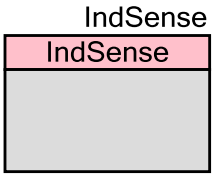
Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the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

5.10



Features

- Supports Inductive sensing up to frequencies of 3 MHz.
- Supports auto-tuning algorithm for easy tuning and reliable operation.
- Supports up to eight simultaneous Inductive sensing inputs.
- Contains an integrated graphical user interface for tuning, testing and debugging.
- Sampling rate up to 10 ksps.

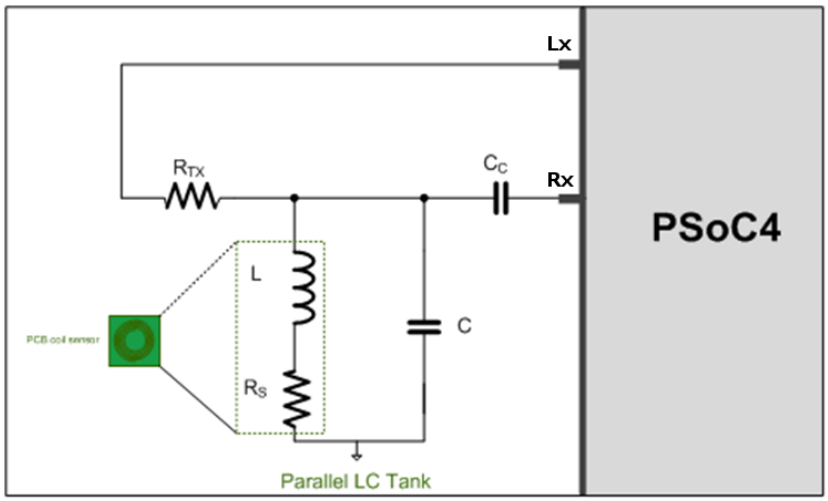
General Description

Inductive sensing technology enables touch and proximity detection for human interface on a wide variety of materials including both ferrous and non-ferrous materials. The touches are detected by measuring small deflections of conductive targets. Cypress's Inductive sensing solution supports up to 16 inputs and it is insensitive to environmental changes and non-conductive objects such as dirt and liquids etc. Touch sensing over metal overlays provides the ability to design cool aesthetics for product user interfaces. Auto-calibration algorithm automatically compensates overlay deformations over time and provides reliable operation.

The inductive sensor is an inductive coil formed using copper trace on a PCB. It is represented electrically as an inductor with a series AC resistance (RS). The coil is combined with a parallel capacitor (C) to form a parallel LC tank. The coil is driven at resonance by a LX voltage where the resonant frequency is defined as:

$$f_0 = \frac{1}{2\pi}\sqrt{\frac{1}{(LC)} - \left(\frac{R_S}{L}\right)^2}$$

The signal from the LC tank is then coupled into Rx electrode through a capacitor CC where it is digitized by an analog to digital converter. A metal object changes the amplitude of oscillation of the LC tank, changing the digitized output code.



The IndSense-Component solution includes a configuration wizard to create and configure inductive sensing widgets, APIs to control the Component from the application firmware, and an *IndSense Tuner* application for tuning, testing, and debugging.

This datasheet includes the following sections:

- *Quick Start* – Helps you quickly configure the Component to create a simple demo.
- *Component Configuration Parameters* – Contains descriptions of the Component's parameters in the configuration wizard.
- *Application Programming Interface* (APIs) – Provides descriptions of all APIs in the firmware library, as well as descriptions of all data structures (Register map) used by the firmware library.
- *IndSense Tuner* – Contains descriptions of all user-interface controls in the tuner application.
- *Electrical Characteristics* – Provides the Component performance specifications and other details such as certification specifications.

When to Use a IndSense Component

Applications for Inductive sensing include:

- Mechanical open/close switch replacement
  - White goods door open/close.
  - Home security and Tamper detection.
- Buttons
  - Industrial keypads
  - Metallic on/off buttons
- Distance measurement
  - Proximity detection
- Rotation detection
  - Flow Meters
  - Fan speed RPM detection
  - Incremental rotary control knob.

Quick Start

This section will help you create a PSoC Creator project with a *Encoder* **Dial** is a widget consisting of two sensors. Each sensor detects the presence or absence of metal. Then, based on the pattern of sensors compared to the pattern of metal on a dial, detects rotation.

- **Button** is a widget consisting of one sensor. Each widget can detect the presence or absence (that is, only two states) of a metal object on the sensor.

Proximity interface.

In order to monitor performance of the sensor using the *IndSense Tuner*, refer to the *Tuner Quick Start* section once the PSoC Creator project has been created.

Step-1: Create a Design in PSoC Creator

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

If required, refer to the following documents for more information:

- *PSoC Creator Quick Start Guide*
- *PSoC Creator User Guide*

Step-2: Place and Configure the IndSense Component

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

Double-click on the Component in the schematic to open the Configure dialog. Type the desired Component name (in this case: *IndSense* for the code in Step-3 to work).

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

Basic Tab

Use this tab to select the *Widget type* and a number of Widgets required for the design. Click '**+**' to add an ISX proximity sensor widget to the design. See *Basic Tab* for more information.

Note Each widget consumes two port pins from the device.

Advanced Tab

Use this tab to configure parameters required for an extensive level of manual tuning. For this project, use the default values for parameters in this tab. This tab has multiple sub-tabs used to systematically arrange parameters. See *Advanced Tab* for details about these parameters.

Step-3: Write Application Code

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

```
#include <project.h>

int main()
{
    CyGlobalIntEnable;                          /* Enable global interrupts */

    IndSense_Start();                           /* Initialize Component */
    IndSense_ScanAllWidgets();                  /* Scan all widgets */

    for(;;)
    {
        /* Do this only when a scan is done */
        if(IndSense_NOT_BUSY == IndSense_IsBusy())
        {
            IndSense_ProcessAllWidgets();       /* Process all widgets */
            if (IndSense_IsAnyWidgetActive())   /* Scan result verification */
            {
                /* add custom tasks to execute when touch detected */
            }

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

Step-4: Assign Pins in Pin Editor

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

Step-5: Build Design

Select ***Build <project name>*** from the ***Build*** menu and see the project build without errors.

Further, you can add custom code in the above project to add indicator such as turn on an LED using GPIO when touch or proximity is detected.

Another choice is to add tuner interface to the project and monitor the status of sensors using tuner GUI tool. Refer to *Tuner Quick Start* section for procedure to add tuner interface to the project and start the tuning sensor on the hardware.

Input / Output Connections

This section describes the various input and output connections for the IndSense Component. These are not exposed as connectable terminals on the Component symbol but these terminals can be assigned to the port pins in the Pins tab of the Design-Wide Resources setting of PSoC Creator. The Pin Editor provides guidelines on the recommended pins for each terminal and does not allow an invalid pin assignment.

Name	I/O Type	Description
$C_{int}A$	Analog	Integration capacitor. Mandatory for operation of the ISX sensing method and required only if the ISX sensing is used. The recommended value is 470pF/5v/X7R or NP0 capacitors.
$C_{int}B$	Analog	
Lx	Digital Output	Transmitter electrodes of ISX widgets. There is one Lx electrode for each sensor.
Rx	Analog	Receiver electrodes of ISX widgets. There is one Rx electrode for each sensor.
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 provides a brief description of all configurable parameters in the Component Configure dialog. This section does not provide design and tuning guidelines. For complete guidelines, refer to the IndSense Design guide.

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

Common Controls

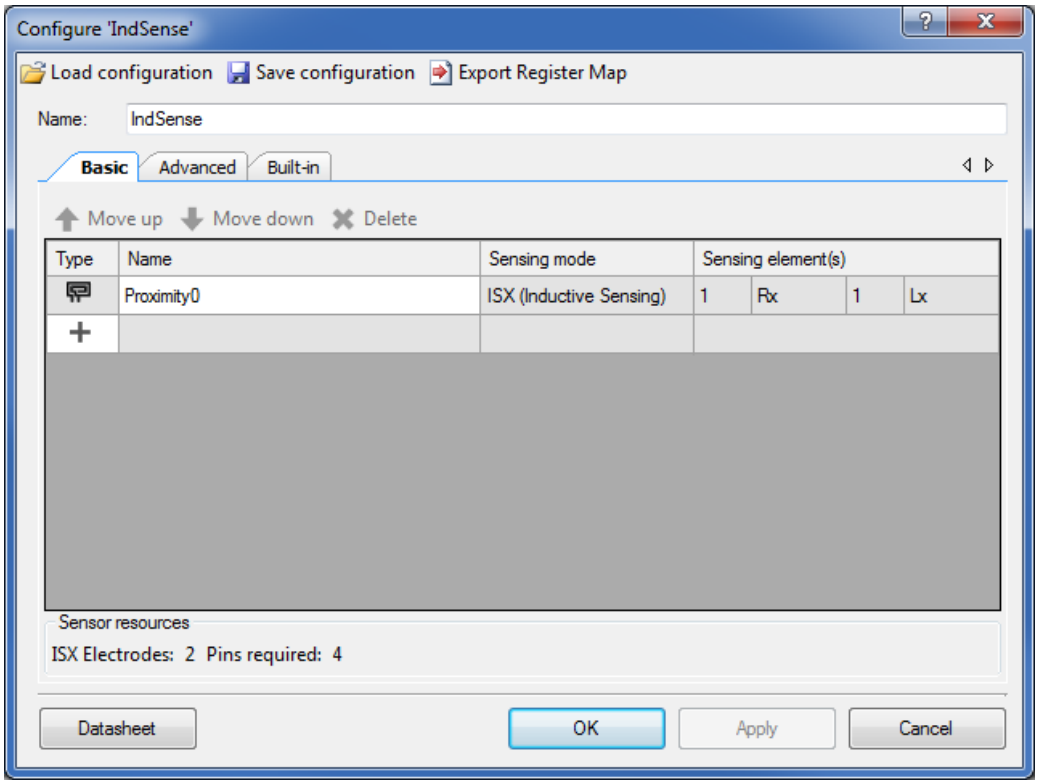
- **Load configuration** – Open (load) a previously saved configuration (XML) file for the IndSense Component.
- **Save configuration** – Save the current Component configuration into a (XML) file.
- **Export Register Map** – The IndSense Component firmware library uses a data structure (known as Register 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 PDF or XML file that can be used as a reference for development.

Basic Tab

The **Basic** tab defines the high-level Component configuration. Use this tab to add *Encoder* **Dial** is a widget consisting of two sensors. Each sensor detects the presence or absence of metal. Then, based on the pattern of sensors compared to the pattern of metal on a dial, detects rotation.

- **Button** is a widget consisting of one sensor. Each widget can detect the presence or absence (that is, only two states) of a metal object on the sensor.

Proximity widgets for the design.

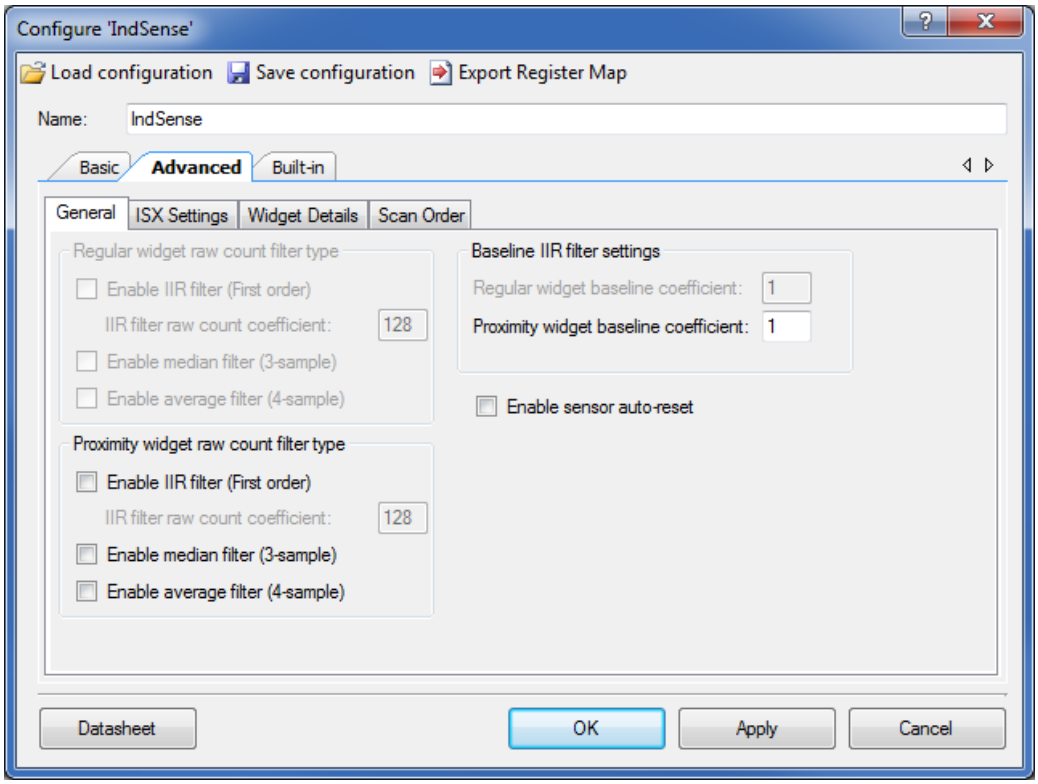


Name	Description
Widget type	A widget is an interface that performs a specific user-interface functionality. ▪ **Encoder Dial** is a widget consisting of two sensors. Each sensor detects the presence or absence of metal. Then, based on the pattern of sensors compared to the pattern of metal on a dial, detects rotation. ▪ **Button** is a widget consisting of one sensor. Each widget can detect the presence or absence (that is, only two states) of a metal object on the sensor. ▪ **Proximity Sensor** is a widget consisting of one sensor. Each sensor in the widget can detect the proximity of conductive objects such as metals. The proximity sensor has two thresholds: ○ *Proximity threshold*: To detect an approaching target ○ *Touch threshold*: To detect a target touch on the sensor.

Name	Description
Widget name	A widget name can be defined to aid in referring to a specific widget in the design. A widget name does not have any effect on functionality or performance and a widget name is used throughout the source code to generate macro values and data structure variables. A maximum of 16 alphanumeric characters (the first letter must be an alphabetic character) is acceptable for a widget name.
Sensing mode	Information**:** ISX sensing method is a Cypress patented method of performing inductive sensing measurements.
Widget Sensing element(s)	Information**:** The sensing element refers to Component terminals assigned to port pins to connect to physical sensors on the user-interface panel. Each ISX widgets uses a pair of electrodes, one Lx and one Rx.
Move up / Move down	Moves the selected widget up or down by one on the list. It defines the widget scanning order. **Note** Widget deleting may break a pin assignment and you will need to repair the assignment in the Pin Editor.
Delete	Deletes the selected widget from the list. **Note** Widget deleting may break a pin assignment and you will need to repair the assignment in the Pin Editor.
ISX electrodes	Information: Indicates the total number of electrodes (port pins) used by the ISX widgets, including the *CintA* and *CintB* capacitors.
Pins required	**Information:** Indicates the total number of port pins required for the design. This does not include port pins used by other Components in the project or SWD pins in the debug mode.

Advanced Tab

This tab provides advanced configuration parameters. Use this tab to configure parameters required for an extensive level of manual tuning.



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

- *General* – Contains all the parameters common for all widgets.
- *ISX Settings* – Contains all hardware parameters common for all widgets.
- *Widget Details* – Contains the parameters specific to widgets and/or sensors.
- *Scan Order* – Provides information scan time.

General Sub-tab

Contains parameters common for all widgets. The table below provides descriptions of parameters in this tab:

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

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

- Median filter
- IIR filter
- Average filter

However, the Component provides the ability to change the order using a low-level processing API. Refer to *Application Programming Interface* for details.

The filter algorithm is executed when any processing API is called by the application layer. When enabled, each filter consumes RAM to store a previous raw count (filter history). If multiple filters are enabled, the total filter history is correspondingly increased so that the size of the total filter history is equal to a sum of all enabled filter histories.

Proximity widget raw count filter type

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

Baseline filter settings

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

The design of baseline IIR filter is the same as the raw count IIR filter, but the filter coefficients are different for baseline and raw count filters to produce a different roll-off. The baseline filter is applied to the filtered raw count (if the widget raw count filters are enabled).

Name	Description
Proximity widget baseline coefficient	Baseline IIR filter coefficient selection for sensors of proximity widgets. The valid range is: 1-255.

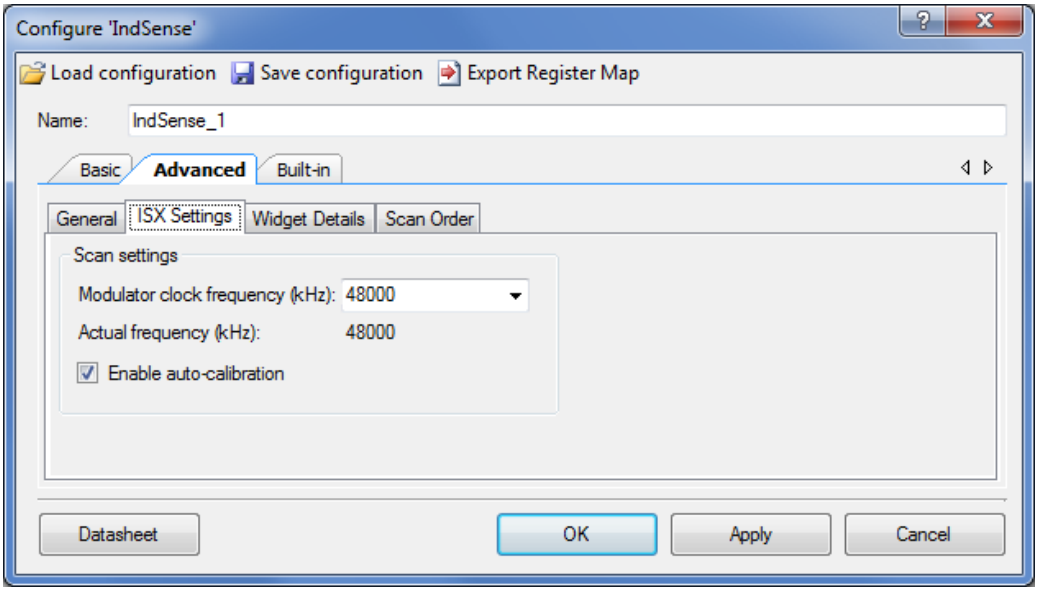
General settings

The general settings are applicable to the whole Component behavior.

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

ISX Settings Sub-tab

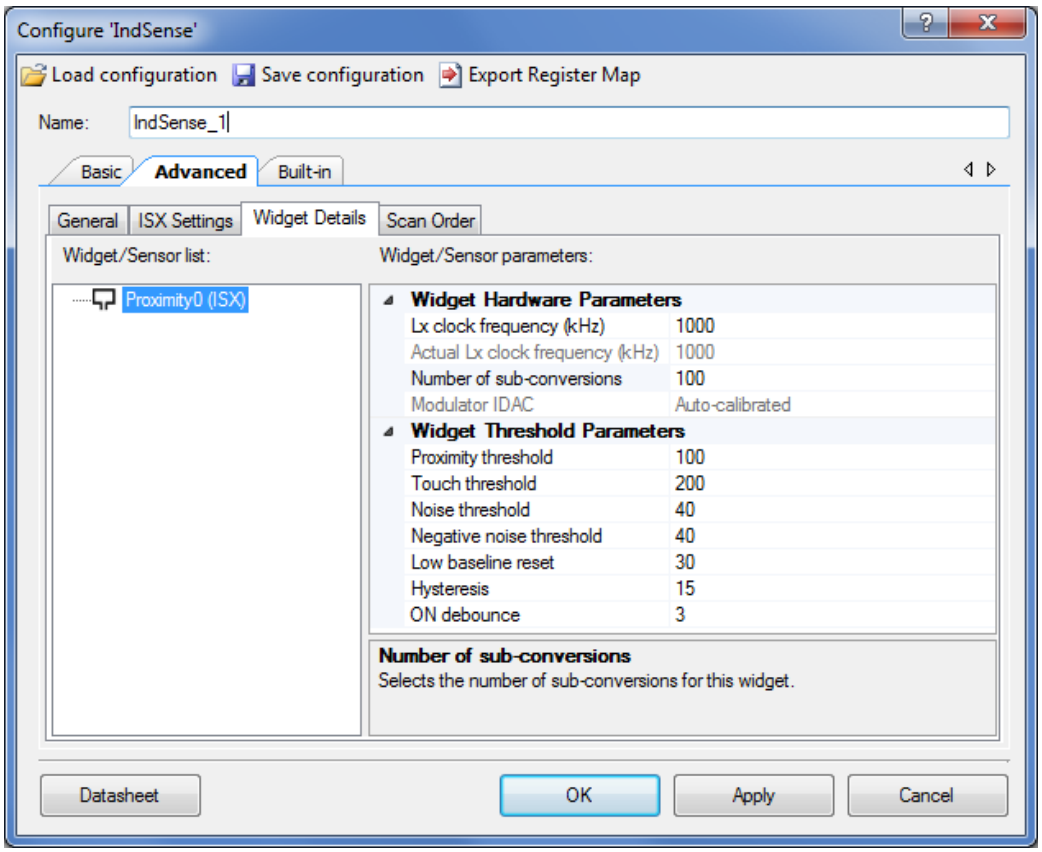
The parameters specific ISX sensing hardware is provided in this tab.



Name	Description
Modulator clock frequency	Selects the modulator clock frequency for the *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 in the design.



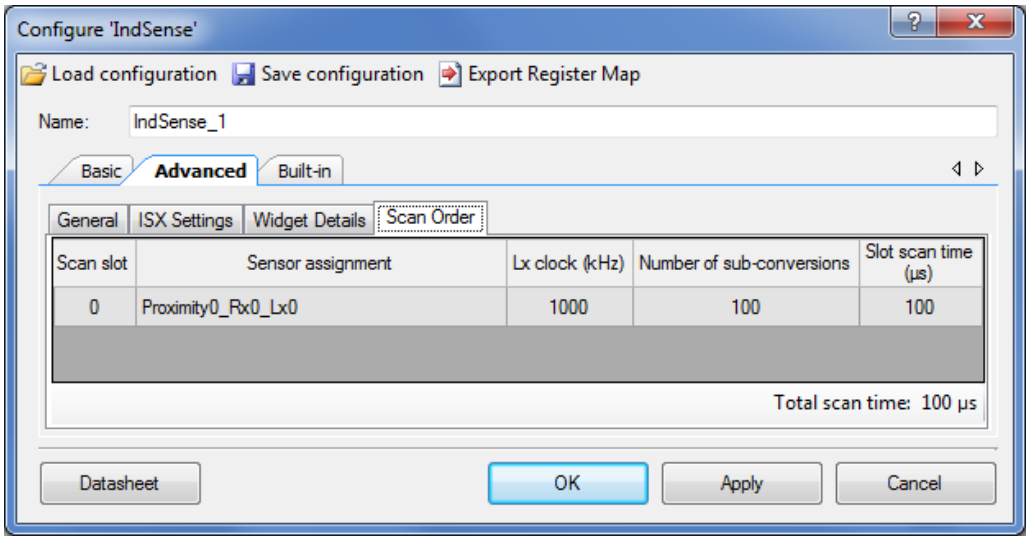
Name	Description
Widget Hardware Parameters	
Lx clock frequency	Sets the ISX Lx clock frequency. The minimum value is 45 kHz and maximum value is 3000 kHz. The LX clock frequency should be set to the resonant frequency of the LC tank. Determine the resonant frequency using the following equation: $$LxClk = f_0 = \frac{1}{2\pi}\sqrt{\frac{1}{(LC)} - \left(\frac{R}{L}\right)^2}$$ Where: ▪ L = Coil Inductance ▪ C = Parallel Capacitance of the LC tank ▪ R = AC series resistance of the coil at resonance. In cases where the AC resistance of the coil is unknown use the simplified expression: $$LxClk = f_0 = \frac{1}{2\pi\sqrt{(LC)}}$$ Enter Lx clock frequency value between minimum and maximum limits which matches the resonant frequency of LC tank, based on availability of the clock divider, the next valid lower value is selected by the Component, and the actual frequency is shown in the read-only label below the drop-down list. **Note** If the HFCLK or *Modulator clock frequency* is changed, the Component automatically recalculates the next closest Lx clock frequency. The Lx electrode is digital output, Refer to *Electrical Characteristics* section for Lx voltage levels details.
Number of sub-conversions	Selects the number of sub-conversions sensor. The number of sub-conversion should meet the following equation: $$N_{Sub} < \frac{2^{16} \bullet TxClk}{ModClk}$$ where, *ModClk* = ISX *Modulator clock frequency* *LxClk* = *Lx clock frequency* N_{Sub} = the value of this parameter.
IDAC Value	Set the IDAC value such that raw count is at about 70% of full scale value. The value of this parameter is automatically set when *Enable auto-calibration* is selected in the *ISX Settings* tab.

Name	Description
Widget Threshold Parameters	
Finger threshold	This parameter is available only for a button widget. 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 a Finger threshold parameter value to be equal to the 80% of the touch signal. The Finger threshold parameter is not available for the *Encoder* **Dial** is a widget consisting of two sensors. Each sensor detects the presence or absence of metal. Then, based on the pattern of sensors compared to the pattern of metal on a dial, detects rotation. ▪ **Button** is a widget consisting of one sensor. Each widget can detect the presence or absence (that is, only two states) of a metal object on the sensor. **Proximity** widget. Instead, Proximity has two thresholds: ▪ *Proximity threshold* ▪ *Touch threshold*
Proximity threshold Touch threshold	The finger threshold parameter is used along with the hysteresis parameter to determine the sensor state as follows: ▪ ON – Signal > (Proximity Threshold + Hysteresis) ▪ OFF – Signal ≤ (Proximity Threshold – Hysteresis). Note that “Signal” in the above equations refers to: Difference Count = Raw Count – Baseline. The proximity sensor supports two levels of detection: ▪ The proximity threshold to detect an approaching of a hand or finger ▪ The touch threshold to detect a finger touch on the sensor similarly to other *Widget type* sensors Note that for valid operation, the Proximity threshold should be higher than the Touch threshold. The threshold parameters such as *Hysteresis* and *ON debounce* are applicable to both detection levels.
Noise threshold	The noise threshold parameter sets the raw count limit. Raw count below the limit is considered as noise, when the raw count is above the Noise Threshold difference count is produced and the baseline is updated only if *Enable sensor auto-reset* is selected (In other words, the baseline remains constant as long as the raw count is above the baseline + noise threshold. This prevents the baseline from following the raw counts during a finger touch detection event). It is recommended to set the noise threshold parameter value to be equal to 2x noise in the raw count or the 40% of signal.

Name	Description
Negative noise threshold	The negative noise threshold parameter sets the raw count limit below which the baseline is not updated for the number of samples specified by the *Low baseline reset* parameter. The negative noise threshold ensures that the baseline does not fall low because of any high-amplitude repeated negative noise spikes on the raw count caused by different noise sources such as ESD events. It is recommended to set the negative noise threshold parameter value to be equal to the *Noise threshold* parameter value.
Low baseline reset	This parameter is used along with the *Negative noise threshold* parameter. It counts the number of abnormally low raw counts required to reset the baseline. If a finger is placed on the sensor during a device startup, the baseline gets initialized to the high raw count value at a startup. When the finger is removed, raw counts fall to a lower value. In this case, the baseline should track the low raw counts. The Low Baseline Reset parameter helps to handle this event. It resets the baseline to the low raw count value when the number of low samples reaches the low baseline-reset number. Note that in this case, once a finger is removed from the sensor, the sensor will not respond to finger touches for low baseline-reset time. The recommended value is 30 which works for most designs.
Hysteresis	The hysteresis parameter is used along with the *Proximity threshold* and *Touch threshold* to determine the sensor state. The hysteresis provides immunity against noisy transitions of the sensor state. See the description of *Proximity threshold* and *Touch threshold* parameter for details. The recommend value for the hysteresis is the 10% *Proximity threshold* and *Touch threshold*.
ON debounce	This parameter selects a number of consecutive IndSense 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. An ON status is reported only when the sensor is touched for a consecutive debounce number of samples. The recommended value for the Debounce parameter is 3 for reliable sensor status detection.

Scan Order Sub-tab

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



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 IndSense firmware library supports the following compilers:

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

In order to use the IAR Embedded Workbench, refer to:

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

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

PSoC Creator\psoc\content\CyComponentLibrary\CyComponentLibrary.cylib\CortexM0\IAR

By default, the instance name of the Component is “IndSense_1” for a first instance of the Component in a given design. It can be renamed to any unique text that follows the syntactic rules for identifiers and the instance name is prefixed to every API function, variable, and constant names. For readability, this section assumes “IndSense” as the instance name.

IndSense High-Level APIs

Description

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

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

Functions

- cystatus IndSense_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 IndSense_Stop(void)
  Stops the Component operation.
- cystatus IndSense_Resume(void)
  Resumes the Component operation if the IndSense_Stop() function was called previously.
- cystatus IndSense_ProcessAllWidgets(void)
  Performs full data processing of all enabled widgets.
- cystatus IndSense_ProcessWidget(uint32 widgetId)
  Performs full data processing of the specified widget if it is enabled.
- void IndSense_Sleep(void)
  Prepares the Component for deep sleep.
- void IndSense_Wakeup(void)
  Resumes the Component after deep sleep power mode.
- cystatus IndSense_SetupWidget(uint32 widgetId)
  Performs the initialization required to scan the specified widget.
- cystatus IndSense_Scan(void)
  Initiates scanning of all the sensors in the widget initialized by IndSense_SetupWidget(), if no scan is in progress.
- cystatus IndSense_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 IndSense_IsBusy(void)
  Returns the current status of the Component (Scan is completed or Scan is in progress).
- uint32 IndSense_IsAnyWidgetActive(void)
  Reports if any widget has detected a touch.
- uint32 IndSense_IsWidgetActive(uint32 widgetId)
  Reports if the specified widget detects a touch on any of its sensors.
- uint32 IndSense_IsSensorActive(uint32 widgetId, uint32 sensorId)
  Reports if the specified sensor in the widget detects a touch.

- uint32 IndSense_IsProximitySensorActive(uint32 widgetId, uint32 proxId)
  Reports the finger detection status of the specified proximity widget/sensor.
- uint32 IndSense_RunTuner(void)
  Establishes synchronized communication with the Tuner application.

Function Documentation

cystatus IndSense_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 IndSense_dsRam based on the user selection in the Component configuration wizard.
2. Configure the hardware to perform capacitive sensing.
3. Calibrate the sensors and find the optimal values for IDACs of each widget / sensor, if the Enable IDAC auto-calibration is enabled in the CSD Setting or CSX Setting tabs.
4. Perform scanning for all the sensors and initialize the baseline history.
5. If the firmware filters are enabled in the Advanced General tab, the filter histories are also initialized.

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

When the Component operation is stopped by the IndSense_Stop() function, the IndSense_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.

cystatus IndSense_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 IndSense_Resume() function or the Component can be reset by calling the IndSense_Start() function.

This function is called when no scanning is in progress. I.e. IndSense_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.

cystatus IndSense_Resume (void)

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

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

Returns:

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

cystatus IndSense_ProcessAllWidgets (void)

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

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

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

If the Self-test library is enabled, this function executes the baseline duplication test. Refer to IndSense_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:

- IndSense_IncrementGestureTimestamp().
- IndSense_SetGestureTimestamp().

Returns:

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

cystatus IndSense_ProcessWidget (uint32 *widgetId*)

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

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

If the Self-test library is enabled, this function executes the baseline duplication test. Refer to IndSense_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:

- IndSense_IncrementGestureTimestamp().
- IndSense_SetGestureTimestamp().

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID can be found in the IndSense Configuration header file defined as IndSense_<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.

void IndSense_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.

void IndSense_Wakeup (void)

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

cystatus IndSense_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 IndSense_ISXSetupWidgetExt() functions and not disconnected.

This function does not start sensor scanning, the IndSense_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 IndSense Configuration header file defined as IndSense_<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.

cystatus IndSense_Scan (void)

This function is called only after the IndSense_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 IndSense_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.

cystatus IndSense_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 IndSense_SetupWidget() and IndSense_Scan() functions are executed by these functions. The status of a sensor scan must be checked using the IndSense_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.

uint32 IndSense_IsBusy (void)

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

Returns:

Returns the current status of the Component:

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

uint32 IndSense_IsAnyWidgetActive (void)

This function reports if any widget has detected a touch or not by extracting information from the wdgtStatus registers (IndSense_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.

uint32 IndSense_IsWidgetActive (uint32 *widgetId*)

This function reports if the specified widget has detected a touch or not by extracting information from the wdgtStatus registers (IndSense_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 IndSense Configuration header file defined as IndSense_<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.

uint32 IndSense_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 (IndSense_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 IndSense_IsProximitySensorActive() function.

Parameters:

widgetId	Specifies the ID number of the widget. A macro for the widget ID can be found in the IndSense Configuration header file defined as IndSense_<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 IndSense Configuration header file defined as IndSense_<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.

uint32 IndSense_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 (IndSense_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 IndSense Configuration header file defined as IndSense_<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 IndSense Configuration header file defined as IndSense_<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.

uint32 IndSense_RunTuner (void)

This function is used to establish synchronized communication between the IndSense 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:

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

IndSense 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 ISX 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 IndSense_ProcessWidgetExt(uint32 widgetId, uint32 mode)
  Performs customized data processing on the selected widget.
- cystatus IndSense_ProcessSensorExt(uint32 widgetId, uint32 sensorId, uint32 mode)
  Performs customized data processing on the selected widget's sensor.
- cystatus IndSense_UpdateAllBaselines(void)
  Updates the baseline for all the sensors in all the widgets.
- cystatus IndSense_UpdateWidgetBaseline(uint32 widgetId)
  Updates the baselines for all the sensors in a widget specified by the input parameter.
- cystatus IndSense_UpdateSensorBaseline(uint32 widgetId, uint32 sensorId)
  Updates the baseline for a sensor in a widget specified by the input parameters.
- void IndSense_InitializeAllBaselines(void)
  Initializes (or re-initializes) the baselines of all the sensors of all the widgets.
- void IndSense_InitializeWidgetBaseline(uint32 widgetId)
  Initializes (or re-initializes) the baselines of all the sensors in a widget specified by the input parameter.
- void IndSense_InitializeSensorBaseline(uint32 widgetId, uint32 sensorId)
  Initializes (or re-initializes) the baseline of a sensor in a widget specified by the input parameters.
- void IndSense_InitializeAllFilters(void)
  Initializes (or re-initializes) the raw count filter history of all the sensors of all the widgets.
- void IndSense_InitializeWidgetFilter(uint32 widgetId)
  Initializes (or re-initializes) the raw count filter history of all the sensors in a widget specified by the input parameter.
- void IndSense_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 IndSense_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 IndSense_CalibrateAllWidgets(void)
  Calibrates the IDACs for all the widgets in the Component to the default target, this function detects the sensing method used by the widgets prior to calibration.
- void IndSense_ISXSetupWidget(uint32 widgetId)
  Performs hardware and firmware initialization required for scanning sensors in a specific widget using the ISX sensing method. The IndSense_ISXScan() function should be used to start scanning when using this function.

- void IndSense_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 IndSense_ISXScanExt() function should be called to initiate the scan when using this function.
- void IndSense_ISXScan(void)
  This function initiates the scan for sensors of the widget initialized by the IndSense_ISXSetupWidget() function.
- void IndSense_ISXScanExt(void)
  Starts the ISX conversion on the preconfigured sensor. The IndSense_ISXSetupWidgetExt() function should be used to setup a widget when using this function.
- void IndSense_ISXCalibrateWidget(uint32 widgetId, uint16 idacTarget)
  Calibrates the raw count values of all the sensors/nodes in an ISX widget.
- void IndSense_ISXConnectLx (IndSense_FLASH_IO_STRUCTconst *lxPtr)
  Connects a LX electrode to the ISX scanning hardware.
- void IndSense_ISXConnectRx (IndSense_FLASH_IO_STRUCTconst *rxPtr)
  Connects an RX electrode to the ISX scanning hardware.
- void IndSense_ISXDisconnectLx (IndSense_FLASH_IO_STRUCTconst *lxPtr)
  Disconnects a LX electrode from the ISX scanning hardware.
- void IndSense_ISXDisconnectRx (IndSense_FLASH_IO_STRUCTconst *rxPtr)
  Disconnects an RX electrode from the ISX scanning hardware.
- cystatus IndSense_GetParam(uint32 paramId, uint32 *value)
  Gets the specified parameter value from the Data Structure.
- cystatus IndSense_SetParam(uint32 paramId, uint32 value)
  Sets a new value for the specified parameter in the Data Structure.

Function Documentation

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

If the Self-test library is enabled, this function executes the baseline duplication test. Refer to IndSense_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:

- IndSense_IncrementGestureTimestamp().
- IndSense_SetGestureTimestamp().

Parameters:

widgetId	Specifies the ID number of the widget to be processed. A macro for the widget ID can be found in the IndSense Configuration header file defined as IndSense_<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] - IndSense_PROCESS_ALL - Execute all the tasks. 3. Bit [5] - IndSense_PROCESS_STATUS - Update the status (on/off, centroid position). 4. Bit [2] - IndSense_PROCESS_DIFFCOUNTS - Update the difference counts. 5. Bit [1] - IndSense_PROCESS_BASELINE - Update the baselines. 6. Bit [0] - IndSense_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.

cystatus IndSense_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 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 IndSense_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 IndSense Configuration header file defined as IndSense_<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 IndSense Configuration header file defined as IndSense_<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] - IndSense_PROCESS_ALL - Executes all the tasks. 3. Bit [2] - IndSense_PROCESS_DIFFCOUNTS - Updates the difference count. 4. Bit [1] - IndSense_PROCESS_BASELINE - Updates the baseline. 5. Bit [0] - IndSense_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.

cystatus IndSense_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 IndSense_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.

cystatus IndSense_UpdateWidgetBaseline (uint32 *widgetId*)

This function performs exactly the same tasks as IndSense_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 IndSense_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 IndSense Configuration header file defined as IndSense_<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.

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

This function performs exactly the same tasks as IndSense_UpdateAllBaselines() and IndSense_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 IndSense_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 IndSense Configuration header file defined as IndSense_<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 IndSense Configuration header file defined as IndSense_<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.

void IndSense_InitializeAllBaselines (void)

Initializes the baseline for all the sensors of all the widgets. Also, this function can be used to re-initialize baselines. IndSense_Start() calls this API as part of IndSense 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:

- IndSense_InitializeAllFilters().
- IndSense_InitializeWidgetFilter().

void IndSense_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:

- IndSense_InitializeAllFilters().
- IndSense_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 IndSense Configuration header file defined as IndSense_<WidgetName>_WDGT_ID.

void IndSense_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 IndSense Configuration header file defined as IndSense_<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 IndSense Configuration header file defined as IndSense_<WidgetName>_SNS<SensorNumber>_ID.

void IndSense_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. IndSense_Start() calls this API as part of IndSense operation initialization.

void IndSense_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 IndSense Configuration header file defined as IndSense_<WidgetName>_WDGT_ID.

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

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

The IndSense_TX_PIN and IndSense_RX_PIN states are not allowed if there are no CSX nor ISX widgets configured in the user's project.

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

Parameters:

widgetId	Specifies the ID number of the widget to change the pin state of the specified sensor. A macro for the widget ID can be found in the IndSense Configuration header file defined as IndSense_<WidgetName>_WDGT_ID.
sensorElement	Specifies the ID number of the sensor element within the widget to change its pin state. Macros for Rx and Tx IDs can be found in the IndSense Configuration header file defined as: • IndSense_<WidgetName>_RX<RXNumber>_ID • IndSense_<WidgetName>_TX<TXNumber>_ID.
state	Specifies the state of the sensor to be set: 1. IndSense_GROUND - The pin is connected to the ground. 2. IndSense_HIGHZ - The drive mode of the pin is set to High-Z Analog. 3. IndSense_TX_PIN - The TX or LX signal is routed to the sensor (only in CSX or ISX sensing method). 4. IndSense_RX_PIN - The pin is connected to the scanning bus (only in CSX or ISX sensing method).

cystatus IndSense_CalibrateWidget (uint32 *widgetId*)

This function performs exactly the same tasks as IndSense_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 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 IndSense Configuration header file defined as IndSense_<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.

cystatus IndSense_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 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.

void IndSense_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 IndSense_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., IndSense_IsBusy() returns a non-busy status.

This function is called by the IndSense_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 IndSense Configuration header file defined as IndSense_<WidgetName>_WDGT_ID.

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

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

This function should be called when no scanning in progress. I.e. IndSense_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 IndSense Configuration header file defined as IndSense_<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 IndSense Configuration header file defined as IndSense_<WidgetName>_SNS<SensorNumber>_ID.

void IndSense_ISXScan (void)

This function performs scanning of all the sensors in the widget configured by the IndSense_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 IndSense_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. IndSense_IsBusy() returns a non-busy status. The widget must be preconfigured by the IndSense_ISXSetupWidget() function if other widget was previously scanned or other type of scan functions were used.

void IndSense_ISXScanExt (void)

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

1. Sets a busy flag in the IndSense_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. IndSense_IsBusy() returns a non-busy status.

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

If disconnection of the sensors is required after calling IndSense_ISXScanExt(), the IndSense_ISXDisconnectLx() and IndSense_ISXDisconnectRx() APIs can be used.

void IndSense_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 IndSense Configuration header file defined as IndSense_<WidgetName>_WDGT_ID.
idacTarget	Specifies the calibration target in percentages of the maximum raw count.

void IndSense_ISXConnectLx (IndSense_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.

void IndSense_ISXConnectRx (IndSense_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.

void IndSense_ISXDisconnectLx (IndSense_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.

void IndSense_ISXDisconnectRx (IndSense_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.

cystatus IndSense_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 IndSense RegisterMap header file as IndSense_<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 IndSense RegisterMap header file defined as IndSense_<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.

cystatus IndSense_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 IndSense RegisterMap header file as IndSense_<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.

Some of the Data Structure registers is interdependent. This function just writes specified value into the desired register without other dependent registers update. A user is responsible for the dependent registers update. Repeated call of IndSense_Start() function performs register alignment.

Parameters:

paramId	Specifies the ID of parameter to set its value. A macro for the parameter ID can be found in the IndSense RegisterMap header file defined as IndSense_<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.

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).

IndSense Macro Callbacks

Macro Callback Function Name	Associated Macro	Description
IndSense_EntryCallback	IndSense_ENTRY_CALLBACK	Used at the beginning of the IndSense interrupt handler to perform additional application-specific actions
IndSense_ExitCallback	IndSense_EXIT_CALLBACK	Used at the end of the IndSense interrupt handler to perform additional application-specific actions
IndSense_StartSampleCallback(uint8 IndSense_widgetId, uint8 IndSense_sensorId)	IndSense_START_SAMPLE_CALLBACK	Used before each sensor scan triggering and deliver the current widget / sensor Id

Global Variables

Description

The section documents the IndSense component related global Variables.

The IndSense 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

- IndSense_RAM_STRUCT IndSense_dsRam

Variable Documentation

IndSense_RAM_STRUCT IndSense_dsRam

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

API Constants

Description

The section documents the IndSense component related API Constants.

Variables

- const IndSense_FLASH_STRUCT IndSense_dsFlash
- const IndSense_FLASH_IO_STRUCT IndSense_ioList[IndSense_TOTAL_ELECTRODES]

Variable Documentation

const IndSense_FLASH_STRUCT IndSense_dsFlash

Constant for the FLASH Data Structure

const IndSense_FLASH_IO_STRUCT IndSense_ioList[IndSense_TOTAL_ELECTRODES]

The array of the pointers to the electrode specific register.

Data Structure

Description

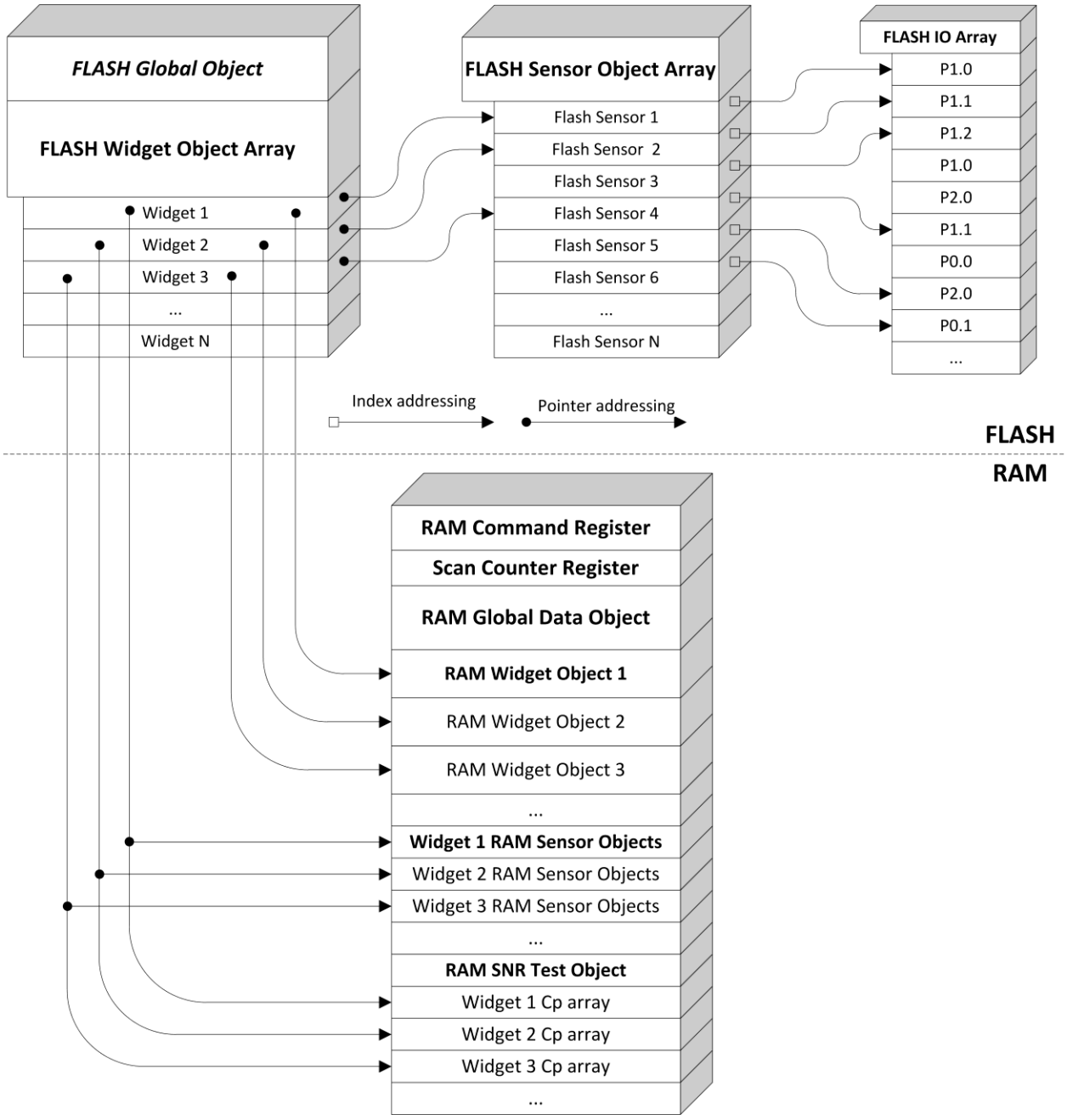
This section provides the list of structures/registers available in the component.

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 IndSense 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. IndSense 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 IndSense 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.

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

The IndSense 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 IndSense_RegisterMap header file list all registers of data structure with the following:

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

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

1. Direct Access

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

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

```
rawCount = IndSense_dsRam.snsList.button0[IndSense_BUTTON0_SNS2_ID].raw[0];
```

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

```
rawCount = IndSense_BUTTON0_SNS2_RAW0_VALUE;
```

2. Getter/Setter Access

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

```
cystatus IndSense_GetParam(uint32 paramId, uint32 *value)
cystatus IndSense_SetParam(uint32 paramId, uint32 value)
```

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

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

```
IndSense_GetParam(IndSense_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):

```
IndSense_SetParam(IndSense_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 IndSense data structure to be exposed to I2C master at primary slave address request once at initialization an application program:

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

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

```
rawCount = *(uint16 *)(I2C_buffer1Ptr + IndSense_BUTTON0_SNS2_RAW0_OFFSET);
```

The current example is applicable to 2-byte registers only. Depends on register size defined IndSense_RegisterMap header file by corresponding macros (IndSense_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 IndSense_RAM_WD_BASE_STRUCT
  Declares common widget RAM parameters.
- struct IndSense_RAM_WD_PROXIMITY_STRUCT
  Declares RAM parameters for the ISX Proximity.
- struct IndSense_RAM_WD_LIST_STRUCT
  Declares RAM structure with all defined widgets.
- struct IndSense_RAM_SNS_STRUCT
  Declares RAM structure for sensors.
- struct IndSense_RAM_SNS_LIST_STRUCT
  Declares RAM structure with all defined sensors.
- struct IndSense_RAM_STRUCT
  Declares the top-level RAM Data Structure.
- struct IndSense_FLASH_IO_STRUCT
  Declares the Flash IO object.
- struct IndSense_FLASH_SNS_STRUCT
  Declares the Flash Electrode object.
- struct IndSense_FLASH_SNS_LIST_STRUCT
  Declares the structure with all Flash electrode objects.
- struct IndSense_FLASH_WD_STRUCT
  Declares Flash widget object.
- struct IndSense_FLASH_STRUCT
  Declares top-level Flash Data Structure.
- struct IndSense_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 IndSense_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.
IndSense_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.
IndSense_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[IndSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widget.
uint16	snsClk	Sets Lx clock divider for ISX Widgets.
uint8	snsClkSource	Register for internal use

struct IndSense_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.

IndSense_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.
IndSense_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[IndSense_NUM_SCAN_FREQS]	Sets the current of the modulation IDAC for the widget.
uint16	snsClk	Sets Lx clock divider for ISX Widgets.
uint8	snsClkSource	Register for internal use
IndSense_THRESHOLD_TYPE	proxTouchTh	The proximity touch threshold.

struct IndSense_RAM_WD_LIST_STRUCT

Go to the top of the Data Structures section.

Data Fields:

IndSense_RAM_WD_PROXIMITY_STRUCT	proximity0	Proximity0 widget RAM structure

struct IndSense_RAM_SNS_STRUCT

Go to the top of the Data Structures section.

Data Fields:

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

struct IndSense_RAM_SNS_LIST_STRUCT

Go to the top of the Data Structures section.

Data Fields:

IndSense_RAM_SNS_STRUCT	proximity0[IndSense_PROXIMITY0_NUM_RX]	Proximity0 sensors RAM structures array

struct IndSense_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[IndSense_WDGT_STATUS_WORDS]	The bitmask that sets which Widgets are enabled and scanned, each bit corresponds to one widget.
uint32	wdgtStatus[IndSense_WDGT_STATUS_WORDS]	The bitmask that reports activated Widgets (widgets that detect a touch signal above the threshold), each bit corresponds to one widget.
IndSense_SNS_STS_TYPE	snsStatus[IndSense_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); 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	modIsxClk	
IndSense_RAM_WD_LIST_STRUCT	wdgtList	RAM Widget Objects.
IndSense_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[IndSense_NUM_SCAN_FREQS]	The sensor raw counts.

struct IndSense_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 IndSense_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 IndSense_FLASH_SNS_LIST_STRUCT

Go to the top of the Data Structures section.

Data Fields:

uint8	notUsed	No ganged sensors available

struct IndSense_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.
IndSense_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	numCols	For ISX Proximity Widgets, the number of sensors.
uint8	numRows	Unused.

struct IndSense_FLASH_STRUCT

Go to the top of the Data Structures section.

Data Fields:

IndSense_FLASH_WD_STRUCT	wdgtArray[IndSense_TOTAL_WIDGETS]	Array of flash widget objects

struct IndSense_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 (bytes)	SRAM (bytes)
ISX Component base configuration (1 sensor)	4430	120
Memory consumption for each additional sensor	+288	+32

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 auto-calibration* parameter is enabled.

IndSense Tuner

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

To make the tuner application work, a communication Component should be added to the project and the Component register map should be exposed to the tuner application.

It is possible to edit the parameters using the Tuner application and apply the new settings to the device using the **To Device** button.

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 IndSense_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 IndSense Design guide.

Tuner Quick Start

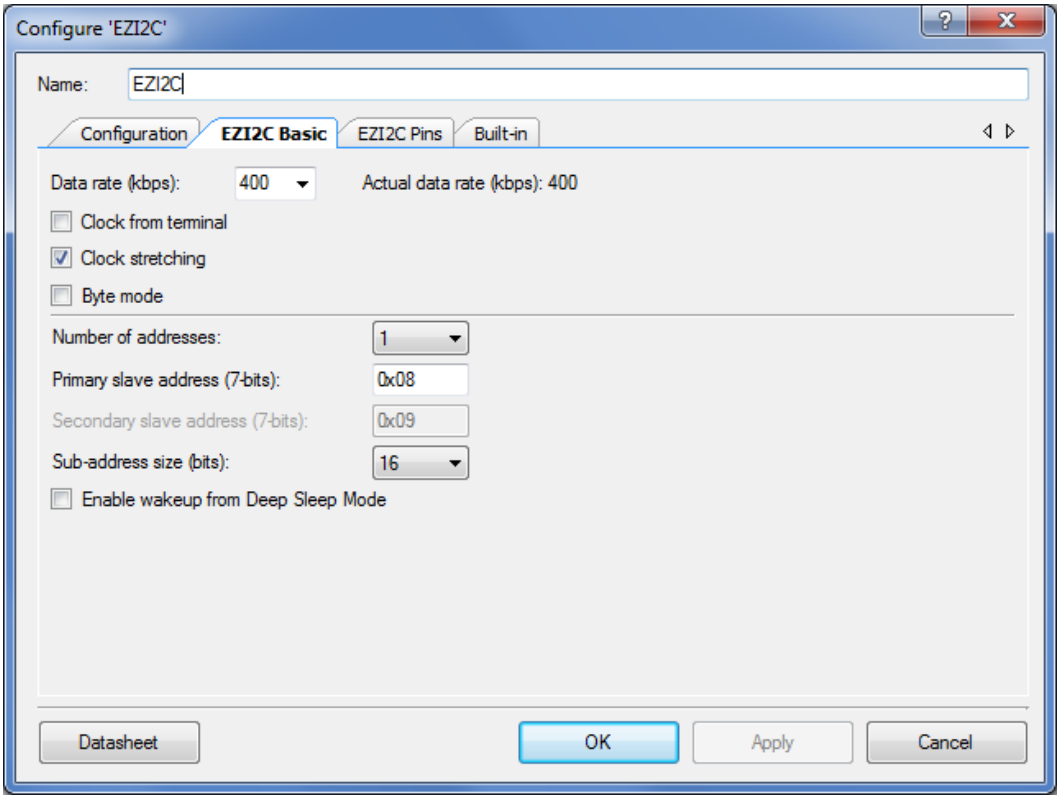
This section show how to set up IndSense tuner via an I^2C interface.

This section is continuation of application *Quick Start* section and assumes procedure documented in application *Quick Start* section is already followed.

Step-1: Place and Configure an EZI2C Component

Drag and drop the EZI2C Slave (SCB Mode) Component from the Component Catalog onto the schematic to add an I2C communication interface to the project. This I2C slave interface is required for transfer data from DUT to Tuner GUI to monitor Component parameters in real time.

Double-click on the EZI2C Component. On the EZI2C Basic tab, set the following parameters.



- Set a Component name (in this case: *EZI2C*).
- Set the Data rate (kbps) to 400
- Set the Primary slave address (7-bits) to 0x08
- Set the Sub-address size (bits) to 16

Press **OK** to save changes and close.

Step-2: Assign I2C Pins in Pin Editor

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

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

Step-3: Modify Application Code

Replace the *main.c* from the Step-3 in the *Quick Start* section with the following code:

```
#include <project.h>

int main()
{
    CyGlobalIntEnable;                          /* Enable global interrupts */

    EZI2C_Start();                              /* Start EZI2C Component */
    /*
    *  Set up communication and initialize data buffer to IndSense data structure
    *  to use Tuner application
    */
    EZI2C_EzI2CSetBuffer1(sizeof(IndSense_dsRam), sizeof(IndSense_dsRam),
                          (uint8 *)&IndSense_dsRam);

    IndSense_Start();                           /* Initialize Component */
    IndSense_ScanAllWidgets();                  /* Scan all widgets */

    for(;;)
    {
        /* Do this only when a scan is done */
        if(IndSense_NOT_BUSY == IndSense_IsBusy())
        {
            IndSense_ProcessAllWidgets();       /* Process all widgets */
            IndSense_RunTuner();                /* To sync with Tuner application */
            if (IndSense_IsAnyWidgetActive())   /* Scan result verification */
            {
                /* add custom tasks to execute when touch detected */
            }

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

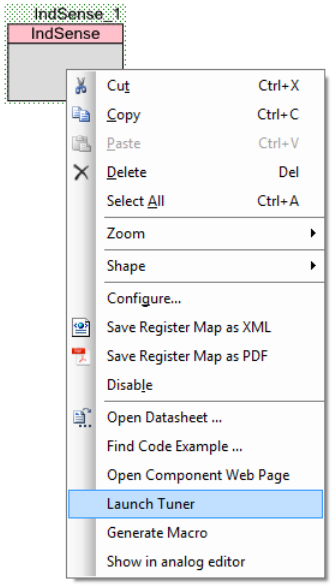
Step-4: Build Design and Program the device

Select ***Build <project name>*** from the ***Build*** menu and see the project build without errors.

Select ***Program*** from the ***Debug*** menu and program the device.

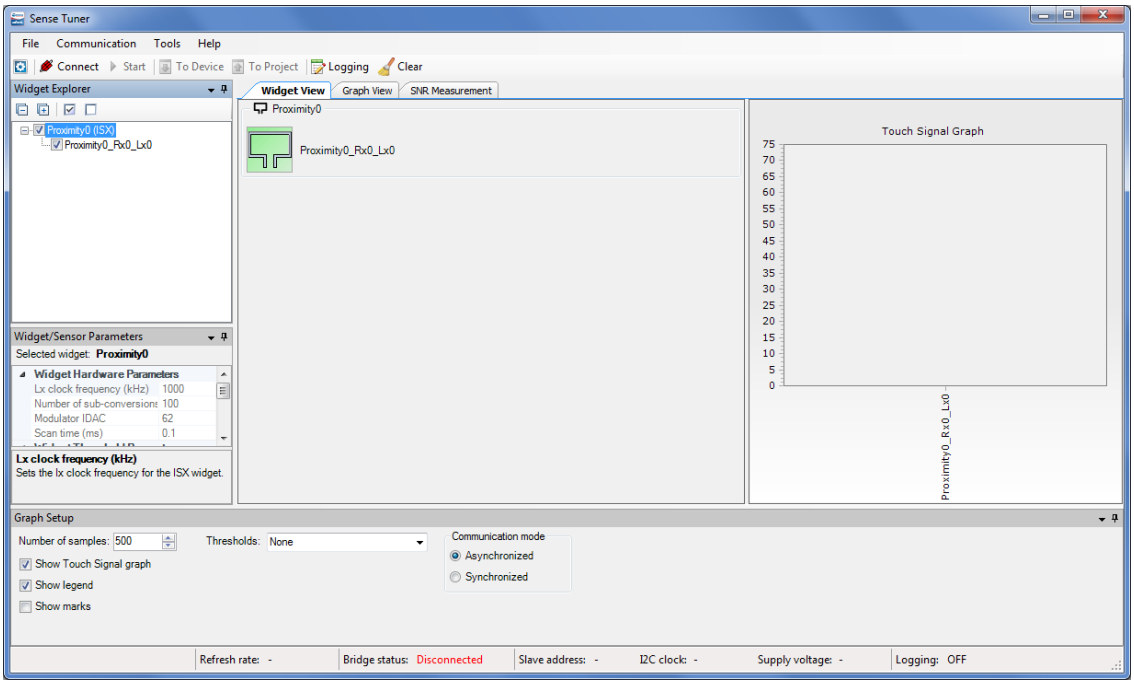
Step-5: Launch the Tuner Application

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



The *IndSense Tuner* application opens as shown below.

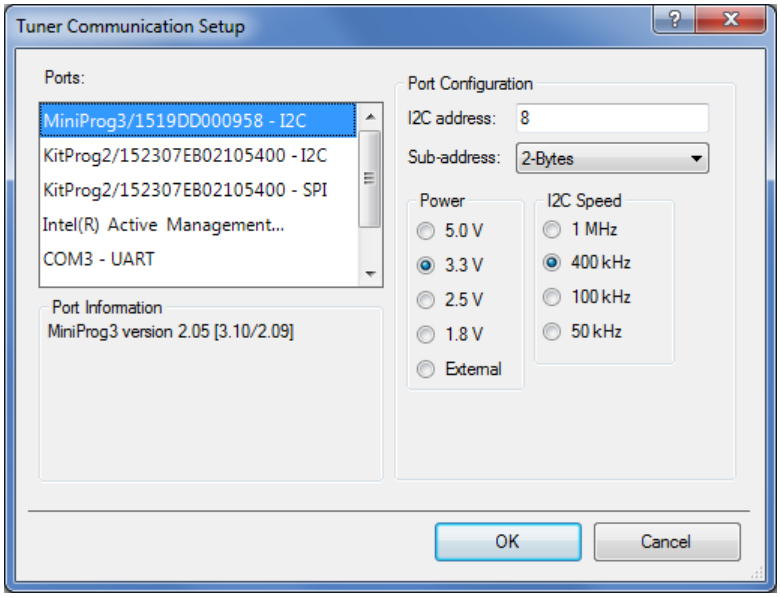
Note that the proximity sensor widget, called Proximity0, is automatically shown in the Widget View panel.



Step-6: Configure Communication Parameters

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

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



Select the appropriate I²C communication device KitProg (or MiniProg3) and set the following parameters:

- **I2C Address**: 8 (or the address set in EzI2C Component configuration wizard).
- **Sub-address**: 2 bytes.
- **I2C Speed**: 400 kHz (or speed set in Component configuration wizard).

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

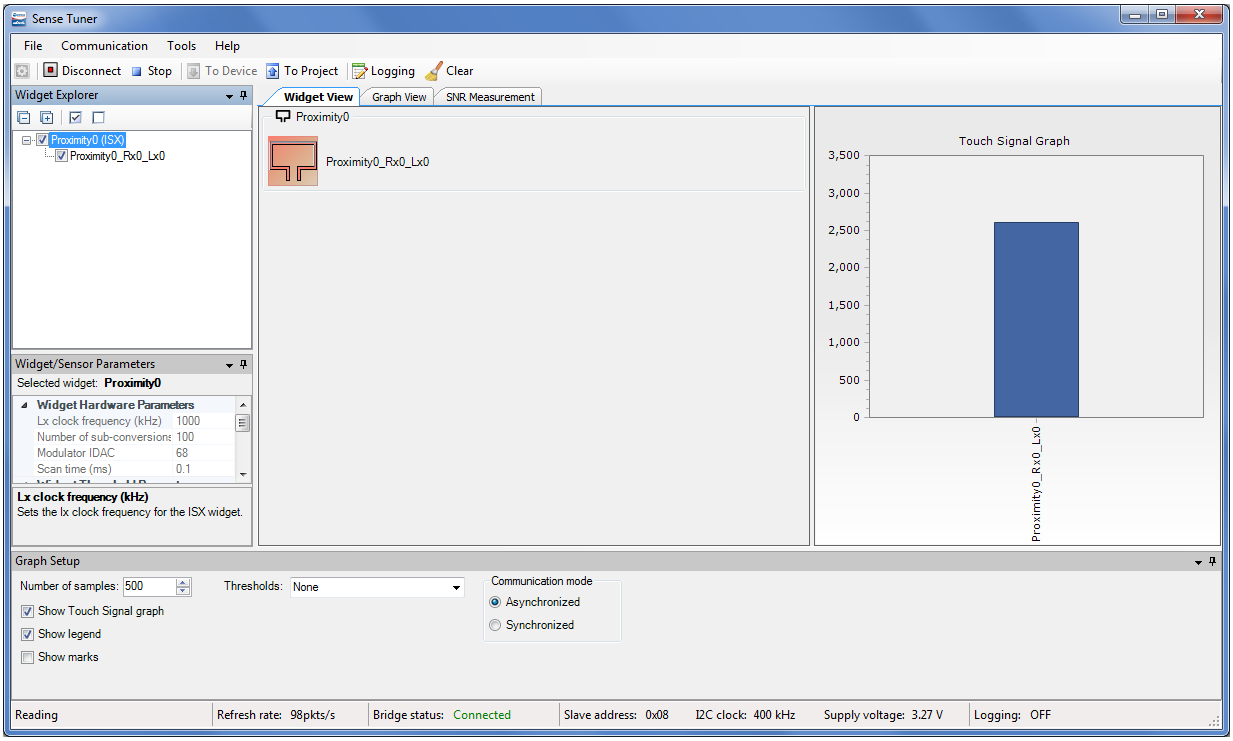
Step-7: Start Communication

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

Check the *Synchronized* control in *Graph Setup Pane*. This ensures that the Tuner only collects the data when IndSense is not scanning. Refer to *Graph Setup Pane* for details of synchronized operation.

The *Status bar* shows the communication bridge connection status and data refresh rate. You can see the status of the Proximity0 widget in in the *Widget View* and signals for the sensor in the *Graph View*. Touch the sensors on the kit to observe IndSense operation.

General Interface



The application consists of the following tabs:

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

Menus

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

- **File > Apply to Device (Ctrl + D)** – Commits the current values of the widget / sensor parameters to the device. This menu item becomes active if a value of any configuration parameter is changed from the Tuner UI (i.e. if the parameter values in the Tuner and the device are different). This is an indication that the changed parameter values need to be applied to the device.
- **File > Apply to Project (Ctrl + S)** – Commits the current values of widget / sensor parameters to the IndSense 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 that is actually saved depends on the currently selected view: it is *Touch Signal Graph* for *Widget View* (only when shown), a combined graph with Sensor Data, Sensor Signal and Status for Graph View, and SNR Raw counts graph for SNR Measurement View.
- **File > Exit (Alt+F4)** – Asks to save changes if there are any, and closes the Tuner. Changes are saved to the PSoC Creator project (merged back by the customizer).
- **Communication > Connect (F4)** – Connects to the device via a communication channel selected in the Tuner Communication Setup dialog. When the channel was not previously selected, dialog is to configure communication is shown.
- **Communication > Disconnect (Shift+F4)** – Closes the communication channel with the connected device.
- **Communication > Start (F5)** – Starts reading data from the device.

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

  - The invalid configuration of the communication channel (Slave address / Data rate / Sub-address size)
  - The invalid data buffer exposed via EZI2C (not *IndSense_dsRam*)
  - The latest customizer parameters modification was not programmed into device.
  - Edit performed in the customizer during tuning session: the Tuner needs to be closed and opened again after the customizer update.
  - The Tuner opened for the wrong project.
- **Communication > Stop (Shift+F5)** – Stops reading data from the device.
- **Tools > Tuner Communication Setup… (F10)** – Opens the configuration dialog to set up a communication channel with the device.
- **Tools > Options** – Opens the configuration dialog to setup different tuner preferences.
- **Help > Help Contents (F1)** – Opens the IndSense 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 various information related to the communication state between the Tuner and the device. This includes:

- **Current operation mode of tuner** – Either **Reading** (when tuner is reading from the device), **Writing** (when the write operation is in progress), or empty (idle – no operation performed).
- **Refresh rate** – Count of read samples performed per second. The count depends on multiple factors: the selected communication channel, communication speed, and amount of time needed to perform a single scan.
- **Bridge status** – Either **Connected**, when the communication channel is active, or d**isconnected** 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 IndSense project. The Widget nodes can be expanded/collapsed to show/hide widget's sensor nodes. It is possible to check/uncheck individual widgets and sensors. The Widget checked status affects its visibility on the *Widget View*, while the sensor checked status controls the visibility of the sensor raw count / baseline / signal / status graph series on the Graph View and signals on the *Touch Signal Graph* on the *Widget View*.

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

Note For the ISX widgets, the sensor tree displays individual nodes (Rx0_Lx0, Rx0_Lx1 …) as contrary to the customizer where the ISX electrodes are displayed (Rx0, Rx1 … Lx0, Lx1 ...).

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

Widget/Sensor Parameters Pane

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

- **Widget Hardware Parameters** – Any change to *Widget Hardware Parameters* requires hardware re-initialization, which can be performed only if the Tuner communicates with the device in the *Synchronized* mode.
- **Widget Threshold Parameters** – 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. When the *Enable auto-calibration* is enabled, the *IDAC Value* parameter is Read-only and displays the IDAC value as calibrated by the Component firmware. When auto-calibration is disabled, the IDAC value entered in the customizer is shown. If the Tuner operates in the *Synchronized* mode, it is possible to edit the value and apply it to the device.

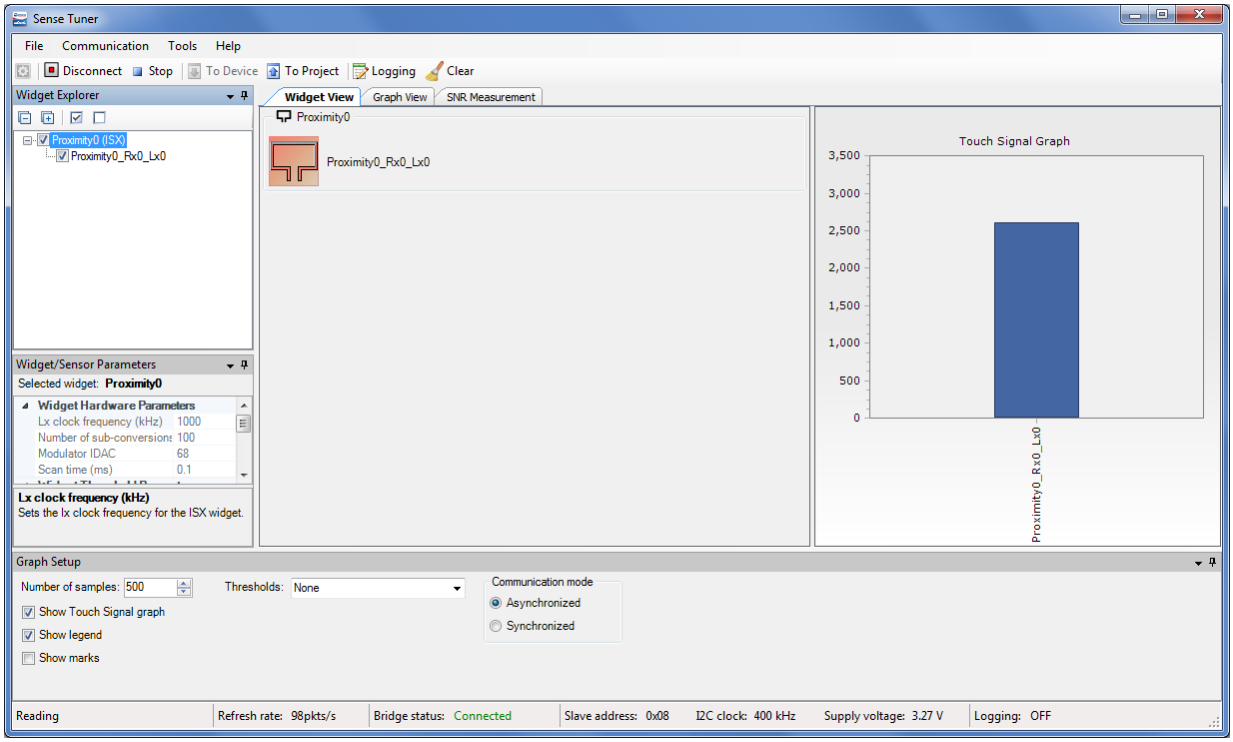
Graph Setup Pane

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

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

- **Show Touch Signal Graph** – Displays the graph window when checked.
- **Show legend** – Displays the sensor series descriptions (with names and colors) on graphs when checked (Sensor Data/Sensor Signal/Status graphs on a *Graph View* and *Touch Signal Graph* on a *Widget View*).
- **Show marks** – When checked, the sensor names are shown as marks over the signal bars on *Touch Signal Graph*.
- **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: IndSense_RunTuner(). When Synchronized Communication mode is selected, the IndSense 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 IndSense_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 IndSense Tuner and data processing happen asynchronously, the IndSense 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 IndSense Tuner is reading the data. As a result, the second sensor data is not processed.

Widget View



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

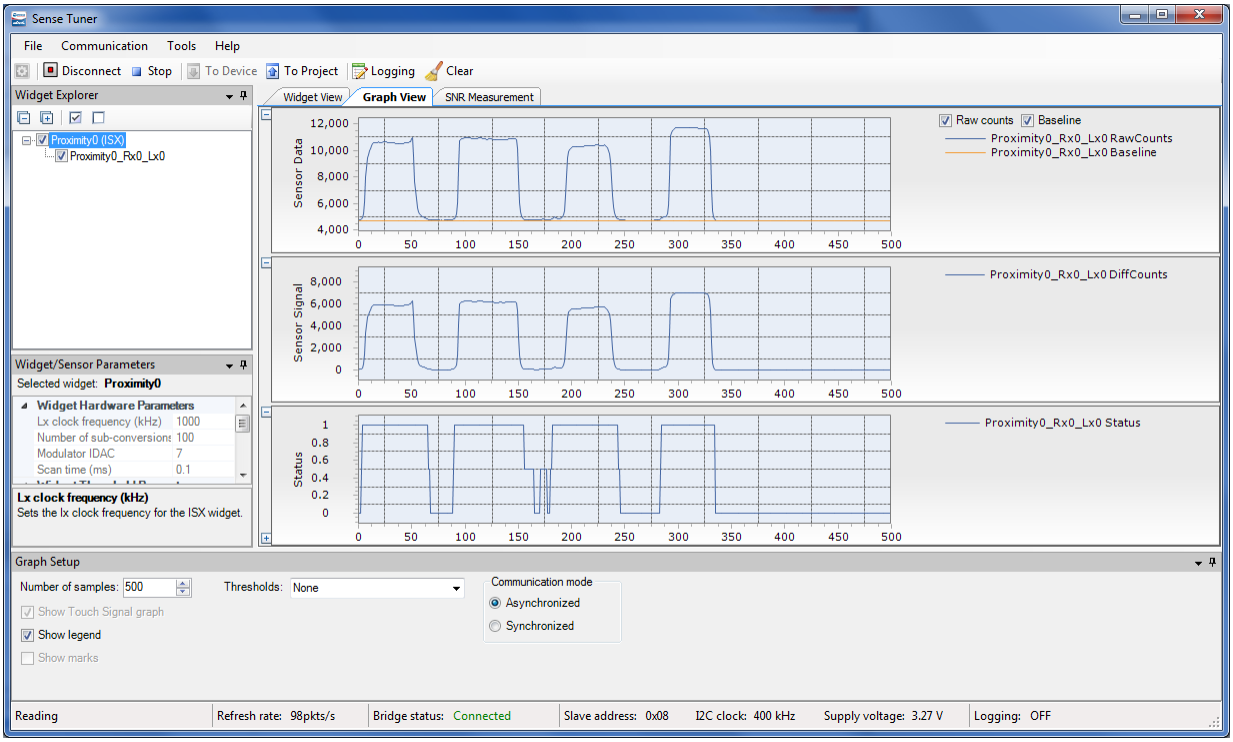
The widget sensors are highlighted red when the device reports their touch status as active.

Some additional features are available depending on the widget type:

Touch Signal Graph

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

Graph View



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

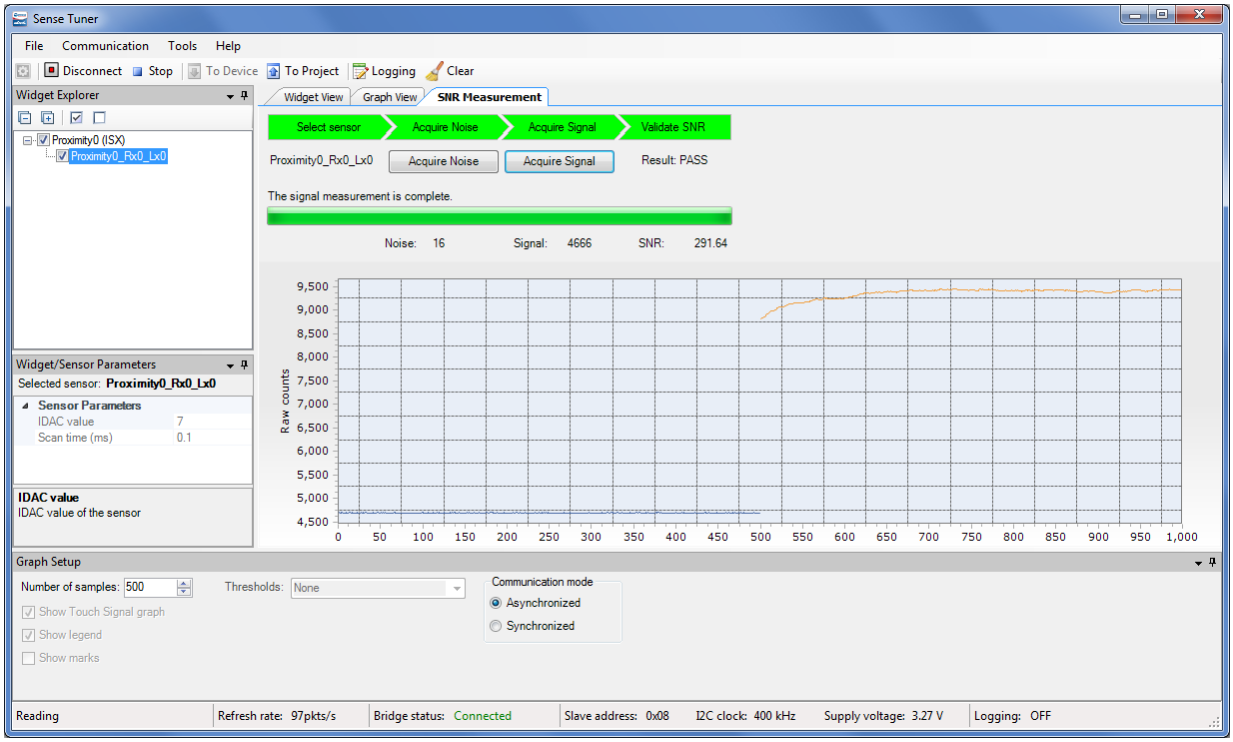
- **Sensor Data graph** – Displays raw counts and baseline.

  It is possible to select which series should be displayed with the checkboxes on the right:

  - Raw counts and baseline series
  - Raw counts only
  - Baseline only

- **Sensor Signal graph** – Displays a signal difference.
- **Status graph** – Displays the sensor status (Touch/No Touch). For proximity sensors, it also shows the proximity status (at 50% of the status axis) along with the touch status (at 100% of the axis).

SNR Measurement



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

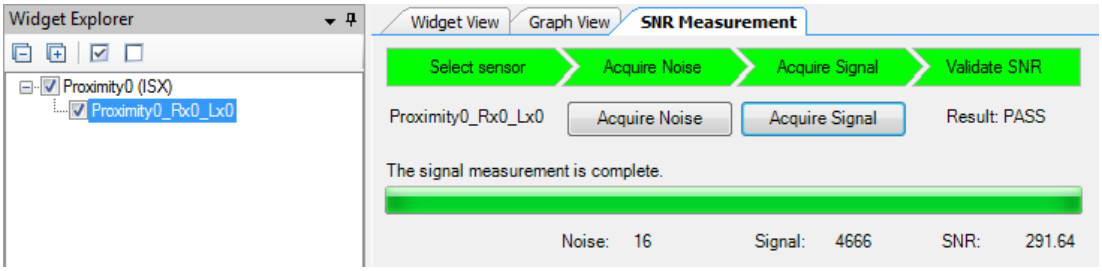
The Tab provides UI to acquire noise and signal samples separately and then calculates a SNR based on the captured data. The obtained value is then validated by a comparison with the required minimum (5 by default, can be configured in the *Tuner Configuration Options*).

Typical flow of SNR measurement

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

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

Description of SNR measurement GUI



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

- **Select sensor** is green when there is a sensor selected; gray otherwise.
- **Acquire noise** is green when noise samples are already collected for the selected sensor; gray otherwise.
- **Acquire signal** is green when signal samples are already collected for the selected sensor; gray otherwise.
- **Validate SNR** is green when both noise and signal samples are collected, and the SNR is above the valid limit; red when the SNR is below the valid limit, and gray when either noise or signal are not yet collected.
- Below the top bar, there are the following controls:
- **Sensor name** label (as selected in the *Widget Explorer Pane*) or None (if no sensor selected).
- **Acquire Noise** is a button disabled when the sensor is not selected or communication is not started. When the acquiring noise is in progress, the button can be used to abort the operation.
- **Acquire Signal** is a button disabled when the sensor is not selected, communication is not started, or noise samples are not yet collected for the selected sensor. When the acquiring signal is in progress, the button can be used to abort the operation.
- **Result** is a label that shows either “N/A” (when the SNR cannot be calculated due to noise/signal samples not yet collected), “PASS” (when SNR is above the required limit), or “FAIL” (when the SNR is below the required limit).

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

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

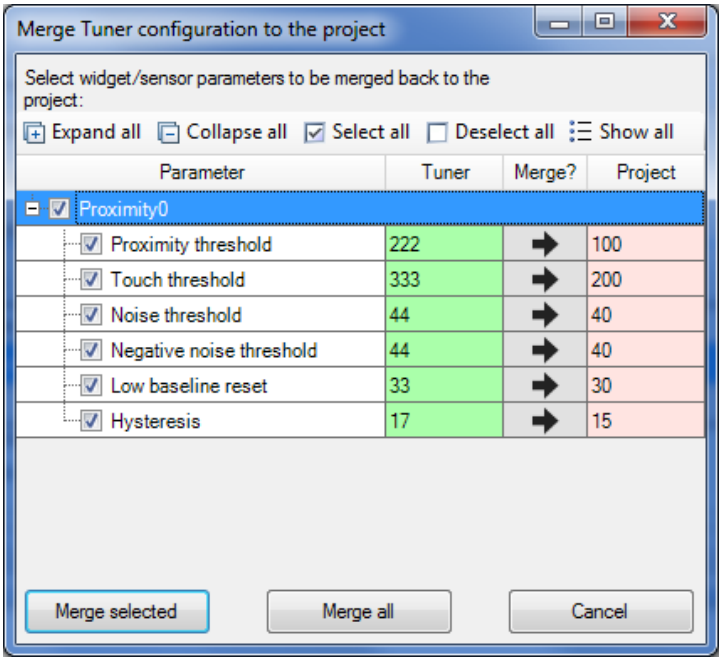
- **Noise** is a label which shows the noise average value calculated during the last noise measurement for the selected sensor, or “N/A” if no noise measurement is performed yet.
- **Signal** is a label that shows the signal average value calculated during the last signal measurement for the selected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

Procedure to Save Tuner Parameters

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

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

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



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

   Note Some parameters can be changed by the device at run-time when *Enable auto-calibration* feature is enabled.

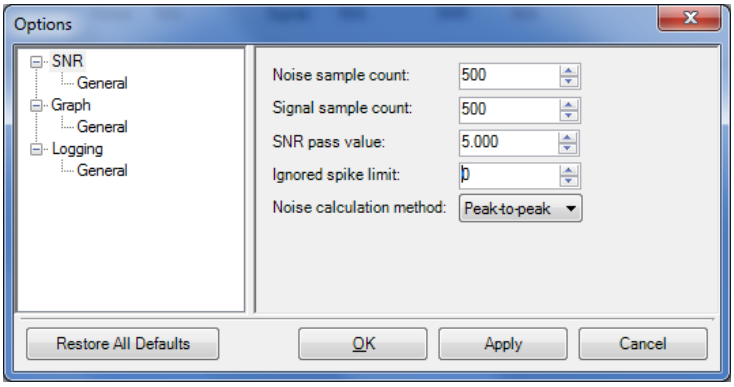
   The Tuner automatically picks up the changed parameters from a device. Clicking **To Project** merges these parameters to the Component and later they can be used as a starting point for manual calibration or tuning.

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

Tuner Configuration Options

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

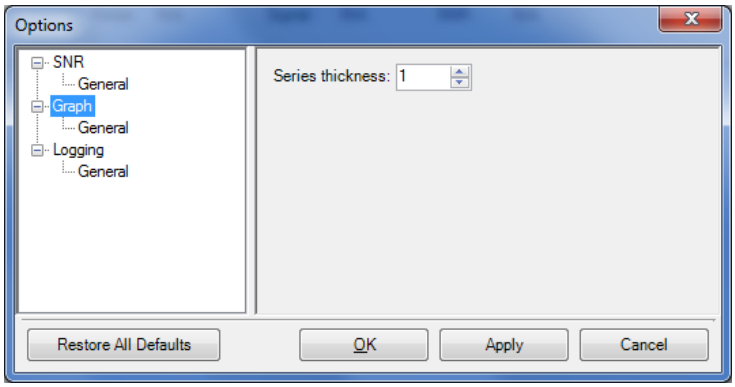
SNR Options



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

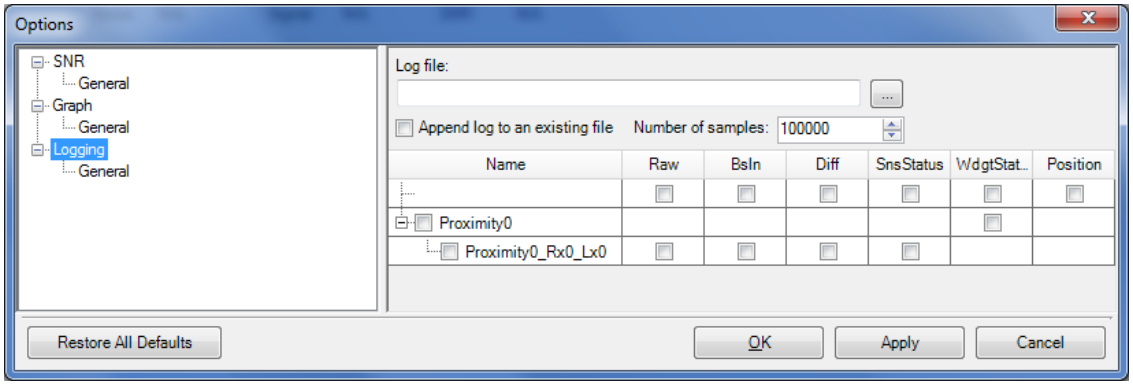
- **RMS** – Calculates noise as a root mean-square of all samples collected during the noise measurement.

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 IndSense 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.
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 IndSense Component has many different configurations. Some of them do not use specific function parameters. To avoid the complier's warning, the following code is used: (void)paramName.
16.7	A	A pointer parameter in a function prototype should be declared as the pointer to const if the pointer is not used to modify the addressed object.	Mostly all data processing for variety configuration, widgets and data types is required to pass the pointers as an argument. The architecture and design are intended for this casting.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	Pointers are used to allow many types of widgets and sensors. The architecture is designed to allow indexing a specific pointer.

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
18.4	R	Unions shall not be used.	There are two general cases in the code where this rule is violated. 1. IndSense_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 v1_10).

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

Electrical Characteristics

Specifications are valid for +25° C, VDD 3.3 V, Cc = 10pF, CintA = CintB = 470 pF except where noted.

Performance Characteristics (Preliminary)

Parameter	Condition / Details	Min	Typ	Max	Units
Maximum sample rate	SysClk = 48MHz, ModClk = 48MHz, IxClk = 3MHz, Sub Conversion = 20, Widget/Sensors scanned = 1, Firmware filters = Disabled, Auto-Calibration = Enabled All other components parameter = default.		16.4		ksps
Calibration time	SysClk = 48MHz, ModClk = 48MHz, IxClk = 1MHz, Sub Conversion = 50, Widget/Sensors scanned = 1, Firmware filters = Disabled, Auto-Calibration = Enabled. All other components parameter = default.		33		ms

Parameter	Condition / Details	Min	Typ	Max	Units
Sensor frequency	Supported LC resonant frequency. ISX parameters = tuned per CY design guide.	51		3,000	kHz
Inductance (L) range	Supported Inductance range. ISX parameter = tuned per CY design guide.	1		9380	uH
Resolution	Hardware resolution			16	bits
Maximum detectable distance	SysClk/MkdClk= 48MHz, Firmware filters = Disabled, Auto-Calibration = Enabled Coil =10 mm, Target Aluminium		4.5		mm
Maximum detectable distance	SysClk/MkdClk= 48MHz, Firmware filters = Disabled, Auto-Calibration = Enabled Coil =30 mm, Target Aluminium		19.5		mm
Average power consumption	LX Frequency= 3MHz, Coil diameter = 10 mm, Signal variation = 0.2% (Minimum detectable change), Report rate <= 100ms. All other components parameter = default.		0.61[1]		mA
Active power consumption	At maximum sample rate, no periodic deepsleep/sleep.		10.01[1]		mA

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.

1 Total chip current.

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
5.10	Added IndSense Button Widget. Added IndSense Encoder Dial Widget.	Expand IndSense functionality.
5.0	Added Data Logging feature in the IndSense Tuner. Added support for UART communication in the IndSense Tuner.	Component improvement.
4.10.a	Updated datasheet.	Updated screen captures, parameter descriptions, and API section to provide more details and clarification.
4.10	The initial version of new Component implementation.	

© Cypress Semiconductor Corporation, 2018. 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.