

## Chapter 4: CAPSENSE™ Technology

At the end of this chapter you will understand the software and tools that are included in Infineon's CAPSENSE™ solution. You will be able to write firmware to create PSoC™ applications using CAPSENSE™ technology and then use the CAPSENSE™ tools to configure them.

### Table of contents

<b>4.1</b>	<b>Overview .....</b>	<b>2</b>
4.1.1	CAPSENSE™ Technology .....	2
4.1.2	Capacitive Sigma Delta (CSD) HW Block .....	5
4.1.3	Multi Sense Converter (MSC) HW Block .....	5
<b>4.2</b>	<b>CAPSENSE™ Configurator .....</b>	<b>6</b>
4.2.1	Toolbar .....	6
4.2.2	Tabs .....	7
<b>4.3</b>	<b>CAPSENSE™ middleware library .....</b>	<b>12</b>
4.3.1	Touch sensing .....	12
4.3.2	CSDADC .....	15
4.3.3	CSDIDAC .....	17
4.3.4	Time multiplexing .....	19
<b>4.4</b>	<b>CAPSENSE™ Tuner .....</b>	<b>20</b>
4.4.1	Device-Tuner communication interface .....	20
4.4.2	Toolbar .....	23
4.4.3	Tuner communication setup .....	25
4.4.4	Widget/Sensor Explorer and Parameters panes .....	26
4.4.5	Tabs .....	27
<b>4.5</b>	<b>Exercises .....</b>	<b>32</b>
Exercise 1: Control LED .....		32
Exercise 2: Add a Tuner communication interface to the LED control project .....		32
Exercise 3: Time multiplexing .....		33

### Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	CY_ISR_PROTO (MyISR) ; make build
<i>Italics</i>	Displays file names and paths	<i>sourcefile.hex</i>
[bracketed, bold]	Displays keyboard commands in procedures	[Enter] or [Ctrl] [C]
Menu > Selection	Represents menu paths	File > New Project > Clone
<b>Bold</b>	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the <b>Debugger</b> icon, and then click <b>Next</b> .

## 4.1 Overview

This chapter covers only the basics of Infineon's CAPSENSE™ technology. For many more details about CAPSENSE™ technology, refer to the [CAPSENSE™ Design Guide](#).

Capacitive touch sensing has changed the face of design in consumer and industrial products. Infineon's CAPSENSE™ solutions bring elegant, reliable, and easy-to-use capacitive touch sensing functionality to your design. Our capacitive touch sensing solutions have replaced more than four billion mechanical buttons, and these solutions enable hundreds of diverse types of sensing applications.

CAPSENSE™ technology offers industry-leading, low-power operation, boasts the industry's widest voltage ranges (1.71V-5.5V), and provides the industry's best solution for liquid tolerance to prevent false touches in wet/moist environments.

Infineon CAPSENSE™ provides both self-capacitance (CSD) and mutual-capacitance (CSX) with the same hardware blocks and middleware libraries so that you can choose the optimum solution for your application. In some applications, the same sensors are used in both modes to provide the best combination of noise immunity, water tolerance and multi-touch capabilities.

CAPSENSE™ technology can be used in a variety of form factors such as buttons, sliders (a linear progression of related buttons whose signals can be interpolated to achieve a fine-grained position), and touch pads or touch screens (a 2-dimensional array of buttons that allow an X-Y position to be interpolated based on row and column sensor measurements).

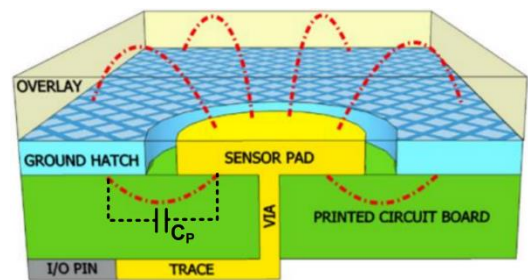
### 4.1.1 CAPSENSE™ Technology

A capacitor simply consists of two conductors separated by an insulator. At the most basic level, Infineon's CAPSENSE™ technology works by creating small capacitors on a PCB, and then measuring their change in capacitance in the presence of a finger. There are two main strategies for measuring this change in capacitance: self-cap and mutual-cap.

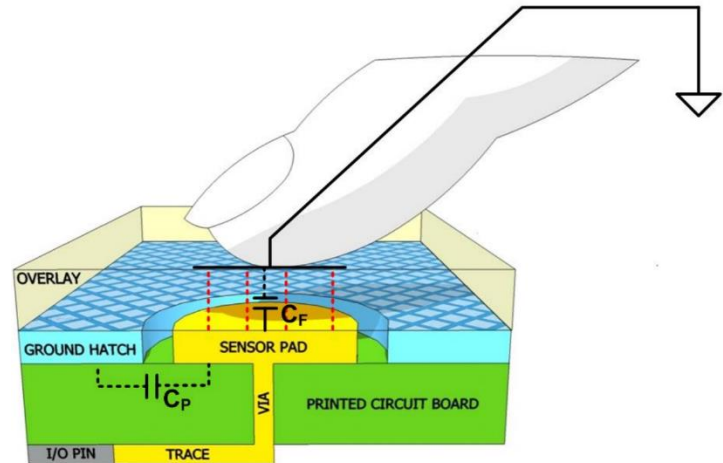
#### 4.1.1.1 Self-capacitance (Self-cap)

A typical self-capacitance based CAPSENSE™ sensor consists of a conductive (usually copper or indium tin oxide) pad of proper shape and size, laid on the surface of a non-conductive material like PCB or glass. A non-conductive overlay serves as the touch surface for the button while PCB traces connect the sensor pads to PSoC™ GPIOs that are configured as CAPSENSE™ sensor pins.

Typically, a ground hatch surrounds the sensor pad to isolate it from other sensors and traces. The intrinsic capacitance of the PCB trace or other connections to a capacitive sensor results in a sensor parasitic capacitance ( $C_p$ ). (Note that the red-colored electric field lines are only a representative of the electric-field distribution around the sensor, the actual electric field distribution is very complex).



When a finger is present on the overlay, the conductive nature and large mass of the human body forms a grounded, conductive plane parallel to the sensor pad. This adds a finger capacitance ( $C_F$ ) in parallel to the parasitic capacitance. Note that the parasitic capacitance  $C_P$  and finger capacitance  $C_F$  are parallel to each other because both represent the capacitance between the sensor pin and ground.

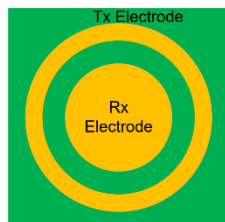


#### 4.1.1.2 Mutual-capacitance (Mutual-cap)

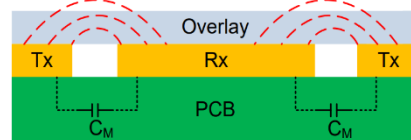
Mutual-capacitance sensing measures the capacitance between two electrodes, which are called transmit (Tx) and receive (Rx) electrodes.

In a mutual-capacitance measurement system, a digital voltage (usually signal switching between VDD and GND) is applied to the Tx pin and the amount of charge received on the Rx pin is measured. The amount of charge received on the Rx electrode is directly proportional to the mutual capacitance ( $C_M$ ) between the two electrodes.

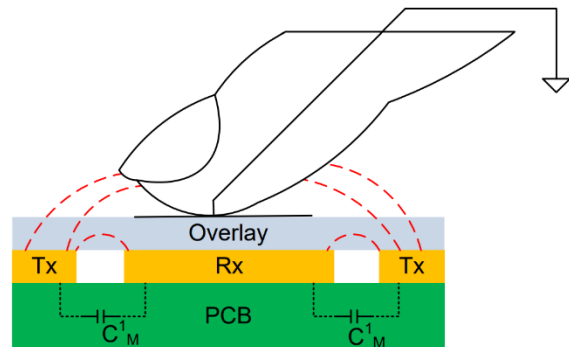
When a finger is placed between the Tx and Rx electrodes, some of the field lines terminate on the figure so the mutual-capacitance between the Tx and Rx electrodes decreases to  $C_M^1$ . Because of the reduction in capacitance between the electrodes, the charge received on the Rx electrode also decreases. The CAPSENSE™ system measures the amount of charge received on the Rx electrode to detect the touch/no touch condition.



Top View



Side View – No Touch



Side View – Touch

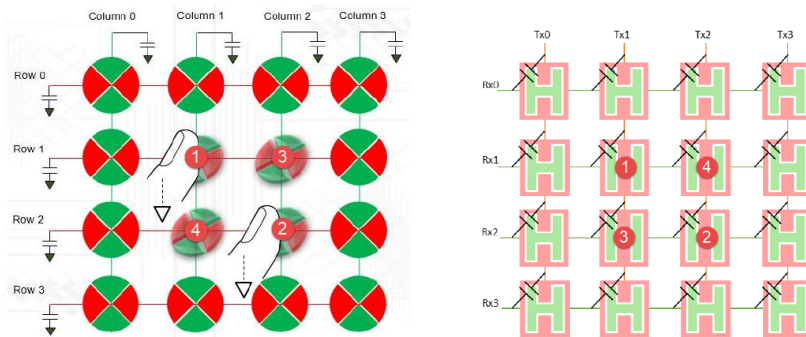
#### 4.1.1.3 Multi-finger Detection

For touchpads and touchscreens, it is often desired to detect more than one finger's position at a time.

If self-cap is used, only one finger's position can be sensed at a time. The sensor is usually made up of rows and columns of self-cap buttons. By using row values for the Y position and column values for the X position,

the finger location can be determined. More than one finger placed on the touchpad or touchscreen leads to multiple row and column positions, so it is not possible to determine which row position goes with each column position. For example, the figure on the left below shows the self-cap case. Fingers placed at locations 1 and 2 would be indistinguishable from fingers placed at locations 3 and 4. In both cases, the same four sensors would be activated – Row 1, Row 2, Column 1 and Column 2.

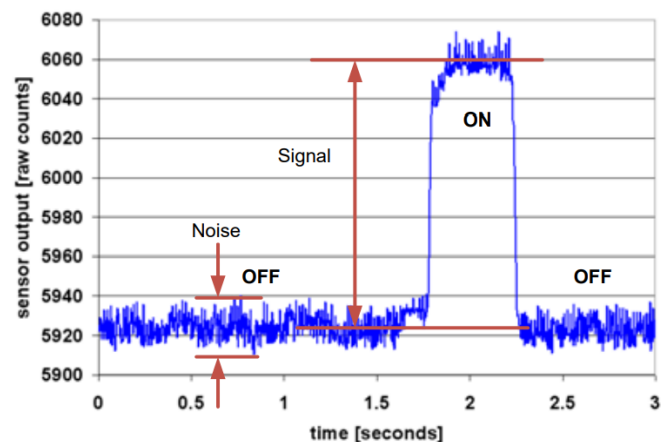
With mutual-cap sensing, the sensors are the intersection between each row and column. Signals are sent individually on each of the Tx lines and are received on each of the Rx lines. Therefore, it is possible to detect a unique X and Y position for as many fingers as desired by interpreting each of the individual intersection signals. In the figure on the right below, each of the four positions labeled 1 – 4 would activate a unique sensor at that intersection.



#### 4.1.1.4 Raw counts

PSoC™ devices use a Capacitive Sigma Delta (CSD) HW block to convert the analog capacitance of a CAPSENSE™ sensor -  $C_s$  - into an equivalent digital value called raw counts. An increase of the value raw counts indicates a finger touch (for mutual-cap the actual capacitance decreases for a touch but the PSoC™ device handles this inversion so that the raw count that you see will always increase for a finger touch).

When a finger touches the sensor,  $C_s$  increases from  $C_P$  to  $C_P + C_F$ , and the raw count value increases. By comparing the change in raw count to a predetermined threshold, logic in the CAPSENSE™ HW decides whether the sensor is active (finger is present).



#### 4.1.1.5 Signal to Noise Ratio (SNR)

The raw counts will vary with respect to time due to inherent noise in the system even if a finger doesn't move. CAPSENSE™ noise is the peak-to-peak variation in raw counts in the absence of a touch, and CAPSENSE™ signal is the average shift in raw counts brought in by a finger touch on the sensor. SNR is the ratio of CAPSENSE™ signal to CAPSENSE™ noise. For a reliable touch detection, it is required to have a minimum SNR of 5:1, but a larger value is recommended if possible. This requirement comes from best practice threshold settings, which enable enough margin between signal and noise in order to provide reliable ON/OFF operation.

Note that there is also drift over longer periods of time due mainly to environmental conditions such as voltage or temperature changes. This is handled by using a baseline value for the no-touch condition that can be periodically updated to account for any drift that occurs.

#### **4.1.1.6 SmartSense auto-tuning**

CAPSENSE™ finger and location detection is a sophisticated algorithm enabled by a combination of hardware and firmware blocks inside PSoC™ devices. As such, it has several hardware and firmware parameters required for proper operation. These parameters need to be tuned to optimum values for reliable touch detection and fast response times.

SmartSense is a CAPSENSE™ tuning method that automatically sets many of these parameters for optimal performance based on the user specified finger capacitance, and continuously compensates for system, manufacturing, and environmental changes.

SmartSense auto-tuning reduces design cycle time and provides stable performance across PCB variations, but requires additional RAM and CPU resources to allow runtime tuning of CAPSENSE™ parameters.

On the other hand, manual tuning requires effort to find the optimum CAPSENSE™ parameters, but allows strict control over characteristics of a capacitive sensing system, such as response time and power consumption. It also allows the use of CAPSENSE™ technology for purposes beyond the conventional button and slider applications, such as proximity and liquid-level-sensing.

SmartSense is the recommended tuning method for all conventional CAPSENSE™ applications. You should use SmartSense auto-tuning if your design meets the following requirements:

- The design is for conventional user-interface application like buttons, sliders, touchpad etc.
- The parasitic capacitance ( $C_P$ ) of the sensors is within the SmartSense supported range.
- The sensor-scan-time chosen by SmartSense allows you to meet the response time and power requirements of the end system.
- SmartSense auto-tuning meets the RAM/flash requirements of the design.

In cases where you need to use manual tuning, it is possible to start with the parameters from SmartSense and then adjust as necessary to meet system performance. This is usually easier than setting manual tuning parameters from scratch.

### **4.1.2 Capacitive Sigma Delta (CSD) HW Block**

The CSD HW block is Infineon's fourth generation capacitive sensing HW block and is the HW block present on the kits used in this course. The CSD HW block enables multiple sensing capabilities on PSoC™ devices including self-cap and mutual-cap capacitive touch sensing, a 10-bit ADC, IDAC, and Comparator. The CSD HW block can support only one function at a time. However, all supported functionality (like touch sensing, ADC, etc.) can be time-multiplexed in a design. For example, you can save the existing state of the CAPSENSE™ middleware, change to a configuration using the ADC middleware, perform ADC measurements, and then switch back to the CAPSENSE™ functionality.

#### **4.1.3 Multi Sense Converter (MSC) HW Block**

The MSC HW block is Infineon's fifth generation capacitive sensing HW block. It supports all of the previously mentioned features of the CSD HW block as well as the following additional features:

- Improved signal to noise ratio – The MSC HW Block implements a new sensing technique called ratiometric sensing which offers a significantly improved noise performance compared to previous generation devices
- Improved refresh rate – The MSC HW block has a higher sensitivity than previous generations devices and requires less time to acquire a signal. In addition to this, two independent MSC HW blocks can scan sensors in parallel to significantly improve the refresh rate when a large number of sensors is being scanned.
- Improved CPU bandwidth – The MSC HW block supports scanning in CPU mode as well as DMA mode. CPU mode is the conventional interrupt driven mode, while DMA mode is capable of scanning almost entirely independently of the CPU. DMA mode reduces the CPU bandwidth requirement of scanning by 82% compared to CPU mode.
- Improved noise immunity – New sensing techniques implemented by the MSC HW block significantly reduce noise induced from the external environment.

*Note:* The CAPSENSE™ Configurator, Middleware, and Tuner support both the CSD and MSC HW blocks.

## 4.2 CAPSENSE™ Configurator

The CAPSENSE™ Configurator is a stand-alone tool included in the ModusToolbox™ tools package. It provides a GUI that allows you to easily create and configure CAPSENSE™ widgets and generate code to control the application firmware. You can launch the CAPSENSE™ Configurator from the Tools section of the Eclipse IDE for ModusToolbox™ Quick Panel. From the CLI, you can launch the configurator from an application directory using the command `make open CY_OPEN_TYPE=capsense-configurator`.

In this section we will take a broad and shallow look at the CAPSENSE™ Configurator using the CSD hardware block. For more information or for using the Configurator with the MSC hardware block, you should refer to the CAPSENSE™ Configurator User Guide, which you can access directly from the CAPSENSE™ Configurator by going to **Help > View Help**. See your device datasheet to determine which CAPSENSE™ hardware block is available on a given device.

This tool saves the settings you select in a CAPSENSE™ configuration file (a file with the extension ".cycapsense"). A ModusToolbox™ project's CAPSENSE™ configuration, by default, is saved in the BSP since it is related closely to the hardware:

`bsps/<Your BSP>/config/design.cycapsense`

*Note:* Many more details about CapSense and Tuning can be found in the PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ design guide. Just do an internet search for "CAPSENSE design guide" and it will be one of the first few results.








### 4.2.1 Toolbar

The CAPSENSE™ Configurator has a toolbar at the top of the window that looks like this:



It has icons that allow you to run commonly used commands by clicking them:



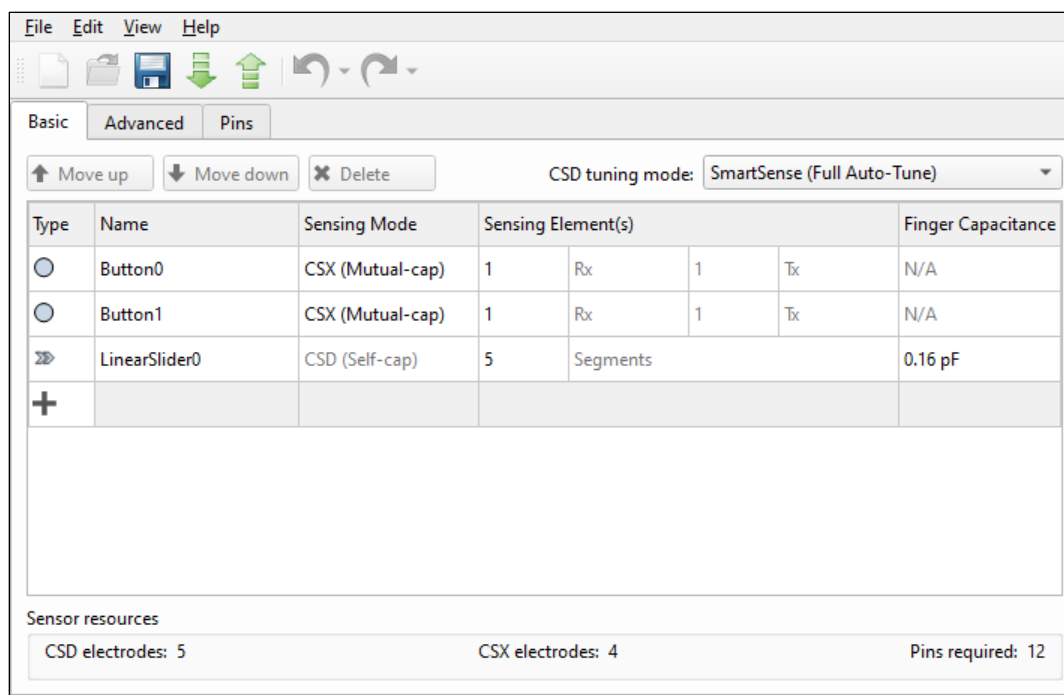
- New Configuration  – This closes the open CAPSENSE™ configuration and resets the configurator's workspace (this button is greyed out unless the tool is launched in stand-alone mode).
- Open Configuration  – This opens a CAPSENSE™ configuration and copies its configuration settings into the configurator's workspace (this button is greyed out unless the tool is launched in stand-alone mode).
- Save Configuration  – This saves the configuration settings in the configurator's workspace to the open CAPSENSE™ configuration. If there isn't a CAPSENSE™ configuration open in the configurator, you will have to specify where to save the configuration. If the open CAPSENSE™ configuration is the active configuration of a ModusToolbox™ project, the following files will be generated automatically every time you save:
  - *bsps/<Your BSP>/config/GeneratedSource/cycfg\_capsense.c*
  - *bsps/<Your BSP>/config/GeneratedSource/cycfg\_capsense.h*
- Import Configuration  – This copies the configuration settings from a specified configuration file into the configurator's workspace. This does NOT open the specified configuration file, it merely copies its configuration settings into the configurator's workspace.
- Export Configuration  – This saves the configuration settings in the configurator's workspace to a new CAPSENSE™ configuration file. Configuration settings do not have to be previously saved to a configuration file for you to export them.
- Undo  – This allows you to undo the last action or sequence of actions
- Redo  – This allows you to redo the last undone action or sequence of undone actions

## 4.2.2 Tabs

There are three tabs used to configure the CAPSENSE™ block. Each tab is discussed separately below.

### 4.2.2.1 Basic

When you first launch the CAPSENSE™ Configurator, you will see the **Basic** tab, which will look something like this:



Type	Name	Sensing Mode	Sensing Element(s)			Finger Capacitance	
○	Button0	CSX (Mutual-cap)	1	Rx	1	Tx	N/A
○	Button1	CSX (Mutual-cap)	1	Rx	1	Tx	N/A
⏏	LinearSlider0	CSD (Self-cap)	5	Segments			0.16 pF
+							

Sensor resources

CSD electrodes: 5      CSX electrodes: 4      Pins required: 12

From this tab you can add/delete widgets as well as assign each widget's sensing mode, sensing elements, and finger capacitance threshold. A widget is simply a CAPSENSE™ sensor or group of CAPSENSE™ sensors that perform specific user-interface functionality. The following widget types are currently supported:

- **Button** – One or more sensors that can detect the presence or absence of a finger on the sensor. There are multiple buttons on each of the PSoC™ kits used in this course.
- **Linear Slider** – More than one sensor arranged in a line in order to detect the movement of a finger on a linear axis. There is one linear slider on each of the PSoC™ kits used in this course.
- **Radial Slider** – More than one sensor arranged in a circle in order to detect the movement of a finger in a radial pattern.
- **Matrix Buttons** – Two or more sensors arranged in a grid pattern in order to detect the presence or absence of a finger on the intersections of vertically and horizontally arranged sensors. This works like an array of buttons but requires fewer sensors than the number of buttons. For example. A 3x3 matrix contains 9 buttons but only requires 6 sensors.
- **Touchpad** – Multiple sensors arranged in a grid pattern in order to detect the presence or absence of a human finger. If a finger is present, the widget will detect the physical position (both X and Y axis) of the touch. This widget works like a trackpad (and is in fact often used in touchpad applications).
- **Proximity** – One or more sensors. Each sensor can detect the proximity of conductive objects, such as a hand, finger, or cheek.

You can also set the CSD tuning mode from this tab. There are three tuning modes available:

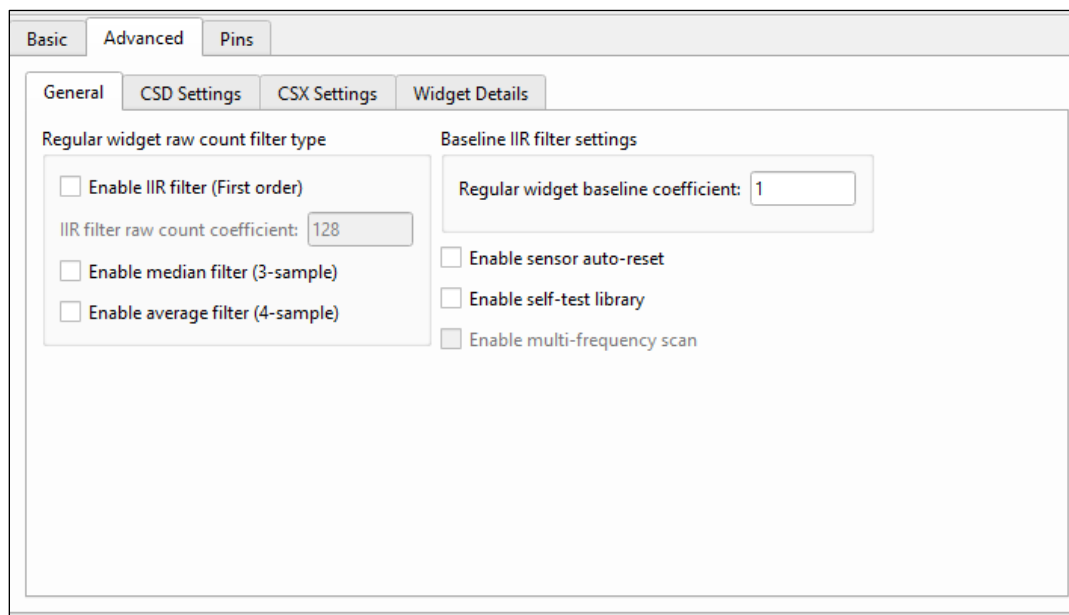
- **SmartSense (Full Auto-Tune)** – This is the quickest and easiest way to tune a design. Most hardware and threshold parameters are automatically tuned by the middleware.
- **SmartSense (Hardware parameters only)** – The hardware parameters are automatically set by the middleware; threshold parameters need to be set manually by the user.



- Manual – SmartSense auto-tuning is entirely disabled, both hardware and threshold parameters need to be set manually by the user.

#### 4.2.2.2 Advanced

If you click on the **Advanced** tab, you will see a window that looks like this:



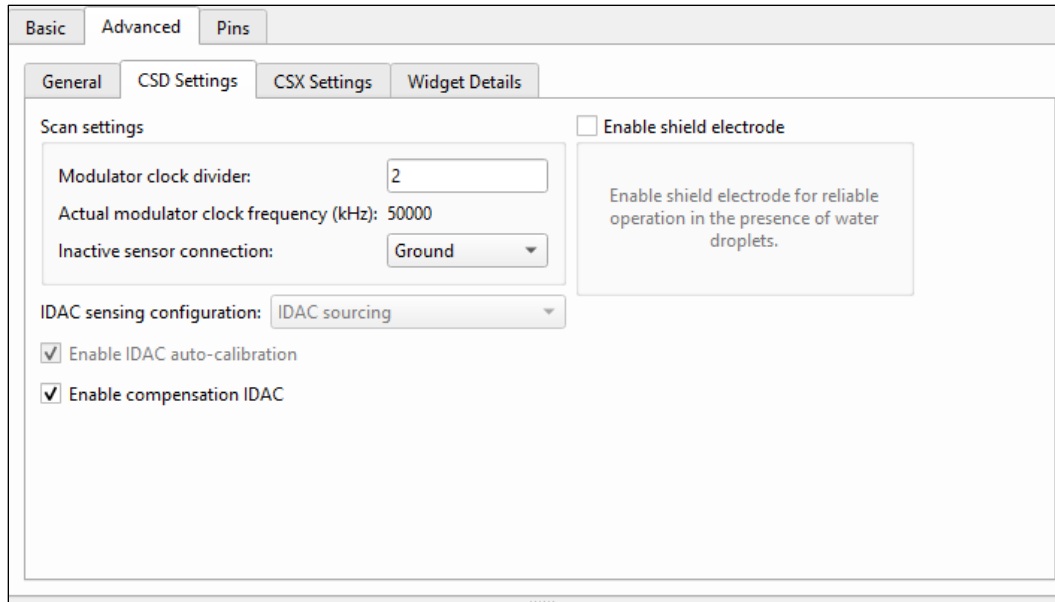
From this tab there are four sub-tabs available:

##### General

In this tab you can configure parameters that are common among all widgets irrespective of the sensing mode used. You can enable and configure filters, enable sensor auto-reset, and enable the self-test library. A complete description of each of these settings and their options can be found in the CAPSENSE™ Configurator User Guide.

##### CSD Settings

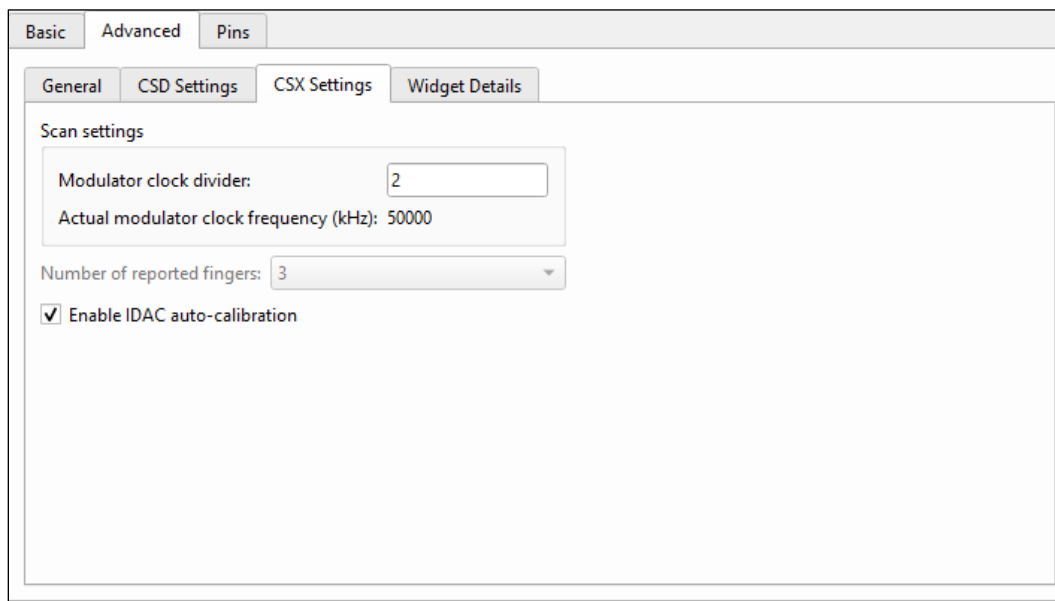
In the **CSD Settings** tab you can modify the parameters that are common among all widgets using the CSD sensing method (self-capacitance). You can configure scan settings, IDAC settings, and enable a shield electrode which is used to reduce parasitic capacitance and for liquid tolerance.



The screenshot shows the 'CSX Settings' tab in the ModusToolbox software. The interface includes a top navigation bar with 'Basic', 'Advanced', and 'Pins' tabs. Below this, there are sub-tabs: 'General', 'CSD Settings', 'CSX Settings' (which is active), and 'Widget Details'. The 'Scan settings' section contains a 'Modulator clock divider' set to 2, an 'Actual modulator clock frequency (kHz)' of 50000, and an 'Inactive sensor connection' dropdown set to 'Ground'. To the right, there is a checkbox for 'Enable shield electrode' which is unchecked, with a note below it stating 'Enable shield electrode for reliable operation in the presence of water droplets.' The 'IDAC sensing configuration' is set to 'IDAC sourcing'. At the bottom, there are two checked checkboxes: 'Enable IDAC auto-calibration' and 'Enable compensation IDAC'.

## CSX Settings

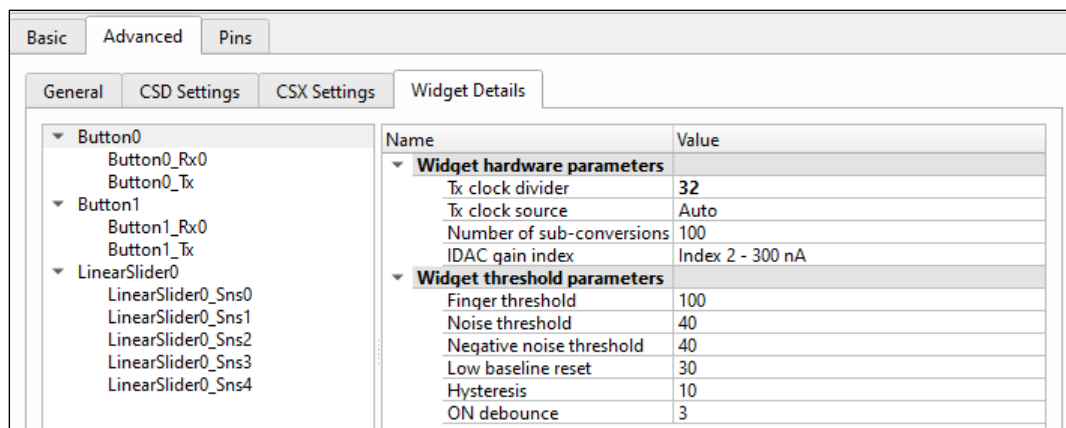
In the **CSX Settings** tab you can modify the parameters that are common among all widgets using the CSX sensing method (mutual-capacitance). You can configure scan settings and IDAC settings.



This screenshot is identical to the one above, showing the 'CSX Settings' tab. It highlights the 'Scan settings' section with the 'Modulator clock divider' at 2, 'Actual modulator clock frequency (kHz)' at 50000, and 'Inactive sensor connection' set to 'Ground'. It also shows the 'IDAC sensing configuration' set to 'IDAC sourcing' and the 'Enable IDAC auto-calibration' checkbox checked.

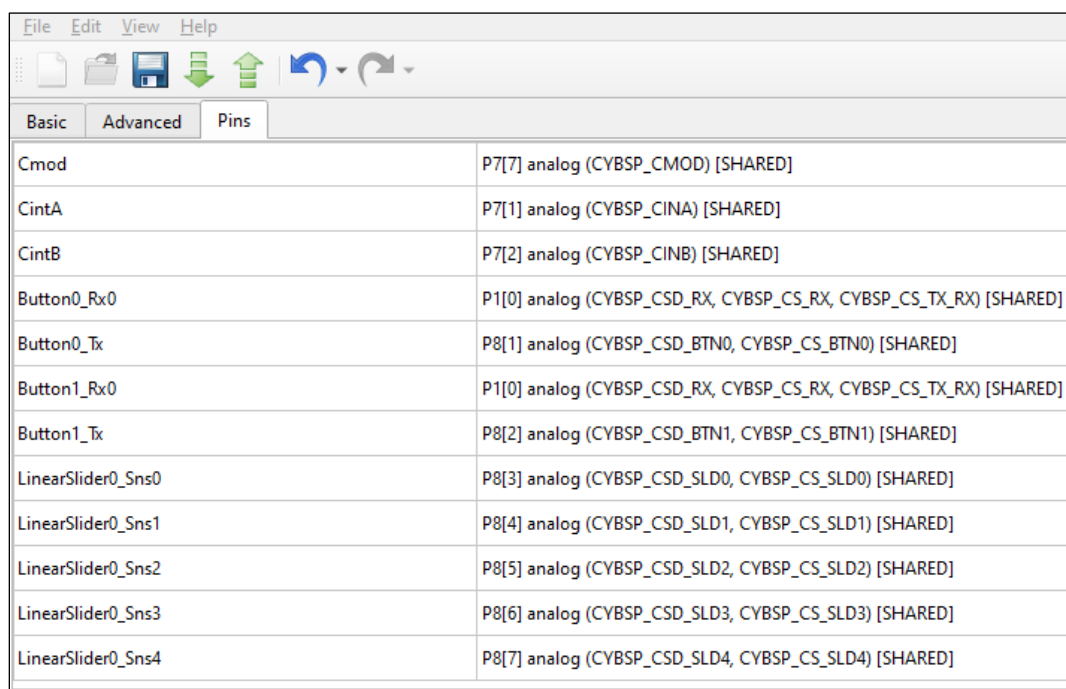
## Widget Details

In the **Widget Details** tab you can modify the parameters that are specific to each widget and sensor. Each widget type has its own unique set of parameters.



### 4.2.2.3 Pins

If you click on the **Pins** tab, you will see a window that looks like this:



From the pins tab you can assign pins to each sensor.

**Note:** The pin selections made on the **Pins** tab cause changes in the device configuration in the **Pins and Peripherals > System CSD** tabs. You can view and set the pins using either configurator, but you can only open one configurator at a time since they both require access to the pin configuration. It is easier to set the pins in the CAPSENSE™ Configurator since it makes all the necessary changes with a single selection of the pin.

## 4.3 CAPSENSE™ middleware library

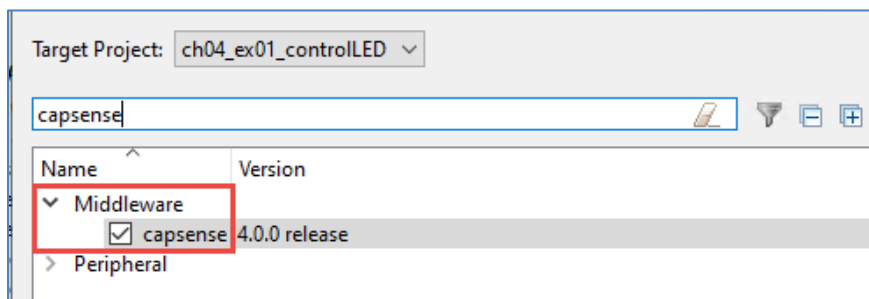
The CAPSENSE™ Middleware Library provides an API that allows you to easily control your CAPSENSE™ design from your application program. The best place to read about the API provided by this library is the CAPSENSE™ Middleware Library documentation, which can be accessed from the Documentation section of the Eclipse IDE for ModusToolbox™ Quick Panel, or from the *docs* directory inside the library. This documentation contains a complete description of the library API, as well as a plethora of code snippets and use case examples. Rather than repeat all of that information here, we will only discuss the basic flow of using each of the CSD HW block functionalities.

The CAPSENSE™ Middleware Library is built on top of the PDL CSD driver. For more information on this driver you can refer to the PDL documentation, which can be accessed from the Documentation section of the Eclipse IDE for ModusToolbox™ Quick Panel or from the library's *docs* directory. The HAL does not support the CSD HW block and there is no HAL based version of the CAPSENSE™ Middleware Library.

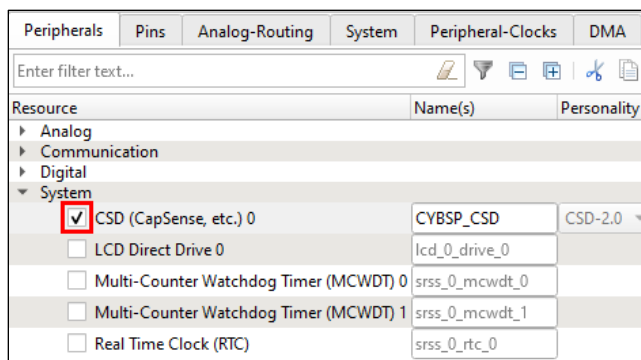
### 4.3.1 Touch sensing

#### 4.3.1.1 Configuration

To set up touch sensing on your device, the first thing you need to do is use the Library Manager to add the capsense middleware library.



Then, open the Device Configurator and verify that the CSD HW block is enabled:



Save and close the Device Configurator and then launch the CAPSENSE™ Configurator to configure the widgets that you want to use in your project. You need to verify or add the appropriate widgets and select the pins for each of the sensors you will be using. You should configure each widget's parameters according to the needs of your project. For more information about all of the options available in the CAPSENSE™ Configurator, you should refer to the CAPSENSE™ Configurator user guide.

**Note:** *If you are using an Infineon development kit that supports CAPSENSE™ technology, the BSP for that kit will include a CAPSENSE™ configuration that has already been set up to work with the kit's CAPSENSE™ sensors.*

Once you've set up your CAPSENSE™ configuration, save it and close the CAPSENSE™ Configurator.

#### 4.3.1.2 Firmware

In your application code, you first need to initialize the CAPSENSE™ block and then you can scan the sensors as often as you like.

##### Initialization

First, `#include "cycfg.h"` and `"cycfg_capsense.h"`. These files contain the code that was generated by the CAPSENSE™ Configurator.

When you are using the CSD HW block for touch sensing, it will generate an interrupt at the end of every sensor sampling, so you need to initialize this interrupt and set up its service routine. To do this you must first set up an interrupt configuration object of type `cy_stc_sysint_t` as follows:

```
const cy_stc_sysint_t CapSense_interrupt_config = {  
    .intrSrc = CYBSP_CSD_IRQ,  
    .intrPriority = CAPSENSE_INTR_PRIORITY  
};
```

**Note:** *If you are going to use the CAPSENSE™ Tuner, it is important that the priority of the CSD interrupt (CAPSENSE\_INTR\_PRIORITY) is lower than the priority of the tuner interrupt. (higher numbers correspond to lower priorities).*

Then you need to set up the CSD interrupt service routine. This should be a static function that only calls the function `Cy_CapSense_InterruptHandler`:

```
static void capsense_isr(void) {  
    Cy_CapSense_InterruptHandler(CYBSP_CSD_HW, &cy_capsense_context);  
}
```

Then call the functions `Cy_SysInt_Init` and `NVIC_EnableIRQ` to initialize and enable the CAPSENSE™ interrupt.

Initialize the CSD HW block by calling the function `Cy_CapSense_Init`.

You then need to enable the CAPSENSE™ firmware modules by calling the function `Cy_CapSense_Enable`.

Finally, you can set up callback functions to occur on specific CAPSENSE™ events. There are two events that can trigger a callback:

- `CY_CAPSENSE_START_SAMPLE_E` – Start of sample callback. This callback will be triggered before each sensor is sampled.
- `CY_CAPSENSE_END_OF_SCAN_E` – End of scan callback. This callback will be triggered after a full scan. That is, after all sensors have been sampled.

You can set up these callbacks by calling the function `Cy_CapSense_RegisterCallback`.

---

## Scanning

Now that you've initialized everything needed for touch sensing, you can start a scan of all your CAPSENSE™ widgets by calling the function `Cy_CapSense_ScanAllWidgets`. If you only want to scan one of your widgets rather than all of them, you can call the functions `Cy_CapSense_SetupWidget` and `Cy_CapSense_Scan` instead of `Cy_CapSense_ScanAllWidgets`.

To tell when the scan is complete you can either set a flag variable in your end of scan callback, or you can call the function `Cy_CapSense_IsBusy` to poll the CAPSENSE™ HW from your application code.

Once the scan is complete you need to process the data that was collected. This is done by calling the function `Cy_CapSense_ProcessAllWidgets`. If you only need to process the data from one widget rather than all widgets, you can call the function `Cy_CapSense_ProcessWidget` instead.

After the scan data has been processed, you can then check what touches have occurred using the following functions:

- `Cy_CapSense_IsSensorActive` – Reports whether the specified sensor detected a touch
- `Cy_CapSense_IsWidgetActive` – Reports whether the specified widget detected a touch on any of its sensors
- `Cy_CapSense_IsAnyWidgetActive` – Reports whether any widget has detected a touch
- `Cy_CapSense_IsProximitySensorActive` – Reports whether the specified proximity sensor/widget detected a conductive object in close proximity
- `Cy_CapSense_GetTouchInfo` – Reports the details of a touch's position on the specified touchpad, matrix buttons, or slider widget
- `Cy_CapSense_DecodeWidgetGestures` – Reports what gestures were detected on the specified widget

Often the process of initiating a scan, waiting for the scan to complete, processing the scan data, and acting on what touches were detected occurs in a loop within your program so that touches are continuously watched for.

## 4.3.2 CSDADC

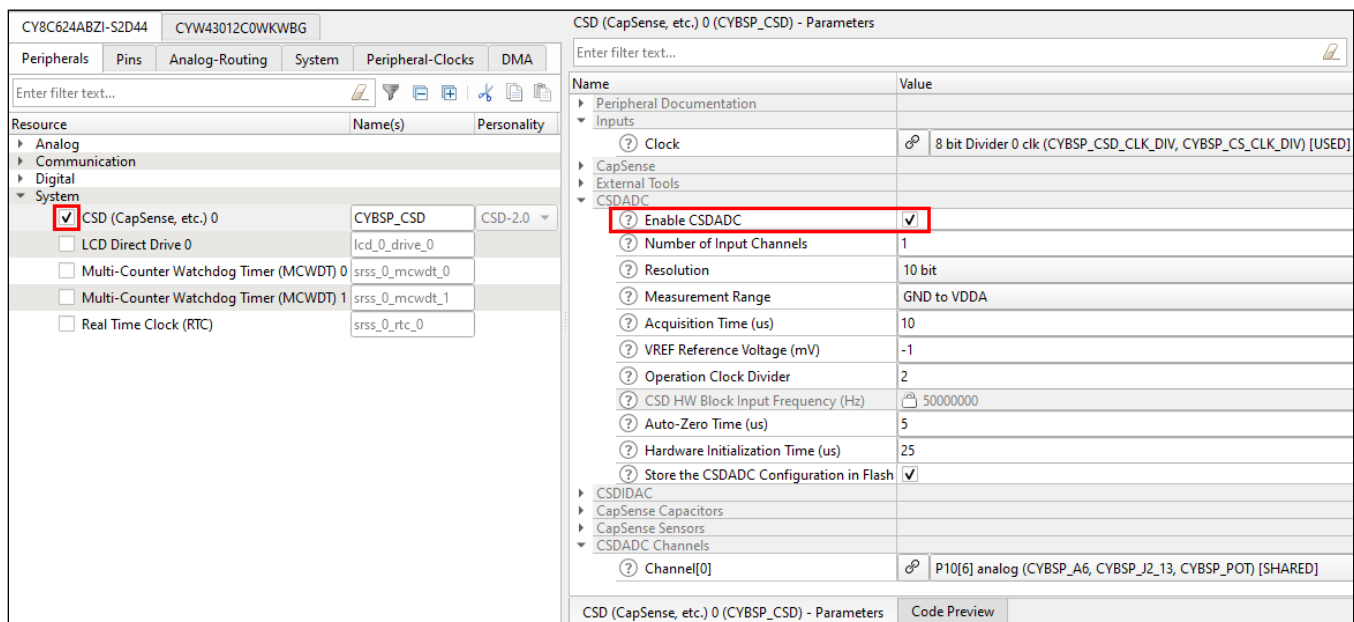
The CSD HW block contains a 10-bit ADC that is normally used in the touch sensing process but can also be used by itself. To make interacting with the CSDADC easier, Infineon has created the CSDADC Middleware Library, which provides an API that allows you to easily configure and make measurements using the CSDADC. The best place to read about the API provided by this library is the [CSDADC Middleware Library Documentation](#).

This library is distinct from the CAPSENSE™ Middleware Library, and must be added to your project separately via the Library Manager:

Name	Version
<input type="checkbox"/> anycloud-ota	5.0.0 release
<input type="checkbox"/> aws-iot-device-sdk-embedded-C	v4_beta release
<input type="checkbox"/> aws-iot-device-sdk-port	2.2.2 release
<input type="checkbox"/> azure-c-sdk-port	1.2.1 release
<input type="checkbox"/> azure-sdk-for-c	1.1.0 release
<input checked="" type="checkbox"/> capsense	3.0.0 release
<input type="checkbox"/> command-console	4.0.0 release
<input checked="" type="checkbox"/> csdadc	2.10.0 release
<input type="checkbox"/> csdidac	2.10.0 release
<input type="checkbox"/> dfu	4.20.0 release
<input type="checkbox"/> emeeprom	2.20.0 release

### 4.3.2.1 Configuration

To use the CSDADC, you first need to open the Device Configurator and verify that the CSD HW block is enabled and enable the CSDADC:



The screenshot shows the Infineon Device Configurator interface. On the left, the 'System' tab is selected, and the 'CSD (CapSense, etc.) 0' component is highlighted. On the right, the 'CSD (CapSense, etc.) 0 (CYBSP\_CSD) - Parameters' dialog is open. The 'Enable CSDADC' checkbox is checked, and the 'Store the CSDADC Configuration in Flash' checkbox is also checked. Other parameters like 'Number of Input Channels', 'Resolution', and 'Measurement Range' are visible.

Configure the CSDADC parameters how you would a regular ADC. Make sure to enable and properly configure whatever pins you connect to it.



### 4.3.2.2 Firmware

#### Initialization

For the firmware, first `#include "cy_csdadc.h"` and `"cy_pdl.h"` in your application code.

The CSDADC generates an interrupt at the end of every conversion or calculation so you need to initialize this interrupt and set up its service routine. To do this you must first set up an interrupt configuration object of type `cy_stc_sysint_t` as follows:

```
const cy_stc_sysint_t csdadc_interrut_config = {  
    .intrSrc = CYBSP_CSD_IRQ,  
    .intrPriority = CSDADC_INTR_PRIORITY  
};
```

Then you need to set up the CSD interrupt service routine. This should be a static function that only calls the function `Cy_CSDADC_InterruptHandler`:

```
static void csdadc_isr(void) {  
    Cy_CSDADC_InterruptHandler(CYBSP_CSD_HW, &cy_csdadc_context);  
}
```

Then call the functions `Cy_SysInt_Init` and `NVIC_EnableIRQ` to initialize and enable the CSDADC interrupt.

Initialize the CSD HW block by calling the function `Cy_CSDADC_Init`.

You then need to enable the CSDADC firmware modules by calling the function `Cy_CSDADC_Enable`.

Finally, you can set up a callback function that will be triggered every time a conversion finishes by calling the function `Cy_CSDADC_RegisterCallback`.

*Note: This end-of-scan callback will only be triggered by the completion of conversions that are run in continuous mode. Conversions that are run in single shot mode will NOT trigger this callback.*

#### Conversion

To start a conversion, call the function `Cy_CSDADC_StartConvert`.

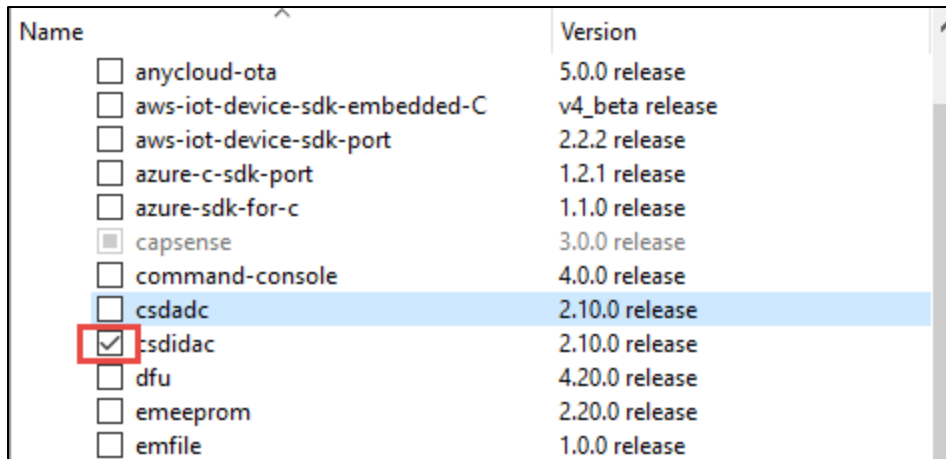
To tell when the scan is complete you can either set a flag variable in your end of conversion callback, or you can call the function `Cy_CSDADC_IsEndConversion` to poll the CSDADC HW from your application code.

Once the conversion is complete you can read the voltage that was measured by calling the function `Cy_CSDADC_GetResultVoltage`.

### 4.3.3 CSDIDAC

The CSD HW block contains a 7-bit, 2-channel current DAC that is normally used in the touch sensing process but can also be used by itself. To make interacting with the CSDIDAC easier, Infineon has created the CSDIDAC Middleware Library, which provides an API that allows you to easily configure and make conversions using the CSDIDAC. The best place to read about the API provided by this library is the [CSDIDAC Middleware Library Documentation](#).

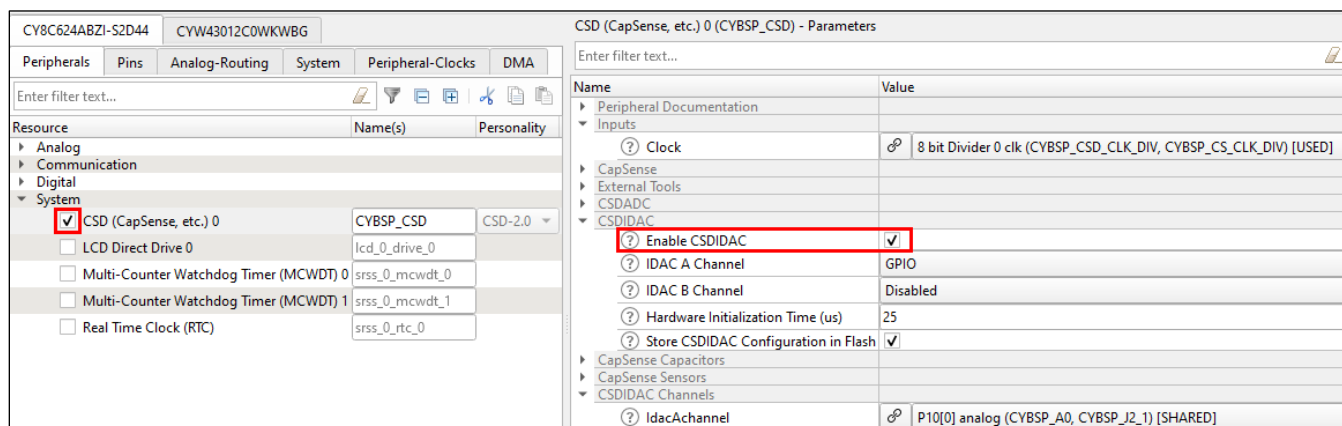
This library is distinct from the CAPSENSE™ Middleware Library, and must be added to your project separately via the Library Manager:



Name	Version
<input type="checkbox"/> anycloud-ota	5.0.0 release
<input type="checkbox"/> aws-iot-device-sdk-embedded-C	v4_beta release
<input type="checkbox"/> aws-iot-device-sdk-port	2.2.2 release
<input type="checkbox"/> azure-c-sdk-port	1.2.1 release
<input type="checkbox"/> azure-sdk-for-c	1.1.0 release
<input checked="" type="checkbox"/> capsense	3.0.0 release
<input type="checkbox"/> command-console	4.0.0 release
<input type="checkbox"/> csdadc	2.10.0 release
<input checked="" type="checkbox"/> csdidac	2.10.0 release
<input type="checkbox"/> dfu	4.20.0 release
<input type="checkbox"/> emeeprom	2.20.0 release
<input type="checkbox"/> emfile	1.0.0 release

### 4.3.3.1 Configuration

To use the CSDIDAC, you first need to open the Device Configurator and verify that the CSD HW block is enabled and enable the CSDIDAC:



Configure the CSDIDAC parameters how you want. Make sure to enable and properly configure whatever pins you connect to it.

### 4.3.3.2 Firmware

In the firmware, first `#include "cy_csdidac.h"` and `"cy_pdl.h"` in your application code.

Initialize the CSD HW block by calling the function `Cy_CSDIDAC_Init`.

Finally, you can enable the CSDIDAC outputs by calling the function `Cy_CSDIDAC_OutputEnable`.

### 4.3.4 Time multiplexing

If your application requires it, you can use the CSD HW block for more than one function (touch sensing, ADC or IDAC) in the same application. To do this you will need to call the time multiplexing functions provided by the appropriate Middleware Library:

- `Cy_CapSense_Save`, `Cy_CSDADC_Save`, `Cy_CSDIDAC_Save`, – These functions save the current configuration of the CSD HW block, CAPSENSE™ middleware, CSDADC, or CSDIDAC and also perform the following actions:
  - Configure pins to the default state and disconnect them from analog buses
  - Disconnect external capacitors from analog buses
  - Set the middleware state to default
  - Release the CSD HW block
  - Release any associated interrupt vectors
- `Cy_CapSense_Restore`, `Cy_CSDADC_Restore`, `Cy_CSDIDAC_Restore` – These functions restore the CSD HW block to the state it was in when the corresponding save function was previously called

*Note:* The restore functions do NOT restore the interrupt vectors. After calling a restore function, you will need to call `Cy_SysInt_Init` to restore the interrupt vector.

The basic flow of an application that uses more than one of the CSD HW block functionalities will look something like this:

1. Initialize CSD HW block for touch sensing
2. Call `Cy_CapSense_Save` to save the initial touch sensing configuration
3. Initialize the CSD HW block for ADC
4. Call `Cy_CSDADC_Save` to save the initial ADC configuration
5. Call `Cy_CapSense_Restore` and `Cy_SysInt_Init` to restore the touch sensing configuration
6. Make touch measurements
7. Call `Cy_CapSense_Save` to save the current touch sensing configuration and release resources
8. Call `Cy_CSDADC_Restore` and `Cy_SysInt_Init` to restore the ADC configuration
9. Make ADC measurements
10. Call `Cy_CSDADC_Save` to save the current ADC configuration and release resources
11. Repeat steps 5-10

## 4.4 CAPSENSE™ Tuner

The CAPSENSE™ Tuner is a stand-alone tool included in the ModusToolbox™ tools package. It provides you with a GUI that allows you to easily connect to, monitor, and tune your CAPSENSE™ applications in real time. You can launch the CAPSENSE™ Tuner from the Tools section of Eclipse IDE for ModusToolbox™ Quick Panel. From the CLI, the tuner can be launched from an application directory using the command `make open CY_OPEN_TYPE=capsense-tuner`.

In this section we will take a broad and shallow look at the CAPSENSE™ Tuner. For more information, you should refer to the CAPSENSE™ Tuner User Guide. You can access this guide directly from the CAPSENSE™ Tuner by going to **Help > View Help**.

### 4.4.1 Device-Tuner communication interface

Before you use the CAPSENSE™ Tuner, you must first create a CAPSENSE™ application with a proper tuner communication interface. The tuner supports both I2C and UART communication interfaces. In this section we will cover setting up an I2C tuner communication interface using both the PDL and the HAL. As usual, the HAL is much simpler so if you have the HAL available, that's the easier approach.

The I2C tuner communication interface can most easily be set up as an EZI2C slave with the following parameters:

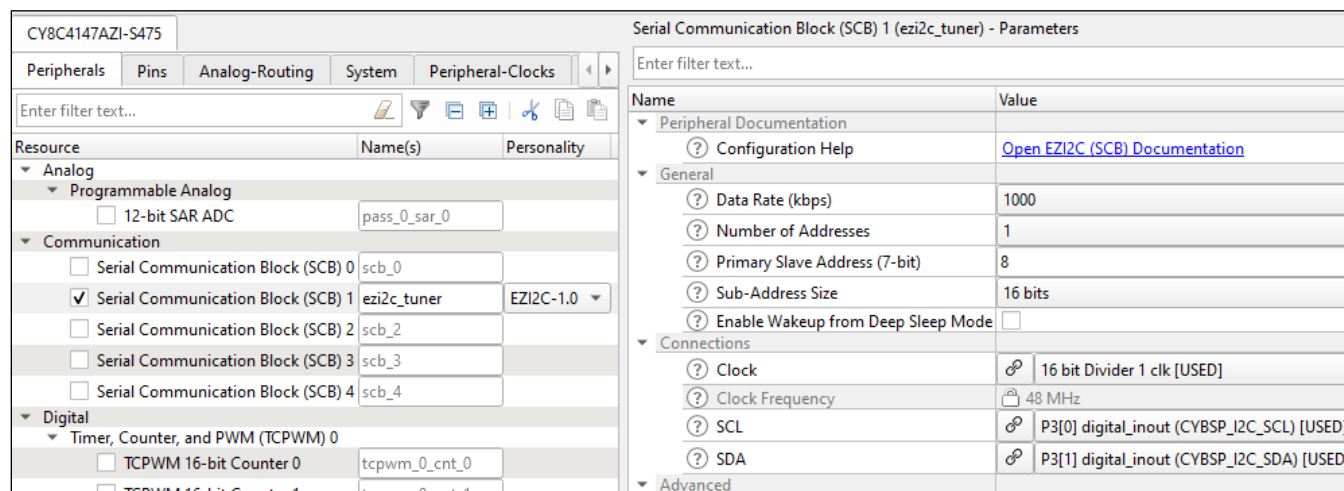
- Slave Address: 80
- Data Buffer: `(uint8 *) &cy_capsense_tuner`
- Buffer Size: `sizeof(cy_capsense_tuner)`
- Buffer R/W Boundary: `sizeof(cy_capsense_tuner)`

`cy_capsense_tuner` is an object of type `cy_stc_capsense_tuner_t`. It is defined in: *bsps/<Your BSP>/config/GeneratedSource/cycfg\_capsense.c*. This file is part of the code that is automatically generated by the CAPSENSE™ Configurator.

#### 4.4.1.1 PDL

##### Configuration

Open the Device Configurator and enable the SCB that can connect to your kit's I<sup>2</sup>C SCL and SDA pins (on the CY8CKIT-149 this is SCB1). Configure the SCB as an EZI2C slave with the following parameters:



**Note:** The sub-address size is set to 16 bits. This allows EZI2C to provide up to 65536 memory locations instead of just 256.

Make sure to enable and properly configure the SCL and SDA pins.

Do **File > Save**, and exit the device configurator.

##### Firmware

In your application code, if you haven't already, `#include "cycfg.h"` and `"cycfg_capsense.h"`.

Initialize the EZI2C block by calling the function `Cy_SCB_EZI2C_Init`. Initialize the EZI2C interrupt by calling the function `Cy_SysInt_Init`. As an argument, this function takes an interrupt configuration object of type `cy_stc_sysint_t`, which should be configured as follows:

```
const cy_stc_sysint_t ezi2c_intr_config = {
    .intrSrc = ezi2c_tuner_IRQ,
    .intrPriority = EZI2C_INTR_PRIORITY,
};
```

**Note:** It is important the priority of the tuner I2C interrupt (`EZI2C_INTR_PRIORITY`) is higher than the priority of the CSD interrupt (lower numbers correspond to higher priorities).

The ISR you pass to this function should only call the function `Cy_SCB_EZI2C_Interrupt` and then exit:

```
static void ezi2c_isr(void) {
    Cy_SCB_EZI2C_Interrupt(ezi2c_tuner_HW, &ezi2c_context);
}
```

Enable the interrupt by calling the function `NVIC_EnableIRQ`, then, set up the EZI2C slave's data buffer by calling the function `Cy_SCB_EZI2C_SetBuffer1` as follows:

```
Cy_SCB_EZI2C_SetBuffer1(ezi2c_tuner_HW, (uint8 *)&cy_capsense_tuner,  
                        sizeof(cy_capsense_tuner), sizeof(cy_capsense_tuner),  
                        &ezi2c_context);
```

Finally, enable the EZI2C firmware module by calling the function `Cy_SCB_EZI2C_Enable`.

There are two modes that the tuner can run in. Usually the synchronous mode is used.

- Asynchronous - The tuner reads data asynchronously with sensor scanning and data processing. Data coherency may be corrupted, as the tuner may read only partially updated sensor data. For example, the tuner may start a read while the CAPSENSE™ firmware is still performing sensor scans. Measurements made in this mode will be less accurate than measurements made in synchronous mode. However, no changes are required to the firmware to run in asynchronous mode.
- Synchronous – The tuner synchronizes data reading and firmware execution to preserve data coherency. The CAPSENSE™ middleware will wait for the tuner's read/write operations to complete before beginning sensor scans or data processing tasks. You must periodically call the function `Cy_CapSense_RunTuner` in your application. A good time to do this is after processing the data from a scan.

After programming this firmware into your kit, you should be able to connect to it via the CAPSENSE™ Tuner.

#### 4.4.1.2 HAL

##### Firmware

In your application code `#include "cycfg_capsense.h"`.

Next you need to configure a `cyhal_ezi2c_slave_cfg_t` object and a `cyhal_ezi2c_cfg_t` object as follows:

```
cyhal_ezi2c_slave_cfg_t ezi2c_slave_config;  
cyhal_ezi2c_cfg_t ezi2c_config;  
  
// Configure Capsense Tuner as EzI2C Slave  
ezi2c_slave_config.buf = (uint8 *)&cy_capsense_tuner;  
ezi2c_slave_config.buf_rw_boundary = sizeof(cy_capsense_tuner);  
ezi2c_slave_config.buf_size = sizeof(cy_capsense_tuner);  
ezi2c_slave_config.slave_address = 8U;  
  
ezi2c_config.data_rate = CYHAL_EZI2C_DATA_RATE_1MHZ;  
ezi2c_config.enable_wake_from_sleep = false;  
ezi2c_config.slave1_cfg = ezi2c_slave_config;  
ezi2c_config.sub_address_size = CYHAL_EZI2C_SUB_ADDR16_BITS;  
ezi2c_config.two_addresses = false;
```

**Note:** *The sub-address size is set to 16 bits. This allows EZI2C to provide up to 32768 memory locations instead of just 256.*

Then simply call the function `cyhal_ezi2c_init` with the configuration objects you just populated and the correct I<sup>2</sup>C pins.



There are two modes that the tuner can run in. Usually the synchronous mode is used.

- **Asynchronous** - The tuner reads data asynchronously with sensor scanning and data processing. Data coherency may be corrupted, as the tuner may read only partially updated sensor data. For example, the tuner may start a read while the CAPSENSE™ firmware is still performing sensor scans. Measurements made in this mode will be less accurate than measurements made in synchronous mode. However, no changes are required to the firmware to run in asynchronous mode.
- **Synchronous** - The tuner synchronizes data reading and firmware execution to preserve data coherency. The CAPSENSE™ middleware will wait for the tuner's read/write operations to complete before beginning sensor scans or data processing tasks. You must periodically call the function `Cy_CapSense_RunTuner` in your application. A good time to do this is after processing the data from a scan.








After programming this firmware into your kit, you should be able to connect to it via the CAPSENSE™ Tuner.





## 4.4.2 Toolbar

The CAPSENSE™ Tuner has a toolbar at the top of it that looks like this:



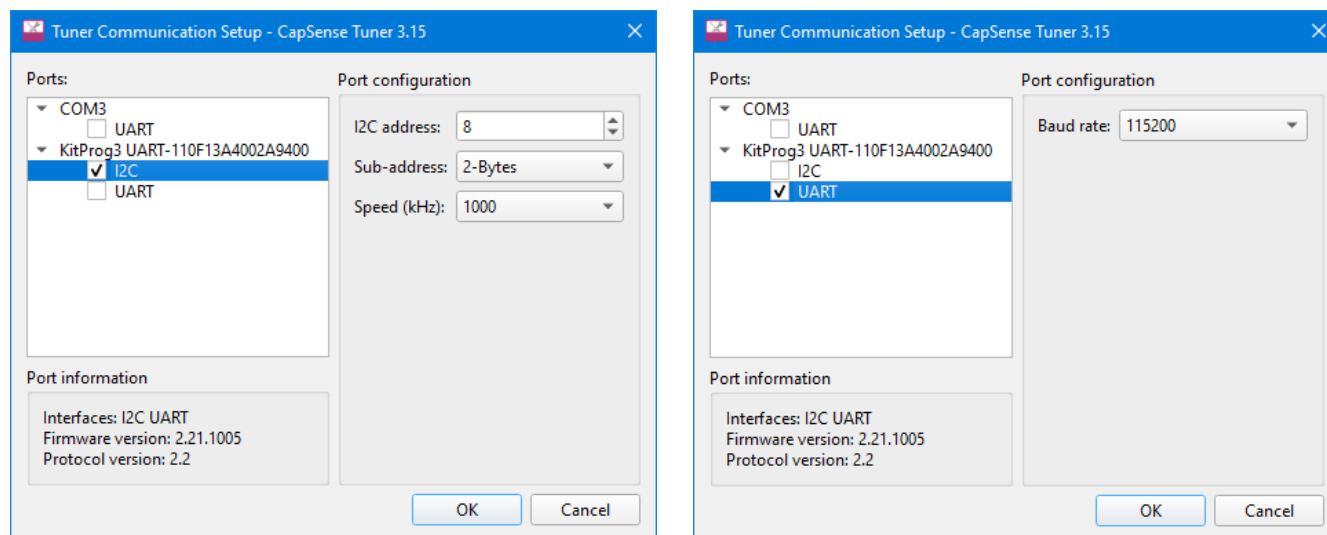
It has icons that allow you to run commonly used commands by clicking them:

- **Tuner Communication Setup**  - This allows you to set up a communication channel with a device
- **Connect to Device**  - This will attempt to connect to a device as you have configured it in Tuner Communication Setup
- **Start Reading from Device**  - This starts reading CAPSENSE™ data from the connected device (This button is greyed out until a device is connected)
- **Open Configuration**  - This allows you to open a CAPSENSE™ configuration (.cycapsense file) (This button is greyed out unless the tuner is launched in stand-alone mode)
- **Apply Configuration to Device**  - This applies any CAPSENSE™ configuration changes you have made to the connected device. (This button is greyed out if there is no device connected, or if there are no new configuration changes to apply)
- **Apply Configuration to Project**  - This saves the configuration settings in the tuner's workspace to the open CAPSENSE™ configuration.
- **Import Configuration**  - This copies the configuration settings from a specified configuration file into the tuner's workspace. This does NOT open the specified configuration file, it merely copies its configuration settings into the tuner's workspace.

- Export Configuration  - This saves the configuration settings in the tuner's workspace to a new CAPSENSE™ configuration file. Configuration settings do not have to be previously saved to a configuration file for you to export them.
- Start/Stop Logging  - This allows you to start and stop tuner logging
- Read Mode – This allows you to switch between asynchronous and synchronous read modes
  - Asynchronous - The tuner reads data asynchronously with sensor scanning and data processing. Data coherency may be corrupted, as the tuner may read only partially updated sensor data. For example, the tuner may start a read while the CAPSENSE™ firmware is still performing sensor scans. Measurements made in this mode will be less accurate than measurements made in synchronous mode.
  - Synchronous – The tuner synchronizes data reading and firmware execution to preserve data coherency. The CAPSENSE™ middleware will wait for the tuner's read/write operations to complete before beginning sensor scans or data processing tasks. This mode requires the function `Cy_CapSense_RunTuner` to be called periodically from the application code.
- Undo  - This allows you to undo the last action or sequence of actions
- Redo  - This allows you to redo the last undone action or sequence of undone actions

### 4.4.3 Tuner communication setup

Clicking the Tuner Communication Setup button  in the toolbar will bring up the following dialog:



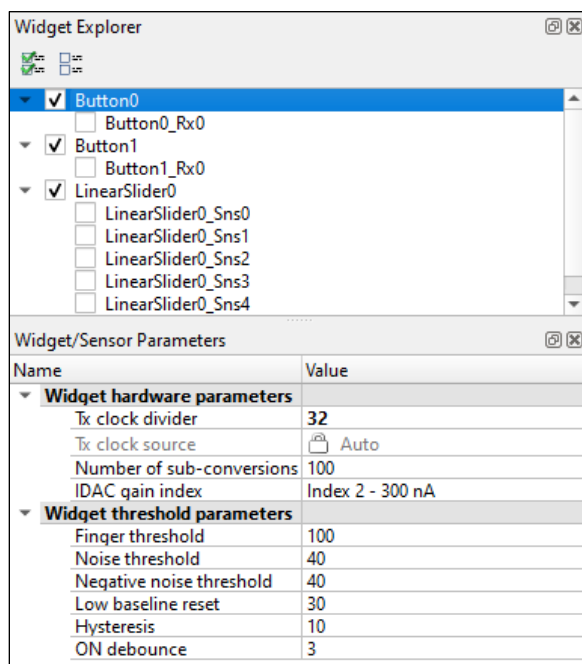
Here you can configure the communication channel that the tuner will use to connect to your device. If you followed the guide in the [Device-Tuner Communication Interface](#) section of this chapter, you will need to configure your tuner as shown on the left.

**Note:** *You cannot program your device while the tuner is connected. To program to your device, you must first disconnect the tuner.*

**Note:** *The tuner will not be able to connect to your device if you already have a serial debugging connection open.*

#### 4.4.4 Widget/Sensor Explorer and Parameters panes


Regardless of what tab you are looking at, on the left side of the CAPSENSE™ Tuner you will see the Widget Explorer and the Widget/Sensor Parameters panes:



**Note:** If you don't see them, you can show them by clicking **View**, and then checking the boxes next to **Widget Explorer** and **Widget/Sensor Parameters**.

The **Widget Explorer** pane contains a tree of all the widgets and sensors used in the CAPSENSE™ application. You can expand/collapse the Widget nodes to show/hide each widget's sensor nodes. You can also check/uncheck individual widgets and sensors. When you check/uncheck a widget, it will add/remove that widget from the **Widget View** tab. When you check/uncheck a sensor, it will add/remove that sensor from the **Graph View** tab and the Touch Signal Graph in the **Widget View** tab.

Clicking on an element in the **Widget Explorer** will pull up that element's parameters in the **Widget/Sensor Parameters** pane. In the **Widget/Sensor Parameters** pane you can edit the selected element's parameters simply by clicking on them and adjusting them.

**Note:** Parameters that are shown with a lock symbol  cannot be edited from the CAPSENSE™ Tuner.

**Note:** You must "Apply to Device" before an updated parameter will take effect.

**Note:** You must "Apply to Project" if you want parameters you changed in the tuner to be saved in the open CAPSENSE™ configuration.

## 4.4.5 Tabs

This section describes each of the tabs in the CAPSENSE™ Tuner.

### 4.4.5.1 Widget View

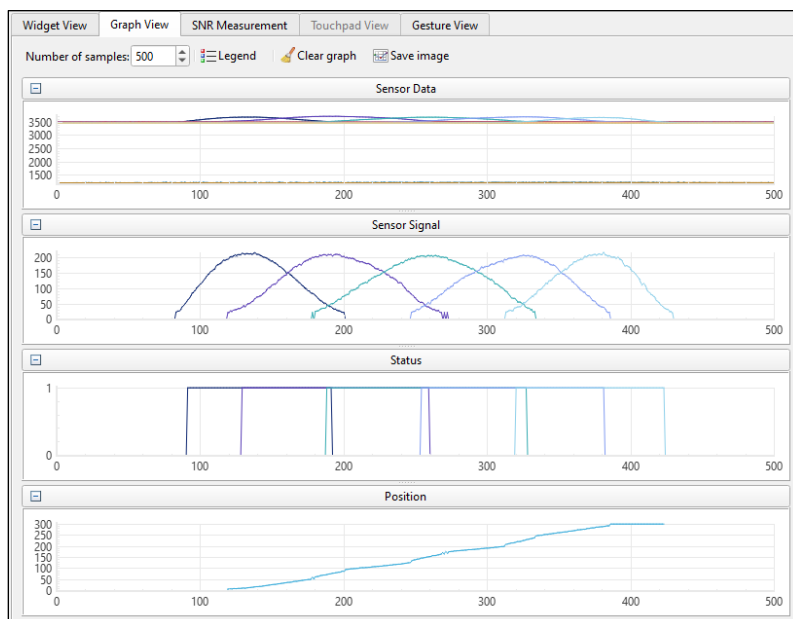
When you first launch the CAPSENSE™ Tuner you will see the **Widget View** tab, which looks like this:



Here you will see icons representing any of the widgets you have checked in the **Widget Explorer** pane. You will also see the Touch Signal Graph, which displays the touch signals of any sensors that you have checked in the **Widget Explorer** pane. You can save a screenshot of the Touch Signal Graph at any time by clicking the **Save Image** button. Whenever you touch one of your CAPSENSE™ elements, you will see the corresponding icon light up, and its touch signal appear on the Touch Signal Graph.

### 4.4.5.2 Graph View

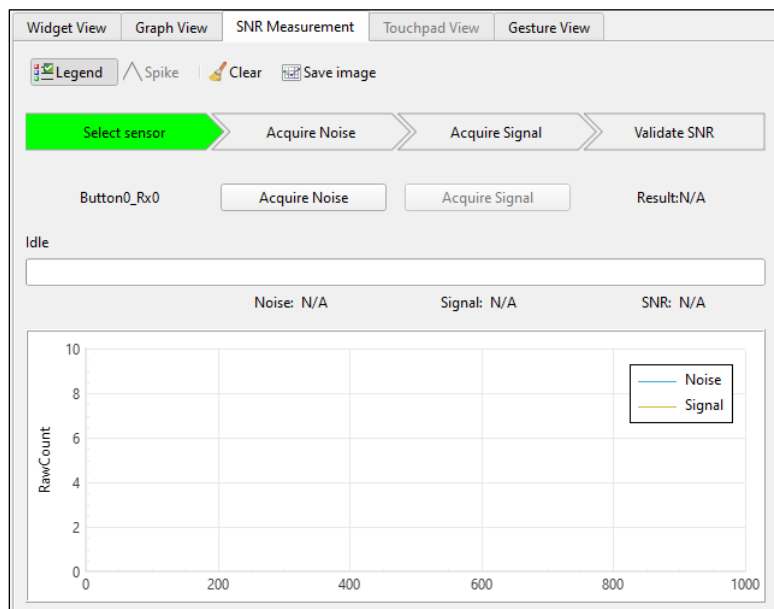
If you click on the **Graph View** tab, you will see a window that looks like this:



Here you will see graphs of all of the relevant data from the sensors you have checked in the **Widget Explorer** pane. You can save a screenshot of all of the open graphs at any time by clicking the **Save Image** button.

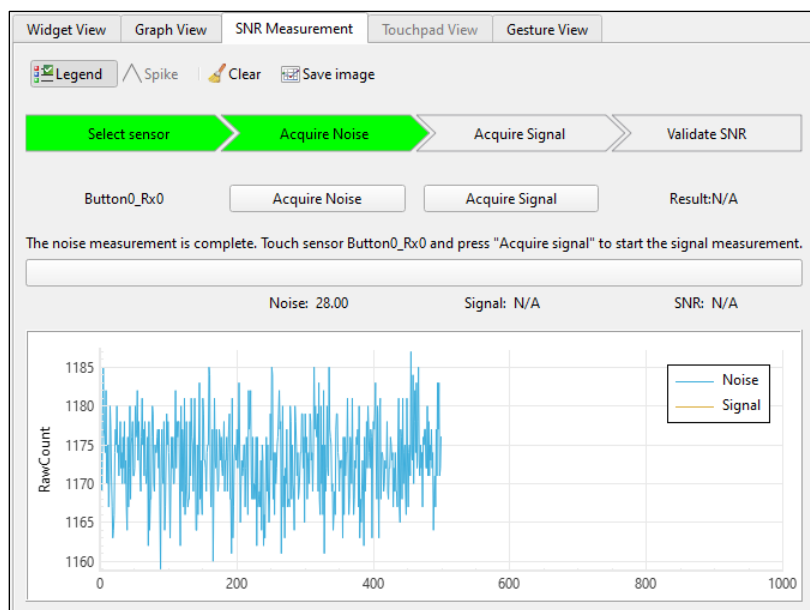
#### 4.4.5.3 SNR Measurement

If you click on the **SNR Measurement** tab, you will see a window that looks like this:



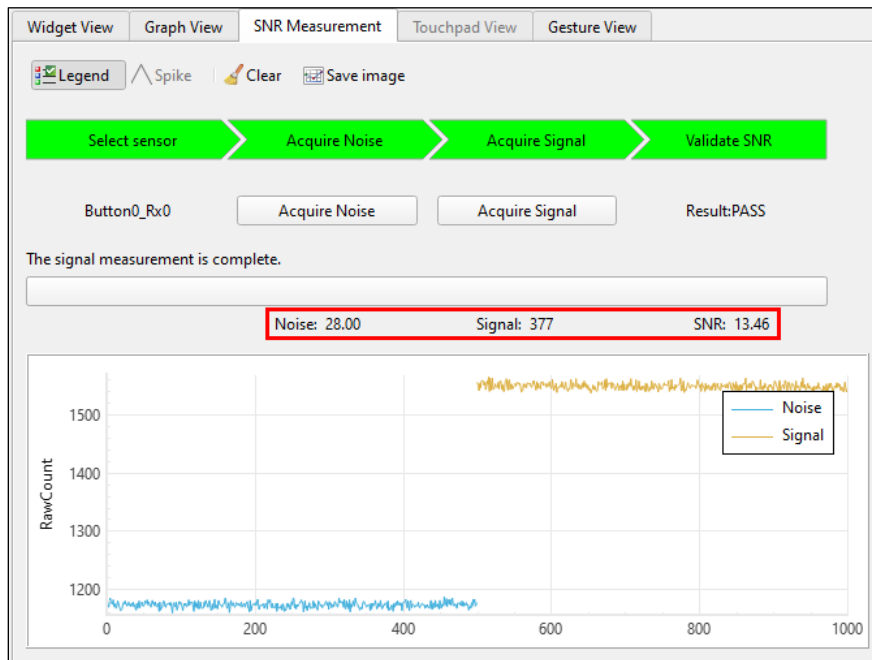
Here you can measure the signal to noise ratio of any of the sensors in the **Widget Explorer** pane. Simply select the sensor whose signal to noise ratio you want to measure from the **Widget Explorer** by clicking it. Then click the **Acquire Noise** button. This will cause the tuner will take a reading of the noise level on the sensor.

**Note:** *To ensure a high accuracy reading it is important that you DO NOT touch ANY of the CAPSENSE™ sensors during the noise reading.*



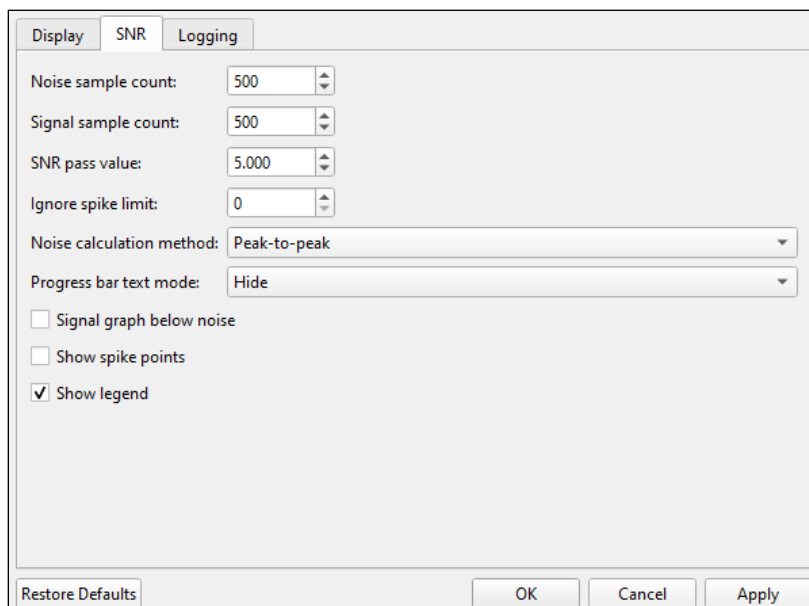
After the noise reading is complete, place your finger on the sensor you are measuring and click the **Acquire Signal** button. This will cause the tuner to take a reading the signal level on the sensor.

*Note:* To ensure a high accuracy reading it is important that you **DO NOT** remove your finger from the sensor until the reading is complete.



Once the signal reading is complete, you will see the noise level, the signal level, and the signal to noise ratio displayed. You can save a screenshot of the graph displayed in this tab at any time by clicking the **Save Image** button.

There are SNR measurement parameters you can edit by going to **Tools > Options > SNR**:



The screenshot shows the 'SNR' tab in the 'Tools > Options' dialog box. The 'Display' tab is selected. The 'SNR' tab contains the following settings:

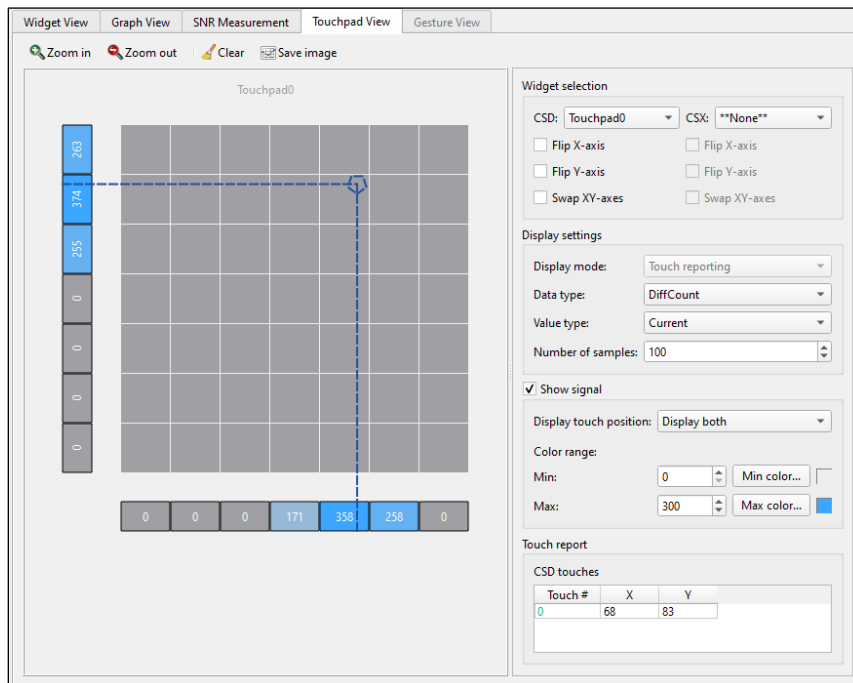
- Noise sample count: 500
- Signal sample count: 500
- SNR pass value: 5.000
- Ignore spike limit: 0
- Noise calculation method: Peak-to-peak
- Progress bar text mode: Hide
- ☐ Signal graph below noise
- ☐ Show spike points
- ☒ Show legend

At the bottom of the dialog box, there are buttons for 'Restore Defaults', 'OK', 'Cancel', and 'Apply'.



#### 4.4.5.4 Touchpad View

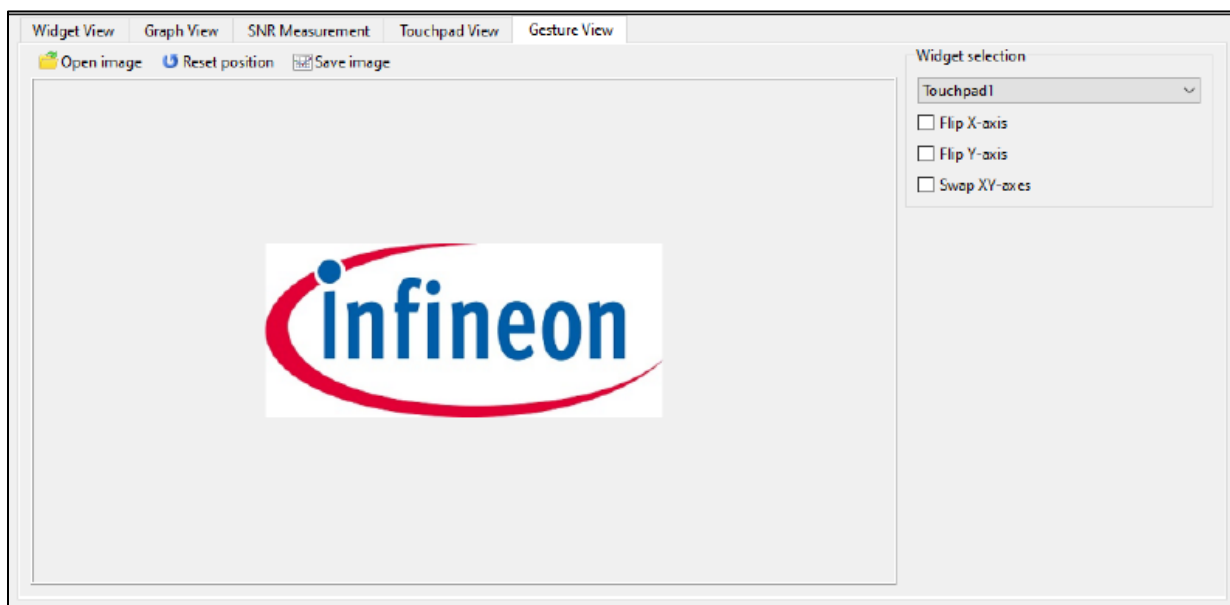
If you click on the **Touchpad View** tab, you will see a window that looks like this:



Here you can see a "heatmap" representation of touch signals from a selected touchpad. Only one CSD and one CSX touchpad can be displayed at a time. This tab is greyed out unless there is a touchpad widget in the configuration.

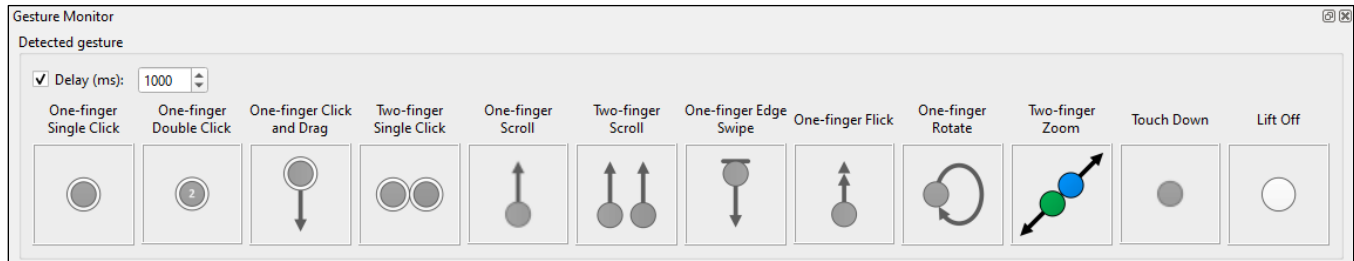
#### 4.4.5.5 Gesture View

If you click on the **Gesture View** tab, you will see a window that looks like this:



Here you will see an image displayed (Infineon Logo by default) that will react to scroll and zoom gestures that you input on your device. You can change what image is displayed using the **Open image** button.

There is also a **Gesture Monitor** pane that will show you what gestures are being detected in real time:



**Note:** This pane is not displayed by default to show it, click **View**, then check the box next to **Gesture Monitor**.

## 4.5 Exercises

### Exercise 1: Control LED

This exercise uses the CY8CKIT-062S2-43012 or CY8CPROTO-062-4343W.

- ☐ 1. Use Project Creator to create a new application called **ch04\_ex01\_controlLED** using **Empty App** as the template.
- ☐ 2. Follow the outline in the [Touch Sensing](#) section of this chapter to enable the use of the CAPSENSE™ buttons and slider on your kit.

*Note:* Don't forget to add the capsense middleware library to the application using the Library Manger.

*Note:* The CAPSENSE™ configuration is already set up in the CAPSENSE™ Configurator for the kit, so you don't need to make any changes but you should review them.

*Note:* The CAPSENSE™ Middleware Library Documentation can be accessed from the Documentation section of the Eclipse IDE for ModusToolbox™ Quick Panel or from the library's docs directory.

- ☐ 3. Initialize a PWM to drive `CYBSP_USER_LED`.
  - Whenever CAPSENSE™ button 0 is pressed, start the PWM.
  - Whenever CAPSENSE™ button 1 is pressed, stop the PWM.
  - Whenever the CAPSENSE™ slider is touched, set the duty of the cycle of the PWM so that it is proportional to where the slider was touched.
- ☐ 4. Program your project to your kit and verify the behavior.
  - When CAPSENSE™ button 0 is pressed the LED should turn on.
  - When CAPSENSE™ button 1 is pressed the LED should turn off.
  - You should be able to adjust the brightness of the LED using the CAPSENSE™ slider.

### Exercise 2: Add a Tuner communication interface to the LED control project

This exercise uses the CY8CKIT-062S2-43012 or CY8CPROTO-062-4343W.

- ☐ 1. Use Project Creator to create a new application called **ch04\_ex02\_controlLED\_tuner** using the **Browse** button on the **Select Application** page to select your previous exercise (ch04\_ex01\_controlLED) as a template.
- ☐ 2. Follow the outline in the [Device-Tuner Communication Interface](#) section of this chapter to enable the CAPSENSE™ Tuner to connect to your kit.

*Note:* Using the HAL with EZI2C is by far the simplest way to do this.

*Note:* Remember that the I2C pins for the kit have the aliases `CYBSP_I2C_SCL` and `CYBSP_I2C_SDA`.

- ☐ 3. Program your project to your kit and open the CAPSENSE™ Tuner using the link in the Tools section of the Eclipse IDE for ModusToolbox™ Quick Panel.
- ☐ 4. Connect the CAPSENSE™ Tuner to your kit, and observe the widgets being pressed in real time.

## Exercise 3: Time multiplexing

This exercise uses the CY8CKIT-062S2-43012. The CY8CPROTO-062-4343W does not have a potentiometer or a 2<sup>nd</sup> user LED so it is not compatible with this exercise.

- ☐ 1. Use Project Creator to create a new application called **ch04\_ex03\_multiplex** using the **Browse** button on the **Select Application** page to select your previous exercise (ch04\_ex02\_controlLED\_tuner) as a template.
- ☐ 2. Follow the outline in the [CSDADC](#) section of this chapter to enable your kit to use the CSDADC to read the voltage on your kit's potentiometer.
- ☐ 3. Setup a second PWM to drive CYBSP\_USER\_LED2.
- ☐ 4. Set the duty cycle of the second PWM use the potentiometer.

*Note: Be sure to use the proper time multiplexing functions in your application code when switching between touch sensing and ADC functionalities of the CSD HW block.*

- ☐ 5. Program your project to your kit and verify the behavior.

The CAPSENSE™ slider continues to control the brightness of CYBSP\_USER\_LED while the potentiometer controls the brightness of CYBSP\_USER\_LED2. The CAPSENSE™ buttons still turn the LEDs on and off.

**Published by**  
**Infineon Technologies AG**  
**81726 Munich, Germany**

**© 2023 Infineon Technologies AG.**  
**All Rights Reserved.**

#### **IMPORTANT NOTICE**

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office ([www.infineon.com](http://www.infineon.com)).

#### **WARNINGS**

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.