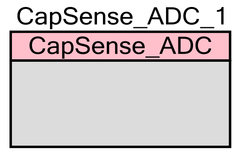
PSoC 4 Capacitive Sensing (CapSense®) ADC

4.10



Features

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

Note This CapSense_ADC v4.X Component (and any version that follows) is a new Component and it is **not** backward-compatible with CapSense_CSD v2.X or older. If you are working on a project with an older Component, Cypress recommends backing it up before replacing the Component in your schematic. The C code written for CapSense_CSD_P4 is not compatible with the new Component. Refer to the *Migration Guide* in this datasheet for details on how the two Components are different.

General Description

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

PRELIMINARY

This CapSense_ADC 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.
- *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.
- *Migration Guide* – Helps to manually transition designs from CapSense_CSD_P4 v2.X or older versions to CapSense_ADC v4.X.

Note Important information such as the CapSense-technology overview, appropriate Cypress device for the design, CapSense system and sensor design guidelines, as well as different interfaces and tuning guidelines necessary for a successful design of a CapSense system is available in the *Getting Started with CapSense®* document and the product-specific 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_ADC 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 Fourth-generation CapSense PSoC 4 family of devices: PSoC 4000S, PSoC 4100S, PSoC 4100PS, and PSoC Analog Coprocessor. However, the *CapSense Tuner* does not support ADC functionality. This feature 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 CapSense with ADC

This section will help you create a PSoC Creator project with a *Button* interface using the CSD *Sensing Mode* and a button to start an analog measurement. In order to monitor performance of the sensor using the *CapSense Tuner*, refer to the *Tuning Quick Start* section once the basic button project has been created.

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_ADC-enabled PSoC 4 device (*Fourth-generation CapSense*) from the drop-down menu in the New Project wizard.

Step-2: Place and Configure the CapSense Component

Drag and drop the CapSense_ADC Component from the Component Catalog onto the design to add the Component's functionality to the project.

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

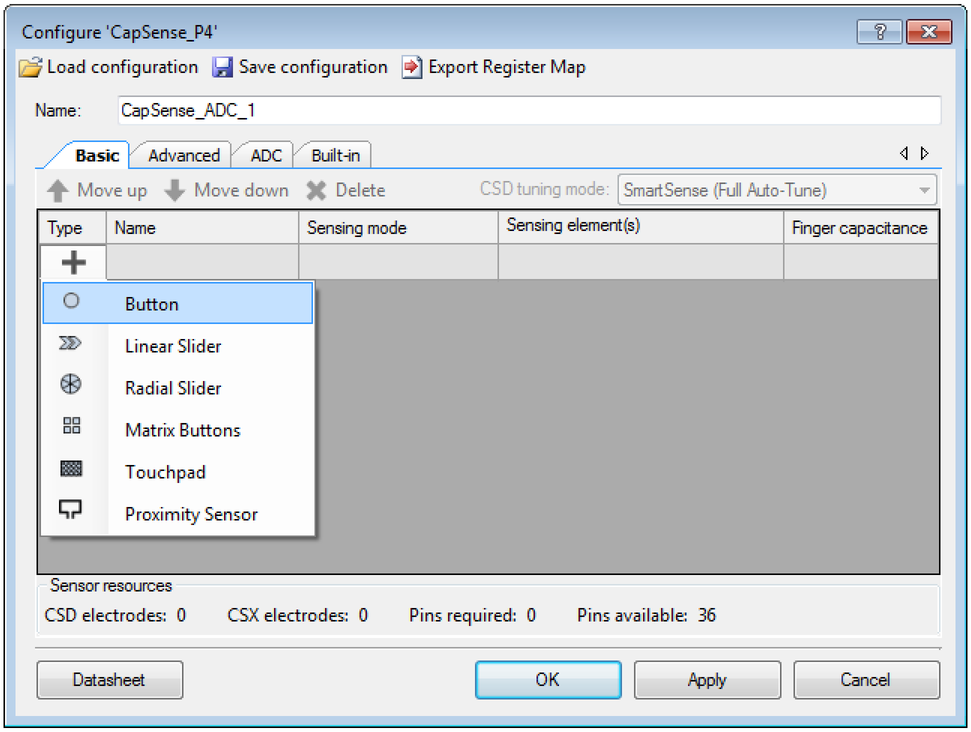
The *Component Configuration Parameters* are arranged over multiple tabs and sub-tabs.

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-6* to work).

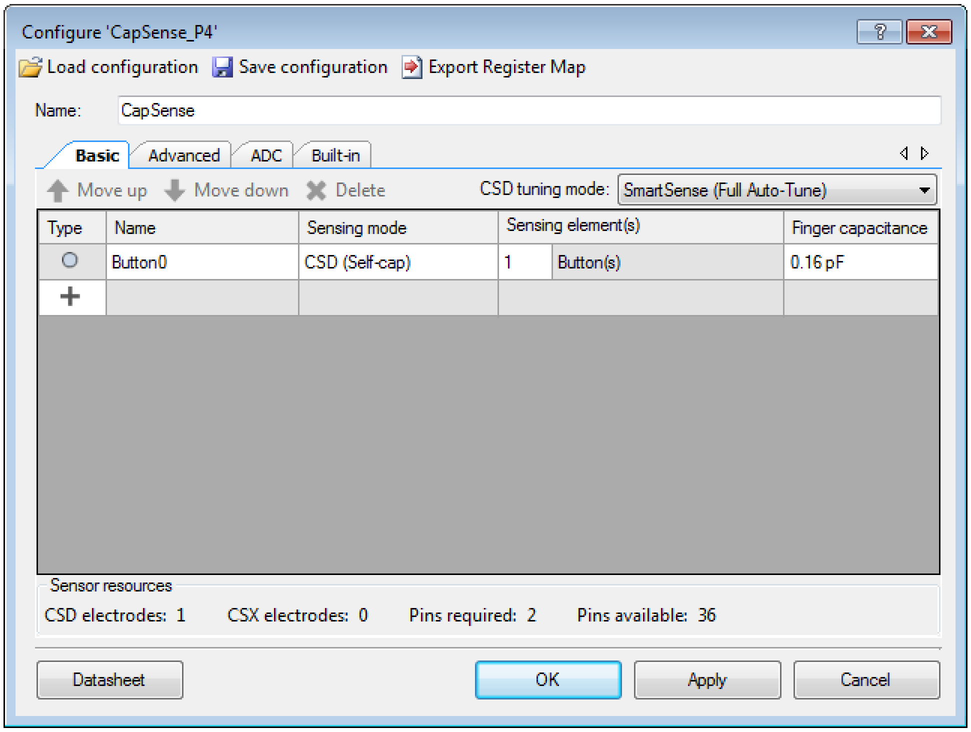
- Click '+' and select the *Widget Type* required from the drop-down list. This Component offers six different types of widgets.



- Add the *Button* 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 Button0 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). Refer to section "CapSense Technology" of *PSoC® 4 CapSense® Design Guide* for more details of these sensing methods.
- 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 value 1 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 the *SmartSense Auto-tuning* mode is enabled.

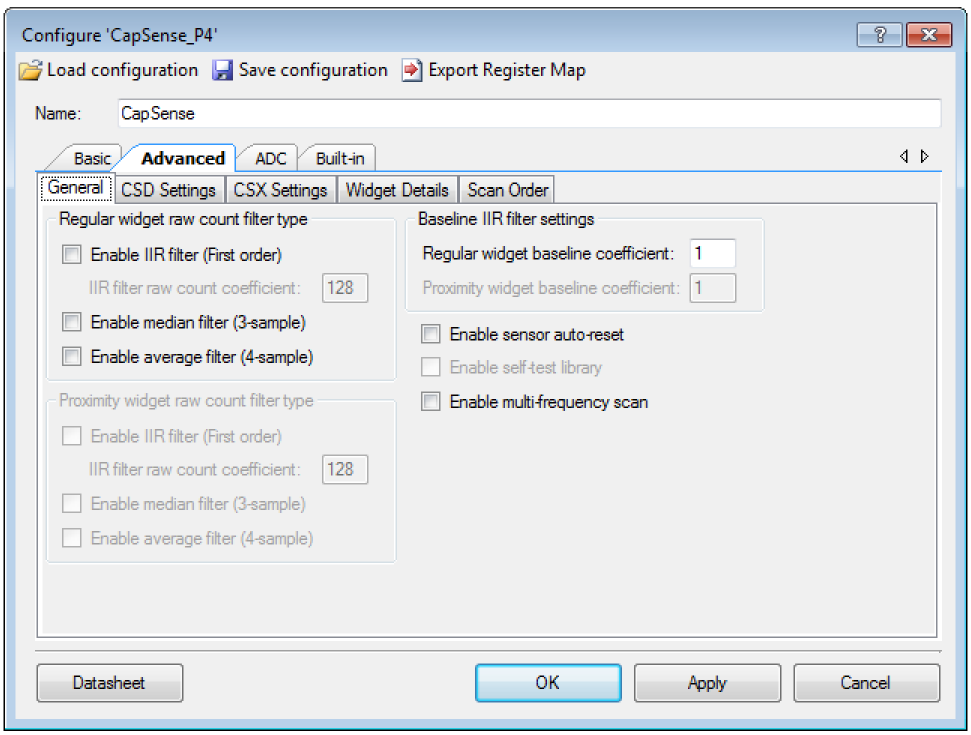
Use the *CSD tuning mode* pull-down (above the table) to select one of the following options:

- *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.

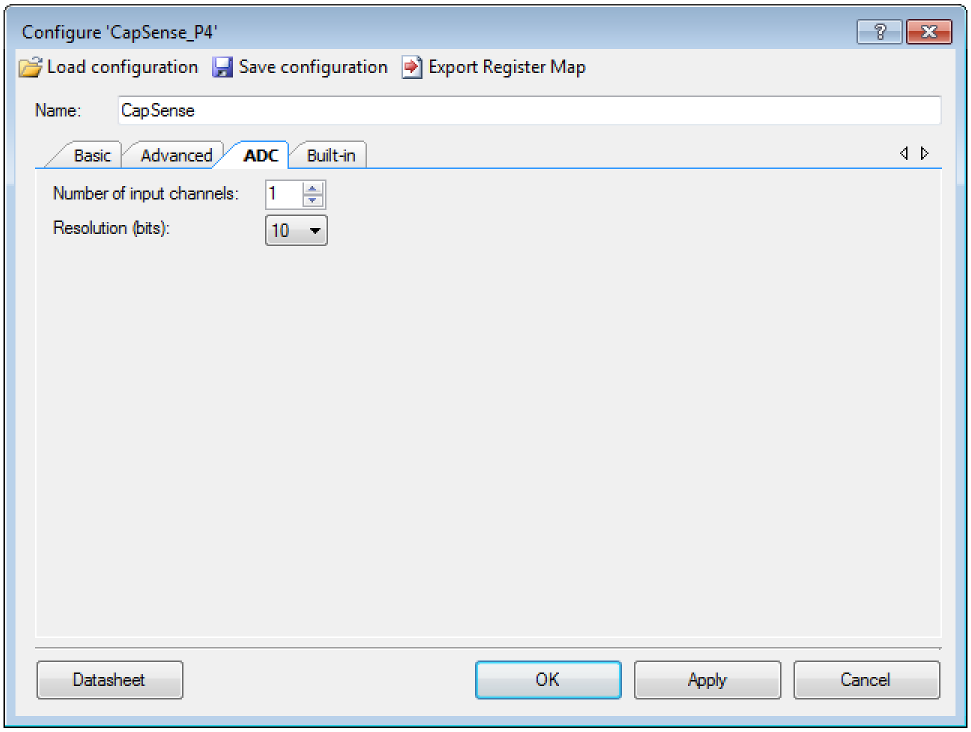


- *General* – This sub-tab contains the parameters common for all widgets in the Component.
- *CSD Settings* – This sub-tab contains all the parameters common for all CSD widgets.
- *CSX Settings* – This sub-tab contains all the parameters common for all CSX widgets.

- *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.

ADC Tab

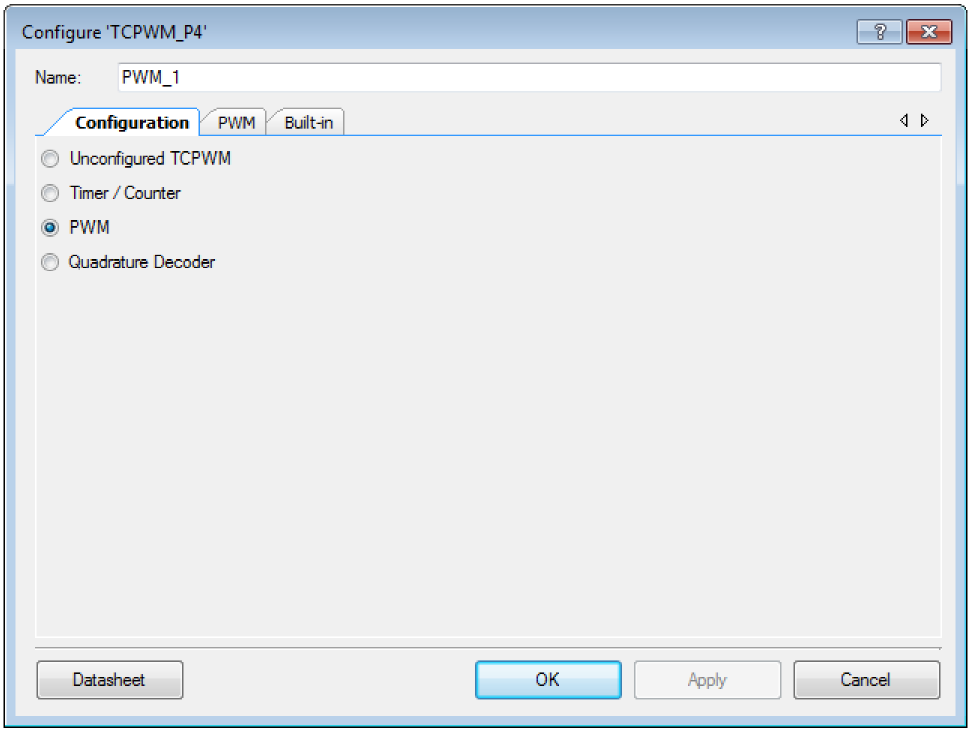
Use this tab to configure parameters required for using the analog-to-digital converter. This tab has two parameters:



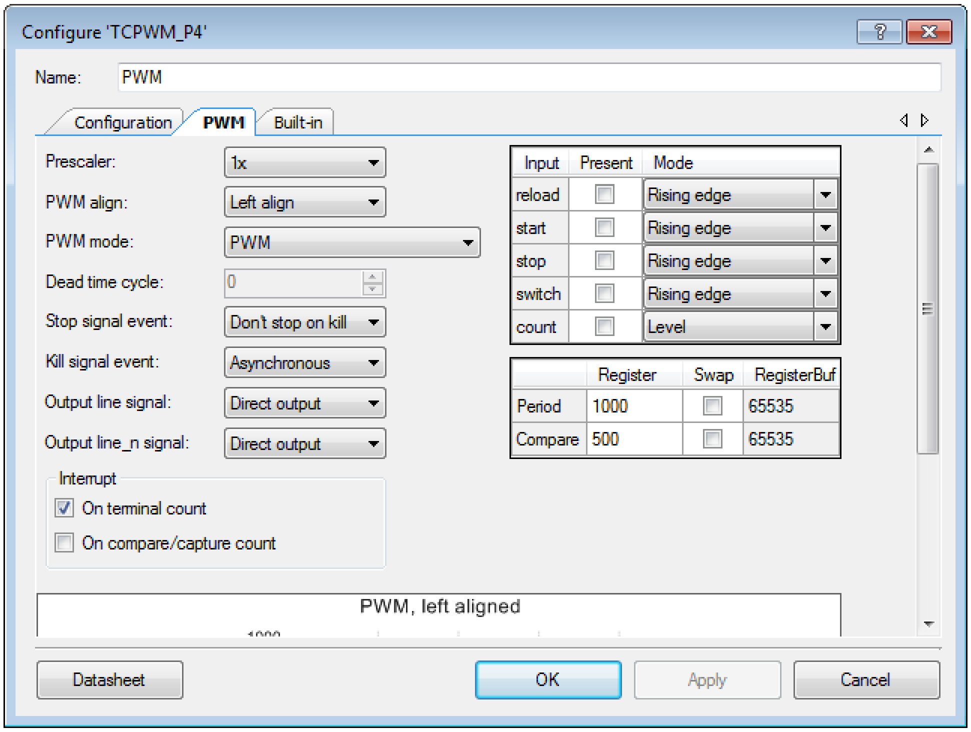
- Number of input channels – This is the number of ADC inputs. Up to ten (10) inputs are possible. The default value of one (1) is used in this quick start.
- Resolution (bits) – This is the precision of ADC measurements. The Component supports 8-bit and 10-bit resolutions. The default value of 10-bits is used in this quick start.

Step-3: Place and Configure a Timer Counter PWM (TCPWM) Component

1. Drag and drop the TCPWM Component from the Component Catalog onto the design.
2. Double-click on the TCPWM Component. You will see the **Configuration** tab.



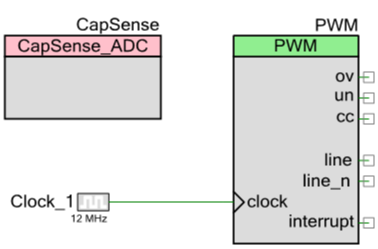
3. On the **PWM** tab, set the following parameters.



- Type the desired Component name (in this case: *PWM* for the code in Step-6 to work).
- Set the Period Register to 1000
- Set the Compare Register to 500

Step-4: Place Clock Component

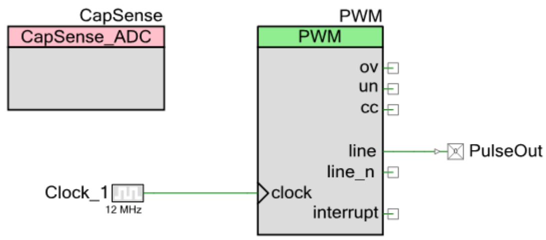
1. Drag and drop the Clock Component from the Component Catalog onto the design.
2. Connect it to the TCPWM's *clock* terminal using a wire.



Step-5: Place and Configure Digital Output Pin Component

1. Drag and drop the Digital Output Pin Component from the Component Catalog onto the design.
2. Connect it to the TCPWM's *line* terminal using a wire.

3. Double click on the Digital Output Pin Component and rename it as desired (in this case: *PulseOut* for the pins in the Pin Editor in *Step-7*.



Step-6: Write Application Code

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

```
#include <project.h>

int main()
{
    uint16 adcResult;

    CyGlobalIntEnable;                          /* Enable global interrupts */

    PWM_Start();                                /* Initialize the TCPWM */
    CapSense_Start();                           /* Initialize the CapSense */
    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 */
            {
                adcResult = CapSense_AdcReadResult_mVolts(CapSense_AdcCHANNEL_0);
                PWM_WriteCompare((1000u * adcResult) / CYDEV_VDDA_MV);
            }
            CapSense_ScanAllWidgets();          /* Start next scan */
        }
    }
}
```

Note The provided example shows the simplest way to use the Component. Another way to use the Component is shown in *CE210311 - CapSense® ADC Sequential* code example.

Step-7: 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-8: Connect External Components to PSoC Device

1. Connect an oscilloscope or LED to the pin associated with the *PulseOut* Component.
2. Connect a voltage source to the pin associated with the *\CapSense:AdcInput\ (Ch0)*.

Step-9: Build Design and Program PSoC Device

1. Select ***Build <project name>*** from the ***Build*** menu.
2. Select ***Program*** from ***Debug*** menu to download the hex file to the device.

Step-10: Observe Project's Operation

The *PulseOut* signal will start with a 50% duty cycle. When the button is touched, an ADC measurement is performed, and the duty cycle is scaled to the chip's VDDA.

Input / Output Connections

These I/Os are not exposed as connectable terminals on the Component symbol. However, these I/Os can be assigned to the port pins in the Design-Wide Resources Pin Editor. The **Pin Editor** provides the guidelines on the recommended pins for each I/O and does not allow an invalid pin assignment.

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

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.

Name [1]	I/O Type	Description
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.

Component Configuration Parameters

This section provides descriptions 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 *PSoC® 4 CapSense® Design Guide*.

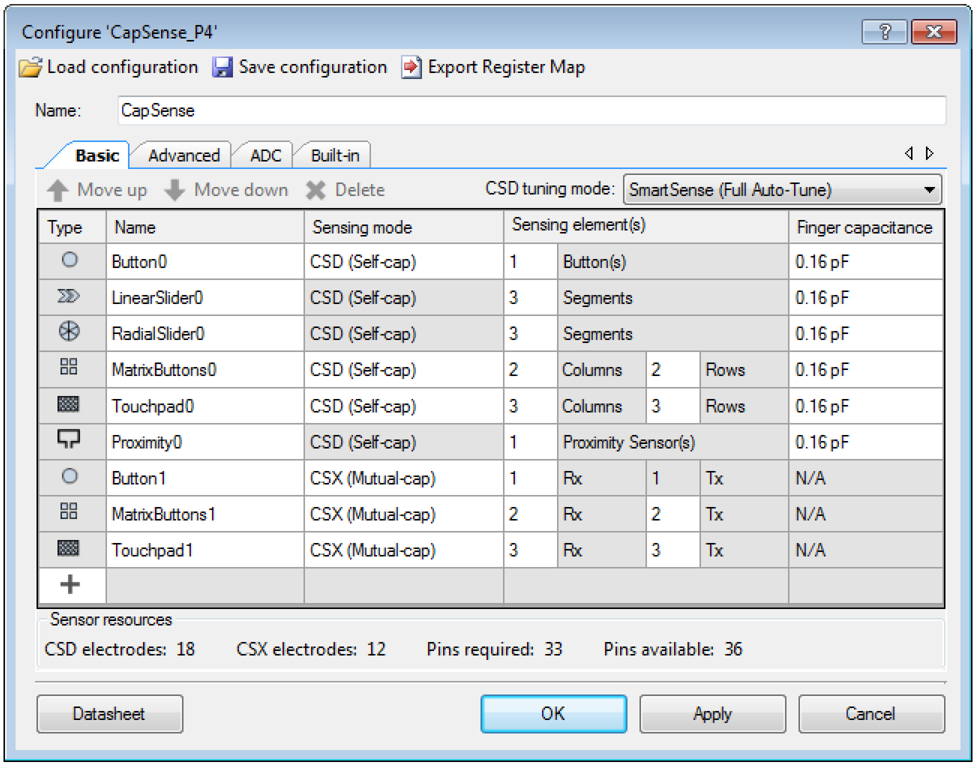
Drag a Component onto the design canvas and double-click to open the dialog.

Common Controls

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



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

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>CSD tuning mode</td><td>
Tuning is a process of finding appropriate values for configurable parameters (Hardware parameters and Threshold parameters) for proper functionality and optimized performance of the CapSense system.
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.
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.
This parameter is a drop-down box to select the tuning mode for CSD widgets only.

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 Set by SmartSense mode. In this mode, the following parameters are automatically tuned:

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

SmartSense (Hardware parameters only) – the Hardware parameters are automatically set by the Component, 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:

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

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

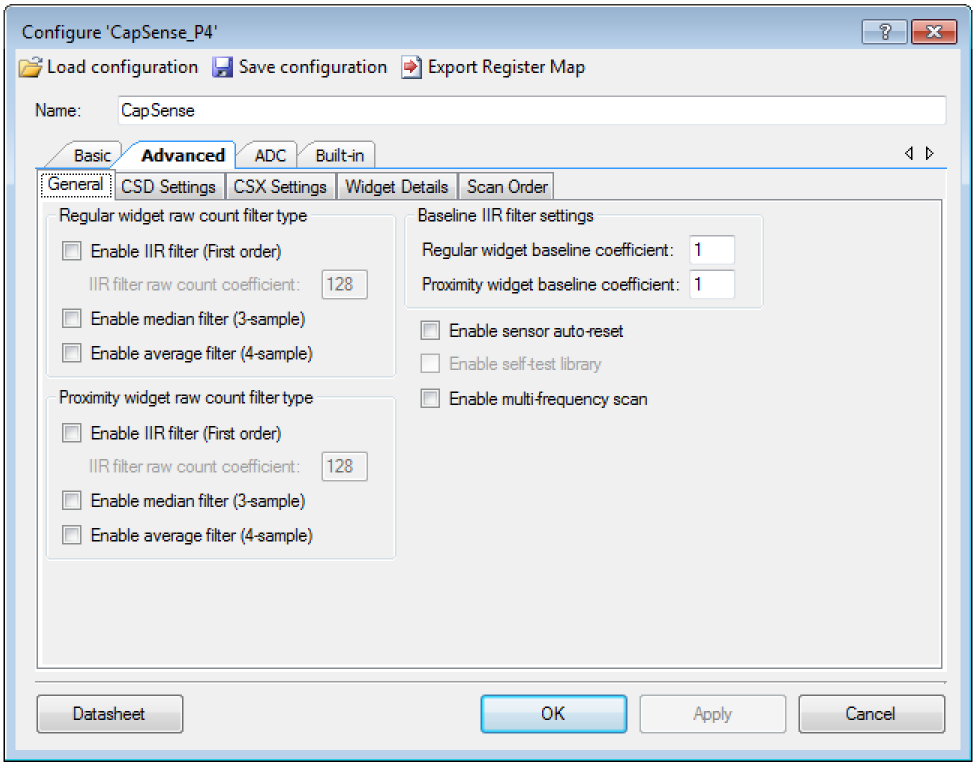
The SmartSense Auto-tuning (both Full Auto-Tune and Hardware parameters only) supports the IDAC Sourcing configuration only.
The SmartSense (Full Auto-Tune) and Enable multi-frequency scan features are mutually exclusive. I.e. if the multi-frequency scan is enabled, it is not possible to enable SmartSense (Full Auto-Tune) or vice-versa.
SmartSense Auto-tuning requires Modulator clock frequency is set to 6000 kHz or higher.
</td></tr>
</table>

Name	Description
Widget Type	A widget is one or a group of sensors that perform a specific user-interface functionality. ▪ **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: o *Proximity threshold*: To detect an approaching hand or finger. o *Touch threshold*: To detect a finger touch on the sensor.
Widget Name	A widget name can be defined to aid in referring to a specific widget in 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.
Sensing Mode	The parameter to select the sensing mode for each widget: ▪ **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.

Name	Description
Widget Sensing Element(s)	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). The following element numbers are supported by the *CSD sensing method*: ▪ *Button* supports 1 to 32 sensors within a widget. ▪ *Linear Slider* supports 3 to 32 segments within a widget. ▪ *Radial Slider* supports 3 to 32 segments within a widget. ▪ *Matrix Buttons* support 2 to 16 rows and columns. The number of total intersections (sensors) is equal to that of rows x columns, limited to the maximum of 32. ▪ *Touchpad* supports 3 to 16 rows and columns. ▪ *Proximity* supports 1 to 16 sensors within a widget. The following element numbers are supported by the *CSX sensing method*: ▪ *Button* – 1 to 32 Rx electrodes (for 1 to 32 sensors) and Tx is fixed to 1. ▪ *Matrix Buttons* – 2 to 16 Tx and Rx. The total intersections (node) number is equal to Tx × Rx, limited to the maximum of 32. ▪ *Touchpad* supports 3 to 16 Tx and Rx. The total intersections (node) number is equal to Tx × Rx. The maximum number of nodes is 256.
Finger capacitance	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 *SmartSense Auto-tuning* algorithm. The supported Finger capacitance range: ▪ In the *SmartSense (Full Auto-Tune)* mode it is 0.1 pF to 1 pF with a 0.02 pF step. ▪ In the *SmartSense (Hardware parameters only)* mode it is 0.02 pF to 20.48 pF on the exponential scale.
Move up / Move down	Moves the selected widget up or down by one on the list. It defines the widget scanning order. **Note** Widget deleting may break a pin assignment and you will need to repair the assignment in the Pin Editor.
Delete	Deletes the selected widget from the list. **Note** Widget deleting may break a pin assignment and you will need to repair the assignment in the Pin Editor.
CSD electrodes	Information: Indicates the total number of electrodes (port pins) used by the CSD widgets, including the *Cmod*, *Csh* and *Shield* electrodes.
CSX electrodes	Information: Indicates the total number of electrodes (port pins) used by the CSX widgets, including the *CintA* and *CintB* capacitors.
Pins required	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 *Pins available* for a project to build successfully.
Pins available	Information**:** Indicates the total number of port pins available for the selected device.

Advanced Tab

This tab provides advanced configuration parameters. In *SmartSense Auto-tuning,* most of the advanced parameters are automatically tuned by the algorithm and the user does not need to set values for these parameters 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.

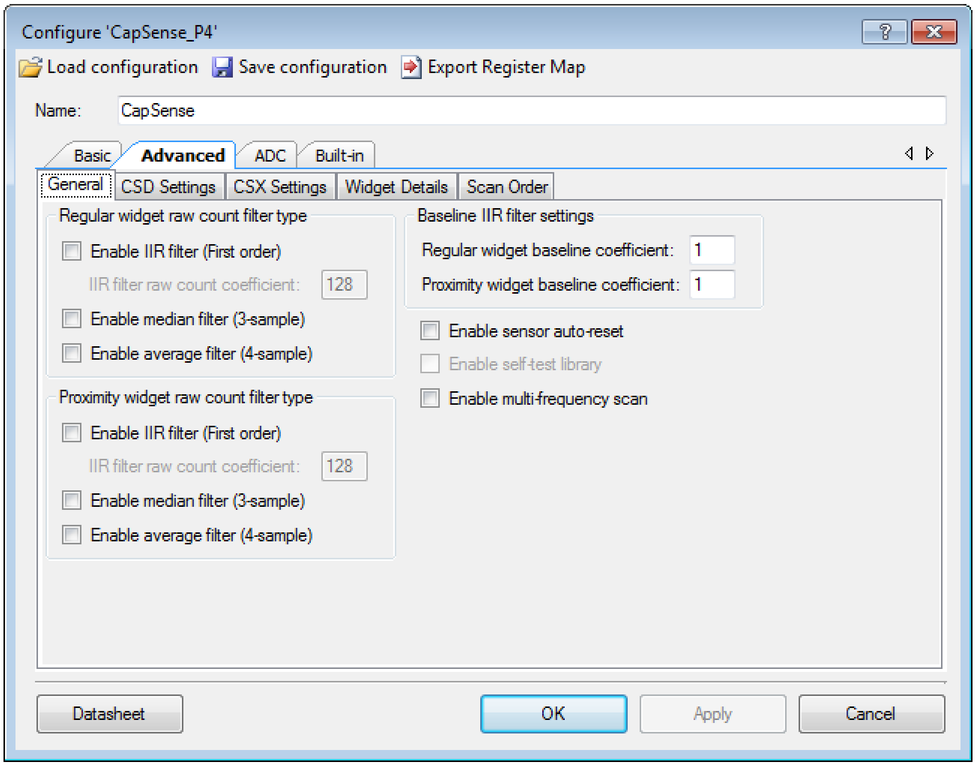


The parameters in the **Advanced** tab are systematically arranged in the five sub-tabs.

- *General* – Contains all parameters common for all widgets respective 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.

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

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 *Regular widget raw count filter type* parameters. The *Proximity* sensors require high-noise reduction. These dedicated parameters allow for setting the proximity filter configuration and behavior differently compared to other widgets.
IIR filter raw count coefficient	
Enable median filter (3-sample)	
Enable average filter (2-sample)	

Baseline filter settings

Baseline filter settings are applied to all sensors baselines. However, 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 *Regular widget baseline* coefficient, but with a dedicated parameter allows controlling the baseline update-rate of the proximity sensors differently compared to other widgets.

General settings

The general settings are applicable to the whole Component behavior.

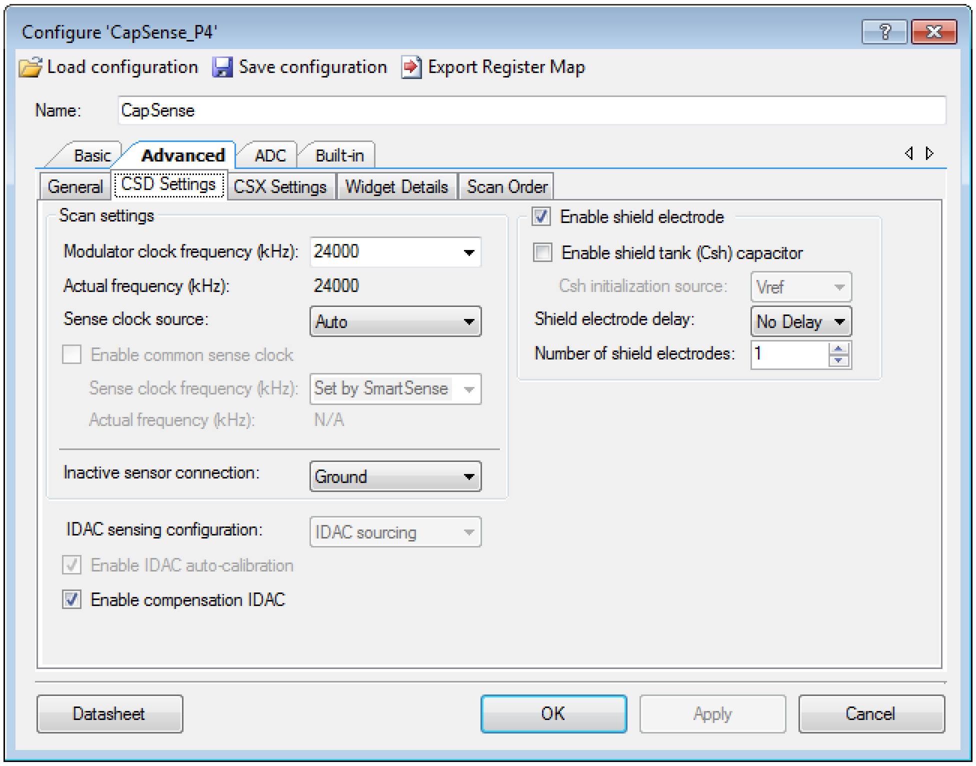
Name	Description
Enable sensor auto-reset	When enabled, the baseline is always updated and when disabled, the baseline is updated only when the difference between the baseline and raw count is less than the noise threshold. When enabled, the feature prevents the sensors from permanently turning on when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

Name	Description
Enable self-test library	The Component provides the **B**uilt-**I**n **S**elf-**T**est (BIST) library to support Class B (IEC-60730), safety integrity level compliant design such as white goods and automotive, and design for manufacturing testing. The library includes a set of tests to validate a board, the Component configuration and operation. Enable the feature to get these advantages, include the safety functions for risk-reduction, validate boards at manufacturing and verify the Component operation in run-time. The provided tests can be classified into two categories: 1. The HW Tests, to confirm the CSD chip hardware and sensor hardware (external to chip), are functional. They include: ▪ Chip analog routing verification ▪ Pin faults checking ▪ PCB-trace opens / shorts checking ▪ External capacitors and sensors capacitance measurement 2. The FW Tests to confirm the integrity of data used for making a decision on the sensor status. They include: ▪ Component global and widget specific configuration verification ▪ Sensor baseline duplication ▪ Sensor raw count and baseline are in the specified range The application level is responsible for running each test at start up and at run-time with some period. We recommend running the tests periodically based on the end-product safety requirements. The execution time of each test is less than 10 ms at HFCLK = 12 MHz. The high-level function CapSense_RunSelfTest() executes a set of tests based on the enable-mask input (argument). The return status contains a PASS/FAIL bit for each test. Additionally, a set of the provided low-level functions allows executing specific widget / sensor tests. Refer to the *Application Programming Interface* section for functions details. **Note** The Component does not perform automatically the CapSense Data Structure error checking if any parameter is changed directly by the application level or through the host communication. We recommend using the CapSense_SetParam() function to modify any parameter value and automatically keep a corresponding CRC register updated. The *SmartSense (Full Auto-Tune)* and self-test features are mutually exclusive. i.e. if the threshold Auto-tuning is enabled, it is not possible to enable the self-test library or vice-versa.

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

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	Selects the modulator clock frequency used for the *CSD sensing method*. The minimum value is 1000 kHz independent from the device selected for the project. The maximum value is device-dependent as follows: ▪ PSoC 4000S/PSoC 4100S/PSoC 4100PS/PSoC Analog Coprocessor: 48000 kHz or HFCLK, whichever is lower. Enter any value between the min and max limits based on the availability of the clock divider, the next valid lower value is selected by the Component, and the actual frequency is shown in the read-only label below the drop-down list. The default value is the highest modulator clock. A higher modulator clock-frequency reduces the sensor scan time, therefore, results in lower power and reduces the noise in the raw counts, so it is recommended to use the highest possible frequency. *SmartSense Auto-tuning* requires *Modulator clock frequency* is set to 6000 kHz or higher.

Name	Description
Sense clock source	*Sense clock frequency* 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 *Sense clock frequency* 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. *Sense clock frequency* is derived from *Modulator clock frequency* using a Sense clock-divider. The following sources are available: ▪ *Direct* – PRS and SSC are disabled and a fixed clock is used. ▪ *PRS8* – The clock spreads using PRS to Modulator Clock / 256. ▪ *PRS12* – The clock spreads using PRS to Modulator Clock / 4096. ▪ *Auto* – The Component automatically selects the optimal SSC, PRS or Direct sources individually for each widget. In addition to the listed above options, the following sense-clock sources are available as follows: ▪ PSoC 4000S and PSoC 4100S devices: *SSC6, SSC7, SSC9* and *SSC10* – The clock spreads using from 6 bits to 10 bits of the sense-clock divider respectively. ▪ PSoC 4100PS and PSoC Analog Coprocessor devices: *SSC2* to *SSC5* – The clock spreads using from 2 bits to 5 bits of the sense-clock divider respectively. The Auto is the recommended sense-clock source selection. The following rules should be adhered at SSC selection: ▪ The ratio between *Modulator clock frequency* and *Sense clock frequency* for the PSoC 4000S and PSoC 4100S devices should be not less than 20 and for PSoC 4100PS and PSoC Analog Coprocessor should be not less then (2^{SSCn-1}+3). ▪ 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 *Modulator clock frequency* and *Sense clock frequency*. ▪ 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. The following rules should be adhered at PRS selection: ▪ At least one full PRS polynomial should finish during scan time.
Enable common Sense clock	When selected, all CSD widgets share the same sense clock at a frequency specified in the *Sense clock frequency (kHz)* parameter, otherwise *Sense clock frequency* can be entered separately for each CSD widget in the *Widget Details* tab. Using a common sense clock for all CSD widgets results in lower power consumption and optimized memory usage, but, if the sensor parasitic capacitance is significantly different for each widget, then a common Sense clock may not produce the optimal performance. To enable *SmartSense Auto-tuning*, this parameter should be unselected because SmartSense will set a Sense clock for each widget based on the sensor properties for the optimal performance.

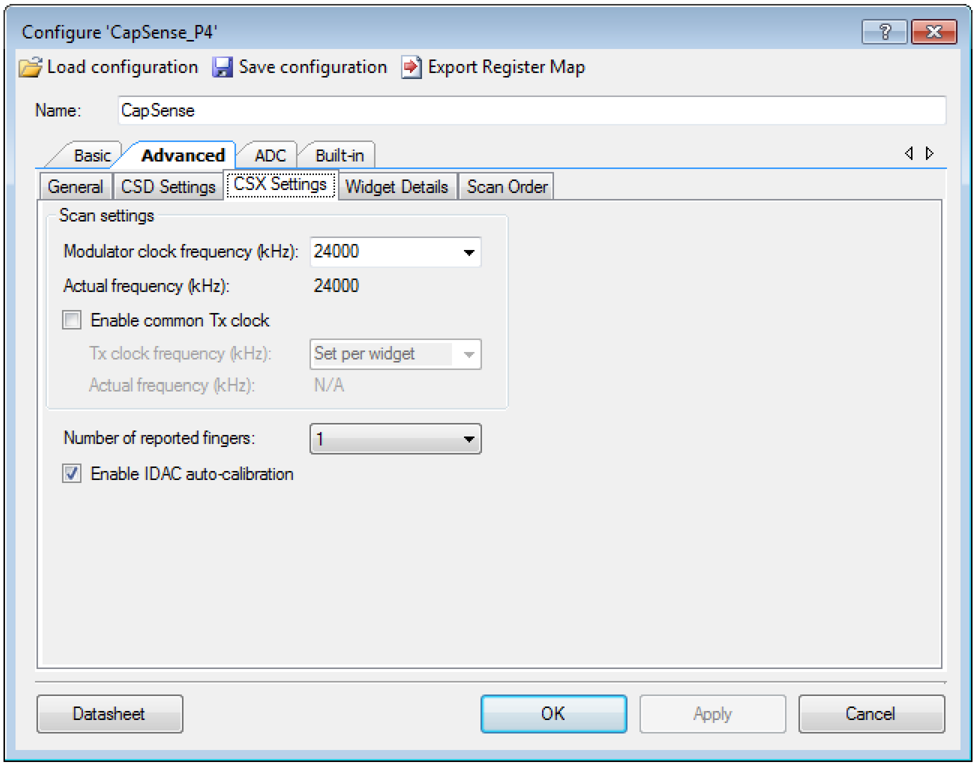
PRELIMINARY

Name	Description
Sense clock frequency	Sets the CSD Sense clock frequency. The minimum value is 45 kHz for all device families. The maximum value depends on the selected device: ▪ PSoC 4000S / PSoC 4100S / PSoC 4100PS / PSoC Analog Coprocessor: 6000 kHz or HFCLK/2, whichever is lower. 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. For more information on selecting the proper divider, see the *PSoC® 4 CapSense® Design Guide*. When SmartSense is selected in *CSD tuning mode*, the Sense Clock frequency is automatically set by the Component to the optimal value by following the (2*5*R*C) rule and this control is grayed out . When *Enable common Sense clock* is unselected, the Sense clock frequency can be set individually for each widget in the *Widget Details* tab, and this control is grayed out. **Note** If the HFCLK or *Modulator clock frequency* is changed, the Component automatically recalculates a next closest Sense clock frequency value to a possible one.
Inactive sensor connection	Selects the state of the sensor when not being scanned. ▪ **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 *Enable shield electrode* check box is set. **Ground** is the recommended selection for this parameter when water tolerance is not required for the design. Select Shield when the design needs water-tolerance or sensor parasitic capacitance reduction in a design.
IDAC sensing configuration	Selects the type of IDAC switching: ▪ **IDAC Sourcing (default)** – Sources current into the modulator capacitor (*Cmod*). The analog switches are configured to alternate between the *Cmod* and GND. IDAC Sourcing is recommended for most designs because of the better signal-to-noise ratio ▪ **IDAC sinking** – Sinks current from the modulator capacitor (*Cmod*). The analog switches are configured to alternate between V_{DD} and *Cmod*. The IDAC sinking provides better robustness to some tests.
Enable IDAC auto-calibration	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 *SmartSense Auto-tuning* parameter can be enabled only when Enable IDAC auto-calibration is selected.
Enable compensation IDAC	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.
Enable shield electrode	The shield electrode is used to reduce the sensor parasitic capacitance, enable water-tolerant CapSense designs and enhance the detection range for the *Proximity* sensors. When the shield electrode is disabled, all configurable parameters associated with the shield electrode are hidden.

Name	Description
Enable shield tank (Csh) capacitor	The shield tank capacitor is used to increase the drive capacity of the shield electrode driver. It should be enabled when the shield electrode capacitance is higher than 100 pF. The recommended value for a shield tank capacitor is 10nF/5v/X7R or an NP0 capacitor. The Shield tank capacitor is not supported in a configuration which includes both CSD- and CSX-sensing based widgets.
Csh initialization source	Selects the initialization source for the shield tank electrode, when *Enable shield tank (Csh) capacitor* is enabled. The two options are available: ▪ **Vref** – Precharge the shield tank by connecting Vref to the Csh capacitor. ▪ **IO Buffer** – Precharge the shield tank by connecting the VDD supply to the Csh capacitor and turning it off using the feedback system when the Csh voltage reaches Vref. This option is available only when the Csh capacitor is assigned to one of the dedicated Csh pins (refer to the device datasheet for pin details) and these dedicated pins are available for Csh when the *CSX sensing method* is not used in the project. The recommended source of precharge is the IO buffer.
Shield electrode delay	Configures the delay between the sensor signal and shield electrode signal for phase alignment. The following options are available for selection: ▪ **No Delay** ▪ **5 ns** ▪ **10 ns** ▪ **20 ns** Most designs work with the **No Delay** option and it is the recommended value.
Number of shield electrodes	Select the number of shield electrodes required in the design. Most designs work with one dedicated shield electrode but, some designs require multiple dedicated shield electrodes to ease the PCB layout routing or to minimize the PCB area used for the shield layer. The minimum value is 0 (i.e. shield signal could be routed to sensors using the *Inactive sensor connection* parameter) and the maximum value is equal to the total number of CapSense-enabled port pins available for the selected device.

CSX Settings Sub-Tab

The parameters in this sub-tab apply to all widgets that use the *CSX sensing method.* If no 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	Selects the modulator clock-frequency used for the *CSX sensing method*. The minimum value is 1000 kHz independent from the device selected for the project. The maximum value is device-dependent as follows: ▪ PSoC 4000S/PSoC 4100S/PSoC 4100PS/PSoC Analog Coprocessor: 48000 kHz or HFCLK, whichever is lower. Enter any value between the min and max limits, based on the availability of the clock divider, the next valid lower value is selected by the Component, and the actual frequency is shown in the read-only label below the drop-down list. 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.

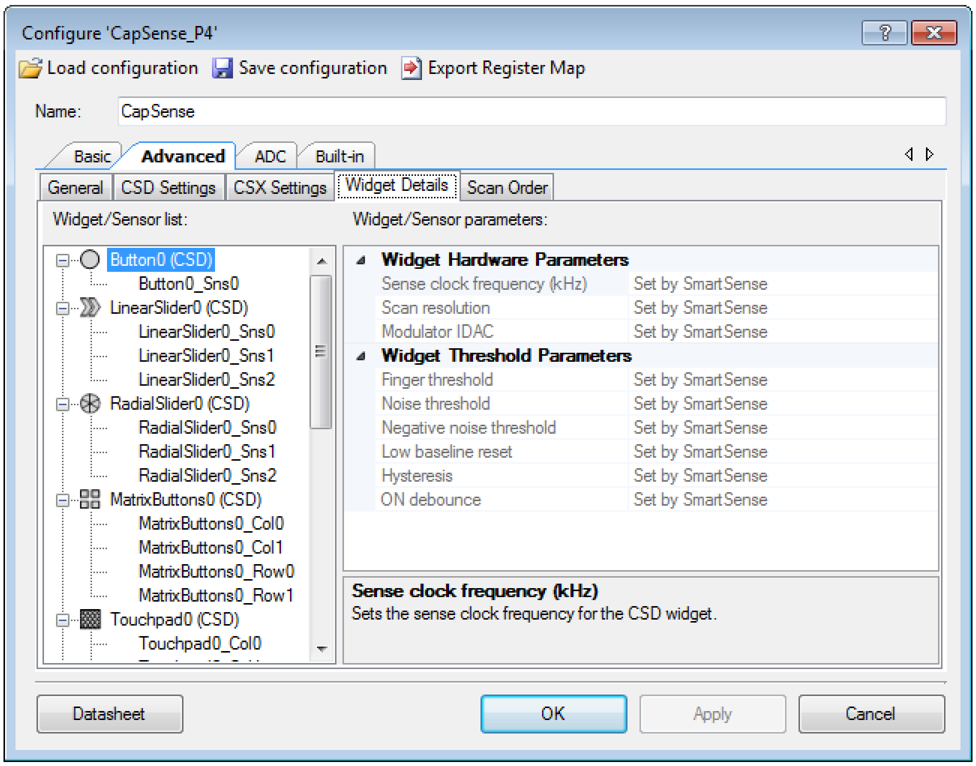
Name	Description
Tx clock source	*Tx clock frequency* is used to sample the input sensor. Both the type of the clock source and the clock frequency are configurable in *Fourth-generation CapSense* devices. The Spread Spectrum Clock (SSC) provides a dithering clock source with a center frequency equal 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. *Tx clock frequency* is derived from *Modulator clock frequency* using a sense clock divider. The following clock sources are available in *Fourth-generation CapSense*: ▪ *Direct* – SSC is disabled and a fixed clock is used. ▪ *Auto* – The Component automatically selects the optimal SSC or Direct sources individually for each widget. In addition to the options listed above, the following sense-clock sources are available: ▪ PSoC 4000S and PSoC 4100S devices: *SSC6, SSC7, SSC9* and *SSC10* – The clock spreads using from 6 bits to 10 bits of the sense-clock divider respectively. ▪ PSoC 4100PS and PSoC Analog Coprocessor devices: *SSC2* to *SSC5* – The clock spreads using from 2 bits to 5 bits of the sense-clock divider respectively. Auto is the recommended Sense clock source selection. The following rules should be followed at SSC selection: ▪ The ratio between *Modulator clock frequency* and *Tx clock frequency* for the PSoC 4000S and PSoC 4100S devices should be not less than 20 and for PSoC 4100PS and PSoC Analog Coprocessor should be not less than (2^{SSCn-1}+3). ▪ 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 *Modulator clock frequency* and *Tx clock frequency*. 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.
Enable common Tx clock	When selected, all CSX widgets share the same Tx clock with the frequency specified in the *Tx clock frequency* (kHz) parameter, otherwise *Tx clock frequency* can be entered separately for each CSX widget in the *Widget Details* tab. Using the common Tx clock for all CSX widgets results in lower power consumption and optimized memory usage and it is the recommended setting for the CSX widgets. 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.
Tx clock frequency	Sets the CSX Tx clock frequency. The minimum value is 45 kHz for all device families. The maximum value is 3000 kHz. 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. The highest Tx clock frequency produces the maximum signal and is the recommended setting. When *Enable common Tx clock* is unselected, the Tx Clock frequency can be set individually for each widget in the *Widget Details* tab, and this control is grayed out. **Note** If the HFCLK or *Modulator clock frequency* is changed, the Component automatically recalculates the next closest Tx clock frequency value to a possible one.

PRELIMINARY

Name	Description
Number of reported fingers	Sets the number of reported fingers for a CSX Touchpad widget only. The available options are from 1 to 3.
Enable IDAC auto-calibration	When enabled, 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 duplexing sensor pattern and without using additional port pins and sensors in the Component.

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>Maximum position</td><td>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.</td></tr>
<tr><td>Maximum X-axis position</td><td rowspan="2">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.</td></tr>
<tr><td>Maximum Y-axis position</td></tr>
<tr><td>Position filter</td><td>Enables the specific filter on a Centroid position value to reduce noise due to varying finger touches.

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.
$$Output = \frac{N}{K} \times input + \frac{(K - N)}{K} \times previous\ Output$$
Where:
K = 256,
N = 128.
$$Output = (input + previous\ Output) / 2$$
Consumes 2 bytes of RAM per each position (filter history).
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).
</td></tr>
<tr><td colspan="2">Widget Hardware Parameters
Note All the Widget Hardware parameters for the CSD widgets are automatically set when SmartSense Auto-tuning is selected in the CSD tuning mode.</td></tr>
<tr><td>Sense clock frequency</td><td>This parameter is identical to the Sense clock frequency parameter in CSD Settings tab. When Enable common Sense clock is unselected in the CSD Settings tab, a sense clock frequency for each widget is set here.</td></tr>
<tr><td>Row sense clock frequency</td><td rowspan="2">These parameters are identical to the Sense clock frequency parameter, and are used to set the sense cock frequency for row and column sensors of Matrix Buttons and Touchpad widgets.</td></tr>
<tr><td>Column sense clock frequency</td></tr>
</table>

PRELIMINARY

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>Tx clock frequency</td><td>This parameter is identical to Tx clock frequency parameter in the CSX Settings tab. When Enable common Tx clock is unselected in CSX Settings tab, a Tx clock frequency for each widget is set here.</td></tr>
<tr><td>Scan resolution</td><td>Selects the scan resolution of CSD widgets (Resolution of capacitance to digital conversion). Acceptable values are from 6 to 16 bits.</td></tr>
<tr><td>Number of sub-conversions</td><td>Selects the number of sub-conversions in the CSX sensing method. The number of sub-conversion should meet the following equation:
$$N_{Sub} < \frac{2^{16} \bullet TxClk}{ModClk}$$
where,
ModClk = CSX Modulator clock frequency
TxClk = Tx clock frequency
N_{Sub} = the value of this parameter.</td></tr>
<tr><td>Modulator IDAC</td><td>Sets the modulator IDAC value for the CSD Button, Slider, or Proximity widget.
The value of this parameter is automatically set when Enable IDAC auto-calibration is selected in the CSD Settings tab.</td></tr>
<tr><td>Row modulator IDAC</td><td rowspan="2">Sets a separate modulator IDAC value for the row and column sensors of the CSD Matrix Buttons and Touchpad widget.
Values of these parameters are automatically set when Enable IDAC auto-calibration is checked in the CSD Settings tab.</td></tr>
<tr><td>Column modulator IDAC</td></tr>
<tr><td colspan="2">Widget Threshold Parameters
Note All the threshold parameters for the CSD widgets are automatically set when SmartSense (Full Auto-Tune) is selected in the CSD tuning mode parameter.</td></tr>
<tr><td>Finger threshold</td><td>The finger threshold parameter is used along with the hysteresis parameter to determine the sensor state as follows:
■ ON: Signal > (Finger Threshold + Hysteresis)
■ OFF: Signal ≤ (Finger Threshold – Hysteresis).
Note that “Signal” in the above equations refers to:
Difference Count = Raw Count – Baseline.
It is recommended to set the Finger threshold parameter value to be equal to the 80% of the touch signal.
The Finger Threshold parameter is not available for the Proximity widget. Instead, Proximity has two thresholds:
■ Proximity threshold
■ Touch threshold</td></tr>
</table>

Name	Description
Noise threshold	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 *Enable sensor auto-reset* is selected (In other words, the baseline remains constant as long as the raw count is above the baseline + noise threshold. This prevents the baseline from following the raw counts during a finger touch detection event). It is recommended to set the noise threshold parameter value to be equal to 2x noise in the raw count or 40% of signal. For the *Linear Slider*, *Radial Slider* and *Touchpad* widgets, the centroid position is calculated by subtracting the noise threshold from the signals. It provides a more accurate position.
Negative noise threshold	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 *Low baseline reset* parameter. The negative noise threshold ensures that the baseline does not fall low because of any high-amplitude repeated negative-noise spikes on the raw count caused by different noise sources such as ESD events. It is recommended to set the negative noise threshold parameter value to be equal to the *Noise threshold* parameter value.
Low baseline reset	This parameter is used along with the *Negative noise threshold* parameter. It counts the number of abnormally low raw counts required to reset the baseline. If a finger is placed on the sensor during a device startup, the baseline gets initialized to 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. The recommended value is 30 which works for most designs.
Hysteresis	The hysteresis parameter is used along with the *Finger threshold* parameter (*Proximity threshold* and *Touch threshold* for Proximity sensor) to determine the sensor state. The hysteresis provides immunity against noisy transitions of the sensor state. See the description of *Finger threshold* parameter for details. The recommend value for the hysteresis is the 10% *Finger threshold*.
ON debounce	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 ▪ Buttons/Matrix button/Proximity – An ON status is reported only when the sensor is touched for a consecutive debounce number of samples. ▪ Sliders/Touchpads – The position status is reported only when any of the sensors is touched for a consecutive debounce number of samples. The recommended value for the Debounce parameter is 3 for reliable sensor status detection.
Proximity threshold	The design of these parameters is the same as for the *Finger threshold* parameters. The proximity sensor requires a higher noise reduction, and supports two levels of detection:

Name	Description
Touch threshold	▪ The proximity level to detect an approaching hand or finger. ▪ The touch level to detect a finger touch on the sensor similarly to other *Widget Type* sensors. Note that for valid operation, the Proximity threshold should be higher than the Touch threshold. The threshold parameters such as *Hysteresis* and *ON debounce* are applicable to both detection levels.
Velocity	Defines the maximum speed of a finger movement in terms of 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	Sets the Compensation IDAC value for each CSD sensor when *Enable compensation IDAC* is selected on *CSD Settings* tab. If *CSD tuning mode* is set to *SmartSense Auto-tuning* or *Enable IDAC auto-calibration* is selected on the *CSD Settings* tab, the value of this parameter is set equal to the Modulator IDAC value at a device power-up for the maximum performance from the sensor. It is recommended to select *Enable IDAC auto-calibration* for robust operation.
IDAC Values	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. When *Enable IDAC auto-calibration* is selected on the *CSX Settings* tab, the value of this parameter is automatically set to the lowest possible value at a device power-up for better performance. It is recommended to select *Enable IDAC auto-calibration* for robust operation.
Selected pins	Selects a port pin for the sensor (CSD sensing) and electrode (CSX sensing). The available options 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.

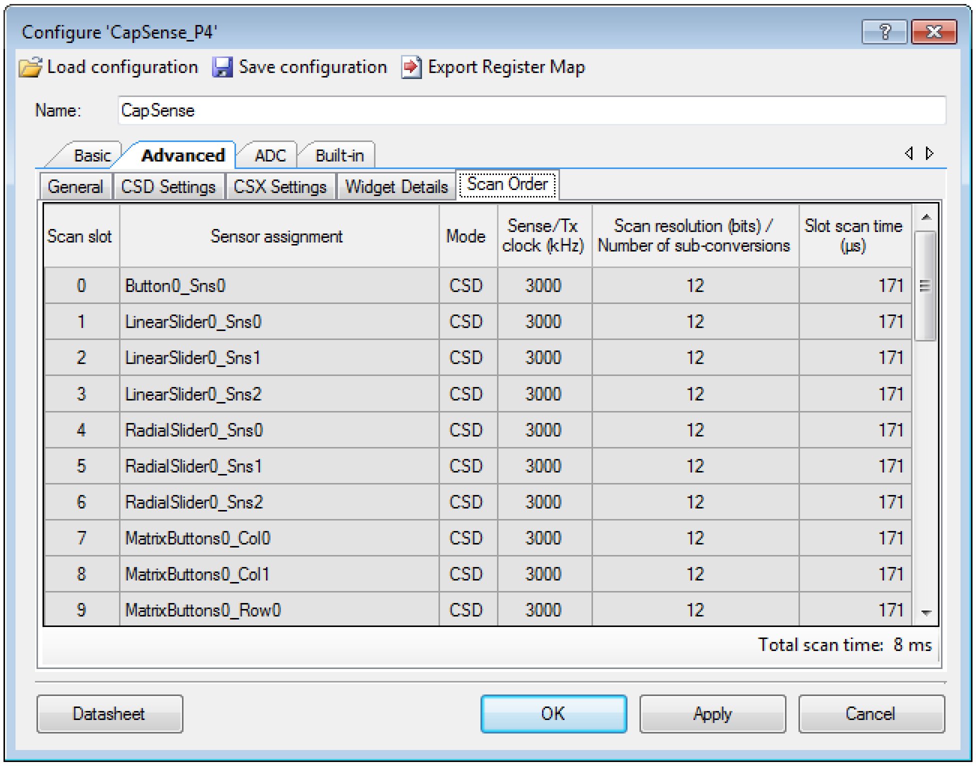
The following table shows which parameters belong to a given widget:

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			√	√	√				√		√				√	√	√	√	√	√			√		√	√

PRELIMINARY

Scan Order Sub-Tab

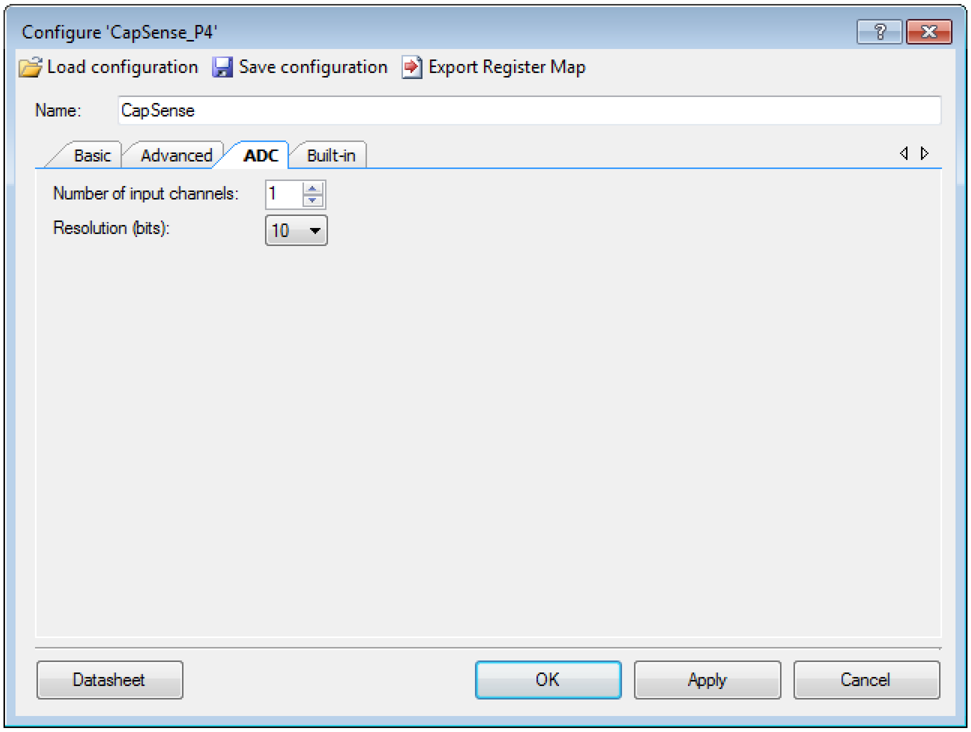
This tab provides total time required to scan all the sensors (does not include the data processing execution time) and scan time for each sensor slot.



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

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. To place the CapSense_ADC catalog, 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: ▪ 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.

Note The CapSense_P4 v4.X firmware API is very different compared with the third-generation API of the CapSense_CSD_P4 Component (v2.40 and before). In addition to the new CSX features, the API has been optimized to reduce power consumption and user code complexity. As a result, applications that run on the older Component will require significant changes to the firmware if you change the design to use the new Component. Refer to the *Step-7: API Comparison* section for details on migrating your firmware to new CapSense API.

The 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

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

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

(Replace vX_XX with the Component version)

By default, the instance name of the Component is “CapSense_ADC_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.

CapSense High Level APIs

Description

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

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

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

Functions

- cystatus CapSense_Start(void)
  Initializes component hardware and firmware modules. This function should be called by the application program prior to calling any other function of the component.
- cystatus CapSense_Stop(void)
  Stops the component operation.
- cystatus CapSense_Resume(void)
  Resumes the component operation if CapSense_Stop() function was called previously.
- cystatus CapSense_ProcessAllWidgets(void)
  Performs full data processing of all enabled widgets.
- cystatus CapSense_ProcessWidget(uint32 widgetId)
  Performs full data processing of the specified widget if it is enabled.
- void CapSense_Sleep(void)
  Prepares the component to deep sleep.
- void CapSense_Wakeup(void)
  Currently this function is empty and exists as a place for future updates, this function shall be used to resume the component after exiting deep sleep.
- uint32 CapSense_RunSelfTest(uint32 testEnMask)
  Runs built-in self-tests specified by the test enable mask.
- cystatus CapSense_SetupWidget(uint32 widgetId)
  Performs the initialization required to scan the specified widget.
- cystatus CapSense_Scan(void)
  Initiates scan of all sensors in the widget which is initialized by CapSense_SetupWidget(), if the no scan is in progress.
- cystatus CapSense_ScanAllWidgets(void)
  Initializes the first enabled widget and scans of all the sensors in the widget, then the same process is repeated for all widgets in the component. I.e. scan all the widgets in the component.
- uint32 CapSense_IsBusy(void)
  Returns the current status of the component (scan completed or scan 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

cystatus CapSense_Start (void)

This function initializes the component hardware and firmware modules and should be 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 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 a scan for all sensors and initialize the baseline history. During the scan, CPU is in the sleep mode to save power.
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 to the data structure registers and calling this function again. This is also done inside the CapSense_RunTuner() function when a restart command is received.

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

Returns:

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

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

cystatus CapSense_Stop (void)

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

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

Returns:

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

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

cystatus CapSense_Resume (void)

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

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

Returns:

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

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

cystatus CapSense_ProcessAllWidgets (void)

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

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

Disabled widgets are not processed. To disable/enable a widget, set appropriate values in the CapSense_WDGT_ENABLE<RegisterNumber>_PARAM_ID register using the CapSense_SetParam() function. This function should be called only after all the sensors in the component are scanned. Calling this function multiple times without sensor scanning causes unexpected behavior.

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

Returns:

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

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

cystatus CapSense_ProcessWidget (uint32 *widgetId*)

This function performs exactly the same tasks as CapSense_ProcessAllWidgets(), but only for a specified widget. This function can be used along with the CapSense_SetupWidget() and CapSense_Scan() functions to scan and process data for a specific widget. This function should be 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 Self-test library is enabled this function executes baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Parameters:

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

Returns:

Returns the status of the widget processing:

PRELIMINARY

- CYRET_SUCCESS if operation is successfully completed
- CYRET_BAD_PARAM if the input parameter is invalid
- CYRET_INVALID_STATE if the specified widget is disabled
- CYRET_BAD_DATA if processing was 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 shall 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.

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

uint32 CapSense_RunSelfTest (uint32 *testEnMask*)

The function performs the tests that correspond to the specified bits in the testEnMask parameter.

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

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

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

Parameters:

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

Returns:

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

- CYRET_SUCCESS - if all tests passed
- CapSense_TST_NOT_EXECUTED - if previously triggered scanning is not completed
- CapSense_TST_BAD_PARAM - if a non-defined test was requested in the testEnMask parameter
- The bit-mask of the failed tests.

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

cystatus CapSense_SetupWidget (uint32 *widgetId*)

This function prepares the component to scan all 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:

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

Returns:

Returns the status of the widget setting up operation:

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

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

cystatus CapSense_Scan (void)

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

Returns:

Returns the status of the scan initiation operation:

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

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

cystatus CapSense_ScanAllWidgets (void)

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

Returns:

Returns the status of operation:

- CYRET_SUCCESS if scanning is successfully started.

- CYRET_BAD_PARAM if all the widgets are disabled.
- CYRET_INVALID_STATE if the previous scanning is not completed and the HW block is busy.
- CYRET_UNKNOWN if there are unknown errors.

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

uint32 CapSense_IsBusy (void)

This function returns the 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 should be made. It is recommended using the critical section (i.e. disable global interrupt) in the application when the device transitions from the active mode to sleep or deep sleep mode.

Returns:

Returns the current status of the component:

- CapSense_NOT_BUSY if there is no scan in progress and a next scan can be initiated.
- CapSense_SW_STS_BUSY if 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 the information from the wdgtStatus registers (CapSense_WDGT_STATUS<X>_VALUE). This function does not process any widget but extracts the processed results from the Data Structure.

Returns:

Returns the touch detection status of all widgets:

- Zero if no touch is detected in all widgets or sensors.
- Non-zero if 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 the information from the wdgtStatus registers (CapSense_WDGT_STATUS<X>_VALUE). This function does not process the widget, but extracts the processed results from the Data Structure.

Parameters:

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

Returns:

Returns the touch detection status of the specified widgets:

- Zero if 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. 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 the information from wdgtStatus registers (CapSense_WDGT_STATUS<X>_VALUE). This function does not process the widget or sensor, but extracts the processed results from the Data Structure.

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

Parameters:

widgetId	Specify the ID number of the widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as

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

Returns:

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

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

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

uint32 CapSense_IsMatrixButtonsActive (uint32 *widgetId*)

This function reports if the specified matrix widget has detected a touch or not by extracting the 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 should be used only with the matrix button widgets. This function does not process the widget, but extracts the processed results from the Data Structure.

Parameters:

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

Returns:

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

1. For the matrix buttons widgets with the CSD sensing mode:
   - Bit [31] if set, indicates that one or more sensors in the widget detected a touch.
   - Bits [30..24] is reserved
   - Bits [23..16] indicates 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 as more than one touch is invalid for the CSD matrix buttons widgets.
   - Bits [15..8] indicates the active row number.
   - Bits [7..0] indicates the active column number.
2. For the matrix buttons widgets with the CSX widgets, each bit (31..0) corresponds to the TX/RX intersection.

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

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

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

Parameters:

widgetId	Specify the ID number of the proximity widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID
proxId	Specify 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 proximity widget. Zero indicates that no touch is detected in the specified sensor / widget or a wrong widgetId / proxId is specified.

- Bits [31..2] is 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 the information from the wdgtStatus registers (CapSense_<WidgetName>_POSITION<X>_VALUE). This function should be used only with the radial or linear slider widgets. This function does not process the widget but extracts the processed results from the Data Structure.

Parameters:

widgetId	Specify the ID number of the 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 if 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 the 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 the processed results from the Data Structure.

Parameters:

widgetId	Specify the ID number of the 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] indicates the Y coordinate.
   - Bits [15..0] indicates 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 should be 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 firmware and as a result, noise and SNR measurements will not be accurate.
- The Tuner 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 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 abnormal behavior from the component.

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

Returns:

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

- CapSense_STATUS_RESTART_DONE based on a received command, the component was restarted.
- CapSense_STATUS_RESTART_NONE, no restart was executed by this function.

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

CapSense Low Level APIs

Description

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

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

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

Functions

- cystatus CapSense_ProcessWidgetExt(uint32 widgetId, uint32 mode)
  Performs customized data processing on the selected widget.
- cystatus CapSense_ProcessSensorExt(uint32 widgetId, uint32 sensorId, uint32 mode)
  Performs customized data processing on the selected widget's sensor.
- cystatus CapSense_UpdateAllBaselines(void)
  Updates the baseline for all sensors in all widgets.
- cystatus CapSense_UpdateWidgetBaseline(uint32 widgetId)
  Updates the baseline for all sensors in a widget specified by input parameter.
- cystatus CapSense_UpdateSensorBaseline(uint32 widgetId, uint32 sensorId)
  Updates the baseline for a sensor in a widget specified by input parameters.
- void CapSense_InitializeAllBaselines(void)
  Initializes (or re-initialize) the baseline of all sensors of all widgets.
- void CapSense_InitializeWidgetBaseline(uint32 widgetId)
  Initializes (or re-initialize) the baseline of all sensors in a widget specified by input parameter.
- void CapSense_InitializeSensorBaseline(uint32 widgetId, uint32 sensorId)
  Initializes (or re-initialize) the baseline of a sensor in a widget specified by input parameters.
- uint32 CapSense_CheckGlobalCRC(void)

PRELIMINARY

Checks the stored CRC of the CapSense_RAM_STRUCT data structure.

- uint32 CapSense_CheckWidgetCRC(uint32 widgetId)
  Checks the stored CRC of the CapSense_RAM_WD_BASE_STRUCT data structure of the specified widget.
- uint32 CapSense_CheckBaselineDuplication(uint32 widgetId, uint32 sensorId)
  Checks that the baseline of the specified widget/sensor is not corrupted by comparing it with a baseline inverse copy.
- uint32 CapSense_CheckBaselineRawcountRange(uint32 widgetId, uint32 sensorId, CapSense_BSLN_RAW_RANGE_STRUCT*ranges)
  Checks that raw count and baseline of the specified widget/sensor are within the specified range.
- uint32 CapSense_CheckSensorShort(uint32 widgetId, uint32 sensorId)
  Checks the specified widget/sensor for shorts to GND or VDD.
- uint32 CapSense_CheckSns2SnsShort(uint32 widgetId, uint32 sensorId)
  Checks the specified widget/sensor for shorts to any other CapSense sensors.
- uint32 CapSense_GetSensorCapacitance(uint32 widgetId, uint32 sensorId)
  Measures the specified widget/sensor capacitance.
- uint32 CapSense_GetShieldCapacitance(void)
  Measures the shield electrode capacitance.
- uint32 CapSense_GetExtCapCapacitance(uint32 extCapId)
  Measures the capacitance of the specified external capacitor.
- void CapSense_SetPinState(uint32 widgetId, uint32 sensorElement, uint32 state)
  Sets the state (drive mode and output state) of 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 state of all pins associated with ganged sensor is updated.
- cystatus CapSense_CalibrateWidget(uint32 widgetId)
  Calibrates the IDACs for all sensors in the specified widget to default target, this function detects sensing method used by the widget prior to calibration.
- cystatus CapSense_CalibrateAllWidgets(void)
  Calibrates the IDACs for all widgets in the component to default target (85%) value, this function detects 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. The CapSense_CSDScan() function should be used to start scanning when using this function.
- 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. The CapSense_CSDScanExt() function should be called to initiate a scan when using this function.
- void CapSense_CSDScan(void)
  This function initiates the scan for sensors of the widget initialized by the CapSense_CSDSetupWidget() function.
- void CapSense_CSDScanExt(void)
  Starts the CSD conversion on the preconfigured sensor. CapSense_CSDSetupWidgetExt() function should be used to setup the widget when using this function.
- cystatus CapSense_CSDCalibrateWidget(uint32 widgetId, uint32 target)
  Executes IDAC calibration for all sensors in the widget specified in an input.
- void CapSense_CSDConnectSns (CapSense_FLASH_IO_STRUCTconst *snsAddrPtr)
  Connects a port-pin used by the sensor to AMUX bus of CapSense block.

- void CapSense_CSDDisconnectSns (CapSense_FLASH_IO_STRUCTconst *snsAddrPtr)
  Disconnects a sensor port-pin from the CapSense block and 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. The CapSense_CSDScan() function should be used to start scanning when using this function.
- 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. The CapSense_CSXScan() function should be called to initiate the scan when using this function.
- void CapSense_CSXScan(void)
  This function initiates the scan for sensors of the widget initialized by the CapSense_CSXSetupWidget() function.
- void CapSense_CSXScanExt(void)
  Starts the CSX conversion on the preconfigured sensor. The CapSense_CSXSetupWidgetExt() function should be used to setup a widget when using this function.
- void CapSense_CSXCalibrateWidget(uint32 widgetId, uint16 target)
  Calibrates the raw count values of all sensors/nodes in a CSX widget.
- void CapSense_CSXConnectTx (CapSense_FLASH_IO_STRUCTconst *txPtr)
  Connects a TX electrode to the CSX scanning hardware.
- void CapSense_CSXConnectRx (CapSense_FLASH_IO_STRUCTconst *rxPtr)
  Connects an RX electrode to the CSX scanning hardware.
- void CapSense_CSXDisconnectTx (CapSense_FLASH_IO_STRUCTconst *txPtr)
  Disconnects a TX electrode from the CSX scanning hardware.
- void CapSense_CSXDisconnectRx (CapSense_FLASH_IO_STRUCTconst *rxPtr)
  Disconnects an RX electrode from the CSX scanning hardware.
- cystatus CapSense_GetParam(uint32 paramId, uint32 *value)
  Gets the specified parameter value from the Data Structure.
- cystatus CapSense_SetParam(uint32 paramId, uint32 value)
  Sets a new value for the specified parameter in the Data Structure.

Function Documentation

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

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

This function can be used with any of the available scan functions. This function should be called only after all 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 the auto-tuning mode is enabled.

The pipeline scan method (i.e. during scanning of a widget perform processing of a 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 Self-test library is enabled this function executes baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Parameters:

widgetId	Specify the ID number of the widget to be processed. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
mode	Specify the type of widget processing that needs to be executed for the specified widget: 1. Bits [31..6] - Reserved 2. Bits [5..0] - CapSense_PROCESS_ALL - Execute all tasks 3. Bit [5] - CapSense_PROCESS_STATUS - Update Status (on/off, centroid position) 4. Bit [4] - CapSense_PROCESS_THRESHOLDS - Update Thresholds (only in CSD auto-tuning mode) 5. Bit [3] - CapSense_PROCESS_CALC_NOISE - Calculate Noise (only in CSD auto-tuning mode) 6. Bit [2] - CapSense_PROCESS_DIFFCOUNTS - Update Difference Counts 7. Bit [1] - CapSense_PROCESS_BASELINE - Update Baselines 8. Bit [0] - CapSense_PROCESS_FILTER - Run Firmware Filter

Returns:

Returns the status of the widget processing operation:

- CYRET_SUCCESS if processing is successfully performed
- CYRET_BAD_PARAM if the input parameter is invalid
- CYRET_BAD_DATA if processing was failed

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

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

This function performs data processes for the specified sensor specified by the mode parameter. The execution order of the requested operations is from LSB to MSB of the mode parameter. If a different order is needed, 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 should be 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 the auto-tuning mode is enabled.

The pipeline scan method (i.e. during scanning of a sensor perform processing of a previously scanned sensor) 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 Self-test library is enabled this function executes baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Parameters:

widgetId	Specify the ID number of the widget to process one of its sensors. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID
sensorId	Specify the ID number of the sensor within the widget to process it. A macro for the sensor ID within a specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID
mode	Specify the type of the sensor processing that needs to be executed for the specified sensor: 1. Bits [31..5] - Reserved 2. Bits [4..0] - CapSense_PROCESS_ALL - Executes all tasks 3. Bit [4] - CapSense_PROCESS_THRESHOLDS - Updates Thresholds (only in

PRELIMINARY

	auto-tuning mode) 4. Bit [3] - CapSense_PROCESS_CALC_NOISE - Calculates Noise (only in auto-tuning mode) 5. Bit [2] - CapSense_PROCESS_DIFFCOUNTS - Updates Difference Count 6. Bit [1] - CapSense_PROCESS_BASELINE - Updates Baseline 7. Bit [0] - CapSense_PROCESS_FILTER - Runs Firmware Filter

Returns:

Returns the status of the sensor process operation:

- CYRET_SUCCESS if processing is successfully performed
- CYRET_BAD_PARAM if the input parameter is invalid
- CYRET_BAD_DATA if processing was failed

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

cystatus CapSense_UpdateAllBaselines (void)

Updates the baseline for all sensors in all 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 Self-test library is enabled this function executes baseline duplication test. Refer to CapSense_CheckBaselineDuplication() for details.

Returns:

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

- CYRET_SUCCESS if operation was successfully completed
- CYRET_BAD_DATA if baseline processing failed

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

cystatus CapSense_UpdateWidgetBaseline (uint32 *widgetId*)

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

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

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

Parameters:

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

Returns:

Returns the status of the specified widget update baseline operation:

- CYRET_SUCCESS if operation was successfully completed
- CYRET_BAD_DATA if baseline processing failed

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

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

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

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

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

Parameters:

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

Returns:

Returns the status of the specified sensor update baseline operation:

- CYRET_SUCCESS if operation was successfully completed
- CYRET_BAD_DATA if baseline processing failed

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

void CapSense_InitializeAllBaselines (void)

Initializes the baseline for all sensors of all 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 sensors of the specified widget.

Parameters:

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

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

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

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

Parameters:

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

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

uint32 CapSense_CheckGlobalCRC (void)

The function calculates the CRC value of the CapSense_RAM_STRUCT data structure and compares it with the stored CRC value CapSense_GLB_CRC_VALUE.

Use this function to verify that the register values in the CapSense_RAM_STRUCT data structure are not corrupted.

If the stored and calculated CRC values differ, the calculated CRC is stored to the CapSense_GLB_CRC_CALC_VALUE register and the CapSense_TST_GLOBAL_CRC bit is set in the CapSense_TEST_RESULT_MASK_VALUE register.

The function never clears the CapSense_TST_GLOBAL_CRC bit. CapSense_SetParam() function is recommended for changing a CapSense_RAM_STRUCT register. Then the CRC is updated automatically.

Another way to launch the test is using CapSense_RunSelfTest() function with the CapSense_TST_GLOBAL_CRC mask.

Returns:

Returns a status of the executed test:

- CYRET_SUCCESS - if the stored CRC matches the calculated CRC
- CapSense_TST_GLOBAL_CRC - if the stored CRC is wrong.

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

uint32 CapSense_CheckWidgetCRC (uint32 *widgetId*)

The function calculates the CRC value of the CapSense_RAM_WD_BASE_STRUCT data structure of the specified widget and compares it with the stored CRC value CapSense_<WidgetName>_CRC_VALUE.

Use this function to verify that the register values in the CapSense_RAM_WD_BASE_STRUCT data structure are not corrupted.

If the stored and calculated CRC values differ:

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

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

It is possible to execute the test for all the widgets using the CapSense_RunSelfTest() function with the CapSense_TST_WDGT_CRC mask. In this case the CapSense_WDGT_CRC_CALC_VALUE and CapSense_WDGT_CRC_ID_VALUE registers contain the CRC and ID of the first detected widget with the wrong CRC.

The CapSense_SetParam() function is recommended for changing a CapSense_RAM_WD_BASE_STRUCT register. Then the CRC is updated automatically.

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

Parameters:

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

Returns:

Returns a status of the test execution:

- CYRET_SUCCESS - if the stored CRC matches the calculated CRC
- CapSense_TST_WDGT_CRC - if the widget CRC is wrong
- CapSense_TST_BAD_PARAM - if the input parameter is invalid.

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

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

The function checks the baseline of the specified widget/sensor by comparing the conformity of the baseline and its inversion.

Use this function to verify that a sensor baseline is not corrupted.

If the baseline does not match its inverse copy:

1. The widget ID is stored to the CapSense_INV_BSLN_WDGT_ID_VALUE register
2. The sensor ID is stored to the CapSense_INV_BSLN_SNS_ID_VALUE register

3. The CapSense_TST_BSLN_DUPLICATION bit is set in the CapSense_TEST_RESULT_MASK_VALUE register.

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

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

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

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

Parameters:

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

Returns:

Returns the status of the test execution:

- CYRET_SUCCESS - if the baseline matches its inverse copy
- CapSense_TST_BSLN_DUPLICATION - if the test failed
- CapSense_TST_BAD_PARAM - if the input parameters are invalid.

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

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

The function checks the baseline and raw count with limits defined by the user. If the limits can be overpassed, the function sets the CapSenseCapSense_TST_BSLN_RAW_OUT_RANGE bit in the CapSense_TEST_RESULT_MASK_VALUE register.

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

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

Parameters:

widgetId	Specify the ID number of the widget to be processed. A macro for the widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
sensorId	Specify the ID number of the sensor within the widget. A macro for the sensor ID within the specified widget can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_SNS<SensorNumber>_ID.
**ranges*	Specify the pointer to the CapSense_BSLN_RAW_RANGE_STRUCT structure with valid ranges for the raw count and baseline.

Returns:

Returns a status of the test execution:

- CYRET_SUCCESS - if the raw count and baseline are within the specified range
- CapSense_TST_BSLN_RAW_OUT_RANGE - if the test failed
- CapSense_TST_BAD_PARAM - if the input parameters are invalid.

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

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

The function performs verification if the specified sensor is shorted to GND or VDD. The test does:

1. Sets all CapSense sensors to the High-Z state
2. Changes the drive mode of the tested sensor to Pull-Down and verifies if there is a logical 0
3. Changes the drive mode of the tested sensor to Pull-Up and verifies if there is a logical 1 If a sensor consists of several electrodes that using the ganged option, all the sensor electrodes are considered as one sensor and are tested together.

If the test detects a short:

1. The widget ID is stored to the CapSense_SHORTED_WDGT_ID_VALUE register
2. The sensor ID is stored to the CapSense_SHORTED_SNS_ID_VALUE register
3. The CapSense_TST_SNS_SHORT bit is set in the CapSense_TEST_RESULT_MASK_VALUE register.

The function never clears the CapSense_TST_SNS_SHORT bit. If the CapSense_TST_SNS_SHORT bit is set, the CapSense_SHORTED_WDGT_ID_VALUE and CapSense_SHORTED_SNS_ID_VALUE registers are not updated.

It is possible to execute the test for all the widgets using the CapSense_RunSelfTest() function with the CapSense_TST_SNS_SHORT mask. In this case the CapSense_SHORTED_WDGT_ID_VALUE and CapSense_SHORTED_SNS_ID_VALUE registers contain the widget and sensor ID of the first detected fail.

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

Parameters:

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

Returns:

Returns a status of the test execution:

- CYRET_SUCCESS - if the sensor of the widget isn't shorted to VDD or GND
- CapSense_TST_SNS_SHORT - if the test failed
- CapSense_TST_BAD_PARAM - if the input parameters are invalid.

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

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

The function performs verification if the specified sensor is shorted to any other CapSense sensors. The test does:

1. Sets all CapSense sensors to the Strong drive mode with a logical 0
2. Changes the drive mode of the tested sensor to Pull-Up and verifies if there is logical 1 If a sensor consists of several electrodes using the ganged option, all the sensor electrodes are considered as one sensor and are tested together.

If the test detects a short:

1. The widget ID is stored to the CapSense_P2P_WDGT_ID_VALUE register
2. The sensor ID is stored to the CapSense_P2P_SNS_ID_VALUE register

3. The CapSense_TST_SNS2SNS_SHORT bit is set in the CapSense_TEST_RESULT_MASK_VALUE register.

The function never clears the CapSense_TST_SNS2SNS_SHORT bit. If the CapSense_TST_SNS2SNS_SHORT bit is set, the CapSense_P2P_WDGT_ID_VALUE and CapSense_P2P_SNS_ID_VALUE registers are not updated.

It is possible to execute the test for all widgets/sensors using the CapSense_RunSelfTest() function with the CapSense_TST_SNS2SNS_SHORT mask. In this case, the CapSense_P2P_WDGT_ID_VALUE and CapSense_P2P_SNS_ID_VALUE registers contain the widget and sensor ID of the first detected fail.

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

Parameters:

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

Returns:

Returns a status of the test execution:

- CYRET_SUCCESS - if the sensor is not shorted to any other CapSense sensor
- CapSense_TST_SNS2SNS_SHORT - if the test failed
- CapSense_TST_BAD_PARAM - if the input parameters are invalid.

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

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

The function measures the specified widget/sensor, returns the measurement result, and stores it into the data structure.

For CSX sensors, the measurement is performed twice: for RX and for TX electrodes.

If a sensor consists of several electrodes using the ganged option, all the sensor electrodes are considered as one sensor and are tested together.

Component configuration does not influence measurement. It is done with the following parameter settings:

- Modulator clock frequency - as high as possible
- Sense clock source - Direct
- IDAC sensing configuration - IDAC sourcing
- Compensation IDAC - Disabled
- Resolution - 10 bits

The measurement consists of several scans to find an optimal IDAC value and sense clock frequency. Then, the sensor capacitance is calculated based on the found values and received raw counts.

All non-measured sensor states are inherited from the component configuration.

- While measuring a CSX sensor, all non-measured sensors and shield electrodes (if enabled) are set to the Strong drive mode.
- While measuring CSD sensor, all CSX sensors are set to the Strong drive mode and all CSD sensors are set to the state defined by the inactive sensor connection parameter. If the shield electrode parameter is enabled, it is also enabled at a CSD sensor capacitance measurement.

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

It is possible to execute the test for all widgets/sensors using the CapSense_RunSelfTest() function with the CapSense_TST_SNS_CAP mask.

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

Parameters:

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

Returns:

Returns a result of the test execution:

- Bits [7..0] the capacitance (in pF) of the CSD sensor or the capacitance of RX electrode if the specified sensor type is CSX
- Bits [15..8] the capacitance (in pF) of TX electrode if the specified sensor type is CSX
- Bit [30] CapSense_TST_BAD_PARAM if the input parameters are invalid.

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

uint32 CapSense_GetShieldCapacitance (void)

The function measures the shield capacitance, returns the measurement result, and stores it into the CapSense_SHIELD_CAP_VALUE register.

If the shield consists of several electrodes, all the electrodes are joined together.

The sensor state is inherited from the component configuration.

- All CSX sensors are set to the Strong drive mode.
- All CSD sensors are set to the state defined by inactive sensor connection parameter.

If the inactive sensor connection parameter is set to the shield, all CSD sensors are connected to the shield and are measured together with the shield.

Component configuration does not influence a measurement. It is always done with the following parameter settings:

- Modulator clock frequency - as high as possible
- Sense clock source - Direct
- IDAC sensing configuration - IDAC sourcing
- Compensation IDAC - disabled
- Resolution - 10 bits

The measurement consists of several scans to find an optimal IDAC value and sense clock frequency. Then the shield capacitance is calculated based on the found values and received raw counts.

The measurement shield capacitance range is from 5pF to 500pF.

It is possible to execute the test using the CapSense_RunSelfTest() function with the CapSense_TST_SH_CAP mask.

Returns:

The shield electorde capacitance (in pF)

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

uint32 CapSense_GetExtCapCapacitance (uint32 *extCapId*)

The function measures the capacitance, returns the measurement result and stores it into the CapSense_EXT_CAP<EXT_CAP_ID>_VALUE register.

The CapSense sensor state during measurement is High-Z Analog.

The measurement capacitance range is from 200pF to 60000pF. The accuracy of the measurement is 10%.

It is possible to execute the test using the CapSense_RunSelfTest() function with the CapSense_TST_EXTERNAL_CAP mask.

Parameters:

extCapId	Specify the ID number of the external capacitor to be measured: • CapSense_TST_CMOD_ID - Cmod capacitor • CapSense_TST_CSH_ID - Csh capacitor • CapSense_TST_CINTA_ID - CintA capacitor • CapSense_TST_CINTB_ID - CintB capacitor

Returns:

Returns a status of the test execution:

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

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

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

This function sets a specified state to the specified sensor element. For CSD widgets sensor element is a sensor number, for CSX widgets it is either 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 should 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.

It is not recommended to call this function directly from the application layer. This function should be used to implement only the user's specific use cases. Functions that perform a setup and scan of the sensor/widget automatically set needed pin states. They do not take into account changes in the design made by the CapSense_SetPinState() function. This function neither check wdgtIndex nor sensorElement for correctness.

Parameters:

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

cystatus CapSense_CalibrateWidget (uint32 *widgetId*)

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

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

Parameters:

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

Returns:

Returns the status of the specified widget calibration:

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

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

cystatus CapSense_CalibrateAllWidgets (void)

Calibrates the IDACs for all widgets in the component to the default target (85% of maximum possible raw count) value. This function detects the sensing method used by the widgets and takes into account the Enable compensation IDAC parameter.

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

Returns:

Returns the status of the calibration process:

- CYRET_SUCCESS if the operation is successfully completed.
- CYRET_BAD_DATA if 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 and also does not start a scanning process. The CapSense_CSDScan() API must be called after initializing the widget to start scanning.

This function should be called when no scanning 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.

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

Parameters:

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

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

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

This function does the same as CapSense_CSDSetupWidget() and also 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 should be called to scan the sensor.

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

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

Parameters:

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

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

void CapSense_CSDScan (void)

This function performs scanning of all 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.

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

This function should be called when no scanning in progress. I.e. CapSense_IsBusy() returns a non-busy status. The widget must be preconfigured by the CapSense_CSDSetupWidget() function if other widget was previously scanned or other type of scan functions were 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 Cmod pre-charging.
4. Starts single scanning.

It is not recommended to call this function directly from the application layer. This function should be used to implement only the user's specific use cases (for faster execution time or pipeline scanning for example). This function should be called when no scanning 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 the next scan if a previous scan was triggered by using the CapSense_CSDScanExt() function. In this case, calling CapSense_CSDSetupWidgetExt() is not required every time before the CapSense_CSDScanExt() function. If a previous scan was triggered in any other way: CapSense_Scan(), CapSense_ScanAllWidgets() or CapSense_RunTuner() (see the CapSense_RunTuner() function description for more details), the sensor must be preconfigured again by using the CapSense_CSDSetupWidgetExt() API prior to calling the CapSense_CSDScanExt() function.

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

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

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

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

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

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

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

Parameters:

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

Returns:

Returns the status of the specified widget calibration:

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

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

void CapSense_CSDConnectSns (CapSense_FLASH_IO_STRUCTconst * *snsAddrPtr*)

Connects a port-pin used by the sensor to the AMUX bus of the CapSense block while a sensor is being scanned. The function does not take into account if the sensor is a ganged sensor and connects only a specified pin.

Scanning should be completed before calling this API.

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

Parameters:

snsAddrPtr	Specify the pointer to the FLASH_IO_STRUCT object belonging to a sensor which

	should be connected to the CapSense block.

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

void CapSense_CSDDisconnectSns (CapSense_FLASH_IO_STRUCTconst * *snsAddrPtr*)

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

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

Parameters:

snsAddrPtr	Specify the pointer to the FLASH_IO_STRUCT 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 also does not start a scanning process. The CapSense_CSXScan() function must be called after initializing the widget to start scanning.

This function should be called when no scanning 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.

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

Parameters:

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

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

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

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

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

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

Parameters:

widgetId	Specify the ID number of the widget to perform hardware and firmware initialization required for scanning a specific sensor in a specific widget. A macro for the widget ID can be found in the CapSense Configuration header file defined as

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

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

void CapSense_CSXScan (void)

This function performs scanning of all 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.

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

This function should be called when no scanning in progress. I.e. CapSense_IsBusy() returns a non-busy status. The widget must be preconfigured by the CapSense_CSXSetupWidget() function if other widget was previously scanned or 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 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.

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

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

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

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

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

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

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

Parameters:

widgetId	Specify the ID number of the CSX widget to calibrate its raw count. A macro for the

PRELIMINARY

	widget ID can be found in the CapSense Configuration header file defined as CapSense_<WidgetName>_WDGT_ID.
target	Specify the calibration target in percentages of the maximum raw count.

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

void CapSense_CSXConnectTx (CapSense_FLASH_IO_STRUCTconst * *txPtr*)

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

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

Parameters:

txPtr	Specify the pointer to the FLASH_IO_STRUCT object belonging to a sensor which should be connected to the CapSense block as Tx pin.

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

void CapSense_CSXConnectRx (CapSense_FLASH_IO_STRUCTconst * *rxPtr*)

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

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

Parameters:

rxPtr	Specify the pointer to the FLASH_IO_STRUCT object belonging to a sensor which should be connected to the CapSense block as Rx pin.

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

void CapSense_CSXDisconnectTx (CapSense_FLASH_IO_STRUCTconst * *txPtr*)

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

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

Parameters:

txPtr	Specify the pointer to the FLASH_IO_STRUCT object belonging to a Tx pin sensor which should be disconnected from the CapSense block.

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

void CapSense_CSXDisconnectRx (CapSense_FLASH_IO_STRUCTconst * *rxPtr*)

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

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

Parameters:

rxPtr	Specify the pointer to the FLASH_IO_STRUCT object belonging to a Rx pin sensor which should be disconnected from the CapSense block.

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

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

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

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

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

Parameters:

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

Returns:

Returns the status of operation:

- CYRET_SUCCESS if the operation is successfully completed.
- CYRET_BAD_PARAM if the input parameter is invalid.

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

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

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

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

- RAM Data Structure if W bit is 1.

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

Parameters:

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

Returns:

Returns the status of operation:

- CYRET_SUCCESS if the operation is successfully completed.
- CYRET_BAD_PARAM if 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

- cystatus CapSense_AdcStartConvert(uint8 chId)
  Initializes the hardware and initiates an analog-to-digital conversion on the selected input channel.
- uint8 CapSense_AdcIsBusy(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.
- cystatus 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

cystatus 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_AdcIsBusy() 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:

chId	The ID of the channel to be converted.

Returns:

The function returns cystatus of its operation.

- CYRET_SUCCESS - A conversion has started.
- CYRET_LOCKED - The hardware is already in-use by a previously initialized conversion or other functionality. No new conversion is started by this API.
- CYRET_BAD_PARAM - An invalid channel Id. No conversion is started.

Go to the top of the ADC Application Public Interface section.

uint8 CapSense_AdcIsBusy (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.

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:

chId	The ID of the channel to be measured

Returns:

The function returns voltage in milli-volts 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:

chId	The ID of the channel to be measured

Returns:

The function returns a voltage in milli-volts or CapSense_AdcVALUE_BAD_CHAN_ID if chId is invalid.

Go to the top of the ADC Application Public Interface section.

cystatus 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.

- CYRET_SUCCESS - The block is configured for the ADC use.
- CYRET_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 should be called when no ADC conversion is in progress.

Go to the top of the ADC Application Public Interface section.

void CapSense_AdcResume (void)

This function 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 critical sections in the code. In an unavoidable situation, the critical section is used and the code is optimized for the shortest execution time.

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

These API should not be used in the application layer.

Functions

- CY_ISR(CapSense_AdcIntrHandler)
  This is an internal ISR function for the ADC implementation.
- CY_ISR(CapSense_CSDPostSingleScan)
  This is an internal ISR function for single-sensor scanning implementation.
- CY_ISR(CapSense_CSDPostMultiScan)
  This is an internal ISR function for multiple-sensor scanning implementation.
- CY_ISR(CapSense_CSDPostMultiScanGanged)
  This is an internal ISR function for multiple-sensor scanning implementation for ganged sensors.

- CY_ISR(CapSense_CSXScanISR)
  This is an internal ISR function to handle CSX sensing method operation.

Function Documentation

CY_ISR (CapSense_AdcIntrHandler)

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.

CY_ISR (CapSense_CSDPostSingleScan)

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

The following tasks are performed for CSDv1 HW IP block:

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

The following tasks are performed for CSDv2 HW IP block:

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

The ISR handler changes 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 a component's generated code. Refer to the Macro Callbacks section of PSoC Creator User Guide for details.

Go to the top of the Interrupt Service Routine section.

CY_ISR (CapSense_CSDPostMultiScan)

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

The following tasks are performed:

1. Disable the CSD interrupt
2. Read the Counter register and update the data structure with raw data
3. Connect the Vref buffer to the AMUX bus
4. Disable the CSD block (after the widget has been scanned)
5. Update 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 a component's generated code. Refer to the Macro Callbacks section of PSoC Creator User Guide for details.

Go to the top of the Interrupt Service Routine section.

CY_ISR (CapSense_CSDPostMultiScanGanged)

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

The following tasks are performed:

1. Disable the CSD interrupt
2. Read the Counter register and update the data structure with raw data
3. Connect the Vref buffer to the AMUX bus
4. Disable the CSD block (after the widget has been scanned)
5. Update 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 a component's generated code. Refer to the Macro Callbacks section of PSoC Creator User Guide for details.

Go to the top of the Interrupt Service Routine section.

CY_ISR (CapSense_CSXScanISR)

This handler covers the following functionality:

- Read result of measurement a 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 scan of the current sensor of the number of bad conversions is greater than Noise Metric Threshold.
- Initiate scan of the next sensor for the multiple-sensor scanning mode.
- Update the Status register in the data structure.
- Switch CSDv2 HW IP block to the default state is scanning of all sensors is completed.

Go to the top of the Interrupt Service Routine section.

Macro Callbacks

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

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

- Define a macro to signal the presence of a callback (in cyapicallbacks.h). This will "uncomment" the function call from the component's source code.
- Write the function declaration using the provided in the table name (in cyapicallbacks.h). 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

Macro Callback Function Name	Associated Macro	Description
		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_STRUCTCapSense_dsRam

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

API Constants

Description

The section documents the CapSense component related API Constants.

Variables

- const CapSense_FLASH_STRUCT CapSense_dsFlash

PRELIMINARY

- const CapSense_FLASH_IO_STRUCT CapSense_ioList[CapSense_TOTAL_ELECTRODES]
- const CapSense_SHIELD_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_STRUCTCapSense_dsFlash

Constant for the FLASH Data Structure

const CapSense_FLASH_IO_STRUCTCapSense_ioList[CapSense_TOTAL_ELECTRODES]

The array of the pointers to the electrode specific register.

const CapSense_SHIELD_IO_STRUCTCapSense_shieldIoList[CapSense_CSD_TOTAL_SHIELD_COUNT]

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

const CapSense_FLASH_IO_STRUCTCapSense_adcIoList[CapSense_ADC_TOTAL_CHANNELS]

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

Data Structure

Description

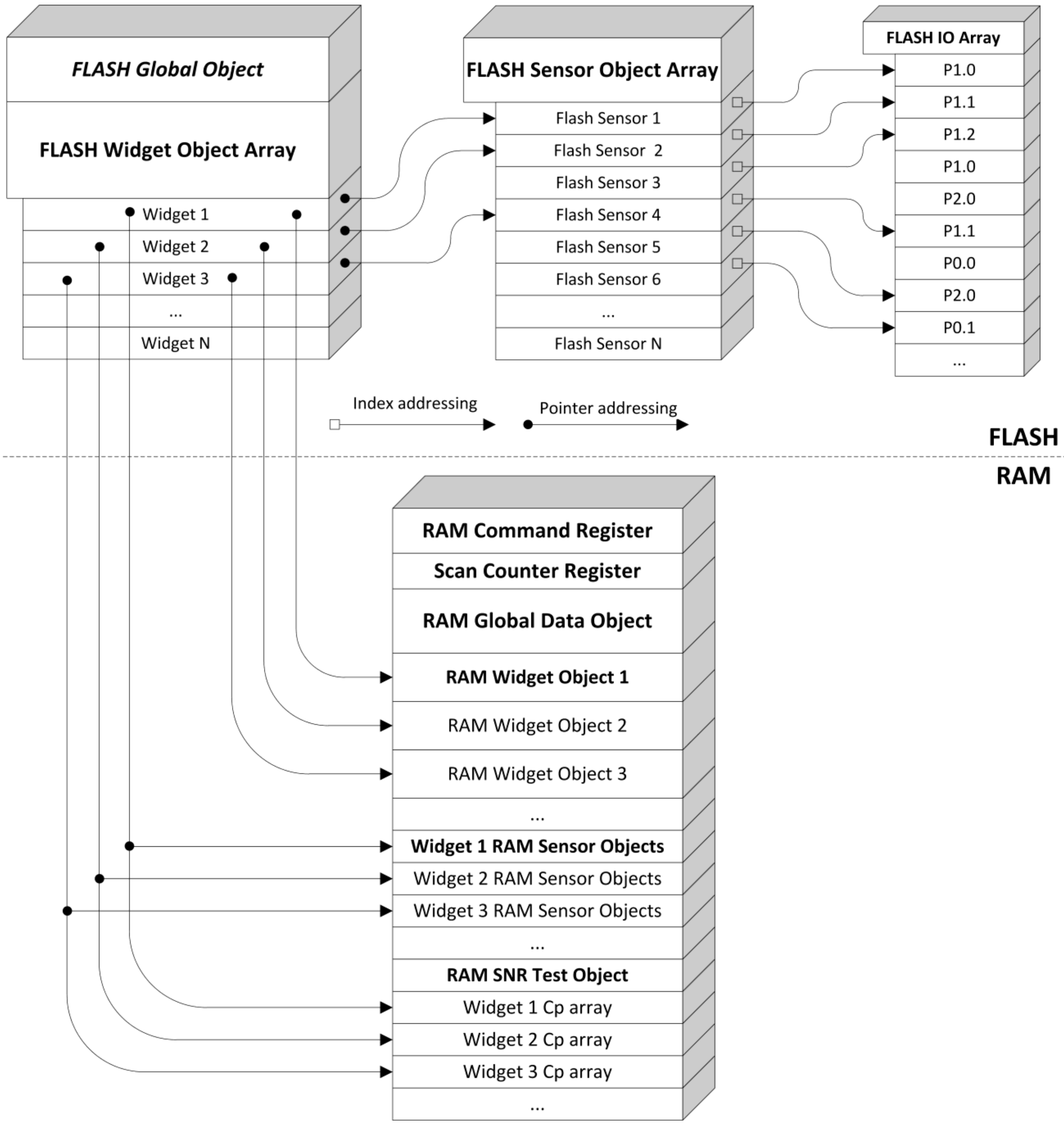
This section provides the list of structures/registers available in the component.

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>)
#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.
- 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_SELF_TEST_STRUCT
  Declare self test data structure.
- struct CapSense_RAM_SNS_CP_STRUCT
  Declare sensor Cp data structure.
- struct CapSense_RAM_STRUCT
  Declares the top-level RAM Data Structure.
- struct CapSense_FLASH_IO_STRUCT
  Declares the Flash IO object.
- struct CapSense_FLASH_SNS_STRUCT
  Declares the Flash Electrode object.
- struct CapSense_FLASH_SNS_LIST_STRUCT
  Declares the structure with all Flash electrode objects.
- struct CapSense_FLASH_WD_STRUCT
  Declares Flash widget object.
- struct CapSense_FLASH_STRUCT
  Declares top-level Flash Data Structure.
- struct CapSense_SHIELD_IO_STRUCT
  Declares the Flash IO structure for Shield electrodes.
- struct CapSense_BSLN_RAW_RANGE_STRUCT
  Define the structure for test of baseline and raw count limits which will be determined by user for every sensor grounding on the manufacturing specific data.

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_THRESHOLD_TY	fingerTh	Widget Finger Threshold.

PRELIMINARY

PE		
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_BUTTON_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_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_LO	lowBslnRst	The widget low baseline reset count. Specifies the

W_BSLN_RST_TYPE		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_SLIDER_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_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.
uint16	position[CapSense_NUM_CENTROIDS]	Reports the widget position.

struct CapSense_RAM_WD_CSD_MATRIX_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_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_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.
uint16	posX	The X coordinate.
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_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.
CapSense_THRESHOLD_TYPE	proxTouchTh	The proximity touch threshold.

struct CapSense_RAM_WD_CSX_MATRIX_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	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_STRUCT

Go to the top of the Data Structures section.

PRELIMINARY

Data Fields:

CapSense_RAM_WD_BUTTON_STRUCT	button0	Button0 widget RAM structure
CapSense_RAM_WD_SLIDER_STRUCT	linearslider0	LinearSlider0 widget RAM structure
CapSense_RAM_WD_SLIDER_STRUCT	radialslider0	RadialSlider0 widget RAM structure
CapSense_RAM_WD_CSD_MATRIX_STRUCT	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	matrixbuttons1	MatrixButtons1 widget RAM structure

struct CapSense_RAM_SNS_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	raw[CapSense_NUM_SCAN_FREQS]	The sensor raw counts.
uint16	bsln[CapSense_NUM_SCAN_FREQS]	The sensor baseline.
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_STRUCT

Go to the top of the Data Structures section.

Data Fields:

CapSense_RAM_SNS_STRUCT	button0[CapSense_BUTTON0_NUM_SENSORS]	Button0 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	linearslider0[CapSense_LINEARSLIDER0_NUM_SENSORS]	LinearSlider0 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	radialslider0[CapSense_RADIALSLIDER0_NUM_SENSORS]	RadialSlider0 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	matrixbuttons0[CapSense_MATRIXBUTTONS0_NUM_COLS+CapSense_MATRIXBUTTONS0_NUM_ROWS]	MatrixButtons0 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	touchpad0[CapSense_TOUCHPAD0_NUM_COLS+CapSense_TOUCHPAD0_NUM_ROWS]	Touchpad0 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	proximity0[CapSense_PROXIMITY0_NUM_SENSORS]	Proximity0 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	button1[CapSense_BUTTON1_NUM_SENSORS]	Button1 sensors RAM structures array
CapSense_RAM_SNS_STRUCT	matrixbuttons1[(CapSense_MATRIXBUTTONS1_NUM_RX)*(CapSense_MATRIXBUTTONS1_NUM_TX)]	MatrixButtons1 sensors RAM structures array

struct CapSense_RAM_SELF_TEST_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint32	testResultMask	Bit mask of test results (PASS/FAIL)
uint16	extCap[CapSense_TST_EXT_CAPS_NUM]	The capacitance of an external capacitor
uint16	intCap[2u]	The capacitance of an internal capacitor
uint16	vddaVoltage	The result of Vdda measurement (mV)
uint16	shieldCap	The shield capacitance
uint16	glbCrcCalc	A calculated CRC for global Component Data
uint16	wdgtCrcCalc	The widget data structure calculated CRC if the correspondent test result bit is set
uint8	wdgtCrcId	The global data structure calculated CRC if the correspondent test result bit is set
uint8	invBslnWdgtId	The first widget ID with mismatched baseline
uint8	invBslnSnsId	The first sensor ID with mismatched baseline
uint8	shortedWdgtId	The first shorted to GND/VDDA widget ID
uint8	shortedSnsId	The first shorted to GND/VDDA sensor ID
uint8	p2pWdgtId	The first widget ID with a sensor shorted to another sensor
uint8	p2pSnsId	The first sensor ID shorted to another sensor

struct CapSense_RAM_SNS_CP_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint8	button0[CapSense_BUTTON0_NUM_SENSORS]	Sensor Cp data for Button0 widget
uint8	linearslider0[CapSense_LINEARSLIDER0_NUM_SENSORS]	Sensor Cp data for LinearSlider0 widget
uint8	radialslider0[CapSense_RADIALSLIDER0_NUM_SENSORS]	Sensor Cp data for RadialSlider0 widget
uint8	matrixbuttons0[CapSense_MATRIXBUTTONS0_NUM_COLS+CapSense_MATRIXBUTTONS0_NUM_ROWS]	Sensor Cp data for MatrixButtons0 widget
uint8	touchpad0[CapSense_TOUCHPAD0_NUM_COLS+CapSense_TOUCHPAD0_NUM_ROWS]	Sensor Cp data for Touchpad0 widget
uint8	proximity0[CapSense_PROXIMITY0_NUM_SENSORS]	Sensor Cp data for Proximity0 widget
uint8	button1[CapSense_BUTTON1_NUM_SENSORS+1u]	Sensor Cp data for Button1 widget
uint8	matrixbuttons1[CapSense_MATRIXBUTTONS1_NUM_RX+CapSense_MATRIXBUTTONS1_NUM_TX]	Sensor Cp data for MatrixButtons1 widget

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	wdgtWorking[CapSense_WDGT_STATUS_WORDS]	The bitmask that reports the self-test status of all Widgets, each bit corresponds to one widget.
uint32	wdgtStatus[CapSense_W	The bitmask that reports activated Widgets (widgets that

	DGT_STATUS_WORDS]	detect a touch signal above the threshold), each bit corresponds to one widget.
CapSense_SNS_STS_TYPE	snsStatus[CapSense_TOTAL_WIDGETS]	For Buttons, Sliders, Matrix Buttons and CSD Touchpad each bit reports status of the individual sensor of the widget: 1 - active (above the finger threshold); 0 - inactive; For the CSD Touchpad and CSD Matrix Buttons, the column sensors occupy the least significant bits. For the Proximity widget, each sensor uses two bits with the following meaning: 00 - Not active; 01 - Proximity detected (signal above finger threshold); 11 - A finger touch detected (signal above the touch threshold); For the CSX Touchpad Widget, this register provides a number of the detected touches. The array size is equal to the total number of widgets. The size of the array element depends on the max number of sensors per widget used in the current design. It could be 1, 2 or 4 bytes.
uint16	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. The array size is equal to the number of ADC channels used in the project.
uint8	adcStatus	Stores the status of ADC.
uint8	adcIdac	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	modCsxClk	The modulator clock divider for the CSX widgets.
uint16	snsCsdClk	The global sense clock divider for the CSD widgets.
uint16	snsCsxClk	Global sense clock divider for the CSX widgets.
uint8	adcResolution	Stores the ADC resolution.
uint8	adcAzTime	Stores the AZ time used for ADC conversion.
uint16	glbCrc	CRC for global data.
CapSense_RAM_WD_LIST_STRUCT	wdgtList	RAM Widget Objects.
CapSense_RAM_SNS_LIST_STRUCT	snsList	RAM Sensor Objects.
CapSense_RAM_SELF_TEST_STRUCT	selfTest	The self test data structure.
CapSense_RAM_SNS_CP_STRUCT	snsCp	The sensor Cp Measurement data structures.
uint8	snrTestWidgetId	The selected widget ID.
uint8	snrTestSensorId	The selected sensor ID.
uint16	snrTestScanCounter	The scan counter.
uint16	snrTestRawCount[CapSense_NUM_SCAN_FREQS]	The sensor raw counts.

struct CapSense_FLASH_IO_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

struct CapSense_FLASH_SNS_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

Go to the top of the Data Structures section.

Data Fields:

CapSense_FLASH_SNS_STRUCT	button0[CapSense_BUTTON0_NUM_SENSORS]	Button0 FLASH electrodes array

struct CapSense_FLASH_WD_STRUCT

Go to the top of the Data Structures section.

Data Fields:

void const *	ptr2SnsFlash	Points to the array of the FLASH Sensor Objects or FLASH IO Objects that belong to this widget. Sensing block uses this pointer to access and configure IOs for the scanning. Bit #2 in WD_STATIC_CONFIG field indicates the type of array: 1 - Sensor Object; 0 - IO Object.
void *	ptr2WdgtRam	Points to the Widget Object in RAM. Sensing block uses it to access scan parameters. Processing uses it to access threshold and widget specific data.
CapSense_RAM_SNS_STRUCT*	ptr2SnsRam	Points to the array of Sensor Objects in RAM. The sensing and processing blocks use it to access the scan data.

void *	ptr2FltrHistory	Points to the array of the Filter History Objects in RAM that belongs to this widget.
uint8 *	ptr2DebounceArr	Points to the array of the debounce counters. The size of the debounce counter is 8 bits. These arrays are not part of the data structure.
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	csdblk	Specifies the CSD block to use on the capable devices. Available only if CSD2x or CSX2x is enabled.
uint16	slotIndex	Slot index occupied by the first sensor in the widget
uint8	senseMethod	Specifies the widget sensing method that could be either WD_CSD_SENSE_METHOD or WD_CSX_SENSE_METHOD
uint8	numCols	For CSD Button and Proximity Widgets, the number of sensors. For CSD Slider Widget, the number of segments. For CSD Touchpad and Matrix Button, the number of the column sensors. For CSX Button, Touchpad and Matrix Button, the number of the Rx electrodes.
uint8	numRows	For CSD Touchpad and Matrix Buttons, the number of the row sensors. For the CSX Button, the number of the Tx electrodes (constant 1u). For CSX Touchpad and Matrix Button, the number of the Tx electrodes.
uint16	xResolution	Sliders: The Linear/Angular resolution. Touchpad: The X-Axis resolution.
uint16	yResolution	Touchpad: The Y-Axis resolution.
uint32	xCentroidMultiplier	The pre-calculated X resolution centroid multiplier used for the X-axis position calculation. Calculated as follows: RADIAL: (WD_X_RESOLUTION * 256) / WD_NUM_COLS; LINEAR: (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);
uint8 *	ptr2SnsCpArr	The pointer to the array with the electrode capacitance value in pF.
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_WD_STRUCT	wdgtArray[CapSense_TOTAL_WIDGETS]	Array of flash widget objects
CapSense_FLASH_SNS_LIST_STRUCT	eltdList	Structure with all Ganged Flash electrode objects

struct CapSense_SHIELD_IO_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

struct CapSense_BSLN_RAW_RANGE_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

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 4 (GCC)

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

Configuration	Memory Consumption	
	Flash	SRAM
Configuration #1: CSX Matrix Button – One widget with 4 Rx and 8 Tx. ADC (disabled): *Number of input channels* = 0.		
Configuration #1	< 4800	< 500
Configuration #1 + *Enable multi-frequency scan* is enabled	< 5200	< 1000
Configuration #2: CSX Touchpad – One widget with 9 Rx and 4 Tx. ADC (disabled): *Number of input channels* = 0.		
Configuration #2	< 7100	< 800
Configuration #2 + *Enable multi-frequency scan* is enabled	< 7500	< 1350
Configuration #3: CSD Buttons – Three widgets with 4, 3 and 3 sensors in each widget, and Manual tuning mode is selected. ADC (disabled): *Number of input channels* = 0 (except were noted).		
Configuration #3	< 5500	< 300
Configuration #3 + *Enable multi-frequency scan* is enabled	< 6000	< 450
Configuration #3 + *Enable self-test library* is enabled	< 10000	< 350
Configuration #3 + *SmartSense (Full Auto-Tune)* mode is selected	< 6600	< 400
Configuration #3 + All firmware raw count filters enabled. The following parameters are used to enable filters: *Enable IIR filter (First order)*, *Enable average filter (4-sample)* and *Enable median filter (3-sample)*.	< 6100	< 400
Configuration #3 + ADC (enabled): *Resolution (bits)* = 10-bit / *Number of input channels* = 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.

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 *PSoC® 4 CapSense® Design Guide*.

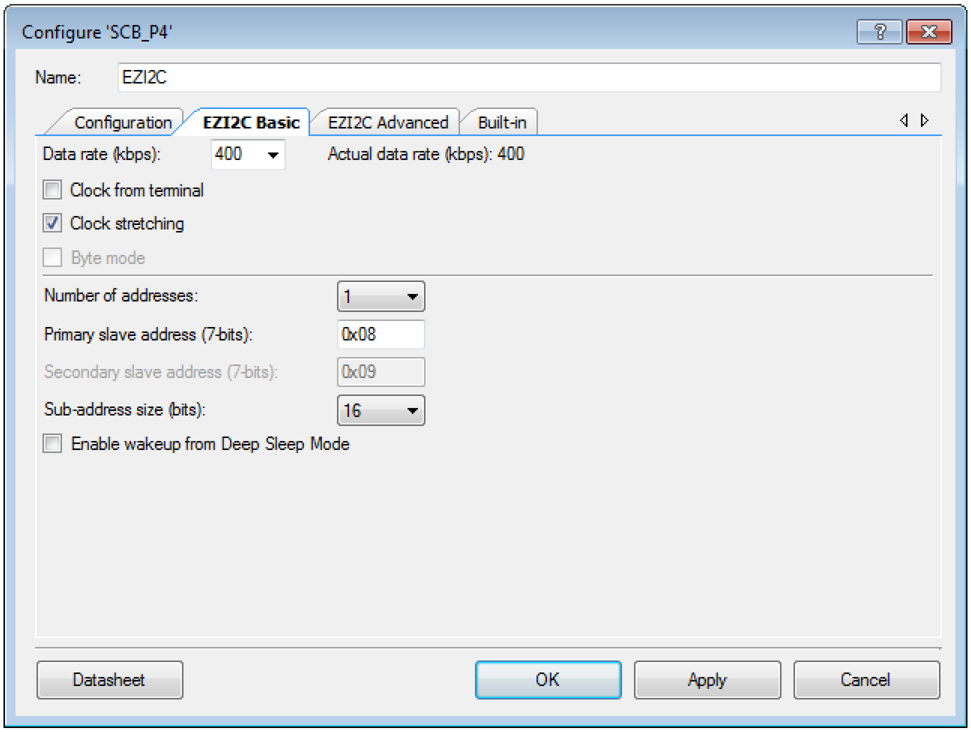
Tuning Quick Start

The following steps show how to set up CapSense tuning across an I^2C 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 Slave (SCB Mode) Component from the Component Catalog onto the schematic to add an I^2C communication interface to the project. This I^2C slave interface is required for the Tuner GUI to monitor 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 the Primary slave address (7-bits) to 0x08
- Set the Sub-address size (bits) to 16

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.

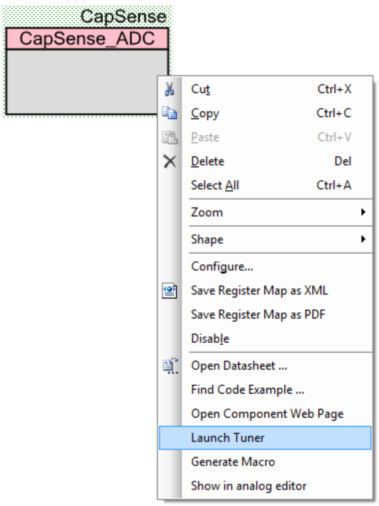
Step-3: Modify Application Code

Replace your *main.c* from the *Step-6* in the *Quick Start* section with the following code:

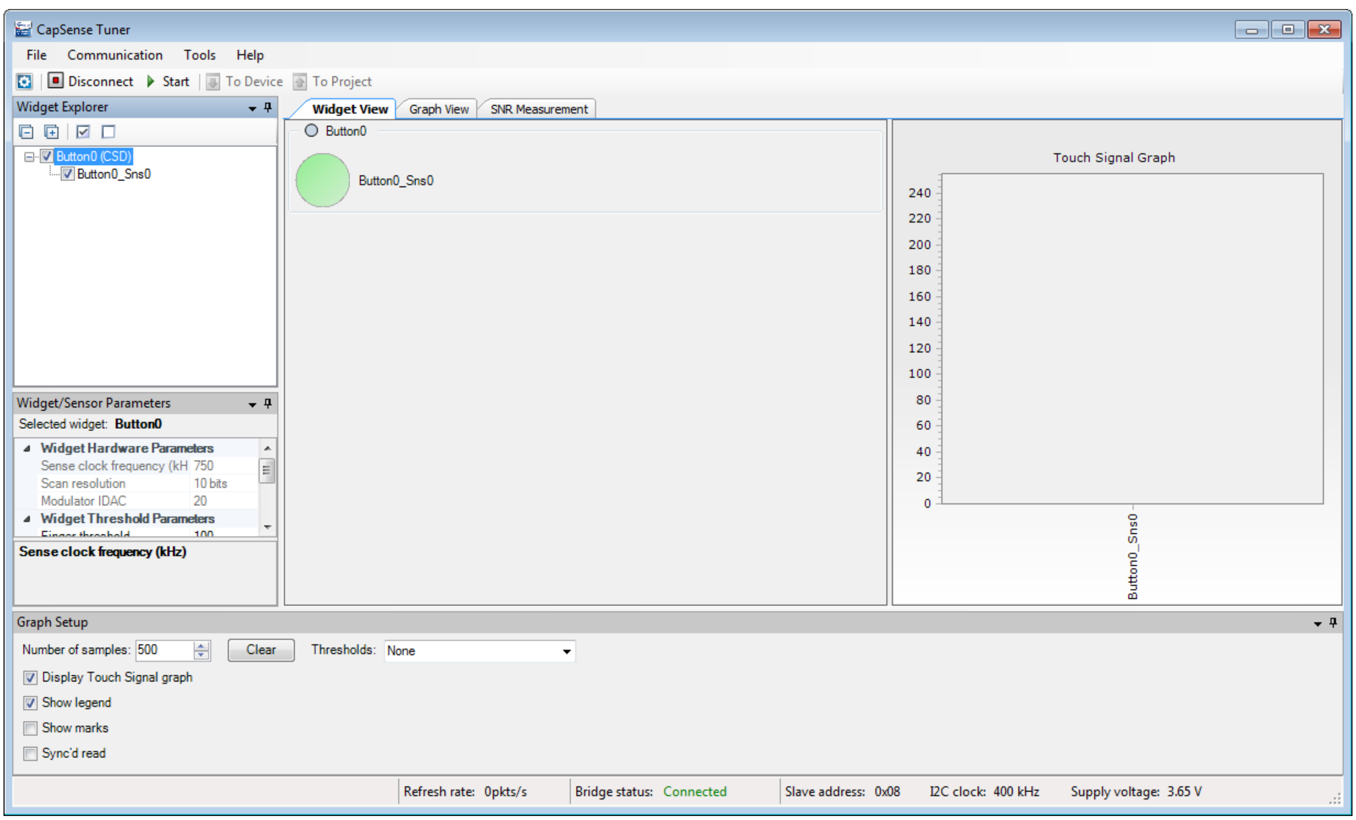
```
#include <project.h>

int main()
{
    uint16 adcResult;

    CyGlobalIntEnable;                          /* Enable global interrupts */

    EZI2C_Start();                              /* Start EZI2C Component */
    /* Set up communication and initialize data buffer to CapSense data structure */
    EZI2C_EzI2CSetBuffer1(sizeof(CapSense_dsRam), sizeof(CapSense_dsRam),
                          (uint8 *)&CapSense_dsRam);

    PWM_Start();                                /* Initialize the TCPWM */
    CapSense_Start();                           /* Initialize the CapSense */
    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 */
            {
                adcResult = CapSense_AdcReadResult_mVolts(CapSense_AdcCHANNEL_0);
                PWM_WriteCompare((1000u * adcResult) / CYDEV_VDDA_MV);
            }
            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 button widget, called Button0, is automatically shown in the Widget View panel.

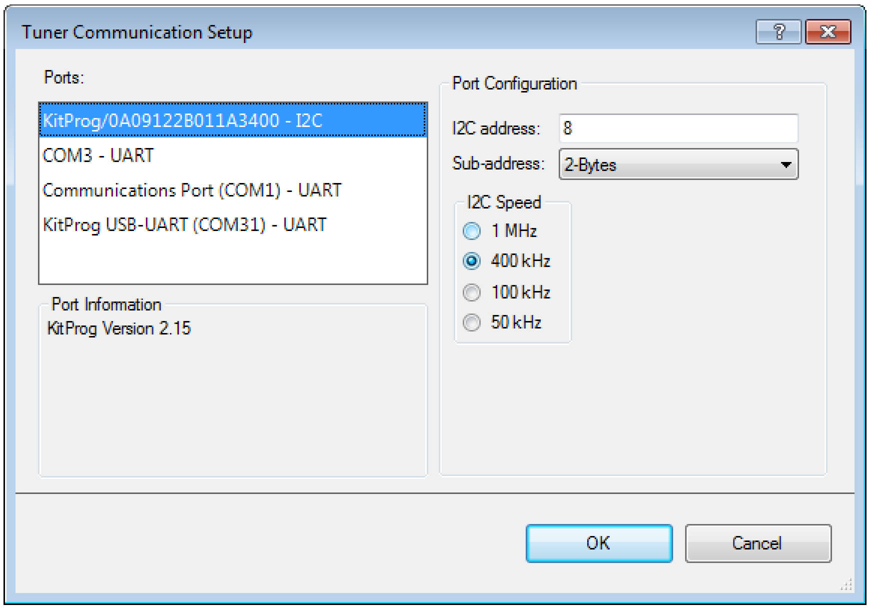


PRELIMINARY

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 I^2C communication device KitProg (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 *Step-1: Place and Configure an EZI2C Component*). Sub-address must be set to 2-Bytes in both places.

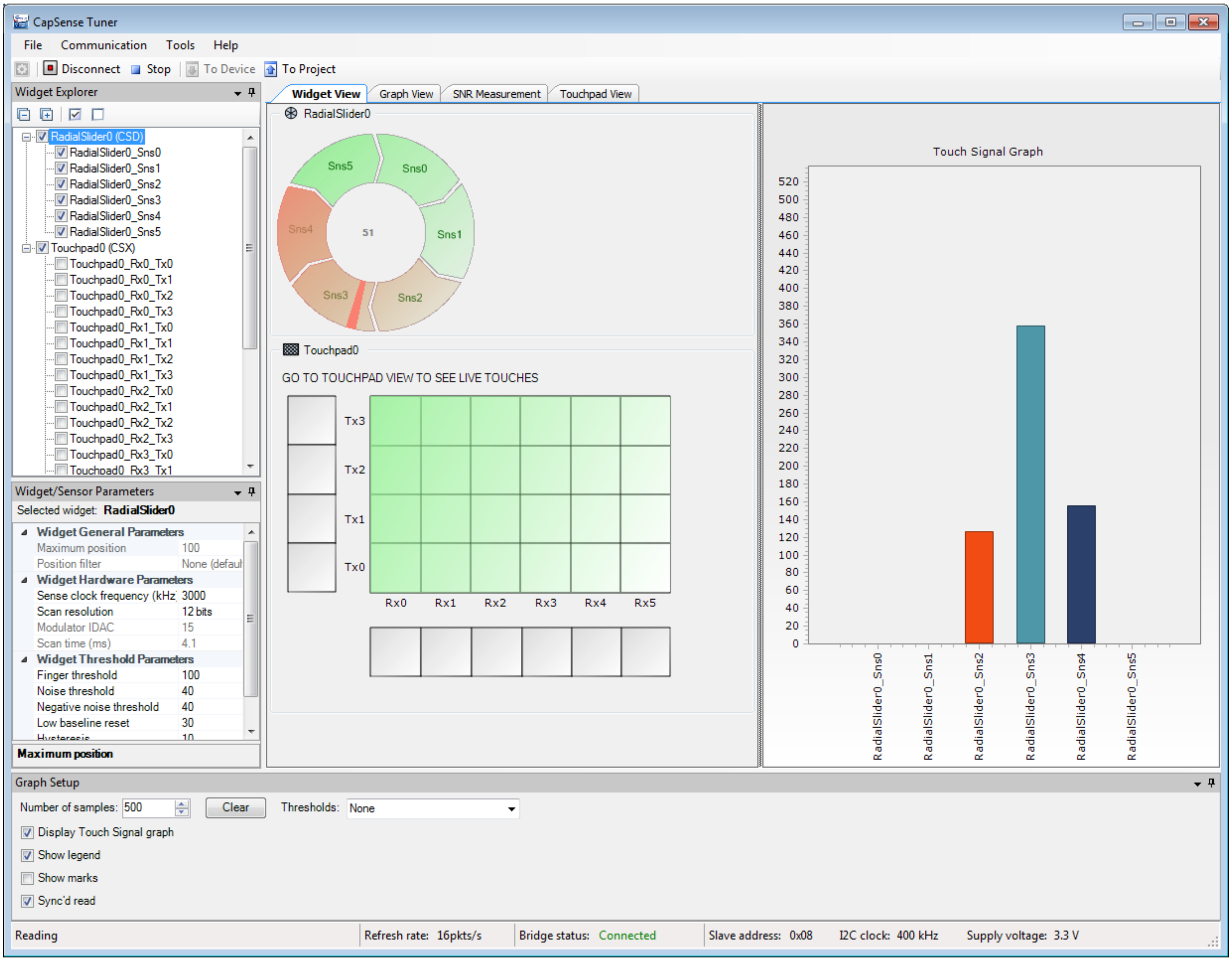
Step-6: Start Communication

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 Button0 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.

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 starts 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 EZI2C (not *CapSense_dsRam*)
  - 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.

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

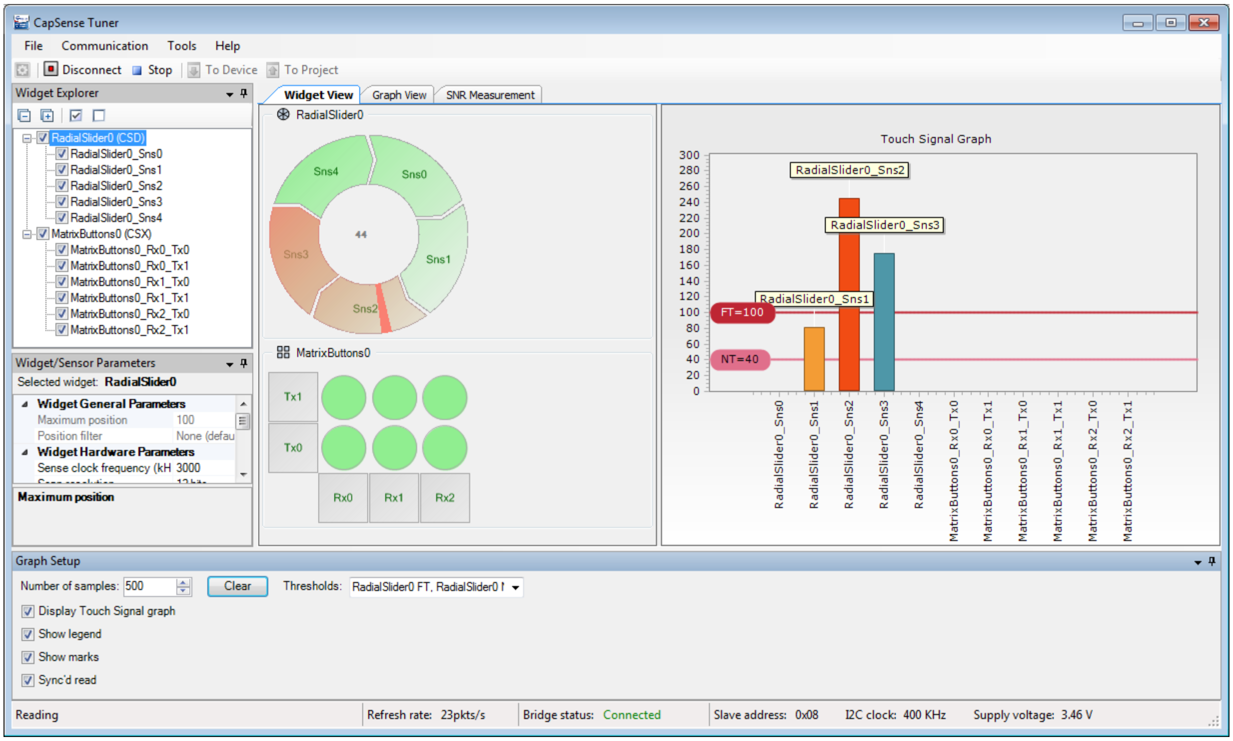
The Graph Setup pane provides quick access to different Tuner configuration options that affect the Tuner graphs display.

- **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.*

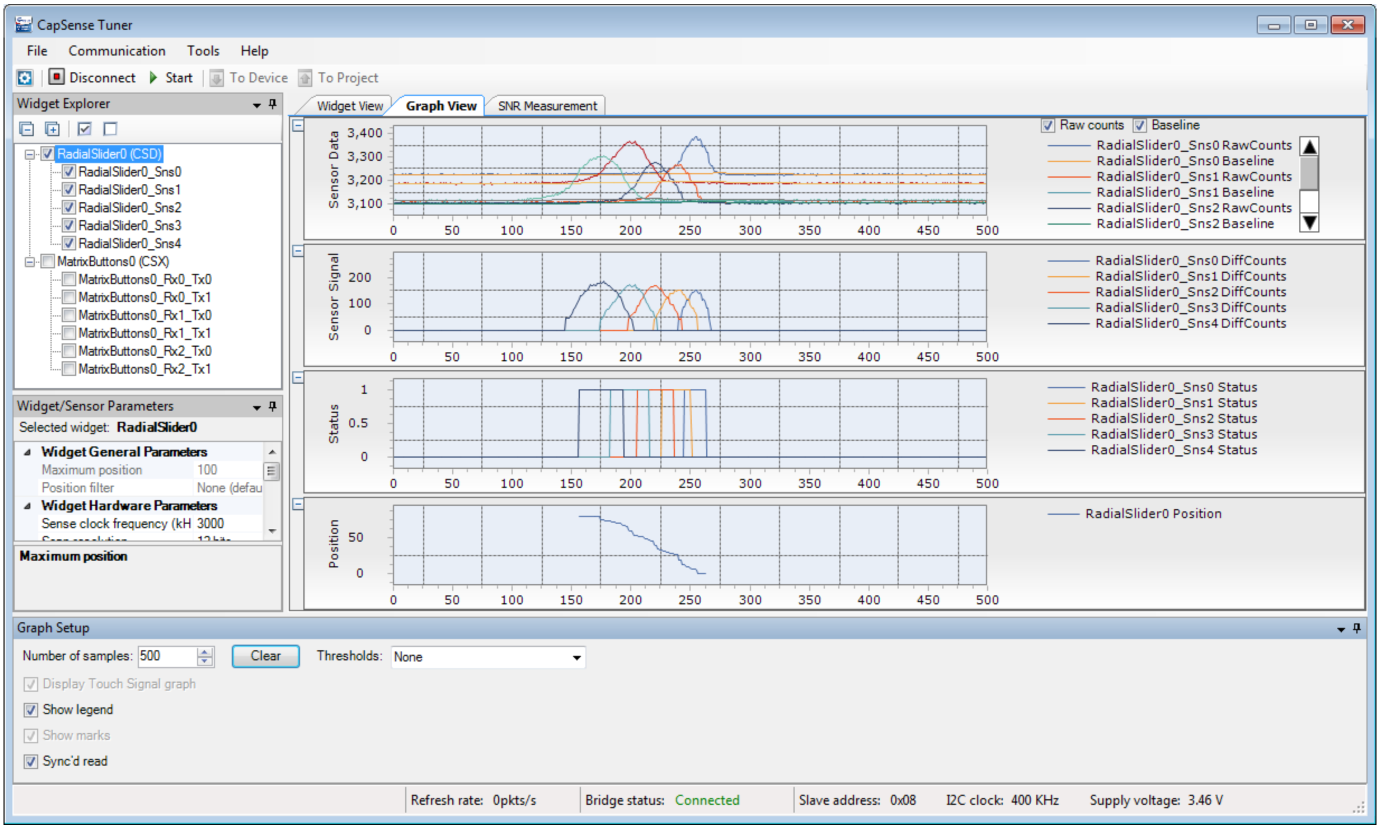
The Widget sensors are highlighted red when the device reports their touch status as active.

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*.

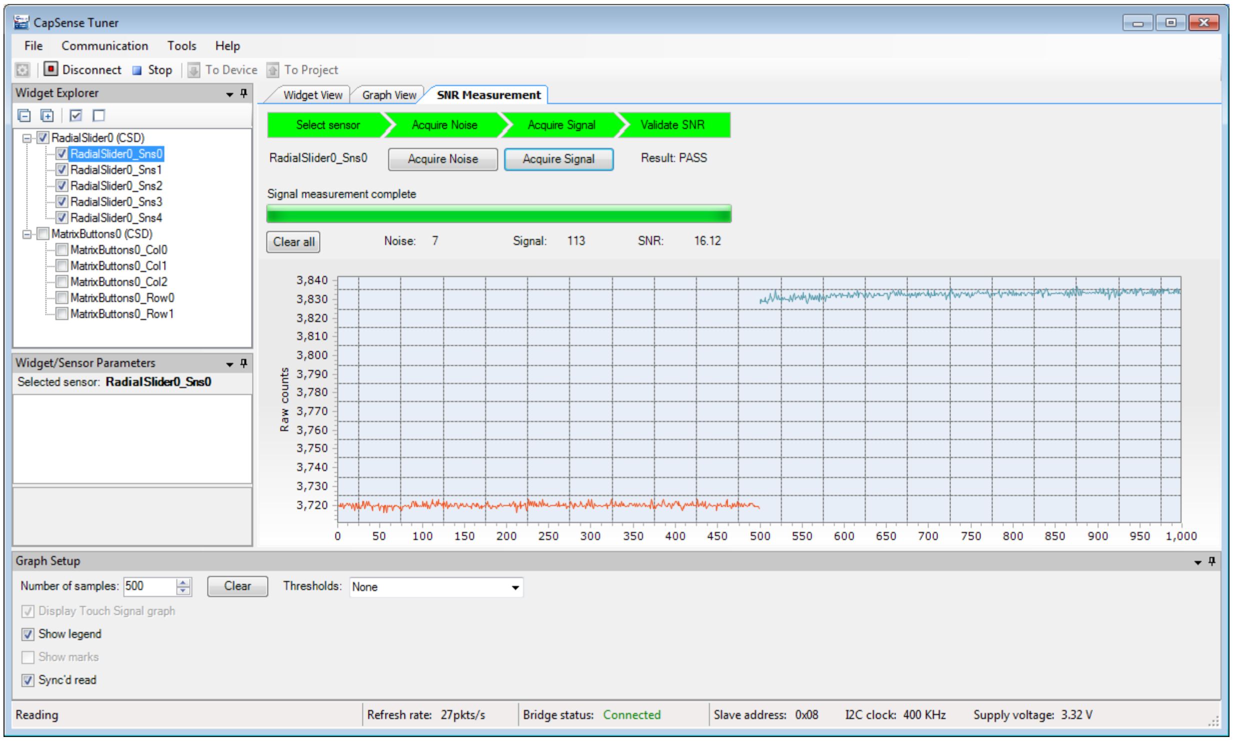
Graph View



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.

SNR Measurement



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

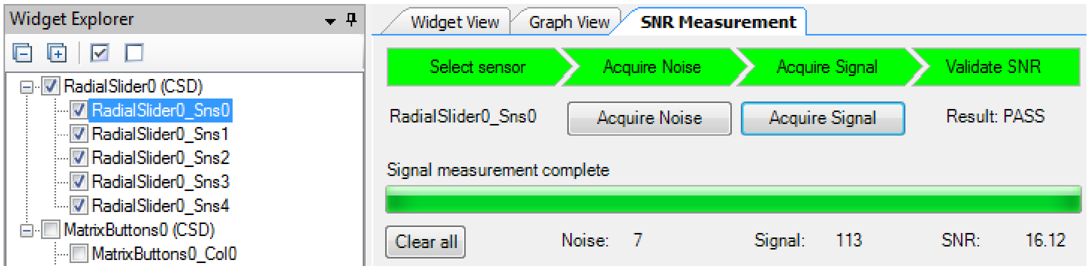
The tab provides UI to acquire noise and signal samples separately and then calculates a SNR based on the captured data. The obtained value is then validated by a comparison with the required minimum (5 by default, can be configured in the *Tuner Configuration Options*).

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
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).

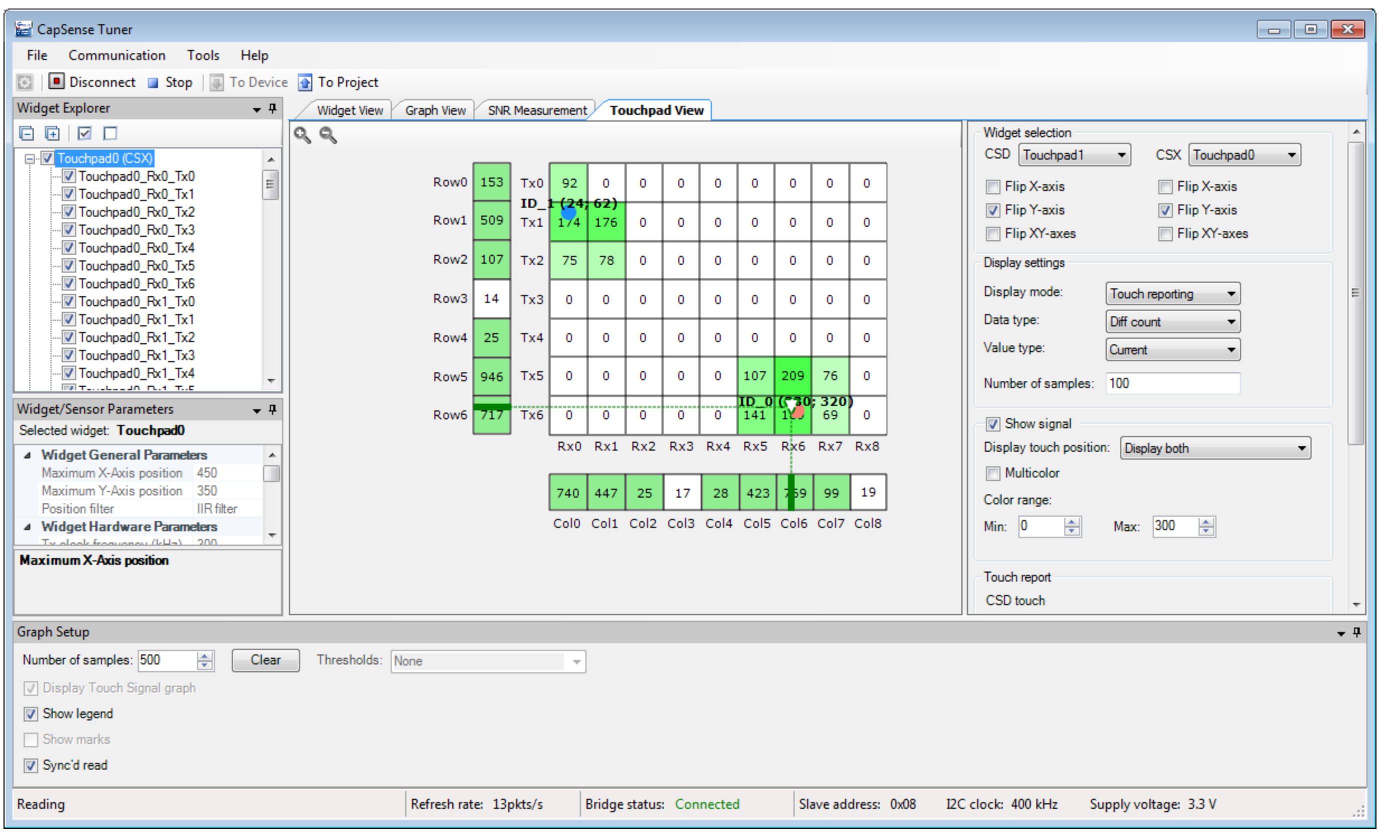
Below these, there is a status label displaying the current status message and progress bar displaying the progress of the current operation.

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 combo box** – Used to select any CSD touchpad displayed in the heatmap. The CSD combo box is grayed out if the CSD touchpad does not exist in the user design.
- **CSX combo box** – Used to select any CSX touchpad displayed in the heatmap. The CSX combo box is grayed out if the CSX touchpad does not exist in the user design.
- **Flip X-axis** – Flips the displayed X-axis correspondingly 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.

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** combo box. 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** combo box. If the CSX touchpad is neither configured nor touch-detected, the touch table is empty. The Component supports simultaneous detection up to three touches for a CSX touchpad touch, so the touch table displays all the detected touches.

Clear

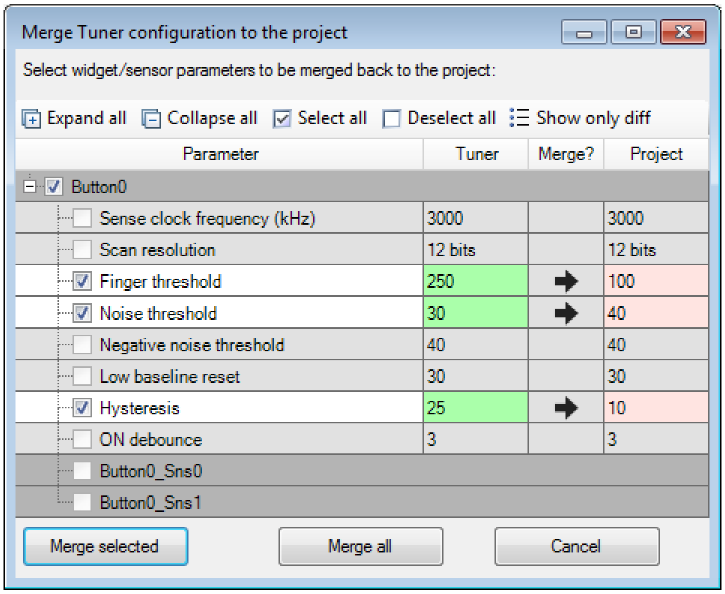
Clears all before drawn elements like lines, traces, etc.

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.

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:

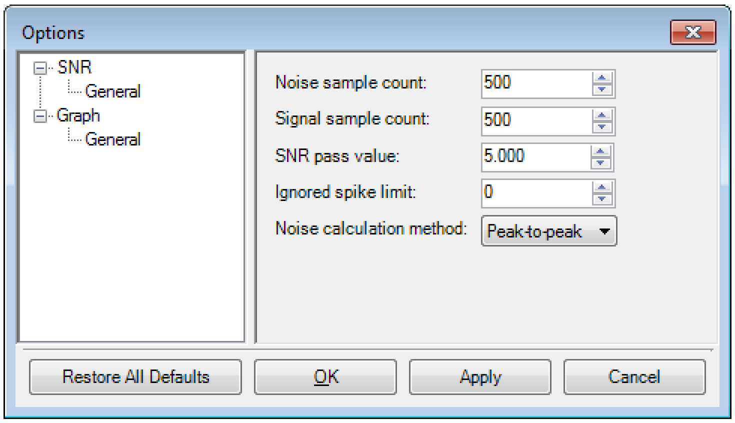
- *SmartSense Auto-tuning*
- CSD *Enable IDAC auto-calibration*
- CSX *Enable IDAC auto-calibration*

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

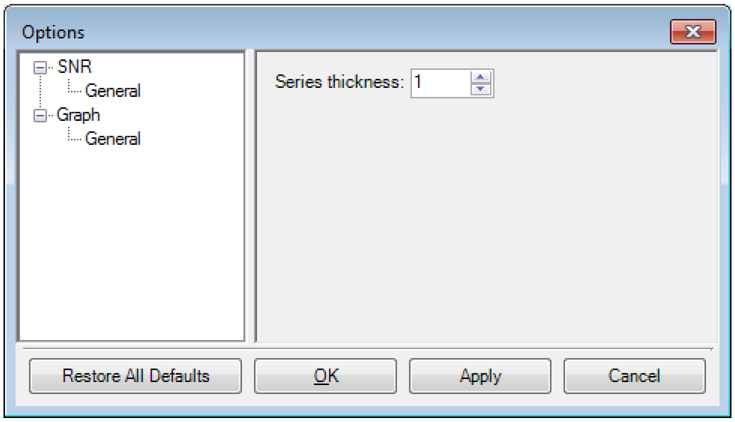
The Tuner application allows setting different configuration options with the Options dialog. Settings are applied on a project basis and divided into groups:

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_P4 Component has the following specific deviations:

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
8.8	R	An external object or function shall be declared in only one file.	Some arrays are generated based on the Component configuration and these arrays are declared locally in the .c source files where they are used instead of in .h include files.
11.4	A	A cast should not be performed between a pointer to object type and a different pointer to object type.	Pointers are used to allow many types of widgets and sensors. The architecture is designed to allow indexing a specific pointer.
12.13	A	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	These violations are reported for the GCC ARM optimized form of the "for" loop that have the following syntax: for(index = COUNT; index --> 0u;) It is used to improve performance.

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
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 Component has many different configurations. Some of them do not use specific function parameters. To avoid the complier's warning, the following code is used: (void)paramName.
16.7	A	A pointer parameter in a function prototype should be declared as the pointer to const if the pointer is not used to modify the addressed object.	Mostly all data processing for variety configuration, widgets and data types is required to pass the pointers as an argument. The architecture and design are intended for this casting.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	Pointers are used to allow many types of widgets and sensors. The architecture is designed to allow indexing a specific pointer.
18.4	R	Unions shall not be used.	There are two general cases in the code where this rule is violated. 1. <INSTANCE_NAME>_PTR_FILTER_VARIANT definition and usage. This union is used to simplify the pointer arithmetic with the Filter History Objects. Widgets may have two kinds of Filter History: Regular History Object and Proximity History Object. The mentioned union defines three different pointers: void, RegularObjPtr, and ProximityObjPtr. 2. APIs use unions to simplify operation with pointers on the parameters. The union defines four pointers: void*, uint8*, uint16*, and uint32*. In all cases, the pointers are verified for proper alignment before usage.
19.7	A	A function should be used in preference to a function-like macro.	Simple function-like macros are used to decrease execution time in time critical functions.

Component Debug Window

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

To open the Component Debug window:

1. Make sure the debugger is running or in the break mode.

2. Choose Windows > Components… from the Debug menu.
3. In the Component Window Selector dialog, select the Component instances to view and click OK.

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

Resources

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

Additionally, the following may be consumed:

- An additional analog mux bus is consumed with a shield electrode enabled in the *CSD Settings* tab.
- One 7-bit IDAC in the CSD block is not consumed (and available for general purpose use) when the Enable compensation IDAC is unselected in the **CSD Settings** tab and CSX *Sensing Mode* is disabled.
- Two analog buses and two 7-bit IDACs are used always when ADC Number of input channels is non-zero and either CSD or CSX mode is enabled.

References

General References

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

Application Notes

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

- *AN64846* – Getting Started with CapSense®
- *AN72362* – Reducing Radiated Emissions in Automotive CapSense® Applications
- *AN85951* – PSoC® 4 CapSense® Design Guide

- *AN92239* – Proximity Sensing with CapSense®

Code Examples

PSoC Creator provides access to code examples in the 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*. The examples that use this Component include:

- *CE210289 - PSoC®4 CapSense® Linear Slider*
- *CE210291 - PSoC® 4 CapSense® One Button*
- *CE210290 - PSoC® 4 CapSense® Low-Power Ganged Sensor*
- *CE210311 - CapSense® ADC Sequential*

Development Kit Boards

Cypress provides a number of development kits. You can access them at the *Cypress Development Kit web page*. Mentioned Code Examples uses the following development kits:

- *CY8CKIT-040 PSoC® 4000 Pioneer Kit*
- *CY8CKIT-042-BLE Bluetooth® Low Energy Pioneer Kit*
- *CY8CKIT-042 PSoC® 4 Pioneer Kit*
- *CY8CKIT-044 PSoC® 4 M-Series Pioneer Kit*
- *CY8CKIT-046 PSoC® 4 L-Series Pioneer Kit*
- *CY8CKIT-041 PSoC® 4 S-Series Pioneer Kit*

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 4100PS and PSoC Analog Coprocessor devices is not available at this time. Once the data is available, the Component datasheet will be updated on the Cypress web site.

CapSense Performance Characteristics

Parameter	Condition	Typical	Units
Sensor Calibration level (Applicable for sensor with highest Cp within a Widget)	Cp = 5 to 45 pF (Single IDAC mode)	85% of full scale ±5 %	-
	Cp = 5 to 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 capacitance >= 0.1 pF	> 5:1	-
	Cp < 45 pF Single IDAC: Finger capacitance >= 0.2 pF Dual IDAC: Finger capacitance >= 0.1 pF	> 4:1	-
Supply (VDD) ripple	V_{DD} > 3.3 V, Finger capacitance = 0.1 pF, V_{DD} ripple +/-50 mV	< 30% of noise	
	V_{DD} < 2 V, internally regulated mode, Finger capacitance = 0.4 pF, V_{DD} ripple +/-50 mV	< 30% of noise	
	V_{DD} < 2 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 *Fourth-generation CapSense* devices.	< 30% of noise	

PRELIMINARY

Parameter	Condition	Typical	Units
Tx Output Voltage	Logic High	> Vddd-0.6	V
	Logic Low	< 0.6	V
Voltage Reference (Vref) (*CSD sensing method*)	VDDA < 2.6V	1.2	V
	2.6V <= VDDA < 3.2V	1.477	V
	3.2V <= VDDA < 4.7V	2.021	V
	4.7V <= VDDA	2.743	V
Voltage Reference (Vref) (*CSX sensing method*)		1.2	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 Vpp = 20 V	< 30%	-
	10 kHz to 1 MHz, noise Vpp = 20 V, Cp < 10 pF	< 30%	-
Interrupt immunity Excessive raw counts noise at asynchronous interrupts is used.	SNR > 5:1	< 30%	-
Current Consumption	1 CSD Button Widget (Ganged Sensor, 4 electrodes). Resolution = 9 bits. Each electrode Cp < 10 pF. Shield Electrode = Disabled. SYSCLK = 16 MHz. No I2C traffic (I2C block ON). Report Rate >= 8 Hz. Chip state = DeepSleep (LFT).	< 7 (PSoC 4000S)	µA
	1 CSD Button Widget, 8 Sensors. Resolution = 9 bits. Each electrode Cp < 10 pF. Shield Electrode = Disabled. SYSCLK = 16 MHz. No I2C traffic (I2C block ON). Report Rate >= 8 Hz Chip state = DeepSleep (LFT).	< 22 (PSoC 4000S)	µA

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).	< 6 (PSoC 4000S)	µA
	1 CSX Touchpad Widget 32 nodes (9 x 4 electrodes). Num of sub-conversions = 25. SYSCLK = 16 MHz. Overlay => 1 mm plastic. 4.8 x 4.8 mm diamond sensors. 9mm metal finger. 1 Touch only. Report Rate >= 8 Hz. Chip state = DeepSleep (LFT).	< 200 (PSoC 4000S)	µA

ADC Performance Characteristics

Parameter	Min	Typ	Max	Units	Details/ Conditions
Voltage Reference (Vref)	-	1.2	-	V	VDDA < 2.733V
	-	2.133	-	V	2.733V <= VDDA < 4.5V
	-	3.840	-	V	4.5V <= VDDA
Resolution	-	-	10	bits	Auto-zeroing is required every millisecond
Number of channels - single ended	-	-	10		
Monotonicity	-	-	-	Yes	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	

Parameter	Min	Typ	Max	Units	Details/ Conditions
Input voltage range - single ended	V_{SSA}	-	V_{DDA}	V	
Input resistance	-	2.2	-	KΩ	
Input capacitance	-	20	-	pF	
Power supply rejection ratio	-	60	-	dB	In V_{REF} (2.4 V) mode with V_{DDA} bypass capacitance of 10 µF
Sample acquisition time	-	10	-	µs	
Conversion time for 8-bit resolution at clock frequency = 48 MHz.	-	-	10.7	µs	Does not include acquisition and processing time.
Conversion time for 10-bit resolution at clock frequency = 48 MHz.	-	-	42.7	µs	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 Characteristic

PSoC 4000S, PSoC 4100S:

Parameter	Description	Min	Typ	Max	Units	Conditions
$IDAC1_{DNL}$	DNL	-1	–	1	LSB	
$IDAC1_{INL}$	INL	-2	–	2	LSB	INL is ±5.5 LSB for VDDA < 2 V
$IDAC2_{DNL}$	DNL	-1	–	1	LSB	
$IDAC2_{INL}$	INL	-2	–	2	LSB	INL is ±5.5 LSB for VDDA < 2 V

PSoC 4100PS, PSoC Analog Coprocessor:

Parameter	Description	Min	Typ	Max	Units	Conditions
$IDAC1_{DNL}$	DNL	-1	–	1	LSB	
$IDAC1_{INL}$	INL	-3	–	3	LSB	
$IDAC2_{DNL}$	DNL	-1	–	1	LSB	
$IDAC2_{INL}$	INL	-3	–	3	LSB	

DC/AC Specifications

Refer to devices specific datasheet *PSoC 4 Device datasheets* for more details.

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
4.10	New Component version.	Fixed the errata item 287117 for the GetExtCapCapacitance() function.
4.0.a	Edited datasheet.	Added errata item 287117 to document issue with GetExtCapCapacitance() function.
4.0	Added support for PSoC 4100PS device family. Renamed ExitCall**B**ack () to ExitCall**b**ack (). Improved the Component.	Fixed issues documented in the following errata items, and removed the errata section: 242894, 253147, 260781, and 232921. Also removed errata item 215127 because this Component does not support the corresponding device. Added IDAC characteristic data.
3.10.b	Edited datasheet.	Removed errata item 248295, because this Component does not support the applicable device. Also added the following issues: 215127, 260781, 232921
3.10.a	Fixed Number of Subconversions equation.	Equation was incorrect.
3.10	Added the following features: CSX Touchpad support Self-test library Multi-frequency scan feature IDAC sinking mode in Fourth generation CapSense	Expanded functionality. Fixed potential issue with Auto mode. Documented potential issue with Inactive sensor connection to shield.
3.0.c	Edited datasheet.	Revised to correct omission of the APIs.
3.0.b	Edited datasheet.	Added Component Errata section to document potential issue with Auto mode.
3.0.a	Removed empty CapSense_SaveConfig() and CapSense_RestoreConfig() APIs	No usage of these API is expected in future.
	Renamed CapSense_IsProximityTouchActive() to CapSense_IsProximitySensorActive() without functionality change	Providing a meaningful name and being consistent with other APIs

Version	Description of Changes	Reason for Changes / Impact
	Changed Sensitivity parameter to Finger Capacitance	Providing a meaningful parameter with intuitive usage
	Added IDAC sensing configuration parameter with IDAC sinking mode	Expanded functionality
	Edited datasheet.	Final characterization data for PSoC 4000S, PSoC 4100S, and PSoC Analog Coprocessor devices is not available at this time. Once the data is available, the Component datasheet will be updated on the Cypress web site.
3.0	The initial version of new Component implementation. This version is not backward compatible with the previous versions. See *Migration Guide* for more information.	Improved implementation of the CapSense Component with PSoC 4 devices.

Migration Guide

CapSense P4 v4.X is a new Component, **not** backward-compatible with CapSense CSD P4 v2.X. So, a design that uses version 2.X requires manual migration to version v4.X to benefit from the new features and enhanced performance.

CapSense P4 is a completely new Component. Projects using CapSense CSD P4 v2.40 (and prior versions) cannot be automatically updated to the new Component. You must back up your project, replace the old Component with CapSense_P4, and set up the parameters as described below. Note that the firmware API is very different in the new Component and it is highly recommended that you read section *Step-7: API Comparison* in order to make changes in your firmware. It is highly recommended that all the new design must start with CapSense P4 v4.X, and the design that requires the features of CapSense P4 v4.X, such as mutual-cap sensing or low power, must be manually migrated. The existing designs in production or minor revisions of the existing product may use version 2.X, however, no further enhancements are planned on that version.

This section provides the guidelines migration to CapSense P4 v4.X. In general, the migration requires the following steps:

- *Step-1: Add Widget / Sensor*
- *Step-2: Parameters: Enable firmware filters*
- *Step-3: Parameters: CSD Settings*
- *Step-4: Parameters: Widgets Details*
- *Step-5: Scan Order*

- *Step-6: Pinout*
- *Step-7: API Comparison*

Differences in supported features

The table below shows the difference in the features supported by the v2.40 and v4.X Components.

CapSense CSD P4 (v2.40)	CapSense P4 (v4.0)	Comments
Gesture	Planned for a future version of the Component.	Contact Cypress customer support if your design requires this feature to provide a solution.
Generic widget type	Planned for a future version of the Component.	Use the Button widget as a replacement.
Guard Sensor	Replaced with the Button widget.	Use the Button widget type to create a guard sensor in the design.
Jitter filter for Raw Counts	Not supported	The Jitter filter is supported only for slider/touchpad positions; this filter is not very effective for noise suppression in raw counts, so use IIR, Median or average filters instead.
Widget Resolution (8-bit)	Not supported	Only the 16-bit widget resolution is supported.
Modulator clock frequency for each sensor	Not supported	The modulator clock frequency is set for the whole Component for optimized performance.
IDAC range (8x)	Not supported	CapSense designs do not require the 8x mode. In order to make the tuning simple, the 8x mode is removed.

Note If a device has more than one HW CSD block, different Component versions should not be used (i.e. do not place v3.0 in HW CSD block1 and v4.0 in HW CSD block2), such configuration is not guaranteed to be functional.

Step-1: Add Widget / Sensor

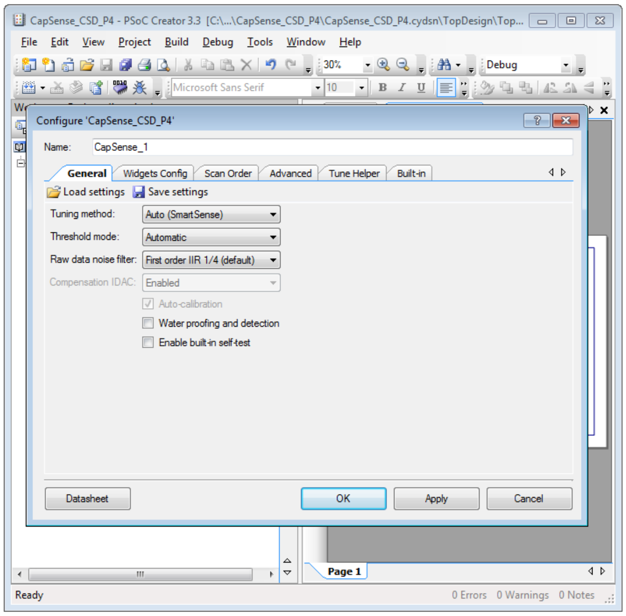
a. In v2.40, the **Widget Config** tab is used to create and configure the widgets, and *Basic Tab* is used to create widgets and *Widget Details* sub-tab under *Advanced Tab* is used to configure the widgets parameters in the v4.X Component.

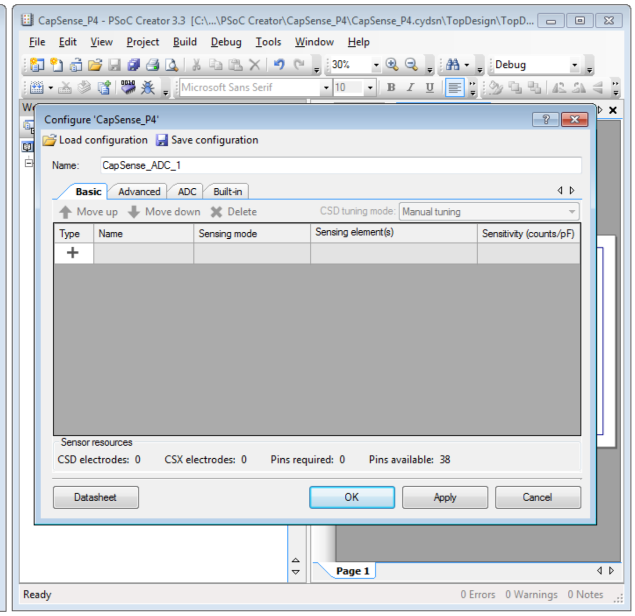
Behavior of all widgets is the same between both versions, except for the Button and Generic widgets. In CapSense v4.X, up to 32 sensors can be created under one *Button* widget, configure and scan all those sensors as a group of sensors called the widget. If the design is based on v2.40 and has multiple button sensors, consider creating several widgets and multiple sensors in the widgets for optimized performance.

Add the widgets in the *Basic Tab*, select a number of sensors or segments of those widgets.

PRELIMINARY

b. Select *CSD sensing method* for all widgets for *Sensing Mode*.





c. The tuning mode selection in the new Component is updated.

Note that there is only one selection in the new Component for the tuning mode instead of two parameters ("Tuning method" and "Threshold mode") in the old Component. The following table shows equivalent tuning modes in CapSense v4.X:

CapSense CSD P4 (v2.40)		CapSense P4 (v4.X)	Comments
Tuning Method	**Threshold Mode**	***CSD tuning mode***	
Auto (SmartSense)	Automatic	*SmartSense (Full Auto-Tune)*	Both *Widget Hardware Parameters* and *Widget Threshold Parameters* are auto-tuned. SmartSense in CapSense CSD P4 v2.40 enables the Compensation IDAC automatically. SmartSense in CapSense P4 v4.X is more flexible and allows operating with a disabled Compensation IDAC. To properly migrate enable the *Enable compensation IDAC* parameter (enabled by default).
Auto (SmartSense)	Flexible	*SmartSense (Hardware parameters only)*	*Widget Hardware Parameters* are auto-tuned and *Widget Threshold Parameters* can be set by users in the customizer.
Manual with run-time tuning	N/A	*Manual*	Set both *Widget Hardware Parameters* and *Widget Threshold*

CapSense CSD P4 (v2.40)		CapSense P4 (v4.X)	Comments
Tuning Method	**Threshold Mode**	***CSD tuning mode***	
Manual	N/A		*Parameters* manually in the customizer.

d. If *SmartSense Auto-tuning* was not used, skip this step.

The new Component has enhanced flexibility for *Finger capacitance* selection (Sensitivity in v2.40 Component) compared to only 10 selections in the v2.40 Component:

- from 0.1pF to 1pF in steps of 0.02pF in *SmartSense (Full Auto-Tune)* mode
- from 0.02pF to 20.48pF on the exponential scale in *SmartSense (Hardware parameters only)* mode.

The following table shows equivalent settings in the v4.X Component for 10 selections in the v2.40 Component. It is also acceptable to select a different user-set value for this parameter to benefit from the CapSense v4.X performance.

CapSense CSD P4 (v2.40)	CapSense P4 (v4.X)	
Sensitivity	***Finger capacitance***	**Signal Representation**
1	0.1	50 Counts/0.1 pF
2	0.2	50 Counts/0.2 pF
3	0.3	50 Counts/0.3 pF
4	0.4	50 Counts/0.4 pF
5	0.5	50 Counts/0.5 pF
6	0.6	50 Counts/0.6 pF
7	0.7	50 Counts/0.7 pF
8	0.8	50 Counts/0.8 pF
9	0.9	50 Counts/0.9 pF
10	1.0	50 Counts/1.0 pF

e. Compensation IDAC in v2.40: If the compensation IDAC is enabled in the basic tab of the v2.40 Component, enable the same (*Enable compensation IDAC*) in *CSD Settings* sub-tab of v4.X.

f. Raw data filter in v2.40: If the Raw data noise filter is used in the v2.40 Component, enable the same (*Regular widget raw count filter type*) in *General* sub-tab of the new Component as described in *Step-2*.

g. Enable Built-in self-test in v2.40: If the self-test is enabled in the v2.40 Component, enable the same (*Enable self-test library*) in *General* sub-tab of the new Component.

h. Water proofing and detection: If water proofing and detection is enabled, enable the shield electrodes (*Enable shield electrode*) in *CSD Settings* sub-tab as described in *Step-3*, create a Button widget instead of guard sensors, and discard the status reported by widgets/sensors when the guard sensor is active in the application program.

Step-2: Parameters: Enable firmware filters

If the design based on the v2.40 Component met the following conditions, migration of the Component configuration is complete and go to the *Step-7* application programming interface section to continue:

- Used Auto SmartSense tuning mode
- Used Automatic threshold mode
- All firmware filters were disabled
- Water proofing and detection in v2.40 was disabled.
- Sensor auto-reset was disabled.

The firmware filter feature in v2.40 allows using only one filter in a design and all widget types must use common filter settings. In the CapSense P4 v4.X Component, the filter feature is enhanced by:

- Allowing coexistence of multiple filters simultaneously in a project.
- Both baseline filter and raw count filter coefficients having more configurable options.
- Both baseline filter and raw count filter for proximity and non-proximity sensors can be configured separately because the proximity filters often require different filter configuration as the proximity sensors are usually more affected by noise.

Raw count filters

The table below shows equivalent configurations between the v2.40 and v4.X Components, enable the filters and select coefficients based on the information in the following table:

CapSense CSD P4 (v2.40)	CapSense P4 (v4.X)	Comments
Raw data noise filter	***Regular widget raw count filter type***	
None	No check box selected	
Median	Select *Enable median filter (3-sample)*	
Averaging	Select *Enable average filter (4-sample)*	Note that v4.X implements averaging of 4 samples compared to 3 samples in v2.40.

CapSense CSD P4 (v2.40)	CapSense P4 (v4.X)	Comments
Raw data noise filter	***Regular widget raw count filter type***	
First order IIR 1/2	Select *Enable IIR filter (First order)* and set *IIR filter raw count coefficient* to 128	Note that v4.X implements more flexibility to set a filter coefficient at 1 from 1 to 128 in steps. This table shows coefficients equivalent to the configuration in previous Component.
First order IIR 1/4 (default)	Select *Enable IIR filter (First order)* and set *IIR filter raw count coefficient* to 64	
First order IIR 1/8	Select *Enable IIR filter (First order)* and set *IIR filter raw count coefficient* to 32	
First order IIR 1/16	Select *Enable IIR filter (First order)* and set *IIR filter raw count coefficient* to 16	
Jitter	Not supported.	The Jitter filter is supported only for the centroid positions with sliders and a touchpad as these filters don't provide any significant benefit to the sensor raw counter filtering.

Make the same for *Proximity widget raw count filter type*.

Baseline filters:

Set the filter coefficients as listed below for equivalent performance of v2.40 from CapSense v4.X Component:

- *Regular widget baseline* coefficient = 1
- *Proximity widget baseline coefficient* = 1

Enable sensor auto-reset

- Select *Enable sensor auto-reset* if it was enabled in the previous Component.

Step-3: Parameters: CSD Settings

The *CSD Settings* sub-tab contains the parameter common to all the CSD widgets available in the **Advanced** tab of the v2.40 Component. Follow the guidelines in the table below to set the parameters in the table below.

Note The parameters requiring selection of a "frequency" are dependent on the HFCLK settings in the Clock Editor. If HFCLK is changed in the Clock Editor, these parameters may need to be set again.

Parameter in v2.40	Parameter in v4.X (*CSD Settings* sub-tab)	Comment
Current source	NA	No change in the functionality between the Component versions.

PRELIMINARY

Parameter in v2.40	Parameter in v4.X (*CSD Settings* sub-tab)	Comment
IDAC range	NA	Not configurable and set to 4x in v4.X as all designs can work with a 4x range.
Analog Switch drive source	*Sense clock source*	The same selection options are available in both versions of the Component, use the same configuration from the previous Component.
Individual frequency settings	*Enable common Sense clock*	If an individual frequency setting was enabled in v2.40, unselect the Enable common clock in the v4.X Component.
Sense clock divider	*Sense clock frequency*	v4.X sets a clock in terms of a "frequency" instead of a divider value. If *Enable common Sense clock* is selected, set *Sense clock frequency* in the *CSD Settings* tab, if not then set *Sense clock frequency* under *Widget Details* for each widget.
Modulator clock divider	*Modulator clock frequency*	V4.X sets a clock in terms of a "frequency" instead of a divider value. Set the same modulator clock in the new Component.
Sensor auto-reset	*Enable sensor auto-reset* (In *General* tab)	No change in the functionality between the Component versions.
Widget resolution	NA	The Widget resolution is not configurable and set to 16-bits in v4.X of the Component. If the previous design used the 16 bits, no change is required. If the previous design used the 8 bits, it is automatically moved to the 16 bits in v4.X.
Negative Noise threshold	*Negative noise threshold* (In *Widget Details* tab)	No change in the functionality between the Component versions. However, v4.X sets these parameters separately for each widget.
Low baseline reset	*Low baseline reset* (In *Widget Details* tab)	
Shield	*Enable shield electrode*	No change in the functionality between the Component versions. These parameters are available in the *CSD Settings* tab only when *Enable shield electrode* is selected.
Shield signal delay	*Shield electrode delay*	
Shield tank capacitor	*Enable shield tank (Csh) capacitor*	
Pre-charge settings (shield tank capacitor)	*Csh initialization source*	
Inactive sensor connection	*Inactive sensor connection*	No change in the functionality between the Component versions.
Compensation IDAC (General tab)	*Enable compensation IDAC*	
Auto-calibration (General tab)	*Enable IDAC auto-calibration*	

Parameter in v2.40	Parameter in v4.X (*CSD Settings* sub-tab)	Comment
Guard sensor	NA	As mentioned in *Step-1*, use the button sensor instead of the guard sensor.
NA	*Number of shield electrodes*	The number of dedicated shield electrodes was fixed to 1 in v2.40. But, v4.X allows using more than one dedicated shield electrode.

Step-4: Parameters: Widgets Details

Parameter in v2.40	Parameter in v4.X (*Widget Details* sub-tab)	Comment
Diplexing	Diplexing	No change in the feature.
API resolution	Maximum position	The parameter renamed, no change in the feature.
Row API resolution	Maximum X-axis position	
Column API resolution	Maximum Y-axis position	
Position Filter	*Position filter*	No disables of the filters in both version of the Component. The *Median Filter (3 sample)* and *Jitter Filter* functionality has not changed, but the name is updated. For *Average Filter (2 sample)*, v2.40 implemented 3 sample average filter and v4.X implements 2 sample average filter. v4.X supports *IIR Filter* (½), instead of IIR filter ½ and ¼ in v2.40.
Sense clock divider (Scan order tab)	*Sense clock frequency*	The v4.X Component allows setting the Sense clock frequency separately for each widget. If a dedicated Sense clock frequency is required for each sensor, create multiple widgets with one sensor each. In addition, *Matrix Buttons* and *Touchpad* widgets set separate sense clocks for rows and columns.
	Row sense clock frequency	
	Column sense clock frequency	
Scan Resolution	*Scan resolution*	The parameter behavior is the same between v2.40 and v4.X except the following changes.
Row Scan Resolution		

Parameter in v2.40	Parameter in v4.X (*Widget Details* sub-tab)	Comment
Column Scan Resolution		The v2.40 Component provides one common Scan resolution for all the sensors in a widget. There is no separate scan Resolution for each sensor in a button widget or no separate scan Resolution for rows and columns of the matrix buttons and touchpads. As the best practice, sensors with similar electrical properties should be grouped as a widget, so that no dedicated scan resolution should be required for each sensor, row or column. If a dedicated scan Resolution is required for each sensor, create multiple widgets with one sensor each. Similarly, create two widgets for column and row sensors, but, this is not the recommended design.
Modulator IDAC (Scan order tab)	*Modulator IDAC* *Row modulator IDAC* *Column modulator IDAC*	The v4.X Component sets the modulator IDAC separately for each widget. If a dedicated modulator IDAC is required for each sensor, create multiple widgets with one sensor each. In addition, *Matrix Buttons* and *Touchpad* widgets set separate modulator IDAC for rows and columns.
Finger Threshold Row Finger Threshold Column Finger Threshold	*Finger threshold* (*Proximity threshold* and *Touch threshold* for *Proximity* widget)	The parameter behavior is the same between v2.40 and v4.X except the following changes. The v2.40 Component provides one common finger threshold for all sensors in a widget. There is no separate finger threshold for each sensor in a button widget or no separate finger threshold for rows and columns of the matrix buttons and touchpads. As the best practice, sensors with similar electrical properties should be grouped as a widget, so that no dedicated scan resolution should be required for each sensor, row or column. If a dedicated finger thresholds is required for each sensor, create multiple widgets with one sensor each. Similarly, create two widgets for column and row sensors, but, this is not the recommended design. For the *Proximity* widget, the threshold is split into two following thresholds: ▪ *Proximity threshold* to detect an approaching hand or a finger ▪ *Touch threshold* to detect a finger touch on the sensor similarly to other *Widget Type* sensors.
Noise Threshold Row Noise Threshold Column Noise Threshold	*Noise threshold*	The same rule as *Finger Threshold* applies.

PRELIMINARY

Parameter in v2.40	Parameter in v4.X (*Widget Details* sub-tab)	Comment
Negative Noise Threshold (Advanced Tab)	*Negative noise threshold*	The parameter behavior is the same between v2.40 and v4.X except the following changes. The v4.X Component setting *Negative noise threshold* separately for each widget, compared to the common value for all widgets in v2.40. Follow the design guide to set values for the negative noise threshold, or set the same value for all widgets for the backward compatibility.
Low baseline reset (Advanced Tab)	*Low baseline reset*	The same rule as *Negative Noise Threshold* applies.
Hysteresis Row Hysteresis Column Hysteresis	*Hysteresis*	The parameter behavior is the same between v2.40 and v4.X except the following changes. All widgets have a dedicated hysteresis in v4.X, and it is used along with the Finger threshold for finger detection. Follow the design guide to set values for the hysteresis, or set a value to zero for the backward compatibility for the *Linear Slider*, *Radial Slider* and *Touchpad* widgets.
Debounce	*ON debounce*	The parameter behavior is the same between v2.40 and v4.X except the following changes. All widgets have a dedicated ON denounce in v4.X, and it is used along with the Finger threshold for detection finger detection. Follow the design guide to set values for the ON debounce, or set a value to zero for the backward compatibility for the *Linear Slider*, *Radial Slider* and *Touchpad* widgets.
Compensation IDAC (Scan order tab)	*Compensation IDAC value*	The behavior is the same because the v4.X Component sets the compensation IDAC separately for each sensor.
NA	*Selected pins*	Each sensor in the *Button*, *Matrix Buttons* and *Proximity* widgets use a dedicated port pin for a sensor or reuses one or more pins from the existing sensors. By reusing the port pins from other sensors, ganged sensors, implementation of CSD and CSX sensing methods on the same port pins can be done.

Step-5: Scan Order

The **Scan Order** tab has no editable contents in v4.X, all the parameters available in the **Scan Order** tab of v2.40 are already configured in the other tabs on v4.X in the steps above.

PRELIMINARY

Step-6: Pinout

Assign the pins in the Pin Editor; this interface is not affected by Component update.

Step-7: API Comparison

The following table lists the APIs whose functionality hasn't changed in v4.X of the Component.

CapSense CSD P4 (v2.40)	CapSense P4 (v4.X)	Comments
IsBusy()	IsBusy()	No major functional changes.
Sleep()	Sleep()	
Stop	Stop()	
Wakeup()	Wakeup()	
InitializeAllBaselines()	InitializeAllBaselines()	
InitializeSensorBaseline()	InitializeSensorBaseline()	
ScanEnabledWidgets()	ScanAllWidgets()	
UpdateSensorBaseline()	UpdateSensorBaseline()	
UpdateThresholds()	ProcessSensorExt()	
UpdateWidgetBaseline()	UpdateWidgetBaseline()	

The following table shows list of API in v2.40 and its functional equivalent in v4.X Component.

CapSense CSD P4 (v2.40)	CapSense P4 (v4.X)	Comments
Start()	Start()	Start API in v4.X also initializes the sensor baselines and Tuner interfaces compared to v2.40.
CheckIsAnyWidgetActive()	IsAnyWidgetActive()	Name updated. These APIs return the same output, but note that the APIs in v4.X do not execute the touch detection algorithm every time an API is called, instead it returns the previously identified status by ProcessWidget() APIs.
CheckIsSensorActive()	IsSensorActive()	
CheckIsWidgetActive()	IsWidgetActive()	
GetCentroidPos()	GetCentroidPos()	
GetRadialCentroidPos()		
GetMatrixButtonPos()	IsMatrixButtonsActive()	
GetTouchCentroidPos()	GetXYCoordinates()	

CapSense CSD P4 (v2.40)	CapSense P4 (v4.X)	Comments
GetBaselineData() GetCompensationIDAC() GetDebounce() GetDiffCountData() GetFingerHysteresis() GetFingerThreshold() GetLowBaseline () GetModulationIDAC() GetScanResolution() GetSenseClkDivider() GetModulatorClkDivider() GetNoiseThreshold() GetSensitivityCoefficient() ReadSensorRaw()	GetParam()	The APIs in v2.40 are used to read the status and output values of the parameter of the Component. In v4.X, these parameter values (or value equivalent parameter in v4.X) can be read using common APIs by passing an appropriate register address as an argument. In addition to the parameters that can be read using these APIs, v4.X provides access to many more other parameters as well as through a register map interface. The register address is defined in the RegisterMap header file. The details of the registers and bit fields of the registers are available in RegisterMap.pdf and RegisterMap.xml files by using the *Export Register Map* feature.
SetBaselineData() SetCompensationIDAC() SetDebounce() SetDiffCountData() SetFingerHysteresis() SetFingerThreshold() SetLowBaselineReset() SetModulationIDAC() SetModulatorClkDivider() SetNegativeNoiseThreshold() SetNoiseThreshold() SetScanResolution() SetSenseClkDivider() SetSensitivity() WriteSensorRaw() SetUnscannedSensorState() EnableWidget() DisableWidget()	SetParam()	The APIs in v2.40 are used to write values to the parameter of the Component. In v4.X, these parameter values (or a value-equivalent parameter in v4.X) can be set using common APIs by passing an appropriate register address and value as arguments. In addition to the parameters that can be read using these APIs, v4.X provides access to many more other parameters as well as through a register map interface. The register address is defined in the RegisterMap header file. The details of the registers and bit fields of the registers are available in RegisterMap.pdf and RegisterMap.xml files by using the *Export Register Map* feature.
InitializeEnabledBaselines()	InitializeAllBaselines() InitializeWidgetBaseline() InitializeSensorBaseline()	The baselines are initialized in Start() API, so this API is discontinued. But, the same functionality can be achieved using one of the three APIs available in v4.X.
UpdateBaselineNoThreshold()	ProcessWidgetExt() ProcessSensorExt()	This API is discontinued, but, the same functionality can be implemented using one of the listed APIs from v4.X

PRELIMINARY

CapSense CSD P4 (v2.40)	CapSense P4 (v4.X)	Comments
UpdateEnabledBaselines()	ProcessAllWidgets() ProcessWidget() ProcessWidgetExt() ProcessSensorExt() UpdateSensorBaseline() UpdateAllBaselines() UpdateWidgetBaseline()	This API is discontinued, but, the same functionality can be implemented using one of the listed APIs from v4.X
DisableRawDataFilters() EnableRawDataFilters()	ProcessWidgetExt() ProcessSensorExt()	These v2.40 APIs are discontinued and the filter is enabled in the firmware if it is enabled in the customizer and executed part of the Process Widget APIs. If required to avoid execution of the filter, even if it is enabled in the customizer, use the one of the listed API from v4.X
ScanSensor()	CSDSetupWidgetExt() CSDScanExt()	This v2.40 API is discounted, but, the same functionality can be achieved using the two APIs in v4.X (both APIs are needed to implement the functionality). Refer to the code examples, design guides (*References*) or API description (*Application Programming Interface*) to learn how to optimize the system performance using these APIs.
ScanWidget()	CSDSetupWidget() CSDScan()	
DisableSensor()	CSDDisconnectSns()	Both APIs disconnect the sensor port pin and are set to an inactive state.
EnableSensor()	CSDConnectSns()	Both APIs connect to a sensor port pin AMUX and the sensor is ready for scan.
ClearSensors()	CSDDisconnectSns()	v2.40 API disconnects all the sensors. Call the v4.X API in a loop to disconnect all the sensors for functional equivalence.
Enable()	NA	The Component is enabled and the tuner and the Component are initialized by the Start API, so this API is discontinued without functional impact.
Init()		
TunerStart()		
SetScanSlotSettings()	CSDSetupWidget()	The v2.40 API loads the settings for scanning a sensor. v4.X API loads the common parameters for all sensors in the widget.
MeasureCmod()	GetExtCapCapacitance()	Name updated. These APIs return the same output.
MeasureCShieldTank()		
MeasureCShield()	GetShieldCapacitance()	
GetSensorCp()	GetSensorCapacitance()	

The following table shows the list of discontinued APIs and feature related these APIs are not available in the Component.

CapSense CSD P4 (v2.40)	CapSense P4 (v4.X)	Comments
DecodeAllGestures()	NA	Not supported
GetDiffDoubleCentroidPos()		
GetDoubleTouchCentroidPos()		
GetScrollCnt()		
GetWidgetNumber()		
GetNormalizedDiffCountData()		
GetNoiseEnvelope()		
ReadCurrentScanningSensor()		
GetIDACRange()		
SetIDACRange()		
SetDriveModeAllPins()		
RestoreDriveModeAllPins()		
SaveConfig()		
RestoreConfig()		