Features

- Supports inductive sensing up to frequencies of 3 MHz
- Supports auto-tuning algorithm for easy tuning and reliable operation
- Supports up to 16 simultaneous inductive sensing inputs
- Contains an integrated graphical user interface for tuning, testing, and debugging
- Sampling rate up to 10 ksp

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 MagSense 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 MagSense Inductive Sensing (MagSense) Component includes a configuration wizard to create and configure inductive sensing widgets, APIs to control the Component from the application firmware, and a MagSense 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.
- **MagSense 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 MagSense 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 Proximity interface.

In order to monitor performance of the sensor using the MagSense 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 MagSense-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 MagSense Component

Drag and drop the MagSense Component from the Component Catalog onto the design schematics to add the MagSense 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: MagSense 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:

```c
#include <project.h>

int main()
{
    CyGlobalIntEnable; /* Enable global interrupts */
    MagSense_Start(); /* Initialize Component */
    MagSense_ScanAllWidgets(); /* Scan all widgets */

    for(;;)
    {
        /* Do this only when a scan is done */
        if (MagSense_NOT_BUSY == MagSense_IsBusy())
        {
            MagSense_ProcessAllWidgets(); /* Process all widgets */
            if (MagSense_IsAnyWidgetActive()) /* Scan result verification */
            {
                /* add custom tasks to execute when touch detected */
            }
            MagSense_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 MagSense 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 MagSense 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.

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

<table>
<thead>
<tr>
<th>Name</th>
<th>I/O Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CintA</td>
<td>Analog</td>
<td>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.</td>
</tr>
<tr>
<td>CintB</td>
<td>Analog</td>
<td></td>
</tr>
<tr>
<td>Lx</td>
<td>Digital Output</td>
<td>Transmitter electrodes of ISX widgets. There is one Lx electrode for each sensor. <strong>Note</strong> To enable the full complement of 16 sensors, it may be necessary to change the <strong>Debug Select</strong> option in the Design-Wide Resources System Editor.</td>
</tr>
<tr>
<td>Rx</td>
<td>Analog</td>
<td>Receiver electrodes of ISX widgets. There is one Rx electrode for each sensor.</td>
</tr>
<tr>
<td>Rsv</td>
<td>N/A</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 MagSense 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 MagSense Component.
- **Save configuration** – Save the current Component configuration into a (XML) file.

- **Export Register Map** – The MagSense 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 *Proximity* widgets for the design.

![Configure MagSense](image)

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Sensing mode</th>
<th>Sensing element(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>🛠️</td>
<td>Proximity0</td>
<td>ISX (Inductive Sensing)</td>
<td>1, Rx, 1, Lx</td>
</tr>
</tbody>
</table>

**Sensor resources:**

- ISX Electrode(s): 2
- Pins required: 4
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Widget type</td>
<td>A widget is an interface that perform a specific user-interface functionality.</td>
</tr>
<tr>
<td></td>
<td>▪ <strong>Encoder Dial</strong> is a widget consisting of two sensors. Each sensor detects the presence or absence of metal, and based on the pattern of sensors compared to the pattern of metal on a dial, detects rotation.</td>
</tr>
<tr>
<td></td>
<td>▪ <strong>Button</strong> is a widget consisting of one sensor. Each widget can detect the presence or absence (i.e. only two states) of a metal object on the sensor.</td>
</tr>
<tr>
<td></td>
<td>▪ <strong>Proximity Sensor</strong> 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:</td>
</tr>
<tr>
<td></td>
<td>o <strong>Proximity threshold</strong>: To detect an approaching target</td>
</tr>
<tr>
<td></td>
<td>o <strong>Touch threshold</strong>: To detect a target touch on the sensor.</td>
</tr>
<tr>
<td>Widget name</td>
<td>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.</td>
</tr>
<tr>
<td>Sensing mode</td>
<td>Information: ISX sensing method is a Cypress patented method of performing inductive sensing measurements.</td>
</tr>
<tr>
<td>Widget Sensing element(s)</td>
<td>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.</td>
</tr>
<tr>
<td>Move up / Move down</td>
<td>Moves the selected widget up or down by one on the list. It defines the widget scanning order.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong>: Widget deleting may break a pin assignment and you will need to repair the assignment in the Pin Editor.</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes the selected widget from the list.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong>: Widget deleting may break a pin assignment and you will need to repair the assignment in the Pin Editor.</td>
</tr>
<tr>
<td>ISX electrodes</td>
<td>Information: Indicates the total number of electrodes (port pins) used by the ISX widgets, including the CintA and CintB capacitors.</td>
</tr>
<tr>
<td>Pins required</td>
<td>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.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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).

\[
Output = \frac{N}{K} \times \text{input} + \left(\frac{K-N}{K}\right) \times \text{previousOutput}
\]

Where:
- \(K\) is always 256,
- \(N\) is the IIR filter raw count coefficient selectable from 1 to 128 in the customizer.

A lower \(N\) (set in 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).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proximity widget baseline coefficient</td>
<td>Baseline IIR filter coefficient selection for sensors of proximity widgets. The valid range is: 1-255.</td>
</tr>
</tbody>
</table>

**General settings**

The general settings are applicable to the whole Component behavior.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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.

![Configuring ISX Sensing in PSoC Creator](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modulator clock frequency</td>
<td>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.</td>
</tr>
<tr>
<td>Enable auto-calibration</td>
<td>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.</td>
</tr>
</tbody>
</table>
Widget Details Sub-tab

This sub-tab contains parameters specific to each widget and sensor in the design.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Widget Hardware Parameters</strong></td>
<td></td>
</tr>
</tbody>
</table>
| 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Number of sub-conversions    | Selects the number of sub-conversions sensor. The number of sub-conversion should meet the following equation:  

\[
N_{Sub} < \frac{2^{16} \times LxClk}{ModClk}
\]

where,
- \(ModClk\) = ISX Modulator clock frequency
- \(LxClk\) = Lx clock frequency
- \(N_{Sub}\) = the value of this parameter.

**Note** If auto-calibration is enabled, the Component will vary the LxClk to try and find the best frequency within 10% of its current setting. It does not adjust \(N_{Sub}\), so it is recommended to choose \(N_{Sub}\) that is no larger than 90% of its maximum value.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDAC Value</td>
<td>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 <em>Enable auto-calibration</em> is selected in the ISX Settings tab.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Widget Threshold Parameters</strong></td>
<td></td>
</tr>
</tbody>
</table>
| Finger threshold     | This parameter is available only for 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 Proximity widget. Instead, Proximity has two thresholds:  
  - Proximity threshold  
  - Touch threshold |
| Proximity 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. |
| Touch threshold      | 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. |
| 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. |
### Name: 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.

### Name: 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**.

### Name: ON debounce

This parameter selects a number of consecutive MagSense 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.

<table>
<thead>
<tr>
<th>Scan slot</th>
<th>Sensor assignment</th>
<th>Lx clock (kHz)</th>
<th>Number of sub-conversions</th>
<th>Slot scan time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ProximityC_Fx0_Lx0</td>
<td>1000</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

**Total scan time:** 300 μs
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 MagSense 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 “MagSense_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 “MagSense” as the instance name.
MagSense High-Level APIs

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

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

Functions
- cystatus MagSense_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 MagSense_Stop(void)
  *Stops the Component operation.*
- cystatus MagSense_Resume(void)
  *Resumes the Component operation if the MagSense_Stop() function was called previously.*
- cystatus MagSense_ProcessAllWidgets(void)
  *Performs full data processing of all enabled widgets.*
- cystatus MagSense_ProcessWidget(uint32 widgetId)
  *Performs full data processing of the specified widget if it is enabled.*
- void MagSense_Sleep(void)
  *Prepares the Component for deep sleep.*
- void MagSense_Wakeup(void)
  *Resumes the Component after deep sleep power mode.*
- cystatus MagSense_SetupWidget(uint32 widgetId)
  *Performs the initialization required to scan the specified widget.*
- cystatus MagSense_Scan(void)
  *Initiates scanning of all the sensors in the widget initialized by MagSense_SetupWidget(), if no scan is in progress.*
- cystatus MagSense_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 MagSense_IsBusy(void)
  *Returns the current status of the Component (Scan is completed or Scan is in progress).*
- uint32 MagSense_IsAnyWidgetActive(void)
  *Reports if any widget has detected a touch.*
- uint32 MagSense_IsWidgetActive(uint32 widgetId)
  *Reports if the specified widget detects a touch on any of its sensors.*
- uint32 MagSense_IsSensorActive(uint32 widgetId, uint32 sensorId)
  *Reports if the specified sensor in the widget detects a touch.*
- uint32 MagSense_IsProximitySensorActive(uint32 widgetId, uint32 proxId)
Reports the finger detection status of the specified proximity widget/sensor.

- `uint32 MagSense_RunTuner(void)`
  Establishes synchronized communication with the Tuner application.

### Function Documentation

#### `cystatus MagSense_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 `MagSense_dsRam` based on the user selection in the Component configuration wizard.
2. Configure the hardware to perform 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 Mode's Setting tabs.
4. Perform scanning for all the sensors and initialize the baseline history.
5. If the firmware filters are enabled in the Advanced General tab, the filter histories are also initialized.

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

When the Component operation is stopped by the `MagSense_Stop()` function, the `MagSense_Start()` function repeats an initialization process including data structure initialization.

**Returns:**

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

Go to the top of the `MagSense High-Level APIs` section.

#### `cystatus MagSense_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 `MagSense_Resume()` function or the Component can be reset by calling the `MagSense_Start()` function.

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

**Returns:**

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

Go to the top of the `MagSense High-Level APIs` section.

#### `cystatus MagSense_Resume (void)`

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

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

**Returns:**

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

Go to the top of the `MagSense High-Level APIs` section.
cystatus MagSense_ProcessAllWidgets (void)
This function performs all data processes for all enabled widgets in the Component. The following tasks are
executed as part of processing all the widgets:
1. Apply raw count filters to the raw counts, if they are enabled in the customizer.
2. Update the thresholds if the SmartSense Full Auto-Tuning is enabled in the customizer.
3. Update the baselines and difference counts for all the sensors.
4. Update the sensor and widget status (on/off), update the centroid for the sliders and the X/Y position for the
touchpads.
This function is called by an application program only after all the enabled widgets (and sensors) in the
Component is scanned. Calling this function multiple times without sensor scanning causes unexpected
behavior.
The disabled widgets are not processed by this function. To disable/enable a widget, set the appropriate values
in the MagSense_WDGT_ENABLE<RegisterNumber>_PARAM_ID register using the MagSense_SetParam() function.
If the Ballistic multiplier filter is enabled the Timestamp must be updated before calling this function using the
MagSense_IncrementGestureTimestamp() function.
If the Self-test library is enabled, this function executes the baseline duplication test. Refer to
MagSense_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:
• MagSense_IncrementGestureTimestamp().
• MagSense_SetGestureTimestamp().
Returns:
Returns the status of the processing operation. If CYRET_SUCCESS is not received, the processing fails
and retries may be required.
Go to the top of the MagSense High-Level APIs section.

cystatus MagSense_ProcessWidget (uint32 widgetId)
This function performs exactly the same tasks as MagSense_ProcessAllWidgets(), but only for a specified
widget. This function can be used along with the MagSense_SetupWidget() and MagSense_Scan() functions to
scan and process data for a specific widget. This function is called only after all the sensors in the widgets ar
scanned. A disabled widget is not processed by this function.
A pipeline scan method (i.e. during scanning of a widget perform processing of the previously scanned widget)
can be implemented using this function and it may reduce the total execution time, increase the refresh rate and
decrease the average power consumption.
If the Ballistic multiplier filter is enabled the Timestamp must be updated before calling this function using the
MagSense_IncrementGestureTimestamp() function.
If the Self-test library is enabled, this function executes the baseline duplication test. Refer to
MagSense_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:
• MagSense_IncrementGestureTimestamp().
• MagSense_SetGestureTimestamp().
Parameters:

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

Go to the top of the MagSense High-Level APIs section.
Returns:
Returns the status of the widget processing:
- CYRET_SUCCESS  - The operation is successfully completed.
- CYRET_BAD_PARAM - The input parameter is invalid.
- CYRET_INVALID_STATE - The specified widget is disabled.
- CYRET_BAD_DATA  - The processing is failed.

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

```c
void MagSense_Sleep (void)
```

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

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

```c
void MagSense_Wakeup (void)
```

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

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

```c
cystatus MagSense_SetupWidget (uint32 widgetId)
```

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

This function does not start sensor scanning, the MagSense_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 MagSense Configuration header file defined as MagSense_<WidgetName>_WDGT_ID. |

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

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

```c
cystatus MagSense_Scan (void)
```

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

Returns:
Returns the status of the scan initiation operation:
- CYRET_SUCCESS  - Scanning is successfully started.
• CYRET_INVALID_STATE - The previous scanning is not completed and the hardware block is busy.
• CYRET_UNKNOWN - An unknown sensing method is used by the widget.

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

**cystatus MagSense_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 MagSense_SetupWidget() and MagSense_Scan() functions are executed by these functions. The status of a sensor scan must be checked using the MagSense_IsBusy() API prior to starting a next scan or setting up another widget.

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

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

**uint32 MagSense_IsBusy (void)**

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

**Returns:**
- Returns the current status of the Component:
  - MagSense_NOT_BUSY - No scan is in progress and a next scan can be initiated.
  - MagSense_SW_STS_BUSY - The previous scanning is not completed and the hardware block is busy.

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

**uint32 MagSense_IsAnyWidgetActive (void)**

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

**Returns:**
- Returns the touch detection status of all the widgets:
  - Zero - No touch is detected in all the widgets or sensors.
  - Non-zero - At least one widget or sensor detected a touch.

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

**uint32 MagSense_IsWidgetActive (uint32 widgetId)**

This function reports if the specified widget has detected a touch or not by extracting information from the wdgtStatus registers (MagSense_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 MagSense Configuration header file defined as MagSense_<WidgetName>_WDGT_ID.

**Returns:**
- Returns the touch detection status of the specified widgets:
  - Zero - No touch is detected in the specified widget or a wrong widgetId is specified.
• Non-zero if at least one sensor of the specified widget is active, i.e. a touch is detected.

Go to the top of the **MagSense High-Level APIs** section.

### uint32 MagSense_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 (MagSense_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 **MagSense_IsProximitySensorActive()** function.

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>widgetId</td>
<td>Specifies the ID number of the widget. A macro for the widget ID can be found in the MagSense Configuration header file defined as MagSense_&lt;WidgetName&gt;_WDGT_ID.</td>
</tr>
<tr>
<td>sensorId</td>
<td>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 MagSense Configuration header file defined as MagSense_&lt;WidgetName&gt;_SNS&lt;SensorNumber&gt;_ID.</td>
</tr>
</tbody>
</table>

**Returns:**

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

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

Go to the top of the **MagSense High-Level APIs** section.

### uint32 MagSense_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 (MagSense_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:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>widgetId</td>
<td>Specifies the ID number of the proximity widget. A macro for the widget ID can be found in the MagSense Configuration header file defined as MagSense_&lt;WidgetName&gt;_WDGT_ID.</td>
</tr>
<tr>
<td>proxId</td>
<td>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 MagSense Configuration header file defined as MagSense_&lt;WidgetName&gt;_SNS&lt;SensorNumber&gt;_ID.</td>
</tr>
</tbody>
</table>

**Returns:**

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

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

Go to the top of the **MagSense High-Level APIs** section.

### uint32 MagSense_RunTuner (void)

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

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

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

---

**MagSense Low-Level APIs**

**Description**

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

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

**Functions**

- `cystatus MagSense_ProcessWidgetExt(uint32 widgetId, uint32 mode)`
  Performs customized data processing on the selected widget.
- `cystatus MagSense_ProcessSensorExt(uint32 widgetId, uint32 sensorId, uint32 mode)`
  Performs customized data processing on the selected widget's sensor.
- `cystatus MagSense_UpdateAllBaselines(void)`
  Updates the baseline for all the sensors in all the widgets.
- `cystatus MagSense_UpdateWidgetItemBaseline(uint32 widgetId)`
  Updates the baselines for all the sensors in a widget specified by the input parameter.
- `cystatus MagSense_UpdateSensorBaseline(uint32 widgetId, uint32 sensorId)`
  Updates the baseline for a sensor in a widget specified by the input parameters.
- `void MagSense_InitializeAllBaselines(void)`
  Initializes (or re-initializes) the baselines of all the sensors of all the widgets.
- `void MagSense_InitializeWidgetItemBaseline(uint32 widgetId)`
Initializes (or re-initializes) the baselines of all the sensors in a widget specified by the input parameter.

- `void MagSense_InitializeSensorBaseline(uint32 widgetId, uint32 sensorId)`
  Initializes (or re-initializes) the baseline of a sensor in a widget specified by the input parameters.

- `void MagSense_InitializeAllFilters(void)`
  Initializes (or re-initializes) the baseline of a sensor in a widget specified by the input parameter.

- `void MagSense_InitializeAllFilters(uint32 widgetId)`
  Initializes (or re-initializes) the raw count filter history of all the sensors of all the widgets.

- `void MagSense_InitializeWidgetFilter(uint32 widgetId)`
  Initializes (or re-initializes) the raw count filter history of all the sensors in a widget specified by the input parameter.

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.

- `void MagSense_SetPinState(uint32 widgetId, uint32 sensorElement, uint32 state)`
  Sets the 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.

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

- `cystatus MagSense_SetupWidgetExt(uint32 widgetId, uint32 sensorId)`
  Performs extended initialization for the specified widget and also performs initialization required for a specific sensor in the widget. This function requires using the `MagSense_ScanExt()` function to initiate a scan.

- `cystatus MagSense_ScanExt(void)`
  Starts a conversion on the pre-configured sensor. This function requires using the `MagSense_SetupWidgetExt()` function to set up the a widget.

- `cystatus MagSense_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 MagSense_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.

The function updates the RAM data structure with the desired state of inactive electrodes for the specified operation mode. The state of pins is not changed in scope of this routine.

- `uint32_t MagSense_SetInactiveElectrodeState(MagSense_OPERATION_MODE_ENUM mode, uint32_t state)`
  The function updates the RAM data structure with the desired state of inactive electrodes for the specified operation mode. The state of pins is not changed in scope of this routine.

Performs hardware and firmware initialization required for scanning sensors in a specific widget using the ISX sensing method. The `MagSense_ISXScan()` function should be used to start scanning when using this function.

- `void MagSense_ISXSetupWidget(uint32 widgetId)`
  Performs hardware and firmware initialization required for scanning sensors in a specific widget using the ISX sensing method. The `MagSense_ISXScan()` function should be used to start scanning when using this function.

- `void MagSense_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 `MagSense_ISXScanExt()` function should be called to initiate the scan when using this function.

- `void MagSense_ISXScan(void)`
  This function initiates the scan for sensors of the widget initialized by the `MagSense_ISXSetupWidget()` function.

- `void MagSense_ISXScanExt(void)`
  This function initiates the scan for sensors of the widget initialized by the `MagSense_ISXSetupWidget()` function.

- `void MagSense_ISXSetupWidgetsExt(uint32 widgetId)`
  Starts the ISX conversion on the preconfigured sensor. The `MagSense_ISXSetupWidgetsExt()` function should be used to setup a widget when using this function.

- `void MagSense_ISXConnectLx(MagSense_FLASH_IO_STRUCT const *lxPtr)`
  Connects a LX electrode to the ISX scanning hardware.

- `void MagSense_ISXConnectRx(MagSense_FLASH_IO_STRUCT const *rxPtr)`
  Connects an RX electrode to the ISX scanning hardware.

- `void MagSense_ISXDisconnectLx(MagSense_FLASH_IO_STRUCT const *lxPtr)`
  Disconnects a LX electrode from the ISX scanning hardware.
void MagSense_ISXDisconnectRx (MagSense_FLASH_IO_STRUCT const *rxPtr)

Disconnects an RX electrode from the ISX scanning hardware.

- cystatus MagSense_GetParam(uint32 paramId, uint32 *value)

Gets the specified parameter value from the Data Structure.

- cystatus MagSense_SetParam(uint32 paramId, uint32 value)

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

Function Documentation

cystatus MagSense_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 Ballistic multiplier filter is enabled the Timestamp must be updated before calling this function using the MagSense_IncrementGestureTimestamp() function.

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

- MagSense_IncrementGestureTimestamp().
- MagSense_SetGestureTimestamp().

Parameters:

<table>
<thead>
<tr>
<th>widgetId</th>
<th>Specifies the ID number of the widget to be processed. A macro for the widget ID can be found in the MagSense Configuration header file defined as MagSense_&lt;WidgetName&gt;_WDGT_ID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>Specifies the type of widget processing to be executed for the specified widget:</td>
</tr>
<tr>
<td></td>
<td>2. Bits [5..0] - MagSense_PROCESS_ALL - Execute all the tasks.</td>
</tr>
<tr>
<td></td>
<td>6. Bit [0] - MagSense_PROCESS_FILTER - Run the firmware filters.</td>
</tr>
</tbody>
</table>

Returns:

Returns the status of the widget processing operation:

- CYRET_SUCCESS - The processing is successfully performed.
- CYRET_BAD_PARAM - The input parameter is invalid.
- CYRET_BAD_DATA - The processing is failed.
Go to the top of the MagSense Low-Level APIs section.

cystatus MagSense_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 MagSense_CheckBaselineDuplication() for details.

Parameters:

<table>
<thead>
<tr>
<th>widgetId</th>
<th>Specifies the ID number of the widget to process one of its sensors. A macro for the widget ID can be found in the MagSense Configuration header file defined as MagSense_&lt;WidgetName&gt;_WDGT_ID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>sensorId</td>
<td>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 MagSense Configuration header file defined as MagSense_&lt;WidgetName&gt;_SNS&lt;SensorNumber&gt;_ID.</td>
</tr>
</tbody>
</table>

Returns:

Returns the status of the sensor process operation:

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

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

cystatus MagSense_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 MagSense_CheckBaselineDuplication() for details.
Returns:
Returns the status of the update baseline operation of all the widgets:
- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_DATA - The baseline processing failed.

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

cystatus MagSense_UpdateWidgetBaseline (uint32 widgetId)
This function performs exactly the same tasks as MagSense_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 MagSense_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 MagSense Configuration header file defined as MagSense_<WidgetName>_WDGT_ID. |

Returns:
Returns the status of the specified widget update baseline operation:
- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_DATA - The baseline processing is failed.

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

cystatus MagSense_UpdateSensorBaseline (uint32 widgetId, uint32 sensorId)
This function performs exactly the same tasks as MagSense_UpdateAllBaselines() and MagSense_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 MagSense_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 MagSense Configuration header file defined as MagSense_<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 MagSense Configuration header file defined as MagSense_<WidgetName>_SNS<SensorNumber>_ID. |

Returns:
Returns the status of the specified sensor update baseline operation:
- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_DATA - The baseline processing failed.

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

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

- `MagSense_InitializeAllFilters()`.
- `MagSense_InitializeWidgetFilter()`.

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

```c
void MagSense_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:

- `MagSense_InitializeAllFilters()`.
- `MagSense_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 MagSense Configuration header file defined as `MagSense_<WidgetName>_WDGT_ID`.

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

```c
void MagSense_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 MagSense Configuration header file defined as `MagSense_<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 MagSense Configuration header file defined as `MagSense_<WidgetName>_SNS<SensorNumber>_ID`.

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

```c
void MagSense_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. `MagSense_Start()` calls this API as part of MagSense operation initialization.

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

```c
void MagSense_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 MagSense Configuration header file defined as `MagSense_<WidgetName>_WDGT_ID`.

Go to the top of the MagSense Low-Level APIs section.
void MagSense_SetPinState (uint32  widgetId, uint32  sensorElement, uint32  state)

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

This function accepts the MagSense_SHIELD and MagSenseSENSOR states as an input only if there is at least one CSD widget. Similarly, this function accepts the MagSense_TX_PIN and MagSense_RX_PIN states as an input only if there is at least one CSX widget in the project.

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

Parameters:

<table>
<thead>
<tr>
<th>widgetId</th>
<th>Specifies the ID of the widget to change the pin state of the specified sensor. A macro for the widget ID can be found in the MagSense Configuration header file defined as MagSense_&lt;WidgetName&gt;_WDGT_ID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>sensorElement</td>
<td>Specifies the ID of the sensor element within the widget to change its pin state. Macros for Rx and Tx IDs can be found in the MagSense Configuration header file defined as:</td>
</tr>
<tr>
<td></td>
<td>• MagSense_&lt;WidgetName&gt;_RX&lt;RXNumber&gt;_ID</td>
</tr>
<tr>
<td></td>
<td>• MagSense_&lt;WidgetName&gt;_TX&lt;TXNumber&gt;_ID</td>
</tr>
<tr>
<td>state</td>
<td>Specifies the state of the sensor to be set:</td>
</tr>
<tr>
<td></td>
<td>1. MagSense_GROUND - The pin is connected to the ground.</td>
</tr>
<tr>
<td></td>
<td>2. MagSense_HIGHZ - The drive mode of the pin is set to High-Z Analog.</td>
</tr>
<tr>
<td></td>
<td>3. MagSense_SHIELD - The shield signal is routed to the pin (available only if CSD sensing method with shield electrode is enabled).</td>
</tr>
<tr>
<td></td>
<td>4. MagSenseSENSOR - The pin is connected to the scanning bus (available only if CSD sensing method is enabled).</td>
</tr>
<tr>
<td></td>
<td>5. MagSense_TX_PIN - The Tx or Lx signal is routed to the sensor (available only if CSX or ISX sensing method is enabled).</td>
</tr>
<tr>
<td></td>
<td>6. MagSense_RX_PIN - The pin is connected to the scanning bus (available only if CSX or ISX sensing method is enabled).</td>
</tr>
</tbody>
</table>

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

cystatus MagSense_SetupWidgetExt (uint32  widgetId, uint32  sensorId)

This function does the same as MagSense_SetupWidget() and also does the following tasks:
1. Connects the first sensor of the widget.
2. Configures the CSD HW block to perform a scan of the specified sensor.

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

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

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

Parameters:

<table>
<thead>
<tr>
<th>widgetId</th>
<th>Specifies the ID number of the widget to perform hardware and</th>
</tr>
</thead>
</table>
firmware initialization required for scanning the specific sensor in the specific widget. A macro for the widget ID can be found in the MagSense Configuration header file defined as MagSense_<WidgetName>_WDGT_ID.

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

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

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

cystatus MagSense_ScanExt (void)
This function performs single scanning of one sensor in the widget configured by the MagSense_SetupWidgetExt() function.

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

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

If disconnection of the sensors is required after calling MagSense_ScanExt(), the MagSense_CSDDisconnectSns() or MagSense_CSXDisconnectTx() or MagSense_CSXDisconnectRx() functions can be used.

Returns:
Returns the status of the scan initiation operation:
- CYRET_SUCCESS - Scanning is successfully started.
- CYRET_INVALID_STATE - The previous scanning is not completed and the hardware block is busy.
- CYRET_UNKNOWN - An unknown sensing method is used by the widget.

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

cystatus MagSense_CalibrateWidget (uint32 widgetId)
This function performs exactly the same tasks as MagSense_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 MagSense Configuration header file defined as MagSense_<WidgetName>_WDGT_ID.
Returns:
Returns the status of the specified widget calibration:
- CYRET_SUCCESS - The operation is successfully completed.
- CYRET_BAD_PARAM - The input parameter is invalid.
- CYRET_BAD_DATA - The calibration failed and the Component may not operate as expected.

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


cystatus MagSense_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.

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

uint32_t MagSense_SetInactiveElectrodeState (MagSense_OPERATION_MODE_ENUM mode, uint32_t state)

The function updates the following registers of RAM data structure:
- MagSense_SCAN_CSD_ISC_VALUE - Connection of inactive CSD and CSX electrodes during the regular CSD scan. By default, this register is initialized with the value of Inactive Sensor Connection combobox on the CSD Settings tab. The MagSense_SCAN_CSD_E value should be used as the Mode parameter to update this register.
- MagSense_SCAN_CSX_ISC_VALUE - Connection of inactive CSD, CSX and the dedicated Shield electrodes during the regular CSX scan. By default, this register is initialized with the value of Inactive Sensor Connection combobox on the CSX Settings tab. The MagSense_SCAN_CSX_E value should be used as the Mode parameter to update this register.
- MagSense_BIST_CSD_SNS_CAP_ISC_VALUE - Connection of inactive CSD and CSX electrodes during measurement of CSD electrodes capacitance. This register is initialized with the MagSense_SNS_CONNECTION_GROUND value by default. The MagSense_BIST_CSD_SNS_CAP_E value should be used as the Mode parameter to update this register.
- MagSense_BIST_CSX_SNS_CAP_ISC_VALUE - Connection of inactive CSD, CSX and the dedicated Shield electrodes during measurement of CSX electrodes (Tx and Rx) capacitance. This register is initialized with the MagSense_SNS_CONNECTION_GROUND value by default. The MagSense_BIST_CSX_SNS_CAP_E value should be used as the Mode parameter to update this register.
- MagSense_BIST_CSD_SH_CAP_ISC_VALUE - Connection of inactive CSD and CSX electrodes measurement of dedicated Shield electrodes capacitance. This register is initialized with the MagSense_SNS_CONNECTION_GROUND value by default. The MagSense_BIST_CSD_SH_CAP_E value should be used as the Mode parameter to update this register.

Parameters:

<table>
<thead>
<tr>
<th>mode</th>
<th>Operation mode, the state of inactive sensors should be configured for.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MagSense_SCAN_CSD_E</td>
<td>Regular CSD scan.</td>
</tr>
<tr>
<td>MagSense_SCAN_CSX_E</td>
<td>Regular CSX scan.</td>
</tr>
<tr>
<td>MagSense_BIST_CSD_SNS_CAP_E</td>
<td>Measurement of the CSD sensor capacitance.</td>
</tr>
<tr>
<td>MagSense_BIST_CSX_SNS_CAP_E</td>
<td>Measurement of the CSX electrode capacitance.</td>
</tr>
</tbody>
</table>

 Parameter Description:

- mode: Operation mode, the state of inactive sensors should be configured for. This parameter can take the following values:
  - MagSense_SCAN_CSD_E - Regular CSD scan.
  - MagSense_SCAN_CSX_E - Regular CSX scan.
  - MagSense_BIST_CSD_SNS_CAP_E - Measurement of the CSD sensor capacitance.
  - MagSense_BIST_CSX_SNS_CAP_E - Measurement of the CSX electrode capacitance.
• **MagSense_BIST_CSD_SH_CAP_E** - Measurement of the dedicated CSD Shield electrode capacitance.

<table>
<thead>
<tr>
<th>state</th>
<th>The desired state of inactive sensors. This parameter can take the following values:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• <strong>MagSense_SNS_CONNECTION_GROUND</strong> - Inactive sensors are connected to the ground.</td>
</tr>
<tr>
<td></td>
<td>• <strong>MagSense_SNS_CONNECTION_HIGHZ</strong> - Inactive sensors are floating (not connected to GND or Shield).</td>
</tr>
<tr>
<td></td>
<td>• <strong>MagSense_SNS_CONNECTION_SHIELD</strong> - Inactive sensors are connected to the shield. This option is available only if the Enable shield electrode check box is set. At least one dedicated shield electrode is required to use the <strong>MagSense_SNS_CONNECTION_SHIELD</strong> option for the <strong>MagSense_BIST_CSD_SH_CAP_E</strong> operation mode.</td>
</tr>
</tbody>
</table>

**Returns:**

Returns the status of the operation:

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

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

**void MagSense_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 **MagSense_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., **MagSense_IsBusy()** returns a non-busy status.

This function is called by the **MagSense_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 MagSense Configuration header file defined as **MagSense_<WidgetName>_WDGT_ID**. |

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

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

This function does the same tasks as **MagSense_ISXSetupWidget()** and also connects a sensor in the widget for scanning. Once this function is called to initialize a widget and a sensor, the **MagSense_ISXScanExt()** function should be called to scan the sensor.
This function should be called when no scanning in progress. I.e. `MagSense_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:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>widgetId</code></td>
<td>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 MagSense Configuration header file defined as <code>MagSense_&lt;WidgetName&gt;_WDGT_ID</code>.</td>
</tr>
<tr>
<td><code>snsIndex</code></td>
<td>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 MagSense Configuration header file defined as <code>MagSense_&lt;WidgetName&gt;_SNS&lt;SensorNumber&gt;_ID</code>.</td>
</tr>
</tbody>
</table>

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

### void MagSense_ISXScan (void)

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

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

### void MagSense_ISXScanExt (void)

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

1. Sets a busy flag in the MagSense_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. `MagSense_IsBusy()` returns a non-busy status.

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

If disconnection of the sensors is required after calling `MagSense_ISXScanExt()`, the `MagSense_ISXDisconnectLx()` and `MagSense_ISXDisconnectRx()` APIs can be used.
void MagSense_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 MagSense Configuration header file defined as MagSense_<WidgetName>_WDGT_ID.
| idacTarget | Specifies the calibration target in percentages of the maximum raw count.

void MagSense_ISXConnectLx (MagSense_FLASH_IO_STRUCT const * 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 MagSense_ISXConnectRx (MagSense_FLASH_IO_STRUCT const * 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 MagSense_ISXDisconnectLx (MagSense_FLASH_IO_STRUCT const * lxPtr)

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

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

Parameters:

| lxPtr | Specifies the pointer to the FLASH_IO_STRUCT object belonging to a Lx pin sensor which should be disconnected from the sensing block.

Go to the top of the MagSense Low-Level APIs section.
void MagSense_ISXDisconnectRx (MagSense_FLASH_IO_STRUCT const * rxPtr)

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

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

Parameters:

| rxPtr | Specifies the pointer to the FLASH_IO_STRUCT object belonging to a Rx pin sensor which should be disconnected from the sensing block. |

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

cystatus MagSense_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 MagSense RegisterMap header file as MagSense_<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. [ TTFWCCCC 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 MagSense RegisterMap header file defined as MagSense_<ParameterName>_PARAM_ID. |
| value | The pointer to a variable to be updated with the obtained value. |

Returns:

Returns the status of the operation:

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

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

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

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

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

Parameters:

<table>
<thead>
<tr>
<th>paramId</th>
<th>Specifies the ID of parameter to set its value. A macro for the parameter ID can be found in the MagSense RegisterMap header file defined as MagSense_&lt;ParameterName&gt;_PARAM_ID.</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Specifies the new parameter's value.</td>
</tr>
</tbody>
</table>

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).

MagSense Macro Callbacks

<table>
<thead>
<tr>
<th>Macro Callback</th>
<th>Associated Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MagSense_EntryCallback</td>
<td>MagSense_ENTRY_CALLBACK</td>
<td>Used at the beginning of the MagSense interrupt handler to perform additional application-specific actions</td>
</tr>
<tr>
<td>MagSense.ExitCallback</td>
<td>MagSense_EXIT_CALLBACK</td>
<td>Used at the end of the MagSense interrupt handler to</td>
</tr>
</tbody>
</table>
Macro Callback Function Name | Associated Macro | Description
--- | --- | ---
MagSense_StartSampleCallback(uint8 MagSense_widgetId, uint8 MagSense_sensorId) | MagSense_START_SAMPLE_CALLBACK | perform additional application-specific actions

Used before each sensor scan triggering and deliver the current widget / sensor Id

Global Variables

Description
The section documents the MagSense component related global Variables. The MagSense 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
- MagSense_RAM_STRUCT MagSense_dsRam

Variable Documentation
MagSense_RAM_STRUCT MagSense_dsRam

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

API Constants

Description
The section documents the MagSense component related API Constants.

Variables
- const MagSense_FLASH_STRUCT MagSense_dsFlash
- const MagSense_FLASH_IO_STRUCT MagSense_ioList[MagSense_TOTAL_ELECTRODES]

Variable Documentation

const MagSense_FLASH_STRUCT MagSense_dsFlash

Constant for the FLASH Data Structure

cost MagSense_FLASH_IO_STRUCT MagSense_ioList[MagSense_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 MagSense 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. MagSense 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 MagSense 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.

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

The MagSense 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 MagSense_RegisterMap header file list all registers of data structure with the following:

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

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

1. Direct Access
   The access to registers is performed through the Data Structure variable MagSense_dsRam and constants MagSense_dsFlash from application program.
   Example of access to the Raw Count register of third sensor of Button0 widget:
   ```c
   rawCount = MagSense_dsRam.snsList.button0[MagSense_BUTTON0_SNS2_ID].raw[0];
   ```
   Corresponding macro to access register value is defined in the MagSense_RegisterMap header file:
   ```c
   rawCount = MagSense_BUTTON0_SNS2_RAW0_VALUE;
   ```

2. Getter/Setter Access
   The access to registers from application program is performed by using two functions:
   ```c
   cystatus MagSense_GetParam(uint32 paramId, uint32 *value)
cystatus MagSense_SetParam(uint32 paramId, uint32 value)
   ```
   The value of paramId argument for each register can be found in MagSense_RegisterMap header file.
   Example of access to the Raw Count register of third sensor of Button0 widget:
   ```c
   MagSense_GetParam(MagSense_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):
   ```c
   MagSense_SetParam(MagSense_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 MagSense data structure to be exposed to I2C master at primary slave address request once at initialization an application program:
   ```c
   EZI2C_Start();
   EZI2C_EzI2CSetBuffer1(sizeof(MagSense_dsRam), sizeof(MagSense_dsRam),
                        (uint8 *)&MagSense_dsRam);
   ```
   Now host can read (write) the whole MagSense Data Structure and get the specified register value by register offset macro available in MagSense_RegisterMap header file:
   ```c
   rawCount = *((uint16 *)(I2C_buffer1Ptr + MagSense_BUTTON0_SNS2_RAW0_OFFSET));
   ```
The current example is applicable to 2-byte registers only. Depends on register size defined in the MagSense_RegisterMap header file by corresponding macros (MagSense_BUTTON0_SNS2_RAW0_SIZE), specific logic should be added to read 4-byte, 2-byte and 1-byte registers.

Data Structures

- **struct MagSense_RAM WD BASE STRUCT**
  Declares common widget RAM parameters.

- **struct MagSense_RAM WD PROXIMITY STRUCT**
  Declares RAM parameters for the ISX Proximity.

- **struct MagSense_RAM WD LIST STRUCT**
  Declares RAM structure with all defined widgets.

- **struct MagSense_RAM SNS STRUCT**
  Declares RAM structure for sensors.

- **struct MagSense_RAM SNS LIST STRUCT**
  Declares RAM structure with all defined sensors.

- **struct MagSense_RAM STRUCT**
  Declares the top-level RAM Data Structure.

- **struct MagSense_FLASH_IO STRUCT**
  Declares the Flash IO object.

- **struct MagSense_FLASH SNS STRUCT**
  Declares the Flash Electrode object.

- **struct MagSense_FLASH SNS LIST STRUCT**
  Declares the structure with all Flash electrode objects.

- **struct MagSense_FLASH WD STRUCT**
  Declares Flash widget object.

- **struct MagSense_FLASH STRUCT**
  Declares top-level Flash Data Structure.

- **struct MagSense_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 MagSense_RAM WD BASE STRUCT**
Go to the top of the Data Structures section.

**Data Fields:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resolution</td>
<td>uint16</td>
<td>Provides scan resolution or number of sub-conversions.</td>
</tr>
<tr>
<td>MagSense_THRESHOLD TYPE</td>
<td>uint8</td>
<td>Widget Finger Threshold.</td>
</tr>
<tr>
<td>noiseTh</td>
<td>uint8</td>
<td>Widget Noise Threshold.</td>
</tr>
<tr>
<td>nNoiseTh</td>
<td>uint8</td>
<td>Widget Negative Noise Threshold.</td>
</tr>
<tr>
<td>hysteresis</td>
<td>uint8</td>
<td>Widget Hysteresis for the signal crossing finger or touch/proximity threshold.</td>
</tr>
<tr>
<td>onDebounce</td>
<td>uint8</td>
<td>Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.</td>
</tr>
</tbody>
</table>
MagSense_LOW_BSLN_RST_TYPE | lowBsInRst | 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.

| struct MagSense_RAM_WD_PROXIMITY_STRUCT | | Go to the top of the Data Structures section.

**Data Fields:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resolution</td>
<td>uint16</td>
<td>Provides scan resolution or number of sub-conversions.</td>
</tr>
<tr>
<td>fingerTh</td>
<td>MagSense_THRESHOLD_D_TYPE</td>
<td>Widget Finger Threshold.</td>
</tr>
<tr>
<td>noiseTh</td>
<td>uint8</td>
<td>Widget Noise Threshold.</td>
</tr>
<tr>
<td>nNoiseTh</td>
<td>uint8</td>
<td>Widget Negative Noise Threshold.</td>
</tr>
<tr>
<td>hysteresis</td>
<td>uint8</td>
<td>Widget Hysteresis for the signal crossing finger or touch/proximity threshold.</td>
</tr>
<tr>
<td>onDebounce</td>
<td>uint8</td>
<td>Widget Debounce for the signal above the finger or touch/proximity threshold. OFF to ON.</td>
</tr>
<tr>
<td>lowBsInRst</td>
<td>MagSense_LOW_BSLN_RST_TYPE</td>
<td>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.</td>
</tr>
<tr>
<td>idacMod[MagSense_NUM_SCAN_FREQS]</td>
<td>uint8</td>
<td>Sets the current of the modulation IDAC for the widget.</td>
</tr>
<tr>
<td>idacGainIndex</td>
<td>uint8</td>
<td>The index of the IDAC gain in the IDAC gain table structure for the widgets.</td>
</tr>
<tr>
<td>snsClk</td>
<td>uint16</td>
<td>Sets Lx clock divider for ISX Widgets.</td>
</tr>
<tr>
<td>snsClkSource</td>
<td>uint8</td>
<td>Register for internal use</td>
</tr>
<tr>
<td>proxTouchTh</td>
<td>MagSense_THRESHOLD_D_TYPE</td>
<td>The proximity touch threshold.</td>
</tr>
</tbody>
</table>

**struct MagSense_RAM_WD_LIST_STRUCT**

Go to the top of the Data Structures section.

**Data Fields:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MagSense_RAM_WD_PROXIMITY_STRUCT</td>
<td>proximity0</td>
<td>Proximity0 widget RAM structure</td>
</tr>
</tbody>
</table>

**struct MagSense_RAM_SNS_STRUCT**

Go to the top of the Data Structures section.
### Data Fields:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint16</code></td>
<td><code>raw[MagSense_NUM_SCAN_FREQS]</code></td>
</tr>
<tr>
<td><code>uint16</code></td>
<td><code>bsln[MagSense_NUM_SCAN_FREQS]</code></td>
</tr>
<tr>
<td><code>uint8</code></td>
<td><code>bslnExt[MagSense_NUM_SCAN_FREQS]</code></td>
</tr>
<tr>
<td><code>MagSense_THRESHOLD_D_TYPE</code></td>
<td><code>diff</code></td>
</tr>
<tr>
<td><code>MagSense_LOW_BSLN_RST_TYPE</code></td>
<td><code>negBslnRstCnt[MagSense_NUM_SCAN_FREQS]</code></td>
</tr>
<tr>
<td><code>uint8</code></td>
<td><code>idacComp[MagSense_NUM_SCAN_FREQS]</code></td>
</tr>
</tbody>
</table>

#### struct MagSense_RAM_SNS_LIST_STRUCT

Go to the top of the Data Structures section.

**Data Fields:**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MagSense_RAM_STRUCT</code></td>
<td><code>proximity0[MagSense_PROXIMITY0_NUM_RX]</code></td>
</tr>
</tbody>
</table>

#### struct MagSense_RAM_STRUCT

Go to the top of the Data Structures section.

**Data Fields:**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint16</code></td>
<td><code>configId</code></td>
</tr>
<tr>
<td><code>uint16</code></td>
<td><code>deviceId</code></td>
</tr>
<tr>
<td><code>uint16</code></td>
<td><code>hwClock</code></td>
</tr>
<tr>
<td><code>uint16</code></td>
<td><code>tunerCmd</code></td>
</tr>
<tr>
<td><code>uint16</code></td>
<td><code>scanCounter</code></td>
</tr>
<tr>
<td><code>volatile uint32</code></td>
<td><code>status</code></td>
</tr>
<tr>
<td><code>uint32</code></td>
<td><code>wdgtEnable[MagSense_WDGT_STATUS_WORDS]</code></td>
</tr>
<tr>
<td><code>uint32</code></td>
<td><code>wdgtStatus[MagSense_WDGT_STATUS_WORDS]</code></td>
</tr>
<tr>
<td><code>MagSense_STS_TYPE</code></td>
<td><code>snsStatus[MagSense_TOTAL_WIDGETS]</code></td>
</tr>
</tbody>
</table>
YPEn | active; 01 - Proximity detected (signal above finger threshold); 11 - A finger touch detected (signal above the touch threshold). For the Button widget, only one bit is used and means: 0 - Not active; 1 - Finger detected. For the Encoder Dial Widget, four bits are used. The lower 2 bits represent current sensor status; the upper 2 bits represent previous sensor statuses. E.g., 0111 means sensor0 active, sensor1 active, sensor0 was active, sensor 1 was not. The array size is equal to the total number of widgets. The size of the array element depends on the max number of sensors per widget used in the current design. It could be 1, 2 or 4 bytes.

| uint16 | csd0Config | The configuration register for global parameters of the SENSE_HW0 block.
| uint8  | modIsxClk  | The modulator clock divider for the ISX widgets.

**MagSense_RAM WD_LIST_STRUCT**

| wdgList | RAM Widget Objects.
| snsList | RAM Sensor Objects.

| uint8  | snrTestWidgetId | The selected widget ID.
| uint8  | snrTestSensorId | The selected sensor ID.
| uint16 | snrTestScanCounter | The scan counter.
| uint16 | snrTestRawCount | The sensor raw counts.
| uint8  | scanCurrentISC | The current inactive sensor connection state for the sensors.

### struct MagSense_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 MagSense_FLASH_SNS_STRUCT
Go to the top of the Data Structures section.

Data Fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint16</td>
<td>firstPinId Index of the first IO in the Flash IO Object Array.</td>
</tr>
<tr>
<td>uint8</td>
<td>numPins Total number of IOs in this sensor.</td>
</tr>
<tr>
<td>uint8</td>
<td>type Sensor type:</td>
</tr>
</tbody>
</table>

struct MagSense_FLASH_SNS_LIST_STRUCT
Go to the top of the Data Structures section.

Data Fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8</td>
<td>notUsed No ganged sensors available</td>
</tr>
</tbody>
</table>

struct MagSense_FLASH_WD_STRUCT
Go to the top of the Data Structures section.

Data Fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void const *</td>
<td>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.</td>
</tr>
<tr>
<td>void *</td>
<td>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.</td>
</tr>
<tr>
<td>MagSense_RAM_SNS_STRUCT*</td>
<td>ptr2SnsRam Points to the array of Sensor Objects in RAM. The sensing and processing blocks use it to access the scan data.</td>
</tr>
<tr>
<td>void *</td>
<td>ptr2FltrHistory Points to the array of the Filter History Objects in RAM that belongs to this widget.</td>
</tr>
<tr>
<td>uint8 *</td>
<td>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.</td>
</tr>
<tr>
<td>uint32</td>
<td>staticConfig Miscellaneous configuration flags.</td>
</tr>
<tr>
<td>uint16</td>
<td>totalNumSns The total number of sensors. For CSD widgets: WD_NUM_ROWS + WD_NUM_COLS. For CSX widgets: WD_NUM_ROWS * WD_NUM_COLS.</td>
</tr>
<tr>
<td>uint8</td>
<td>wdgType 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</td>
</tr>
<tr>
<td>uint8</td>
<td>numCols For ISX Proximity Widgets, the number of sensors.</td>
</tr>
<tr>
<td>uint8</td>
<td>numRows Unused.</td>
</tr>
</tbody>
</table>

struct MagSense_FLASH_STRUCT
Go to the top of the Data Structures section.
Data Fields:

| MagSense_FLASH_WD_STRUCT | wdgArray[MagSense_TOTAL_WIDGETS] | Array of flash widget objects |

struct MagSense_BSLN_RAW_RANGE_STRUCT

Go to the top of the Data Structures section.

Data Fields:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint16</td>
<td>bslnHiLim</td>
<td>Upper limit of a sensor baseline.</td>
</tr>
<tr>
<td>uint16</td>
<td>bslnLoLim</td>
<td>Lower limit of a sensor baseline.</td>
</tr>
<tr>
<td>uint16</td>
<td>rawHiLim</td>
<td>Upper limit of a sensor raw count.</td>
</tr>
<tr>
<td>uint16</td>
<td>rawLoLim</td>
<td>Lower limit of a sensor raw count.</td>
</tr>
</tbody>
</table>

Go to the top of the MagSense Low-Level APIs section.
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:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Memory Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISX Component base configuration (1 sensor)</td>
<td>4430</td>
</tr>
<tr>
<td>Memory consumption for each additional sensor</td>
<td>120</td>
</tr>
<tr>
<td>+288</td>
<td>+32</td>
</tr>
</tbody>
</table>

**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.

MagSense Tuner

The MagSense 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 MagSense_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 MagSense Design guide.
Tuner Quick Start

This section shows how to set up MagSense tuner via an I²C 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.

Note MiniProg3 does not support UART communication. You can use KitProg, MiniProg3 or MiniProg4 debugger/programmer kit as the USB-I2C Bridge.

Step-3: Modify Application Code

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

```c
#include <project.h>

int main()
{
    CyGlobalIntEnable; /* Enable global interrupts */
    EZI2C_Start(); /* Start EZI2C Component */
    /* Set up communication and initialize data buffer to MagSense data structure
     * to use Tuner application
     */
    EZI2C_EzI2CSetBuffer1(sizeof(MagSense_dsRam), sizeof(MagSense_dsRam), (uint8 *)&MagSense_dsRam);
    MagSense_Start(); /* Initialize Component */
    MagSense_ScanAllWidgets(); /* Scan all widgets */

    for(;;)
    {
        /* Do this only when a scan is done */
        if(MagSense_NOT_BUSY == MagSense_IsBusy())
        {
            MagSense_ProcessAllWidgets(); /* Process all widgets */
            MagSense_RunTuner(); /* To sync with Tuner application */
            if (MagSense_IsAnyWidgetActive()) /* Scan result verification */
            {
                /* add custom tasks to execute when touch detected */
            }
        }
        MagSense_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 MagSense Component in the schematic and select **Launch Tuner** from the context menu.
The **MagSense 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 MagSense 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 the *Widget View* and signals for the sensor in the *Graph View*. Touch the sensors on the kit to observe MagSense 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 MagSense 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 *MagSense_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 MagSense 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 **disconnected** otherwise.

- **Slave address** [I2C specific] – The address of the I2C slave configured for the current communication channel.

- **I2C clock** [I2C specific] – The data rate used by the I2C communication channel.

- **Supply voltage** – The supply voltage.

- **Logging** – Either **ON** (when the data logging to a file in progress) or **OFF** otherwise.
Widget Explorer Pane

The Widget explorer pane contains a tree of widgets and sensors used in the MagSense 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 MagSense 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: MagSense_RunTuner(). When Synchronized Communication mode is selected, the MagSense 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 MagSense_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 MagSense Tuner and data processing happen asynchronously, the MagSense 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 MagSense 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 MagSense Component has the following specific deviations:

<table>
<thead>
<tr>
<th>MISRA-C:2004 Rule</th>
<th>Rule Class (Required/Advisory)</th>
<th>Rule Description</th>
<th>Description of Deviation(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.8</td>
<td>R</td>
<td>An external object or function shall be declared in only one file.</td>
<td>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.</td>
</tr>
<tr>
<td>11.4</td>
<td>A</td>
<td>A cast should not be performed between a pointer to object type and a different pointer to object type.</td>
<td>Pointers are used to allow many types of widgets and sensors. The architecture is designed to allow indexing a specific pointer.</td>
</tr>
</tbody>
</table>
| 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 MagSense Component has many different configurations. Some of them do not use specific function parameters. To avoid the compiler's warning, the following code is used: (void)paramName. |
| 16.7              | A                              | A pointer parameter in a function prototype should be declared as the pointer to const if the pointer is not used to modify the addressed object. | Mostly all data processing for variety configuration, widgets and data types is required to pass the pointers as an argument. The architecture 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. |
MISRAC2004 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. MagSense_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)**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Condition / Details</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum sample rate</td>
<td></td>
<td></td>
<td>10</td>
<td>ksps</td>
<td></td>
</tr>
<tr>
<td>Calibration time</td>
<td>SysClk = 48MHz, ModClk = 48MHz, LxClk = 1MHz, Sub Conversion = 50, Widget/Sensors scanned = 1, Firmware filters = Disabled, Auto-Calibration = Enabled. All other components parameter = default.</td>
<td></td>
<td>33</td>
<td>ms</td>
<td></td>
</tr>
<tr>
<td>Sensor frequency</td>
<td>Supported LC resonant frequency. ISX parameters = tuned per CY design guide.</td>
<td>45</td>
<td>3,000</td>
<td>kHz</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Condition / Details</td>
<td>Min</td>
<td>Typ</td>
<td>Max</td>
<td>Units</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>Inductance (L) range</td>
<td>Supported Inductance range, ISX parameter = tuned per CY design guide.</td>
<td>1</td>
<td></td>
<td>10,00</td>
<td>uH</td>
</tr>
<tr>
<td>Resolution</td>
<td>Hardware resolution</td>
<td></td>
<td></td>
<td>16</td>
<td>bits</td>
</tr>
<tr>
<td>Maximum detectable distance</td>
<td>SysClk/MkdClk= 48MHz, Firmware filters = Disabled, Auto-Calibration = Enabled</td>
<td>4.5</td>
<td></td>
<td></td>
<td>mm</td>
</tr>
<tr>
<td>Maximum detectable distance</td>
<td>SysClk/MkdClk= 48MHz, Firmware filters = Disabled, Auto-Calibration = Enabled</td>
<td>19.5</td>
<td></td>
<td></td>
<td>mm</td>
</tr>
<tr>
<td>Average power consumption</td>
<td>LX Frequency= 3MHz, Coil diameter = 10 mm, Signal variation = 0.2%</td>
<td></td>
<td></td>
<td>0.61*</td>
<td>mA</td>
</tr>
<tr>
<td></td>
<td>(Minimum detectable change), Report rate &lt;= 100ms.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>All other components parameter = default.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active power consumption</td>
<td>At maximum sample rate, no periodic deepsleep/sleep.</td>
<td></td>
<td></td>
<td>10.01*</td>
<td>mA</td>
</tr>
</tbody>
</table>

*Total chip current

**IDAC Characteristic**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Units</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDAC1DNL</td>
<td>DNL</td>
<td>-1</td>
<td>-</td>
<td>1</td>
<td>LSB</td>
<td></td>
</tr>
<tr>
<td>IDAC1INL</td>
<td>INL</td>
<td>-2</td>
<td>-</td>
<td>2</td>
<td>LSB</td>
<td>INL is ±5.5 LSB for VDDA &lt; 2 V</td>
</tr>
<tr>
<td>IDAC2DNL</td>
<td>DNL</td>
<td>-1</td>
<td>-</td>
<td>1</td>
<td>LSB</td>
<td></td>
</tr>
<tr>
<td>IDAC2INL</td>
<td>INL</td>
<td>-2</td>
<td>-</td>
<td>2</td>
<td>LSB</td>
<td>INL is ±5.5 LSB for VDDA &lt; 2 V</td>
</tr>
</tbody>
</table>

**DC/AC Specifications**

Refer to device-specific datasheet *PSoc 4 Device datasheets* for more details.
## Component Changes

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

<table>
<thead>
<tr>
<th>Version</th>
<th>Description of Changes</th>
<th>Reason for Changes / Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.0.a</td>
<td>Minor datasheet edits.</td>
<td></td>
</tr>
<tr>
<td>7.0</td>
<td>Added limitation of Number of Subconversions used in the calibration algorithm.</td>
<td>Defect fixing.</td>
</tr>
<tr>
<td>6.0</td>
<td>New Component implementation.</td>
<td></td>
</tr>
</tbody>
</table>

© Cypress Semiconductor Corporation, 2019. 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, ModusToolbox, 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.