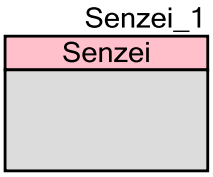
Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

PSoC 4 Multiple Phenomena Sensing (Senzei®)

7.0



Features

- Offers industry-leading CapSense® capacitive-sensing and MagSense™ inductive-sensing
- Supports CapSense self-capacitance (CSD) and mutual-capacitance (CSX) sensing methods
- Supports MagSense inductive-sensing (ISX), up to frequencies of 3 MHz
- Features SmartSense™ auto-tuning technology for CapSense capacitive-sensing that accelerates the design cycle
- Supports various sensor widgets, such as buttons, matrix buttons, sliders, touchpads, and proximity sensors
- Provides ultra-low power consumption and liquid tolerant CapSense capacitive-sensing technology
- Integrates and easy-to-use, comprehensive, graphical Tuner GUI tool for real-time tuning, testing, and debugging
- Provides superior immunity against external noise and low radiated emission.

Contains built-in self-test (BIST) library for implementing Class-B requirements for CapSense capacitive-sensing

General Description

Senzei is a hybrid sensing solution, incorporating the strengths of CapSense and MagSense. Senzei 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. It features CapSense support for various sensor widgets using both CSX and CSD sensing methods, and MagSense inductive sensing widgets, delivering robust, intelligent, and easy-to-use sensing solutions for a variety of applications in the consumer, industrial, automotive, and IoT spaces.

- The Senzei Component solution includes a configuration wizard to create and configure Senzei sensor widgets, APIs to control the Component from the application firmware, and a *Tuner* application for tuning, testing, and debugging for easy and smooth designof CapSense and MagSense applications. This datasheet includes the following

sections:*Component Configuration Parameters* – Contains descriptions of the Component's parameters in the configuration wizard.

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

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, the product-specific *CapSense design guide*, and the *Inductive Sensing 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 Senzei Component

Applications for Senzei hybrid-sensing include:

- Mechanical open/close switch replacement (metal detection)
  - Wihite goods and door open/close
  - Home security and Tamper detection
- Buttons and sliders
  - White goods buttons, keypads, and sliders
  - Industrial keypads and sliders
  - Metallic on/off buttons
  - Linear encoders
- Distance measurements
  - Proximity detection
- Rotation detection
  - Flow meters
  - Fan speed RPM detection
  - Rotary Encoders

Limitations

This Component supports the PSoC 4700S series of the PSoC4 family of devices.

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

Input / Output Connections

This section describes the various input and output connections for the Senzei Component. These are not exposed as connectable terminals on the Component symbol but these terminals can be assigned to the port pins in the Pin Editor tab of the design wide resources setting of PSoC Creator. The Pin Editor provides guidelines on the recommended pins for each terminal and does not allow an invalid pin assignment. The MagSense electrodes have a preferred port and pin. If a different pin is assigned to them in the design wide resources, then Creator will display a Note similar to “mpr.M0053:Information from the design wide resources Pin Editor has overridden the control file entry for [this pin].”

The Senzei Component requires that the first inductive sensing capable port have its maximum number of sensors before populating another port with inductive sensors.

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 and ISX sensing methods and required only if one of those methods is used. The recommended value is 470pF/5V/X7R or NP0 capacitors.
$C_{int}B$ [2]	Analog	
C_{sh} [2]	Analog	Shield tank capacitor. Used for an improved shield electrode driver when the CSD sensing is used. This capacitor is optional. The recommended value is 10nF/5V/X7R or an NP0 capacitor.
Shield	Analog	Shield electrode. Reduces the effect of the parasitic capacitance (Cp) of the sensor in the CSD sensing method. The number of shields depends on the user selection in the Component configuration wizard.
Sns	Analog	Sensors of CSD widgets. The number of sensors depends on the CSD widgets selected.
Tx	Digital Output	Transmitter electrodes of CSX widgets. The number of sensors depends on the CSX widgets selected.

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

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

Name [1]	I/O Type	Description
Lx	Digital Output	Transmitter electrodes of ISX widgets. There is one Lx electrode for each sensor. **Note** To enable the full complement of sensors, it may be necessary to change the **Debug Select** option in the Design-Wide Resources System Editor.
Rx	Analog	Receiver electrodes of CSX or ISX widgets. The number of sensors depends on the widgets selected.
Rsv	N/A	This pin is reserved for internal use. There is a reserved pin on an Inductive Sensing port as long as there is exactly one sensor on that port.

Component Configuration Parameters

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

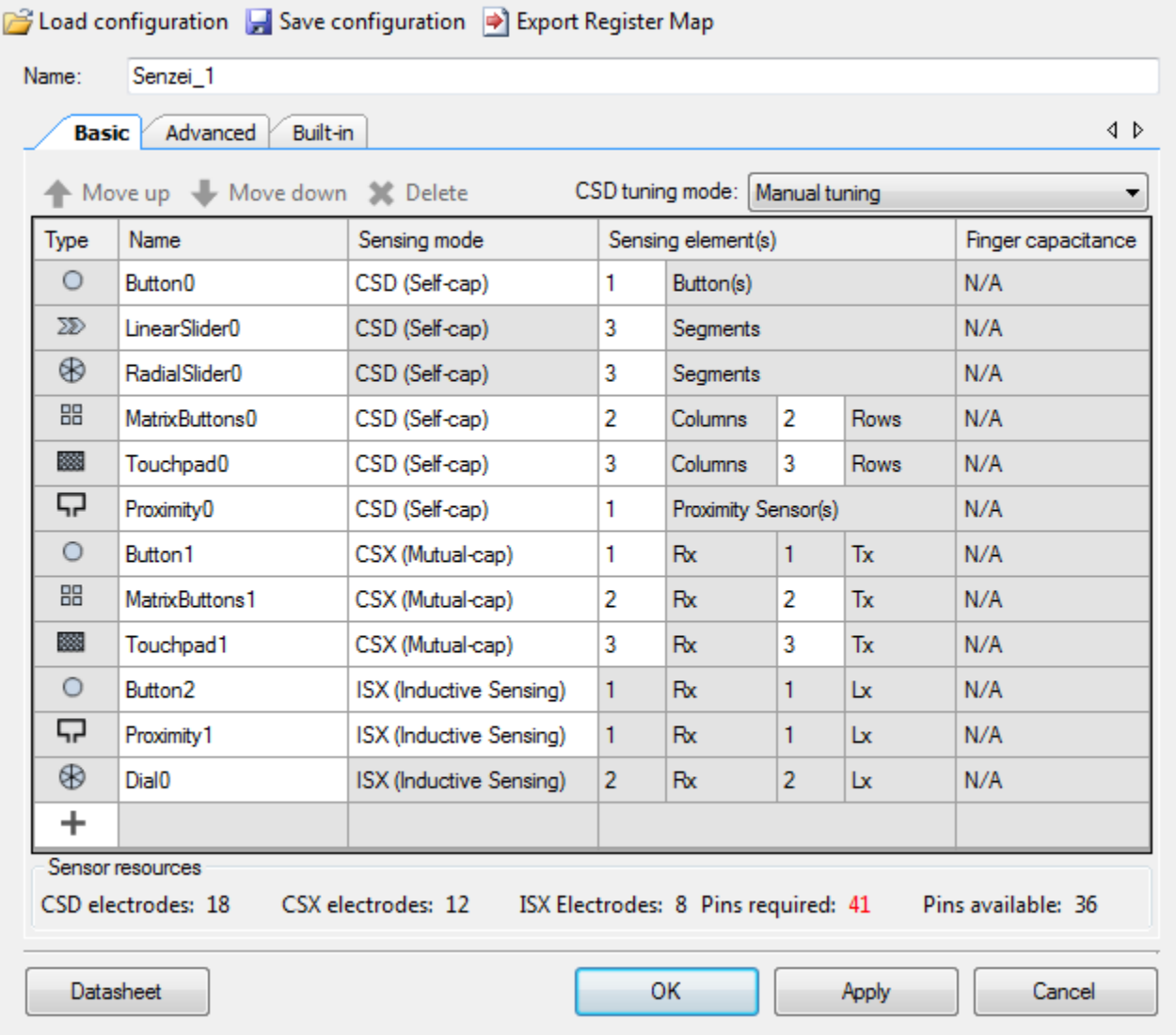
Drag a Component onto the design canvas and double-click to open the dialog.

Common Controls

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

Basic Tab

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



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

Name	Description

CSD tuning mode	Tuning is a process of finding appropriate values for configurable parameters (Hardware parameters and Threshold parameters) for proper functionality and optimized performance of the CapSense system.
	SmartSense Auto-tuning is an algorithm embedded in the Component that automatically finds the optimum values for configurable parameters, based on the hardware properties of the capacitive sensors, therefore avoids the manual tuning process by the user.
	Configurable parameters that affect the operation of the sensing hardware are called Hardware parameters. Parameters that affect the operation of the touch-detection firmware algorithm are called Threshold parameters.
	This parameter is a drop-down menu to select the tuning mode for CSD widgets only.
	▪ **SmartSense (Full Auto-Tune)** – This is the quickest way to tune a design. Most hardware and threshold parameters are automatically tuned by the Component and the GUI displays them as *Set by SmartSense* mode. In this mode, the following parameters are automatically tuned:
	  ○ *CSD Settings* tab: *Enable common sense clock*, *Enable IDAC auto-calibration, Sense clock frequency*
	  ○ *Widget Details* tab: The CSD-related parameters of the *Widget Hardware Parameters* and *Widget Threshold Parameters* groups
	  ○ *Widget Details* tab: the *Compensation IDAC value* parameter if *Enable compensation IDAC* is set.
	▪ **SmartSense (Hardware parameters only)** – The Hardware parameters are automatically set by the Component. Threshold parameters are set manually. This mode consumes less memory and less CPU processing time. This consumes lower average power. In this mode, the following parameters are automatically tuned:
	  ○ *CSD Settings* tab: *Enable common sense clock*, *Enable IDAC auto-calibration, Sense clock frequency*
	  ○ *Widget Details* tab: The CSD-related parameters of the *Widget Hardware Parameters* group
	  ○ *Widget Details* tab: *Compensation IDAC value* parameter if *Enable compensation IDAC* is set.
	▪ **Manual** –SmartSense auto-tuning is disabled. The *Widget Hardware Parameters* and *Widget Threshold Parameters* must be tuned manually. The is the lowest memory and CPU process-time consumption mode.
	SmartSense Auto-tuning (both Full Auto-Tune and Hardware parameters only) supports the *IDAC sourcing* configuration only.
	If the SmartSense (Full Auto-Tune) is enabled, then *Enable multi-frequency scan* cannot be enabled.
	Also, if SmartSense (Full Auto-Tune) is enabled, the *Enable self-test library* cannot be enabled.
	SmartSense auto-tuning requires the *Modulator clock frequency* to be set at 6000 kHz or higher.
	SmartSense operating conditions (see *Performance Characteristics*):
	▪ Sensor capacitance Cp range 5 pF to 45 pF
	▪ The typical value of the GPIO resistance is 500 Ω and the recommended external resistance is 560 Ω.

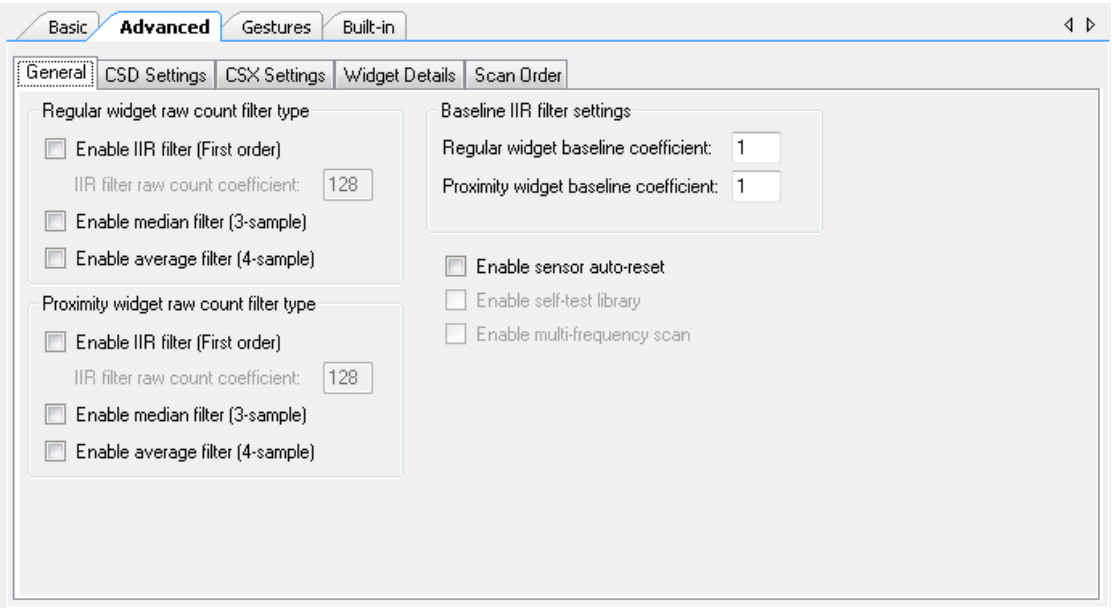
Name	Description
Widget Type	A widget is one sensor or a group of sensors that perform a specific user-interface function. The following describe the widgets types: ▪ **Button** – One or more sensors. Each sensor in the widget can detect the presence or absence (i.e., only two states) of a finger on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

Name	Description
Sensing mode	The parameter to select the sensing mode for each widget: ▪ **CSD sensing method (Capacitive Sigma Delta) –** A Cypress patented method of performing self-capacitance measurements. All widget types support CSD sensing. ▪ **CSX sensing method –** A Cypress patented method of performing mutual-capacitance measurements. Only buttons, matrix buttons, and touchpad widgets support CSX sensing. ▪ **ISX sensing method –** A Cypress patented method of performing inductive sensing measurements. Only buttons, proximity sensors, and encoder dials support ISX Sensing mode.
Widget Sensing element(s)	A sensing element refers to the Component terminals assigned to port pins to connect to physical sensors on a user-interface panel (such as a pad or layer on a PCB, ITO, or FPCB). The following element numbers are supported by the *CSD sensing method*: ▪ *Button –* Supports 1 to 32 sensors within a widget. ▪ *Linear Slider –* Supports 3 to 32 segments within a widget. ▪ *Radial Slider –* Supports 3 to 32 segments within a widget. ▪ *Matrix Buttons –* Support 2 to 16 rows and columns. The number of total intersections (sensors) is equal to that of rows x columns, limited to the maximum of 32. ▪ *Touchpad –* Supports 3 to 16 rows and columns. ▪ *Proximity –* Supports 1 to 16 sensors within a widget. The following element numbers are supported by the *CSX sensing method*: ▪ *Button* – 1 to 32 Rx electrodes (for 1 to 32 sensors) and Tx is fixed to 1. ▪ *Matrix Buttons* – 2 to 16 Tx and Rx. The total intersections (node) number is equal to Tx × Rx limited to the maximum of 32. ▪ *Touchpad –* Supports 3 to 16 Tx and Rx. The total intersections (node) number is equal to Tx × Rx. The maximum number of nodes is 256. The following element numbers are supported by the *ISX sensing method*. • *Button* – Exactly 1 Lx and 1 Rx. • *Proximity* – Exactly 1 Lx and1 Rx. • *Encoder Dial* – Exactly 2 Lx and 2 Rx.
Finger capacitance	Finger capacitance is defined as capacitance introduced by a user touch on the sensors. This parameter is used to indicate how a sensitive CSD widget is tuned by the *SmartSense Auto-tuning* algorithm. The supported Finger capacitance range: ▪ *SmartSense (Full Auto-Tune)* mode – 0.1 pF to 1 pF with a 0.02-pF step. ▪ *SmartSense (Hardware parameters only)* mode – 0.02 pF to 20.48 pF on the exponential scale. CapSense sensor sensitivity is inversely proportional to a finger capacitance value. A smaller value of finger capacitance provides higher sensitivity for a sensor. To detect a user touch on a thick overlay (4-mm plastic overlay), finger capacitance is set to a small value (e.g., 0.1pF). For a sensor with a thin overlay or no overlay, the 0.1pF finger capacitance setting makes the sensor too sensitive and may cause false touches. For robust operation, it is important to set the appropriate finger capacitance value by considering the sensor size and overlay thickness of the design. Refer to the *CapSense design guide* for more information.

Name	Description
Move up / Move down	Moves the selected widget up or down by one on the list. It defines the widget scanning order. **Note** Moving a widget may break a pin assignment, which requires repairing the assignment in the Pin Editor.
Delete	Deletes the selected widget from the list. **Note** Deleting a widget may break a pin assignment, which requires repairing the assignment in the Pin Editor.
CSD electrodes	Indicates the total number of electrodes (port pins) used by the CSD widgets, including the *Cmod*, *Csh* and *Shield* electrodes.
CSX electrodes	Indicates the total number of electrodes (port pins) used by the CSX widgets, including the *CintA* and *CintB* capacitors.
ISX electrodes	Information: Indicates the total number of electrodes (port pins) used by the ISX widgets, excluding the *CintA* and *CintB* capacitors.
Pins required	Indicates the total number of port pins required for the design. This does not include port pins used by other Components in the project or SWD pins in Debug mode. The number of **Pins required** must always be less than or equal to that of *Pins availabl* for a project to build successfully. **Pins required** includes the number of CSD, CSX, and ISX electrodes, *Cmod*, *Csh, Shield*, *CintA* ,and *CintB* electrodes.
Pins available	Indicates the total number of port pins available for the selected device.

Advanced Tab

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

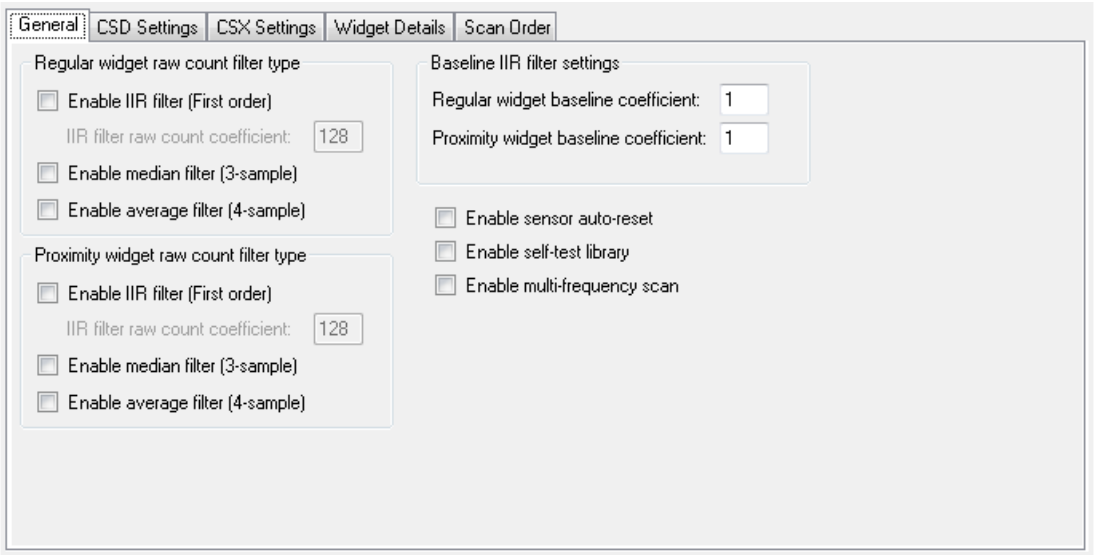


The parameters in the Advanced tab are arranged in the follwing sub-tabs.

- *General* – Contains parameters common for all widgets irrespective of the sensing method used for the widgets.
- *CSD Settings* – Contains parameters common for all widgets using the CSD sensing method. This tab is relevant only if one or more widgets use the CSD Sensing mode.
- *CSX Settings* – Contains parameters common for all widgets using the CSX sensing method. This tab is relevant only if one or more widgets use the CSX Sensing mode.
- *ISX Settings* – Contains all hardware parameters common for all ISX widgets. This tab is relevant only if one or more widgets use the ISX Sensing mode.
- *Widget Details* – Contains parameters specific to widgets and/or sensors.
- *Scan Order* – Provides information such as scan time for each sensor and total scan time for all sensors.

General Sub-Tab

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



These parameters are described in the following sections:

Regular widget raw count filter type

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

Name	Description
Enable IIR filter (First order)	Enables the infinite-impulse response filter (See equation below) with a step response similar to an RC low-pass filter, thereby passing the low-frequency signals (finger touch responses). $Output = \frac{N}{K} \times input + \frac{(K-N)}{K} \times previousOutput$ where: *K* is always 256. *N* is the IIR filter raw count coefficient selectable from 1 to 128 in the customizer. A lower N (set in the *IIR filter raw count coefficient* parameter) results in lower noise, but slows down the response. This filter eliminates high-frequency noise. Consumes 2 bytes of RAM per each sensor to store a previous raw count (filter history).

Name	Description
IIR filter raw count coefficient	The coefficient (N) of IIR filter for raw counts is explained in the *Enable IIR filter (First order)* parameter. The range of valid values: 1-128
Enable median filter (3-sample)	Enables a non-linear filter that takes three of most recent samples and computes the median value. This filter eliminates spike noise typically caused by motors and switching power supplies. Consumes 4 bytes of RAM per each sensor to store a previous raw count (filter history).
Enable average filter (4-sample)	The finite impulse response filter (no feedback) with equally weighted coefficients. It takes four of most recent samples and computes their average. Eliminates periodic noise (e.g. noise from AC mains). Consumes 6 bytes of RAM per each sensor to store a previous raw count (filter history).

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

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

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

Proximity widget raw count filter type

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

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

Baseline filter settings

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

The design baseline IIR filter is the same as the raw count *Enable IIR filter (First order)* parameter, but filter coefficients can be separate for both baseline and raw count filters to

produce a different roll-off. The baseline filter is applied to a filtered raw count (if the widget raw count filters are enabled).

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

General settings

General settings are applicable to the whole Component behavior.

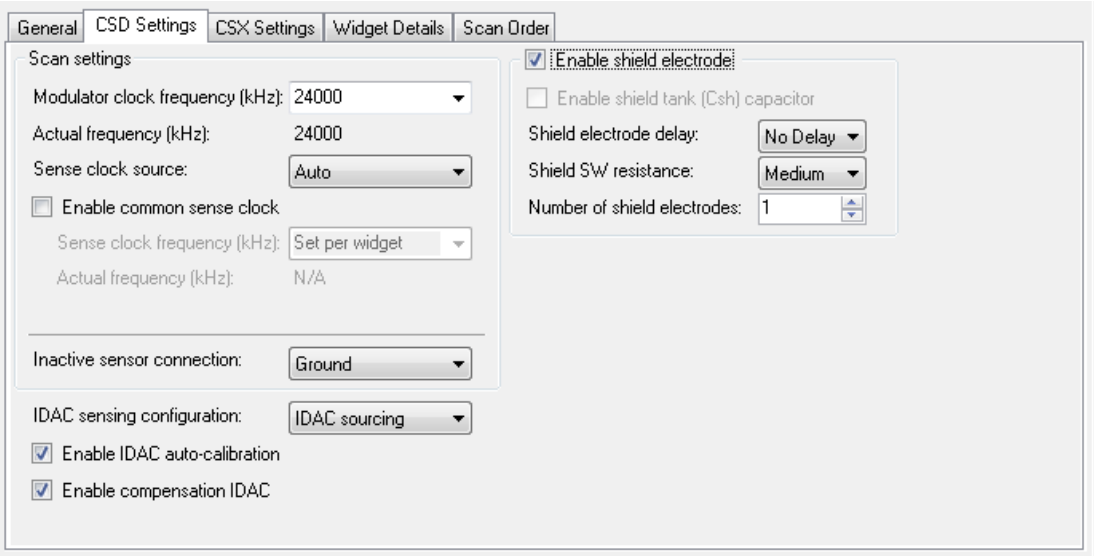
Name	Description
Enable sensor auto-reset	When enabled, the baseline is always updated. When disabled, the baseline is updated only when the difference between the baseline and raw count is less than the noise threshold. When enabled, this feature prevents sensors from permanently turning on when the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

Name	Description
Enable self-test library	For CSD and CSX Sensing modes, 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. For ISX Sensing mode, a subset of BIST functionality is provided. The library includes a set of tests for board validation, as well as Component configuration and operation. Enable the feature to get these advantages. Include the safety functions for risk-reduction, validate boards at manufacturing, and verify the Component operation in run-time. The provided tests are classified into two categories: 1. HW Tests – To confirm the CSD block and sensor hardware (external to chip) are functional: • Chip analog routing verification • Pin faults checking • PCB-trace opens / shorts checking • External capacitors and sensors capacitance measurement * • VDDA measurement * 2. FW Tests – To confirm the integrity of data used for decision making on the sensor status: • Component global and widget specific configuration verification * • Sensor baseline duplication * • Sensor raw count and baseline are in the specified range * The application layer is responsible for running each test at start and run-time as required by the product requirements. The high-level function CapSense_RunSelfTest() executes a set of tests based on an enable-mask input. This function allows running all tests or only the selected tests. The return status contains a PASS/FAIL bit for each test. Also, a set of low-level functions allows executing tests specific to a widget and a sensor. The execution time of each test is less than 10 ms at PeriClk = 12 MHz when low-level functions are used. Refer to the *Application Programming Interface* section for more details. * Functionality for these tests is provided for ISX Sensing mode **Note** Use CapSense_SetParam() to update the CapSense Data Structure parameters. Any other method invalids the CRC. **Note** If *SmartSense (Full Auto-Tune)* is enabled, the self-test library cannot be enabled. **Note** BIST and scanning cannot operate simultaneously. The status of a sensor scan must be checked using the CapSense_IsBusy() function prior to run testing.

Name	Description
Enable multi-frequency scan	For CSD and CSX Sensing modes, the multi-frequency scan (MFS) performs a triple-sensor scan with different frequencies. For ISX Sensing mode, MFS is not supported. After the triple-sensor scan, the Component 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. **Side effects:** ▪ Increased flash and RAM usage. Refer to the *Memory Usage* section for details. ▪ Increased the sensor scan duration by three times and partially processing time. ▪ The multi-frequency scan changes the IMO clock. All Components which reuse IMO for critical time-dependent operations will be affected by the CapSense Component. For example, the communication-oriented Component.

CSD Settings Sub-Tab

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



These parameters are described in the following table:

Name	Description
Modulator clock frequency	Selects the modulator clock frequency used for the *CSD sensing method*. It is the operating frequency of the CSD block. The minimum value is 1000 kHz. The maximum value is 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. This results in lower average power consumption and reduces the noise in the raw counts. Cypress recommends using the highest possible frequency. *SmartSense Auto-tuning* requires the *Modulator clock frequency* to be set at 6000 kHz or higher.

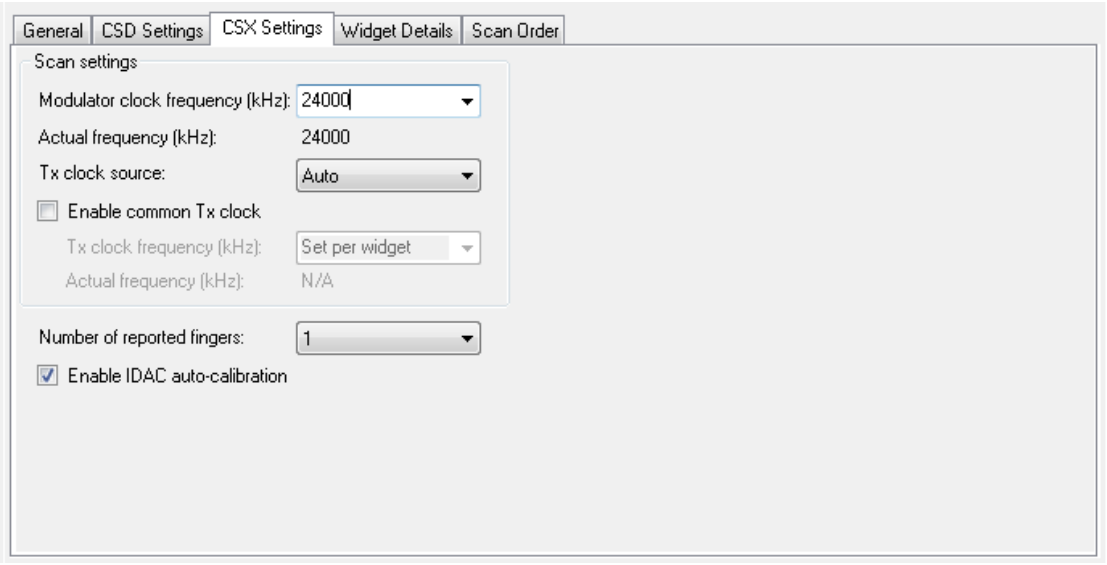
Name	Description
Sense clock source	*Sense clock frequency* is derived from the *Modulator clock frequency* using a clock-divider and is used to sample the sensor. Both the clock source and clock frequency are configurable. The Spread Spectrum Clock (SSC) provides a dithering clock source with a center frequency equal to the frequency set in the *Sense clock frequency* parameter. The PRS clock source spreads the clock using the pseudo-random sequencer and the Direct source disables both SSC and PRS sources and uses a fixed-frequency clock. Both PRS and SSC reduce the radiated noise by spreading the clock and improve the immunity against external noise. Using a higher number of bits of SSC and PRS lowers the radiation and increases the immunity against external noise. The following sources are available: ▪ *Direct* – PRS and SSC are disabled and a fixed clock is used. ▪ *PRS8* – The clock spreads using PRS to Modulator Clock / 256. ▪ *PRS12* – The clock spreads using PRS to Modulator Clock / 4096. ▪ *Auto* – The Component automatically selects optimal SSC, PRS or Direct sources individually for each widget. The Auto is the recommended sense clock source selection. In addition to the listed above options, the following sense-clock sources are available as follows: ▪ *SSC6, SSC7, SSC9* and *SSC10* – The clock spreads using a range of 6 bits to 10 bits of the sense-clock divider respectively. The following rules and recommendations for the SSC selection: ▪ The ratio between the *Modulator clock frequency* and *Sense clock frequency* must be greater than or equal to 20. ▪ 20% of the ratio between the *Modulator clock frequency* and *Sense clock frequency* should be greater or equal to the SSC frequency range = 32. It allows varying the ratio between the Modulator and Sense clock frequencies to 32 different clocks evenly spaced over +/- 10% from the center frequency. $160 \le \frac{ModClk}{SnsClk}$ Where *ModClk* is the *Modulator clock frequency* and *SnsClk* is *Sense clock frequency*. ▪ At least one full-spread spectrum polynomial should end during the scan time: $\frac{2^N - 1}{ModClk} \ge \frac{2^{SSCN} - 1}{SnsClk}$ where *N* is the *Scan resolution*, *SSCN* is the number of bits used for SSC (6, 7, 9 and 10), *ModClk* is *Modulator clock frequency* and *SnsClk* is *Sense clock frequency*. ▪ The number of sub-conversions for the widget should be an integer multiple of the SSC polynomial selected. For example, if SSC6 is selected, the number of the sub-conversion should be multiple of (2^{SSC6}-1) = 63.

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

Name	Description
Enable IDAC auto-calibration	When enabled, values of the CSD widget IDACs are automatically set by the Component. It includes IDAC code and IDAC gain. Select the Enable IDAC Auto-calibration parameter for robust operation. The SmartSense Auto-tuning parameter can be enabled only when the Enable IDAC auto-calibration is selected.
Enable compensation IDAC	The compensation IDAC is used to compensate for sensor parasitic capacitance to improve performance. Enabling the compensation IDAC is recommended unless one IDAC is required for general purpose (other than CapSense) in the project.
Enable shield electrode	The shield electrode is used to reduce the sensor parasitic capacitance, enable water-tolerant CapSense designs and enhance the detection range for the *Proximity* sensors. When the shield electrode is disabled, configurable parameters associated with the shield electrode are hidden.
Enable shield tank (Csh) capacitor	The shield tank capacitor is used to increase the drive capacity of the shield electrode driver. It should be enabled when the shield electrode capacitance is higher than 100 pF. The recommended value for a shield tank capacitor is 10nF/5V/X7R or an NP0 capacitor. The shield tank capacitor is not supported in configurations that include both CSD and CSX sensing-based widgets.
Shield SW resistance	Selects the resistance of switches used to drive the shield electrode. The four options: ▪ **Low** ▪ **Medium (default)** ▪ **High** ▪ **Low EMI**
Number of shield electrodes	Selects the number of shield electrodes required in the design. Most designs work with one dedicated shield electrode, but some designs require multiple dedicated shield electrodes to ease the PCB layout routing or to minimize the PCB area used for the shield layer. The minimum value is 0 (i.e. shield signal could be routed to sensors using the *Inactive sensor connection* parameter) and the maximum value is equal to the total number of CapSense-enabled port pins available for the selected device.

CSX Settings Sub-tab

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



These parameters are described in the following table:

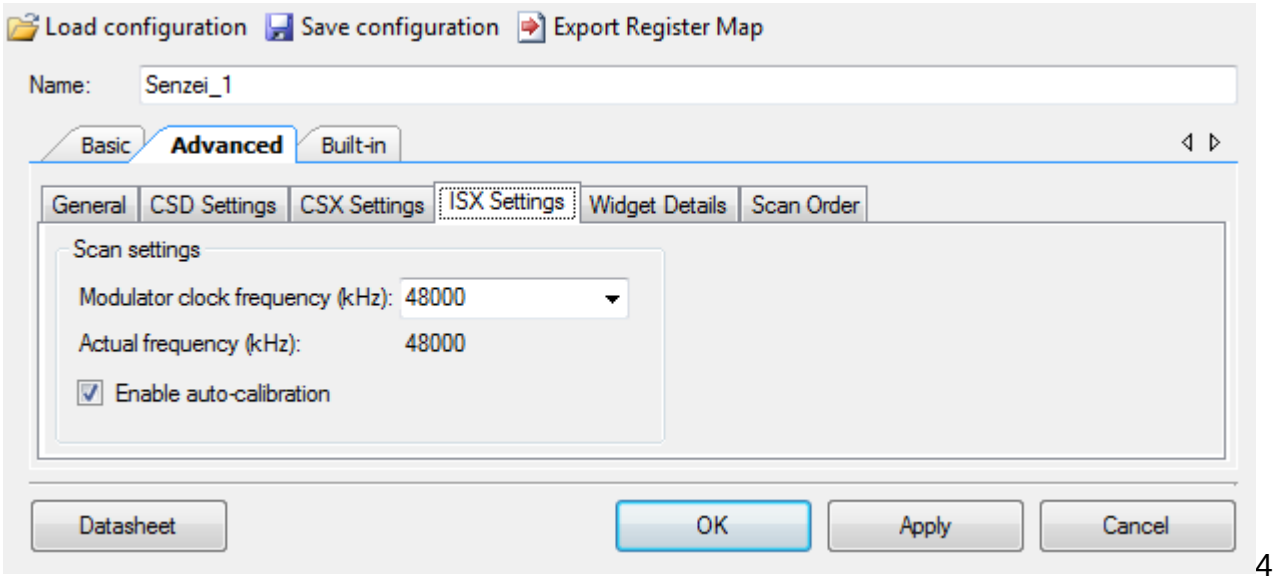
Name	Description
Modulator clock frequency	Selects the modulator clock frequency used for the *CSX sensing method*. It is the operating frequency of the CSD block. The minimum value is 1000 kHz. The maximum value is 48000 kHz or HFCLK, whichever is lower. Enter any value between the min and max limits, basing on the availability of the clock divider, the next valid lower value is selected by the Component, and the actual frequency appears in the read-only label below the drop-down list. A higher modulator clock-frequency reduces the sensor scan time, results in lower power, and reduces the noise in raw counts. Cypress recommends using the highest possible frequency.

Name	Description
Tx clock source	The *Tx clock frequency* derives from the *Modulator clock frequency* using a clock-divider and is used to sample the sensor. Both the type of the clock source and the clock frequency are configurable. The Spread Spectrum Clock (SSC) provides a dithering clock source with a center frequency equal to the frequency set in the *Tx clock frequency* parameter and the Direct source disables the SSC source and uses a fixed frequency clock. The SSC reduces the radiated noise by spreading the clock and improves the immunity against external noise. Using a higher number of bits of SSC lowers the radiation and increases the immunity against external noise. The following clock sources are available: ▪ *Direct* – SSC is disabled and a fixed clock is used. ▪ *Auto* – The Component automatically selects optimal SSC or Direct sources individually for each widget. Auto is the recommended Sense clock source selection. ▪ The rules and recommendations for the SSC selection: ▪ The ratio between the *Modulator clock frequency* and *Tx clock frequency* must be greater than or equal to 20. ▪ 20% of the ratio between the *Modulator clock frequency* and *Tx clock frequency* should be greater or equal to the SSC frequency range = 32. It allows varying the ratio between the Modulator and Tx clock frequencies to 32 different clocks evenly spaced over +/- 10% from the center frequency. $160 \le \frac{ModClk}{TxClk}$ where *ModClk* is the *Modulator clock frequency* and *TxClk* is *Tx clock frequency*. ▪ It is recommended that at least one full-spread spectrum polynomial should end during the scan time. $N_{Sub} \ge \left(2^{SSCN} - 1\right)$ where N_{Sub} is the *Number of sub-conversions*, *SSCN* is the number of bits used for SSC (6, 7, 9 and 10). ▪ It is recommended that *Number of sub-conversions* for the widget should be an integer multiple of the SSC polynomial selected. For example, if SSC6 is selected, the number of sub-conversion should be multiple of (2^{SSC6}-1) = 63.
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, the *Tx clock frequency* is entered separately for each CSX widget in the *Widget Details* tab. Using the common Tx clock for all CSX widgets results in lower power consumption and optimized memory usage and it is the recommended setting for the CSX widgets. But, in rare cases, if the electrode properties capacitance is significantly different for each widget, a common Tx clock may not produce the optimal performance.

Name	Description
Tx clock frequency	Sets the Tx clock frequency. The minimum value is 45 kHz for all device families. The maximum value is 3000 kHz. Set any value between the min and max limits, basing on the clock divider availability, the next valid lower value is selected by the Component, and the actual frequency appears in the read-only label below the drop-down list. The highest Tx clock frequency produces the maximum signal and is the recommended setting. When *Enable common Tx clock* is unselected, the Tx clock frequency is set individually for each widget in the *Widget Details* tab, and this control is grayed out. **Note** If the PeriClk frequency or *Modulator clock frequency* is changed, the Component automatically recalculates the next closest Tx clock frequency value to a possible one.
Inactive electrode connection	Selects the state of all non-scanned electrode during CSX scanning: ▪ **Ground (default)** – Inactive sensors are connected to ground. ▪ **High-Z** – Inactive sensors are floating (not connected to GND). **Ground** is the recommended selection for this parameter.
Number of reported fingers	Sets the number of reported fingers for a CSX Touchpad widget only. The available options are from 1 to 3.
Enable IDAC auto-calibration	When enabled, IDAC values are automatically set by the Component. It includes IDAC code only (IDAC gain is not included). It is recommended to select the Enable IDAC auto-calibration for robust operation.

ISX Settings Sub-tab

The parameters specific ISX sensing hardware is provided in this tab.

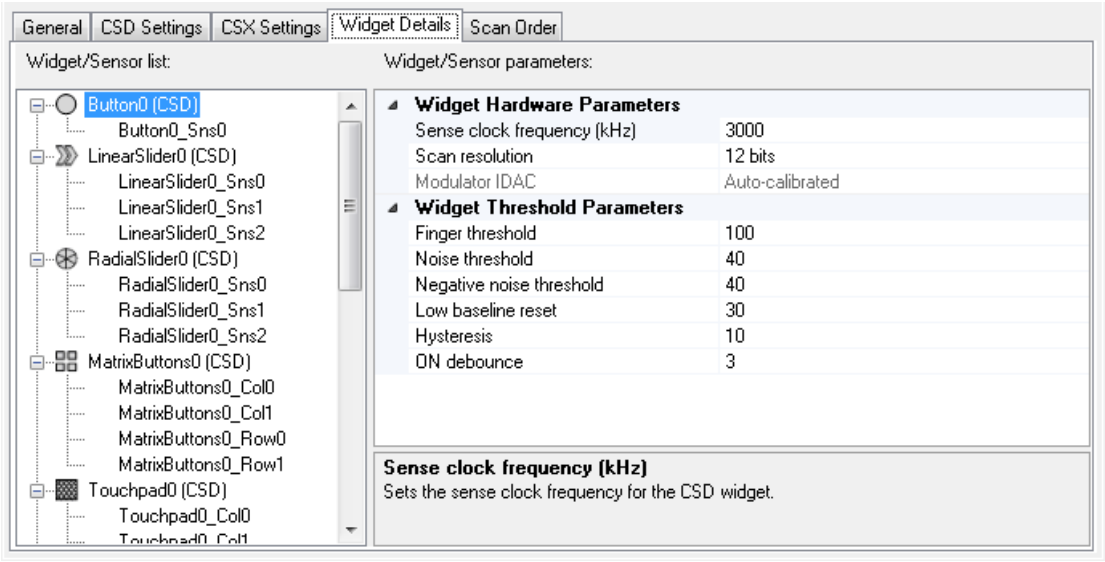


4

Name	Description
Modulator clock frequency	Selects the modulator clock frequency for the **ISX sensing method** *ISX Sensing method*. The minimum value is 1000 kHz and maximum value is 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 closest valid lower value shall be 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 average power consumption, so it is recommended to use the highest possible frequency.
Enable auto-calibration	When enabled, values of the IDACs for ISX widgets are automatically set by the Component, and finds the optimal Lx frequency. It is recommended to select the Enable auto-calibration for easy tuning experience and robust operation.

Widget Details Sub-tab

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



These parameters are described in the following table:

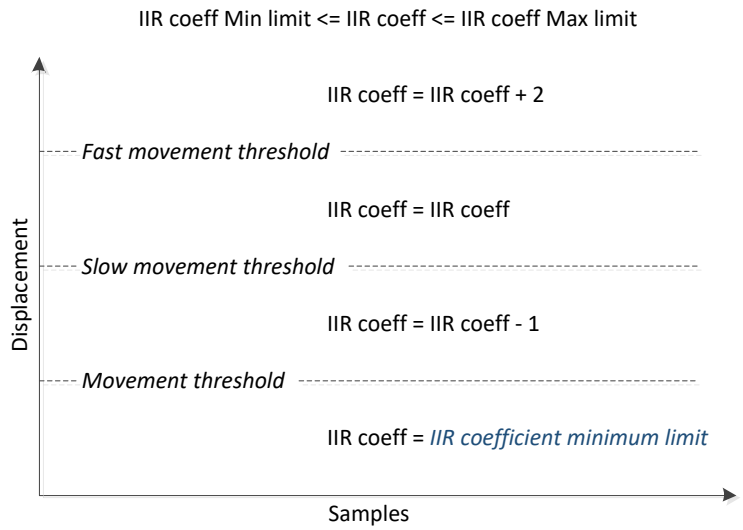
Name	Description
Widget General Parameters	
Diplexing	Enabling Diplexing allows doubling the slider physical touch sensing area by using a specific duplexing sensor pattern and without using additional port pins and sensors in the Component.
Maximum position	Represents the maximum Centroid position for the slider. A touch on the slider would produce a position value from 0 to the maximum position-value set. **No Touch** would produce 0xFFFF.

Name	Description
Maximum X-axis position	Represents the maximum column (X-axis) Centroid position and row (Y-axis) Centroid positions for a touchpad. A touch on the touchpad would produce a position value from 0 to the maximum position set. **No Touch** would produce 0xFFFF.
Maximum Y-axis position	
Widget Hardware Parameters **Note** All Widget Hardware parameters for CSD widgets are automatically set when *SmartSense (Full Auto-Tune)* is selected in the *CSD tuning mode*.	
Sense clock frequency	This parameter is identical to the *Sense clock frequency* parameter in the *CSD Settings* tab. When *Enable common sense clock* is unselected in the *CSD Settings* tab, a sense-clock frequency for each widget is set here.
Row sense clock frequency	These parameters are identical to the *Sense clock frequency* parameter, and are used to set a sense-clock frequency for row and column sensors of the *Matrix Buttons* and *Touchpad* widgets.
Column sense clock frequency	
Tx clock frequency	This parameter is identical to the *Tx clock frequency* parameter in the *CSX Settings* tab. When *Enable common Tx clock* is unselected in the *CSX Settings* tab, a Tx clock frequency for each widget is set here.
Lx clock frequency	This parameter controls the frequency at which the inductive sensing coil will be stimulated. The Lx clock frequency for each ISX widget is set here.
Scan resolution	Selects the scan resolution of CSD widgets (resolution of capacitance to digital conversion). Acceptable values are from 6 to 16 bits.
Number of sub-conversions	Selects the number of sub-conversions in the *CSX sensing method* or *ISX Sensing method*.**ISX sensing method** $$N_{Sub} < \frac{2^{16} \times XClk}{ModClk}$$ where, *ModClk* is the *Modulator clock frequency* *XClk* is the *Tx clock frequency* or *Lx clock frequency* N_{Sub} is the value of this parameter
Modulator IDAC	Sets the modulator IDAC value for the CSD Button, Slider, or Proximity widget. Or any ISX Widget. The value of this parameter is automatically set when *Enable IDAC auto-calibration* is selected in the *CSD Settings* tab or *ISX Settings* tab.
Row modulator IDAC	Sets a separate modulator IDAC value for the row and column sensors of the CSD *Matrix Buttons* and *Touchpad widget*. These parameters values are automatically set when *Enable IDAC auto-calibration* is checked in the *CSD Settings* tab.
Column modulator IDAC	

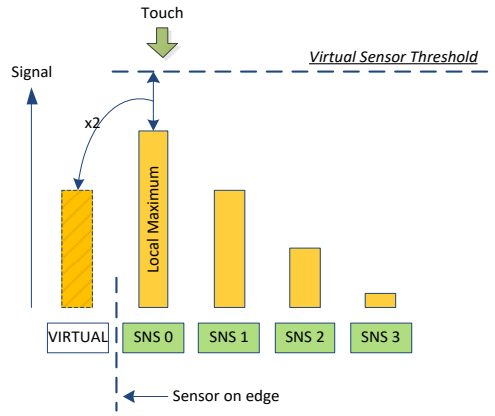
Name	Description
Gain IDAC	Sets the IDAC gain. The default value corresponds to 2.4uA for CSD sensors and 0.3uA for CSX sensors per one IDAC code set in *Modulator IDAC* / *Compensation IDAC value* and *IDAC Values*. The value of this parameter is automatically set for CSD widgets when the CSD IDAC auto-calibration is enabled.
Widget Threshold Parameters **Note** All the threshold parameters for the CSD widgets are automatically set when *SmartSense (Full Auto-Tune)* is selected in the *CSD tuning mode* parameter.	
Finger threshold	The finger threshold parameter is used along with the hysteresis parameter to determine the sensor state as follows: ▪ ON – Signal > (Finger Threshold + Hysteresis) ▪ OFF – Signal ≤ (Finger Threshold – Hysteresis). Note that “Signal” in the above equations refers to: Difference Count = Raw Count – Baseline. It is recommended to set the Finger threshold parameter value equal to the 80% of the touch signal. The Finger Threshold parameter is not available for the *Proximity* widget. Instead, Proximity has two thresholds: ▪ *Proximity threshold* ▪ *Touch threshold*
Noise threshold	Sets a raw count limit below which a raw count is considered as noise. When a raw count is above the Noise Threshold, a difference count is produced and the baseline is updated only if *Enable sensor auto-reset* is selected. In other words, the baseline remains constant as long as the raw count is above the baseline + noise threshold. This prevents the baseline from following raw counts during a finger touch detection event. It is recommended to set the noise threshold parameter value equal to 2x noise in the raw count or the 40% of the signal.
Negative noise threshold	Sets a raw count limit below which the baseline is not updated for the number of samples specified by the *Low baseline reset* parameter. The negative noise threshold ensures that the baseline does not fall low because of any high-amplitude repeated negative-noise spikes on a raw count caused by different noise sources such as ESD events. It is recommended to set the negative noise threshold parameter value equal to the *Noise threshold* parameter value.

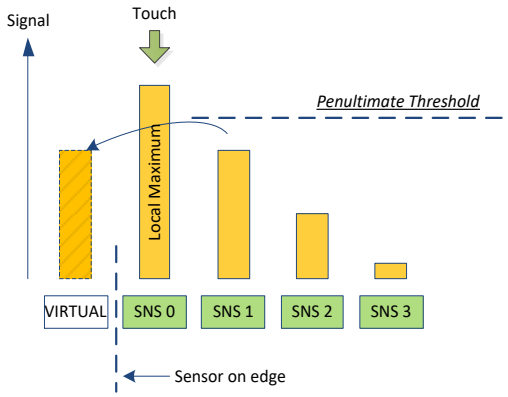
Name	Description
Low baseline reset	This parameter is used along with the *Negative noise threshold* parameter. It counts the number of abnormally low raw counts required to reset the baseline. If a finger is placed on the sensor during a device startup, the baseline gets initialized to a high raw count value at a startup. When the finger is removed, the raw count falls to a lower value. In this case, the baseline should track low raw counts. The Low Baseline Reset parameter helps handle this event. It resets the baseline to a low raw count value when the number of low samples reaches the low-baseline reset number. **Note** After a finger is removed from the sensor, the sensor will not respond to finger touches for low baseline-reset time. The recommended value is 30, which works for most designs.
Hysteresis	The hysteresis parameter is used along with the *Finger threshold* parameter (*Proximity threshold* and *Touch threshold* for Proximity sensor) to determine the sensor state. The hysteresis provides immunity against noisy transitions of the sensor state. See the description of the *Finger threshold* parameter for details. The recommend value for the hysteresis is the 10% *Finger threshold*.
ON debounce	Selects a number of consecutive CapSense scans during which a sensor must be active to generate an ON state from the Component. Debounce ensures that high-frequency, high-amplitude noise does not cause false detection ▪ Buttons/Matrix buttons/Proximity – An ON status is reported only when the sensor is touched for a consecutive debounce number of samples. ▪ Sliders/Touchpads – The position status is reported only when any of the sensors is touched for a consecutive debounce number of samples. The recommended value for the Debounce parameter is 3 for reliable sensor status detection.
Proximity threshold Touch threshold	The design of these parameters is the same as for the *Finger threshold* parameters. The proximity sensor requires a higher noise reduction, and supports two levels of detection: ▪ The proximity level to detect an approaching hand or finger ▪ The touch level to detect a finger touch on the sensor similarly to other Widget Type sensors Note that for valid operation, the Proximity threshold must be lower than the Touch threshold. The threshold parameters such as *Hysteresis* and *ON debounce* are applicable to both detection levels.
Velocity	Defines the maximum speed of a finger movement in terms of the squared distance of the touchpad resolution. The parameter is applicable for a multi-touch touchpad (CSX Touchpad) only. If the detected position of the next scan is further than the defined squared distance, then this touch is considered as a separate touch with a new touch ID.
Position Filter Parameters These parameters enable firmware filters on a centroid position to reduce noise. These filters are available for Slider and Touchpad widgets only. If multiple filters are enabled, the execution order corresponds to the listed below and the total RAM consumption increases so that the size of the total filter history is equal to a sum of all enabled filter histories.	

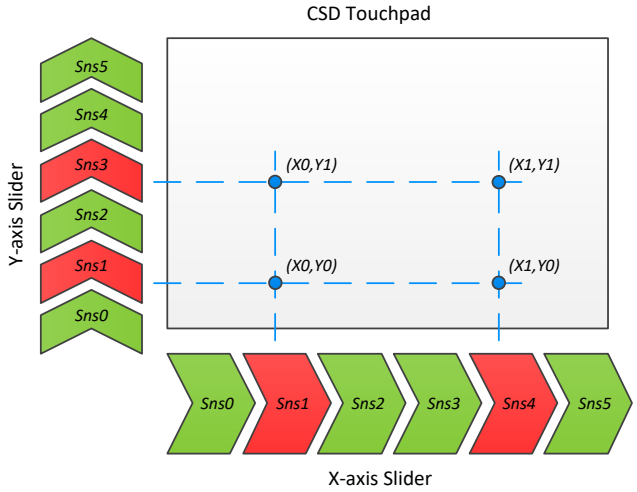
Name	Description
Median filter	Enables a non-linear filter that takes three of most recent samples and computes the median value. This filter eliminates the spikes noise typically caused by motors and switching power supplies. Consumes 4 bytes of RAM per each position (filter history).
IIR filter	Enables the infinite-impulse response filter (see equation below) with a step response. $Output = \frac{N}{K} \times Input + \frac{(K-N)}{K} \times prevOutput$ where: *K* is always 256; *N* is the IIR filter raw count coefficient selectable from 1 to 255 in the customizer. A lower N (set in the *IIR filter coefficient* parameter) results in lower noise, but slows down the response. This filter eliminates high-frequency noise. Consumes 2 bytes of RAM per each position (filter history).
IIR filter coefficient	The coefficient (N) of the IIR filter for a position as explained in the *IIR filter* parameter. The range of valid values: 1-255.
Adaptive IIR filter	Enables the Adaptive IIR filter. It is the IIR filter that changes its IIR coefficient according to the speed of the finger movement. This is done to smooth the fast movement of the finger and at the same time control properly the position movement. The filter coefficients are automatically adjusted by the adaptive algorithm with the speed of the finger movement. If the finger moves slowly, the IIR coefficient decreases; if the finger moves fast, the IIR coefficient increases from the existing value. Consumes 3 bytes of RAM per each position (filter history). When this filter is enabled, the *Adaptive IIR Filter Parameters* are available for configuration. The adaptive IIR filter is available for gesture-enabled part numbers.
Average filter	Enables the finite-impulse response filter (no feedback) with equally weighted coefficients. It takes two of most recent samples and computes their average. Eliminates periodic noise (e.g. noise from AC mains). Consumes 2 bytes of RAM per each position (filter history).
Jitter filter	This filter eliminates the noise in the position data that toggles between the two most recent values. If the most recent position value is greater than the previous one, the current position is decremented by 1; if it is less, the current position is incremented by 1. The filter is most effective at low noise. Consumes 2 bytes of RAM per each position (filter history).

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

Name	Description
Speed coefficient	Defines the value at which the position movement is interpolated when the movement is classified as slow movement. The reported position displacement is multiplied by this parameter.
Divisor value	Defines the divisor value used to create a fraction for the acceleration and speed coefficients. The interpolated position coordinates are divided by the value of this parameter.
X-axis speed threshold	Defines the threshold to distinguish fast and slow movement on the X axis. If the X-axis position displacement reported between two consecutive scans exceeds this threshold, then it is considered as fast movement, otherwise as slow movement.
Y-axis speed threshold	Defines the threshold to distinguish fast and slow movement on the Y axis. If the Y-axis position displacement reported between two consecutive scans exceeds this threshold, then it is considered as fast movement, otherwise as slow movement.
Centroid Parameters Centroid parameters are available for the CSD Touchpad widgets only.	
Centroid type	Selects a sensor matrix size for centroid calculation. The 5x5 centroid (also known as Advanced Centroid) provides benefits such as *Two finger* detection, *Edge correction*, and improved accuracy. If Advanced Centroid is selected, the below parameters are configured as well.
Cross coupling position threshold	Defines the cross coupling threshold. This value is subtracted from the sensor signal used for centroid position calculation to improve the accuracy. The threshold should be equal to a sensor signal when a finger is near the sensor but is not touching the sensor. This can be determined by slowly dragging the finger across the panel and finding the inflection point of the difference counts at the base of the curve. The difference value at this point is the Cross-coupling threshold. The default value is 5.
Edge correction	This feature is available if the *Centroid type* is configured to 5x5. When enabled, a matrix of centroid calculation is updated with virtual sensors on the edges of a touchpad. It improves the accuracy of the reported position on the edges. When enabled, two more parameters must be configured: *Virtual sensor threshold* and *Penultimate threshold*.

Name	Description
Virtual sensor threshold	This parameter is applicable only if *Edge correction* is enabled and it is used to calculate a signal (difference count) for a virtual sensor used for the edge correction algorithm. A touch position on a slider or touchpad is calculated using a signal from the local-maxima sensor and its neighboring sensors. A touch on the edge sensor of a slider or touchpad does not accurately report a position because the edge sensor lacks signal from one side of neighboring sensors of the local-maxima sensor.  If the *Edge correction* is enabled, the algorithm adds a virtual neighbor sensor to correct the deviation in the reported position. The Virtual sensor signal is defined by the Virtual sensor threshold: $DiffCount_{VIRTUAL} = (Threshold_{VIRTUAL} - DiffCount_{SNS0}) \times 2$ where: *DiffCount* $_{VIRTUAL}$ is the virtual sensor difference count; *Threshold* $_{VIRTUAL}$ is the virtual sensor threshold; *DiffCount* $_{SNS0}$ is the sensor 0 difference count. The conditions for a virtual sensor (and *Edge correction* algorithm) to be applied: ▪ Local-maxima detected on the edge sensor ▪ Difference count from the penultimate sensor less than the Penultimate threshold.

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

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

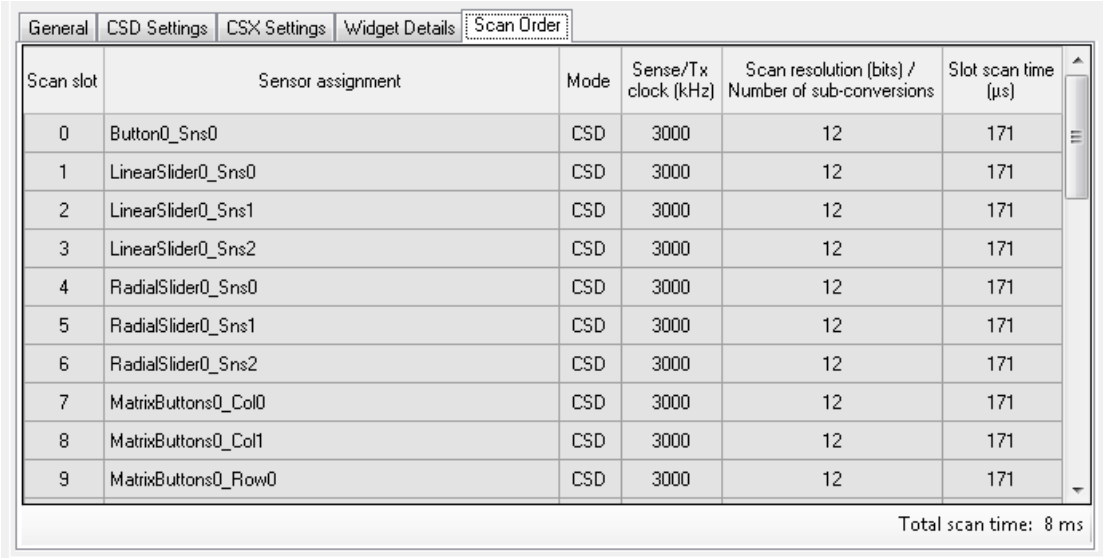
The following table shows which Widget / Sensor parameters belong to a given widget type.

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

Parameters	Widget Type											
	CSD Widget					CSX Widget				ISX Widget		
	Button	Linear Slider	Radial Slider	Matrix Buttons	Touchpad	Proximity	Button	Matrix Buttons	Touchpad	Button	Proximity	Encoder Dial
IIR filter coefficient		√	√		√				√	√	√	√
Adaptive IIR filter		√	√		√				√	√	√	√
Average filter		√	√		√				√	√	√	√
Jitter filter		√	√		√				√			
Ballistic multiplier		√	√		√				√			
Adaptive IIR Filter Parameters												
Position movement threshold		√	√		√				√			
Position slow movement threshold		√	√		√				√			
Position fast movement threshold		√	√		√				√			
IIR coefficient maximum limit		√	√		√				√			
IIR coefficient minimum limit		√	√		√				√			
IIR coefficient divisor		√	√		√				√			
Centroid Parameters												
Centroid type					√							
Cross-coupling position threshold					√							
Edge correction					√							
Virtual sensor threshold					√							
Penultimate threshold					√							
Two finger detection					√							

Scan Order Sub-Tab

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



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

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

Application Programming Interface

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

The compilers the CapSense firmware library supports:

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

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

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

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

(Replace vX_XX with the Component version)

By default, the instance name of the Component is “Senzei_1” for the first instance of a Component in a given design. It can be renamed to any unique text that follows the syntactic rules for identifiers. The instance name is prefixed to every function, variable, and constant name. For readability, this section assumes “CapSense” as the instance name.

Senzei 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. The different initialization that is required based on a the sensing method or type of widgets is automatically handled by these APIs, therefore these APIs are agnostic to sensing methods, features and widget type.

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

Functions

- cystatus Senzei_Start(void)
  Initializes the Component hardware and firmware modules. This function is called by the application program prior to calling any other function of the Component.
- cystatus Senzei_Stop(void)
  Stops the Component operation.
- cystatus Senzei_Resume(void)
  Resumes the Component operation if the Senzei_Stop() function was called previously.
- cystatus Senzei_ProcessAllWidgets(void)
  Performs full data processing of all enabled widgets.
- cystatus Senzei_ProcessWidget(uint32 widgetId)
  Performs full data processing of the specified widget if it is enabled.
- void Senzei_Sleep(void)
  Prepares the Component for deep sleep.
- void Senzei_Wakeup(void)
  Resumes the Component after deep sleep power mode.
- uint32 Senzei_DecodeWidgetGestures(uint32 widgetId)
  Decodes all enabled gestures for the specified widget and returns the gesture code.
- void Senzei_IncrementGestureTimestamp(void)
  Increases the timestamp register for the predefined timestamp interval.
- void Senzei_SetGestureTimestamp(uint32 timestampValue)
  Rewrites the timestamp register by the specified value.
- uint32 Senzei_RunSelfTest(uint32 testEnMask)
  Runs built-in self-tests specified by the test enable mask.
- cystatus Senzei_SetupWidget(uint32 widgetId)
  Performs the initialization required to scan the specified widget.
- cystatus Senzei_Scan(void)
  Initiates scanning of all the sensors in the widget initialized by Senzei_SetupWidget(), if no scan is in progress.
- cystatus Senzei_ScanAllWidgets(void)
  Initializes the first enabled widget and scanning of all the sensors in the widget, then the same process is repeated for all the widgets in the Component, i.e. scanning of all the widgets in the Component.
- uint32 Senzei_IsBusy(void)
  Returns the current status of the Component (Scan is completed or Scan is in progress).

- uint32 Senzei_IsAnyWidgetActive(void)
  Reports if any widget has detected a touch.
- uint32 Senzei_IsWidgetActive(uint32 widgetId)
  Reports if the specified widget detects a touch on any of its sensors.
- uint32 Senzei_IsSensorActive(uint32 widgetId, uint32 sensorId)
  Reports if the specified sensor in the widget detects a touch.
- uint32 Senzei_IsProximitySensorActive(uint32 widgetId, uint32 proxId)
  Reports the finger detection status of the specified proximity widget/sensor.
- uint32 Senzei_IsMatrixButtonsActive(uint32 widgetId)
  Reports the status of the specified matrix button widget.
- uint32 Senzei_GetCentroidPos(uint32 widgetId)
  Reports the centroid position for the specified slider widget.
- uint32 Senzei_GetXYCoordinates(uint32 widgetId)
  Reports the X/Y position detected for the specified touchpad widget.
- uint32 Senzei_RunTuner(void)
  Establishes synchronized communication with the Tuner application.

Function Documentation

cystatus Senzei_Start (void)

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

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

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

When the Component operation is stopped by the Senzei_Stop() function, the Senzei_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 Senzei High-Level APIs section.

cystatus Senzei_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 Senzei_Resume() function or the Component can be reset by calling the Senzei_Start() function.

This function is called when no scanning is in progress. I.e. Senzei_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 Senzei High-Level APIs section.

cystatus Senzei_Resume (void)

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

1. Reset all the Widgets/Sensors statuses.
2. Configure the hardware to perform 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 Senzei High-Level APIs section.

cystatus Senzei_ProcessAllWidgets (void)

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

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

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

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

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

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

If the ballistic multiplier filter is enabled, make sure the timestamp is updated before calling this function. Use one of the following functions to update the timestamp:

- Senzei_IncrementGestureTimestamp().
- Senzei_SetGestureTimestamp().

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 Senzei High-Level APIs section.

cystatus Senzei_ProcessWidget (uint32 *widgetId*)

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

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

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

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

If the specified widget has enabled ballistic multiplier filter, make sure the timestamp is updated before calling this function. Use one of the following functions to update the timestamp:

- Senzei_IncrementGestureTimestamp().
- Senzei_SetGestureTimestamp().

Parameters:

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

Returns:

Returns the status of the widget processing:

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

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

void Senzei_Sleep (void)

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

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

void Senzei_Wakeup (void)

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

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

uint32 Senzei_DecodeWidgetGestures (uint32 *widgetId*)

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

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

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

Parameters:

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

Returns:

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

- Senzei_NON_VALID_PARAMETER
- Senzei_NO_GESTURE
- Senzei_UNRECOGNIZED_GESTURE
- Senzei_ONE_FINGER_TOUCHDOWN
- Senzei_ONE_FINGER_LIFT_OFF
- Senzei_ONE_FINGER_SINGLE_CLICK

- Senzei_ONE_FINGER_DOUBLE_CLICK
- Senzei_ONE_FINGER_CLICK_AND_DRAG
- Senzei_ONE_FINGER_SCROLL_UP
- Senzei_ONE_FINGER_SCROLL_DOWN
- Senzei_ONE_FINGER_SCROLL_RIGHT
- Senzei_ONE_FINGER_SCROLL_LEFT
- Senzei_ONE_FINGER_SCROLL_INERTIAL_UP
- Senzei_ONE_FINGER_SCROLL_INERTIAL_DOWN
- Senzei_ONE_FINGER_SCROLL_INERTIAL_RIGHT
- Senzei_ONE_FINGER_SCROLL_INERTIAL_LEFT
- Senzei_ONE_FINGER_FLICK_UP
- Senzei_ONE_FINGER_FLICK_DOWN
- Senzei_ONE_FINGER_FLICK_RIGHT
- Senzei_ONE_FINGER_FLICK_LEFT
- Senzei_ONE_FINGER_FLICK_UP_RIGHT
- Senzei_ONE_FINGER_FLICK_DOWN_RIGHT
- Senzei_ONE_FINGER_FLICK_DOWN_LEFT
- Senzei_ONE_FINGER_FLICK_UP_LEFT
- Senzei_ONE_FINGER_EDGE_SWIPE_UP
- Senzei_ONE_FINGER_EDGE_SWIPE_DOWN
- Senzei_ONE_FINGER_EDGE_SWIPE_RIGTH
- Senzei_ONE_FINGER_EDGE_SWIPE_LEFT
- Senzei_ONE_FINGER_ROTATE_CW
- Senzei_ONE_FINGER_ROTATE_CCW
- Senzei_TWO_FINGER_SINGLE_CLICK
- Senzei_TWO_FINGER_SCROLL_UP
- Senzei_TWO_FINGER_SCROLL_DOWN
- Senzei_TWO_FINGER_SCROLL_RIGHT
- Senzei_TWO_FINGER_SCROLL_LEFT
- Senzei_TWO_FINGER_SCROLL_INERTIAL_UP
- Senzei_TWO_FINGER_SCROLL_INERTIAL_DOWN
- Senzei_TWO_FINGER_SCROLL_INERTIAL_RIGHT
- Senzei_TWO_FINGER_SCROLL_INERTIAL_LEFT
- Senzei_TWO_FINGER_ZOOM_IN
- Senzei_TWO_FINGER_ZOOM_OUT

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

void Senzei_IncrementGestureTimestamp (void)

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

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

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

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

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

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

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

void Senzei_SetGestureTimestamp (uint32 *timestampValue*)

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

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

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

Parameters:

timestampValue	Specifies the timestamp value (in ms).

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

uint32 Senzei_RunSelfTest (uint32 *testEnMask*)

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

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

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

- Senzei_CheckGlobalCRC()
- Senzei_CheckWidgetCRC()
- Senzei_CheckBaselineDuplication()
- Senzei_CheckIntegritySensorPins()
- Senzei_GetSensorCapacitance()
- Senzei_GetShieldCapacitance()
- Senzei_GetExtCapCapacitance()
- Senzei_GetVdda()

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

Parameters:

testEnMask	Specifies the tests to be executed. Each bit corresponds to one test. It is possible to launch the function with any combination of the available tests. • Senzei_TST_GLOBAL_CRC - Verifies the RAM structure CRC of global parameters. • Senzei_TST_WDGT_CRC - Verifies the RAM widget structure CRC for all the widgets. • Senzei_TST_BSLN_DUPLICATION - Verifies the baseline consistency of all the sensors (inverse copy). • Senzei_TST_SNS_SHORT - Checks all the sensors for a short to GND / VDD / other sensors. • Senzei_TST_SNS_CAP - Measures all the sensors

	capacitance. • Senzei_TST_SH_CAP - Measures the shield capacitance. • Senzei_TST_EXTERNAL_CAP - Measures the capacitance of the available external capacitors. • Senzei_TST_VDDA - Measures the Vdda voltage. • Senzei_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 - All the tests are passed.
- Senzei_TST_NOT_EXECUTED - The previously triggered scanning is not completed.
- Senzei_TST_BAD_PARAM - A non-defined test was requested in the testEnMask parameter.
- Senzei_TST_GLOBAL_CRC - Fails the test of the RAM structure CRC of global parameters.
- Senzei_TST_WDGT_CRC - Fails the test of the RAM widget structure CRC of any widget.
- Senzei_TST_BSLN_DUPLICATION - Fails the baseline consistency test of any sensor.
- Senzei_TST_SNS_SHORT - Fails the test of the short to GND or VDD of any sensor.
- Senzei_TST_SNS_CAP - Fails the execution of any sensor capacitance measurement test.
- Senzei_TST_SH_CAP - Fails the execution of the shield capacitance measurement test.
- Senzei_TST_EXTERNAL_CAP - Fails the execution of any external capacitor capacitance measurement test.
- Senzei_TST_VDDA - Fails the execution of the Vdda voltage measurement test.

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

cystatus Senzei_SetupWidget (uint32 *widgetId*)

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

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

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

Parameters:

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

Returns:

Returns the status of the widget setting up operation:

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

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

cystatus Senzei_Scan (void)

This function is called only after the Senzei_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 Senzei_IsBusy() API prior to starting a next scan or setting up another widget.

Returns:

Returns the status of the scan initiation operation:

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

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

cystatus Senzei_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 Senzei_SetupWidget() and Senzei_Scan() functions are executed by these functions. The status of a sensor scan must be checked using the Senzei_IsBusy() API prior to starting a next scan or setting up another widget.

Returns:

Returns the status of the operation:

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

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

uint32 Senzei_IsBusy (void)

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

Returns:

Returns the current status of the Component:

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

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

uint32 Senzei_IsAnyWidgetActive (void)

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

Returns:

Returns the touch detection status of all the widgets:

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

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

uint32 Senzei_IsWidgetActive (uint32 *widgetId*)

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

Parameters:

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

Returns:

Returns the touch detection status of the specified widgets:

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

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

uint32 Senzei_IsSensorActive (uint32 *widgetId*, uint32 *sensorId*)

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

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

Parameters:

widgetId	Specifies the ID number of the widget. A macro for the widget ID can be found in the Senzei Configuration header file defined as Senzei_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget to get its touch detection status. A macro for the sensor ID within the specified widget can be found in the Senzei Configuration header file defined as Senzei_<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 Senzei High-Level APIs section.

uint32 Senzei_IsProximitySensorActive (uint32 *widgetId*, uint32 *proxId*)

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

Parameters:

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

Returns:

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

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

- Bit [0] indicates that a proximity is detected.

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

uint32 Senzei_IsMatrixButtonsActive (uint32 *widgetId*)

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

Parameters:

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

Returns:

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

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

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

uint32 Senzei_GetCentroidPos (uint32 *widgetId*)

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

Parameters:

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

Returns:

Returns the centroid position of a specified slider widget:

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

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

uint32 Senzei_GetXYCoordinates (uint32 *widgetId*)

This function reports a touch position (X and Y coordinates) value of a specified touchpad widget by extracting information from the wdgtStatus registers (Senzei_<WidgetName>_POS_Y_VALUE). This function should be

used only with the touchpad widgets. This function does not process the widget but extracts processed results from the Data Structure.

Parameters:

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

Returns:

Returns the touch position of a specified touchpad widget:

1. If a touch is detected:
   - Bits [31..16] indicate the Y coordinate.
   - Bits [15..0] indicate the X coordinate.
2. If no touch is detected or a wrong widgetId is specified:
   - Senzei_TOUCHPAD_NO_TOUCH.

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

uint32 Senzei_RunTuner (void)

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

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

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

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

Returns:

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

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

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

Senzei 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 abbreviations of sensing methods in the name are specified for that sensing method appropriately and should be used only with dedicated widgets having that mode. All other functions are general to all

sensing methods, some of the APIs detect the sensing method used by the widget and executes tasks as appropriate.

Functions

- cystatus Senzei_ProcessWidgetExt(uint32 widgetId, uint32 mode)
  Performs customized data processing on the selected widget.
- cystatus Senzei_ProcessSensorExt(uint32 widgetId, uint32 sensorId, uint32 mode)
  Performs customized data processing on the selected widget's sensor.
- cystatus Senzei_UpdateAllBaselines(void)
  Updates the baseline for all the sensors in all the widgets.
- cystatus Senzei_UpdateWidgetBaseline(uint32 widgetId)
  Updates the baselines for all the sensors in a widget specified by the input parameter.
- cystatus Senzei_UpdateSensorBaseline(uint32 widgetId, uint32 sensorId)
  Updates the baseline for a sensor in a widget specified by the input parameters.
- void Senzei_InitializeAllBaselines(void)
  Initializes (or re-initializes) the baselines of all the sensors of all the widgets.
- void Senzei_InitializeWidgetBaseline(uint32 widgetId)
  Initializes (or re-initializes) the baselines of all the sensors in a widget specified by the input parameter.
- void Senzei_InitializeSensorBaseline(uint32 widgetId, uint32 sensorId)
  Initializes (or re-initializes) the baseline of a sensor in a widget specified by the input parameters.
- void Senzei_InitializeAllFilters(void)
  Initializes (or re-initializes) the raw count filter history of all the sensors of all the widgets.
- void Senzei_InitializeWidgetFilter(uint32 widgetId)
  Initializes (or re-initializes) the raw count filter history of all the sensors in a widget specified by the input parameter.
- uint32 Senzei_CheckGlobalCRC(void)
  Checks the stored CRC of the Senzei_RAM_STRUCT data structure.
- uint32 Senzei_CheckWidgetCRC(uint32 widgetId)
  Checks the stored CRC of the Senzei_RAM_WD_BASE_STRUCT data structure of the specified widget.
- uint32 Senzei_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 Senzei_CheckBaselineRawcountRange(uint32 widgetId, uint32 sensorId, Senzei_BSLN_RAW_RANGE_STRUCT*ranges)
  Checks that raw count and baseline of the specified widget/sensor are within the specified range.
- uint32 Senzei_CheckIntegritySensorPins(uint32 widgetId, uint32 sensorId)
  Checks the specified widget/sensor for shorts to GND, VDD or other sensors.
- uint32 Senzei_GetSensorCapacitance(uint32 widgetId, uint32 sensorElement, Senzei_TST_MEASUREMENT_STATUS_ENUM *measurementStatusPtr)
  Measures the specified widget/sensor capacitance.
- uint32 Senzei_GetShieldCapacitance(Senzei_TST_MEASUREMENT_STATUS_ENUM *measurementStatusPtr)
  Measures the shield electrode capacitance.
- uint32 Senzei_GetExtCapCapacitance(uint32 extCapId)
  Measures the capacitance of the specified external capacitor.
- uint16 Senzei_GetVdda(void)

Measures and returns the VDDA voltage.

- void Senzei_SetPinState(uint32 widgetId, uint32 sensorElement, uint32 state)
  Sets the state (drive mode and output state) of the port pin used by a sensor. The possible states are GND, Shield, High-Z, Tx or Rx, Sensor. If the sensor specified in the input parameter is a ganged sensor, then the state of all pins associated with the ganged sensor is updated.
- cystatus Senzei_SetupWidgetExt(uint32 widgetId, uint32 sensorId)
  Performs extended initialization for the specified widget and also performs initialization required for a specific sensor in the widget. This function requires using the Senzei_ScanExt() function to initiate a scan.
- cystatus Senzei_ScanExt(void)
  Starts a conversion on the pre-configured sensor. This function requires using the Senzei_SetupWidgetExt() function to set up the a widget.
- cystatus Senzei_CalibrateWidget(uint32 widgetId)
  Calibrates the IDACs for all the sensors in the specified widget to the default target, this function detects the sensing method used by the widget prior to calibration.
- cystatus Senzei_CalibrateAllWidgets(void)
  Calibrates the IDACs for all the widgets in the Component to the default target, this function detects the sensing method used by the widgets prior to calibration.
- uint32_t Senzei_SetInactiveElectrodeState(Senzei_OPERATION_MODE_ENUM mode, uint32_t state)
  The function updates the RAM data structure with the desired state of inactive electrodes for the specified operation mode. The state of pins is not changed in scope of this routine.
- void Senzei_CSDSetupWidget(uint32 widgetId)
  Performs hardware and firmware initialization required for scanning sensors in a specific widget using the CSD sensing method. This function requires using the Senzei_CSDScan() function to start scanning.
- void Senzei_CSDSetupWidgetExt(uint32 widgetId, uint32 sensorId)
  Performs extended initialization for the CSD widget and also performs initialization required for a specific sensor in the widget. This function requires using the Senzei_CSDScanExt() function to initiate a scan.
- void Senzei_CSDScan(void)
  This function initiates a scan for the sensors of the widget initialized by the Senzei_CSDSetupWidget() function.
- void Senzei_CSDScanExt(void)
  Starts the CSD conversion on the preconfigured sensor. This function requires using the Senzei_CSDSetupWidgetExt() function to set up the a widget.
- cystatus Senzei_CSDCalibrateWidget(uint32 widgetId, uint32 target)
  Executes the IDAC calibration for all the sensors in the widget specified in the input.
- void Senzei_CSDConnectSns (Senzei_FLASH_IO_STRUCTconst *snsAddrPtr)
  Connects a port pin used by the sensor to the AMUX bus of the sensing HW block.
- void Senzei_CSDDisconnectSns (Senzei_FLASH_IO_STRUCTconst *snsAddrPtr)
  Disconnects a sensor port pin from the sensing HW block and the AMUX bus. Sets the default state of the un-scanned sensor.
- void Senzei_CSXSetupWidget(uint32 widgetId)
  Performs hardware and firmware initialization required for scanning sensors in a specific widget using the CSX sensing method. This function requires using the Senzei_CSXScan() function to start scanning.
- void Senzei_CSXSetupWidgetExt(uint32 widgetId, uint32 sensorId)
  Performs extended initialization for the CSX widget and also performs initialization required for a specific sensor in the widget. This function requires using the Senzei_CSXScan() function to initiate a scan.
- void Senzei_CSXScan(void)
  This function initiates a scan for the sensors of the widget initialized by the Senzei_CSXSetupWidget() function.
- void Senzei_CSXScanExt(void)

Starts the CSX conversion on the preconfigured sensor. This function requires using the Senzei_CSXSetupWidgetExt() function to set up a widget.

- cystatus Senzei_CSXCalibrateWidget(uint32 widgetId, uint16 target)
  Calibrates the raw count values of all the sensors/nodes in a CSX widget.
- void Senzei_CSXConnectTx (Senzei_FLASH_IO_STRUCTconst *txPtr)
  Connects a Tx electrode to the CSX scanning hardware.
- void Senzei_CSXConnectRx (Senzei_FLASH_IO_STRUCTconst *rxPtr)
  Connects an Rx electrode to the CSX scanning hardware.
- void Senzei_CSXDisconnectTx (Senzei_FLASH_IO_STRUCTconst *txPtr)
  Disconnects a Tx electrode from the CSX scanning hardware.
- void Senzei_CSXDisconnectRx (Senzei_FLASH_IO_STRUCTconst *rxPtr)
  Disconnects an Rx electrode from the CSX scanning hardware.
- void Senzei_ISXSetupWidget(uint32 widgetId)
  Performs hardware and firmware initialization required for scanning sensors in a specific widget using the ISX sensing method. The Senzei_ISXScan() function should be used to start scanning when using this function.
- void Senzei_ISXSetupWidgetExt(uint32 widgetId, uint32 snsIndex)
  Performs extended initialization for the ISX widget and also performs initialization required for a specific sensor in the widget. The Senzei_ISXScanExt() function should be called to initiate the scan when using this function.
- void Senzei_ISXScan(void)
  This function initiates the scan for sensors of the widget initialized by the Senzei_ISXSetupWidget() function.
- void Senzei_ISXScanExt(void)
  Starts the ISX conversion on the preconfigured sensor. The Senzei_ISXSetupWidgetExt() function should be used to setup a widget when using this function.
- void Senzei_ISXCalibrateWidget(uint32 widgetId, uint16 idacTarget)
  Calibrates the raw count values of all the sensors/nodes in an ISX widget.
- void Senzei_ISXConnectLx (Senzei_FLASH_IO_STRUCTconst *lxPtr)
  Connects a LX electrode to the ISX scanning hardware.
- void Senzei_ISXConnectRx (Senzei_FLASH_IO_STRUCTconst *rxPtr)
  Connects an RX electrode to the ISX scanning hardware.
- void Senzei_ISXDisconnectLx (Senzei_FLASH_IO_STRUCTconst *lxPtr)
  Disconnects a LX electrode from the ISX scanning hardware.
- void Senzei_ISXDisconnectRx (Senzei_FLASH_IO_STRUCTconst *rxPtr)
  Disconnects an RX electrode from the ISX scanning hardware.
- cystatus Senzei_GetParam(uint32 paramId, uint32 *value)
  Gets the specified parameter value from the Data Structure.
- cystatus Senzei_SetParam(uint32 paramId, uint32 value)
  Sets a new value for the specified parameter in the Data Structure.

Function Documentation

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

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

This function can be used with any of the available scan functions. This function is called only after all the sensors in the specified widget are scanned. Calling this function multiple times with the same mode without

sensor scanning causes unexpected behavior. This function ignores the value of the wdgtEnable register. The Senzei_PROCESS_CALC_NOISE and Senzei_PROCESS_THRESHOLDS flags are supported by the CSD sensing method only when Auto-tuning mode is enabled. The pipeline scan method (i.e. during scanning of a widget, processing of a previously scanned widget is performed) can be implemented using this function and it may reduce the total scan/process time, increase the refresh rate and decrease the power consumption.

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

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

If the specified widget has enabled ballistic multiplier filter, make sure the timestamp is updated before calling this function. Use one of the following functions to update the timestamp:

- Senzei_IncrementGestureTimestamp().
- Senzei_SetGestureTimestamp().

Parameters:

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

Returns:

Returns the status of the widget processing operation:

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

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

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

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

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

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

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

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

Parameters:

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

Returns:

Returns the status of the sensor process operation:

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

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

cystatus Senzei_UpdateAllBaselines (void)

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

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

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

Returns:

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

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

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

cystatus Senzei_UpdateWidgetBaseline (uint32 *widgetId*)

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

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

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

Parameters:

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

Returns:

Returns the status of the specified widget update baseline operation:

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

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

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

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

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

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

Parameters:

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

Returns:

Returns the status of the specified sensor update baseline operation:

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

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

void Senzei_InitializeAllBaselines (void)

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

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

- Senzei_InitializeAllFilters().
- Senzei_InitializeWidgetFilter().

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

void Senzei_InitializeWidgetBaseline (uint32 *widgetId*)

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

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

- Senzei_InitializeAllFilters().
- Senzei_InitializeWidgetFilter().

Parameters:

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

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

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

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

Parameters:

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

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

void Senzei_InitializeAllFilters (void)

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

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

void Senzei_InitializeWidgetFilter (uint32 *widgetId*)

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

Parameters:

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

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

uint32 Senzei_CheckGlobalCRC (void)

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

If the stored and calculated CRC values differ, the calculated CRC is stored to the Senzei_GLB_CRC_CALC_VALUE register and the Senzei_TST_GLOBAL_CRC bit is set in the Senzei_TEST_RESULT_MASK_VALUE register. The function never clears the Senzei_TST_GLOBAL_CRC bit.

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

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

Returns:

Returns a status of the executed test:

- CYRET_SUCCESS - The stored CRC matches the calculated CRC
- Senzei_TST_GLOBAL_CRC - The stored CRC is wrong.

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

uint32 Senzei_CheckWidgetCRC (uint32 *widgetId*)

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

If the stored and calculated CRC values differ:

1. The calculated CRC is stored to the Senzei_WDGT_CRC_CALC_VALUE register
2. The widget ID is stored to the Senzei_WDGT_CRC_ID_VALUE register
3. The Senzei_TST_WDGT_CRC bit is set in the Senzei_TEST_RESULT_MASK_VALUE register.

The function never clears the Senzei_TST_WDGT_CRC bit. If the Senzei_TST_WDGT_CRC bit is set, the Senzei_WDGT_CRC_CALC_VALUE and Senzei_WDGT_CRC_ID_VALUE registers are not updated.

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

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

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

Parameters:

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

Returns:

Returns a status of the test execution:

- CYRET_SUCCESS - The stored CRC matches the calculated CRC.
- Senzei_TST_WDGT_CRC - The widget CRC is wrong.
- Senzei_TST_BAD_PARAM - The input parameter is invalid.

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

uint32 Senzei_CheckBaselineDuplication (uint32 *widgetId*, uint32 *sensorId*)

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

If the baseline does not match its inverse copy:

1. The widget ID is stored to the Senzei_INV_BSLN_WDGT_ID_VALUE register
2. The sensor ID is stored to the Senzei_INV_BSLN_SNS_ID_VALUE register
3. The Senzei_TST_BSLN_DUPLICATION bit is set in the Senzei_TEST_RESULT_MASK_VALUE register.

The function never clears the Senzei_TST_BSLN_DUPLICATION bit. If the Senzei_TST_BSLN_DUPLICATION bit is set, the Senzei_INV_BSLN_WDGT_ID_VALUE and Senzei_INV_BSLN_SNS_ID_VALUE registers are not updated.

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

The function updates the wdgtWorking register Senzei_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 Senzei Component. All Senzei processing functions like Senzei_ProcessAllWidgets() or Senzei_UpdateSensorBaseline() automatically verify the baseline value before using it and update its inverse copy after processing. If fail is detected during a baseline update a CYRET_BAD_DATA result is returned. The baseline initialization functions do not verify the baseline and update the baseline inverse copy.

Parameters:

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

Returns:

Returns the status of the test execution:

- CY_RET_SUCCESS - The baseline matches its inverse copy.
- Senzei_TST_BSLN_DUPLICATION - The test failed.
- Senzei_TST_BAD_PARAM - The input parameters are invalid.

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

uint32 Senzei_CheckBaselineRawcountRange (uint32 *widgetId*, uint32 *sensorId*, Senzei_BSLN_RAW_RANGE_STRUCT* *ranges*)

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

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

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

Parameters:

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

Returns:

Returns a status of the test execution:

- CYRET_SUCCESS - The raw count and baseline are within the specified range
- Senzei_TST_BSLN_RAW_OUT_RANGE - The test failed and baseline or raw count or both are out of the specified limit.
- Senzei_TST_BAD_PARAM - The input parameters are invalid.

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

uint32 Senzei_CheckIntegritySensorPins (uint32 *widgetId*, uint32 *sensorId*)

This function performs several tests to verify a specified sensor that is not electrically shorted and in good condition to reliably detect user interactions.

This function performs tests to check if the specified sensor is shorted to:

- GND, VDD
- Other GPIOs used by Senzei (such as sensors, Tx, Rx, shield electrodes, and external capacitors)
- Other non-Senzei GPIOs (only if they are configured in a strong high or low state during the test execution).

The absolute resistance of an electrical short must be less than 1500 Ohm including all series resistors on the sensor for the short to be detected to GND, VDD or GPIOs. For example, if a series resistor on a sensor is 560 Ohm (as recommended) and the sensor is shorted with another sensor, the function can detect a short of 380 Ohm or less as there are two 560 ohm resistors between the shorted sensor GPIOs.

The function executes the following flow to detect a short:

- Configures all the Component controlled GPIOs to strong-drive-low, and the specified sensor GPIO to resistive Pull up mode.
- A delay of Senzei_SNS_SHORT_TIME_VALUE in microseconds.
- Checks the status of the specified sensor for the expected state (logic high).
- Configures all Senzei controlled GPIOs to strong-drive-high, and the specified sensor GPIO to resistive Pull down mode.
- A delay of Senzei_SNS_SHORT_TIME_VALUE in microseconds.
- Checks the status of the specified sensor for the expected state (logic low).
- The test result is stored in Senzei Data Structure. A short is reported only when the sensor status check returns an unexpected state.

Due to the sensor parasitic capacitance and internal pull-up/down resistor, logic high-to-low (and vice versa) transitions require setting time before checking the sensor status. A 2-us delay is used as the setting time and can that be changed using the Senzei_SNS_SHORT_TIME_VALUE parameter.

This function updates the following statuses if a short is detected:

- The widget ID is stored to the Senzei_SHORTED_WDGT_ID_VALUE register.
- The sensor ID is stored to the Senzei_SHORTED_SNS_ID_VALUE register.
- The Senzei_TST_SNS_SHORT bit is set in the Senzei_TEST_RESULT_MASK_VALUE register.
- If Senzei_TST_SNS_SHORT is already set due to a previously detected fault on any of the sensor, this function does not update the Senzei_SHORTED_WDGT_ID_VALUE and Senzei_SHORTED_SNS_ID_VALUE registers. For this reason, clear Senzei_TST_SNS_SHORT prior to calling this function.
- The widget is disabled by clearing the correcting bit (Senzei_WDGT_WORKING<Number>_VALUE) in the wdgtWorking register. The widget is ignored by a high-level function during further scans and data processing tasks. The application layer can be set to the corresponding bit to restore the widget operation.
- To check all the Component sensor at once, use the Senzei_RunSelfTest() function with the Senzei_TST_SNS_SHORT mask.
- To detect an electrical short or fault condition with resistance higher than 1500 ohm, the Senzei_GetSensorCapacitance() function can be used as the fault condition changes the sensor capacitance, hence it can be detected.

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID is in the Senzei Configuration header file defined as Senzei_<WidgetName>_WDGT_ID.
sensorId	Specifies the ID number of the sensor within the widget. A macro for the sensor ID within the specified widget is in the Senzei Configuration

	header file defined as Senzei_<WidgetName>_SNS<SensorNumber>_ID.

Returns:

Returns a status of the test execution:

- CYRET_SUCCESS - The sensor does not have a short and is in the working condition.
- Senzei_TST_SNS_SHORT - A short is detected on the specified sensor.
- Senzei_TST_BAD_PARAM - The input parameters are invalid.

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

uint32 Senzei_GetSensorCapacitance (uint32 *widgetId*, uint32 *sensorElement*, Senzei_TST_MEASUREMENT_STATUS_ENUM * *measurementStatusPtr*)

This function measures the capacitance of the specified sensor element in femtofarads.

For a CSX sensor, measurement is done on either Rx or Tx electrodes. For a CSD sensor, measurement is done on a sensor (refer to the sensorElement description). If the specified sensor element is a ganged one, the capacitance is measured for all the electrodes that belong to the sensor.

For the latest measurement, the result is stored in the Senzei_RAM_SNS_CP_STRUCT data structure and the Senzei_<WidgetName>_PTR2SNS_CP_VALUE register is the pointer to the array with the measured capacitance of all sensors in the widget.

In addition to the measuring sensor capacitance, this function is used to identify various fault conditions with sensors such as electrically opened sensors. For example, the PCB track is broken or shorted to other nodes in the system - in all of these conditions, the sensor capacitance is changed which can be compared against pre determined capacitance for the sensor to detect a fault condition.

This function must not be called if the Component is in the busy state.

Capacitance is measured independently of sensor scan configuration. To measure sensor capacitance, the CSD sensing method is used and the capacitance is calculated as follows:

Cs = R * IGain * ICode / ((2^Res - 1) * Vref * SnsClk)

where (default value):

- Cs - the sensor capacitance.
- R - the measured raw count value.
- IGain - idac gain
- ICode - idac code (Compensation IDAC is disabled).
- Res - the scanning resolution.
- Vref - the reference voltage.
- SnsClk - the sensor clock frequency.

The measurement is performed differently for the 3rd and 4th generation CSD HW blocks.

- 3rd generation CSD HW block: The function performs a successive approximation search algorithm to find appropriate IDAC code for the sensor in the specified widget that provides a raw count to the target of the 85% level of the raw count maximum value. Then Cs is calculated. If the raw count is within +/- 10% of the desired target the function returns through measurementStatusPtr Senzei_TST_MEASUREMENT_SUCCESS, otherwise Senzei_TST_MEASUREMENT_LOW_LIMIT or Senzei_TST_MEASUREMENT_HIGH_LIMIT is returned. The measured Cp range for the 3rd generation CSD HW block is from 5 pF to 65 pF. The default meaurement parameter values are:
  - IGain = 1.2 uA
  - Res = 12 bits
  - Vref = 1.2 V
  - SnsClk = 1.5 MHz

- 4th generation CSD HW block: The function performs up to 4 scans to reach the raw count in a range between 7.5% and 45% of the maximum value (2^Res - 1). If a raw count is less than 7.5% of the maximum limit (2^Res - 1), the function returns Senzei_TST_LOW_LIMIT through the measurementStatus pointer. If a raw count is between 7.5% and 45% of the maximum, the function calculates the sensor capacitance, updates the register map and returns Senzei_TST_SUCCESS. If a raw count is above 45% of the maximum, the function measures again with a 4x increased current (I), and repeats the measurement until the raw count is within 7.5% to 45% of the maximum value or gets 4x bigger. The minimum measurable input by this function for the 4th generation CSD HW block is 1pF and the maximum is 384pF limited by the RC time constant (Cs < 1 / (2*5*SnsClk*R), where R is the total sensor series resistance which includes on-chip GPIO resistance ~500 Ohm and external series resistance). The default meaurement parameter values are:
  - IGain = 1.2 uA
  - Res = 12 bits
  - Vref = 1.2 V
  - SnsClk = 375 kHz
-

The measurement accuracy for both generation is about 15%.

By default, sensors that are not being measured are configured to strong-drive-low during the measurement. This state can be changed to High-Z or Shield using the Senzei_SetInactiveElectrodeState() function. CSD and CSX sensors have independent parameters to configure inactive electrode states.

A Cmod capacitor is required for the measurement. If a dedicated Cmod is not available (e.g. the design has CSX widgets only), CintA and CintB capacitors are combined by the Component to form Cmod.

The sensor measurement is done on all the sensors using the Senzei_RunSelfTest() function along with the Senzei_TST_SNS_CAP mask. The measurement operation requires reconfiguration of the hardware, hence measurement of all sensors together is recommended to avoid hardware reconfiguration for optimized firmware execution.

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID is in the Senzei Configuration header file defined as Senzei_<WidgetName>_WDGT_ID.
sensorElement	Specifies the ID of the sensor element within the widget to be measured.

For the CSD widgets, sensorElement is the sensor ID and is in the Senzei Configuration header file defined as:

- Senzei_<WidgetName>_SNS<SensorNumber>_ID.

For the CSX widgets, sensorElement is defined either as Rx ID or Tx ID. The first Rx in a widget corresponds to sensorElement = 0, the second Rx in a widget corresponds to sensorElement = 1, and so on. The last Tx in a widget corresponds to sensorElement = (RxNum + TxNum - 1). Macros for Rx and Tx IDs can be found in the Senzei Configuration header file defined as:

- Senzei_<WidgetName>_RX<RXNumber>_ID
- Senzei_<WidgetName>_TX<TXNumber>_ID.

Parameters:

measurementStatusPtr	Specifies the pointer to the Senzei_TST_MEASUREMENT_STATUS_ENUM variable where the result of the function execution is stored: • Senzei_TST_MEASUREMENT_SUCCESS - The measurement completes successfully, the result is valid. • Senzei_TST_MEASUREMENT_BAD_PARAM - The input parameter is invalid. • Senzei_TST_MEASUREMENT_LOW_LIMIT - The measured capacitance is below the minimum possible value. The

	measurement result is invalid. It is possible that the sensor was shorted to VDD or a sensor PCB track was broken (open sensor). • Senzei_TST_MEASUREMENT_HIGH_LIMIT - The measured capacitance is above the maximum possible value. The measurement result is invalid. It is possible that the sensor was shorted to GND. • Senzei_TST_MEASUREMENT_ERROR - an unexpected fault occurred during the measurement, the measurement may need to be repeated.

Returns:

Returns the capacitance of a sensor element in femtofarads. The status of the measurement is returned through the measurementStatus pointer. If measurementStatus is not equal to Senzei_TST_MEASUREMENT_SUCCESS, a fail occurs and the returned capacitance should be ignored. For more detail, refer to the measurementStatus description.

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

uint32 Senzei_GetShieldCapacitance (Senzei_TST_MEASUREMENT_STATUS_ENUM * *measurementStatusPtr*)

This function measures the capacitance of the shield electrode and returns a result. If the shield consists of several electrodes, the total capacitance of all shield electrodes is reported. The measured capacitance is stored in the Senzei_SHIELD_CAP_VALUE register of the data structure.

This function uses an algorithm identical to the sensor capacitance measurement (for more detail, refer to Senzei_GetSensorCapacitance()).

In addition to measuring the shield capacitance, this function is used to identify various fault conditions with a shield electrode such as an electrically open shield electrode, e.g. the PCB track is broken or shorted to other nodes in the system – in all of these conditions, this function returns elevated capacitance that can be compared against pre-determined capacitance for the shield electrode to detect a fault condition.

By default, all sensors are configured to Strong-drive-low mode while measuring the shield capacitance. This state can be changed to High-Z or Shield using the Senzei_SetInactiveElectrodeState() function and repeat the measurement.

This test can be executed using the Senzei_RunSelfTest() function with the Senzei_TST_SH_CAP mask.

Parameters:

measurementStatusPtr	Specifies the pointer to the Senzei_TST_MEASUREMENT_STATUS_ENUM variable where the result of the function execution is stored: • Senzei_TST_MEASUREMENT_SUCCESS - The measurement completes successfully, the result is valid. • Senzei_TST_MEASUREMENT_BAD_PARAM - The input parameter is invalid. • Senzei_TST_MEASUREMENT_LOW_LIMIT - The measured capacitance is below the minimum possible value. The measurement result is invalid. It is possible that the sensor was shorted to VDD or a sensor PCB track was broken (open sensor). • Senzei_TST_MEASUREMENT_HIGH_LIMIT - The measured capacitance is above the maximum possible value. The measurement result is invalid. It is possible that the sensor was shorted to GND. • Senzei_TST_MEASUREMENT_ERROR - an unexpected fault

	occurred during the measurement, the measurement may need to be repeated.

Returns:

Returns the capacitance of a shield in femtofarads. The status of measurement is returned through the measurementStatus pointer. If measurementStatus is not Senzei_TST_MEASUREMENT_SUCCESS, the returned value should be ignored. For more details, refer to the measurementStatus description.

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

uint32 Senzei_GetExtCapCapacitance (uint32 *extCapId*)

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

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

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

Parameters:

extCapId	Specifies the ID number of the external capacitor to be measured: • Senzei_TST_CMOD_ID - Cmod capacitor • Senzei_TST_CSH_ID - Csh capacitor • Senzei_TST_CINTA_ID - CintA capacitor • Senzei_TST_CINTB_ID - CintB capacitor

Returns:

Returns a status of the test execution:

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

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

uint16 Senzei_GetVdda (void)

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

Returns:

The VDDA voltage in mV.

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

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

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

This function accepts the Senzei_SHIELD and Senzei_SENSOR states as an input only if there is at least one CSD widget. Similarly, this function accepts the Senzei_TX_PIN and Senzei_RX_PIN states as an input only if there is at least one CSX widget in the project.

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

Parameters:

widgetId	Specifies the ID of the widget to change the pin state of the specified sensor. A macro for the widget ID can be found in the Senzei Configuration header file defined as Senzei_<WidgetName>_WDGT_ID.
sensorElement	Specifies the ID of the sensor element within the widget to change its pin state. For the CSD widgets, sensorElement is the sensor ID and can be found in the Senzei Configuration header file defined as • Senzei_<WidgetName>_SNS<SensorNumber>_ID. For the CSX widgets, sensorElement is defined either as Rx ID or Tx ID. The first Rx in a widget corresponds to sensorElement = 0, the second Rx in a widget corresponds to sensorElement = 1, and so on. The last Tx in a widget corresponds to sensorElement = (RxNum + TxNum). Macros for Rx and Tx IDs can be found in the Senzei Configuration header file defined as: • Senzei_<WidgetName>_RX<RXNumber>_ID • Senzei_<WidgetName>_TX<TXNumber>_ID.
state	Specifies the state of the sensor to be set: 1. Senzei_GROUND - The pin is connected to the ground. 2. Senzei_HIGHZ - The drive mode of the pin is set to High-Z Analog. 3. Senzei_SHIELD - The shield signal is routed to the pin (available only if CSD sensing method with shield electrode is enabled). 4. Senzei_SENSOR - The pin is connected to the scanning bus (available only if CSD sensing method is enabled). 5. Senzei_TX_PIN - The Tx or Lx signal is routed to the sensor (available only if CSX or ISX sensing method is enabled). 6. Senzei_RX_PIN - The pin is connected to the scanning bus (available only if CSX or ISX sensing method is enabled).

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

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

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

1. Connects the first sensor of the widget.
2. Configures the CSD HW block to perform a scan of the specified sensor.

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

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

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

Parameters:

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

	sensor in a specific widget. A macro for the sensor ID within a specified widget can be found in the Senzei Configuration header file defined as Senzei_<WidgetName>_SNS<SensorNumber>_ID

Returns:

Returns the status of the operation:

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

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

cystatus Senzei_ScanExt (void)

This function performs single scanning of one sensor in the widget configured by the Senzei_SetupWidgetExt() function.

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

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

If disconnection of the sensors is required after calling Senzei_ScanExt(), the Senzei_CSDDisconnectSns() or Senzei_CSXDisconnectTx() or Senzei_CSXDisconnectRx() functions can be used.

Returns:

Returns the status of the scan initiation operation:

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

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

cystatus Senzei_CalibrateWidget (uint32 *widgetId*)

This function performs exactly the same tasks as Senzei_CalibrateAllWidgets, but only for a specified widget. This function detects the sensing method used by the widgets and uses the Enable compensation IDAC parameter. For ISX mode, the frequency is also calibrated.

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

Parameters:

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

Returns:

Returns the status of the specified widget calibration:

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

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

cystatus Senzei_CalibrateAllWidgets (void)

Calibrates the IDACs for all the widgets in the Component to the default target value. This function detects the sensing method used by the widgets and regards the Enable compensation IDAC parameter. For ISX mode, the frequency is also calibrated.

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

This function is available when the ISX Enable Auto-calibration parameter is enabled.

Returns:

Returns the status of the calibration process:

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

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

uint32_t Senzei_SetInactiveElectrodeState (Senzei_OPERATION_MODE_ENUM *mode*, uint32_t *state*)

The function updates the following registers of RAM data structure:

- Senzei_SCAN_CSD_ISC_VALUE - Connection of inactive CSD and CSX electrodes during the regular CSD scan. By default, this register is initialized with the value of Inactive Sensor Connection combobox on the CSD Settings tab. The Senzei_SCAN_CSD_E value should be used as the Mode parameter to update this register.
- Senzei_SCAN_CSX_ISC_VALUE - Connection of inactive CSD, CSX and the dedicated Shield electrodes during the regular CSX scan. By default, this register is initialized with the value of Inactive Sensor Connection combobox on the CSX Settings tab. The Senzei_SCAN_CSX_E value should be used as the Mode parameter to update this register.
- Senzei_BIST_CSD_SNS_CAP_ISC_VALUE - Connection of inactive CSD and CSX electrodes during measurement of CSD electrodes capacitance. This register is initialized with the Senzei_SNS_CONNECTION_GROUND value by default. The Senzei_BIST_CSD_SNS_CAP_E value should be used as the Mode parameter to update this register.
- Senzei_BIST_CSX_SNS_CAP_ISC_VALUE - Connection of inactive CSD, CSX and the dedicated Shield electrodes during measurement of CSX electrodes (Tx and Rx) capacitance. This register is initialized with the Senzei_SNS_CONNECTION_GROUND value by default. The Senzei_BIST_CSX_SNS_CAP_E value should be used as the Mode parameter to update this register.
- Senzei_BIST_CSD_SH_CAP_ISC_VALUE - Connection of inactive CSD and CSX electrodes measurement of dedicated Shield electrodes capacitance. This register is initialized with the Senzei_SNS_CONNECTION_GROUND value by default. The Senzei_BIST_CSD_SH_CAP_E value should be used as the Mode parameter to update this register.

Parameters:

mode	Operation mode, the state of inactive sensors should be configured for. This parameter can take the following values: • Senzei_SCAN_CSD_E - Regular CSD scan. • Senzei_SCAN_CSX_E - Regular CSX scan. • Senzei_BIST_CSD_SNS_CAP_E - Measurement of the CSD sensor capacitance. • Senzei_BIST_CSX_SNS_CAP_E - Measurement of the CSX electrode capacitance. • Senzei_BIST_CSD_SH_CAP_E - Measurement of the dedicated CSD Shield electrode capacitance.
state	The desired state of inactive sensors. This parameter can take the following values: • Senzei_SNS_CONNECTION_GROUND - Inactive sensors are connected to the ground.

	• Senzei_SNS_CONNECTION_HIGHZ - Inactive sensors are floating (not connected to GND or Shield). • Senzei_SNS_CONNECTION_SHIELD - Inactive sensors are connected to the shield. This option is available only if the Enable shield electrode check box is set. At least one dedicated shield electrode is required to use the Senzei_SNS_CONNECTION_SHIELD option for the Senzei_BIST_CSD_SH_CAP_E operation mode.

Returns:

Returns the status of the operation:

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

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

void Senzei_CSDSetupWidget (uint32 *widgetId*)

Note:

This function is obsolete and kept for backward compatibility only. The Senzei_SetupWidget() function should be used instead.

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

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

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

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

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

Parameters:

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

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

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

Performs extended initialization for the CSD widget and also performs initialization required for a specific sensor in the widget. This function requires using the Senzei_CSDScanExt() function to initiate a scan.

Note:

This function is obsolete and kept for backward compatibility only. The Senzei_SetupWidgetExt() function should be used instead.

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

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

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

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

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

Parameters:

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

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

void Senzei_CSDScan (void)

This function initiates a scan for the sensors of the widget initialized by the Senzei_CSDSetupWidget() function.

Note:

This function is obsolete and kept for backward compatibility only. The Senzei_Scan() function should be used instead.

This function performs scanning of all the sensors in the widget configured by the Senzei_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 Senzei_Scan() API if the given widget uses the CSD sensing method.

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

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

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

void Senzei_CSDScanExt (void)

Starts the CSD conversion on the preconfigured sensor. This function requires using the Senzei_CSDSetupWidgetExt() function to set up the a widget.

Note:

This function is obsolete and kept for backward compatibility only. The Senzei_ScanExt() function should be used instead.

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

1. Sets the busy flag in the Senzei_dsRam structure.
2. Performs the clock-phase alignment of the sense and modulator clocks.
3. Performs the Cmod pre-charging.
4. Starts single scanning.

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

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

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

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

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

Executes the IDAC calibration for all the sensors in the widget specified in the input.

Note:

This function is obsolete and kept for backward compatibility only. The Senzei_CalibrateWidget() function should be used instead.

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 fails if the achieved raw count is outside of the range specified by the target and acceptable calibration deviation.

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

Parameters:

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

Returns:

Returns the status of the specified widget calibration:

- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_PARAM - The input parameter is invalid.
- CYRET_BAD_DATA - The calibration failed and the Component may not operate as expected.
- CYRET_TIMEOUT - The calibration failed due to timeout.
- CYRET_INVALID_STATE - The previous scanning is not completed and the hardware block is busy.

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

void Senzei_CSDConnectSns (Senzei_FLASH_IO_STRUCTconst * *snsAddrPtr*)

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

Scanning should be completed before calling this API.

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

Parameters:

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

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

void Senzei_CSDDisconnectSns (Senzei_FLASH_IO_STRUCTconst * *snsAddrPtr*)

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

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

Parameters:

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

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

void Senzei_CSXSetupWidget (uint32 *widgetId*)

Performs hardware and firmware initialization required for scanning sensors in a specific widget using the CSX sensing method. This function requires using the Senzei_CSXScan() function to start scanning.

Note:

This function is obsolete and kept for backward compatibility only. The Senzei_SetupWidget() function should be used instead.

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

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

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

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

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

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

Parameters:

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

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

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

Performs extended initialization for the CSX widget and also performs initialization required for a specific sensor in the widget. This function requires using the Senzei_CSXScan() function to initiate a scan.

Note:

This function is obsolete and kept for backward compatibility only. The Senzei_SetupWidgetExt() function should be used instead.

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

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

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

Parameters:

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

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

void Senzei_CSXScan (void)

This function initiates a scan for the sensors of the widget initialized by the Senzei_CSXSetupWidget() function.

Note:

This function is obsolete and kept for backward compatibility only. The Senzei_Scan() function should be used instead.

This function performs scanning of all the sensors in the widget configured by the Senzei_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 Senzei_Scan() API if the given widget uses the CSX sensing method.

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

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

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

void Senzei_CSXScanExt (void)

Starts the CSX conversion on the preconfigured sensor. This function requires using the Senzei_CSXSetupWidgetExt() function to set up a widget.

Note:

This function is obsolete and kept for backward compatibility only. The Senzei_ScanExt() function should be used instead.

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

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

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

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

If disconnection of the sensors is required after calling Senzei_CSXScanExt(), the Senzei_CSXDisconnectTx() and Senzei_CSXDisconnectRx() APIs can be used.

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

cystatus Senzei_CSXCalibrateWidget (uint32 *widgetId*, uint16 *target*)

Calibrates the raw count values of all the sensors/nodes in a CSX widget.

Note:

This function is obsolete and kept for backward compatibility only. The Senzei_CalibrateWidget() function should be used instead.

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

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

Parameters:

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

Returns:

Returns the status of the operation:

- Zero - All the sensors in the widget are calibrated successfully.
- Non-Zero - Calibration failed for any sensor in the widget.

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

void Senzei_CSXConnectTx (Senzei_FLASH_IO_STRUCTconst * *txPtr*)

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

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

Parameters:

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

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

void Senzei_CSXConnectRx (Senzei_FLASH_IO_STRUCTconst * *rxPtr*)

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

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

Parameters:

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

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

void Senzei_CSXDisconnectTx (Senzei_FLASH_IO_STRUCTconst * *txPtr*)

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

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

Parameters:

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

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

void Senzei_CSXDisconnectRx (Senzei_FLASH_IO_STRUCTconst * *rxPtr*)

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

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

Parameters:

rxPtr	Specifies the pointer to the FLASH_IO_STRUCT object belonging to an Rx pin sensor to be disconnected from the sensing HW block.

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

void Senzei_ISXSetupWidget (uint32 *widgetId*)

This function initializes the widgets specific common parameters to perform the ISX 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 Senzei_ISXScan() function must be called after initializing the widget to start scanning.

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

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

It is recommended to not 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	Specifies the ID number of the widget to perform hardware and firmware initialization required for scanning sensors in the specific widget. A macro for the widget ID can be found in the Senzei Configuration header file defined as Senzei_<WidgetName>_WDGT_ID.

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

void Senzei_ISXSetupWidgetExt (uint32 *widgetId*, uint32 *snsIndex*)

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

This function should be called when no scanning in progress. I.e. Senzei_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	Specifies the ID number of the widget to perform hardware and firmware initialization required for scanning a specific sensor in a specific widget. A macro for the widget ID can be found in the Senzei Configuration header file defined as Senzei_<WidgetName>_WDGT_ID.
snsIndex	Specifies the ID number of the sensor within the widget to perform hardware and firmware initialization required for scanning a specific sensor in a specific widget. A macro for the sensor ID within a specified widget can be found in the Senzei Configuration header file defined as Senzei_<WidgetName>_SNS<SensorNumber>_ID.

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

void Senzei_ISXScan (void)

This function performs scanning of all the sensors in the widget configured by the Senzei_ISXSetupWidget() 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 Senzei_Scan() API if the given widget uses the ISX 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. Senzei_IsBusy() returns a non-busy status. The widget must be preconfigured by the Senzei_ISXSetupWidget() function if other widget was previously scanned or other type of scan functions were used.

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

void Senzei_ISXScanExt (void)

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

1. Sets a busy flag in the Senzei_dsRam structure.
2. Configures the Lx 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. Senzei_IsBusy() returns a non-busy status.

The sensor must be preconfigured by using the Senzei_ISXSetupWidgetExt() API prior to calling this function. The sensor remains ready for the next scan if a previous scan was triggered by using the Senzei_ISXScanExt() function. In this case, calling Senzei_ISXSetupWidgetExt() is not required every time before the Senzei_ISXScanExt() function. If a previous scan was triggered in any other way: Senzei_Scan(), Senzei_ScanAllWidgets() or Senzei_RunTuner() (see the Senzei_RunTuner() function description for more

details), the sensor must be preconfigured again by using the Senzei_ISXSetupWidgetExt() API prior to calling the Senzei_ISXScanExt() function.

If disconnection of the sensors is required after calling Senzei_ISXScanExt(), the Senzei_ISXDisconnectLx() and Senzei_ISXDisconnectRx() APIs can be used.

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

void Senzei_ISXCalibrateWidget (uint32 *widgetId*, uint16 *idacTarget*)

Performs a rough calibration of IDAC values, then incrementally searches a small range of frequencies around the widget's Lx frequency to find the optimal Lx frequency. Then performs a 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 ISX Enable auto-calibration parameter is enabled.

Parameters:

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

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

void Senzei_ISXConnectLx (Senzei_FLASH_IO_STRUCTconst * *lxPtr*)

This function connects a port pin (Lx electrode) to the forcing 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:

lxPtr	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a sensor which should be connected to the sensing block as Lx pin.

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

void Senzei_ISXConnectRx (Senzei_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	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a sensor which should be connected to the sensing block as Rx pin.

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

void Senzei_ISXDisconnectLx (Senzei_FLASH_IO_STRUCTconst * *lxPtr*)

This function disconnects a port pin (Lx electrode) from the forcing signal.

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:

lxPtr	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a

	Lx pin sensor which should be disconnected from the sensing block.

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

void Senzei_ISXDisconnectRx (Senzei_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	Specifies the pointer to the FLASH_IO_STRUCT object belonging to a Rx pin sensor which should be disconnected from the sensing block.

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

cystatus Senzei_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 Senzei RegisterMap header file as Senzei_<ParameterName>_PARAM_ID. The paramId is a special enumerated value generated by the customizer. The format of paramId is as follows:

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

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

Parameters:

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

Returns:

Returns the status of the operation:

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

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

cystatus Senzei_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 Senzei RegisterMap header file as Senzei_<ParameterName>_PARAM_ID. The paramId is a special enumerated value generated by the customizer. The format of paramId is as follows:

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

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

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

Parameters:

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

Returns:

Returns the status of the operation:

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

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

Interrupt Service Routine

Description

The Senzei 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 Senzei 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(Senzei_CSDPostSingleScan)
  This is an internal ISR function for the single-sensor scanning implementation.
- CY_ISR(Senzei_CSDPostMultiScan)
  This is an internal ISR function for the multiple-sensor scanning implementation.
- CY_ISR(Senzei_CSDPostMultiScanGanged)
  This is an internal ISR function for the multiple-sensor scanning implementation for ganged sensors.
- CY_ISR(Senzei_CSXScanISR)
  This is an internal ISR function to handle the CSX sensing method operation.

Function Documentation

CY_ISR (Senzei_CSDPostSingleScan)

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

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

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

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

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

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

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

Go to the top of the Interrupt Service Routine section.

CY_ISR (Senzei_CSDPostMultiScan)

This ISR handler is triggered when the user calls the Senzei_Scan() or Senzei_ScanAllWidgets() APIs.

The following tasks are performed:

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

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

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

Go to the top of the Interrupt Service Routine section.

CY_ISR (Senzei_CSDPostMultiScanGanged)

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

The following tasks are performed:

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

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

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

Go to the top of the Interrupt Service Routine section.

CY_ISR (Senzei_CSXScanISR)

This handler covers the following functionality:

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

Go to the top of the Interrupt Service Routine section.

Macro Callbacks

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

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

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

Senzei Macro Callbacks

Macro Callback Function Name	Associated Macro	Description

Macro Callback Function Name	Associated Macro	Description
Senzei_EntryCallback	Senzei_ENTRY_CALLBACK	Used at the beginning of the Senzei interrupt handler to perform additional application-specific actions
Senzei_ExitCallback	Senzei_EXIT_CALLBACK	Used at the end of the Senzei interrupt handler to perform additional application-specific actions
Senzei_StartSampleCallback(uint8 Senzei_widgetId, uint8 Senzei_sensorId)	Senzei_START_SAMPLE_CALLBACK	Used before each sensor scan triggering and deliver the current widget / sensor Id

Global Variables

Description

The section documents the Senzei component related global Variables.

The Senzei 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

- Senzei_RAM_STRUCT Senzei_dsRam

Variable Documentation

Senzei_RAM_STRUCTSenzei_dsRam

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

API Constants

Description

The section documents the Senzei component related API Constants.

Variables

- const Senzei_FLASH_STRUCT Senzei_dsFlash
- const Senzei_FLASH_IO_STRUCT Senzei_ioList[Senzei_TOTAL_ELECTRODES]
- const Senzei_SHIELD_IO_STRUCT Senzei_shieldIoList[Senzei_CSD_TOTAL_SHIELD_COUNT]

Variable Documentation

const Senzei_FLASH_STRUCT Senzei_dsFlash

Constant for the FLASH Data Structure

const Senzei_FLASH_IO_STRUCT Senzei_ioList[Senzei_TOTAL_ELECTRODES]

The array of the pointers to the electrode specific register.

const Senzei_SHIELD_IO_STRUCT Senzei_shieldIoList[Senzei_CSD_TOTAL_SHIELD_COUNT]

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

Data Structure

Description

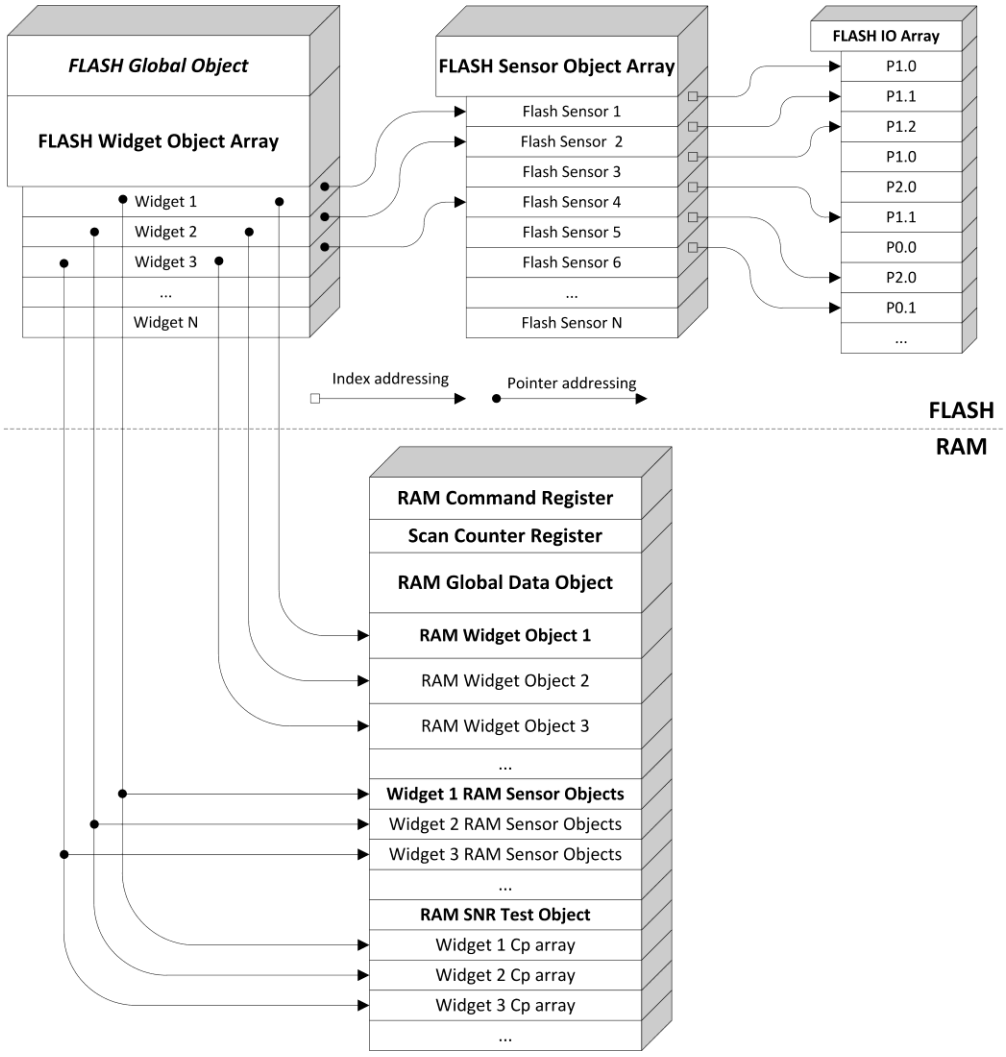
This section provides the list of structures/registers available in the component.

The key responsibilities of Data Structure are as follows:

- The Data Structure is the only data container in the component.
- It serves as storage for the configuration and the output data.
- All other component FW part as well as an application layer and Tuner SW use the data structure for the communication and data exchange.

The Senzei 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. Senzei 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 Senzei 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.

Senzei Data Structure does not perform error checking on the data written to Senzei 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 Senzei Data Structure.

The Senzei 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 Senzei_RegisterMap header file list all registers of data structure with the following:

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

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

1. Direct Access

The access to registers is performed through the Data Structure variable Senzei_dsRam and constants Senzei_dsFlash from application program.
Example of access to the Raw Count register of third sensor of Button0 widget:

```
rawCount = Senzei_dsRam.snsList.button0[Senzei_BUTTON0_SNS2_ID].raw[0];
```

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

```
rawCount = Senzei_BUTTON0_SNS2_RAW0_VALUE;
```

2. Getter/Setter Access

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

```
cystatus Senzei_GetParam(uint32 paramId, uint32 *value)
cystatus Senzei_SetParam(uint32 paramId, uint32 value)
```

The value of paramId argument for each register can be found in Senzei_RegisterMap header file.
Example of access to the Raw Count register of third sensor of Button0 widget:

```
Senzei_GetParam(Senzei_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):

```
Senzei_SetParam(Senzei_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 Senzei data structure to be exposed to I2C master at primary slave address request once at initialization an application program:

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

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

```
rawCount = *(uint16 *)(I2C_buffer1Ptr + Senzei_BUTTON0_SNS2_RAW0_OFFSET);
```

The current example is applicable to 2-byte registers only. Depends on register size defined Senzei_RegisterMap header file by corresponding macros (Senzei_BUTTON0_SNS2_RAW0_SIZE) specific logic should be added to read 4-byte, 2-byte and 1-byte registers.

Data Structures

- struct ADAPTIVE_FILTER_CONFIG_STRUCT
  Declares Adaptive Filter configuration parameters.
- struct ADVANCED_CENTROID_POSITION_STRUCT
  Declares Advanced Centroid position structure.
- struct ADVANCED_CENTROID_TOUCH_STRUCT
  Declares Advanced Centroid touch structure.
- struct SMARTSENSE_CSD_NOISE_ENVELOPE_STRUCT
  Declares Noise envelope data structure for CSD widgets when SmartSense is enabled.
- struct Senzei_RAM_WD_BASE_STRUCT
  Declares common widget RAM parameters.
- struct Senzei_RAM_WD_BUTTON_STRUCT
  Declares RAM parameters for the CSD Button.

- struct Senzei_RAM_WD_SLIDER_STRUCT
  Declares RAM parameters for the Slider.
- struct Senzei_RAM_WD_CSD_MATRIX_STRUCT
  Declares RAM parameters for the CSD Matrix Buttons.
- struct Senzei_RAM_WD_CSD_TOUCHPAD_STRUCT
  Declares RAM parameters for the CSD Touchpad.
- struct Senzei_RAM_WD_PROXIMITY_STRUCT
  Declares RAM parameters for the CSD Proximity.
- struct Senzei_RAM_WD_CSX_MATRIX_STRUCT
  Declares RAM parameters for the CSX Matrix Buttons.
- struct Senzei_RAM_WD_LIST_STRUCT
  Declares RAM structure with all defined widgets.
- struct Senzei_RAM_SNS_STRUCT
  Declares RAM structure for sensors.
- struct Senzei_RAM_SNS_LIST_STRUCT
  Declares RAM structure with all defined sensors.
- struct Senzei_RAM_STRUCT
  Declares the top-level RAM Data Structure.
- struct Senzei_FLASH_IO_STRUCT
  Declares the Flash IO object.
- struct Senzei_FLASH_SNS_STRUCT
  Declares the Flash Electrode object.
- struct Senzei_FLASH_SNS_LIST_STRUCT
  Declares the structure with all Flash electrode objects.
- struct Senzei_FLASH_WD_STRUCT
  Declares Flash widget object.
- struct Senzei_FLASH_STRUCT
  Declares top-level Flash Data Structure.
- struct Senzei_SHIELD_IO_STRUCT
  Declares the Flash IO structure for Shield electrodes.
- struct Senzei_BSLN_RAW_RANGE_STRUCT
  Defines the structure for test of baseline and raw count limits which will be determined by user for every sensor grounding on the manufacturing specific data.

Data Structure Documentation

struct ADAPTIVE_FILTER_CONFIG_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

struct ADVANCED_CENTROID_POSITION_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

struct ADVANCED_CENTROID_TOUCH_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

struct SMARTSENSE_CSD_NOISE_ENVELOPE_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

struct Senzei_RAM_WD_BASE_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
Senzei_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
Senzei_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[Senzei_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widget.
uint8	rowIdacMod[Senzei_	Unused.

	NUM_SCAN_FREQS]	
uint8	idacGainIndex	The index of the IDAC gain in the IDAC gain table structure for the widgets.
uint16	snsClk	Sets Lx clock divider for ISX 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	Unused.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance

struct Senzei_RAM_WD_BUTTON_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
Senzei_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
Senzei_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[Senzei_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widget.
uint8	rowIdacMod[Senzei_NUM_SCAN_FREQS]	Unused.
uint8	idacGainIndex	The index of the IDAC gain in the IDAC gain table structure for the widgets.
uint16	snsClk	Sets Lx clock divider for ISX 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	Unused.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance

struct Senzei_RAM_WD_SLIDER_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
Senzei_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
Senzei_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[Senzei_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widget.
uint8	rowIdacMod[Senzei_NUM_SCAN_FREQS]	Unused.
uint8	idacGainIndex	The index of the IDAC gain in the IDAC gain table structure for the widgets.
uint16	snsClk	Sets Lx clock divider for ISX 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	Unused.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
uint16	position[Senzei_NUM_CENTROIDS]	Reports the widget position.

struct Senzei_RAM_WD_CSD_MATRIX_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
Senzei_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the

		finger or touch/proximity threshold. OFF to ON.
Senzei_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[Senzei_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widget.
uint8	rowIdacMod[Senzei_NUM_SCAN_FREQS]	Unused.
uint8	idacGainIndex	The index of the IDAC gain in the IDAC gain table structure for the widgets.
uint16	snsClk	Sets Lx clock divider for ISX 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	Unused.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
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 Senzei_RAM_WD_CSD_TOUCHPAD_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
Senzei_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
Senzei_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[Senzei_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widget.
uint8	rowIdacMod[Senzei_NUM_SCAN_FREQS]	Unused.

uint8	idacGainIndex	The index of the IDAC gain in the IDAC gain table structure for the widgets.
uint16	snsClk	Sets Lx clock divider for ISX 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	Unused.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
uint16	posX	The X coordinate.
uint16	posY	The Y coordinate.

struct Senzei_RAM_WD_PROXIMITY_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
Senzei_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
Senzei_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[Senzei_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widget.
uint8	rowIdacMod[Senzei_NUM_SCAN_FREQS]	Unused.
uint8	idacGainIndex	The index of the IDAC gain in the IDAC gain table structure for the widgets.
uint16	snsClk	Sets Lx clock divider for ISX 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	Unused.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance
Senzei_THRESHOLD_T	proxTouchTh	The proximity touch threshold.

YPE		

struct Senzei_RAM_WD_CSX_MATRIX_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	resolution	Provides scan resolution or number of sub-conversions.
Senzei_THRESHOLD_TYPE	fingerTh	Widget Finger Threshold.
uint8	noiseTh	Widget Noise Threshold.
uint8	nNoiseTh	Widget Negative Noise Threshold.
uint8	hysteresis	Widget Hysteresis for the signal crossing finger or touch/proximity threshold.
uint8	onDebounce	Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.
Senzei_LOW_BSLN_RST_TYPE	lowBslnRst	The widget low baseline reset count. Specifies the number of samples the sensor has to be below the Negative Noise Threshold to trigger a baseline reset.
uint8	idacMod[Senzei_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widget.
uint8	rowIdacMod[Senzei_NUM_SCAN_FREQS]	Unused.
uint8	idacGainIndex	The index of the IDAC gain in the IDAC gain table structure for the widgets.
uint16	snsClk	Sets Lx clock divider for ISX 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	Unused.
uint16	sigPFC	The 75% of signal per user-defined finger capacitance

struct Senzei_RAM_WD_LIST_STRUCT

Go to the top of the Data Structures section.

Data Fields:

Senzei_RAM_WD_BUTTON_STRUCT	button0	Button0 widget RAM structure
Senzei_RAM_WD_SLIDER_STRUCT	linearslider0	LinearSlider0 widget RAM structure
Senzei_RAM_WD_SLIDER_STRUCT	radialslider0	RadialSlider0 widget RAM structure

Senzei_RAM_WD_CSD_MATRIX_STRUCT	matrixbuttons0	MatrixButtons0 widget RAM structure
Senzei_RAM_WD_CSD_TOUCHPAD_STRUCT	touchpad0	Touchpad0 widget RAM structure
Senzei_RAM_WD_PROXIMITY_STRUCT	proximity1	Proximity1 widget RAM structure
Senzei_RAM_WD_BUTTON_STRUCT	button1	Button1 widget RAM structure
Senzei_RAM_WD_CSX_MATRIX_STRUCT	matrixbuttons1	MatrixButtons1 widget RAM structure
Senzei_RAM_WD_PROXIMITY_STRUCT	proximity0	Proximity0 widget RAM structure

struct Senzei_RAM_SNS_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

struct Senzei_RAM_SNS_LIST_STRUCT

Go to the top of the Data Structures section.

Data Fields:

Senzei_RAM_SNS_STRUCT	button0[Senzei_BUTTON0_NUM_SENSORS]	Button0 sensors RAM structures array
Senzei_RAM	linearslider0[Senzei_L	LinearSlider0 sensors RAM structures array

_SNS_STRUCT	INEARSLIDER0_NUM_SENSORS]	
Senzei_RAM_SNS_STRUCT	radialslider0[Senzei_RADIALSLIDER0_NUM_SENSORS]	RadialSlider0 sensors RAM structures array
Senzei_RAM_SNS_STRUCT	matrixbuttons0[Senzei_MATRIXBUTTONS0_NUM_COLS+Senzei_MATRIXBUTTONS0_NUM_ROWS]	MatrixButtons0 sensors RAM structures array
Senzei_RAM_SNS_STRUCT	touchpad0[Senzei_TOUCHPAD0_NUM_COLS+Senzei_TOUCHPAD0_NUM_ROWS]	Touchpad0 sensors RAM structures array
Senzei_RAM_SNS_STRUCT	proximity1[Senzei_PROXIMITY1_NUM_SENSORS]	Proximity1 sensors RAM structures array
Senzei_RAM_SNS_STRUCT	button1[Senzei_BUTTON1_NUM_SENSORS]	Button1 sensors RAM structures array
Senzei_RAM_SNS_STRUCT	matrixbuttons1[(Senzei_MATRIXBUTTONS1_NUM_RX)*(Senzei_MATRIXBUTTONS1_NUM_TX)]	MatrixButtons1 sensors RAM structures array
Senzei_RAM_SNS_STRUCT	proximity0[Senzei_PROXIMITY0_NUM_RX]	Proximity0 sensors RAM structures array

struct Senzei_RAM_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint16	configId	16-bit CRC calculated by the customizer for the component configuration. Used by the Tuner application to identify if the FW corresponds to the specific user configuration.
uint16	deviceId	Used by the Tuner application to identify device-specific configuration.
uint16	hwClock	Used by the Tuner application to identify the system clock frequency.
uint16	tunerCmd	Tuner Command Register. Used for the communication between the Tuner GUI and the component.
uint16	scanCounter	This counter gets incremented after each scan.
volatile uint32	status	Status information: Current Widget, Scan active, Error code.
uint32	wdgtEnable[Senzei_WDGT_STATUS_WORDS]	The bitmask that sets which Widgets are enabled and scanned, each bit corresponds to one widget.
uint32	wdgtStatus[Senzei_WDGT_STATUS_WOR	The bitmask that reports activated Widgets (widgets that detect a touch signal above the

	DS]	threshold), each bit corresponds to one widget.
Senzei_SNS_STS_TYPE	snsStatus[Senzei_TOTAL_WIDGETS]	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 Buttonwidget, only one bit is used and means: 0 - Not active; 1 - Finger detected. For the Encoder Dial Widget, four bits are used. The lower 2 bits represent current sensor status; the upper 2 bits represent previous sensor statuses. E.g., 0111 means sensor0 active, sensor1 active, sensor0 was active, sensor 1 was not. The array size is equal to the total number of widgets. The size of the array element depends on the max number of sensors per widget used in the current design. It could be 1, 2 or 4 bytes.
uint16	csd0Config	The configuration register for global parameters of the SENSE_HW0 block.
uint8	modCsdClk	The modulator clock divider for the CSD widgets.
uint8	modCsxClk	The modulator clock divider for the CSX widgets.
uint8	modIsxClk	The modulator clock divider for the ISX widgets.
Senzei_RAM_WD_LIST_STRUCT	wdgtList	RAM Widget Objects.
Senzei_RAM_SNS_LIST_STRUCT	snsList	RAM Sensor Objects.
uint8	snrTestWidgetId	The selected widget ID.
uint8	snrTestSensorId	The selected sensor ID.
uint16	snrTestScanCounter	The scan counter.
uint16	snrTestRawCount[Senzei_NUM_SCAN_FREQS]	The sensor raw counts.
uint8	scanCsdISC	The inactive sensor connection state for the CSD sensors.
uint8	scanCsxISC	The inactive sensor connection state for the CSX sensors.
uint8	scanCurrentISC	The current inactive sensor connection state for the sensors.

struct Senzei_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 Senzei_FLASH_SNS_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

struct Senzei_FLASH_SNS_LIST_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint8	notUsed	No ganged sensors available

struct Senzei_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.
Senzei_RAM_SNS_STRUCT*	ptr2SnsRam	Points to the array of Sensor Objects in RAM. The sensing and processing blocks use it to access the scan data.
void *	ptr2FltrHistory	Points to the array of the Filter History Objects in RAM that belongs to this widget.
uint8 *	ptr2DebounceArr	Points to the array of the debounce counters. The size of the debounce counter is 8 bits. These arrays are not part of the data structure.
uint32	staticConfig	Miscellaneous configuration flags.
uint16	totalNumSns	The total number of sensors. For CSD widgets: WD_NUM_ROWS + WD_NUM_COLS. For CSX widgets:

		WD_NUM_ROWS * WD_NUM_COLS.
uint8	wdgtType	Specifies one of the following widget types: WD_BUTTON_E, WD_LINEAR_SLIDER_E, WD_RADIAL_SLIDER_E, WD_MATRIX_BUTTON_E, WD_TOUCHPAD_E, WD_PROXIMITY_E
uint8	senseMethod	Specifies the widget sensing method.
uint8	numCols	For ISX Proximity Widgets, the number of sensors.
uint8	numRows	Unused.
uint16	xResolution	Sliders: The Linear/Angular resolution. Touchpad: The X-Axis resolution.
uint16	yResolution	Touchpad: The Y-Axis resolution.
uint32	xCentroidMultiplier	The pre-calculated X resolution centroid multiplier used for the X-axis position calculation. Calculated as follows: RADIAL: (WD_X_RESOLUTION * 256) / WD_NUM_COLS; LINEAR and TOUCHPAD: (WD_X_RESOLUTION * 256) / (WD_NUM_COLS - CONFIG); where CONFIG is 0 or 1 depends on CentroidMultiplerMethod parameter
uint32	yCentroidMultiplier	The pre-calculated Y resolution centroid multiplier used for the Y-axis position calculation. Calculated as follows: (WD_Y_RESOLUTION * 256) / (WD_NUM_ROWS - CONFIG); where CONFIG is 0 or 1 depends on CentroidMultiplerMethod parameter
SMARTSENSE_CSD_NOISE_ENVELOPE_STRUCT*	ptr2NoiseEnvlp	Unused.
uint8	iirFilterCoeff	The position IIR filter coefficient.

struct Senzei_FLASH_STRUCT

Go to the top of the Data Structures section.

Data Fields:

Senzei_FLASH_WD_STRUCT	wdgtArray[Senzei_TOTAL_WIDGETS]	Array of flash widget objects

struct Senzei_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 Senzei_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 a given Component configuration.

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

PSoC 4 (GCC)

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

Configuration	Memory Consumption	
	Flash	SRAM
Configuration #1: CSX Matrix Button – One widget with 4 Rx and 8 Tx.		
Configuration #1	< 5300	< 500
Configuration #1 + *Enable multi-frequency scan* is enabled	< 5700	< 1000
Configuration #2: CSX Touchpad – One widget with 9 Rx and 4 Tx.		
Configuration #2	< 7500	< 800
Configuration #2 + *Enable multi-frequency scan* is enabled	< 7900	< 1350
Configuration #3: CSD Buttons – Three widgets with 4, 3 and 3 sensors in each widget, and Manual tuning mode is selected.		
Configuration #3	< 5800	< 300
Configuration #3 + *Enable multi-frequency scan* is enabled	< 6300	< 450
Configuration #3 + *Enable self-test library* is enabled	< 10900	< 400
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)*.	< 6200	< 400

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

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

Tuner

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

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

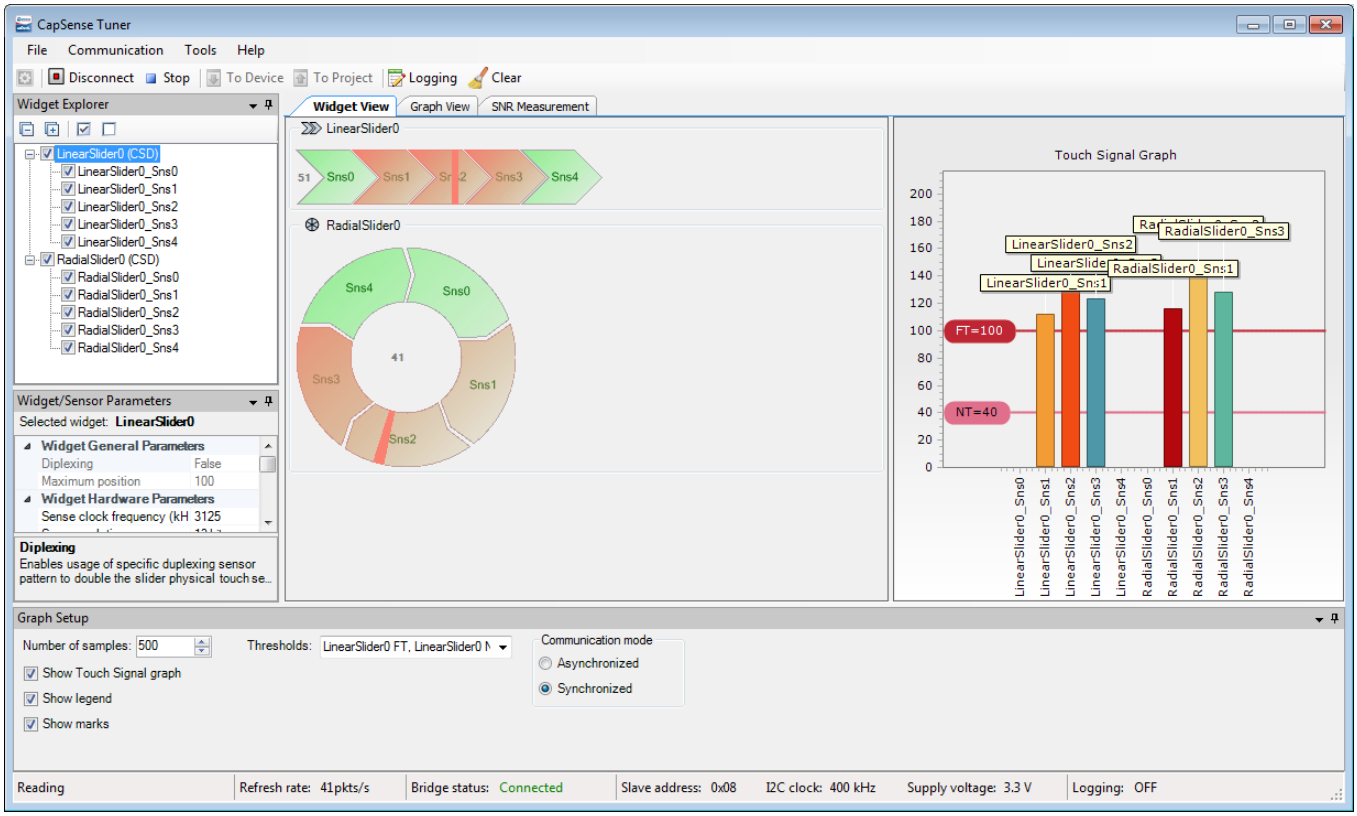
To edit the parameters, use the Tuner application and apply the new settings to the device using the **To Device** button. You can do this when using *Manual* or *SmartSense (Hardware parameters only)* modes for tuning.

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

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

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

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 commands to control and navigate the Tuner:

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

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

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

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

Toolbar

Contains frequently used buttons that duplicate the main menu items:

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

Status Bar

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

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

Widget Explorer Pane

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

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

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

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

Widget/Sensor Parameters Pane

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

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

- **Sensor Parameters** – Sensor-specific parameters. The Tuner application displays only *IDAC Values* or/and *Compensation IDAC value*. The parameter is not present for the CSD widget when *Enable compensation IDAC* is disabled on the customizer *CSD Settings* tab. When CSD *Enable IDAC auto-calibration* or/and CSX *Enable IDAC auto-calibration* is enabled, the parameter is Read-only and displays the IDAC value as calibrated by the Component firmware. When auto-calibration is disabled, the IDAC value entered in the Configure dialog is shown. If the Tuner is in **Synchronized** mode, you can edit the value and apply it to the device.
- **Filter Parameters** and **Centroid Parameters** – Cannot be modified at run-time from the Tuner, because unlike the other parameters, these parameter values reside in the Flash widget structures that cannot be modified at run-time.
- **Gesture Parameters** – *Synchronized* communication mode must be selected to update the Gesture parameters during run-time from the Tuner application.

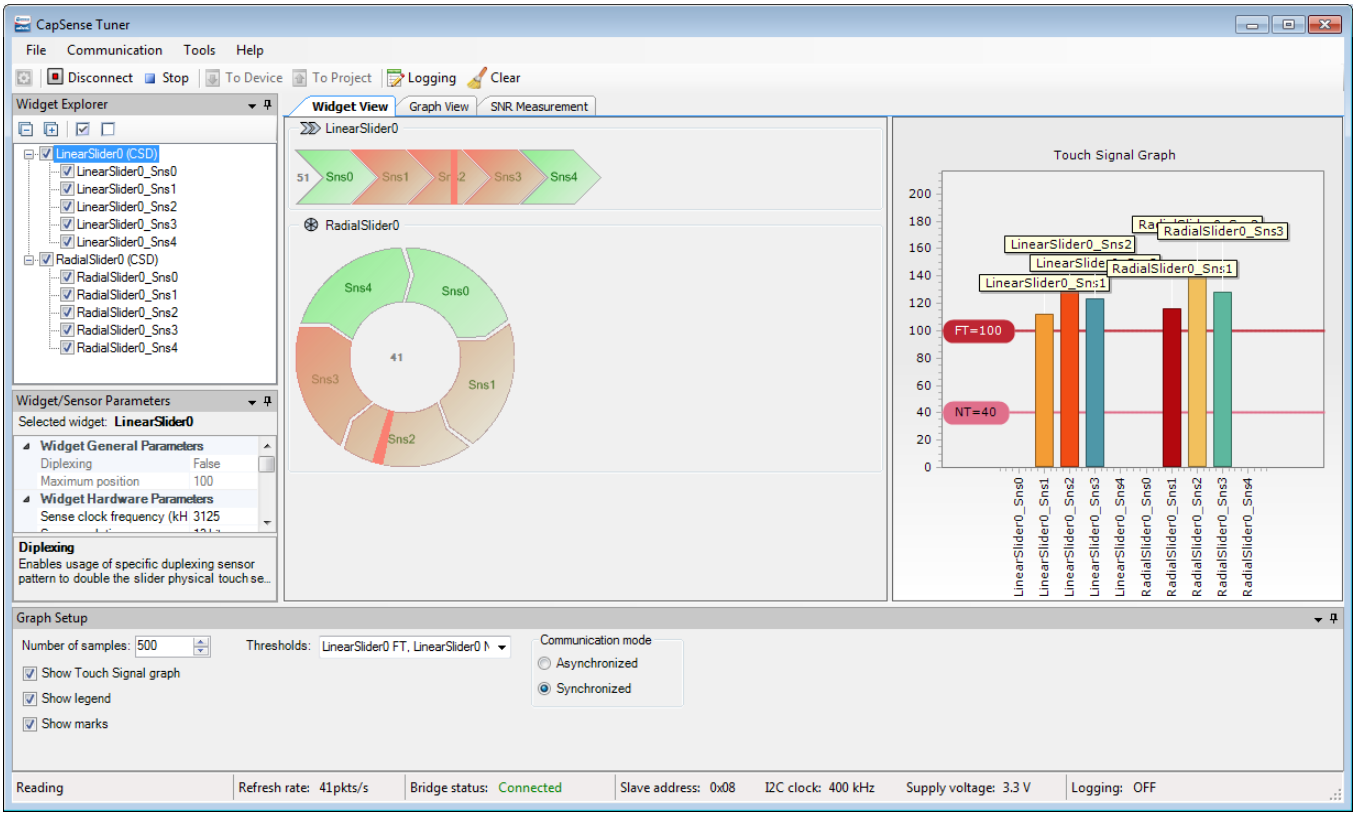
Graph Setup Pane

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

- **Number of samples** – Defines the total amount of data samples shown on a single graph.
- **Show legend** – Displays the sensor series descriptions (with names and colors) in graphs when checked (Sensor Data/Sensor Signal/Status graphs in the *Graph View* and a *Touch Signal Graph* in the *Widget View*).
- **Show marks** – When checked, the sensor names appear as marks over the signal bars in the *Touch Signal Graph*.
- **Show Touch Signal graph** – When checked, a *Touch Signal Graph* appears.
- **Thresholds** – A drop-down menu with checkboxes to enable the threshold visualization in the *Touch Signal Graph* and a Sensor Signal graph in the *Graph View* tab.
- **Communication mode** – Selects Tuner communication mode with a device. Two options are available (when the EZI2C Component is used):
  - **Synchronized** – This communication mode is available when a FW loop periodically calls a corresponding Tuner function: CapSense_RunTuner(). When Synchronized Communication mode is selected, the CapSense Tuner manages an execution flow by suspending scanning during the Read operation. Before starting data reading, the Tuner sends a **OneScan** command to the device. The device performs one cycle of scanning and the second call of CapSense_RunTuner() hangs the FW flow until a new command is received. The Tuner reads all the needed data and sends a **OneScan** command again.

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

Widget View



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

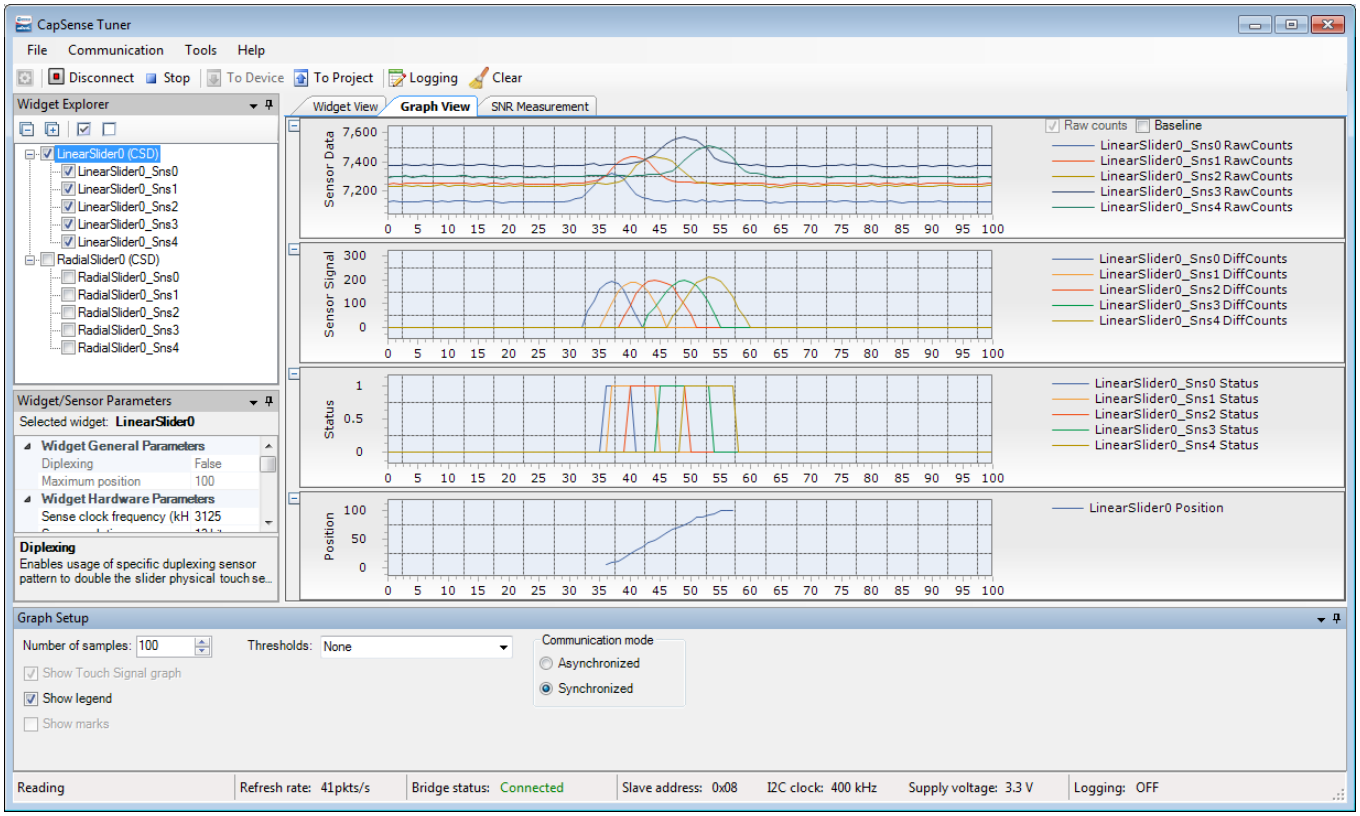
The Widget sensors are highlighted red when the device reports their touch status as active.

Some additional features are available depending on the widget type:

Touch Signal Graph

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

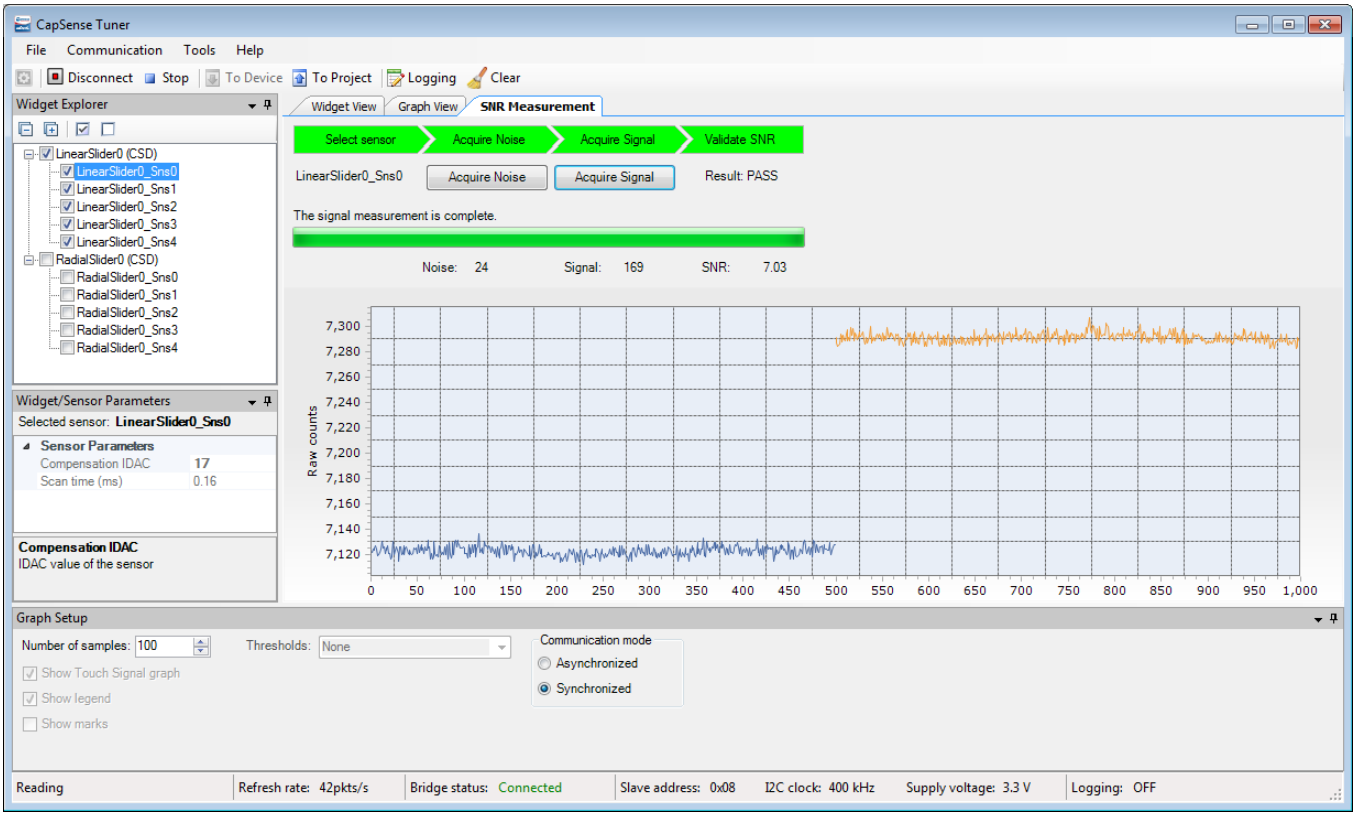
Graph View



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

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

SNR Measurement



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

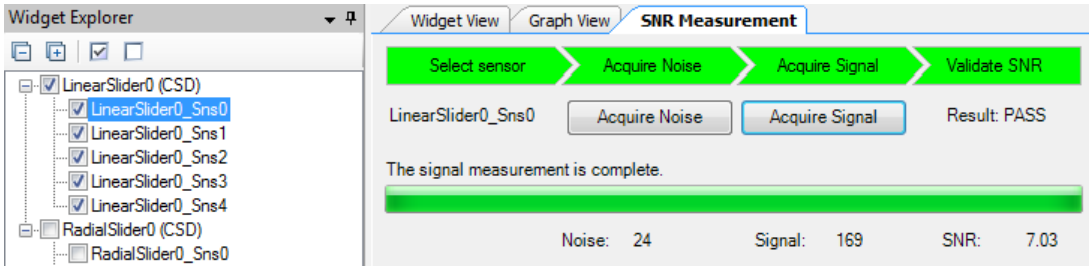
The tab provides UI to acquire noise and signal samples separately and then calculates a SNR basing on the captured data. The obtained value is then validated by a comparison with the required minimum (5 by default, can be configured in the *Tuner Configuration Options*).

Typical Flow of SNR Measurement

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

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

Description of SNR Measurement GUI



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

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

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

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

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

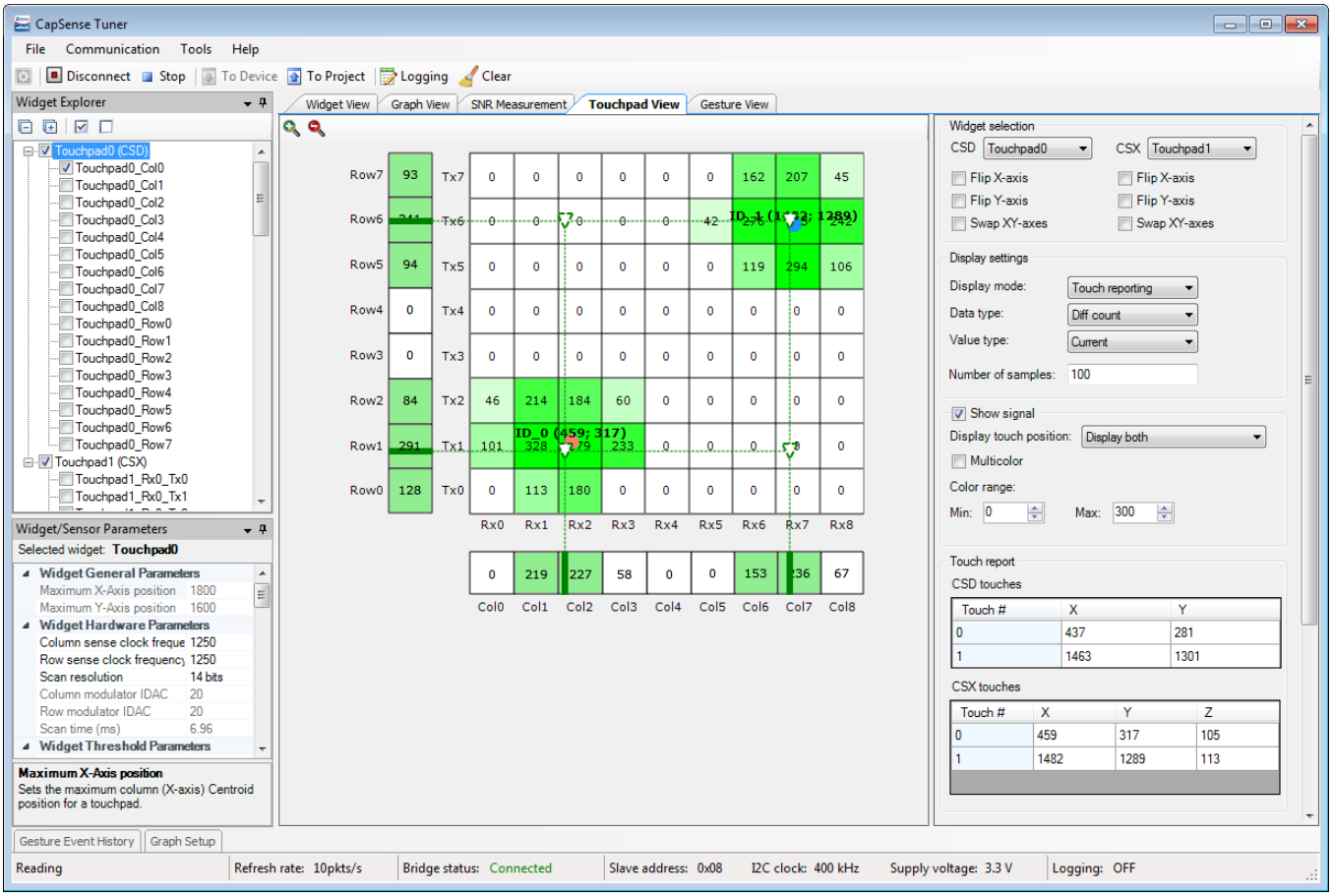
Below the status label, there are the following controls:

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

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

Touchpad View

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



The following options are available:

Widget Selection

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

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

Display settings

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

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

Show signal

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

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

- Display only CSD
- Display both

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

Touch report

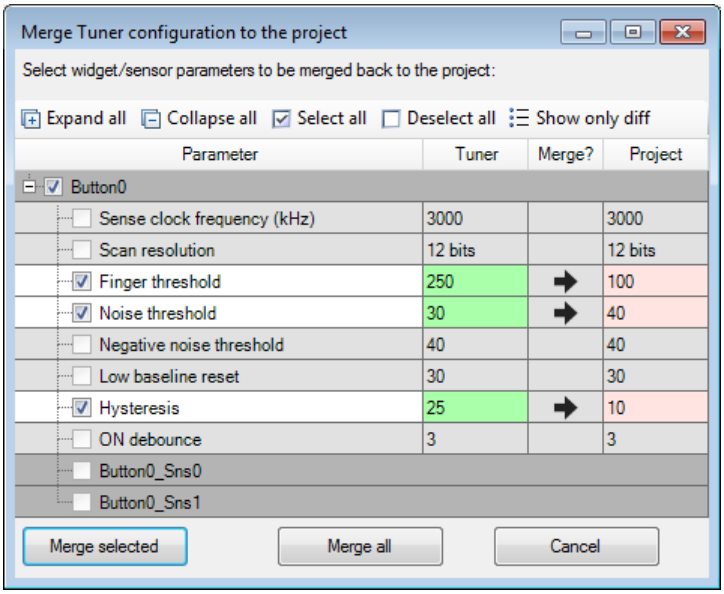
- CSD touches table – Displays the current X and Y touch position of the CSD touchpad configured in **CSD combo box.** If the CSD touchpad is neither configured nor touch-detected, the touch table is empty. When *Two finger* detection is enabled for a CSD touchpad, then two touch positions are reported.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

Procedure to Save Tuner Parameters

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

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

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



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

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

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

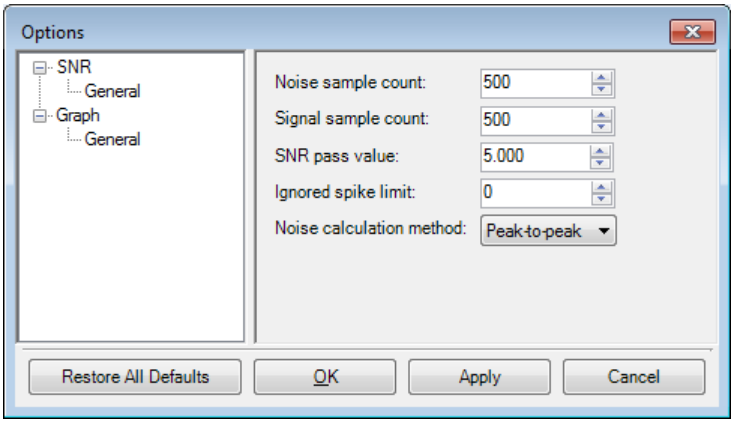
   The Tuner automatically picks up the changed parameters from a device. Clicking **To Project** merges these parameters to the Component and later they can be used as a starting point for manual calibration or tuning.

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

Tuner Configuration Options

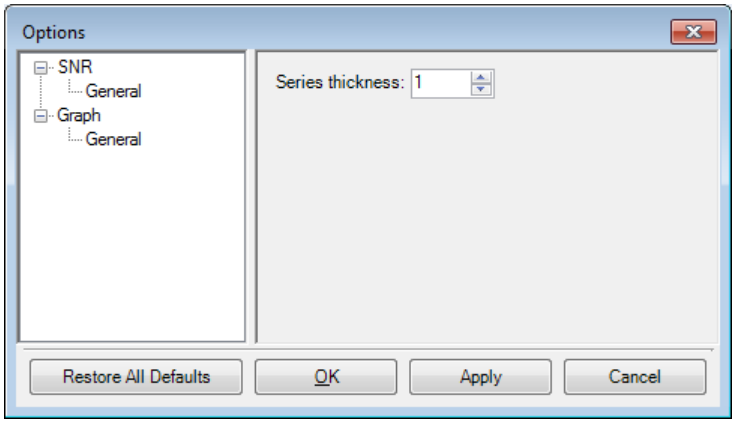
The Tuner application allows setting different configuration options with the Options dialog. Settings are applied on per-project basis and divided into groups:

SNR Options



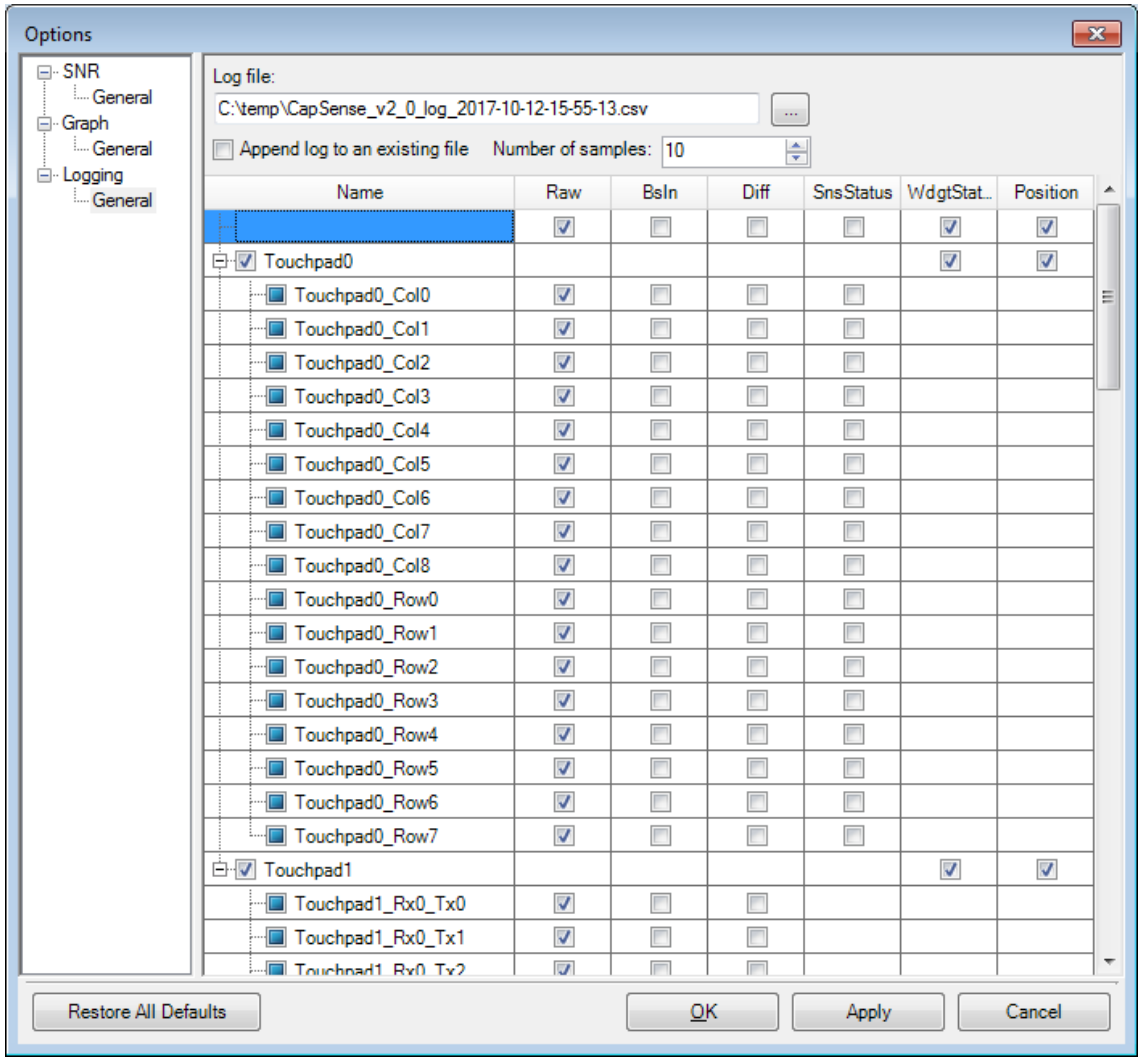
- **Noise sample count** – The count of samples to acquire during the noise measurement operation.
- **Signal sample count** – The count of samples to acquire during the signal measurement operation.
- **SNR pass value** – The minimal acceptable value of the SNR.
- **Ignore spike limit** – Ignores a specified number of the highest and the lowest spikes at noise / signal calculation. That is, if you specify number 3, then three upper and lower three raw counts are ignored separately for the noise calculation and for the signal calculation.
- **Noise calculation method** – Allows selecting the method to calculate the noise average. The following methods are available for selection:
  - **Peak-to-peak** (by default) – Calculates noise as a difference between the maximum and minimum value collected during the noise measurement.
  - **RMS** – Calculates noise as a root mean-square of all samples collected during the noise measurement.

Graph options



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

Data Log Options



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

MISRA Compliance Report

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

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

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

The CapSense_P4 Component has the following specific deviations:

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

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

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

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

Component Debug Window

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

To open the Component Debug window:

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

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

Resources

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

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

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

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

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

Additionally, the following may be consumed:

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

References

General References

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

Application Notes

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

- *AN64846* – Getting Started with CapSense®
- *AN72362* – Reducing Radiated Emissions in Automotive CapSense® Applications
- *AN85951* – PSoC® 4 CapSense® Design Guide
- *AN92239* – Proximity Sensing with CapSense®

Code Examples

PSoC Creator provides access to code examples in the Find Code Example dialog. For Component-specific examples, open the dialog from the Component Catalog or an instance of the Component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the "Find Code Example" topic in the PSoC Creator Help for more information.

There are also numerous code examples that include schematics and code examples available online at the *Cypress Code Examples web page*. The examples that use this Component include:

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

Development Kit Boards

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

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

- *CY8CKIT-041 PSoC® 4 S-Series Pioneer Kit*

Electrical Characteristics

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

Performance Characteristics

Parameter	Condition	Typical	Units
Sensor Calibration level (Applicable for sensor with highest Cp within a Widget)	Cp = 5 to 45 pF (Single IDAC mode)	85% of full scale ±5 %	-
	Cp = 5 to 45 pF (Dual IDAC mode)	85% of full scale ±10 %	-
Touch signal accuracy The touch signal is the difference between measured raw counts with and without a finger present on a sensor (difference count).		Not less than 10% of sensor sensitivity.	-
Supported Sensor Cp range		Min: 5. Max: 45	pF
SNR (Noise Floor) The simple ratio of (Signal/Noise) is called the CapSense SNR. It is usually simplified to [(Finger Signal/Noise): 1]	Cp < 35 pF Single IDAC: Finger capacitance >= 0.2 pF Dual IDAC: Finger capacitance >= 0.1 pF	> 5:1	-
	Cp < 45 pF Single IDAC: Finger capacitance >= 0.2 pF Dual IDAC: Finger capacitance >= 0.1 pF	> 4:1	-
Supply (VDD) ripple	VDD > 3.3 V, Finger capacitance = 0.1 pF, VDD ripple +/-50 mV	< 30% of noise	
	VDD < 2 V, internally regulated mode. Finger capacitance = 0.4 pF, VDD ripple +/-50 mV	< 30% of noise	
	VDD < 2 V, externally regulated mode. Finger capacitance = 0.4 pF, VDD ripple +/-25 mV	< 30% of noise	
GPIO Sink Current	10 mA per GPIO on multiple pin to sink max current. Device max = 25 mA.	< 30% of noise	

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	50/60 Hz, noise Vpp = 20 V	< 30%	-
Finger-Conducted AC Noise is the change in the sensor raw count when AC noise is applied on the sensor (injected into the system)	10 kHz to 1 MHz, noise Vpp = 20 V, Cp < 10 pF	< 30%	-
Interrupt immunity Excessive raw counts noise at asynchronous interrupts used.		< 30%	-
Current Consumption	1 CSD Button Widget (Ganged Sensor, 4 electrodes). Resolution = 9 bits. Each electrode Cp < 10 pF. Shield Electrode = Disabled. SYSCLK = 16 MHz. No I2C traffic (I2C block ON). Report Rate >= 8 Hz. Chip state = DeepSleep (LFT).	< 6 (PSoC 4000)	µA
		< 7 (PSoC 4000S)	µA
	1 CSD Button Widget, 8 Sensors. Resolution = 9 bits. 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).	< 18 (PSoC 4000)	µA
		< 22 (PSoC 4000S)	µA
		< 6 (PSoC 4000)	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. 9 mm metal finger. 1 Touch only. Report Rate >= 8 Hz. Chip state = DeepSleep (LFT).	< 150 (PSoC 4000)	µA
		< 200 (PSoC 4000S)	µA

IDAC Characteristic

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

DC/AC Specifications

Refer to device-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
7.0.b	Updated datasheet to add description for CSX inactive electrode connection.	Description was missing.
7.0.a	Minor datasheet edits.	
7.0	Initial version	Combine functionality of CapSense and MagSense

© Cypress Semiconductor Corporation, 2019-2020. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC (“Cypress”). This document, including any software or firmware included or referenced in this document (“Software”), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress’s patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

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

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