8	rowIdacMod[CapSense_N UM_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 the 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	offDebounce	The Widget Debounce for a lift-off event. ON to OFF.
uint16	posX	The X coordinate.

PRELIMINARY

uint16	posY	The Y coordinate.
--------	------	-------------------

struct CapSense_RAM_WD_PROXIMITY_STRUCT

Go to the top of the [Data Structures](#) section.

Data Fields:

uint16	crc	CRC for the whole Widget Object in RAM (not only the common part)
uint16	resolution	Provides scan resolution for the CSD Widgets. Provides number of the sub-conversions for the CSX Widgets.
CapSense_TH RESHOLD_TY PE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger threshold. OFF to ON.
CapSense_LO W_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	bslnCoeff	The widget baseline filter coefficient N (for IIR 2 to 8) or baseline update threshold (for bucket method 1 to 255)
uint8	idacMod[CapSense_NUM _SCAN_FREQS]	Sets the current of the modulation IDAC for the CSD widgets. For the CSD Touchpad and Matrix Button widgets sets the current of the modulation IDAC for the column sensors. Not used for the CSX widgets.
uint8	rowIdacMod[CapSense_N UM_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 the 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	offDebounce	The Widget Debounce for a lift-off event. ON to OFF.
CapSense_TH RESHOLD_TY PE	proxTouchTh	The proximity touch threshold.

struct CapSense_RAM_WD_CSX_MATRIX_STRUCT

Go to the top of the [Data Structures](#) section.

**PRELIMINARY**

Data Fields:

uint16	crc	CRC for the whole Widget Object in RAM (not only the common part)
uint16	resolution	Provides scan resolution for the CSD Widgets. Provides number of the sub-conversions for the CSX Widgets.
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 threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger 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	bslnCoeff	The widget baseline filter coefficient N (for IIR 2 to 8) or baseline update threshold (for bucket method 1 to 255)
uint8	idacMod[CapSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the CSD widgets. For the CSD Touchpad and Matrix Button widgets sets the current of the modulation IDAC for the column sensors. Not used for the CSX widgets.
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 the 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	offDebounce	The Widget Debounce for a lift-off event. ON to OFF.

struct CapSense_RAM_WD_LIST_STRUCTGo 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_RADIALSLIDER_STRUCT	radialslider0	RadialSlider0 widget RAM structure

PRELIMINARY

M_WD_SLIDER_STRUCT		
CapSense_RAM_WD_CSD_MATRIX_STRUCT_I	matrixbuttons0	MatrixButtons0 widget RAM structure
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_I	matrixbuttons1	MatrixButtons1 widget RAM structure

struct CapSense_RAM_SNS_STRUCTGo 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.
uint16	bslnInv[CapSense_NUM_SCAN_FREQS]	The bit inverted 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]	CSD Widgets: The compensation IDAC value. CSX Widgets: The balancing IDAC value.

struct CapSense_RAM_SNS_LIST_STRUCTGo 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_STRUCT	radialslider0[CapSense_RADIALSLIDER0_NUM_SENSORS]	RadialSlider0 sensors RAM structures array

**PRELIMINARY**

M_SNS_STRUC_T	ADIALSLIDER0_NUM_SENSORS]	
CapSense_RAM_SNS_STRUC_T	matrixbuttons0[CapSense_MATRIXBUTTONS0_NUM_COLS+CapSense_MATRIXBUTTONS0_NUM_ROWS]	MatrixButtons0 sensors RAM structures array
CapSense_RAM_SNS_STRUC_T	touchpad0[CapSense_TOUCHPAD0_NUM_COLS+CapSense_TOUCHPAD0_NUM_ROWS]	Touchpad0 sensors RAM structures array
CapSense_RAM_SNS_STRUC_T	proximity0[CapSense_PROXIMITY0_NUM_SENSORS]	Proximity0 sensors RAM structures array
CapSense_RAM_SNS_STRUC_T	button1[CapSense_BUTTON1_NUM_SENSORS]	Button1 sensors RAM structures array
CapSense_RAM_SNS_STRUC_T	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	tunerCmd	Tuner Command Register. Used for the communication between the Tuner GUI and the component.
uint16	scanCounter	This counter gets incremented after each scan.
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

PRELIMINARY

		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	adcResult[CapSense_ADC_TOTAL_CHANNELS]	Stores the latest ADC result for the channel. The array size is equal to the number of ADC channels used in the project.
uint16	adcCode[CapSense_ADC_TOTAL_CHANNELS]	Stores the latest ADC conversion result for the channel.
uint8	adcStatus	Stores the status of ADC.
uint8	adcldac	ADC IDAC
uint16	csd0Config	The configuration register for global parameters of the CSD0 block.
uint16	csd1Config	The configuration register for global parameters of the CSD1 block.
uint8	modCsdClk	The modulator clock divider for the CSD widgets.
uint8	modCsxCk	The modulator clock divider for the CSX widgets.
uint16	snsCsdClk	The global sense clock divider for the CSD widgets.
uint16	snsCsxCk	Global sense clock divider for the CSX widgets.
uint8	adcResolution	Stores the ADC resolution.
uint8	adcAzTime	Stores the AZ time used for ADC conversion.
CapSense_RAM_WD_LIST_STRUCT	wdgtList	RAM Widget Objects.
CapSense_RAM_SNS_LIST_STRUCT	snsList	RAM Sensor Objects.
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_STRUCTGo to the top of the [Data Structures](#) section.**Data Fields:**

GPIO_PRT_Type *	pcPtr	Pointer to the base port register of the IO.
uint8	pinNumber	Position of the IO in the port.

struct CapSense_FLASH_SNS_STRUCTGo 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: <ul style="list-style-type: none"> • ELTD_TYPE_SELF_E - CSD sensor; • ELTD_TYPE_MUT_TX_E - CSX Tx sensor; • ELTD_TYPE_MUT_RX_E - CSX Rx sensor;

struct CapSense_FLASH_SNS_LIST_STRUCTGo to the top of the [Data Structures](#) section.**Data Fields:**

CapSense_FLASH_SNS_LIST_STRUCT	proximity0[CapSense_PROXIMITY0_NUM_SENSORS]	Proximity0 FLASH electrodes array
--	---	-----------------------------------

struct CapSense_FLASH_WD_STRUCTGo 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 *	ptr2FiltrHistory	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.
uint16	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

PRELIMINARY



		for the X-axis position calculation. Calculated as follows: RADIAL: $(WD_X_RESOLUTION * 256) / WD_NUM_COLS$; LINEAR: $(WD_X_RESOLUTION * 256) / (WD_NUM_COLS - 1)$; TOUCHPAD: the same as LINEAR
uint32	yCentroidMultiplier	The pre-calculated Y resolution centroid multiplier used for the Y-axis position calculation. Calculated as follows: $(WD_Y_RESOLUTION * 256) / (WD_NUM_ROWS - 1)$;
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.
const uint8 *	ptr2DiplexTable	The pointer to the Flash Diplex table that is used by the slider centroid algorithm.
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.

struct CapSense_FLASH_STRUCT

Go to the top of the [Data Structures](#) section.

Data Fields:

CapSense_FLASH_WIDGET_STRUCT	wdgtArray[CapSense_TO TAL_WIDGETS]	Array of flash widget objects
CapSense_FLASH_SENSOR_LIST_STRUCT	eltdList	Structure with all Ganged Flash electrode objects

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 given Component configuration.

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

PSoC 6 (GCC)

The following Component configuration is used to represent the memory usage (**Preliminary**):

Configuration	Memory Consumption	
	Flash	SRAM
<i>Configuration #1: CSX Matrix Button – One widget with 4 Rx and 8 Tx.</i> ADC (disabled): <i>Number of input channels</i> = 0.		
Configuration #1	< 4800	< 500
<i>Configuration #2: CSX Touchpad – One widget with 9 Rx and 4 Tx.</i> ADC (disabled): <i>Number of input channels</i> = 0.		
Configuration #2	< 7100	< 800
<i>Configuration #3: CSD Buttons – Three widgets with 4, 3 and 3 sensors in each widget, and <i>Manual</i> tuning mode is selected.</i> ADC (disabled): <i>Number of input channels</i> = 0 (except where noted).		
Configuration #3	< 5500	< 300
Configuration #3 + <i>SmartSense (Full Auto-Tune)</i> mode is selected	< 6600	< 400
Configuration #3 + All firmware raw count filters enabled. The following parameters are used to enable filters: <i>Enable IIR filter (First order)</i> , <i>Enable average filter (4-sample)</i> and <i>Enable median filter (3-sample)</i> .	< 6100	< 400
Configuration #3 + ADC (enabled): <i>Resolution (bits)</i> = 10-bit / <i>Number of input channels</i> = 10.	<7600	<300

Note 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.

PRELIMINARY



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 should be added to the project and the Component register map should be exposed to the tuner application.

It is possible to edit the parameters using the Tuner application and apply the new settings to the device using the **To Device** button when using the *Manual* or *SmartSense (Hardware parameters only)* modes for tuning. In the *SmartSense (Hardware parameters only)* mode, all the threshold parameters can be modified. In the *Manual* mode, all the parameters can be modified. 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 *Sync'd read* control in *Graph Setup Pane* is enabled. The *Sync'd read* control can be enabled when the FW flow regularly calls the CapSense_RunTuner() API. If this API is not present in the application code, then the synchronized read 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 design guide.

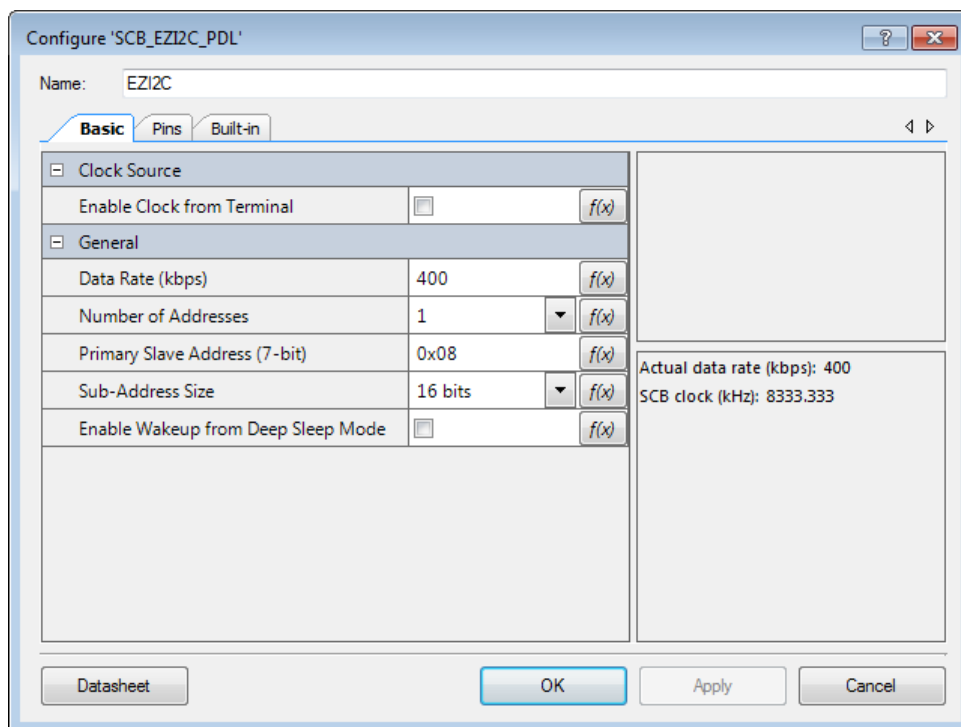
Tuning Quick Start with EzI2C

The following steps show how to set up CapSense tuning across an I2C communication channel. These steps extend the application described in the *Quick Start* section.

Step-1: Place and Configure an EZI2C Component

1. Drag and drop the EZI2C (SCB) Component from the Component Catalog onto the schematic to add an I²C communication interface to the project. This I²C slave interface is required for the Tuner GUI to monitor Component parameters in real time.
2. Double-click on the EZI2C Component.

3. On the **EZI2C Basic** tab, set the following parameters.



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

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

Step-2: Assign I2C Pins in Pin Editor

Double-click the Design-Wide Resources Pin Editor (in the Workspace Explorer) and assign physical pins for the I2C SCL and SDA pins.

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

Step-3: Modify Application Code

Replace your *main_cm4.c* from the [Step-3](#) in the [Quick Start](#) section with the following code:

```
#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_SetBuffer1((uint8_t *)&CapSense_dsRam,
                     sizeof(CapSense_dsRam),
                     sizeof(CapSense_dsRam));

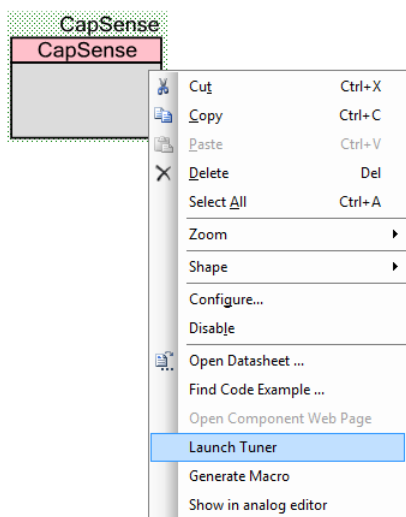
    CapSense_Start();                             /* Initialize Component */
    CapSense_ScanAllWidgets();                     /* Scan all widgets */

    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 */
            }

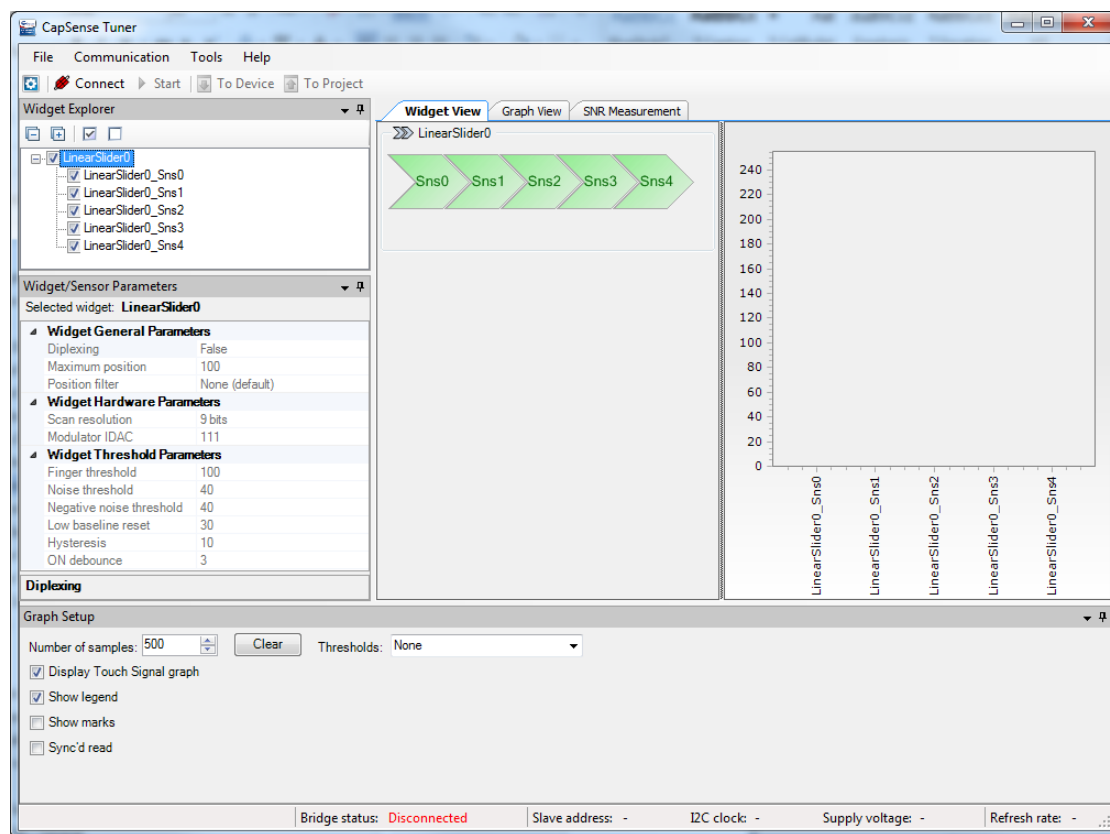
            CapSense_ScanAllWidgets();             /* Start next scan */
        }
    }
}
```


Step-4: Launch the Tuner Application

Right-click on 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, is automatically shown in the Widget View panel.



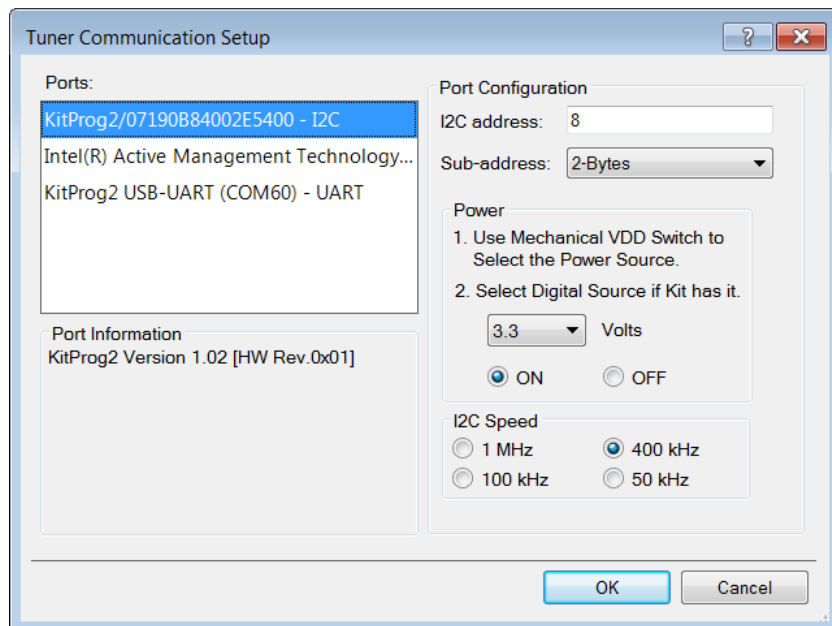
PRELIMINARY



Step-5: Configure Communication Parameters

In order to establish communication between the tuner and target device you must configure the tuner communication parameters to match those of the I2C Component.

1. Open the Tuner Communication Setup dialog by selecting *Tools > Tuner Communication Setup...* in the menu or clicking *Tuner Communication Setup* button.



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

Note The I2C address, Sub-address, and I2C speed fields in the Tuner communication setup must be identical to the Primary slave address, Sub-address size, and Data rate parameters in the EZI2C Component Configure dialog (see *Tuning Quick Start with EzI2C*). Sub-address must be set to 2-Bytes in both places.

Step-6: Start Communication

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

Check the *Sync'd read* control in *Graph Setup Pane*. This ensures that the Tuner only collects the data when CapSense is not scanning. Refer to *Graph Setup Pane* for details of 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 in the *Widget View* and signals for

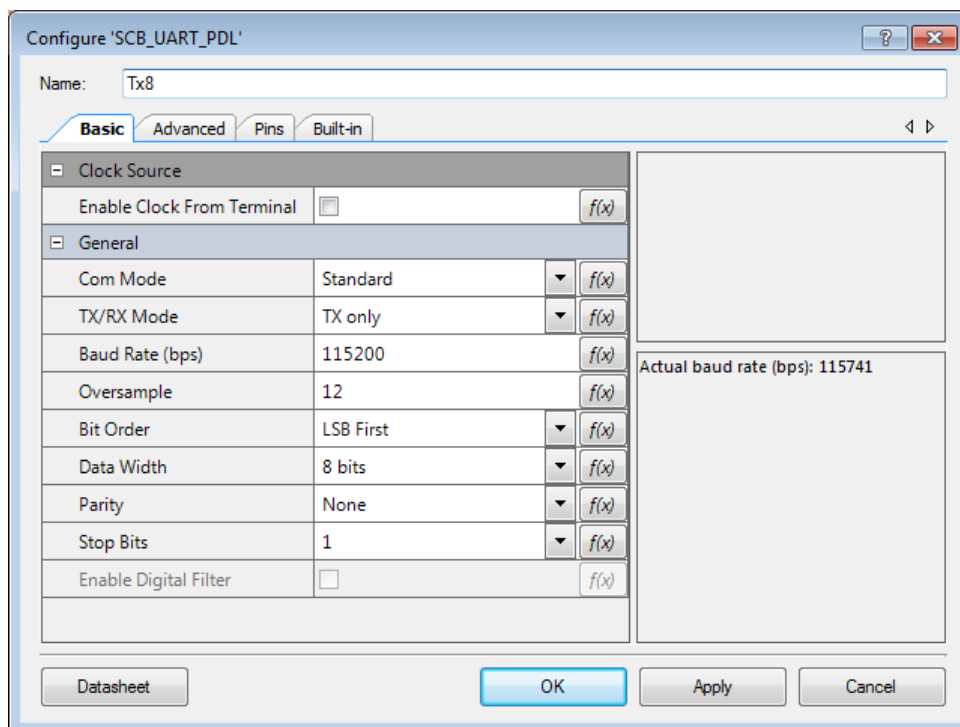
each of the five sensors in the [Graph View](#). Touch the sensors on the kit to observe CapSense operation.

Tuning Quick Start with UART

The following steps show how to set up CapSense tuning across an UART communication channel. These steps extend the application described in the [Quick Start](#) section.

Step-1: Place and Configure an UART (SCB) Component

1. Drag and drop the UART (SCB) Component from the Component Catalog onto the schematic to add an UART communication interface to the project. This UART interface is required for the Tuner GUI to monitor Component parameters in real time.
2. Double-click on the UART (SCB) Component.
3. On the UART (SCB) **Basic** tab, set the parameters as shown on the image:



- Type the desired Component name (in this case: Tx8).
- Set the TX/RX Mode to TX only. The CapSense Tuner allows only monitoring data received from a device and does not support sync'd read mode.
- Set the Data rate (bps) to 115200
- Set the Data Width to 8 bits

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

PRELIMINARY



Step-2: Assign Tx Pin in Pin Editor

Double-click the Design-Wide Resources Pin Editor (in the Workspace Explorer) and assign physical pin for the Tx8:tx.

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

Step-3: Modify Application Code

Replace your *main_cm4.c* from the [Step-3](#) in the [Quick Start](#) section with the following code:

```
#include <project.h>

uint8 header[] = {0x0D, 0x0A};
uint8 tail[] = {0x00, 0xFF, 0xFF};

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 */

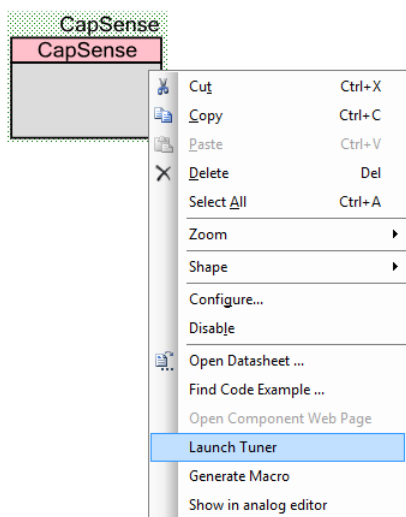
            /* Send packet header */
            Tx8_PutArrayBlocking((uint8 *)(&header), sizeof(header));
            /* Send packet with CapSense data */
            Tx8_PutArrayBlocking((uint8 *)(&CapSense_dsRam), sizeof(CapSense_dsRam));
            /* Send packet tail */
            Tx8_PutArrayBlocking((uint8 *)(&tail), sizeof(tail));

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

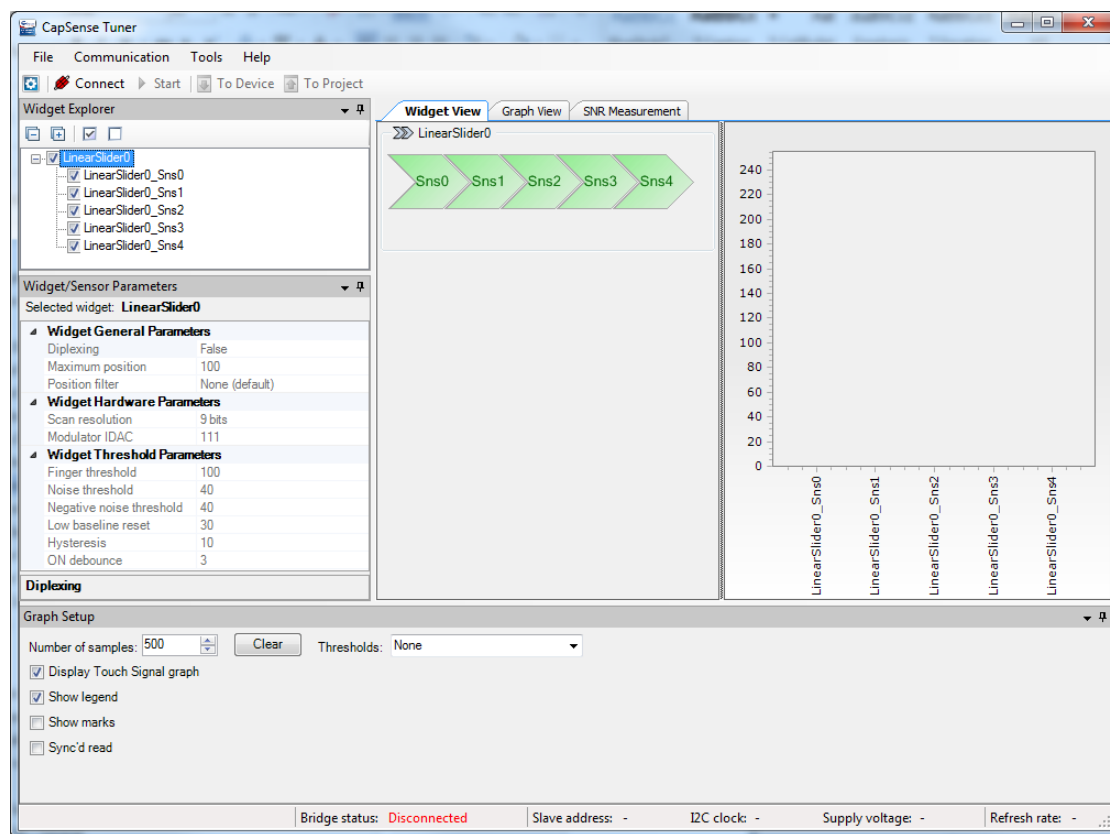
            CapSense_ScanAllWidgets(); /* Start next scan */
        }
    }
}
```

Step-4: Launch the Tuner Application

Right-click on 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, is automatically shown in the Widget View panel.



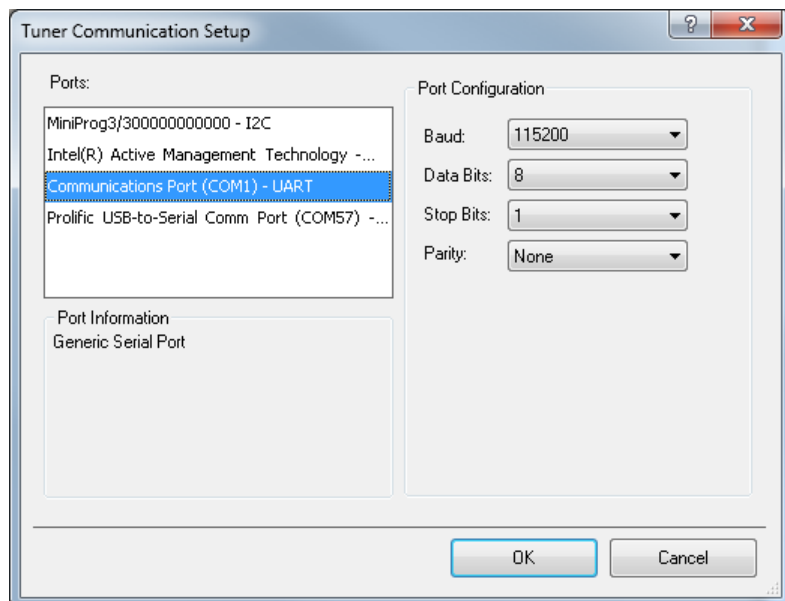
PRELIMINARY



Step-5: Configure Communication Parameters

In order to establish communication between the tuner and target device you must configure the tuner communication parameters to match those of the UART SCB Component.

1. Open the Tuner Communication Setup dialog by selecting *Tools > Tuner Communication Setup...* in the menu or clicking *Tuner Communication Setup* button.



2. Select the appropriate UART communication device 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

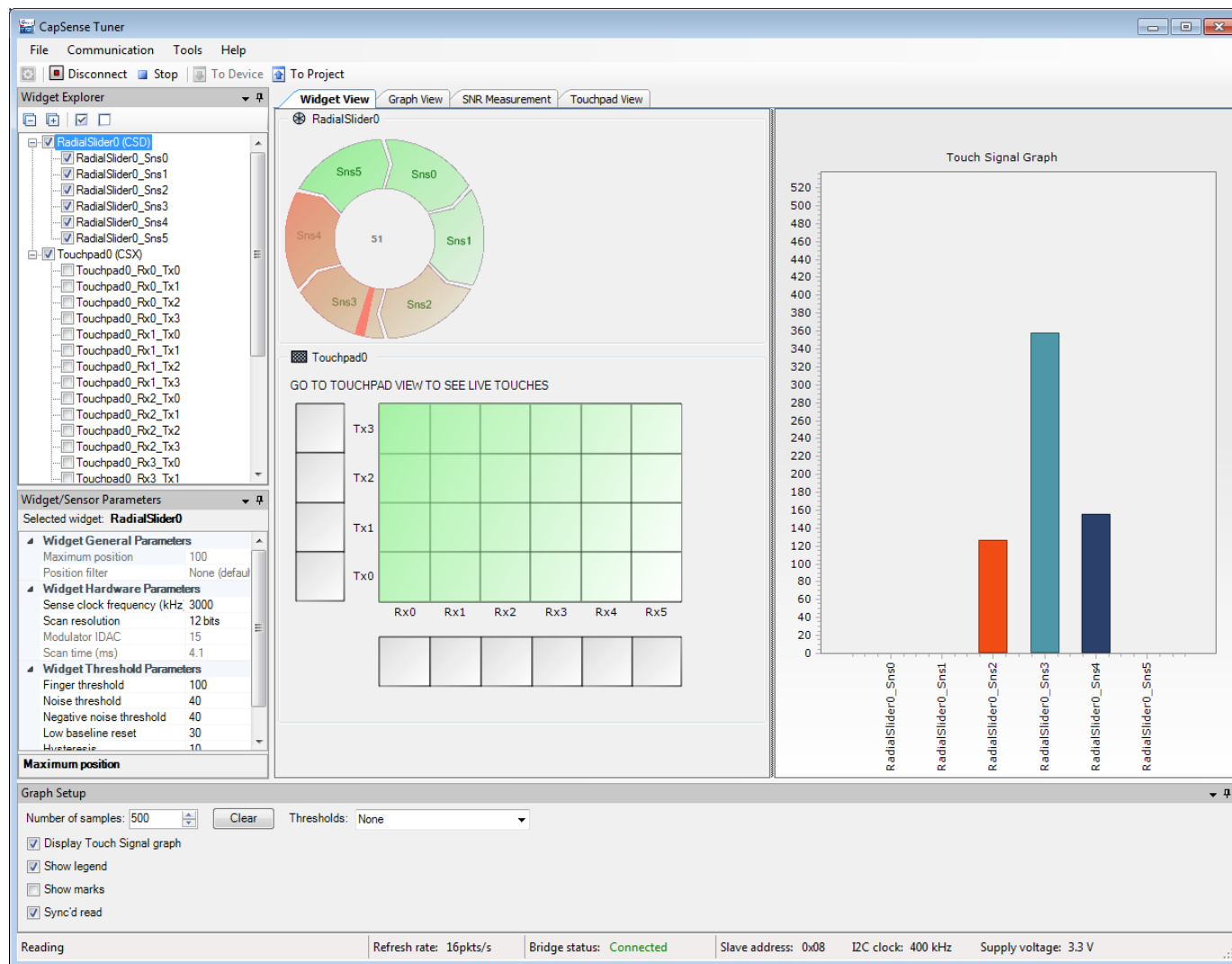
Click *Connect* to establish connection and then *Start* buttons to extract data.

The *Sync'd read* control in *Graph Setup Pane* is grayed out and is not available with UART communication. It means that Tuner are not able to write any data into device. Refer to *Graph Setup Pane* for details of 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 in the *Widget View* and signals for

each of the five sensors in the *Graph View*. Touch the sensors on the kit to observe 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.

PRELIMINARY



Menus







The main menu provides the following commands to help control and navigate the Tuner:

- **File > Apply to Device (Ctrl + D)** – Commits the current values of the widget/sensor parameters to the device. This menu item becomes active if a value of any configuration parameter is changed from the Tuner UI (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 [Tuner Parameter Saving Flow](#) 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 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 is shown.
- **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 communication protocol (not *CapSense_dsRam* / wrong header-tail of packet at UART communication)
 - The latest customizer parameters modification was not programmed into device.
 - Edit performed in the customizer during tuning session: the Tuner needs to be closed and opened again after the customizer update.
 - The Tuner 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 **File > Apply to Device** menu item.
-  – Duplicates the **File > Apply to Project** menu item.

Status bar

The status bar displays various information related to the communication state between the Tuner and the device. This includes:

- **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** – 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.

PRELIMINARY







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 on the [Widget View](#), while the sensor checked status controls the visibility of the sensor raw count / baseline / signal / status graph series on the Graph View and signals on the [Touch Signal Graph](#) on the [Widget View](#).

Selection of widget or sensor in the [Widget Explorer Pane](#) updates the selection in the [Widget/Sensor Parameters Pane](#). It is possible to select multiple widget or sensor nodes to edit multiple parameters at once. For example, you can edit the Finger Threshold parameter for all widgets at once.

Note For CSX widgets, the sensor tree displays individual nodes (Rx0_Tx0, Rx0_Tx1 ...) as 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 at once, 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 includes:

- **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 \(Hardware parameters only\)](#) in the CapSense customizer. In the [Manual](#) tuning mode (for both CSD and CSX widgets), any change to [Widget Hardware Parameters](#) requires hardware re-initialization which can be performed only if the Tuner communicates with the device in Synchronized mode.
- **Widget Threshold Parameters** – Cannot be modified for the CSD widgets when the [CSD tuning mode](#) is set to [SmartSense \(Full Auto-Tune\)](#) in the customizer. In the [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** – Sensors-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 customizer is shown. If the Tuner operates in *Sync'd read*, it is possible to edit the value and apply it to the device.

Graph Setup Pane

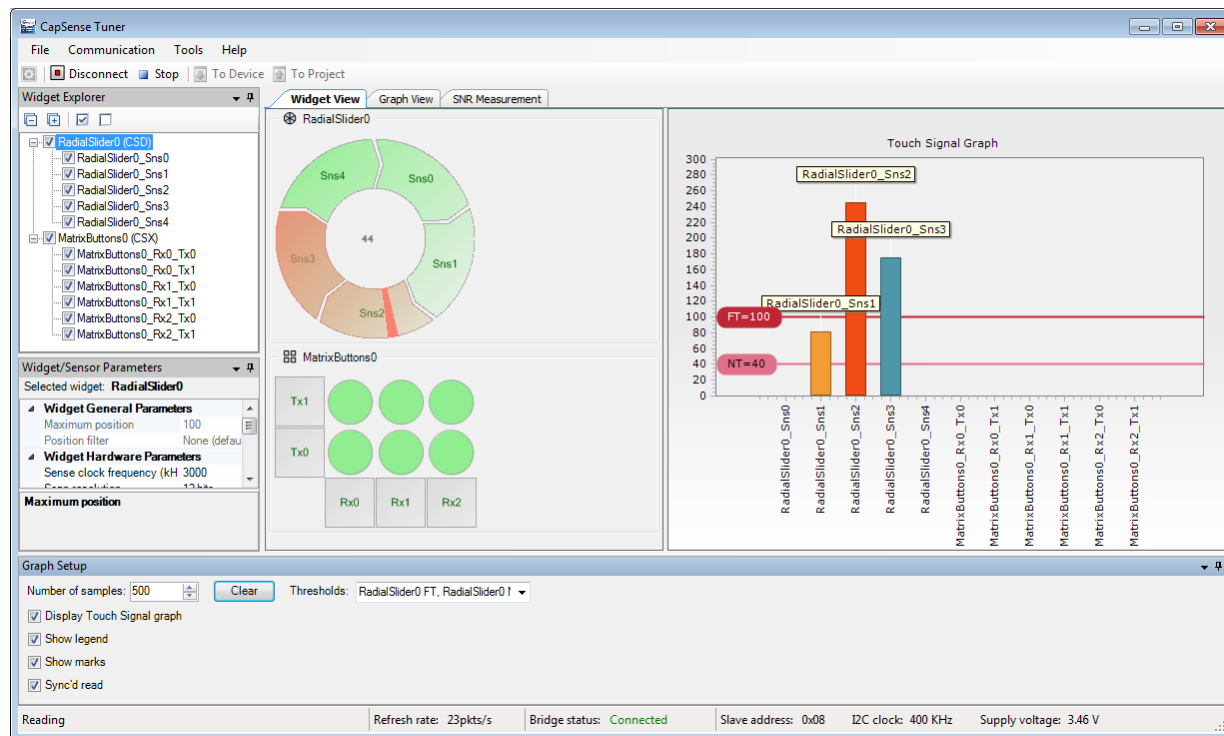
The Graph Setup pane provides quick access to different Tuner configuration options that affect the Tuner graphs display.

- **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) on graphs when checked (Sensor Data/Sensor Signal/Status graphs on a *Graph View* and *Touch Signal Graph* on a *Widget View*).
- **Show marks** – When checked, the sensor names are shown as marks over the signal bars on *Touch Signal Graph*.
- **Sync'd read** – Controls the communication mode of the Tuner. The Sync'd read mode is available when a FW loop periodically calls a corresponding Tuner API.

When unchecked, the Tuner reads data asynchronously to sensor scanning. Because reading data by CapSense Tuner and data processing happen asynchronously, it is possible that CapSense Tuner will 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 reads the data. As a result, the second sensor data is not processed.

When the synchronized read mode is enabled, the CapSense Tuner manages an execution flow by suspending scanning during read operation. Before starting data reading, the Tuner sends a **Suspend** command to the device. The device hangs the FW flow until a **Resume** command is received. The Tuner reads all the needed data and sends a **Resume** command. The device restores operation by executing the next scan.

Widget View



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

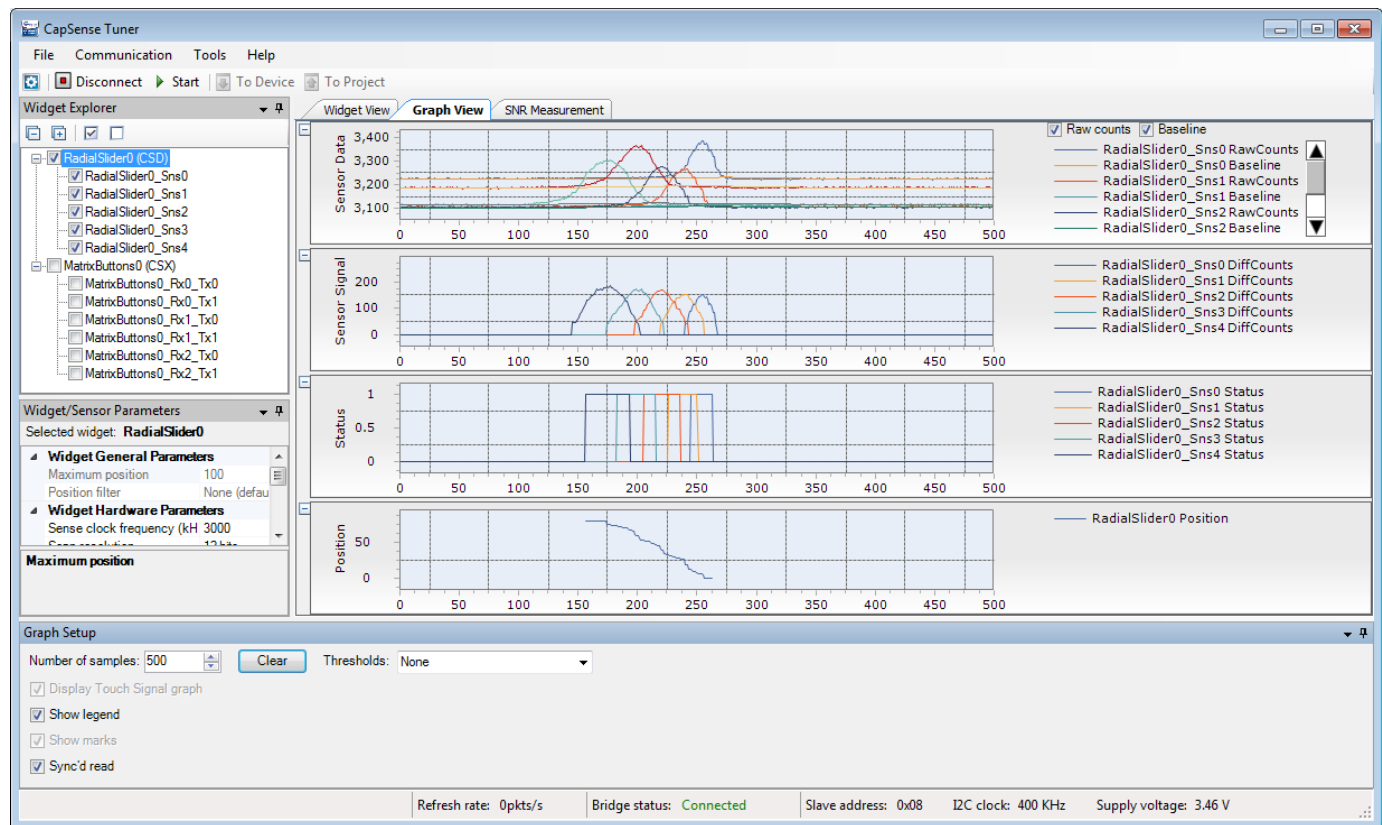
The Widget sensors are highlighted red when the device reports their touch status as active.

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 that is selected in the *Widget Explorer Pane*.

Graph View



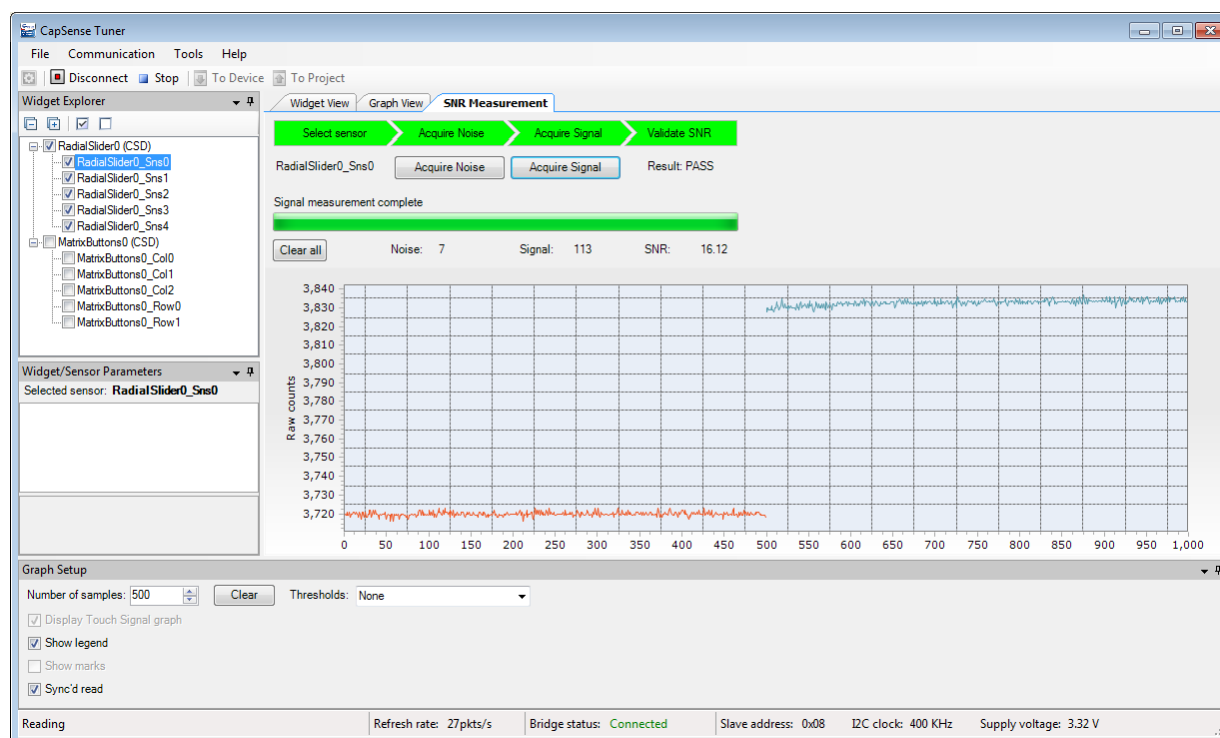
Displays graphs for selected sensors in the *Widget Explorer Pane*. The following charts are available:

- **Sensor Data graph** – Displays raw counts and baseline. It is possible to select which series should be displayed with the checkboxes on the right:
 - ☐ Raw counts and baseline series
 - ☐ Raw counts only
 - ☐ Baseline only
- **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.

PRELIMINARY



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 based 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

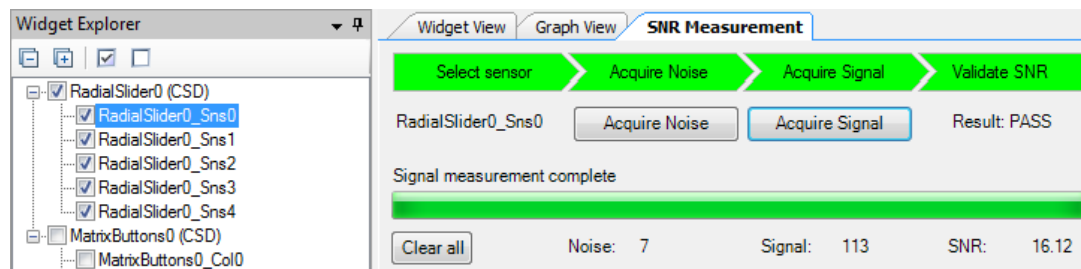
1. Connect to the device and start communication (by pressing the **Connect**, then **Start** buttons on the toolbar).
2. Switch to the **SNR Measurement** tab.
3. Select a sensor in the [Widget Explorer Pane](#) located at the left of the **SNR Measurement** tab.
4. Make sure no touch is present on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



PRELIMINARY

10. Observe the SNR label is updated with the 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** is green when there is a sensor selected; gray otherwise.
- **Acquire noise** is green when noise samples are already collected for the selected sensor; gray otherwise.
- **Acquire signal** is green when signal samples are already collected for the selected sensor; gray otherwise.
- **Validate SNR** is 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 bar, there are the following controls:
- **Sensor name** label selected in the *Widget Explorer Pane* or None (if no sensor selected).
- **Acquire Noise** is a button 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** is a button 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** is a label that shows either “N/A” (when the SNR cannot be calculated due to noise/signal samples not yet collected), “PASS” (when SNR is above the required limit), or “FAIL” (when the SNR is below the required limit).

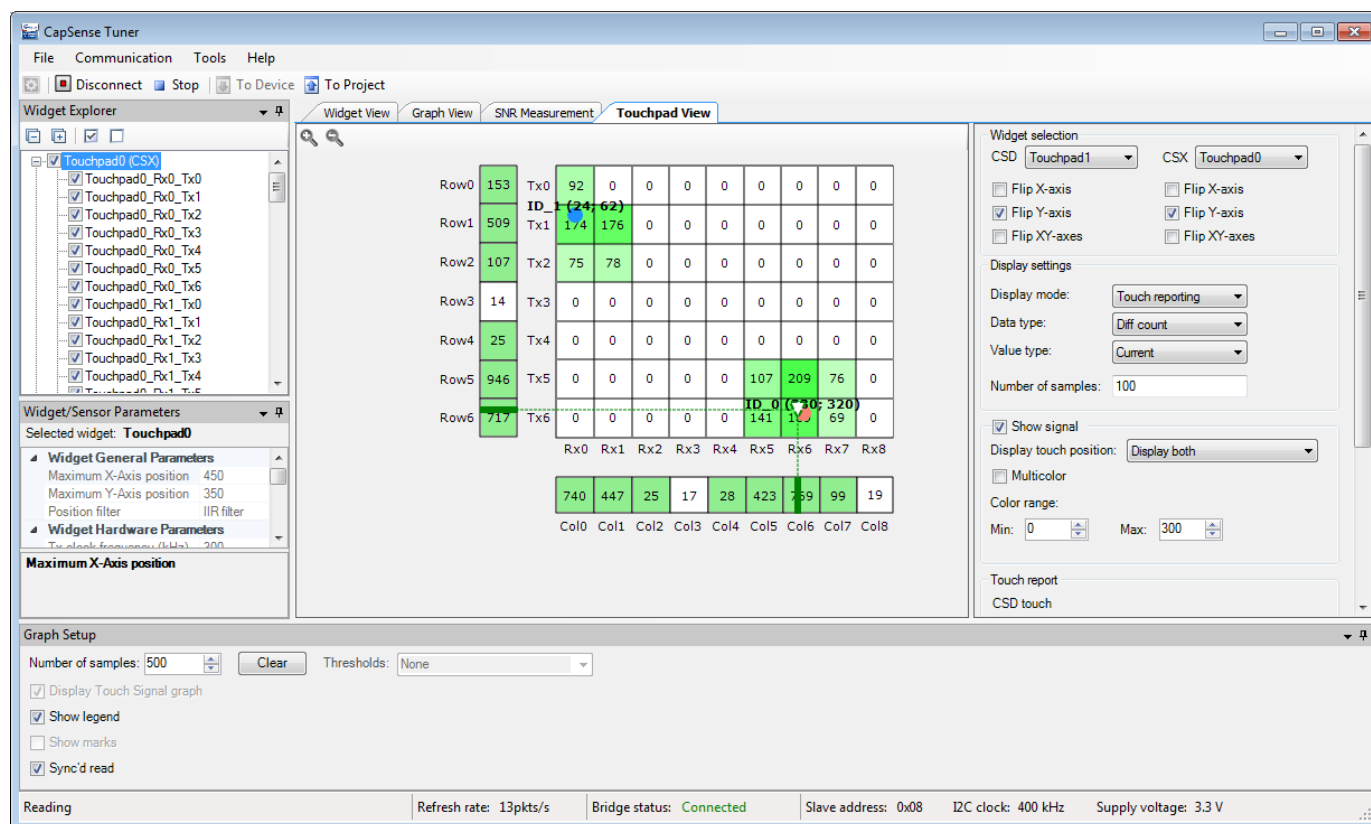
Below these, there is a status label displaying the current status message and progress bar displaying the progress of the current operation.

At the bottom of the control area, there are the following controls:

- **Clear all** is a button that allows clearing all measured data. When data acquisition is in progress, the operation is restarted (i.e. all samples collected so far are discarded, and measurement is started from scratch).
- **Noise** is a 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** is a label that shows the signal average value calculated during the last signal measurement for the selected sensor, or “N/A” if no signal measurement was performed yet.
- **SNR** is a label that shows a calculated SNR value. This is the result of Signal/Noise division rounded up to 2 decimal points. When a SNR cannot be calculated, “N/A” is displayed instead.

Touchpad View

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



The following options are available:

Widget Selection

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

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

Display settings

Manages heatmap data that should 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 a type of the 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.

PRELIMINARY



Show signal

Enables displaying data for each sensor if checked, otherwise displays only touches. It is applicable for the CSX touchpad only.

- **Display touch position** – Defines positions from which the touchpad is displayed. The three options available:
 - ☐ Display only CSX
 - ☐ Display only CSD
 - ☐ Display both
- **Multicolor** – When the checked heatmap uses the rainbow color palette to display sensor signals, otherwise 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.

CSD touch table

Displays the current X and Y touch position of the CSD touchpad configured in **CSD combobox**. If the CSD touchpad is neither configured nor touch-detected, the touch table is empty.

CSX touches table

Displays the X, Y, and Z values of the detected touches of the CSX touchpad configured in **CSX combobox**. 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.

Clear

Clears all before drawn elements like lines, traces, etc.

Tuner Parameter Saving Flow

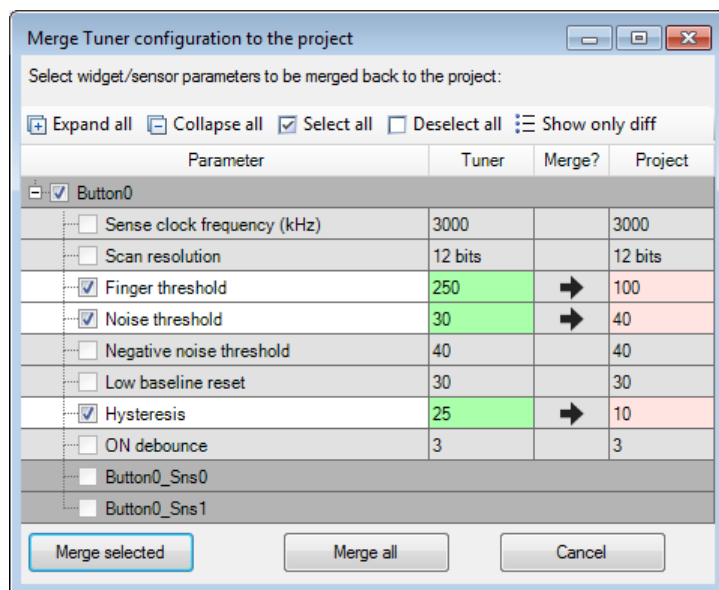
Changes to widget/sensor parameters made during the tuning session are not automatically applied to the PSoC Creator project. Follow these steps to merge parameters modified by the tuner back to the Component instance:

1. Close the tuner application.
2. Whenever there are some changes not yet saved to the project (with the “Apply to Project” button), the dialog “Do you want to save the updated CapSense parameters?” appears. Accept the dialog.
3. Open the Component customizer GUI.



PRELIMINARY

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



4. Click the **Merge all** or **Merge selected** buttons to apply the Tuner changed parameters to project. Click the **Cancel** button to leave the Customizer parameters unchanged.

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

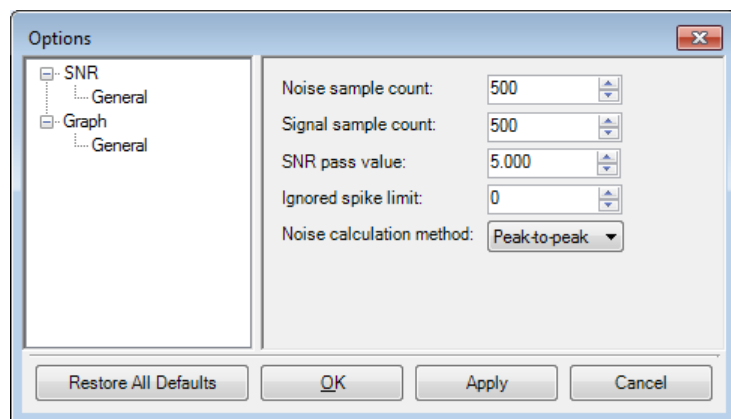
- *SmartSense Auto-tuning*
- *CSD Enable IDAC auto-calibration*
- *CSX Enable IDAC auto-calibration*

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

Tuner Configuration Options

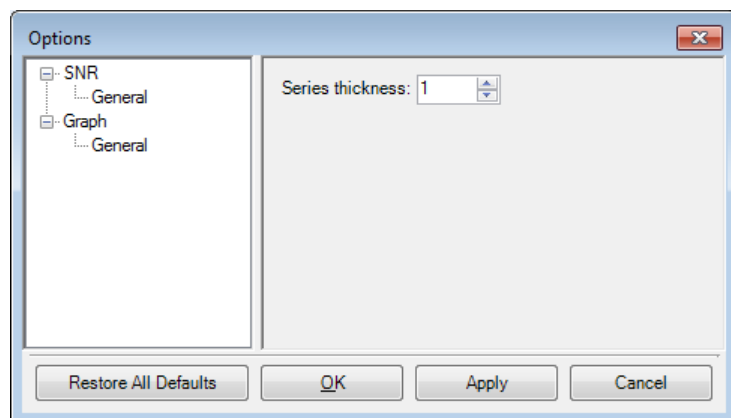
The Tuner application allows setting different configuration options with the Options dialog. Settings are applied on a project basis and divided into groups:

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.

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 the 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 Component has the following specific deviations:

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
1.1	R	Number of macro definitions exceeds 1024 - program does not conform strictly to ISO:C90.	The Component generates macros for every registers of CapSense Data Structure. For many configurations numbers of macros can exceeds 1024 (ISO90)
11.3 (3.1)	A (R)	Cast between a pointer to object and an integral type.	The cast from unsigned int to pointer does not have any unintended effect, as it is a consequence of the definition of a structure based on hardware registers.
11.4	A	Cast from a pointer to void to a pointer to object type.	The architecture of the Component is intended for this casting to simplify the operation with pointers.

PRELIMINARY



MISRA-C:2004 Rule	Rule Class (Required/Advisory)	Rule Description	Description of Deviation(s)
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.
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 compiler'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 of the Component is intended for this casting.
17.4	R	Performing pointer arithmetic.	The architecture of the Component is intended to perform pointer arithmetic to achieve the best performance. It cannot be avoided, so are manually checked and reviewed to be safe.
18.4	R	Unions shall not be used.	There are two general cases in the code where this rule is violated. 1. CapSense_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.
21.1	R	Minimisation of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	The component contains redundant operations introduced because of generalized implementation approach.

Resources

The CapSense Component consumes one CSD (CapSense Sigma-Delta) block, two Analog Mux bus, 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.

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

- Only 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.

References

General References

- [Cypress Semiconductor web site](#)
- PSoC 6 Device datasheets – the link is not available yet

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
- [AN92239](#) – Proximity Sensing with CapSense®

Code Examples

PSoC Creator provides access to code examples in the 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 "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](#).

PRELIMINARY



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-062-BLE PSoC® 6 Pioneer Kit – the link is not available yet

DC and AC Electrical Characteristics

Specifications are valid for +25°C, VDD 3.3V, Cmod = 2.2nF, Csh = 10nF, and CintA = CintB = 470pF except where noted.

Note Final characterization data for PSoC 6 devices is not available at this time. Once the data is available, the Component datasheet will be updated on the Cypress web site.

CapSense Performance Characteristics (**Preliminary**)

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 10 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 cap capacitance ≥ 0.1 pF	> 5:1	-
	Cp < 45 pF Single IDAC: Finger capacitance ≥ 0.2 pF Dual IDAC: Finger cap 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	



PRELIMINARY

Parameter	Condition	Typical	Units
	$V_{DD} < 2\text{ V}$, externally regulated mode, Finger capacitance = 0.4 pF, V_{DD} ripple +/-25 mV	< 30% of noise	
GPIO Sink Current	10 mA per GPIO on multiple pin to sink max current. Device max = 25 mA for all other PSoCs	< 30% of noise	
Tx Output Voltage	Logic High	$> V_{DD} - 0.6$	V
	Logic Low	< 0.6	V
Reference Voltage (Vref) (CSD sensing method)	$V_{DDA} < 2.2\text{ V}$	1.160	V
	$2.2\text{ V} \leq V_{DDA} < 2.75\text{ V}$	1.600	V
	$2.75\text{ V} \leq V_{DDA}$	2.130	V
Reference Voltage (Vref) (CSX sensing method)		1.160	V
Finger-Conducted AC Noise This is the change in the sensor raw count when AC noise is applied to the sensor (injected into the system)	50/60 Hz, noise $V_{pp} = 20\text{ V}$	< 30%	-
	10 kHz to 1 MHz, noise $V_{pp} = 20\text{ V}$, $C_p < 10\text{ pF}$	< 30%	-
Interrupt immunity Excessive raw counts noise at asynchronous interrupts is used.	SNR > 5:1	< 30%	-
Current Consumption	1 CSD Button Widget (Ganged Sensor, 4 electrodes). Resolution = 9 bits. Each electrode $C_p < 10\text{ pF}$. Shield Electrode = Disabled. SYSCLK = 16 MHz. No I2C traffic (I2C block ON). Report Rate $\geq 8\text{ Hz}$. Chip state = DeepSleep (LFT).	TBD	μA
	1 CSD Button Widget, 8 Sensors. Resolution = 9 bits. Each electrode $C_p < 10\text{ pF}$. Shield Electrode = Disabled. SYSCLK = 16 MHz. No I2C traffic (I2C block ON). Report Rate $\geq 8\text{ Hz}$. Chip state = DeepSleep (LFT).	TBD	μA

PRELIMINARY



Parameter	Condition	Typical	Units
	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).	TBD	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. 9mm metal finger. 1 Touch only. Report Rate >= 8 Hz. Chip state = DeepSleep (LFT).	TBD	μA

ADC Performance Characteristics (**Preliminary**)

Parameter	Min	Typ	Max	Units	Details/ Conditions
Reference Voltage (Vref)	-	1.160	-	V	VDDA < 2.2 V
	-	1.600	-	V	2.2 V ≤ VDDA < 2.75 V
	-	2.130	-	V	2.75 V ≤ VDDA
Resolution	-	-	10	bits	Auto-zeroing is required every millisecond
Number of channels - single ended	-	-	10		
Monotonicity	-	-	-	-	Yes
Gain error	-	-	±2	%	In V _{REF} (2.4 V) mode with V _{DDA} bypass capacitance of 10 μF
Input offset voltage	-	-	3	mV	In V _{REF} (2.4 V) mode with V _{DDA} bypass capacitance of 10 μF
Current consumption	-	-	0.25	mA	



PRELIMINARY

Parameter	Min	Typ	Max	Units	Details/ Conditions
Input voltage range - single ended	V_{SSA}	-	V_{DDA}	V	
Input resistance	-	2.2	-	K Ω	
Input capacitance	-	20	-	pF	
Power supply rejection ratio	-	60	-	dB	In V_{REF} (2.4 V) mode with V_{DDA} bypass capacitance of 10 μ F
Sample acquisition time	-	10	-	μ s	
Conversion time for 8-bit resolution at clock frequency = 48 MHz.	-	-	10.7	μ s	Does not include acquisition and processing time.
Conversion time for 10-bit resolution at clock frequency = 48 MHz.	-	-	42.7	μ s	Does not include acquisition and processing time.
Signal-to-noise and Distortion ratio (SINAD)	-	61	-	dB	With 10Hz input sine wave, external 2.4V reference, V_{REF} (2.4 V) mode
Input bandwidth without aliasing	-	-	22.4	KHz	8-bit resolution
Integral Non Linearity. 1 KSPS	-	-	2	LSB	$V_{REF} = 2.4$ V or greater
Differential Non Linearity. 1 KSPS	-	-	1	LSB	

IDAC Characteristics (**Preliminary**)

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	

DC/AC Specifications

Refer to devices specific datasheet PSoC 6 Device datasheets for more details.

PRELIMINARY



Component Changes

This section lists the major changes in the Component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.0.a	Datasheet edits	Updated Features and General Description sections. Updated various screen captures and descriptions.
1.0	The initial version of new Component.	CapSense Component with PSoC 6 devices support.

© Cypress Semiconductor Corporation, 2016-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

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

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



PRELIMINARY