SmartSense™ Auto-tuning CapSense® Sigma-Delta Datasheet SmartSense V 1.60

Copyright © 2010-2013 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks			API Memory (Bytes)				Pins per sensor
				Manual Threshold Mode		Automatic Threshold Mode		
	Digital	Analog CT	Analog SC	Flash	RAM	Flash	RAM	
CY8C21x34B. Use of flash, RAM, and pins varies by the number of sensors.								
IMO = 24 MHz Configuration	3	2	1	2143	29	3680	29	2-5
IMO = 12/6 MHz Configuration	4	2	1	2169	29	3733	29	2-5
Each additional CapSense® button	-	-	-	10	14	9	25	1
Each additional slider element	-	-	-	10	14	9	25	1
Static code and RAM increase when capacitive slider with five elements is used	-	-	-	694	105	689	160	-
Static code and RAM increase when slider diplexing is used	-	-	-	10	0	10	0	1

For one or more fully configured, functional example projects that use this user module, go to www.cypress.com/psocexampleprojects.

Features and Overview

- Implements CapSense® capacitive sensing in the CY8C21x34B family of PSoC® devices.
- Auto-tunes configurable system parameters in runtime to account for sensor, IC, and PCB characteristics.
- Supports up to 24 capacitive sensors and 4 sliders.[1]
- Supports parasitic sensor capacitance range of 5 pF to 45 pF.
- Detects touches as low as 0.1 pF; that is, a finger can be detected through up to 15 mm of glass or 5 mm of plastic.
- High immunity to AC line noise, other EMI, and power supply noise.
- Supports capacitive sensors configured as independent buttons and also as dependent arrays to form sliders.

- Provides a diplexing option that allows assigning two slider elements for every dedicated I/O pin.
- Supports slider resolution greater than the physical pitch through interpolation.
- Provides shield electrode for reliable operation with high parasitic capacitance and also in the presence of a water film.
- Enables guided sensor and pin assignments using the SmartSense™ Wizard.
- Supports proximity sensing up to a range of 5 cm (with onboard PCB trace).

[1)] The stack offset must be increased to 80 or higher (go to **Project** > **Settings** > **Stack page offset**).

Note This user module supports only C language projects. Assembly language (ASM) projects are not supported.

Quick Start

1. Select and place user modules that require dedicated pins (for example, I^2C and LCD). Assign ports and pins as needed.
2. Select and place the SmartSense User Module.
3. Right-click the SmartSense User Module in the Workspace Explorer to access the SmartSense Wizard (refer to the SmartSense Wizard section).
4. Set the required number of sensors, sliders, and radial sliders.
5. For sliders, enter the parameters specific to sliders.
6. Assign each of the sensors to an unused pin.
7. Enter the pins that will be connected to the external modulation capacitor (C_{mod}) and feedback resistor (Rb).
8. Right-click the SmartSense User Module in the Workspace Explorer to access the Properties list. Enter the pin that will be used to shield the sensor, if required (see the Parameters and Resources section).
9. Generate the application and switch to the Application Editor.
10. Adapt the sample code as required to implement independent sensors, sliding sensors, and a touchpad.
11. Program the PSoC device on the target board with the .hex file generated by PSoC Designer™.

Introduction

The SmartSense 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 consist of a conductive surface, usually a pad etched on the 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:

- Clean, 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.
- Improved reliability with no components that wear or have finite cycle life.
- Improved spill resistance due to lack of mechanical interface penetrations.
- Reduced tooling cost by eliminating penetrations or other mechanical features needed for mechanical input devices.

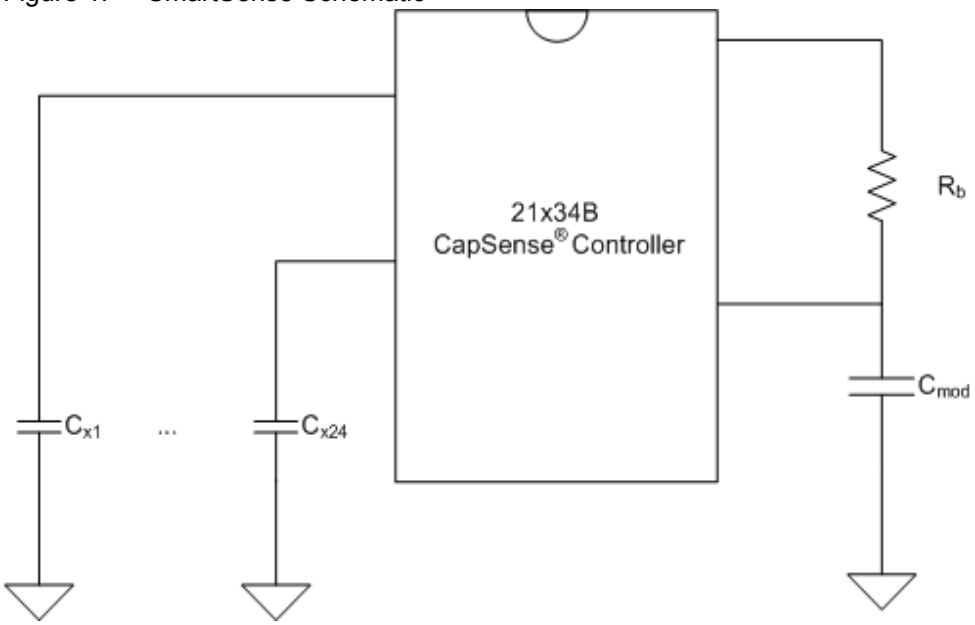
Similar to all Cypress CapSense solutions, SmartSense Auto-tuning offers superior immunity to conducted and radiated noise interference, with the additional benefit of auto-tuning. Auto-tuning gives run time compensation for IC and PCB characteristics and environmental changes to ensure reliable sensor 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 the CapSense system parameters. Auto-tuning automatically compensates for such changes, with no need for retuning. Furthermore, 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 SmartSense User Module consists of PCB level, IC level, and software components.

PCB Level

Figure 1 shows a schematic of the SmartSense User Module. 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, section: PCB layout guidelines for more information on PCB level CapSense implementation.

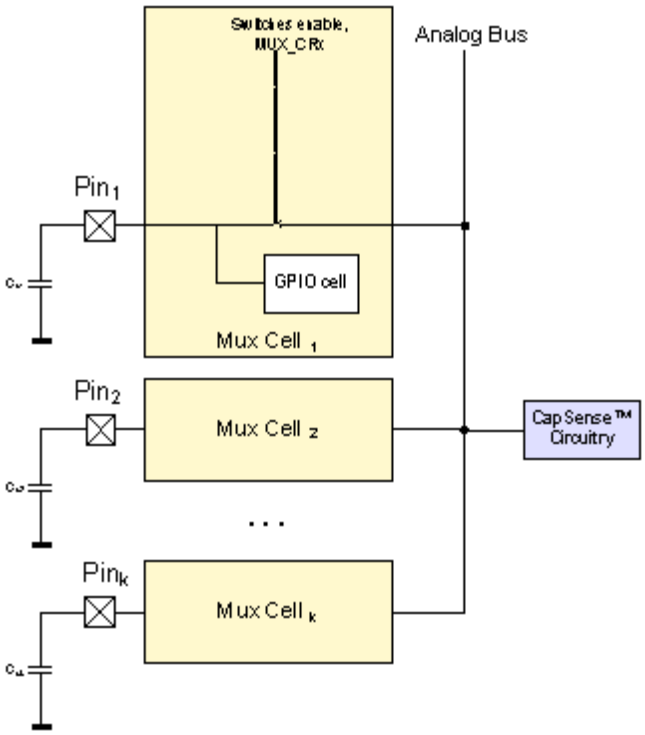
Figure 1. SmartSense Schematic



IC Level

The CY8C21x34B devices have an Analog MUX (AMUX) Bus that enables connecting capacitive sensing analog circuitry to any PSoC pin. The SmartSense User Module connects the active sensor to the AMUX Bus, which enables 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 2.

Figure 2. CY8C21x34B AMUX Block Diagram



Software

The attributes of the SmartSense software component are:

- Auto-tuning algorithms configure the analog capacitive sensing circuitry in runtime for optimal performance. These algorithms take into account the 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.
- In the case of consecutive, dependent sensors (for example, sliders), 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 given number of slider elements.

Recommended Reading

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

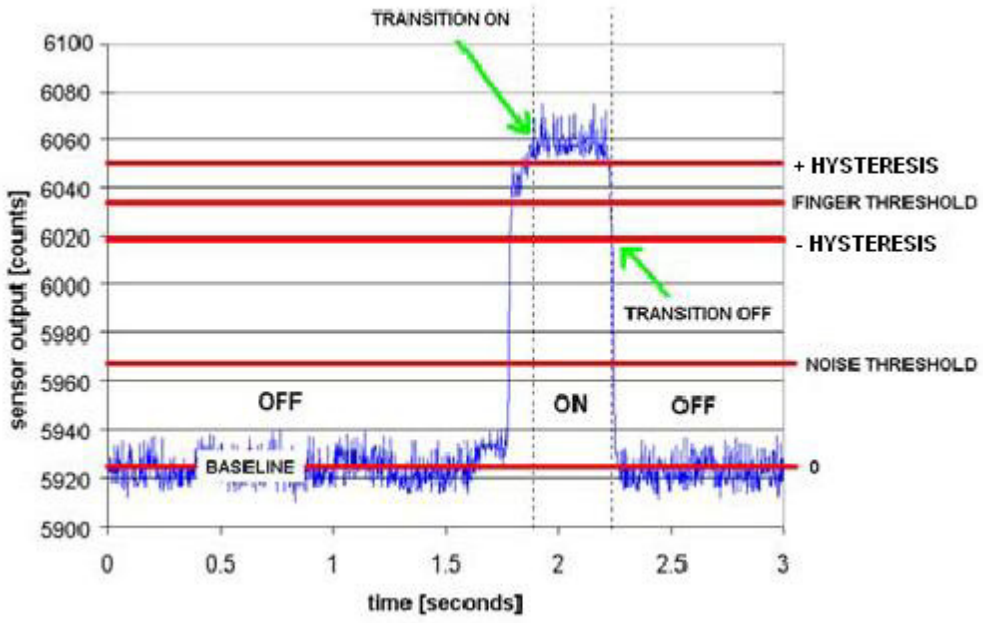
- *CY8C21x34B Series PSoC Mixed Signal Array Technical Reference Manual, section:* Two Column Limited Analog, Digital Clocks, I/O Analog Multiplexer
- Getting Started with CapSense Design Guide
- CY8C21x34/B Design Guide
- *Design Aids - CapSense Data Viewing Tool* – AN2397
- User Modules - Software Implementation of Universal Asynchronous Transmitter – AN2399

Capacitance Sensing Implementation

Buttons

CapSense buttons are analogous to mechanical push-buttons. They are used for discreet controls such as on/off switches, function keys, menu keys, and so on. API functions monitor the capacitance signals from each sensor and compare them to the threshold levels calculated by the SmartSense auto-tuning algorithms. When a sensor is touched, its capacitance signal increases; if SmartSense decision logic determines that the increase is enough, that sensor is activated. Figure 3 shows a typical signal (blue line) from a sensor when it is being activated. SmartSense automatically sets the thresholds (red lines) based on your input to provide the required system behavior.

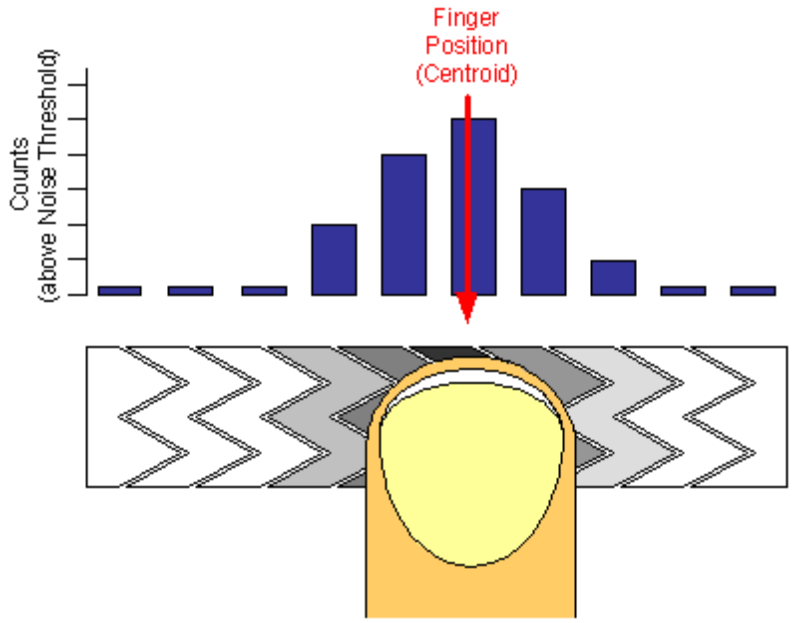
Figure 3. Capacitance Signal from a Sensor when it is being Activated



Sliders

CapSense sliders are analogous to mechanical potentiometers. Sliders are used for controls that require a continuum of levels, such as lighting dimmers, volume control, graphic equalizers, speed controls, and so on. A CapSense slider is implemented with an array of adjacent sensors. When a slider is actuated by a finger, several adjacent sensors register an increase in capacitance signal. This is shown in Figure 4. The exact position of the touch is found by computing the centroid location of the set of activated sensors. The practical minimum number of sensors in a slider is five, and the maximum is limited only by the number of available I/O pins on the PSoC device.

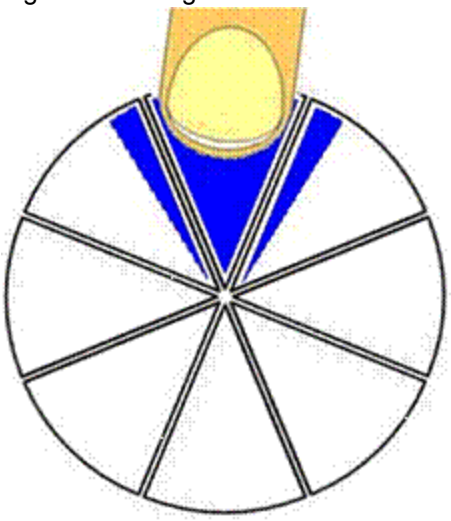
Figure 4. Interpolated Centroid Position of a Finger on a Slider



Radial Sliders

SmartSense supports two slider types: linear and radial. Linear sliders have a beginning and an end, whereas radial sliders, as shown in Figure 5, do not. In either case, when a touch occurs, the centroid algorithm takes into account the signals from sensors adjacent to the sensor with the largest signal to interpolate the exact position of the touch. Radial sliders are not diplexed. The SmartSense User Module has two special API functions for radial sliders. The first function SmartSense_wGetRadiaPos() returns the centroid location, and the second function SmartSense_wGetRadialInc() returns the finger shift in resolution units. When the finger moves in a clockwise direction, SmartSense_wGetRadialInc() returns a positive offset. The reference point (0) is located in the center of the first sensor. The Resolution is limited to (number of pins used for sensors – 1) x 2^8 – 1 or (2 x pins used for sensors – 1) x 2^8 – 1 for diplexed sliders.

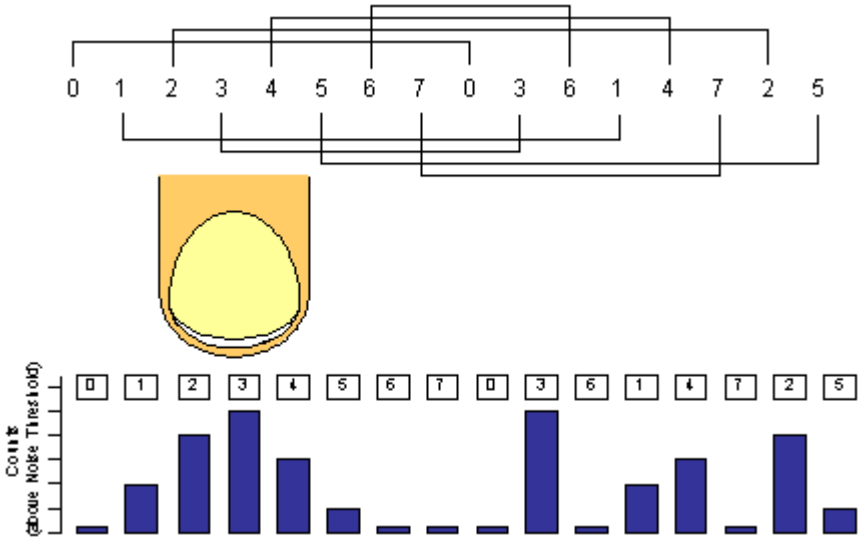
Figure 5. Finger touches Radial Slider



Diplexing

When Diplexing is used, each pin on the CapSense controller that is designated as a slider element is mapped to two physical locations in the array of slider sensors. The first (or numerically lower) half of the physical locations is mapped according to the port pin assigned in the SmartSense Wizard. The second (or upper) half of the physical sensor locations is automatically mapped using the pattern shown in Figure 6:

Figure 6. Indexing of Diplexed Slider Array by SmartSense



The close proximity of strong signals in the lower half of the slider results in the same levels aliased into the upper half. However, in the upper half, the results are scattered and noncontiguous. The centroid algorithm searches for strong adjacent sets of signals to declare the resolved slider position. The pattern used for mapping upper half sensors ensures that a valid signal pattern in one half does not result in a valid signal pattern in the other half, as shown in Figure 6.

Ensure that the mapping of sensors to pins on the PCB matches the Index by 3 sequence used by the diplexing algorithm. The diplex sensor index table is automatically generated by the SmartSense Wizard when you select diplexing. Table 1 shows the diplexing sequences for up to 22 slider segments diplexed into 10 PSoC I/O pins.

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

Slider Segment Selection Guidelines for the Diplex Slider

Selecting the number of segments needed for a slider mainly depends on the physical length of the slider. However, special care must be taken when you decide the number of segments for a diplexing slider.

In a diplexing slider design, one sensor is used as two different physical slider segments to increase the physical length of slider. The number of segments that are completely covered by a finger touch must be less than the number of sensors between two segments derived from the same sensor. This ensures the proper working of the diplex slider.

For example, in the case of a 10-segment slider (5 sensors), two slider segments derived from sensor 3 are separated by only two sensors (sensor 4 and 0). In this case, a finger touch must not completely cover more than two sensor segments to ensure the proper working of the slider.

For a 12-segment slider, one finger touch must not cover more than 3 segments. Similarly, for a 18-segment slider, one finger touch must not completely cover more than 4 segments.

Interpolation and Scaling

In slider applications, it is necessary to determine the position with finer granularity than the physical pitch of the sensors. SmartSense accomplishes this by interpolating the position of the finger using a centroid calculation on the signal from sensors adjacent to the sensor with the largest signal. The array is first scanned to verify that the signal pattern is valid. The passing requirement is for some 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 wanted scale. This is handled in the high level APIs.

Slider sensor count and resolution are set in the SmartSense 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 listed 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 calculated using this equation:

Resolution = (Number of Sensors - 1) x Multiplier

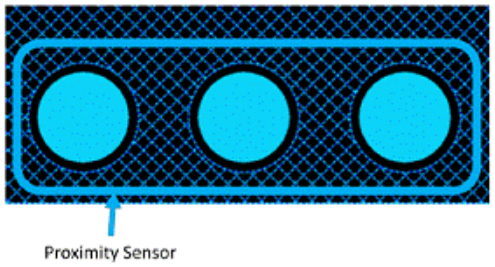
The centroid is held in a 24-bit unsigned integer, and its resolution is a function of the number of sensors in the slider and the multiplier.

Note When designing layout for sliders, ensure that the parasitic capacitance of slider segments are reasonably well matched.

Proximity

A proximity sensor detects the presence of a hand or another conductive object before it makes contact with 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. One implementation of a proximity sensor consists of a long trace on the perimeter of the user interface, as shown in the following figure:

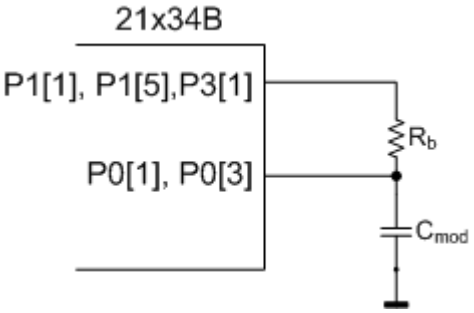
Figure 7. Long Trace on the Perimeter of the User Interface



Selecting Feedback Components

The user module requires an external modulation capacitor C_{mod} and a modulator feedback resistor R_b. The capacitor can be connected to the P0[1], P0[3] port pins, and Vss ground. The feedback resistor R_b can be connected to port pins P1[1], P1[5], P3[1], and the capacitor pin. The pins are selected by the user module parameter setting. Do not use pins selected for modulator component connection for any other purposes.

Figure 8. External Components Connection



The recommended value for the modulation capacitor (Cmod) is 10 nF. A ceramic capacitor should be used. The temperature capacitance coefficient is not important.

The value of modulator feedback resistor Rb should be 15K.

Permanent connection of a low-resistance feedback resistor to the P1[1] pin can cause ISSP programming faults. Use another pin for this. Use the P3[1] pin on packages where it is available. Refer to the Knowledge Base article, How to add P1[4] for Rb in CY8C21234B SmartSense Device, if you want to set the modulator feedback resistor to the P1[4] pin.

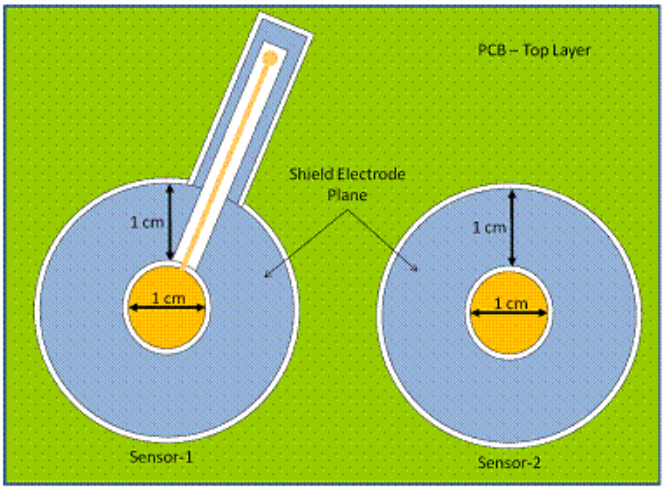
Future versions of the SmartSense User Module may allow use of additional pins for connecting the feedback resistor that allow the use of a second I^2C port.

Driven Shield Electrode

A driven shield electrode is an optional feature. The benefits of this feature include improved sensor sensitivity required for proximity applications and the prevention of false sensor triggering when water is present on the overlay.

A shield electrode must be located behind the sensors for proximity applications, and outside the sensor electrode for waterproofing. The following figure shows a driven shield electrode placed outside the sensor to implement water proofing.

Figure 9. Driven Shield Electrode PCB Layout



Power Supply Requirement

Table 3. SmartSense Power Supply Requirement

Parameter	Min	Typical	Max	Unit	Test Conditions and Comments
V_{DD}	2.7	5	5.25	V	

Placement

The blocks for the user module are automatically placed when the user module is instantiated; alternate placements are not available. The user module supports two selection options as the project IMO settings:

- SmartSense for IMO = 24 MHz, uses only 3 digital blocks
- SmartSense for IMO = 12 MHz and 6MHz, uses all 4 digital blocks

The SmartSense User Module for IMO = 24 MHz consumes 3 digital blocks (DBB00, DBB01, and DCB02). The DCB03 block is available for user purposes. The SmartSense User Module for IMO = 12 MHz and 6 MHz consumes all 4 digital blocks (DBB00, DBB01, DCB02, and DCB03).

User modules that consume specific pin resources, including the LCD and I2CHW, must be placed before establishing port pin connections for the SmartSense User Module. The configuration selections are reflected in the Wizard when it is opened.

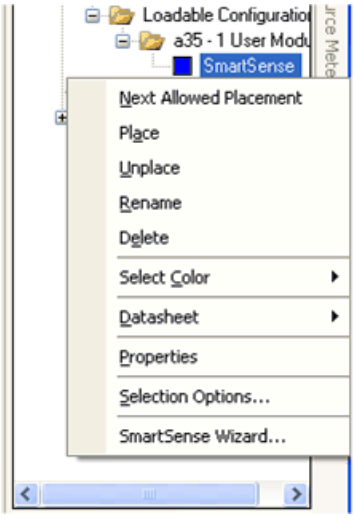
Avoid P1[0] and P1[1] when placing capacitive sensor connections. These pins are used for programming the part and may have excess routing capacitance affecting sensor sensitivity and noise.

SmartSense Wizard

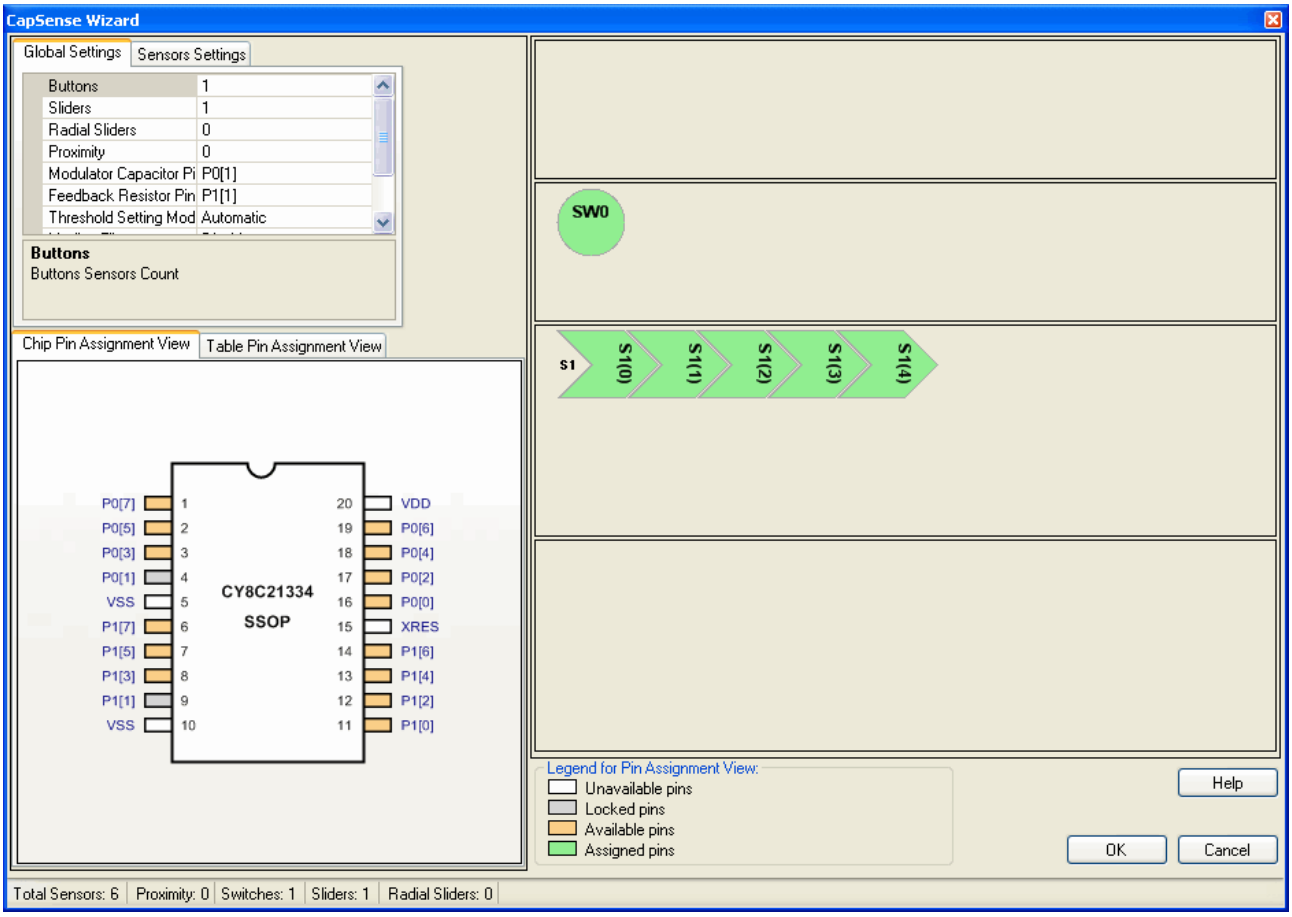
The SmartSense Wizard is used to set up the pinout for your CapSense buttons, sliders, and proximity sensors. You can choose a configuration and assign the buttons and segments using a drag and drop interface.

Wizard Access

1. To access the Wizard, right-click any block of the SmartSense in the Device Editor Interconnect View, then select the SmartSense Wizard with a left-mouse click.



2. The Wizard opens showing the numeric entry boxes for the number of sensors and the number of slider sensors.



Wizard Pin Legend

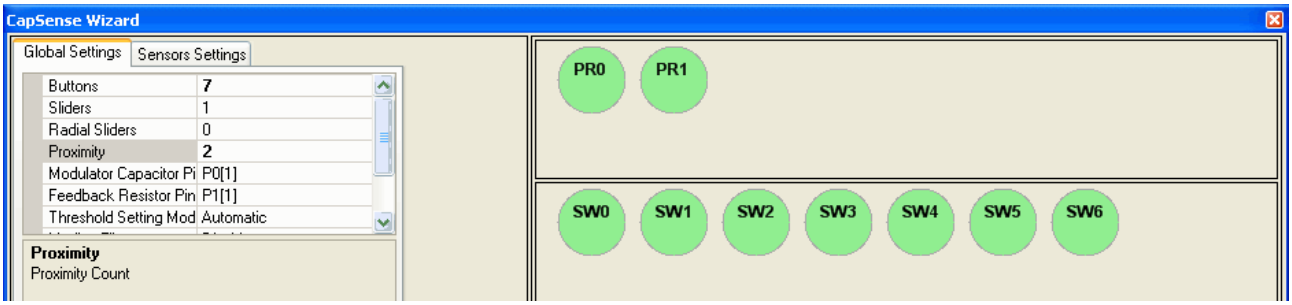
White – The pin cannot be used as a CapSense input.

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.

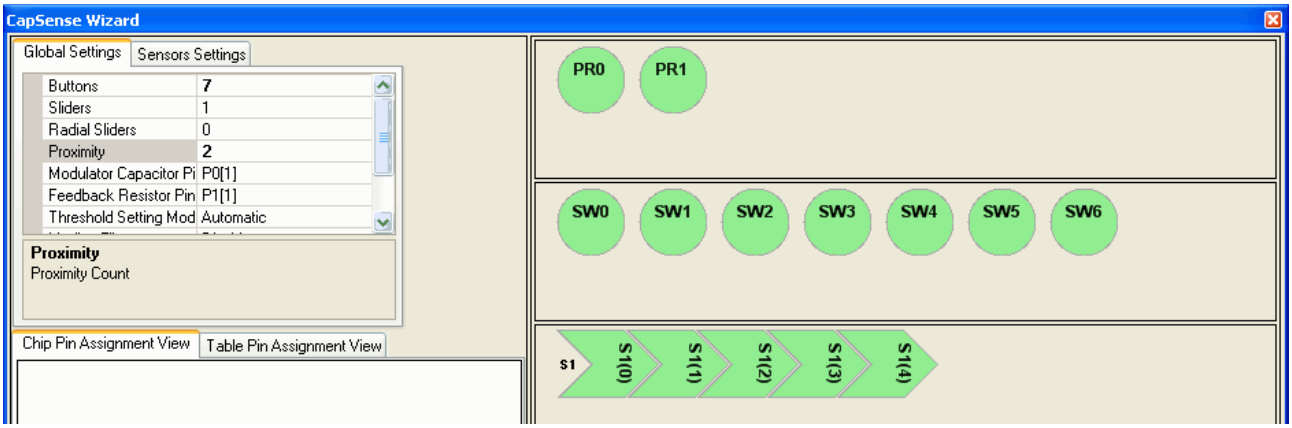
Orange – The pin is available for assignment.

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

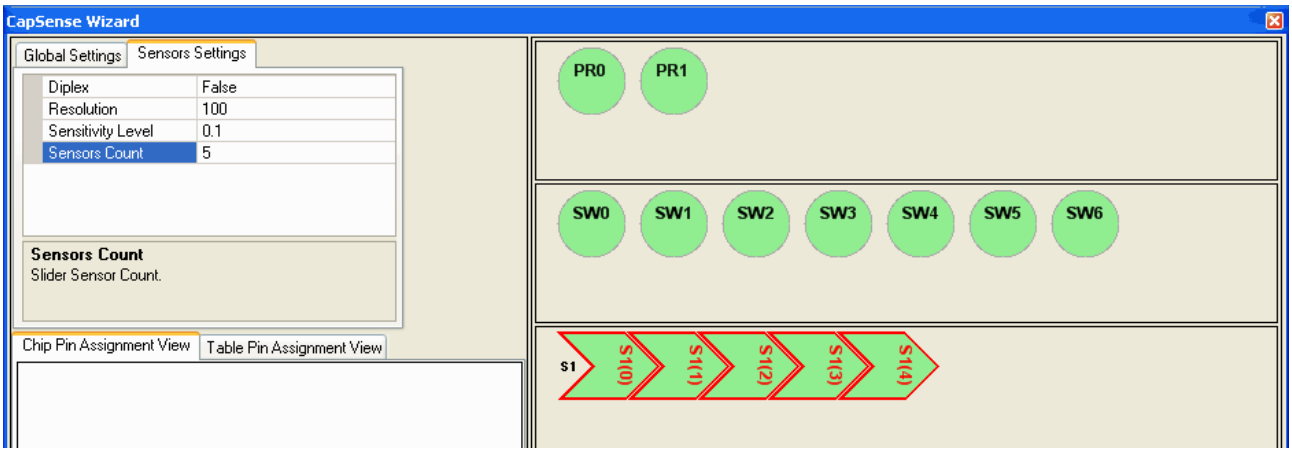
3. Type the number of independent sensors. The number of sensors is limited to the number of pins available.



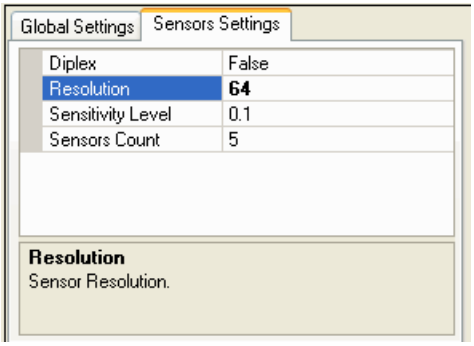
4. Type the number of sliders. X-Y touchpads require two sliders (only one is selected below).



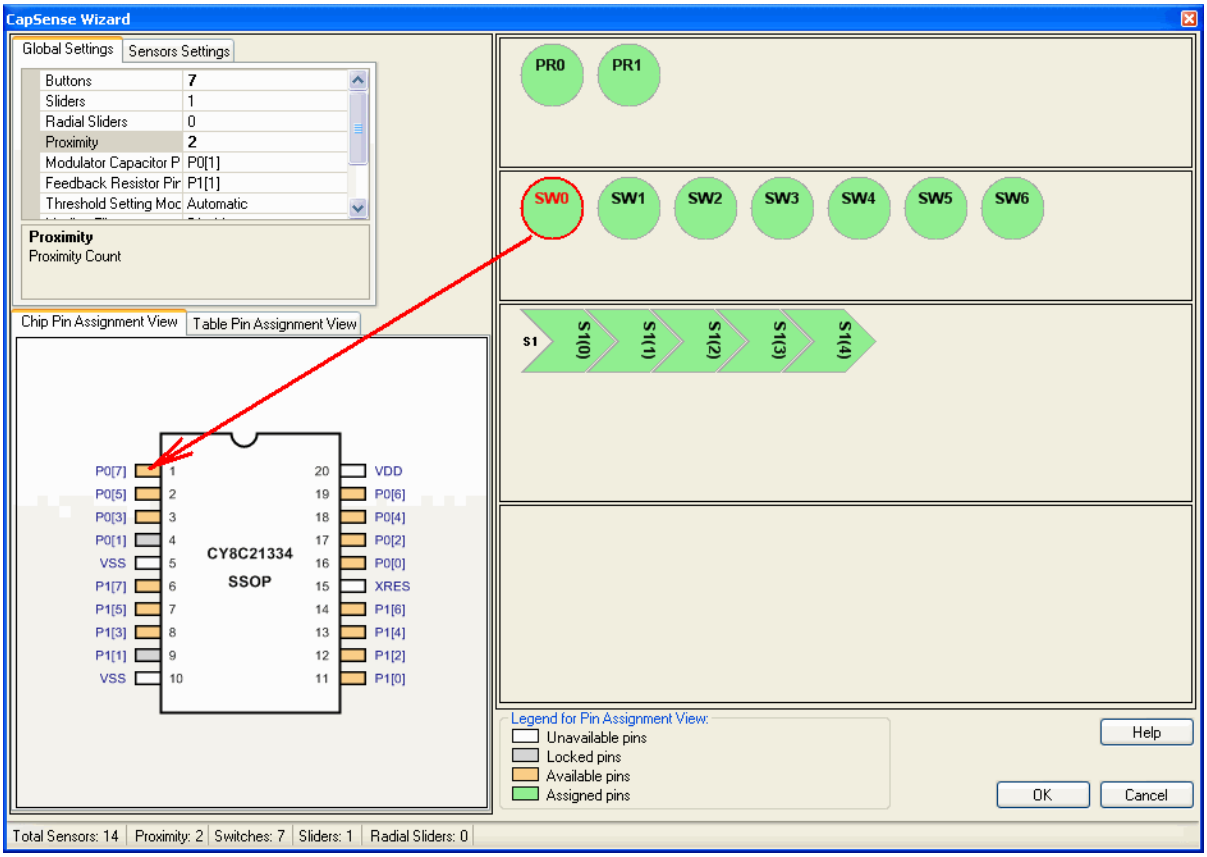
5. Click a slider to enter the settings for that slider. Select the **Sensor Settings** tab. Type the number of sensor elements in the slider. The practical minimum number of sensors in a slider sensor is five. The maximum is limited by the pin count.



6. 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 2^8 - 1 for diplexed sliders. The software will attempt to interpolate touches to this resolution based on the relative strength of the signals on adjacent slider sensors.



7. 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 outlined in the previous section on Diplexing. See the Diplexing section to find Diplexing tables for pin connections.
8. Left-click a sensor and drag it onto 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.



9. Repeat this procedure for the remaining independent sensors.
10. Mapping individual slider sensors onto physical port pins is the same as for individual sensors.
11. Enter the pin that the external Modulation Capacitor will be connected to in the SmartSense Wizard.

12. Enter the pin that the external Feedback Resistor will be connected to in the SmartSense Wizard.
13. Click **OK** to accept 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 now.

If you want 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 unassigned and you can then reassign it.

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 SmartSense_Table.asm.

Wizard Parameters description

Wizard Parameters are described in the Global Settings and Sensor Settings tabs.

Global Settings Tab

Global Settings Tab consists of following parameters: Buttons, Sliders, Radial Sliders, Modulator Capacitor Pin and Feedback Resistor Pin.

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

This is the number of proximity sensors.

Modulator Capacitor Pin

This parameter sets the pin to connect to the external modulator capacitor (C). Choose from the available pins P0[1] and P0[3].

Feedback Resistor Pin

This parameter sets the pin to connect to the external feedback resistor (Rb). Choose from the available pins: P1[1], P1[5], and P3[1]. Some pins are not available on some device packages. If some of these pins are used for other purposes (for example, allocated for sensor connection), they are not available for selection in the user module parameter list. Future versions of the user module may allow additional pins to be used for connecting the feedback resistor. This allows the use of a second I2C port on packages that have no P3 port. Use pins P1[5] or P3[1] to avoid programming problems.

Threshold Setting Mode

Select between Automatic and Manual threshold setting.

Median Filter

The median filter looks at the three most recent rawcount samples from a sensor and reports the median value. It is used to remove short noise spikes. This setting is not applicable to proximity sensors.

IIR Filter

This infinite impulse response (IIR) filter of order 1 reduces noise in the conversion result (raw count). The filter must be disabled by default. This setting is not applicable to proximity sensors.

IIR Filter Coefficient

Coefficient of 1/4 implies that the filtered output should be 1/4th of the present rawcount and 3/4th of the previous rawcount.

Sensor Settings Tab

The Sensor Settings tab consists of the following parameters: Finger Threshold, Sensitivity Level, Diplex, Resolution, and Sensors Count. Additional parameters are applicable to proximity sensors: Approaching Speed, IIR Filter, and IIR Filter Coefficient.

Finger Threshold

This parameter is used to adjust the button finger threshold of each sensor. Set this value to 70-80% of the monitored SmartSense_baSnsSignal.

Diplex

This option enables/disables diplex for slider. See the "Diplexing" section of the user module data-sheet 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

This is the number of physical sensors in slider or radial slider.

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 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

Adjusts baseline update rate for different sensor approaching speeds. Options: Fast, Medium, and Slow. The default value is Fast. This setting is applicable to proximity sensors only.

IIR Filter

This infinite impulse response (IIR) filter of order 1 reduces noise in the conversion result (raw count). The filter is enabled by default. This setting is applicable to proximity sensors only.

IIR Filter Coefficient

Coefficient of 1/4 implies that the filtered output is 1/4th of the present rawcount and 3/4th of the previous rawcount. This setting is applicable to proximity sensors only.

Tables Produced by the Wizard

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 SmartSense_Table.asm.

Sensor Table

The Sensor table 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:

```
SmartSense_Sensor_Table:
_SmartSense_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 plus 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 multiplier by which the slider's centroid is scaled to achieve the resolution specified in the SmartSense Wizard.

```
SmartSense_Group_Table:
_SmartSense_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
 SmartSense_Diplex_Table:
_SmartSense_Diplex_Table:
 db >DiplexTable_0, <DiplexTable_0
 db >DiplexTable_1, <DiplexTable_1
```

Finger Threshold Table

The Finger Threshold Table defines the Normalized Finger Threshold for each sensor:

```
SmartSense_Finger_Threshold_Table:
_SmartSense_Finger_Threshold_Table:
   db 100   ; Buttons
   db 255   ; Buttons
```

Sensitivity Level Table

The Sensitivity Level Table defines the Sensor Sensitivity for each sensor:

```
SmartSense_Sensitivity_Level_Table:
_SmartSense_Sensitivity_Level_Table:
   db 1   ; Buttons
   db 3   ; Buttons
```

Parameters and Resources

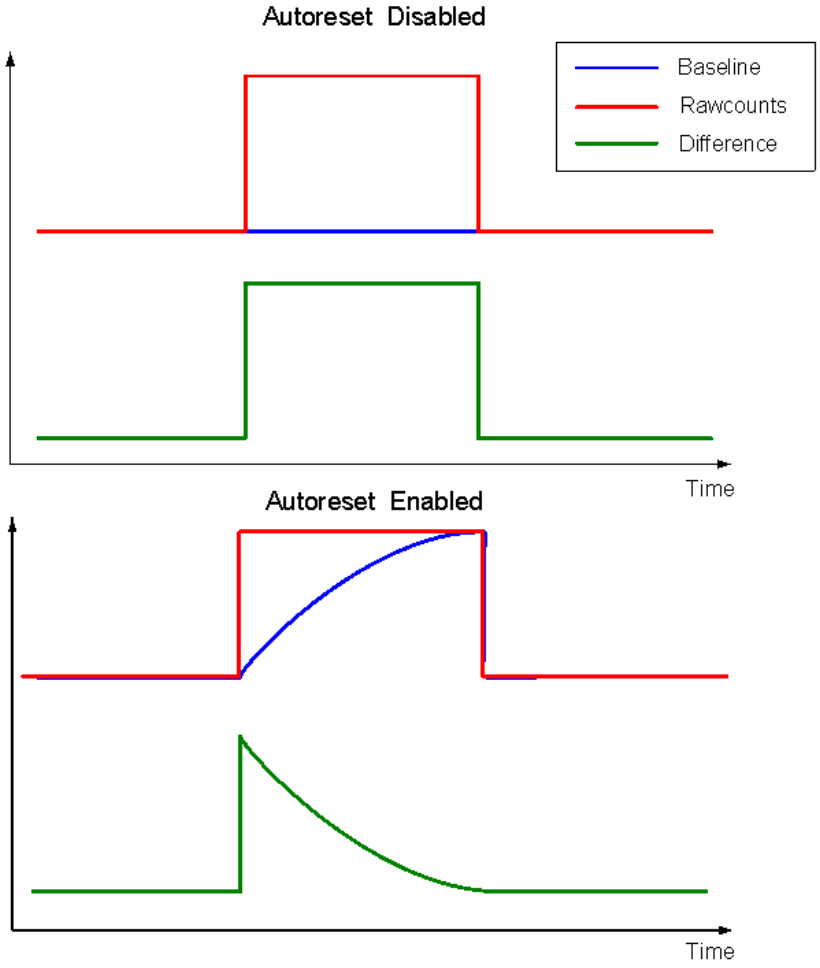
After completing the configuration and I/O pin assignment in the SmartSense Wizard, the user module parameters must be set. Note that for any user module parameter change to take affect, the project must be regenerated.

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 10 illustrates this parameter's effect on the 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 10. 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.

ShieldElectrodeOut

The shielding electrode signal source can be selected from one of the free digital row buses (Row_0_Output_1 - Row_0_Output_3). Each row output can be routed to one of three pins. Set the Row LUT Function to A. The default setting is None.

Sensor Scan Time

The time required to scan a sensor depends on its parasitic capacitance (Cp). If you know the Cp of a sensor, its scan time can be calculated using the following table:

Table 4. Scan time for different values of IMO, sensitivity level and Cp

IMO = 24 MHz					
Sensitivity Level = 0.1		**Sensitivity Level = 0.2/0.3**		**Sensitivity Level = 0.4/0.5**	
Cp Range (pf)	**Scan Time (us)**	**Cp Range (pf)**	**Scan Time (us)**	**Cp Range (pf)**	**Scan Time (us)**
Up to 11	1360	Up to 11	770	Up to 11	504
12 – 21	2020	12 – 22	1360	12 – 22	770
22 – 37	3400	22 – 37	2020	22 – 37	1360
37 – 45	6100	38 – 45	3400	38 – 45	2020

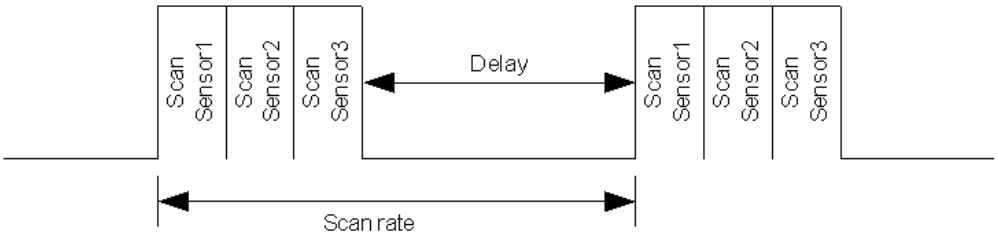
IMO = 12 MHz					
Sensitivity Level = 0.1		**Sensitivity Level = 0.2/0.3**		**Sensitivity Level = 0.4/0.5**	
Cp Range (pf)	**Scan Time (us)**	**Cp Range (pf)**	**Scan Time (us)**	**Cp Range (pf)**	**Scan Time (us)**
Up to 11	2400	Up to 11	1400	Up to 11	900
12 – 22	3700	12 – 22	2400	12 – 22	1400
22 – 37	6400	22 – 37	3700	22 – 37	2400
38 – 45	11700	38 – 45	6400	38 – 45	3700

IMO = 6 MHz					
Sensitivity Level = 0.1		**Sensitivity Level = 0.2/0.3**		**Sensitivity Level = 0.4/0.5**	
Cp Range (pf)	**Scan Time (us)**	**Cp Range (pf)**	**Scan Time (us)**	**Cp Range (pf)**	**Scan Time (us)**
Up to 11	4800	Up to 11	2800	Up to 11	1680
12 – 22	7600	12 – 22	4800	12 – 22	2800
22 – 37	13200	22 – 37	7600	22 – 37	4800
38 – 45	24000	38 – 45	13200	38 – 45	7600

Sensor Scan Rate Selection Guidelines

Scan rate is the rate at which sensors are scanned. An example of a 3-button design is shown in the following figure. All sensors in the design are scanned sequentially and there is a delay before the next sensor scan is initiated.

Figure 11. Typical Sensor Scan



To ensure proper working of the baseline, it is recommended to maintain a scan rate of 15 ms or more in a design. This indicates that a design with less number of sensors must add a delay to make the sensor scan rate equal to or greater than 15 ms. A design with more number of sensors may not need any delay as scanning all sensors itself may consume 15 ms. A good design may put the CapSense controller in sleep mode, instead of the firmware delay routine, to create a low power design.

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 given 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. You can inspect these values for debugging purposes, however. For example, you can use a charting tool to display the contents of the arrays. There several global arrays:

- SmartSense_waSnsBaseline[]

- SmartSense_waSnsResult[]
- SmartSense_waSnsDiff[]
- SmartSense_baSnsOnMask[]

SmartSense_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:

- SmartSense_UpdateAllBaselines()
- SmartSense_UpdateSensorBaseline()
- SmartSense_InitializeBaselines()

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

- SmartSense_ScanSensor()
- SmartSense_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 SmartSense_baSnsOnMask[] data is updated by these functions:

- SmartSense_bIsSensorActive()
- SmartSense_bIsAnySensorActive()

SmartSense_Start

Description:

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

C Prototype:

```
void  SmartSense_Start()
```

Assembly:

```
lcall  SmartSense_Start
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_Stop

Description:

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

C Prototype:

```
void  SmartSense_Stop()
```

Assembly:

```
lcall  SmartSense_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_Resume

Description:

Resumes the user module operation after SmartSense_Stop call.

C Prototype:

```
void SmartSense_Resume()
```

Assembly:

```
lcall SmartSense_Resume
```

Parameters:

None

Return Value:

None

Side Effects

**

SmartSense_ScanSensor

Description:

Scans the selected sensor. Each sensor is uniquely identified by its position in the Sensor Table. This position or Sensor Number is assigned by the SmartSense Wizard.

C Prototype:

```
void SmartSense_ScanSensor(BYTE bSensor);
```

Assembly:

```
mov A,  bSensor
```

```
lcall SmartSense_ScanSensor
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects

**

SmartSense_ScanAllSensors

Description:

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

C Prototype:

```
void SmartSense_ScanAllSensors()
```

Assembly:

```
lcall  SmartSense_ScanAllSensors
```

Parameter:

None

Return Value:

None

Side Effects:

**

SmartSense_UpdateSensorBaseline

Description:

The historical count value, calculated independently for each sensor, is called the sensor's baseline. This baseline is updated using the “Bucket Method”.

The Bucket Method uses the following algorithm:

1. Every time the SmartSense_UpdateSensorBaseline() is called, a difference count is calculated by subtracting the previous baseline from the raw count value. This difference is stored in the SmartSense_waSnsDiff[] array.
2. If Sensors Autoreset is disabled, each time SmartSense_UpdateSensorBaseline() is called, the difference count is compared to the noise threshold. If the difference is below the noise threshold, it is accumulated into a virtual bucket. If the difference is above the noise threshold, the bucket is not updated. If Sensors Autoreset is enabled, the difference is accumulated into a virtual bucket regardless of the noise threshold parameter.
3. After the accumulated difference counts in the virtual bucket reach the BaselineUpdateThreshold, the baseline is incremented by one and the bucket is reset to 0.
4. If the difference count is below the noise threshold, the value held in the waSnsDiff[] array is reset to 0. As a result, this array does not contain elements with values greater than 0, but below the NoiseThreshold.

C Prototype:

```
void SmartSense_UpdateSensorBaseline(BYTE bSensorNum)
```

Assembly:

```
mov   A,   bSensorNum
lcall  SmartSense_UpdateSensorBaseline
```

Parameter:

A => Sensor Number

Return Value:

None

Side Effects:

**

SmartSense_UpdateAllBaselines

Description:

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

C Prototype:

```
void SmartSense_UpdateAllBaselines()
```

Assembly:

```
lcall  SmartSense_UpdateAllBaselines
```

Parameter:

None

Return Value:

None

Side Effects:

**

SmartSense_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 SmartSense_baSnsOnMask[] array.

C Prototype:

```
BYTE  SmartSense_bIsSensorActive(BYTE bSensorNum)
```

Assembly:

```
mov  A,  bSensorNum
lcall SmartSense_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:

**

SmartSense_bIsAnySensorActive

Description:

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

C Prototype:

```
BYTE  SmartSense_bIsAnySensorActive()
```

Assembly:

```
lcall SmartSense_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:

**

SmartSense_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 SmartSense Wizard. This function is available only if slider is defined by the SmartSense Wizard.

C Prototype:

```
WORD  SmartSense_wGetCentroidPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall SmartSense_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 SmartSense_blsSensorActive() routine to determine which slider segments are touched.

SmartSense_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 SmartSense Wizard.

C Prototype:

```
WORD  SmartSense_wGetRadialPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall SmartSense_wGetRadialPos
```

Parameters:

bSnsGroup A => Group Number

This parameter is the Group Number of radial slide. You can get its number through the SmartSense Wizard on the left side of the 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 SmartSense_blsSensorActive() routine to determine which slider segments are touched.

SmartSense_wGetRadialInc

Description:

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

C Prototype:

```
WORD  SmartSense_wGetRadialInc(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall SmartSense_wGetRadialInc
```

Parameters:

bSnsGroup A => Group Number

This parameter is the Group Number of radial slide. You can get its number through the SmartSense 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, 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 SmartSense_wGetRadialPos(), because it uses internal data stored in global variables by the latter.

SmartSense_InitializeSensorBaseline

Description:

Loads the SmartSense_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 SmartSense_InitializeSensorBaseline(BYTE bSensorNum)
```

Assembly:

```
mov A, bSensorNum
lcall SmartSense_InitializeSensorBaseline
```

Parameters:

A => Sensor Number

Return Value:

None

Side Effects:

**

SmartSense_InitializeBaselines

Description:

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

C Prototype:

```
void SmartSense_InitializeBaselines()
```

Assembly:

```
lcall SmartSense_InitializeBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_ClearSensors

Description:

Clears all sensors to the nonsampling state by sequential disconnecting each sensor from the analog mux bus and shunting each sensor to ground.

C Prototype:

```
void  SmartSense_ClearSensors()
```

Assembly:

```
lcall  SmartSense_ClearSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_SetDefaultFingerThreshold

Description:

Initializes Sensor Finger Threshold, Sensor Noise Threshold, Sensor Negative Noise Threshold, Sensor Baseline update threshold, Sensor Hysteresis, Sensor Negative Noise Threshold and Low Baseline Reset.

C Prototype:

```
void  SmartSense_SetDefaultFingerThreshold()
```

Assembly:

```
lcall  SmartSense_SetDefaultFingerThreshold
```

Parameters:

None

Return Value:

None

Side Effects:

**

SmartSense_LoadFixedParameters

Description:

Computes and loads values to low baseline reset, hysteresis, noise threshold, and baseline update threshold.

C Prototype:

```
void  SmartSense_LoadFixedParameters(BYTE bSensorNumber)
```

Assembly:

```
mov A, bSensorNumber
lcall SmartSense_LoadFixedParameters
```

Parameters:

A => Sensor Number

Return Value:

None.

Side Effects:

**

SmartSense_UpdateSensorSignal

Description:

Updates the SmartSense_baSnsSignal[] array with Normalized difference count.

C Prototype:

```
void  SmartSense_UpdateSensorSignal(BYTE bSensorNumber)
```

Assembly:

```
mov A, bSensorNumber
lcall SmartSense_UpdateSensorSignal
```

Parameters:

A => Sensor Number

Return Value:

None.

Side Effects:

**

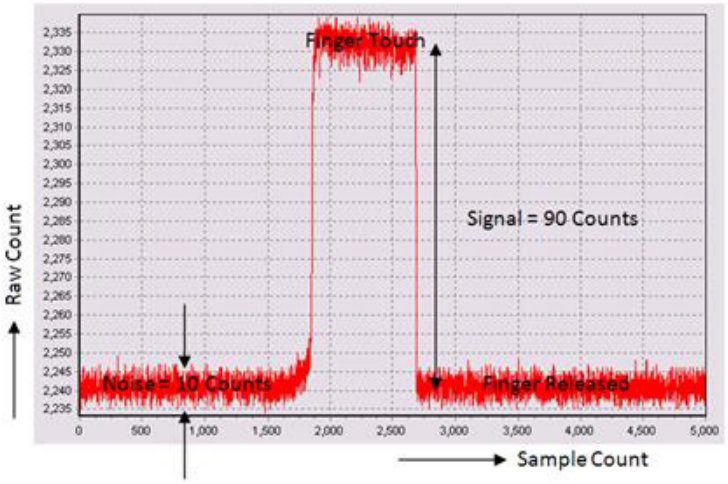
Step-By-Step Tuning Guide

Here are the basic guidelines for configuring the user module in a typical CapSense application using the CY3280-21x34 UCC board as a test example:

1. Prepare the target board. Assemble the target application PCB and fix the overlay on it (for this example an acrylic overlay of 1.5 mm is used). Use nonconductive glue or special adhesive tape for this purpose. Avoid air gaps between the PCB and the overlay, because it can reduce the sensitivity substantially and cause multiple false button triggers due to the air gap shifting under the touch.
2. Set up a real-time monitoring tool to monitor data. For the SmartSense configuration, use a PC charting tool that enables you to observe one or more data series in real-time. The sensor rawcount (waSnsResult), baseline (waSnsBaseline), difference counts (waSnsDiff) , signal (baSnsSignal), and button finger threshold (baBtnFThreshold) must be observed during the tuning process. You can use the UART-USB Bridge or the I2C-USB Bridge for this. Do not use the LCD or any other numerical displays to monitor data, because they are slow and do not allow visualizing the data dynamics.
3. The default configurations to start with are:
    - Set IMO = 24 MHz, CPU_Clock = SysClk/1
    - Place the SmartSense User Module configuration for IMO = 24 MHz
    - Disable "Sensor Autoreset". Set "Debounce" to 3, and set "Shield Electrode Out" to None.
    - Under Global Settings in the CapSense wizard, assign one sensor under the category "Button". Select the modulation Capacitor and Feedback resistor pin as well. Set the threshold setting mode as manual, and disable the Median and IIR filters. Ensure that in the board, the Rb value is 15K and the C_{mod} is 10 nF.
    - Under the Sensor Settings tab, set "Sensitivity Level" to Low and "Finger Threshold" to 96.
4. Generate/Build the project with the sample firmware code provided in this user module datasheet.
5. Monitor the sensor raw count and calculate the SNR. Figure 11 shows the monitored sensor rawcount. According to CapSense best practices, the SNR for a robust design must be greater than 5. The mon-

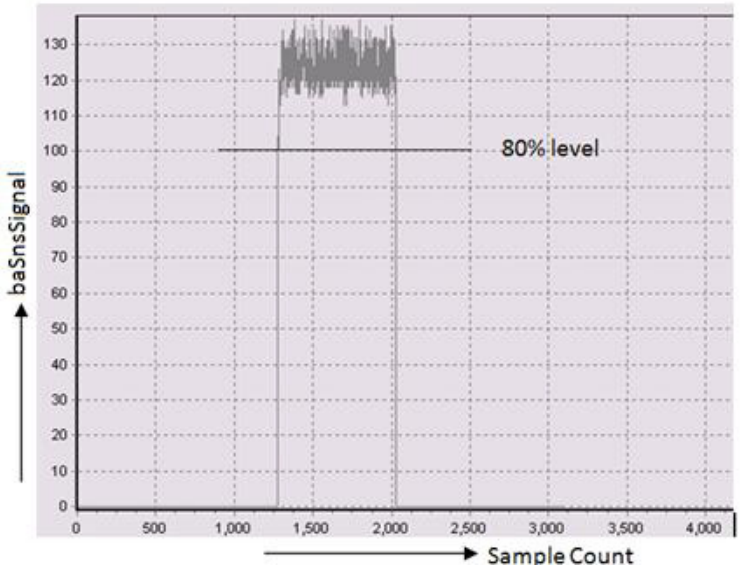
itored SNR in this case is found to be 9, which is above 5. Read SNR improvement techniques in step 7 if the SNR is found to be less than 5.

Figure 12. Monitored Sensor Rawcount



6. To complete the tuning process, you must also set the finger threshold. Monitor the sensor signal baSnsSignal. Determine 80% of its maximum value of which the following sample is nearly 100 counts.

Figure 13. Monitoring the Sensor Signal baSnsSignal



Now open the CapSense wizard, and under the Sensor Settings tab set the finger threshold to 100 counts.You can skip this step if the application does not need precise control of the finger threshold, or if the automatic threshold setting mode is selected. The automatic threshold mode uses an algorithm that dynamically adjusts the finger threshold based on the noise seen in the system. You should choose the manual mode over the automatic threshold mode under the following scenarios:

- If the RAM/ROM footprint is smaller than the number of sensors it must support.
- Designs that require precisely adjusted sensor finger thresholds.
- Designs that need to be operated under noisy conditions.

7. The SmartSense sensor tuning process is completed with the previous step. However, there could be scenarios where the SNR achieved in Step 5 is below the requirement of 5:1, for example, when using a thicker overlay. In such cases, follow these recommendations:
    - Change the sensitivity level from "Low" to "Medium". If the SNR of 5:1 is still not achieved, change the level to "High".
    - Sometimes, the requirement of 5:1 cannot be achieved due to excess noise observed in the system. Under these circumstances, filters should be used. Start with an IIR filter of coefficient 1/2. If this does not reduce the noise, change the IIR coefficient to 1/4.

Eliminate Possible Resource Use Conflicts

Ensure that you do not alter the hardware configurations used by this user module. This includes:

- The GlobalOutEven_0 and Row_0_Output_0 buses that are used internally. Do not connect any sources to these buses.
- Do not change the Row_0_Output_0 bus output LUT function. It should be selected to **A**.
- Do not change the comparator buses LUT functions. The Comparator Bus_0 LUT function should be set to **B**, the Comparator Bus_1 LUT should be set to **~A**.
- The analog module clock source should be set to **VC1**.
- The VC1, VC2, VC3 dividers, and the VC3 source are set internally by the user module. The values entered in the Global Resources are overwritten at runtime.
- When using a shield electrode, set the row LUT function to **A**.

Interrupt Duration Management

Manage your Interrupt Service Routine (ISR) duration carefully when sensor scanning is active. The 8-bit counter is clocked directly from VC_1. The worse case overflow interval for VC_1 is:

$$T_{owf} = VC_1 \frac{256}{F_{IMO}}$$

Equation 2

F_{IMO} – IMO frequency; VC_1= 2

Special attention should be given to asynchronous communication routines, such as I2C, SPI, UART, and sleep timer interrupts. Ensure that they are shorter than the estimated worst case overflow time from Equation 3. Re-enable global interrupts from low priority interrupts (such as sleep timer and I2C) to avoid missing the counter overflow interrupt. In some cases, the source code of problematic ISRs should be optimized (for example, I2CHW and EZI2C have long interrupt handlers). Other interrupts should be re-enabled from the I2CHW interrupt by calling the M8C_EnableGInt macro.

ISSP Pins Possible Conflicts

Permanent connection of a low resistance feedback resistor to the P1[1] pin can cause ISSP programming faults. Use another pin for this. Use the P3[1] pin on packages where it is available. Future versions of the CSD User Module may allow additional pins that can be used to connect the feedback resistor that allow the use of a second I^2C port.

Sample Firmware Source Code

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 SmartSense module
//-------------------------------------------------------------------------------

#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 ;
SmartSense_Start();
SmartSense_InitializeBaselines() ; //scan all sensors first time, init baseline
// Loop Forever
while(1)
{
SmartSense_ScanAllSensors();
          SmartSense_UpdateAllBaselines();
for (bIndex = 0; bIndex < SmartSense_TotalSensorCount; bIndex++)
          {
               SmartSense_UpdateSensorSignal(bIndex);
          }
//detect if any sensor is pressed
          if(SmartSense_bIsAnySensorActive())
          {
               // Add user code here to proceed the sensor touching
          }
// now we are ready to send all status variables to chart program
          // communication here
```

```
//
// OUTPUT SmartSense_waSnsResult[x] <- Raw Counts
// OUTPUT SmartSense_waSnsDiff[x] <- Difference
// OUTPUT SmartSense_waSnsBaseline[x] <- Baseline
// OUTPUT SmartSense_baSnsOnMask[x] <- Sensor On/Off
// OUTPUT SmartSense_baSnsSignal[x] <- normalized sensor signal

}
}
```

The following code scans the sensors and transmits the values to a PC charting tool using the EzI2Cs User Module or TX8SW User Module if there is no excessive I2C traffic during scans. To reduce the caused by I2C traffic noise the line:

```
mov [_bEzI2Cs_Traffic_Flag], 1
```

should be added at beginning of EzI2Cs ISR in the EzI2CsINT.asm Library Source File. The EzI2Cs User Module and TX8SW User Module should be placed and configured in the PD Chip Editor.

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

struct I2C_Regs
{
    WORD wRawCount[SmartSense_TotalSensorCount];
}MyI2C_Regs;

BYTE bEzI2Cs_Traffic_Flag;

void main(void)
{
    BYTE bIndex;
    M8C_EnableGInt ; // Uncomment this line to enable Global Interrupts

    TX8SW_Start(); // Start TX8SW

    EzI2Cs_SetRamBuffer(sizeof(MyI2C_Regs), 1, (BYTE *) &MyI2C_Regs);
    // Initialize I2C and enable I2C interrupts
    EzI2Cs_Start();

    SmartSense_Start();
SmartSense_InitializeBaselines() ; //scan all sensors first time, init baseline

while(1)
    {
      while(EzI2Cs_bBusy_Flag); // Wait for I2C transaction completion
      bEzI2Cs_Traffic_Flag = 0; // Reset I2C transaction flag

SmartSense_ScanAllSensors();
      if (!bEzI2Cs_Traffic_Flag) // Send data if there is no I2C Traffic during scan
{
        SmartSense_UpdateAllBaselines(); //Update all baseline levels;
SmartSense_bIsAnySensorActive();
        // Send RawCounts of All Sensors via I2C
        for (bIndex = 0; bIndex < SmartSense_TotalSensorCount; bIndex++)
```

```
        {
            M8C_DisableGInt;
            MyI2C_Regs.wRawCount[bIndex] = SmartSense_waSnsResult[bIndex];
            M8C_EnableGInt;
        }

        // Send RawCounts, Baselines and Differences of All Sensors via TX8
      TX8SW_PutCRLF();  // Send Header
      TX8SW_Write((char *) (SmartSense_waSnsResult), SmartSense_TotalSensorCount*2);
       TX8SW_Write((char *) (SmartSense_waSnsBaseline),SmartSense_TotalSensorCount*2);
        TX8SW_Write((char *) (SmartSense_waSnsDiff), SmartSense_TotalSensorCount*2);
        TX8SW_PutChar(0x00);  // Send Tail
        TX8SW_PutChar(0xFF);
        TX8SW_PutChar(0xFF);
      }
    }
}
```

Configuration Registers

Table 5. Block CMP, Register: ACE_CR1

Bit	7	6	5	4	3	2	1	0
Value	0	1	0	0	1	1	1	1

Table 6. Block CMP, Register: ACE_CR2

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	Power

Power: 0x01 Turns on power to analog block. 0x00 Turns off power to analog block.

Table 7. Block PWM, Register: Function

Bit	7	6	5	4	3	2	1	0
Value	0	0	1	0	0	0	0	1

Table 8. Block PWM, Register: Input

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	1	1	1	0	0

Data input high, 0x10. Selects clock input from oscillator output (comparator bus routed through globals).

Table 9. Block PWM, Register: Output

Bit	7	6	5	4	3	2	1	0
Value	0	1	0	0	0	1	0	0

Table 10. Block PWM, Register: Period

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	1	1	1	1

PWM divides by 16, enabling counting time and post-count processing interval.

Table 11. Block PWM, Register: Compare

Mode/Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	1	1	0

PWM counts time determined by Compare value. At start of sampling, counting is disabled while data from previous count is read and processed.

Table 12. Block Counter16_LSB, Register: Function

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	1

Table 13. Block Counter16_LSB, Register: Input

Mode/Bit	7	6	5	4	3	2	1	0
Value	1	0	0	0	1	1	0	0

Data = 0x80 (Row_output_0), Clock = 0x0X (SysClk direct, in output reg).

Table 14. Block Counter16_LSB, Register: Output

Mode/Bit	7	6	5	4	3	2	1	0
Value	1	1	0	0	0	0	0	0

Input clock = Use SysClk direct.

Table 15. Block Counter16_LSB, Register: Data2

Bit	7	6	5	4	3	2	1	0
Value	Data Out LSB							

Table 16. Block Counter16_MSB, Register: Function

Bit	7	6	5	4	3	2	1	0
Value	0	0	1	0	0	0	0	1

Table 17. Block Counter16_MSB, Register: Input

Mode/Bit	7	6	5	4	3	2	1	0
Value	1	1	0	0	1	1	0	0

Data input chained from LSB, 0xC0.

Period Method: Input clock = 0x00, SysClk direct.

Table 18. Block Counter16_MSB, Register: Output

Mode/Bit	7	6	5	4	3	2	1	0
Value	1	1	0	0	0	0	0	0

Input clock = SysClk direct.

Table 19. Block Counter16_MSB, Register: Data2

Bit	7	6	5	4	3	2	1	0
Value	Data Out MSB							

Appendix

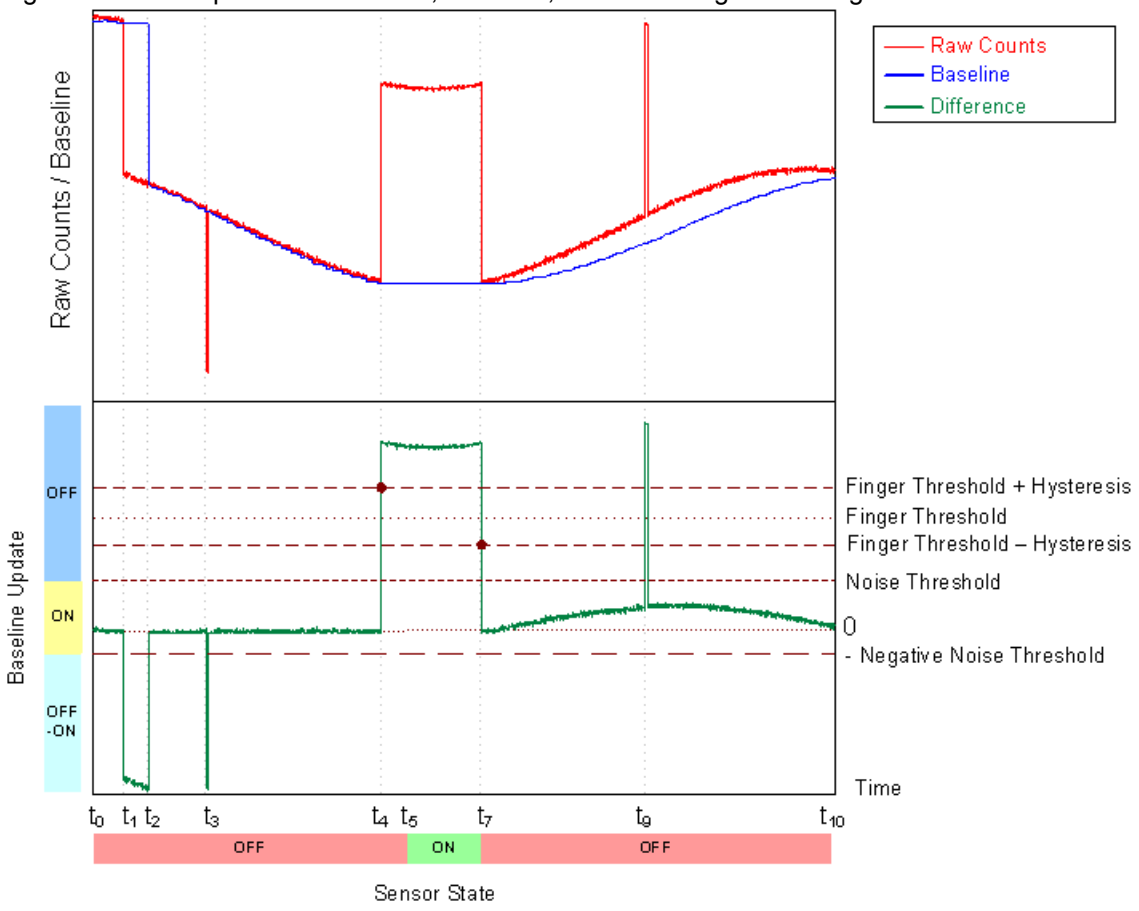
The following sections contain information beyond what is usually included in user module datasheets. The detailed information was developed by Cypress engineers to help you successfully design CapSense applications. Some of this information may be moved into application notes in the future.

Interaction of SmartSense Parameters

The following figures illustrate the baseline update and decision logic operation, and are useful to better understand how to set user module parameters for optimum performance. Figure 13 illustrates system operation when the Sensors Autoreset parameter is set to **Disabled**. Figure 14 illustrates the Sensors Autoreset parameter **Enabled**. The Finger Threshold, Noise Threshold, Hysteresis, and Negative Noise Threshold are shown together with Difference signal (Raw Count - Baseline). Data was collected during some artificial tests that demonstrate system operation at both slow and rapid raw data changes. The slow

changes can be caused by temperature or humidity variations, and the rapid changes can be triggered by a sensor touch, an ESD event, or the influence of a strong RF field.

Figure 14. Example of Raw Counts, Baseline, Difference Signals Change With SensorsAutoreset Set to Disabled



At t_0, the raw counts are close to the baseline level and start to drop slowly because of humidity or temperature changes. Because the raw count change between two successive conversions does not exceed the NegativeNoiseThreshold parameter (by absolute value), the baseline is updated by tracking the Raw Count minimum value, holding the lower value of raw count signal.

At t_1, the raw count drops sharply and the negative difference exceeds the NegativeNoiseThreshold. This situation can happen if the device is powered on when a finger is on the sensor and then the finger is removed after a period of time. At this time, the baseline update mechanism is frozen and an internal timeout counter is activated. The baseline is reset when the difference signal is below the NegativeNoiseThreshold for LowBaselineReset samples. This happens at t_2.

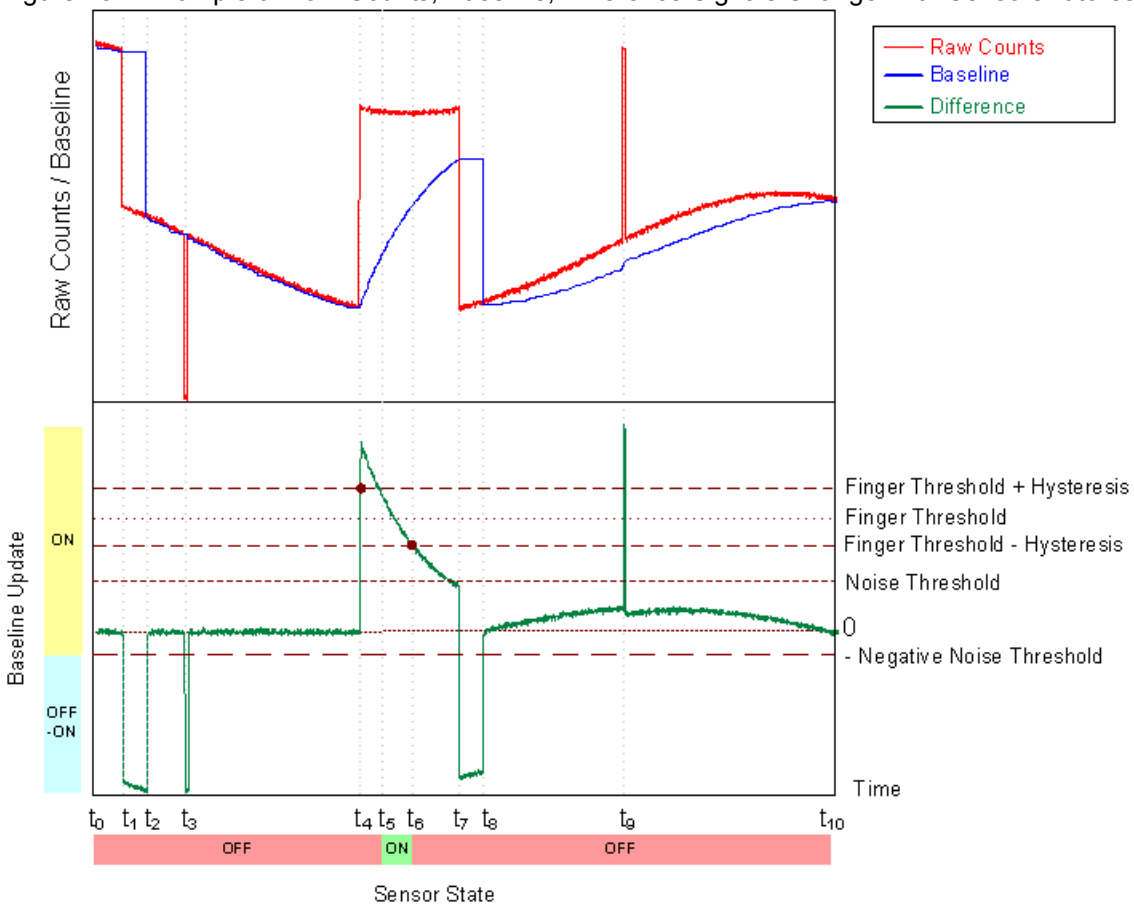
The second large negative difference signal spike happens at t_3. This spike may have been triggered by an ESD event, for example. Because the spike duration in the sample count is less than the LowBaselineReset parameter, the baseline is kept on hold and the spike is filtered. This prevents a false baseline reset and the resulting false touch detection.

The sensor is touched at t_4. When the difference signal exceeds the FingerThreshold + Hysteresis value, the internal debounce counter is activated. If the signal exceeds this value for more than Debounce samples, the sensor state is set to on. This happens at t_5. The sensor state reverts back to the off state immediately when the difference signal drops below the FingerThreshold – Hysteresis level at t_7. The

short positive spike at t_9 is filtered by the debounce counter because the spike duration in sample units does not exceed the Debounce value.

The raw count drifts up slowly between t_7 and t_{10}. The baseline is updated using the bucket algorithm when the difference signal is below the NoiseThreshold (SensorsAutoreset is set to Disabled), the difference signal is proportional to the drift rate. It is possible to control the baseline update speed using the BaselineUpdate Threshold parameter. Lower parameter values give faster baseline update speeds.

Figure 15. Example of Raw Counts, Baseline, Difference Signals Change With SensorsAutoreset Set to Enabled



The system operation in Figure 14 is similar to the operation in the previous case, except for the following differences:

- The touch duration is decreased because of the active baseline update algorithm while the sensor is touched, t_6.
- After the finger is removed, the baseline is reset after LowBaselineReset samples (t_8), which blocks touch detection for a short time. This serves as an additional debounce mechanism.

Version History

Version	Originator	Description
1.00	DHA	Initial version.
1.10	DHA	User module support extended to CY8C21x34B devices.
1.20	DHA	1. Transferred the Diplex Table from "AREA UserModules" to "AREA lit". 2. Set the default "DiplexTable" parameter value to 0x0112. 3. Added the "DiplexUsed" parameter to improve code compression. 4. Updated the SmartSense_bIsAnySensorActive() API function to address an issue with returned sensors mask. 5. Moved the user module API functions SmartSense_Start(), SmartSense_UpdateAllBaselines(), and SmartSense_UpdateSensorBaseline() from the "UserModules" to the "text AREA". 6. Added new parameters: Approaching speed, Proximity IIR Filter, and Proximity IIF Filter Coefficient. 7. Updated the CalibrateSingleSensor() API function. 8. Updated the SmartSense_Calibrate API to fix incorrect variables initialization. 9. Renamed "Switches" to "Buttons" in the wizard GUI. 10. Updated the SmartSense_Start function for the Proximity sensor. 11. Updated the SmartSense_wGetCentroidPos function to improve the functionality of the Slider centroid algorithm. 12. Updated the InternalMainScanSensor() API to modify CPU_CLK, stop and enable PRS, and disable and enable interrupts. 13. Updated the “Features and Overview” and “Sensor Scan Time” sections.

Version	Originator	Description
1.30	DHA	1. Updated area declarations to support Imagecraft optimization. 2. The LoadParameters, SetPrescaler, SetScanMode, and Resume APIs were made identical to the SmartSense_EMC User Module. 3. Deleted SmartSenseTUNE.c. 4. For CY8C21x34 devices, the LoadParameters, SetDefaultFingerThreshold, and UpdateSensorSignal APIs are moved to asm from C. 5. For CY8C20xx6 devices, the LoadParameters API is moved to asm. 6. Updated the wizard to properly indicate the available pins. 7. Added conditional compilation in case 0 sensors is used. 8. Updated the SmartSense_ClearSensors API description in this user module datasheet. 9. Updated the user module wizard help. Added a description of the slider resolution parameter min/max values. 10. Added the M8C_ClearWDT macro to the InternalMainScanSensor function code to resolve watchdog timer coexistence.
1.40	DHA	1. Corrected resolution value calculation in UM wizard to address the error after change in diplexing. 2. Updated code for baSensorSensitivity array initialization. 3. For CY8C21x34 devices, modified ScanSensor API to avoid Watchdog reset. Exported LoadFixedParameters API function from the assembly file. Removed redundant InitializeBaselines API function call from Start API function 4. For CY8C20x66 devices, updated code for baSensorSensitivity array initialization. 5. Removed CS_MISC register use. 6. Datasheet updates: Added Cmod section; added note about wizard behavior when Cmod is not assigned.
1.50	DHA	1. Increased number of proximity sensors to max allowed pins. 2. Explained limitations of dynamic reconfiguration in datasheet. 3. Added CYRF89435 and CY8C200x5 device support.

Version	Originator	Description
1.60	MYKZ	1. Added Resume() function to User Module API. 2. Fixed problem with saving information for sliders. 3. Updated baseline algorithm to check for negative difference counts. 4. Added GetSnsParasiticCapacitance to User Module API. 5. Added build error message when user attempts to build project without first calling the user module wizard. 6. Removed default value for feedback resistor pin and fixed feedback pin handling in User Module wizard. 7. Updated Stop() function to support Rb pins. 8. Added CY7C69000 support. 9. Updated Precharge() function to correct Cmod connection to GND. 10. Limited slider resolution to 100.

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.

Copyright © 2010-2013 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.