SmartSense Dual-Channel Datasheet SmartSense2X V 1.10

Resources	PSoC® Blocks				API Memory (Bytes)		Pins (in addition to sensors)
	CapSense®	I²C/SPI	Timer	Comparator	Flash	RAM	
CY8C22x45, CY8C21x45							
Single channel (one button)	1	0	–	1			1+1 optional
Threshold settings (manual)					1813	38	
With auto-tune to sensitivity					2919	48	
With auto-tune to noise					2936	48	
Double channel (one button)	2	–	–	2			2+2 optional
Threshold settings (manual)					2370	41	
With auto-tune to sensitivity					3485	51	
With auto-tune to noise					3493	51	
Each additional CapSense button (manual threshold settings)	–	–	–	–	7	6	1
Static code and RAM increase when using capacitive slider with 5 elements (manual threshold settings)	–	–	–	–	676	112	5
Each additional slider segment (manual)	–	–	–	–	8	16	1
With auto-tune to noise	2	–	–	–	7	26	–
With auto-tune to sensitivity	2	–	–	–	7	26	–
Static code and RAM increase during slider diplexing (5 sensors for manual threshold settings)	–	–	–	–	25	–	–

Features and Overview

- Dual-channel SmartSense™.
- Manual and auto threshold setting for finger detection.
- Configurable sensitivity for each sensor.
- Background scan.
- Autoreset and button debounce.
- Buttons, sliders, and proximity sensors.
- Enables guided sensor and pin assignments using the SmartSense2X wizard.

Introduction

The SmartSense2X User Module implements CapSense capacitive sensing. CapSense is a human-interface technology that operates by detecting the capacitance of the human body. This is done using sensors that have a conductive surface, usually a pad etched on the printed circuit board (PCB). Because CapSense detects body capacitance, it can sense through insulating layers, such as plastic or glass overlays. These overlays usually constitute the external enclosure of the end product. These attributes make CapSense an attractive alternative to mechanical input devices, such as push buttons and potentiometers. The major benefits of a CapSense user interface are:

- Improved reliability; none of the components wear or have finite cycle life
- Improved spill-resistance due to lack of mechanical interface penetrations
- Cleaner and more aesthetically pleasing designs
- Reduced form-factor possibilities for smaller end products.
- Addition of advanced user interface features, such as LED effects and proximity sensing.
- Reduced tooling cost by eliminating penetrations or other mechanical features needed for mechanical input devices.

Similar to all Cypress CapSense solutions, SmartSense auto-tuning offers superior immunity to conducted and radiated noise interference, with the additional benefit of auto-tuning. Auto-tuning gives run-time compensation for IC and PCB characteristics, and environmental changes to ensure reliable sensor operation. For example, prototype and production PCBs frequently exhibit different material properties that affect sensor parasitic capacitance. This effect can be significant enough to require retuning of the CapSense system parameters. Auto-tuning automatically compensates for such changes, with no need for retuning. Furthermore, the auto-tuning algorithms in SmartSense continuously monitor sensor data to compensate for environment conditions, such as temperature and ambient noise level, to maintain proper sensor function.

The SmartSense2X User Module consists of PCB level, IC level, and software components.

PCB Level

Figure 1 and Figure 2 show the SmartSense2X User Module schematics for single and double channels. The physical sensor is typically a conductive pattern constructed on a PCB connected to a PSoC I/O pin

with an insulating overlay. See the design guide, Getting Started with CapSense, for more information on PCB level CapSense implementation.

Figure 1. SmartSense2X Single-channel Schematic

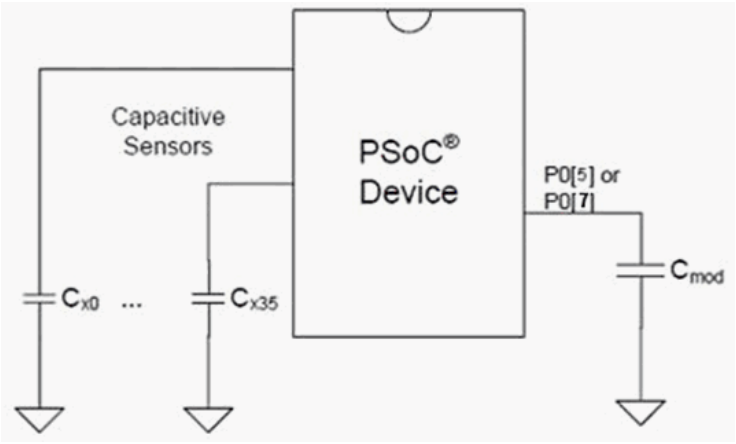
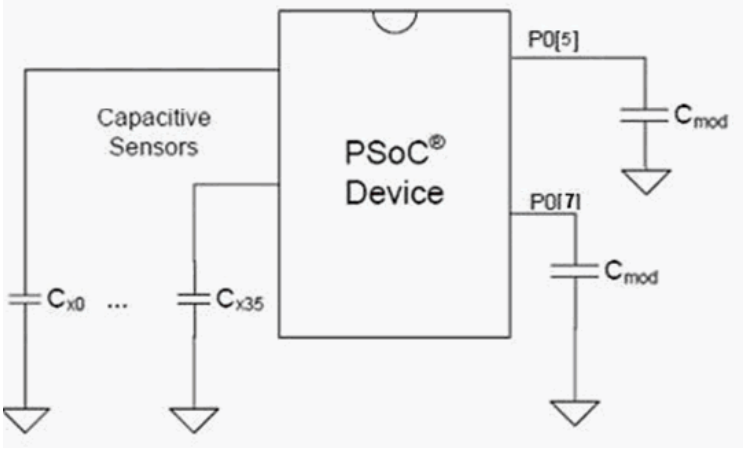


Figure 2. SmartSense2X Double-Channel Schematic



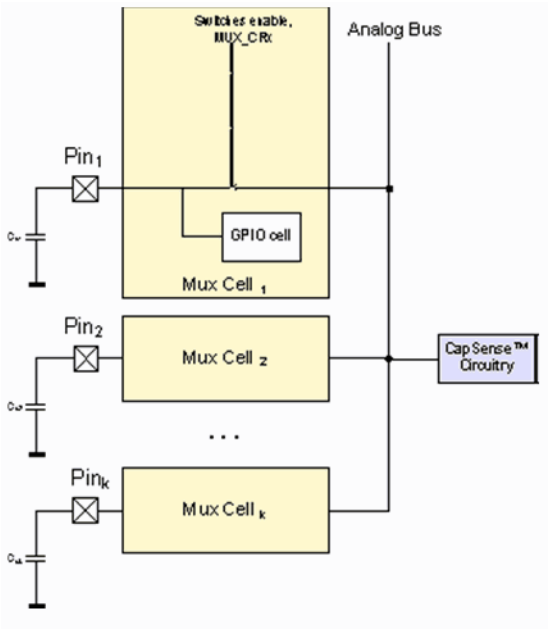
IC Level

The SmartSense2X User Module enables the capacitive sensing algorithm using a sigma-delta modulator with many of the CSD parameters configured automatically and dynamically during runtime and power-on-reset (POR). The SmartSense2X User Module's hardware use is similar to CSD2X, except for the addition of a software layer, which includes the proprietary SmartSense algorithm with background scanning. This user module is available in the CY8C22x45 and CY8C21x45 devices.

The CY8C22x45 and CY8C21x45 devices have an analog mux (AMUX) bus that allows connecting capacitive sensing analog circuitry to any PSoC pin. The SmartSense2X User Module connects the active sensor to the AMUX bus, enabling the CapSense circuitry to measure its capacitance and translate that

capacitance into a digital code. The firmware serially scans the sensors by sequentially setting corresponding bits in the MUX_CRx registers. This is represented in Figure 3.

Figure 3. AMUX Block Diagram



Software

The SmartSense2X software component has the following attributes:

- Auto-tuning algorithms configure the analog capacitive sensing circuitry in runtime for optimal performance. These algorithms take into account physical sensor characteristics, IC characteristics, and the Sensor Sensitivity user module parameter.
- The raw count value from the capacitance converter circuitry is analyzed in runtime by API functions to make sensor state decisions and to compensate for environmental changes.
- Background scanning algorithm enables the CPU to perform other tasks while the scan is in progress in hardware.
- In the case of consecutive, dependent sensors (for example, sliders and touchpads), API functions are given to interpolate a position with greater resolution than the physical pitch of the sensors.
- High-level software functions accommodate slider diplexing so that one I/O pin can be routed to two physical sensors. This reduces by half the number of I/O pins consumed for a specific number of slider elements.

Recommended Reading

Cypress recommends reading the following documents before implementing a CapSense design using the SmartSense2X User Module. These documents are available at the Cypress website: www.cypress.com.

- Getting Started with CapSense
- CY8C20xx6A/H CapSense Design Guide
- CY8C21x34/B CapSense Design Guide
- CY8C20x34 CapSense Design Guide
- CY8CMBR2044 CapSense Design Guide
- AN2397 – CapSense Data Viewing Tools

Capacitance Sensing Implementation

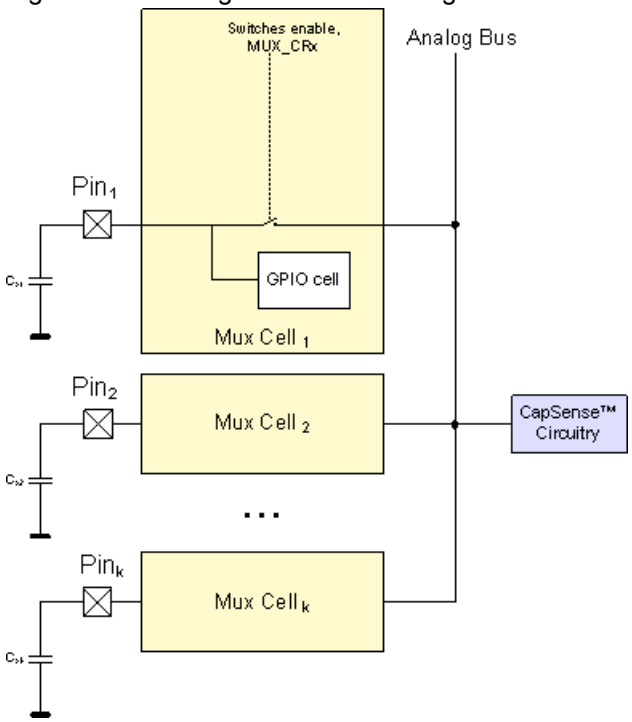
The decision logic is implemented in firmware. The firmware analyzes capacitance measurement, tracks the slow capacitance change due to environmental factors, and runs decision logic to detect button touches and calculate slider position.

Scanning an Array of Sensors

The CY8C22x45 and CY8C21x45 family of devices has a built- in analog bus, which allows capacitive sensor connections to any PSoC pin. The SmartSense2x User Module uses internal precharge switches to charge active sensors. The CapSense circuitry, internal to the PSoC, scans the sensors.

The firmware performs sensor scanning in series by setting corresponding bits in the MUX_CRx registers.

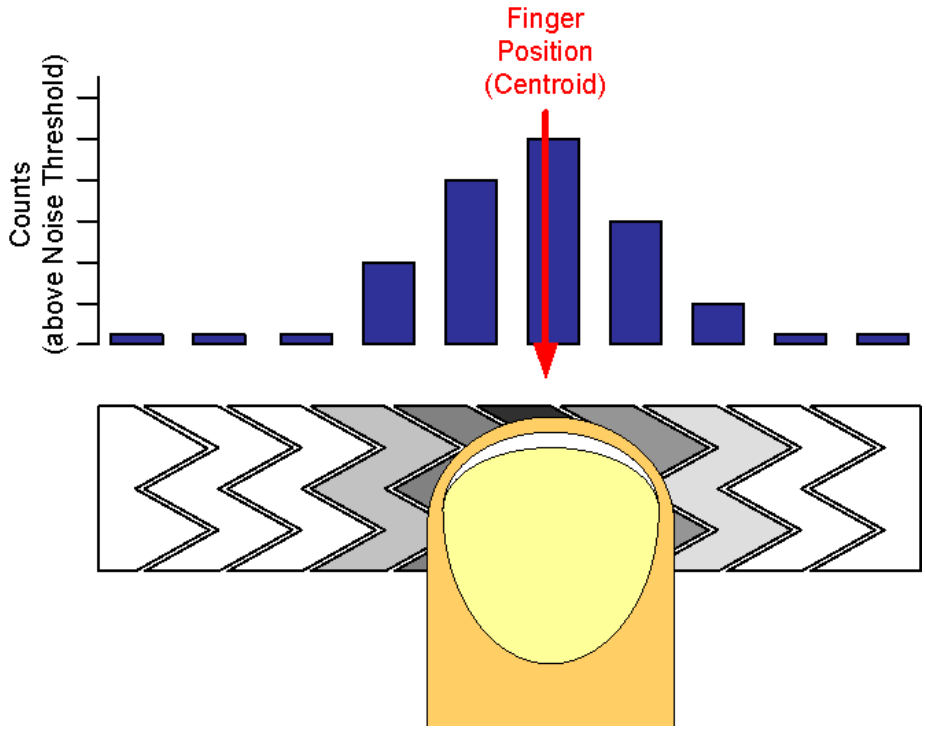
Figure 4. Analog Bus with Precharge Switches



Sliders

Sliders are used for controls that require gradual adjustments. Examples include a lighting control (dimmer), volume control, graphic equalizer, and speed control. These sensors are mechanically adjacent to one another. Actuating one sensor results in partial actuation of physically adjacent sensors. The actual position in the slider is found by computing the centroid location of the set of activated sensors. Sliders are accommodated in the CSD2x wizard by establishing groups, in which each group of sliders has a specific order. The practical lower limit number for sensor sliders is five, while the upper limit is simply the number of sensor positions available on the selected PSoC device.

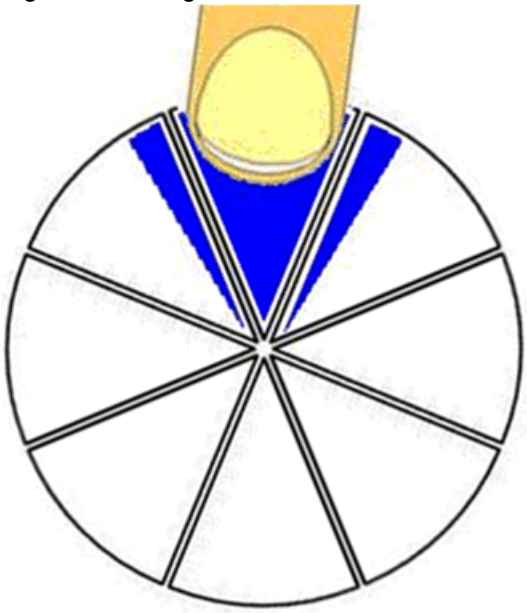
Figure 5. Ordering Physical Sensor Locations



The close proximity of strong signals in one half of the slider results in the same levels aliased into the upper half, but the results are scattered. The sensing algorithms search for strong adjacent sets of signals to declare the resolved slider position.

Radial Sliders

Figure 6. Finger Touches Radial Slider



There are two slider types in the SmartSense2X User Module: linear and radial. While linear sliders have a beginning and an end, radial sliders do not. When a touch happens, the centroid calculation algorithm takes into account sensor counts of the switches to the right and left of the current switch. Radial sliders are not diplexed.

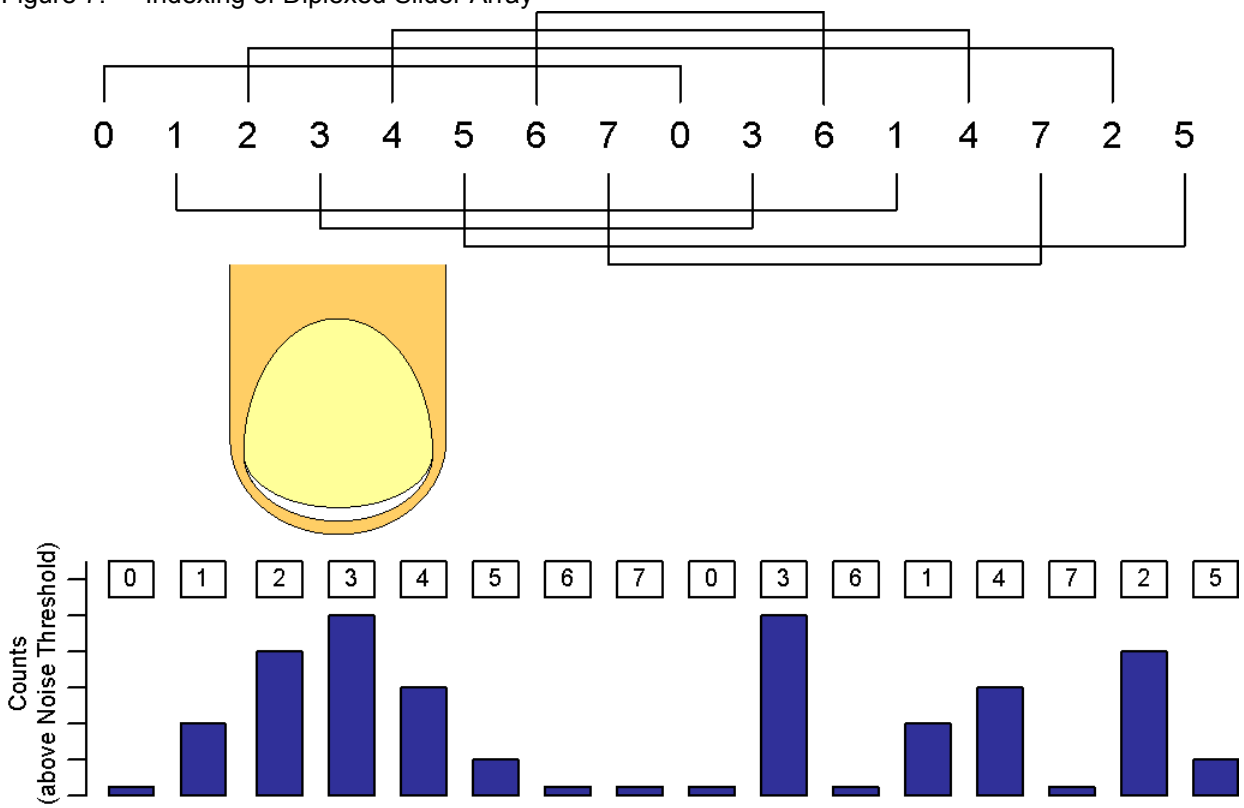
The SmartSense2X User Module has two API functions that support radial sliders. The first function, CSD2X_wGetRadiaPos(), returns a centroid location and the second, CSD2X_wGetRadialInc(), returns finger shift in resolution units. When the finger moves in a clockwise direction, it is a positive offset.

Diplexing

Each PSoC sensor connection in a slider is mapped to two physical locations in the array of slider sensors. The first (or numerically lower) half of the physical locations is mapped sequentially to the base assigned sensors, with the designer assigning the port pin using the SmartSense2X wizard. The second (or upper) half of the physical sensor locations is automatically mapped by an algorithm in the wizard and listed in an include file. The order is established in such a manner that adjacent sensor actuation in one half does not result in adjacent sensor actuation in the other half. Take care to determine this order and map it to the PCB.

Though there are many methods to order the second half of the physical sensor locations, the simplest is to index all of the even sensors in the upper half, followed by all of the odd sensors. Other methods include indexing by other values. The method selected for this user module is to index by three.

Figure 7. Indexing of Diplexed Slider Array



You must balance sensor capacitance in the slider. Depending on the sensor or PCB layouts, there may be longer routes for some of the sensor pairs. The diplex sensor index table is automatically generated by the SmartSense2X wizard when you select diplexing. The following table illustrates the diplexing sequences for different slider segments count.

Table 1. Diplexing Sequence for Different Slider Segment Counts

Total Slider Segment Count	Segment Sequence
10	0,1,2,3,4,0,3,1,4,2
12	0,1,2,3,4,5,0,3,1,4,2,5
14	0,1,2,3,4,5,6,0,3,6,1,4,2,5
16	0,1,2,3,4,5,6,7,0,3,6,1,4,7,2,5
18	0,1,2,3,4,5,6,7,8,0,3,6,1,4,7,2,5,8
20	0,1,2,3,4,5,6,7,8,9,0,3,6,9,1,4,7,2,5,8
22	0,1,2,3,4,5,6,7,8,9,10,0,3,6,9,1,4,7,10,2,5,8
24	0,1,2,3,4,5,6,7,8,9,10,11,0,3,6,9,1,4,7,10,2,5,8,11
26	0,1,2,3,4,5,6,7,8,9,10,11,12,0,3,6,9,12,1,4,7,10,2,5,8,11

Total Slider Segment Count	Segment Sequence
28	0,1,2,3,4,5,6,7,8,9,10,11,12,13,0,3,6,9,12,1,4,7,10,13,2,5,8,11
30	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,0,3,6,9,12,1,4,7,10,13,2,5,8,11,14
32	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0,3,6,9,12,15,1,4,7,10,13,2,5,8,11,14
34	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,0,3,6,9,12,15,1,4,7,10,13,16,2,5,8,11,14
36	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,0,3,6,9,12,15,1,4,7,10,13,16,2,5,8,11,14,17
38	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,0,3,6,9,12,15,18,1,4,7,10,13,16,2,5,8,11,14,17
40	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,0,3,6,9,12,15,18,1,4,7,10,13,16,19,2,5,8,11,14,17
42	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,0,3,6,9,12,15,18,1,4,7,10,13,16,19,2,5,8,11,14,17,20
44	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,2,5,8,11,14,17,20
46	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20
48	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,0,3,6,9,12,15,18,21,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20,23
50	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,2,5,8,11,14,17,20,23
52	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23
54	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,0,3,6,9,12,15,18,21,24,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23,26
56	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,0,3,6,9,12,15,18,21,24,27,1,4,7,10,13,16,19,22,25,2,5,8,11,14,17,20,23,26

Interpolation and Scaling

In sliding sensor and touchpad applications, it is often necessary to determine finger (or other capacitive object) position to greater resolution than the native pitch of the individual sensors. The contact area of a finger on a sliding sensor or a touchpad is often larger than any single sensor.

To calculate the interpolated position using a centroid, the array is first scanned to verify that a specific sensor location is valid. The requirement is for a certain number of adjacent sensor signals to be above a noise threshold. When the strongest signal is found, this signal and those contiguous signals larger than the noise threshold are used to compute a centroid. As few as two and as many as (typically) eight sensors are used to calculate the centroid in the form of:

Equation 1

$$N_{Cent} = \frac{n_{i-1}(i-1) + n_i i + n_{i+1}(i+1)}{n_{i-1} + n_i + n_{i+1}}$$

The calculated value is typically fractional. To report the centroid to a specific resolution, for example, a range of 0 to 100 for 12 sensors, the centroid value is multiplied by a calculated scalar. It is more efficient to combine the interpolation and scaling operations into a single calculation and report this result directly in the desired scale. This is handled in the High-Level APIs section.

Slider sensor count and resolution are set in the SmartSense2X wizard. A scaling value is calculated by the wizard and stored as fractional values. The multiplier for the centroid resolution is contained in three bytes with the definitions given in Table 2.

Table 2. Centroid Multiplier Bit Definition for Sliders

Resolution Multiplier MSB								
Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Multiplier	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
Resolution Multiplier ISB								
Multiplier	128	64	32	18	16	8	4	2
Resolution Multiplier LSB								
Multiplier	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256

The resolution is found by using this equation:

Resolution = (Number of Sensors – 1) x Multiplier

The centroid is held in a 24-bit unsigned integer and its resolution is a function of the number of sensors in the slider and the multiplier.

Feedback Component Selection Guidelines

The SmartSense2X User Module requires an external modulation capacitor and also supports an optional shielding electrode. This section explains how to select external components.

Modulation Capacitor

This user module requires an external modulation capacitor, C_{mod}. The capacitor can be connected to the P0[5] or P0[7] port pin. The pin is selected by the user module parameter setting in the case of single-channel CSD. These pins are fixed in the case of dual-channel CSD as shown in Figure 2. Do not use pins selected for the modulator component connection for any other purposes. Cypress recommends using a 560-ohm series resistor on all CapSense sensor traces for RF interference suppression. This resistor must be placed near the PSoC device.

The recommended value for the modulation capacitor is 2.2 nF.

Shielding Electrode

Some applications require reliable operation in the presence of water films or droplets. White goods, automotive applications, various industrial applications, and others need capacitive sensors that do not give false triggering because of water, ice, and humidity changes. In this case, a separate shielding electrode can be used. This electrode is located behind or outside the sensing electrode. When water films are located on the device insulation overlay surface, the coupling between the shielding and sensing

electrodes increases. The shielding electrode allows you to reduce the influence of parasitic capacitance, which gives you more dynamic range for processing sense capacitance changes.

In some applications it is useful to select the shielding electrode signal and placement relative to the sensing electrode, such that increasing the coupling between these electrodes causes the opposite of the touch change of the sensing electrode capacitance measurement. This simplifies the work of the high-level software API. The SmartSense2X User Module supports separate output for the shielding electrode.

Figure 8. Possible Shield Electrode PCB Layout

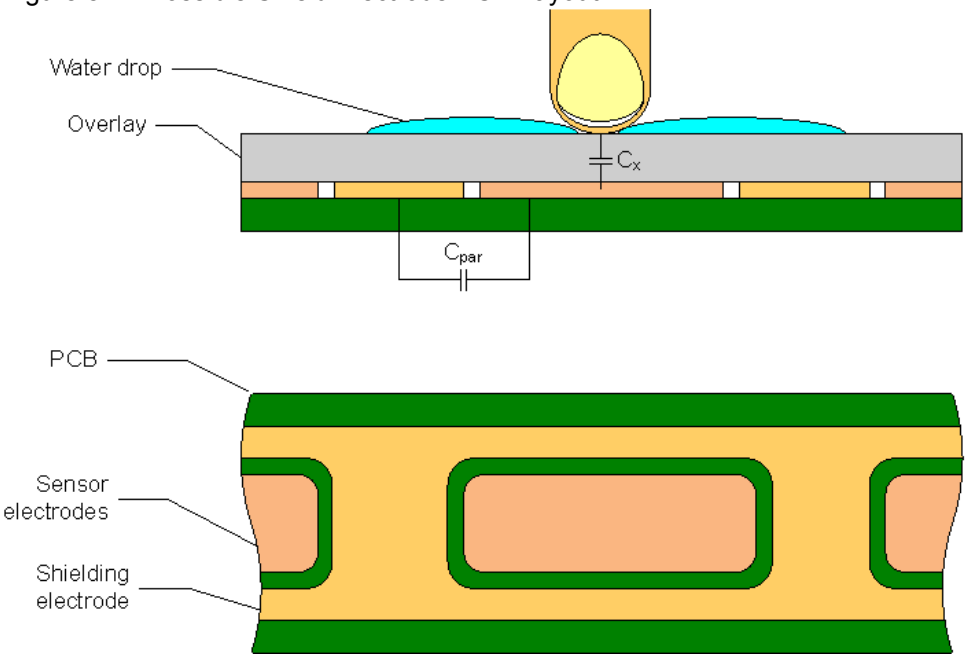


Figure 8 illustrates a possible layout configuration for the button's shield electrode. The shield electrode is especially useful for transparent ITO touchpad devices, where it blocks the LCD drive electrode's noise influence and reduces stray capacitance at the same time.

In this example, the button is covered by a shielding electrode plane. As an alternative, the shielding electrode can be located on the opposite PCB layer, including the plane under the button. A hatch pattern, with a fill ratio of about 30 to 40%, is recommended (refer to the guide, Getting Started with CapSense, for exact recommendations). No additional ground plane is required.

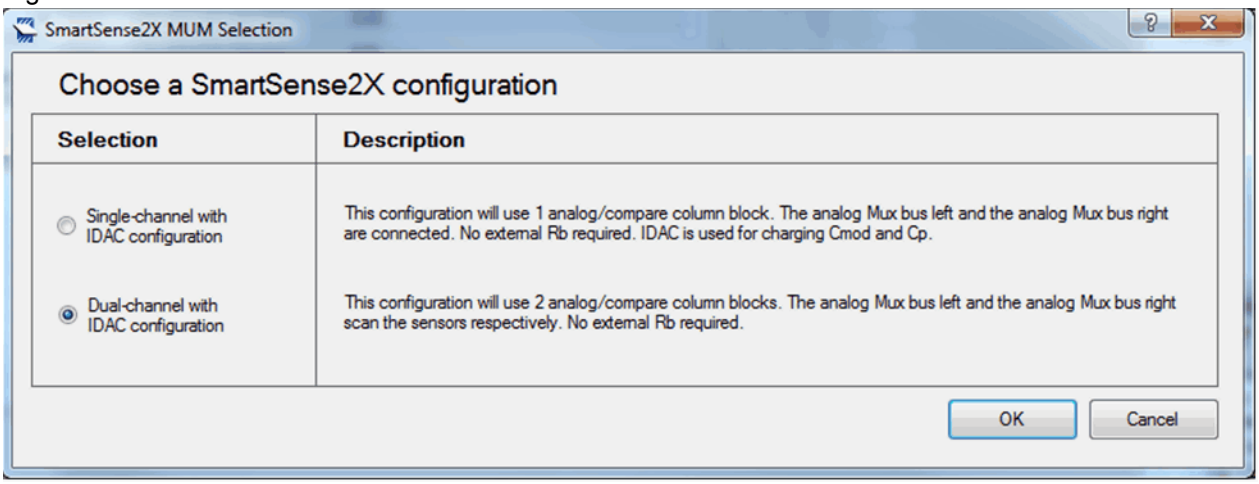
If there are water drops between the shielding and sensing electrodes, it increases the coupling capacitance, C_{par} and reduces the charging time by coupling more charge through the shield signal. In SmartSense, the firmware tunes the reference voltage in such a manner that the raw count increase from this charge coupling is close to zero or slightly negative.

In this user module, the same signal used for the precharge clock is supplied to the shield electrode. The shield electrodes can be connected to GOO[6] for the Left Channel and GOO[7] for the right channel. Set the drive mode to Strong Slow to reduce ground noise and radiated emissions. The optional slew-limiting resistor can be connected between the PSoC device and the shielding electrode.

Placement

The blocks for the user module are automatically placed when the user module is instantiated. Placements are fixed for both Single and Dual Channel configurations. This user module supports two configuration options: single-channel and dual-channel. These configurations can be accessed through the SmartSense2X Multi User Module (MUM) wizard.

Figure 9. SmartSense2X MUM Wizard



The SmartSense2X User Module consumes one CapSense block, one ACE comparator block, one IDAC block, and one analog MUX for each channel. User modules that require specific pin resources, including LCD and I2CHW, must be placed before starting the SmartSense2X wizard to establish pin connections for the SmartSense2X User Module.

Avoid P1[0] and P1[1] when placing capacitive sensor connections. These pins are used for programming the part and may have excess routing capacitance affecting sensor sensitivity and noise.

Figure 10. Single-channel Block Resources

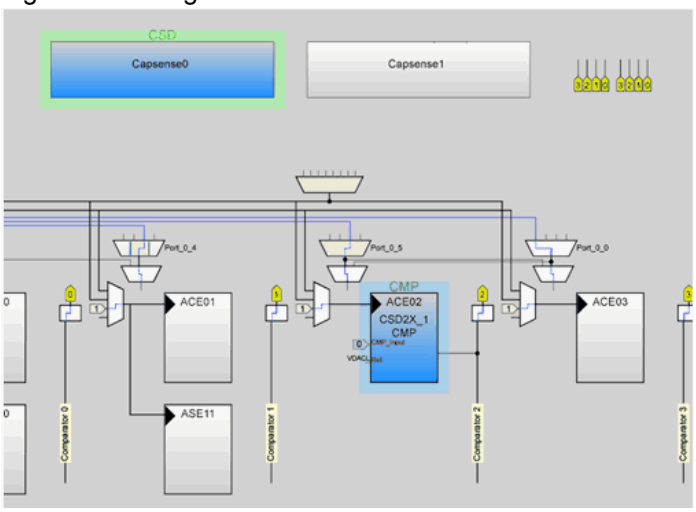
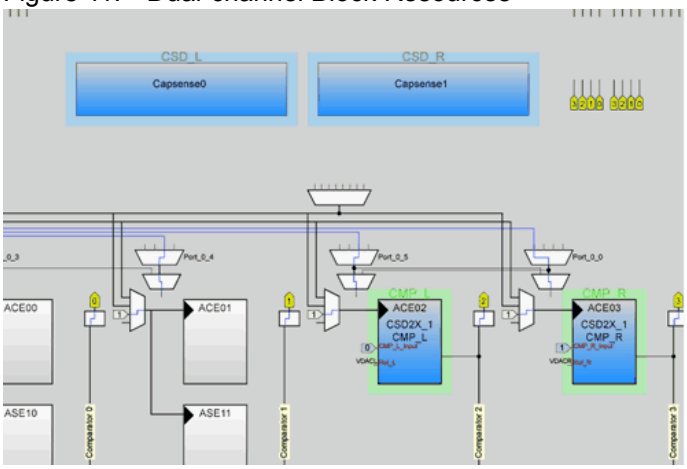


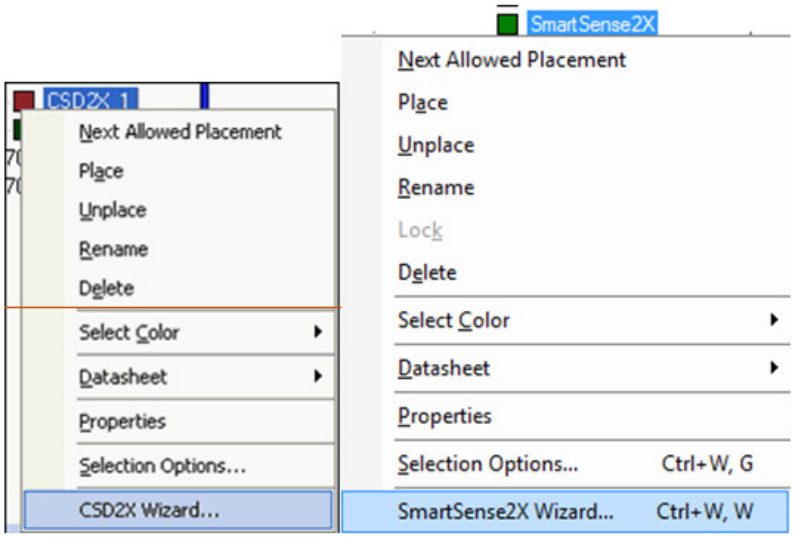
Figure 11. Dual-channel Block Resources



SmartSense2X Wizard

This wizard is used to set up the pinout for your CapSense buttons, sliders, and proximity sensors. You can choose a configuration and assign the buttons and segments using a drag and drop interface.

1. To access the wizard, right-click the user module in the Workspace Explorer and select the SmartSense2X Wizard.



2. The wizard opens showing the numeric entry boxes for the number of sensors and the number of slider sensors. The following figures show the SmartSense2X wizard for single-channel and dual-channel configurations respectively.

Figure 12. SmartSense2X Single-Channel Configuration

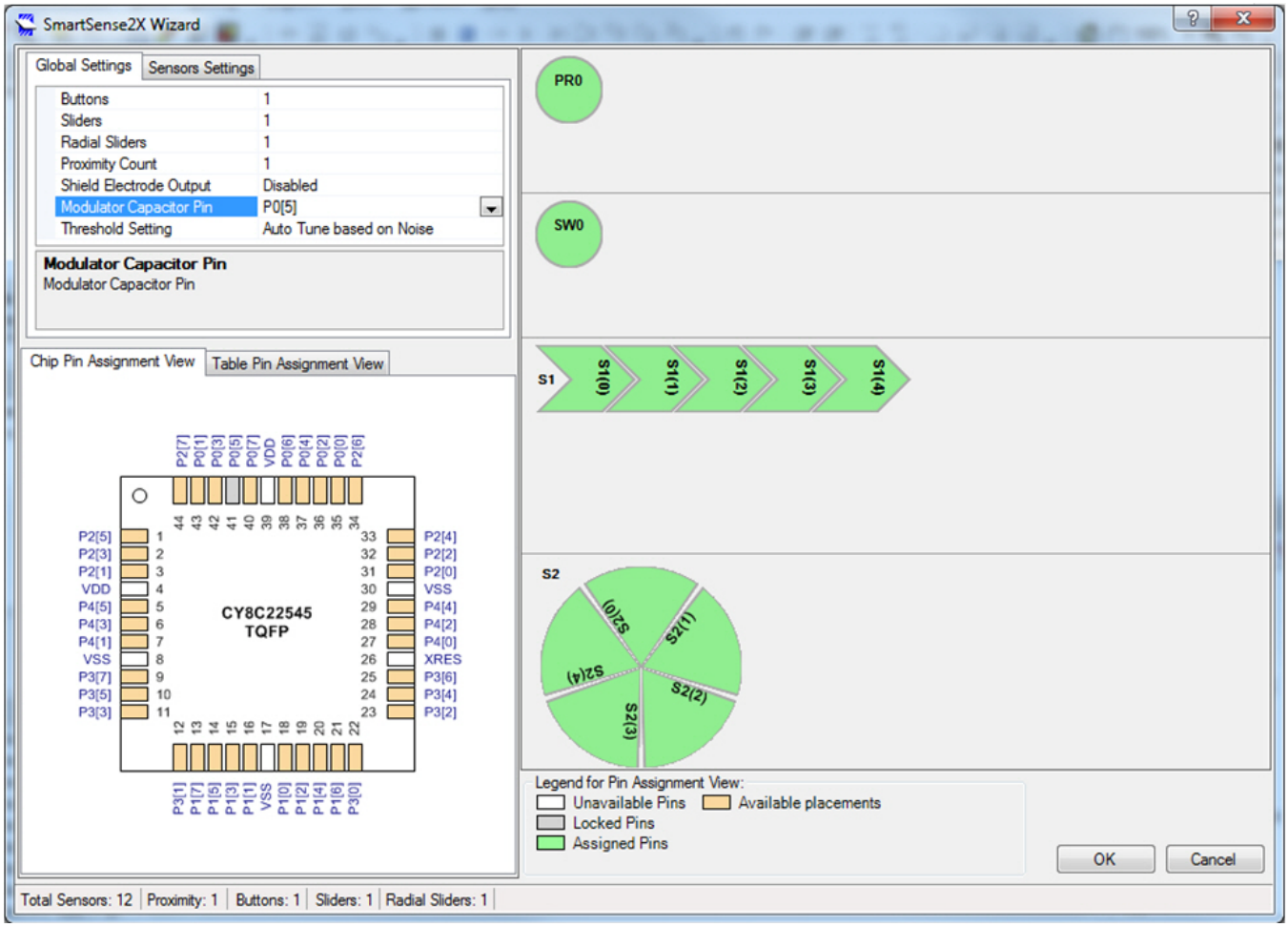
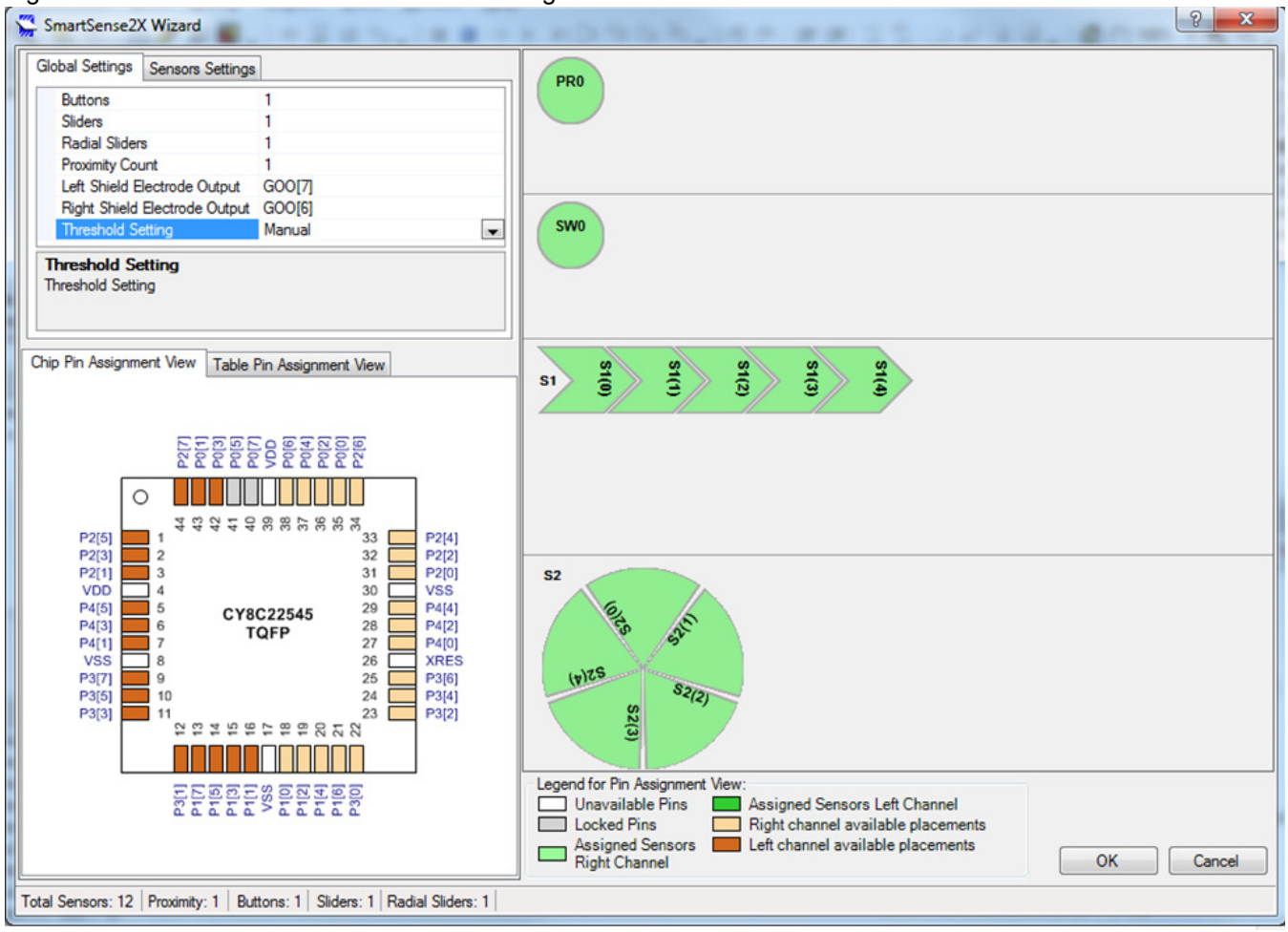


Figure 13. SmartSense2X Dual-Channel Configuration



Wizard Pin Legend

White – The pin cannot be used as a CapSense input.

Gray – The pin is locked. There are two possible causes for this: The first possibility is that another user module, such as the LCD or I^2C, has claimed the pin. The second possibility is that the name of the pin has been changed from its default. To return the pin name to its default, expand the pin in the Pinout view, and select **Default** from the **Select** menu. The pin is now available for assignment in the Wizard.

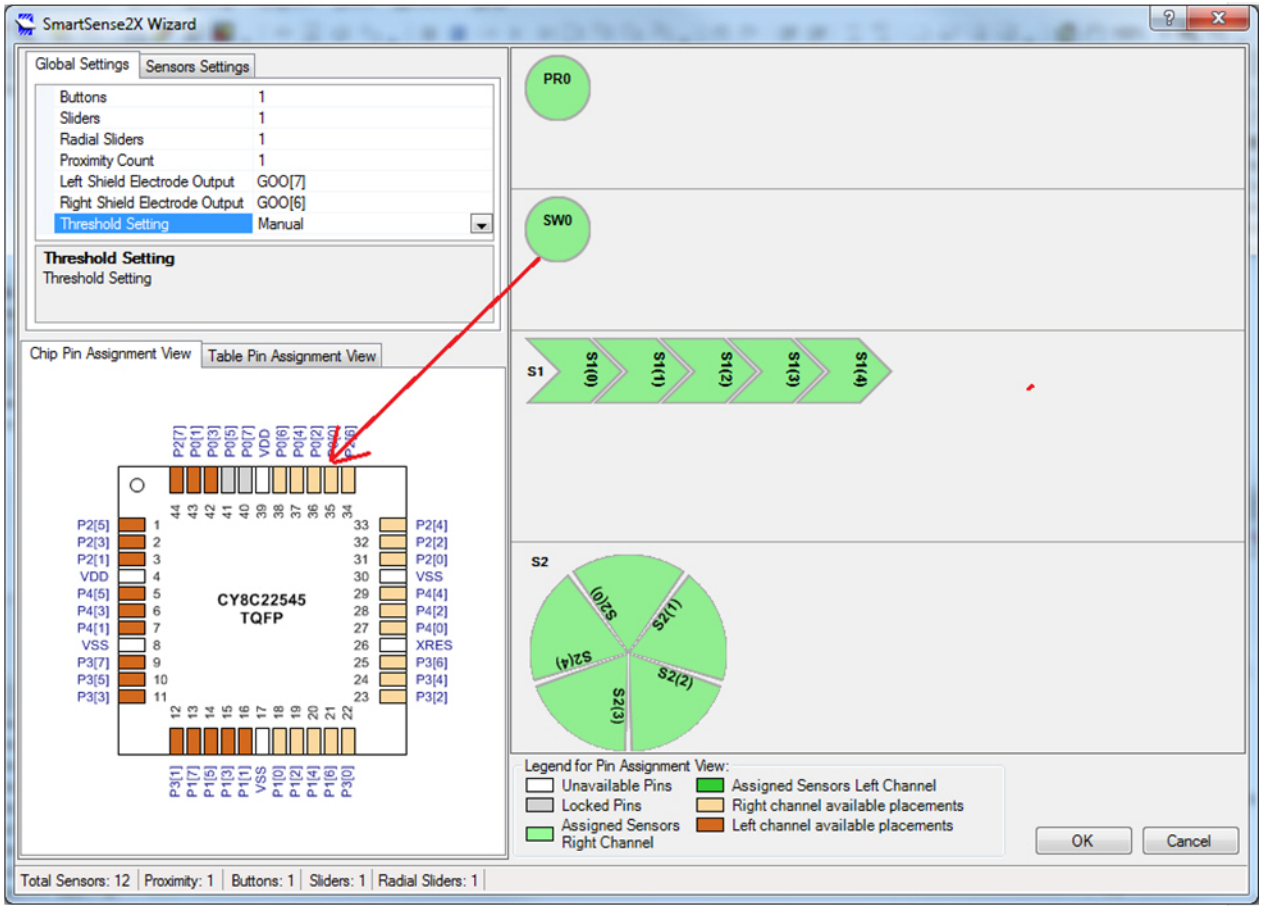
Orange/Brown – The pin is available for assignment. In dual channel, orange is the right channel and brown is the left channel.

Green – The pin has been assigned as a CapSense input.

3. Type the number of independent buttons, sliders, radial sliders, and proximity sensors. The total number of sensors (buttons plus slider elements) is limited to the number of pins available. After entering the data, press the [Enter] key to update the display with the new value.
4. Select the **Shield Electrode** and **Threshold** settings. For more details on these settings, refer to the Global Settings Tab section on page 18.
5. Type the number of sliders. X-Y touchpads require two sliders.

6. Click a slider/sensor/proximity sensor to enter the sensor settings and select the **Sensor Settings** tab. Enter the Diplex, Finger Threshold, Resolution, Sensitivity Level, and Sensors Count parameters. For more details of these parameters, see the Sensor Settings Tab section on page 18.
7. Type the output resolution. The minimum value is five. The maximum value is: (number of pins used for sensors - 1) x 2^8 - 1 or (2 x number of pins used for sensors - 1) x 28 – 1 for diplexed sliders. The software attempts to interpolate touches to this resolution based on the relative strength of the signals on adjacent slider sensors.
8. Select Diplex, if required. This maps the number of pins selected for sensors to twice as many sensor locations on the board. Only the first half of the diplex sensors is shown; the second half is automatically mapped, as described in the Diplexing section on page 7.
9. Left-click a sensor and drag it to any available pin. The port pin is green after selection and it is no longer available. Change the sensor assignments by dragging the sensor of the port pin.

Figure 14. SmartSense2X Wizard – Pin Placement



10. Repeat for the remaining independent sensors.
11. Mapping individual slider sensors to physical port pins is the same as for individual sensors.
12. Enter the pin to which the external modulation capacitor will be connected in the SmartSense wizard (only for the single channel mode; the dual channel setting is fixed).
13. Select if Shield electrode is required or not. In dual channel, you can enable/disable for individual channels separately.
14. Select the type of threshold setting desired in the system. Refer to the Sensor Settings tab for details.
15. To change the pin assignment, place your cursor on the assigned pin, click the pin, and drag and drop it outside the switches box. The pin is now unassigned and you can, then, reassign it.

16. Click **OK** to accept the data and return to PSoC Designer.

Sensor placement is now complete. Right-click in the Device Editor window and select **Refresh** to update the pin connections. Set user module parameters and generate the application. You can now adapt a sample project.

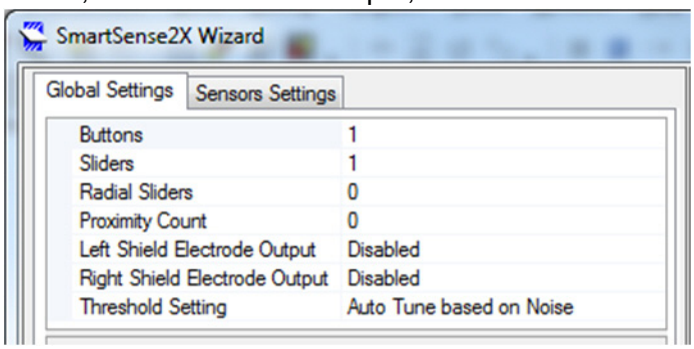
After completing the Wizard, click **Generate Application**. Based on your entries for sensor count, pin assignment, diplexing, and resolution, a set of tables is generated. The tables are located in SmartSense2X_Table.asm.

Wizard Parameters Description

Wizard Parameters are described in the Global Settings Tab and Sensor Settings Tab sections.

Global Settings Tab

The Global Settings tab consists of the following parameters: Buttons, Sliders, Radial Sliders, Proximity Count, Shield Electrode Output, and Threshold Setting.



Buttons

This is the number of physical button sensors.

Sliders

This is the number of physical linear sliders sensors.

Radial Sliders

This is the number of physical radial sliders sensors.

Proximity Count

A proximity sensor detects the presence of a hand or another conductive object before it contacts the touch surface. For example, imagine a hand stretched out to operate a car audio system in the dark. The proximity sensor causes the buttons of the audio system to glow using backlight LEDs when your hand is near.

Shield Electrode Output

This parameter enables or disables the algorithm that supports the optional shield electrode. The dual-channel configuration has the Left Shield Electrode Output and Right Shield Electrode Output properties for the left and right channels respectively. The shield electrode, if enabled, is routed to GOO[6] (P1[6] or P3[6]) for the left channel, or to GOO[7] (P1[7] or P3[7]) for the right channel in the CSD2X Wizard. On devices with fewer pins, P3[6] and P3[7] may be unavailable. For a single channel, the connection depends on whether the right or left CapSense channel is used. The drive mode for the selected pin should be set to Strong.

Modulator Capacitor Pin

SmartSense2X requires an external modulation capacitor, Cmod, connected from Vss to one of two dedicated PSoC pins – P0[5] or P0[7]. The SmartSense2X dual-channel configuration requires two external modulation capacitors that must be connected to the P0[5] and P0[7] pins (the modulator capacitor pin property is not available and pins P0[5] and P0[7] are locked).

The selected pins must not be used for any other purpose. The recommended value for the external modulation capacitor is 2.2 nF. Make sure you use a ceramic capacitor. The temperature capacitance coefficient is not important. Cypress recommends using a 560-ohm series resistor on all CapSense sensor traces for RF interference suppression. This resistor must be placed near the PSoC device.

Threshold Setting

Selects between automatic and manual threshold setting. The threshold setting parameter has three options:

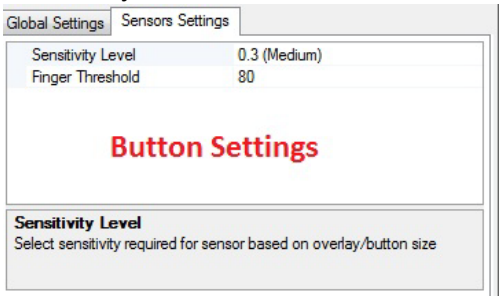
- Manual
  - Finger threshold is set manually with values ranging from 1 to 255. In this case, the Finger Threshold parameter should be adjusted for each sensor in the Sensor Settings Tab. This option can be used when you want to set the finger threshold manually.

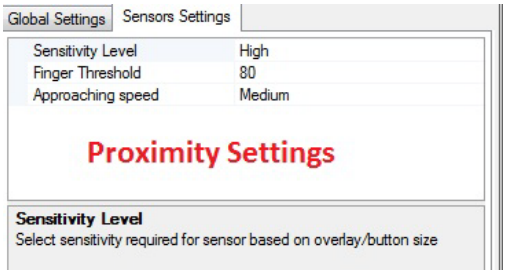
- Auto Tune based on Sensitivity
  - In this option, the user module sets the finger threshold to the sensitivity value selected by the user and the parasitic capacitance of the sensor. This option can be used when you want the user module to set the finger threshold automatically to the sensitivity you selected, irrespective of the noise in the system.

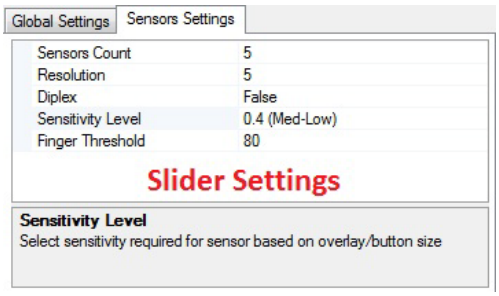
- Auto Tune based on Noise
  - In this option, the user module dynamically adjusts the thresholds during runtime to provide a 5:1 SNR. This option is used when you need a robust 5:1 SNR in your CapSense system.

Sensor Settings Tab

The Sensor Settings Tab consists of the following parameters: Diplex, Finger Threshold, Resolution, Sensitivity Level, and Sensors Count.







Finger Threshold

This threshold is used to determine the state of each button sensor. If any sensor is active, the bIsAnySensorActive() function returns a 1. If all sensors are off, the bIsAnySensorActive() function returns a 0. Possible values range from 1 to 255. The default value is 80. The Finger Threshold is visible only if the 'manual' mode is selected for the Threshold Setting global parameter.

The Finger Threshold property is disabled when "Threshold Setting" is set to "Auto Tune based on Sensitivity" or "Auto Tune based on Noise.

Diplex

This option enables or disables the slider diplex. See the Diplexing section for more information.

Resolution

This option sets the sensor resolution in the range from 5 to (number of pins used for sensors – 1) × 2^8 – 1 or (2 × number of pins used for sensors – 1) × 2^8 – 1 for diplexed sliders.

Sensors Count

Sensors Count is the number of physical sensors in the slider or radial slider. This parameter is available for Linear and Radial Sliders only.

Sensitivity Level

Sensor sensitivity is used to adjust the signal from a sensor. The higher the sensitivity, the more signal is received on touching the sensor. Designs with thicker overlays require higher sensitivity to be associated with the sensors.

The available settings are High (0.1 pf), Medium High (0.2 pf), Medium Low (0.3 pf), Low (0.4 pf), and Lowest (0.5 pf), if the threshold setting mode is set to automatic. For the manual threshold mode, the available settings are High, Medium, and Low.

Sensitivity setting is also available for proximity sensors. You can set this parameter based on the proximity distance requirement. The available options are High, Medium, and Low. Setting the sensitivity to High gives the largest proximity distance. However, this setting should be used in designs only if the proximity sensors are shielded using the shield electrode according to the best practices.

Approaching Speed

This parameter decides the rate at which the baseline of the proximity sensor is updated. It has following values: slow, medium, and fast, and the default setting is Fast. This parameter is available if the Threshold Setting parameter is set to Manual.

Sensor Table

The Sensor table is located in the *SmartSense2X_Table.asm* and consists of a 2-byte entry for each sensor. The first byte is the port number and the second byte is the bit mask for the bit (not the bit number). An example for a table with six sensors is as follows:

```
SmartSense2X_Sensor_Table:
_ SmartSense2X_Sensor_Table
dw 0x0140 // Port 1 Bit 6
dw 0x0301 // Port 3 Bit 0
dw 0x0304 // Port 3 Bit 2
dw 0x0308 // Port 3 Bit 3
dw 0x0302 // Port 3 Bit 1
dw 0x0108 // Port 1 Bit 3
```

Group Table

The Group table defines each of the groups of button sensors or sliders. There is one entry for each slider and one for the independent button sensors. The first entry is always the independent buttons. Each entry is six bytes. The first byte is the index in the Sensor Table where the group starts. The second byte is the number of sensors in that group. The third byte signifies whether the slider is diplexed or not (4 is diplexed, 0 is not diplexed). The fourth, fifth, and sixth bytes are the fixed point multipliers by which the slider's centroid is scaled to achieve the resolution specified in the SmartSense2X wizard.

```
SmartSense2X_Group_Table:
_ SmartSense2X_Group_Table:
; Group Table:
; Origin Count Diplex? DivBtwSw(wholeMSB, wholeLSB, fractByte)
db 0x0, 0x3,    0x00,    0x00, 0x00, 0x00 ; Buttons
db 0x3, 0x8,    0x4,      0x0,   0x0,   0x44 ; Slider 1
```

Diplex Table

Diplex table scan order data is produced for a group that is a slider and with diplexing enabled. Otherwise, a label is created, but no data is placed. The table consists of two parts: sensor mapping for each slider, and a reference for each separate slider to its table. A typical example for an eight-sensor slider is shown here:

```
DiplexTable_0:
; This group is not a diplexed slider
DiplexTable_1:
 db 0,1,2,3,4,5,6,7,0,3,6,1,4,7,2,5// 8 switch slider

 SmartSense2X_Diplex_Table:
_SmartSense2X_Diplex_Table:
 db >DiplexTable_0, <DiplexTable_0
 db >DiplexTable_1, <DiplexTable_1
```

Order Table

The Order Table consists of one byte for each sensor. This byte is the left sensor order in raw counts result array without channel dependence. The table is generated by the wizard and reflects the sensor order.

```
SmartSense2X_Order_Table_Left:
_SmartSense2X_Order_Table_Left:
DB 0x01 // Position 1
SmartSense2X_Order_Table_Right:
_SmartSense2X_Order_Table_Right:
DB 0x00,0x02 // Position 0 and 2
```

Finger Threshold Table

The Finger Threshold table defines the normalized finger threshold for each sensor. A typical example for a one-button and five-sensor slider is shown here.

```
SmartSense2X_Finger_Threshold_Table:
_ SmartSense2X_Finger_Threshold_Table:
db 255; Buttons
db 1, 1, 1, 1, 1; Sliders
```

Sensitivity Level Table

The Sensitivity Level Table defines the sensor sensitivity for each sensor. A typical example for a one-button and five-sensor slider is shown here.

```
SmartSense2X_Sensitivity_Level_Table:
_ SmartSense2X_Sensitivity_Level_Table:
db 2; Buttons
db 2, 2, 2, 2, 2; Sliders
```

Table 3. Sensitivity Level Table Values for Buttons and Sliders

Parameter Value	Table Value
0.1 (High)	1
0.2 (Med-High)	2
0.3 (Medium)	3
0.4 pF (Med-Low)	4
0.5 pF (Low)	5

Table 4. Sensitivity Level Table Values for Proximity Sensors

Parameter Value	Table Value
High	1
Low	5

Approaching Speeds Table

The Approaching Speeds Table defines the approaching speeds for each proximity sensor. This table also has optional default values (equal to 2) for buttons and sliders that are intended to simplify the ASM code. A typical example is shown here:

```
SmartSense2X_Approaching_Speeds:
_ SmartSense2X_Approaching_Speeds:
db 2, 2, 1, 0, 2
```

Table 5. Approaching Speeds Table Values for Buttons and Proximity Sensors

Parameter Value	Table Value	Notes
Slow	1	
Medium	0	
Fast	2	Buttons and slides have this value too.

Parameters and Resources

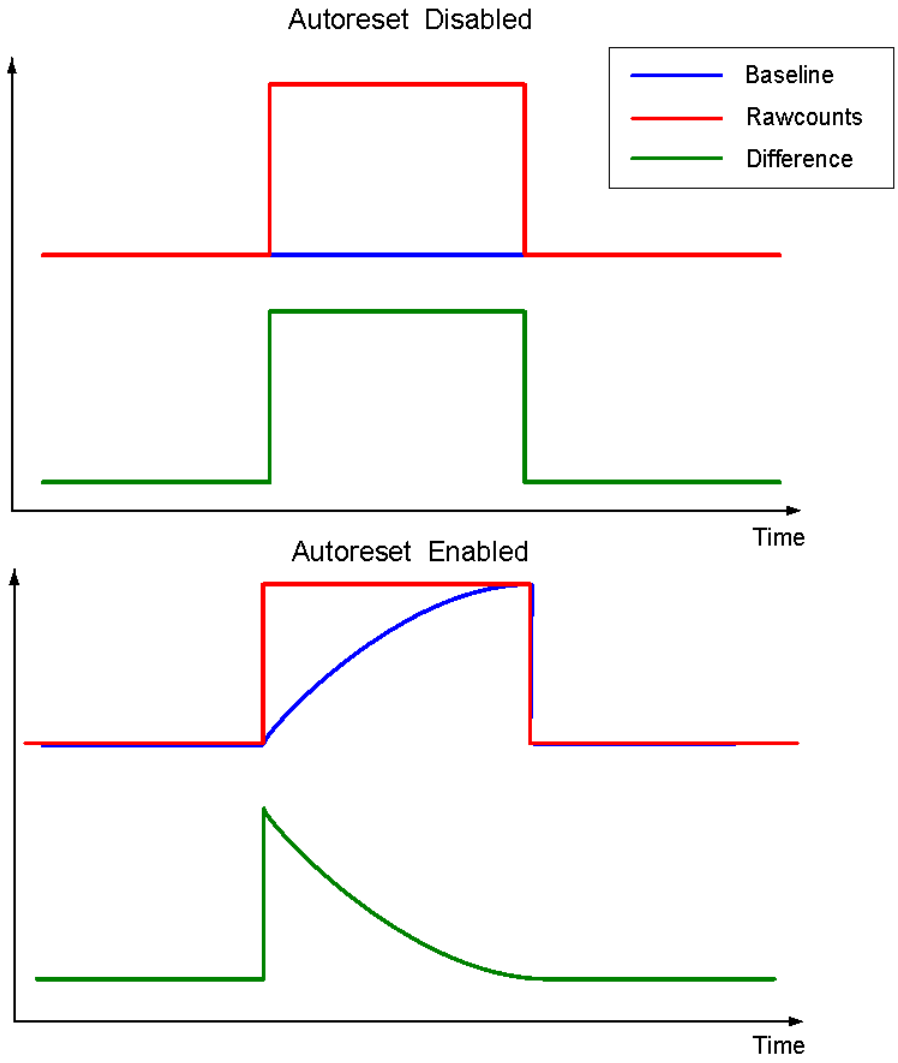
After completing configuration and I/O pin assignment in the SmartSense wizard, set the user module parameters. Note that for any user module parameter change to take affect, the project must be regenerated.

Sensors Autoreset

This parameter determines whether the baseline is updated at all times or only when the signal difference is below the Noise Threshold. The default value for this parameter is "Enabled", that is, the baseline is updated only when the difference between the raw count and the baseline is below the Noise Threshold. Figure 15 illustrates this parameter's effect on baseline update. When Sensors Autoreset is set to **Enabled**, the baseline is always updated without regard to Noise Threshold. This limits the maximum activated time of sensors (typically to 5 - 10s). However, this provides the benefit of preventing sensors from getting stuck due to sudden rises in raw counts that are not caused by a touch. Such sudden rises can be caused by a large power supply voltage fluctuation, a high energy RF noise source, or rapid temperature change.

When Sensors Autoreset is Disabled, the baseline is updated only when the difference between raw count and baseline is below the Noise Threshold. This parameter should generally be left in its default "Disabled" state. See the Appendix section for additional explanation of this parameter.

Figure 15. Effect of the Sensor Autoreset Parameter on Baseline Update



Debounce

This parameter adds a debounce counter to the sensor active transition. For a sensor to transition from inactive to active, the difference count value must stay above finger threshold plus hysteresis for the number of samples specified by this parameter. The debounce counter is incremented by the bIsSensorActive or bIsAnySensorActive API functions.

Possible values are 1 to 255. A setting of '1' has no debounce, but gives the fastest response. The default setting is 3.

Background Scanning

This parameter enables and disables background scan. The default setting is ‘Disable’. If it is set to Enable, then background scanning is implemented.

FMEA_Shorts_Test

This parameter enables/disables the FMEA GND, VDD, and Sensor short tests. The default setting is Disabled.

FMEA_Cmod_Test

This feature enables the test for Cmod and Rb calculation. The default setting is Disabled.

Application Programming Interface

The Application Programming Interface (API) functions are provided as part of the user module to enable you to deal with the module at a higher level. This section specifies the interface to each function together with related constants provided by the include files.

Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns the SmartSense_1 to the first instance of this user module in a particular project. It can be changed to any unique value that follows the syntactic rules for identifiers. The assigned instance name becomes the prefix of every global function name, variable, and constant symbol. In the following descriptions the instance name has been shortened to SmartSense for simplicity.

Note ** In this, as in all user module APIs, the values of the A and X register may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X before the call if those values are required after the call. This "registers are volatile" policy was selected for efficiency reasons and has been in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers must also ensure their code observes the policy. Though some user module API functions may leave A and X unchanged, there is no guarantee they may do so in the future.

For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

Entry Points are supplied to initialize the SmartSense, start it sampling, and stop the SmartSense. In all cases, the instance name of the module replaces the SmartSense prefix shown in the following entry points. Failure to use the correct instance name is a common cause of syntax errors.

API functions use different global arrays. Do not alter these arrays manually. However, you can inspect these values for debugging purposes. For example, you can use a charting tool to display the contents of the arrays. Here are some of the global arrays:

- SmartSense2X_waSnsBaseline[]
- SmartSense2X_waSnsResult[]
- SmartSense2X_waSnsDiff[]
- SmartSense2X_baSnsOnMask[]
- SmartSense2X_baDAC[]
- SmartSense2X_baDACCodeBaseline[]
- SmartSense2X_bScanComplete

SmartSense2X_waSnsBaseline[] – This is an integer array that contains the baseline data of each sensor. The array size is equal to the sensor count. The SmartSense_waSnsBaseline[] array is updated by these functions:

- SmartSense2X_UpdateAllBaselines()
- SmartSense2X_UpdateSensorBaseline()
- SmartSense2X_InitializeBaselines()

SmartSense2X_waSnsResult[] – This is an integer array that contains the raw data of each sensor. The array size is equal to the sensor count. The SmartSense2X_waSnsResult[] data is updated by these functions:

- SmartSense2X_ScanSensor()
- SmartSense2X_ScanAllSensors().

SmartSense_waSnsDiff [] – This is an integer array that contains the difference between the raw data and the baseline data of each sensor. The array size is equal to the sensor count.

SmartSense_baSnsOnMask[] – This is a byte array that holds the sensor on or off state (for buttons or sliders). SmartSense_baSnsOnMask[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). SmartSense_baSnsOnMask[1] contains the masked bits for sensors 8 through 15 (if they are needed), and so on. This byte array contains as many elements as are necessary to contain all the placed sensors. The value of a bit is 1 if the button is on and 0 if the button is off. The SmartSense2X_baSnsOnMask[] data is updated by these functions:

- SmartSense2X_bIsSensorActive()
- SmartSense2X_bIsAnySensorActive()

SmartSense2X_baDAC [] – This is a BYTE array that contains the IDAC Register value for each sensor. The values from this array are copied to the SmartSense2X_bIdacValue variable before scan.

SmartSense2X_baDACCodeBaseline [] – This table contains calibrated current values for each sensor, which can be changed in runtime. This can be useful in complex projects.

For double-channel configuration:

```
SmartSense2X_baDACCodeBaselineL:
_SmartSense2X_baDACCodeBaselineL:
SmartSense2X_baDACCodeBaselineR:
_SmartSense2X_baDACCodeBaselineR:
```

For single-channel configuration:

```
SmartSense2X_baDACCodeBaselineR:
_SmartSense2X_baDACCodeBaselineR:
```

SmartSense2X_bScanComplete [] – This variable is valid only when the background scanning feature is enabled. This variable should get set to 1 whenever sensor scanning is completed.

SmartSense2X_Start

Description:

Initializes registers and starts the user module. This function must be called before calling any other user module functions.

C Prototype:

```
void  SmartSense2X_Start()
```

Assembly:

```
lcall  SmartSense2X_Start
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense2X_Stop

Description:

Restores the SmartSense2X block to its idle default configuration, releases the AMUX bus for other purposes, disables internal interrupts, and calls SmartSense2X_ClearSensors() to reset all sensors to their inactive state.

C Prototype:

```
void  SmartSense2X_Stop()
```

Assembly:

```
lcall  SmartSense2X_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense2X_Resume

Description:

Resumes the user module operation after SmartSense2X_Stop call.

C Prototype:

```
void SmartSense2X_Resume()
```

Assembly:

```
lcall SmartSense2X_Resume
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense2X_ScanSensor

Description:

Scans the selected sensors. Each sensor has a unique number within the sensor array. For single-channel configurations, this number is assigned by the CSD2X Wizard in sequence. Sw0 is sensor 0, Sw1 is sensor 1, and so on. For two-channel configurations, the sensor number is a value from 0 to Maximum Channel Sensor Number. For example, if there are two sensors on the left channel, their values are 0 and 1, respectively. If there are also two sensors on the right channel, their values are also 0 and 1, respectively. If a value of 0xFF is passed into this function as a sensor number, no sensor is scanned for that channel.

C Prototype:

In Singe Channel Configuration:

```
void SmartSense2X_ScanSensor(BYTE bSensor);
```

In Double Channel Configuration:

```
void  SmartSense2X_ScanSensor(BYTE bSensorLeft, BYTE bSensorRight);
```

Assembly:

In Singe Channel Configuration:

```
mov A,  bSensor
lcall SmartSense2X_ScanSensor
```

In Double Channel Configuration:

```
mov   A,  bSensorLeft
mov   X,  bSensorRight
lcall SmartSense2X_ScanSensor
```

Parameters:

In Singe Channel Configuration:

A => Sensor Number

In Double Channel Configuration:

A => Sensor Number Left

X => Sensor Number Right

Return Value:

None

Side Effects

**

SmartSense2X_ScanAllSensors

Description:

Scans all of the configured sensors by calling SmartSense2X_ScanSensor() for each sensor.

C Prototype:

```
void SmartSense2X_ScanAllSensors()
```

Assembly:

```
lcall  SmartSense2X_ScanAllSensors
```

Parameter:

None

Return Value:

None

Side Effects:

**

SmartSense2X_UpdateAllBaselines

Description:

Uses the SmartSense2X_bUpdateSensorBaseline() function to update the baselines for all sensors.

C Prototype:

```
void SmartSense2X_UpdateAllBaselines()
```

Assembly:

```
lcall  SmartSense2X_UpdateAllBaselines
```

Parameter:

None

Return Value:

None

Side Effects:

**

SmartSense2X_bIsSensorActive

Description:

Checks the difference count array for the given sensor compared to its finger threshold. Hysteresis is taken into account. The Hysteresis value is added or subtracted from the finger threshold based on whether the sensor is currently on. If it is active, the threshold is lowered. If it is inactive, the threshold is raised. This function also updates the sensor's bit in the SmartSense2X_baSnsOnMask[] array.

C Prototype:

```
BYTE  SmartSense2X_bIsSensorActive(BYTE bSensorNum)
```

Assembly:

```
mov  A,  bSensorNum
lcall SmartSense2X_bIsSensorActive
```

Parameters:

bSensor A => Sensor Number

Return Value:

Returns value of 1 if active, 0 if not active

A => 1 – Selected sensor is active, 0 – Selected sensor is not active.

Side Effects:

**

SmartSense2X_bIsAnySensorActive

Description:

Checks the difference count array for all sensors compared to their finger threshold. Calls SmartSense2X_bIsSensorActive() for each sensor so that the SmartSense2X_baSnsOnMask[] array is up to date after calling this function.

C Prototype:

```
BYTE  SmartSense2X_bIsAnySensorActive()
```

Assembly:

```
lcall SmartSense2X_bIsAnySensorActive
```

Parameters:

None

Return Value:

Returns value of 1 if active, 0 if not active

A => 1 – One or more sensors are active, 0 – No sensors are active.

Side Effects:

**

SmartSense2X_wGetCentroidPos

Description:

Checks a linear slider array for a centroid. If there is a centroid, the offset and length are stored in temporary variables and the centroid position is calculated to the resolution specified in the SmartSense2X Wizard. This function is available only if slider is defined by the SmartSense2X Wizard.

C Prototype:

```
WORD  SmartSense2X_wGetCentroidPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall SmartSense2X_wGetCentroidPos
```

Parameters:

bSnsGroup A => Group Number

This parameter is the Group Number of the slider. Group 0 is always the independent buttons. Sliders are contained in group 1 and higher.

Return Value:

Position value of the slider, LSB in A and MSB in X.

Side Effects:

This routine modifies the difference counts by subtracting the noise threshold value and should only be called once after each scan to avoid getting negative difference values. If your application monitors difference count signals, call this routine after difference count data transmission.

If any slider sensor is active, the function returns values from zero to the Resolution value set in the Wizard. If no sensors are active, the function returns -1 (FFFFh). If an error occurs during execution of the centroid/diplexing algorithm, the function returns -1 (FFFFh). You can use the SmartSense2X_blsSensorActive() routine to determine which slider segments are touched.

SmartSense2X_wGetRadialPos

Description:

Checks a radial slider array for a centroid. If there is a centroid, the centroid position is calculated to the resolution specified in the SmartSense2X Wizard.

C Prototype:

```
WORD  SmartSense2X_wGetRadialPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall SmartSense2X_wGetRadialPos
```

Parameters:

bSnsGroup A => Group Number

This parameter is the Group Number of radial slide. You can get its number through the SmartSense2X Wizard on the left side of radial slider representation (for example, "s2" means the radial slider Group Number is 2).

Return Value:

Position value of the radial slider, LSB in A and MSB in X.

Side Effects:

This routine modifies the difference counts by subtracting the noise threshold value and should only be called once after each scan to avoid getting negative difference values and baseline update. If your application monitors difference count signals, call this routine after difference count data transmission.

If any slider sensor is active, the function returns values from zero to the Resolution value set in the Wizard. If no sensors are active, the function returns -1 (FFFFh). If an error occurs during execution of the centroid algorithm, the function returns -1 (FFFFh). You can use the SmartSense2X_blsSensorActive() routine to determine which slider segments are touched.

SmartSense2X_wGetRadialInc

Description:

Returns actual finger shift, the difference between current and previous finger positions. This function works in conjunction with SmartSense2X_wGetRadialPos().

C Prototype:

```
WORD  SmartSense2X_wGetRadialInc(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall SmartSense2X_wGetRadialInc
```

Parameters:

bSnsGroup A => Group Number

This parameter is the Group Number of radial slide. You can get its number through the SmartSense2X Wizard on the left side of radial slider representation (for example, "s2" means the radial slider Group Number is 2).

Return Value:

Finger shift value is positive, if clockwise, and negative, if anti-clockwise, LSB in A and MSB in X.

Finger shift value is the difference between current and previous finger positions.

Side Effects:

The routine must be called only after calling SmartSense2X_wGetRadialPos(), because it uses internal data stored in global variables by the latter.

SmartSense2X_InitializeSensorBaseline

Description:

Loads the SmartSense2X_waSnsBaseline[bSensor] array element with an initial value by scanning the selected sensor. The raw count value is copied into the baseline array element for the selected sensor. This function can be used to reset the baseline of an individual sensor.

C Prototype:

```
void SmartSense2X_InitializeSensorBaseline(BYTE bSensorNum)
```

Assembly:

```
mov A, bSensorNum
lcall SmartSense2X_InitializeSensorBaseline
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

SmartSense2X_InitializeBaselines

Description:

Loads the SmartSense2X_waSnsBaseline[] array with initial values by scanning each sensor. The raw count values are copied in to baseline array for each sensor.

C Prototype:

```
void SmartSense2X_InitializeBaselines()
```

Assembly:

```
lcall SmartSense2X_InitializeBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense2X_ClearSensors

Description:

Clears all sensors to the non-sampling state by sequentially calling SmartSense2X_wGetPortPin() and SmartSense2X_DisableSensor() for eachs of the sensors.

C Prototype:

```
void  SmartSense2X_ClearSensors()
```

Assembly:

```
lcall  SmartSense2X_ClearSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense2X_wReadSensor

Description:

Returns the key Raw scan value in A (LSB) and X (MSB).

C Prototype:

```
WORD  SmartSense2X_wReadSensor(BYTE bSensor)
```

Assembly:

```
mov A, bSensor
lcall SmartSense2X_wReadSensor
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

SmartSense2X_wGetPortPin

Description:

Returns the port number and pin mask for a given sensor. The passed parameter indexes and selects the data from the SmartSense2X_Sensor_Table2[]. The return value can be passed to the SmartSense2X_EnableSensor(), SmartSense2X_DisableSensor().

C Prototype:

```
WORD  SmartSense2X_wGetPortPin(BYTE bSensor)
```

Assembly:

```
mov   A, bSensor
lcall SmartSense2X_wGetPortPin
```

Parameters:

bSensor – the range is from 0 to (n – 1), where n is the total number of sensors set in the SmartSense2X Wizard plus the number of sensors included in sliders. The sensor number is used by SmartSense2X_wGetPortPin() to determine port and bit mask for the selected active sensor.

Return Value:

A => Sensor Bitmask

X => Port Number

Side Effects:

**

SmartSense2X_EnableSensor

Description:

Configures the selected sensor to measure during the next measurement cycle. The port and sensor can be selected using the SmartSense2X_wGetPortPin() function, with the port number and sensor bitmask loaded into X and A, respectively. Drive modes are modified to place the selected port and pin into Analog High Z mode and to enable the correct Analog Mux Bus input.

C Prototype:

```
void  SmartSense2X_EnableSensor(BYTE bMask, BYTE bPort)
```

Assembly:

```
mov   X, bPort
mov   A, bMask
lcall SmartSense2X_EnableSensor
```

Parameters:

A => Sensor Bitmask

X => Port Number

bMask - Bitmask for given sensor

bPort - Port Number for given key

Return Value:

None

Side Effects:

**

SmartSense2X_DisableSensor

Description:

Disables the sensor selected by the SmartSense2X_wGetPortPin() function. The drive mode is changed to Strong (001). This effectively grounds the sensor. The connection from the port pin to the AnalogMuxBus is turned off. The function parameters are returned by SmartSense2X_wGetPortPin() function.

C Prototype:

```
void  SmartSense2X_DisableSensor(BYTE bMask, BYTE bPort)
```

Assembly:

```
mov   X, bPort
mov   A, bMask
lcall SmartSense2X_DisableSensor
```

Parameters:

A => Sensor Bitmask

X => Port Number

bMask - Bitmask for given sensor

bPort - Port Number for given key

Return Value:

None

Side Effects:

**

The following APIs are generated only if background scanning is enabled.

BOOL SmartSense2X_bIsScanComplete

Description:

Checks the Scan Complete flag in the bScanComplete variable and returns TRUE or FALSE. This API should also reset this flag after the call. This API is available if the BG Scan property is set to Enable.

C Prototype:

```
BOOL SmartSense2X_bIsScanComplete()
```

Assembly:

```
lcall  SmartSense2X_bIsScanComplete
```

Parameters:

None

Return Value:

Returns value of 1 if active, 0 if not active

A => 1 - scan is complete, 0 - scan is incomplete

Side Effects:

**

BYTE SmartSense2X_bFMEA_CheckGndShort(void)

Description:

This API checks if any of the sensors (including shield electrode if enabled) is shorted to ground. The SmartSense2X_bFMEA_CheckVddShort() function updates the SmartSense2X_baSensorShortGnd[] array, which is valid only when the FMEA feature is enabled. SmartSense2X_baSensorShortGnd[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). SmartSense2X_baSensorShortGnd[1] contains the masked bits for sensors 8 through 15 (if they are needed), and so on. This byte array contains as many elements as are necessary to contain all the placed sensors.

There is an additional array, SmartSense2X_baSnsShortChk[], which has the same structure as that of SmartSense2X_baSensorShortGnd[]. This SmartSense2X_baSnsShortChk[] is updated by all bFMEA_CheckSensorShort, bFMEA_CheckVddShort, bFMEA_CheckGndShort, and bFMEA_Cmod_Check (or bFMEA_Left_Cmod_Check, bFMEA_Right_Cmod_Check) APIs.

The optional SmartSense2X_baShieldShort variable contains the Shield Electrode status.

SmartSense2X_baShieldShort values for Single configurations:

Constant (Single Configuration)	Value	Description
SmartSense2X_SHIELD_NO_SHORT	0x00	Shield Electrode is not shorted to GND
SmartSense2X_SHIELD_GND_SHORT	0x02	Shield Electrode is shorted to GND

SmartSense2X_baShieldShort values for Dual configurations:

Constant (Single Configuration)	Value	Description
SmartSense2X_LEFT_SHIELD_NO_SHORT	0x00	Left Shield Electrode is not shorted to GND
SmartSense2X_LEFT_SHIELD_GND_SHORT	0x02	Left Shield Electrode is shorted to GND
SmartSense2X_RIGHT_SHIELD_NO_SHORT	0x00	Right Shield Electrode is not shorted to GND
SmartSense2X_RIGHT_SHIELD_GND_SHORT	0x02	Right Shield Electrode is shorted to GND

C Prototype:

```
BYTE SmartSense2X_bFMEA_CheckGndShort(void);
```

Assembly:

```
lcall  SmartSense2X_bFMEA_CheckGndShort
```

Parameters:

None

Returns:

1: If any sensor is shorted to GND

0:If none of the sensors is shorted to GND.

BYTE SmartSense2X_bFMEA_CheckVddShort(void)

Description:

This API checks if any of the sensors (including shield electrode if enabled) is shorted to Vdd. The SmartSense2X_bFMEA_CheckVddShort() function updates the SmartSense2X_baSensorShortVdd[] array, which is valid only when the FMEA feature is enabled. SmartSense2X_baSensorShortVdd[0] contains the masked bits for sensors 0 through 7 (sensor 0 is bit 0, sensor 1 is bit 1). SmartSense2X_baSensorShortVdd[1] contains the masked bits for sensors 8 through 15 (if they are needed), and so on. This byte array contains as many elements as are necessary to contain all the placed sensors.

There is an additional array, SmartSense2X_baSnsShortChk[], which has the same structure as that of SmartSense2X_baSensorShortGnd[]. This SmartSense2X_baSnsShortChk[] is updated by all bFMEA_CheckSensorShort, bFMEA_CheckVddShort, bFMEA_CheckGndShort, bFMEA_Cmod_Check (or bFMEA_Left_Cmod_Check, bFMEA_Right_Cmod_Check) APIs.

The optional SmartSense2X_baShieldShort variable contains the Shield Electrode status.

SmartSense2X_baShieldShort values for Single configurations:

Constant (Single Configuration)	Value	Description
SmartSense2X_SHIELD_NO_SHORT	0x00	Shield Electrode is not shorted to VDD
SmartSense2X_SHIELD_VDD_SHORT	0x02	Shield Electrode is shorted to VDD

SmartSense2X_baShieldShort values for Dual configurations:

Constant (Single Configuration)	Value	Description
SmartSense2X_LEFT_SHIELD_NO_SHORT	0x00	Left Shield Electrode is not shorted to VDD
SmartSense2X_LEFT_SHIELD_VDD_SHORT	0x02	Left Shield Electrode is shorted toVDD
SmartSense2X_RIGHT_SHIELD_NO_SHORT	0x00	Right Shield Electrode is not shorted to VDD
SmartSense2X_RIGHT_SHIELD_VDD_SHORT	0x02	Right Shield Electrode is shorted to VDD

C Prototype:

```
BYTE SmartSense2X_bFMEA_CheckVddShort(void);
```

Assembly:

```
lcall  SmartSense2X_bFMEA_CheckVddShort
```

Parameters:

None

Returns:

1: If any sensor is shorted to VDD

0:If none of the sensors is shorted to VDD.

SmartSense2X_bFMEA_CheckSensorShort(void)

Description

This API checks if any of the sensors is shorted to another sensor. There is an additional array, SmartSense2X_baSnsShortChk[], which has the same structure as the SmartSense2X_baSensorShortGnd[]. This SmartSense2X_baSnsShortChk[] is updated by all bFMEA_CheckSensorShort, bFMEA_CheckVddShort, bFMEA_CheckGndShort, and bFMEA_Cmod_Check (or, bFMEA_Left_Cmod_Check, bFMEA_Right_Cmod_Check) APIs.The optional SmartSense2X_baShieldShort variable contains the Shield Electrode status.

SmartSense2X_baShieldShort values for single configurations:

Constant (Single Configuration)	Value	Comments
SmartSense2X_SHIELD_NO_SHORT	0x00	Shield Electrode is not shorted to any sensor
SmartSense2X_SHIELD_SENS_SHORT	0x04	Shield Electrode is shorted to a sensor

SmartSense2X_baShieldShort values for dual configurations:

Constant (Single Configuration)	Value	Comments
SmartSense2X_LEFT_SHIELD_NO_SHORT	0x00	Left Shield Electrode is not shorted to any sensor
SmartSense2X_LEFT_SHIELD_SENS_SHORT	0x04	Left Shield Electrode is shorted to a sensor
SmartSense2X_RIGHT_SHIELD_NO_SHORT	0x00	Right Shield Electrode is not shorted to any sensor
SmartSense2X_RIGHT_SHIELD_SENS_SHORT	0x40	Right Shield Electrode is shorted to a sensor
SmartSense2X_SHIELD_SHIELD_SHORT	0x80	Right Shield Electrode is shorted to Left Shield Electrode

C Prototype

```
BYTE SmartSense2X_bFMEA_CheckSensorShort(void)
```

Assembly

```
lcall  SmartSense2X_bFMEA_CheckSensorShort
```

Parameters

None

Return Value

1: If any sensor is shorted to another sensor0: If no sensor is shorted to another sensor.

Side Effects

This API does not check the sensor to sensor short correctly if one of sensors is shorted to Vdd. The SmartSense2X_bFMEA_CheckVddShort() and SmartSense2X_bFMEA_CheckGndShort() APIs should be used to check Gnd or Vdd short.

SmartSense2X_bFMEA_Cmod_Check(void) - Single Channel

SmartSense2X_bFMEA_Left_Cmod_Check(void) - Dual Channel

SmartSense2X_bFMEA_Right_Cmod_Check(void) - Dual Channel

Description

This API is available only if the IDAC method is selected and if the FMEA feature is enabled. This checks Cmod for short test and also whether it is within the valid range or not. The valid range is defined by the recommended minimum and maximum Cmod values, with an error of 20%. The minimum Cmod value is 3.7 nF and the maximum Cmod value is 56.4 nF. The SmartSense2X_bFMEA_Cmod_Check() function updates SmartSense2X_wCmod_Val (SmartSense2X_wCmod_L_Val and SmartSense2X_wCmod_R_Val for Dual Channel Configurations) WORD variable. This variable contains the value of Cmod in nF scaled to 10. For example, SmartSense2X_wCmod_Val = 220 corresponds to 22 nF (or 0.022 uF). This value is valid only if the SmartSense2X_bFMEA_Cmod_Check() API returns bRetVal.4.

An additional array, SmartSense2X_baSnsShortChk[], has the same structure as the SmartSense2X_baSensorShortGnd[]. This SmartSense2X_baSnsShortChk[] is updated by all bFMEA_CheckSensorShort, bFMEA_CheckVddShort, bFMEA_CheckGndShort, and bFMEA_Cmod_Check (or, bFMEA_Left_Cmod_Check, bFMEA_Right_Cmod_Check) APIs.

C Prototype

Single Channel:

```
BYTE SmartSense2X_bFMEA_Cmod_Check(void);
```

Dual Channel:

```
BYTE SmartSense2X_bFMEA_Left_Cmod_Check(void);
BYTE SmartSense2X_bFMEA_Right_Cmod_Check(void);
```

Assembly

```
lcall  SmartSense2X_bFMEA_Cmod_Check
```

Parameters

None

Return Value

bRetVal.0: Cmod shorted to GND

bRetVal.1: Cmod shorted to VDD

bRetVal.4: Cmod within +20%

bRetVal.8: Cmod is low (under the valid range) or Cmod is disconnected.

bRetVal.16: Cmod is high (over the valid range)

Single channel:

Constants	Value	Description
SMARTSENSE2X_CMOD_SHORTED_TO_GND	0	Cmod shorted to GND
SMARTSENSE2X_CMOD_SHORTED_TO_VDD	1	Cmod shorted to Vdd
SMARTSENSE2X_CMOD_WITHIN_20	4	Cmod within ±20%
SMARTSENSE2X_CMOD_IS_LOW	8	Cmod is out of range
SMARTSENSE2X_CMOD_IS_HIGH	16	Cmod is out of range

Dual channel:

Constants	Value	Description
SMARTSENSE2X_LEFT_CMOD_SHORTED_TO_GND	0	Left Cmod shorted to GND
SMARTSENSE2X_LEFT_CMOD_SHORTED_TO_VDD	1	Left Cmod shorted to Vdd
SMARTSENSE2X_LEFT_CMOD_WITHIN_20	4	Left Cmod within ±20%
SMARTSENSE2X_LEFT_CMOD_IS_LOW	8	Left Cmod is out of range
SMARTSENSE2X_LEFT_CMOD_IS_HIGH	16	Left Cmod is out of range
SMARTSENSE2X_RIGHT_CMOD_SHORTED_TO_GND	0	Right Cmod shorted to GND
SMARTSENSE2X_RIGHT_CMOD_SHORTED_TO_VDD	1	Right Cmod shorted to Vdd
SMARTSENSE2X_RIGHT_CMOD_WITHIN_20	4	Right Cmod within ±20%
SMARTSENSE2X_RIGHT_CMOD_IS_LOW	8	Right Cmod is out of range
SMARTSENSE2X_RIGHT_CMOD_IS_HIGH	16	Right Cmod is out of range

DC and AC Electrical Characteristics

Table 6. DC and AC Electrical Characteristics

Parameters	Description	Conditions	Min	Typ	Max	Units
Cp (max)	Supported sensor parasitic capacitance max value	Over –40 to +80 °C	45	–	–	pF
Cp (min)	Supported sensor parasitic capacitance min value	Over –40 to +80 °C	–	–	5	pF

Sample Firmware Source Code

Example 1. This code starts the user module and continuously scans the sensors. The communication section can be used to communicate values to a PC charting tool.

```
//-------------------------------------------------------------------------
// Sample C code for the SmartSense2X module
// Scanning all sensors continuously
//-------------------------------------------------------------------------

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

void main(void)
{
    M8C_EnableGInt;
    SmartSense2X_Start();
    SmartSense2X_InitializeBaselines() ; //scan all sensors first time, init baseline
    // Loop Forever

    while (1)
    {
        SmartSense2X_ScanAllSensors(); //scan all sensors in array (buttons and
        //sliders)
        SmartSense2X_UpdateAllBaselines(); //Update all baseline levels;
        //detect if any sensor is pressed
        if(SmartSense2X_bIsAnySensorActive())
        {
        // Add user code here to proceed with sensor touching
        }
        // now we are ready to send all status variables to chart program
        // communication here
        // OUTPUT SmartSense2X_waSnsResult[x] <- Raw Counts
        // OUTPUT SmartSense2X_waSnsDiff[x] <- Difference
        // OUTPUT SmartSense2X_waSnsBaseline[x] <- Baseline
        // OUTPUT SmartSense2X_baSnsOnMask[x] <- Sensor On/Off
    }
}
```

Example 2. This code starts the user module and continuously scans the sensors; however, unlike Example 1, looping through the sensors is done in the user code. The communication section can be used to communicate values to a PC charting tool.

```
//-------------------------------------------------------------------------
// Sample C code for the SmartSense2X module
// Scanning all sensors continuously
//-------------------------------------------------------------------------

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all user modules

void main(void)
{
    BYTE bIndex;
    M8C_EnableGInt;
    SmartSense2X_Start();
    SmartSense2X_InitializeBaselines() ; //Scan all sensors first time, init baseline
    while (1) //Loop forever
    {
        for(bIndex=0; bIndex < SmartSense2X_TotalSensorCount; bIndex++)
        //Loop through all sensors
        {
            SmartSense2X_ScanSensor(bIndex);           // Scan sensors
            SmartSense2X_UpdateSensorBaseline(bIndex); // Run baseline filter
            // detect if any sensor is pressed
            if(SmartSense2X_bIsSensorActive(bIndex))
            {
            // Add user code here to process the sensor touching
            }
        }
        // now we are ready to send all status variables to chart program
        // communication here
        //
        // OUTPUT SmartSense2X_waSnsResult[x] <- Raw Counts
        // OUTPUT SmartSense2X_waSnsDiff[x] <- Difference
        // OUTPUT SmartSense2X_waSnsBaseline[x] <- Baseline
        // OUTPUT SmartSense2X_baSnsOnMask[x] <- Sensor On/Off
    }
}
```

Example 3. This code starts the user module and continuously scans the sensors when the background scanning feature is enabled. The communication section can be used to communicate values to a PC charting tool.

```
//-------------------------------------------------------------------------
// Sample C code for the SmartSense2X module
// Scanning all sensors continuously
//-------------------------------------------------------------------------

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all user modules
void main(void)
{
    M8C_EnableGInt;
    SmartSense2X_Start();
    SmartSense2X_InitializeBaselines() ; //scan all sensors first time, init baseline
    //
    // Loop Forever
```

```
    //
    //Trigger first scan
    SmartSense2X_ScanAllSensors(); //scan all sensors in array (buttons and
    //sliders)
    while (1)
    {
        while(!SmartSense2X_bIsScanComplete()); // wait until scanned

            SmartSense2X_UpdateAllBaselines(); //Update all baseline levels;
            //detect if any sensor is pressed
               if(SmartSense2X_bIsAnySensorActive())
               {
                    // Add user code here to proceed the sensor touching
               }
                // Do other tasks

    SmartSense2X_ScanAllSensors(); //scan all sensors in array (buttons and
    //sliders)
            // now we are ready to send all status variables to chart program
            // communication here
            //
            // OUTPUT SmartSense2X_waSnsResult[x] <- Raw Counts
            // OUTPUT SmartSense2X_waSnsDiff[x] <- Difference
            // OUTPUT SmartSense2X_waSnsBaseline[x] <- Baseline
            // OUTPUT SmartSense2X_baSnsOnMask[x] <- Sensor On/Off

    }
}
```

Version History

Version	Originator	Description
1.00	DHA	Initial version.
1.10	HPHA	1. Fixed problem with saving information for sliders. 2. Split APIs for linear and radial sliders optimize RAM and ROM usage. 3. Updated ClearSensors() function to eliminate erroneous behavior when carry is set before calling the API. 4. Resolution is now tuned when using the Manual Threshold setting. 5. Added FMEA APIs. 6. Added CY8C28xxx support. 7. Added new configuration for CY8C2xx45 family. 8. Added SetDebounce(), SetSliderIdac() functions to User Module API. 9. Limited Number of sensors on CY8C28xxx family.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.