CapSense® Successive Approximation Datasheet CSA V 1.50

Copyright © 2006-2015 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	I²C/SPI	CapSense	Timer	flash	RAM	
CY8C20x66, CY8C20x36, CY8C20336AN, CY8C20436AN, CY8C20636AN, CY8C20xx6AS, CY8C20x46, CY8C20x96	–	1	–	1198	129	1

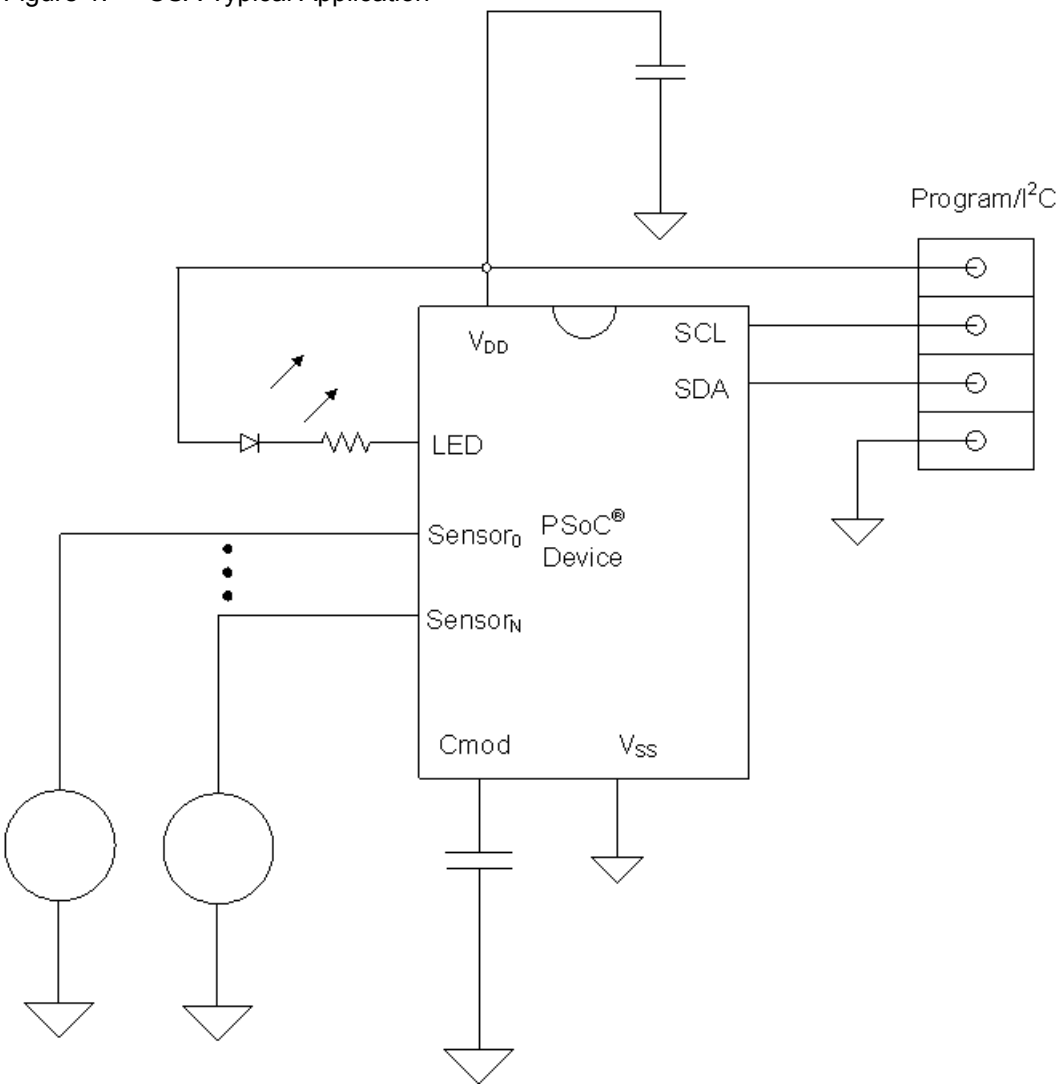
For one or more fully configured, functional example projects that use this user module go to www.cypress.com/psocexampleprojects.

Features and Overview

- Scan 1 to 28 capacitive sensors
- Scan capacitive sliders with 2 to 28 elements
- Slider physical resolution doubling using diplexing
- Slider interpolated resolution up to 1 part in 65535
- Generate touch-pad using multiple slider sensors
- Adjustable sensor sensitivity, detection threshold and sampling rate
- Guided sensor/pin assignments using the CSA Wizard
- Integrated baseline update algorithm for handling temperature changes
- Compensate for environmental and physical sensor variations

The CapSense® Successive Approximation (CSA) User Module implements an array of capacitive touch sensors using switched capacitor circuitry, an analog multiplexer, digital counting functions, and high level software routines to compensate for environmental and physical sensor variations. The sensor array can consist of combinations of independent sensors, sliding sensors, and touchpads implemented as a pair of orthogonal sliding sensors. High level software routines accommodate slider diplexing. Slider diplexing allows a single pin to measure two electrical sensors in two different physical locations. Diplexing provides resolution enhancement of the slider without the cost of an additional I/O.

Figure 1. CSA Typical Application



Quick Start

1. Select and place user modules requiring dedicated pins (such as I^2C or LCD) if used, assign ports and pins.
2. Select and place the CSA User Module.
3. Right click the CSA User Module to access the CSA Wizard.
4. Set button sensor count, slider configuration, and pin assignments and associations.
5. Set pins and global user module parameters.
6. Generate the application and switch to Application Editor.
7. Adapt sample code to implement buttons, sliders, or touch pad.

Functional Description

The capacitive sensor consists of physical, electrical, and software components.

Each sensor measured by the CSA User Module is a capacitor with one side grounded and the other side connected to a PSoC pin. The presence of the conductive object increases the capacitance of the sensor to ground. This capacitance change controls sensor activation.

DC and AC Electrical Characteristics

Unless specified otherwise, Vdd = 2.7V to 5.0V, and C_{mod} = X7R-type capacitor, ±20% tolerance.

Table 1. Electrical Specifications for the CSA User Module

Symbol	Description	Conditions	Min	Typ	Max	Unit
C_P	Parasitic Capacitance[a]	see note on C_P ranges[b]	5		50	pF
C_F	Finger Capacitance[c]		0.1			pF
N	Output Counter Resolution		16		16	bit
S_{FINGER}	Finger Sensitivity[d]	C_P = 5 to 15 pF IDAC Setting = 10 Settling Time = 200 C_{mod} = 2200 pF CSA Clock = 6 MHz IMO = 12 MHz CPU = 3 MHz	500			counts/pF
		C_P = 9 to 27 pF IDAC Setting = 10 Settling Time = 255 C_{mod} = 3300 pF CSA Clock = 3 MHz, IMO = 12 MHz CPU = 3 MHz	500			counts/pF
		C_P=16 to 50 pF IDAC Setting = 15 Settling Time = 255 C_{mod} = 5600 pF CSA Clock = 3 MHz IMO=12 MHz CPU=3 MHz	500			counts/pF

Symbol	Description	Conditions	Min	Typ	Max	Unit
t_C	Conversion Time, Single Sensor.	C_P = 5 to 15 pF IDAC Setting = 10 Settling Time = 200 C_{mod} = 2200 pF CSA Clock = 6 MHz IMO = 12 MHz CPU = 3 MHz			1600	µs/sensor
		C_P = 9 to 27 pF IDAC Setting = 10 Settling Time = 255 C_{mod} = 3300 pF CSA Clock = 3 MHz IMO = 12 MHz CPU = 3 MHz			2500	µs/sensor
		C_P = 16 to 50 pF IDAC Setting = 15 Settling Time = 255 C_{mod} = 5600 pF CSA Clock = 3 MHz IMO = 12 MHz CPU = 3 MHz			2500	µs/sensor
I_{DDCS}	Average Supply Current	Vdd = 3.3V C_P = 5 to 50 pF 4 button scan 100 ms report rate		35	50	µA
R_S	Series Resistor for RF immunity[e]	High conductivity traces (copper or silver ink) longer than 25 mm	300		560	ohms

a. C_P includes package-related capacitance of 2-3 pF.

b. Finger Sensitivity and Conversion Time specified over one of three C_P ranges: (5-15 pF), (9-27 pF), (16-50 pF)

c. Finger Capacitance is the increase in C_P caused by a finger touch on the sensor.

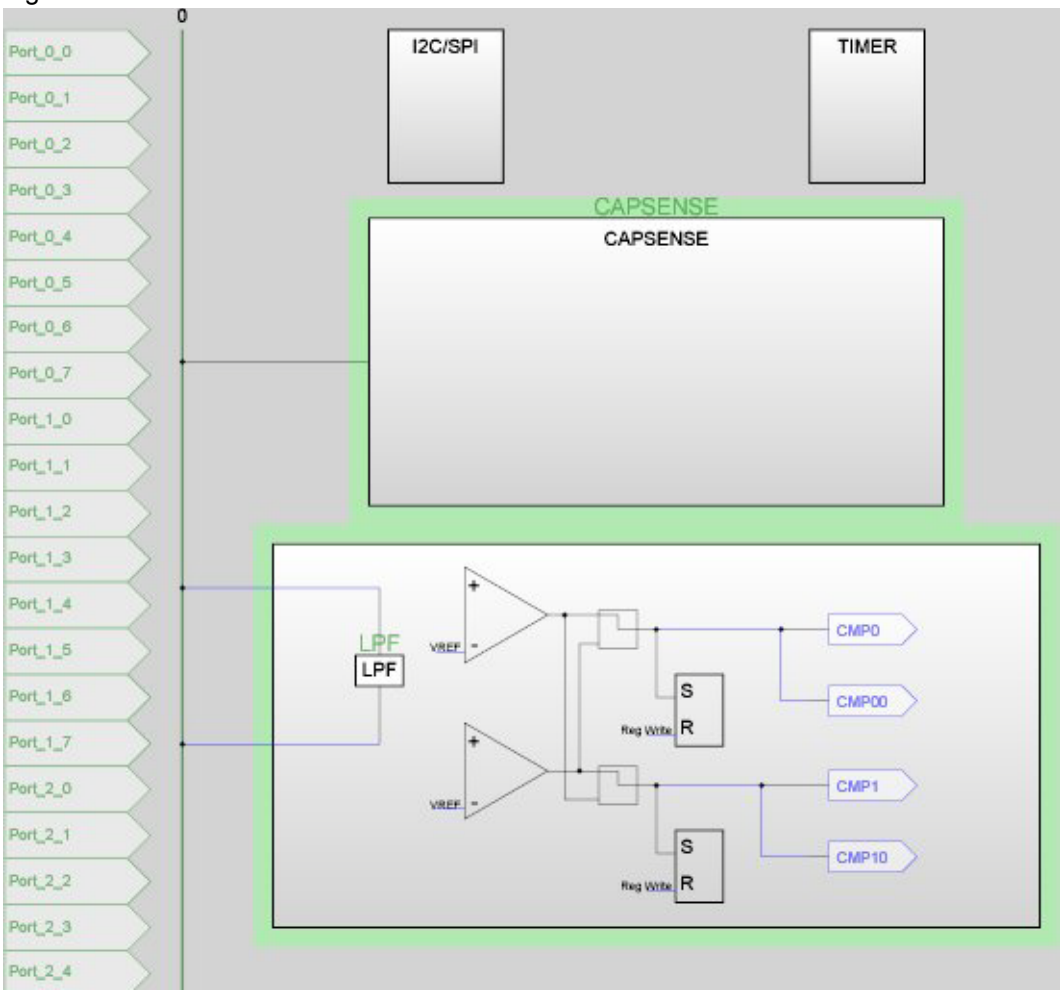
d. IDAC Setting is selected so that a finger touch produces a signal of at least 50 counts.

e. R_S not required for low conductivity ITO thin film. Resistors are placed within 10 mm of PSoC pin.

Placement

The blocks for the user module are automatically placed when the UM is instantiated, alternate placements are not available. User modules that consume specific pin resources, including the LCD and EZI2Cs, must be placed before establishing port pin connections for the CSA User Module. These selections are reflected in the wizard when it is opened.

Figure 2. Placement of the CSA User Module



Do not use P1[0] and P1[1] when placing capacitive sensor connections. These pins are used for programming the part and may have excess routing capacitance that degrades sensor performance.

Application Information

This section discusses step-by-step how to design a capacitive sensing system using the CSA User Module. It assumes the PSoC device is powered by Vdd = 2.7V to 5.0V, and that the C_{mod} is an X7R-type capacitor.

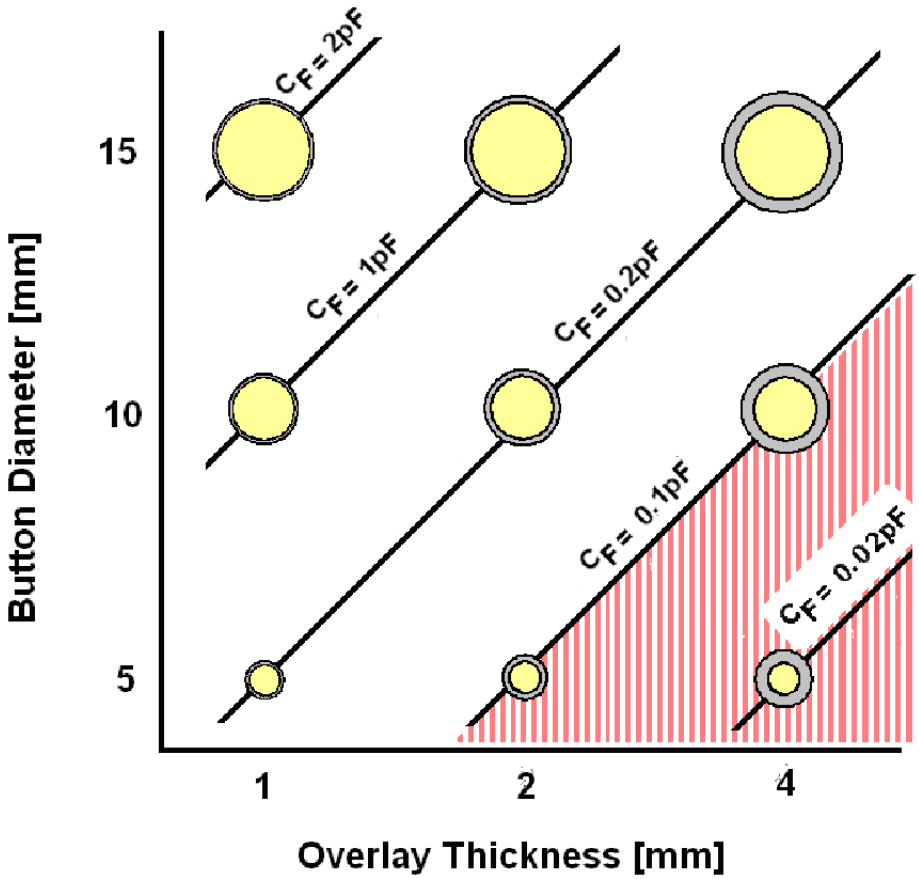
1. **Button Size**: Use the plot in Figure 3 to select a button size that produces the correct level of finger capacitance for a given overlay thickness. The minimum finger capacitance, C_F, is 0.1 pF, but 0.2 pF is recommended for the additional design margin it offers. The clearance between the button and ground fill is equal to the overlay thickness.

   Does the board design combined with the overlay meet the minimum requirement for Finger Capacitance

   If the answer is no, use a thinner overlay, or increase the size of the sensor.

   If the answer is yes, then go to Step 2.

Figure 3. Button Diameter vs. Overlay Thickness



Example: The plastic overlay is 2 mm thick. The design goal is C_F = 0.2 pF. Figure 3 shows that the button diameter needs to be 10 mm, with 2 mm of clearance between sensor pad and the ground fill.

2. **CP Within 5 to 50 pF**: Check the range of C_P that will be monitored by the CSA User Module.
   Estimate capacitive loading on each PSoC pin using the rule of thumb that trace capacitance contributes about 2 pF/inch on a 63mil-thick 2-layer board with 8mil traces.
   Does every sensor fit in the C_P range of 5 pF to 50 pF

   If the answer is no, move the sensor pad closer to the PSoC device, or consider using another CapSense method, such as CSD. Any traces longer than 24" will exceed the 50pF limit.

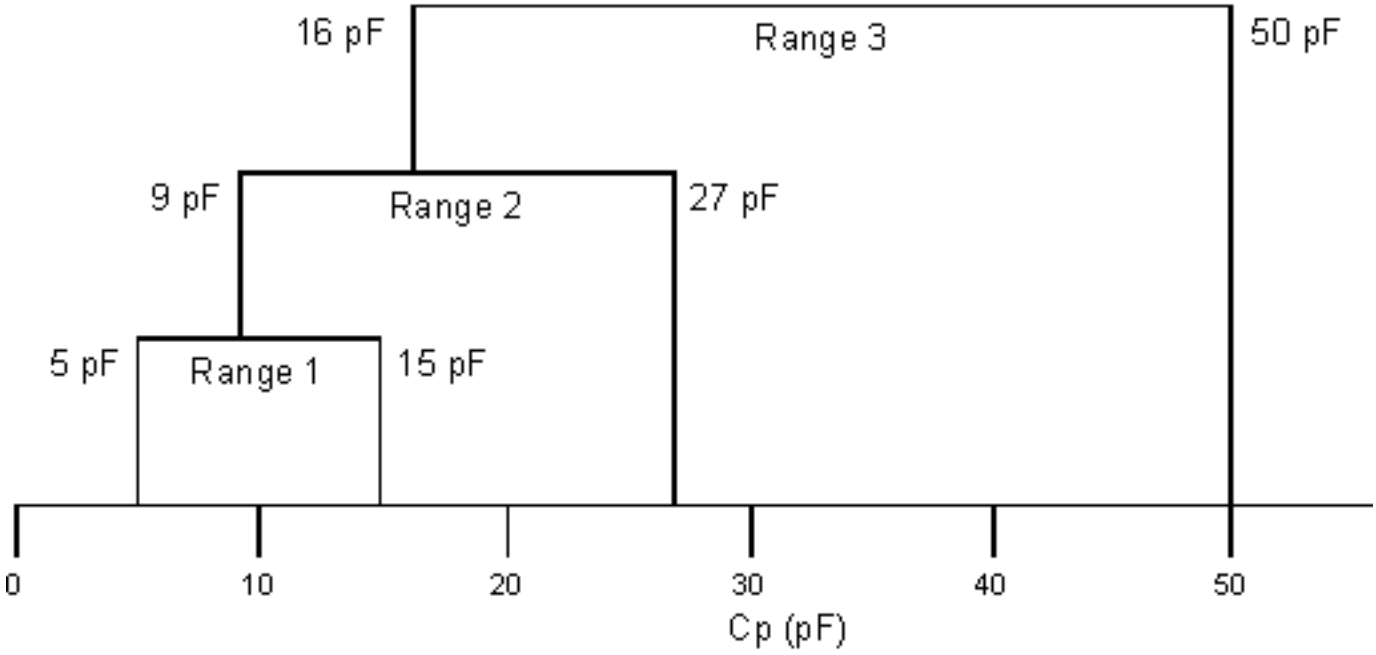
   If the answer is Yes, then go to Step 3.

   Example: The system has 12 touch sensors. The distance from the sensor pad to the PSoC pin is less than 4" for all sensors. The C_P for these sensors range from 9 pF to 18 pF for the bare PCB. Taking into account the packaging effect of the PSoC device (add 3 pF), the total loading on the PSoC pins range from 12 pF to 21 pF, which easily fits in the C_P limits of 5-50 pF.

3. **CP Range**: Finger Sensitivity and Conversion Time are specified over one of the ranges of C_P shown in Figure 4. The ranges are 5-15 pF, 9-27 pF, and 16-50 pF.

   Find the range which best fits the C_P values found in Step 1, and apply the parameters shown in the 'Conditions' column of the DC and AC Electrical Charecteristics section. Proceed to step 4.

Figure 4. The Ranges for CSA are (5-15 pF), (9-27 pF), and 16-50 pF).



Example: The total loading on the PSoC pins range from 12 pF to 21 pF. The C_P range that best fits these values is Range 2, which goes between 9 pF and 27 pF.

4. **Difference Counts and Thresholds**: Estimate difference counts, and set the Finger and Noise Threshold parameters. Difference counts are the increase in counts caused by a finger touch on the sensor. Difference counts are found using the finger sensitivity, S_{FINGER}, and fin-

ger capacitance, C_F, in Equation 1. Finger threshold and noise threshold are found using Equation 2 and Equation 3 (see CY8C20xx6A/H CapSense Design Guide).

Equation 1

$$DifferenceCounts = S_{Finger} \times C_F$$

Equation 2

$$FingerThreshold = 0.75 \times DifferenceCounts$$

Equation 3

$$NoiseThreshold = 0.5 \times DifferenceCounts$$

Compute these three values and go to Step 5.

Example: The system has the following parameters.

C_P = 10 to 22 pF, C_F = 0.2 pF

IDACSetting = 7

SettlingTime = 245

C_{mod} = 2700 pF, X7R (+/-20%)

CSA Clock = 6 MHz

IMO = 12 MHz

CPU = 6 MHz

Solve for difference counts, finger threshold, and noise threshold.

Difference counts = 500 counts/pF * 0.2 pF = 100 counts

Finger threshold = .75 * 100 = 75 counts

Noise Threshold = .5 * 100 = 50 counts

5. **Scan Time**: Estimate the scan time, t_{SCAN}, for scanning all the sensors using the conversion time, t_C, in Equation 4. Conversion time is 900 µs per sensor.

Equation 4

$$t_{SCAN} = t_C \times (numberofsensors)$$

Compute this value and go to Step 6.

Example: There are 12 sensors in the system defined in Step 4.

tSCAN = 900 ms/sensor * 12 sensors = 10.8 ms

6. **Average Supply Current**: Estimate the average supply current, I_{DDCS}, using the active current, I_{active}, the sleep current, I_{sleep}, and the report rate in Equation 5.

Equation 5

$$I_{DDCS} = [t_{SCAN} \times I_{active} + (ReportRate - t_{SCAN}) \times I_{sleep}] / (ReportRate)$$

Example: Continuing from Step 5, add the following parameters.

ReportRate = 100 ms

I_active = 1.13 mA

I_sleep = 2.6 mA

Solve for the average supply current, IDDSC.

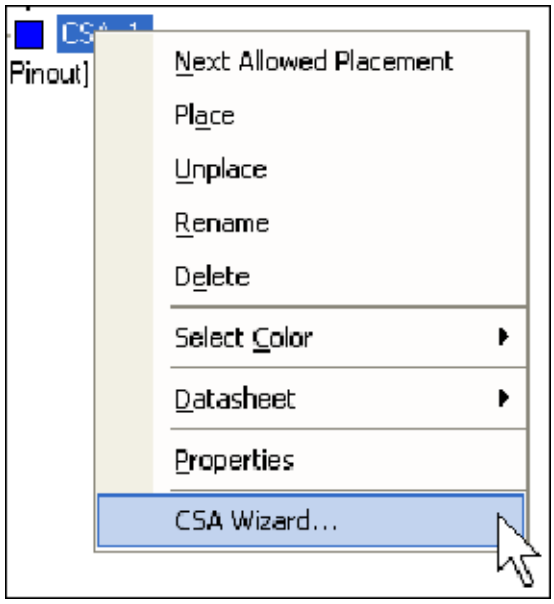
I_{DDCS} = [10.8 ms * 1.13 mA + 89.2 ms * 2.6 mA] / 100 ms = 124 mA

7. **Series Resistors**: Determine the need for series resistors for RF immunity.
   Example: PCB traces are copper and are over 25 mm long. Under these conditions, 560Ω series resistors are recommended for all CapSense inputs.

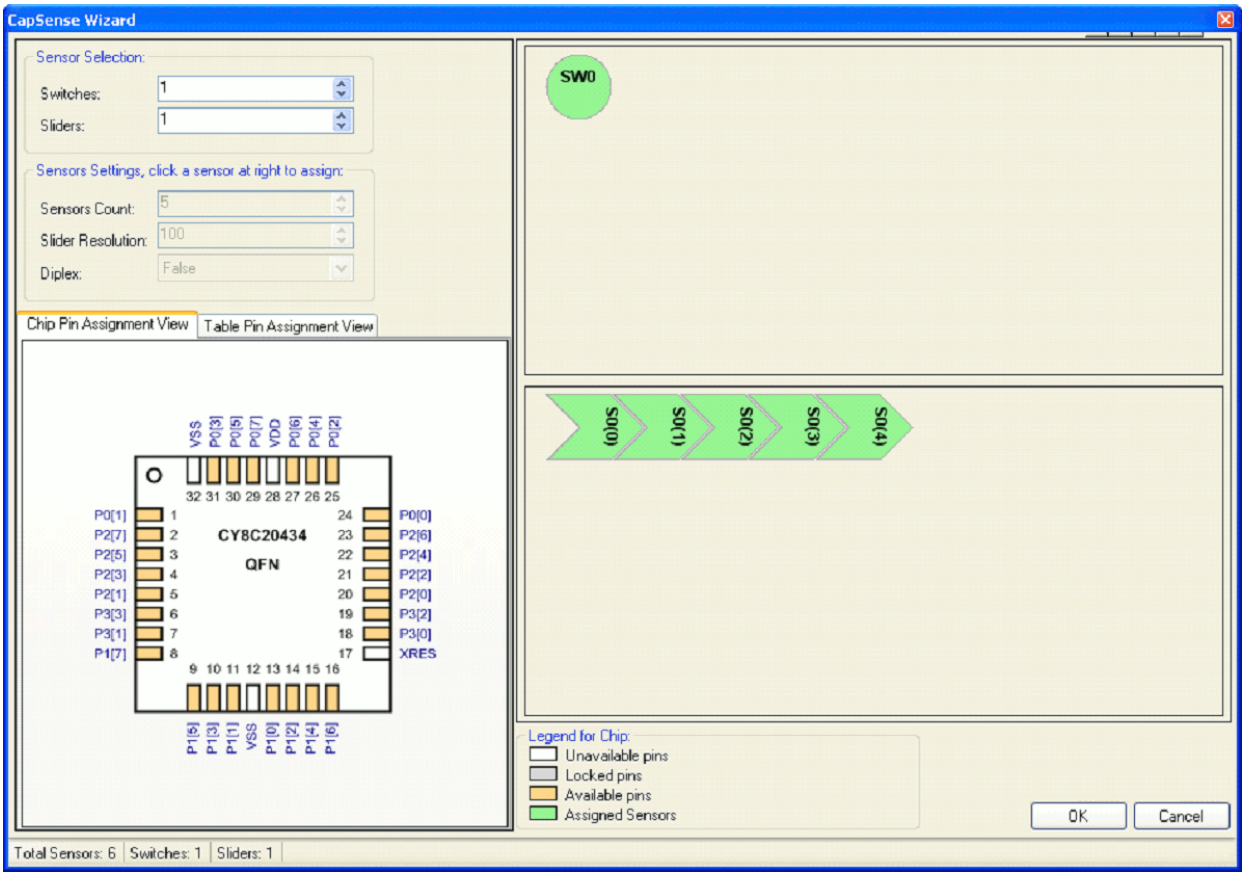
Wizard

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

1. To access the Wizard, right click any block of the CSA in the Device Editor Interconnect View, then select the CSA 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

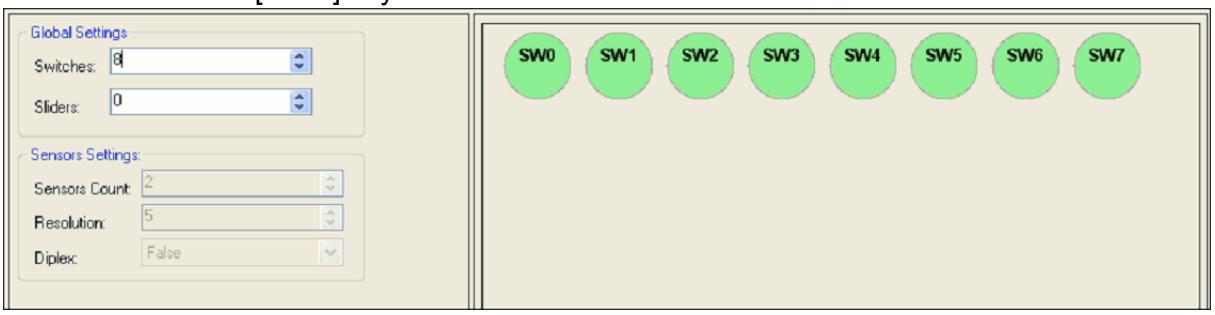
White – The pin cannot be used as a CapSense input.

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, in the Pinout view expand the pin, from the **Select** menu, select **Default**. The pin is now available for assignment in the wizard.

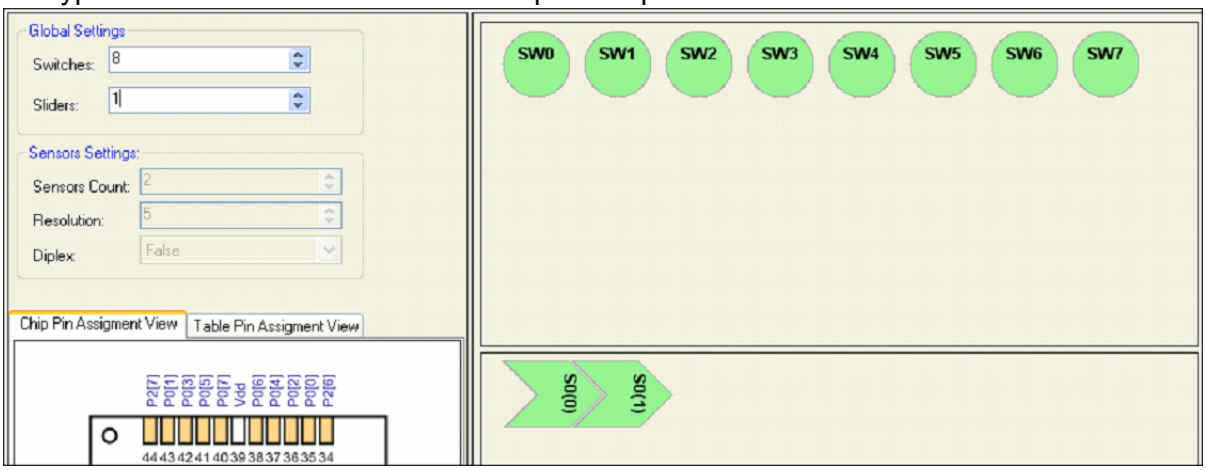
Orange – The pin is available for assignment.

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. Press the [Enter] key to enter the new value for the number of sensors.



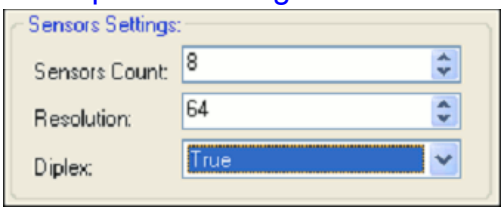
4. Type the number of sliders. X-Y touchpads require two sliders.



5. Click the sliders to enable the Sensor Settings. Type the number of sensor elements in each slider. The practical minimum number of sensors in a slider sensor is five, the maximum is limited by pin count. After entering the data, press the [Enter] key to enter the new value.
6. Type the output resolution. The minimum value is five. The maximum value is (# of pins used for sensors – 1) x 2^{16} – 1 or (2 x pins used for sensors – 1) x 2^{16} -1 for diplexed sliders.

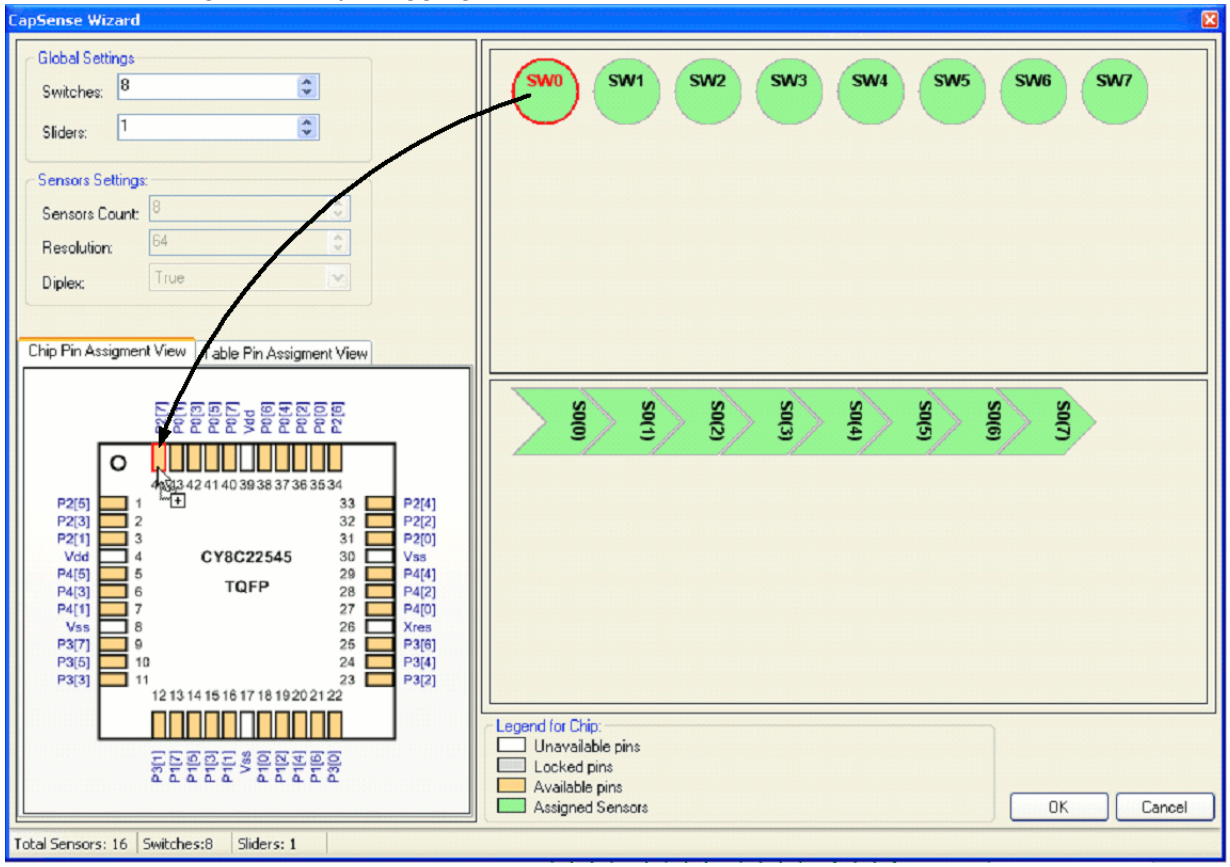


7. Select Diplex, if desired. This maps the number of pins selected for sensors to twice as many sensor locations on the board. Only the first half of the diplex sensors is shown. See CY8C20xx6A/H CapSense Design Guide to find Diplexing tables for pin connections.

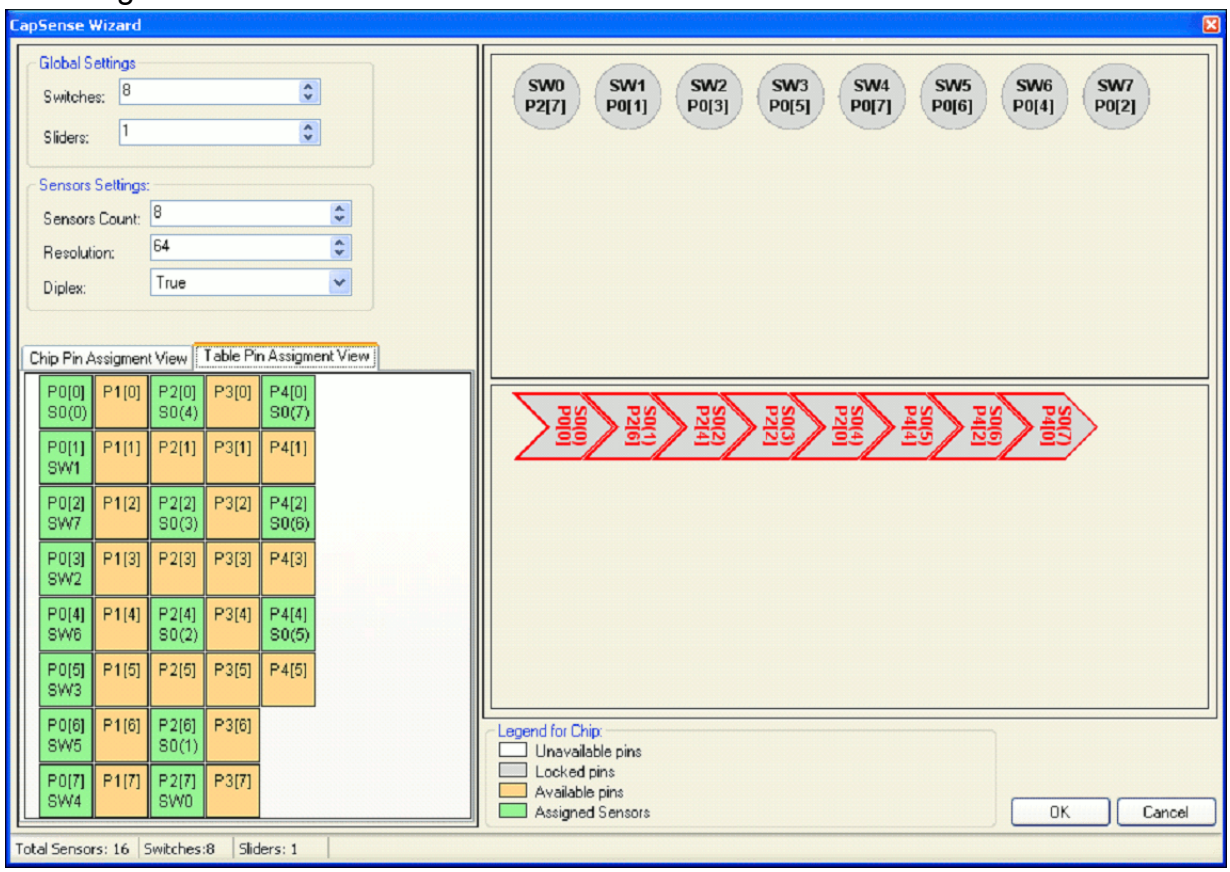


8. Assign switches or sensors to pins by dragging the switch or sensor onto the pin in the Pin Assignment View. You can choose to drag switches or sensors onto pins in the Chip Pin Assignment View or the

Table Pin Assignment View. The port pin is green after selection and is no longer available. Change sensor assignments by dragging the sensor off of the port pin.



9. Repeat for the remainder of independent sensors. Mapping of individual slider sensors onto physical port pins is the same as for individual sensors. 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 adapt a sample project now, if you wish.

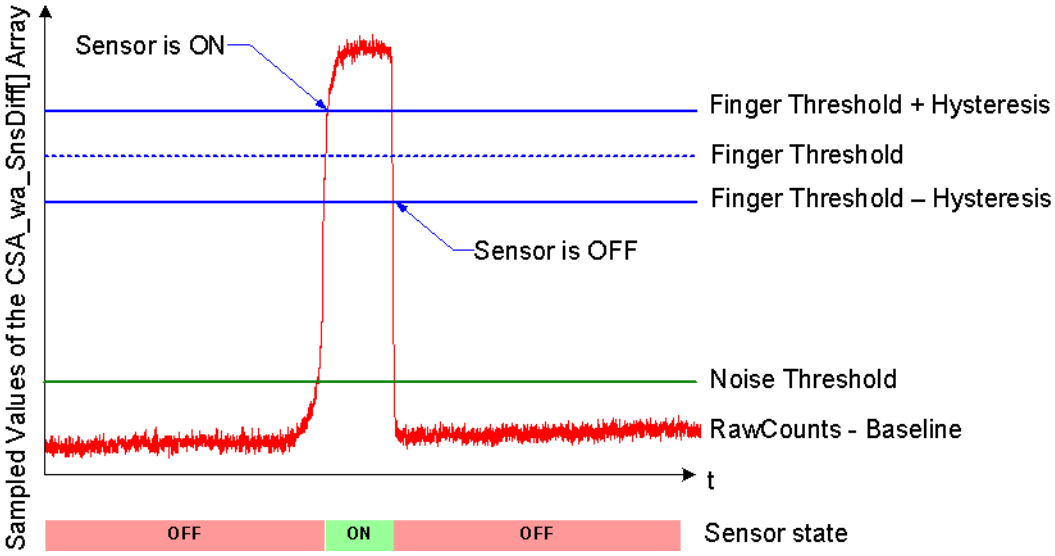
If you want change pin assignment, place your cursor on the assigned pin, click the pin, and drag and drop it outside the switches box. The pin is unassigned and you can then reassign it.

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 CSD_Table.asm

Parameters and Resources

The optimal settings for the user module parameters are determined by the board layout and button dimensions. Refer to the Application Information section for the proper parameter values.

Figure 5. Terminology for the CSA Parameters Using a Difference Counts Waveform



Finger Threshold

This threshold is used to determine the state of each button sensor using the difference counts wave form. If the difference count stored in the wa_SnsDiff[] array is at or above the Finger Threshold, the sensor is active.

Possible values range from 3 to 255.

Noise Threshold

For individual sensors, this parameter sets the count value above which the baseline is not updated.

For slider sensors, it sets the count value below which the results are not counted in the calculation of the centroid.

Possible values are 3 to 255.

Baseline Update Threshold

When the new raw count value is above the current baseline and the difference is below the noise threshold, the difference between the current baseline and the raw count is accumulated into a "bucket". When the bucket fills, the baseline increments and the bucket is emptied. This parameter sets the threshold that the bucket must reach for the baseline to increment.

Possible values are 0 to 255.

SettlingTime

The SettlingTime parameter controls the software delay that allows the voltage on the C_{mod} capacitance to stabilize. The loop has 9 CPU cycles per iteration. Select a SettlingTime that is at least 5 × R × C, where R = 1 ÷ (Clock × C_P) and C = C_{mod}.

Possible values are 2 to 255.

IDACSetting

This parameter controls the slope of the ADC ramp voltage on C_{mod}. A low setting produces a slow ramp and large count values. A high setting produces a fast ramp and small count values.

Possible values are 4 to 255.

ExternalCap

The ExternalCap parameter chooses the PSoC pin to which the C_{mod} capacitor connects. The recommended range of C_{mod} is 1200 pF to 5600 pF. The value of C_{mod} impacts finger sensitivity and conversion time. See the DC and AC Electrical Characteristics section for details.

Possible values are None, P0[1], and P0[3].

Hysteresis

The Hysteresis parameter adds or subtracts from the finger threshold depending on whether the sensor is currently active or inactive. If the sensor is off, the difference count must overcome the finger threshold plus hysteresis. If the sensor is on, the difference count must go below the finger threshold minus hysteresis. It is used to add debouncing and "stickiness" to the finger detection algorithm.

Possible values are 0 to 255. However, the setting must be lower than the FingerThreshold parameter setting.

Debounce

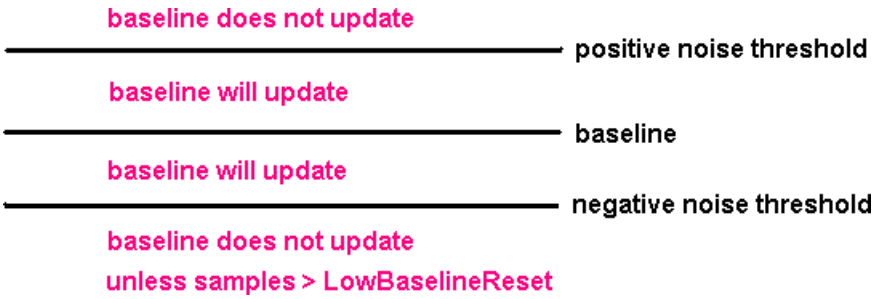
The Debounce parameter adds a debounce counter to the sensor active transition. For the sensor to transition from inactive to active the difference count value must stay above the finger threshold plus hysteresis for the number of samples specified.

Possible values are 1 to 255. A setting of 1 provides no debouncing.

NegativeNoiseThreshold

The NegativeNoiseThreshold parameter adds a negative difference count threshold. If the current raw count is below the baseline by more counts than the NegativeNoiseThreshold parameter and the difference between them is greater than this threshold, the baseline is not updated. However, if the current raw count stays in the low state (difference greater than threshold) for the number of samples specified by the LowBaselineReset parameter, the baseline is reset.

Possible values are 0 to 255.



LowBaselineReset

The LowBaselineReset parameter works together with the NegativeNoiseThreshold parameter. If the sample count values are below the baseline minus the NegativeNoiseThreshold for the specified number of samples, the baseline is set to the new raw count value. It counts the number of abnormally low samples required to reset the baseline. It is used to correct for the finger-on-at-startup condition.

Possible values are 0 to 255.

Sensors Autoreset

This parameter determines whether the baseline is updated at all times or only when the signal difference is below the Noise Threshold. When set to **Enabled** the baseline is updated constantly. This setting limits the maximum time duration of the sensor (typical values are 5 – 10s), but it prevents the sensors from permanently turning on when the raw count suddenly rises without anything touching the sensor. This sudden rise can be caused by a large power supply voltage fluctuation, a high energy RF noise source, or a very quick temperature change.

When the parameter is set to **Disabled** the baseline is updated only when raw count and baseline difference is below the Noise Threshold parameter.

Possible values are Enabled and Disabled.

High Level API

The High Level API parameter can be Enabled or Disabled. Setting the parameter to Disable saves RAM and ROM by excluding the high level APIs provided by the user module. The parameter should be left in the Enabled state unless you need extra RAM and ROM and do not need the high level API.

Possible values are Enabled and Disabled.

Clock

The Clock parameter can be used to increase the amount of effective resistance of the sensor. If the sensor area is large, the effective resistance may be too high for the auto calibration of the switched capacitor circuit. Touchpad rows/columns or large proximity sensors may encounter decreased sensitivity. In this case, the settling voltage is too far beneath the comparator threshold. Setting a larger divider of the IMO increases the effective resistance, compensating for the high capacitance.

Possible values are IMO, IMO/2, IMO/4, and IMO/8.

Application Programming Interface

The Application Programming Interface (API) routines are provided as part of the user module to allow your code to interact with the user module without dependence on its implementation details. This section specifies each function of the interface together with related constants provided by the include files.

Note

In this, as in all user module APIs, the values of the A and X register are 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 in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers need to make certain that 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, start, and stop the CSA User Module. In all cases the instance name of the module replaces the CSA prefix shown in the following entry points. Failure to use the correct instance name is a common cause of syntax errors.

Software Control Parameters

Control parameters passed to the APIs include:

bSnsGroup

Reference to a specific group of sensors used as a slider. Used by CSA_bGetCentroidPos to select which group of sensors to update.

Buttons are contained in group 0. Sliders are contained in group 1 and higher.

bSensor

The sensor number is used by CSA_wGetPortPin to determine port and bit mask (bPort and bMask) for the selected active sensor. CSA_wGetPortPin returns bMask and bPort. These two are used by CSA_EnableSensor and CSA_DisableSensor to determine specific sensor selection. They are also used by CSA_wReadSensor to set which sensor's counts are returned.

CSA Data Arrays

API functions use several global arrays. You should not alter these arrays manually. You can inspect these values for debugging purposes.

CSA_waSnsResult

This array holds the 16-bit raw count values for each sensor.

CSA_waSnsBaseline

This array holds the 16-bit baseline values for each sensor.

CSA_waSnsDiff

This array holds the 16-bit difference count values for each sensor (raw count - baseline).

CSA_baSnsOnMask

This 8-bit array holds the sensor on or off data (for buttons or sliders). Each element in the array holds the sensor state for up to 8 sensors. CSA_baSnsOnMask[0] contains the masked bits for sensors zero through seven (sensor zero is bit 0, sensor one is bit 1). CSA_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 the bit is 1 if the sensor is on and 0 if the sensor is off.

CSA_baDACCodeScan

This array holds the 8-bit IDACSetting parameter values that set the slope of the ADC linear ramp voltage on Cmod. This parameter can be configured individually for each sensor as long as the array is loaded immediately after the call to CSA_Start (before initializing the baselines or scanning any sensors).

CSA_baDACCodeBaseline

This array holds the 8-bit calibration setting for each sensor that is determined automatically in CSA_Start. The values in this array give a relative measure of the parasitic capacitance loading each CapSense input.

Data Tables

Based upon your entries for sensor count, pin assignment, diplexing, and resolution, the wizard generates a set of data tables. The sensor table is located in CSA_table.asm. The group and diplex tables are located in CSAHL.asm.

CSA_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). The table includes all independent sensors, then each slider sensor in order. An example for a table with six sensors is:

```
CSA_Sensor_Table:
_CSA_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
```

CSA_Group_Table

The group table defines each of the groups of sensors or slider sensors. The first entry in each line of the table is the starting sensor number for the group. The second entry is the number of sensors in the group. The third entry is 0 if not diplexed. The fourth, fifth, and sixth entries are combined to form the fixed point multiplier value for the centroid calculation. An example with seven sensors is shown below:

```
CSA_Group_Table:
CSA_Group_Table:
// Group Table
//        Origin Count  Diplex  SliceMultiplier
      db    0,    0x7,    0x0,    0x00     // Buttons
```

In projects with independent sensors and slider sensors, the set of independent sensors and each slider sensor has a separate entry in the group table, as shown below:

```
CSA_Group_Table:
CSA_Group_Table:
; Group Table
;         Origin Count  Diplex   DivBtwSns(wholeMSB, wholeLSB, fractByte)
      db    0,    0x7,    0x0,    0x00,    0x00,    0x00 ; Buttons
      db    0x6,  0xA,    0x4,    0x0,     0x7,     0xE5 ; Slider 1
```

CSA_Diplex_Table

The diplex table defines the mapping of the full range of sensors for each slider sensor. 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 a ten sensor slider is shown below.

```
DiplexTable_0:
; This group is not a diplexed slider

DiplexTable_1:
    db    0,1,2,3,4,5,6,7,8.9,0,3,6,9,1,4,7,2,5,8   // 10 switch slider
```

```
CSA_Diplex_Table:
_CSA_Diplex_Table:
    db >DiplexTable_0, <DiplexTable_0
    db >DiplexTable_1, <DiplexTable_1
```

Basic APIs

Basic APIs are used to start and stop the user module.

CSA_Start()

Description:

Calibrates the CSA User Module for each sensor. Disconnects all sensor pins from the Analog Mux Bus. All sensor pins are shunted to ground. Connects the C_{mod} capacitor to the system.

C Prototype:

```
void  CSA_Start()
```

Assembly:

```
lcall  CSA_Start
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSA_Stop()

Description:

Disables the CapSense block. Calls CSA_ClearSensors to disconnect all sensor pins from the Analog Mux Bus and shunt them to ground.

C Prototype:

```
void  CSA_Stop()
```

Assembly:

```
lcall  CSA_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

**

Low Level APIs

Low level APIs are used to acquire sensor data.

CSA_ScanSensor

Description:

Scans a single sensor to determine the raw count value representing its capacitance. This routine assumes that the CSA_Start function was called before its execution. The routine is a blocking call that waits for the CapSense block interrupt CSA_ISR to complete. It then transfers the data from the 16-bit counter to the CSA_waSnsResult array.

C Prototype:

```
void  CSA_ScanSensor(BYTE bSensor)
```

Assembly:

```
mov  A,  bSensor
lcall CSA_ScanSensor
```

Parameters:

bSensor: Range is 0 to n-1 where n is the total of the number of sensors set in the CSA Wizard plus the number of sensors included in slider sensors.

Return Value:

None

Side Effects:

**

CSA_ScanAllSensors

Description:

Scans all of the configured sensors by calling CSA_ScanSensor for each sensor index.

C Prototype:

```
void  CSA_ScanAllSensors()
```

Assembly:

```
lcall  CSA_ScanAllSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSA_ClearSensors

Description:

CSA_DisableSensor for each of the sensors. The sensors pins are disconnected from the Analog Mux Bus and shunted to ground.

C Prototype:

```
void  CSA_ClearSensors()
```

Assembly:

```
lcall  CSA_ClearSensors
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSA_wReadSensor

Description:

Returns the raw count value for a given sensor.

C Prototype:

```
WORD  CSA_wReadSensor(BYTE bSensor)
```

Assembly:

```
mov A, bSensor
lcall CSA_wReadSensor
```

Parameters:

bSensor: Range is 0 to n-1 where n is the total of the number of sensors set in the CSA Wizard plus the number of sensors included in slider sensors.

Return Value:

The raw count value of the sensor, LSB in A and MSB in X.

Side Effects:

**

CSA_wGetPortPin

Description:

Returns the port number and pin mask for a given sensor. The passed parameter indexes and selects the data from the CSA_Sensor_Table.

C Prototype:

```
WORD  CSA_wGetPortPin(BYTE bSensor)
```

Assembly:

```
mov  A,  bSensor
lcall     CSA_wGetPortPin
```

Parameters:

bSensor: Range is 0 to n-1 where n is the total of the number of sensors set in the CSA Wizard plus the number of sensors included in slider sensors.

Return Value:

bPort and bMask: The port number and bit mask used to determine specific sensor selection.

Side Effects:

**

CSA_EnableSensor

Description:

Configures the selected sensor for scanning during the next measurement cycle. The port and pin of the sensor are selected using the CSA_wGetPortPin routine, with the port number and sensor bit mask 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  CSA_EnableSensor(BYTE bMask, BYTE bPort)
```

Assembly:

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

Parameters:

bPort and bMask: The port number and bit mask used to determine specific sensor selection.

Return Value:

None

Side Effects:

**

CSA_DisableSensor

Description:

Disconnects a sensor so it is no longer an input. Typically, this call is used in combination with CSA_wGetPortPin. The connection from the port pin to the AnalogMuxBus is turned off. The drive mode is changed to Strong (01) and the data register bit is set to zero. This grounds the sensor.

C Prototype:

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

Assembly:

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

Parameters:

bPort and bMask: The port number and bit mask used to determine specific sensor selection.

Return Value:

None

Side Effects:

**

High Level APIs

High level APIs are used to process sensor data acquired by the low level APIs.

CSA_UpdateSensorBaseline

Description:

Update the sensor baseline for one sensor. 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. Each time CSA_UpdateSensorBaseline is called, a difference count is calculated by subtracting the previous baseline from the raw count value. This difference is stored in the waSnsDiff array.
2. Each time CSA_UpdateSensorBaseline is called, the difference count is compared to the noise threshold. If the difference is below the noise threshold, half of the difference is added/accumulated in a virtual bucket. If the difference is above the noise threshold, the baseline is not updated.
3. Once the accumulated difference counts in the virtual bucket have reached the BaselineUpdateThreshold, the baseline is incremented 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 set to 0.

C Prototype:

```
void CSA_UpdateSensorBaseline(BYTE bSensor)
```

Assembly:

```
mov   A,   bSensor
lcall  CSA_UpdateSensorBaseline
```

Parameter:

bSensor: Range is 0 to n-1 where n is the total of the number of sensors set in the CSA Wizard plus the number of sensors included in slider sensors.

Return Value:

```
None
```

Side Effects:

**

CSA_UpdateAllBaselines

Description:

Uses the CSA_bUpdateSensorBaseline routine to update the baselines for all sensors

C Prototype:

```
void  CSA_UpdateAllBaselines()
```

Assembly:

```
lcall CSA_UpdateAllBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSA_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 the state of the sensor. 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 CSA_baSnsOnMask array.

C Prototype:

```
BYTE  CSA_bIsSensorActive(BYTE bSensor)
```

Assembly:

```
mov  A,  bSensor
lcall CSA_bIsSensorActive
```

Parameters:

bSensor: Range is 0 to n-1 where n is the total of the number of sensors set in the CSA Wizard plus the number of sensors included in slider sensors.

Return Value:

Return value of 1 if active, 0 if not active

Side Effects:

**

CSA_bIsAnySensorActive

Description:

Checks the difference count array for all sensors compared to their finger threshold. Calls CSA_bIsSensorActive for each sensor so the CSA_baSnsOnMask array is up to date after calling this function.

C Prototype:

```
BYTE  CSA_bIsAnySensorActive()
```

Assembly:

```
lcall CSA_bIsAnySensorActive
```

Parameters:

None

Return Value:

Return value of 1 if active, 0 if not active

Side Effects:

**

CSA_SetDefaultFingerThresholds

Description:

Loads the CSA_baBtnFThreshold array with the FingerThreshold parameter value. This function must be called before scanning if the CSA_baBtnFThreshold array is not manually loaded with custom values.

C Prototype:

```
BYTE  CSA_SetDefaultFingerThresholds()
```

Assembly:

```
lcall CSA_SetDefaultFingerThresholds
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSA_InitializeBaselines

Description:

Loads the CSA_waSnsBaseline array with initial values by scanning each sensor.

C Prototype:

```
void  CSA_InitializeBaselines()
```

Assembly:

```
lcall CSA_InitializeBaselines
```

Parameters:

None

Return Value:

None

Side Effects:

**

CSA_InitializeSensorBaseline

Description:

Loads the CSA_waSnsBaseline array with an initial value for a particular sensor. Used to reset the baseline for a particular sensor.

C Prototype:

```
void  CSA_InitializeSensorBaseline(BYTE bSensor)
```

Assembly:

```
lcall CSA_InitializeSensorBaseline
```

Parameters:

bSensor: Range is 0 to n-1 where n is the total of the number of sensors set in the CSA Wizard plus the number of sensors included in slider sensors.

Return Value:

None

Side Effects:

**

CSA_wGetCentroidPos

Description:

Checks a difference array for a centroid. If one exists, the offset and length are stored in temporary variables and the centroid position is calculated to the resolution specified in the CSA Wizard.

C Prototype:

```
WORD  CSA_wGetCentroidPos(BYTE bSnsGroup)
```

Assembly:

```
mov  A,  bSnsGroup
lcall CSA_wGetCentroidPos
```

Parameters:

bSnsGroup: Reference to a specific group of sensors used as a slider.

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. The routine should be called only 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.

Note: If noise counts on the slider segments are greater than the noise threshold, this subroutine may generate a false centroid result. The noise threshold should be set carefully (high enough above the noise level) so that noise does not generate a false centroid.

Sample Firmware Source Code

Scanning Buttons and Turning LEDs On and Off

This code is written for the CSA UCC board (CY3280-20x34) and Linear Slider Module board (CY3280-SLM).

```
//------ Sample code for CSA buttons that LEDs On and Off   ----------
//------     pin assignments for Linear Slider Module plugged   ----------
//-------     into CSA UCC board, CY3280-20x43+SLM            ----------

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

void main(void)
{
//initialize LED states
  PRT0DR |= 0b00100010;  //turn-off LED on P0[5],P0[1]
PRT1DR |= 0b00000100;  //turn-off LED on P1[2]
PRT2DR |= 0b10100000;  //turn-off LED on P2[7],P2[5]

//Set port drive modes for LEDs
  PRT0DM0 |= 0b00100010;  //strong on P0[5],P0[1]
  PRT0DM1 &=~0b00100010;

  PRT1DM0 |= 0b10100100;  //strong on P1[2]
  PRT1DM1 &=~0b10100100;

PRT2DM0 |= 0b10100000;  //strong on P2[5],P2[7]
  PRT2DM1 &=~0b10100000;

M8C_EnableGInt;  //enable global interrupts for use with CSA

CSA_Start();     //initialize the CSA User Module
  CSA_SetDefaultFingerThresholds();   //Load finger thresholds
  CSA_InitializeBaselines();  //Set baselines to current count

  while(1)  //infinite loop scanning buttons
  {
    CSA_ScanAllSensors();    //sample all buttons
CSA_UpdateAllBaselines();  //compute baseline, all buttons

// control the LEDs using the sensor states.
// LED ON if active, OFF if not active.
// Check buttons in sequence.
    if (CSA_bIsSensorActive(0))
    {
      PRT1DR &= ~0b00000100;   //turn-on LED on P1[2]
    }
    else
    {
      PRT1DR |= 0b00000100;  //turn-off LED on P1[2]
    }
    if (CSA_bIsSensorActive(1))
    {
```

```
      PRT0DR &= ~0b00100000;  //turn-on LED on P0[5]
    }
    else
    {
      PRT0DR |= 0b00100000;  //turn-off LED on P0[5]
    }
    if (CSA_bIsSensorActive(2))
    {
      PRT0DR &= ~0b00000010;  //turn-on LED on P0[1]
    }
    else
    {
      PRT0DR |= 0b00000010;  //turn-off LED on P0[1]
    }
    if (CSA_bIsSensorActive(3))
    {
      PRT2DR &= ~0b10000000;  //turn-on LED on P2[7]
    }
    else
    {
      PRT2DR |= 0b10000000;  //turn-off LED on P2[7]
    }
    if (CSA_bIsSensorActive(4))
    {
      PRT2DR &= ~0b00100000;  //turn-on LED on P2[5]
    }
    else
    {
      PRT2DR |= 0b00100000;  //turn-off LED on P2[5]
    }
  }
}
```

Controlling LED Intensity Using a Linear Slider

This code is written for the CSA UCC board (CY3280-20x34) and Linear Slider Module board (CY3280-SLM).

```
//------ Sample code for CSA slider controlling LED intensity ----------
//------   pin assignments for Linear Slider Module plugged   ----------
//-------  into CSA UCC board, CY3280-20x43+SLM               ----------

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

int wCentroid = 0;  //estimated finger position; 0xffff for no finger
int wPos = 0;  //estimated finger position
int wLED_PWM; //controls LED intensity

void main(void)
{
//initialize LED states
  PRT0DR |= 0b00100010;  //turn-off LED on P0[5],P0[1]
PRT1DR |= 0b00000100;  //turn-off LED on P1[2]
```

```
PRT2DR |= 0b10100000;  //turn-off LED on P2[7],P2[5]

//Set port drive modes for LEDs
  PRT0DM0 |= 0b00100010;  //strong on P0[5],P0[1]
  PRT0DM1 &=~0b00100010;

  PRT1DM0 |= 0b10100100;  //strong on P1[2]
  PRT1DM1 &=~0b10100100;

PRT2DM0 |= 0b10100000;  //strong on P2[5],P2[7]
  PRT2DM1 &=~0b10100000;

M8C_EnableGInt;  //enable global interrupts for use with CSA

CSA_Start();     //initialize the CSA User Module
  CSA_SetDefaultFingerThresholds();   //Load finger thresholds
  CSA_InitializeBaselines();  //Set baselines to current count

  while(1)  //infinite loop scanning slider
  {
    CSA_ScanAllSensors();    //sample all sensors
CSA_UpdateAllBaselines();  //compute baseline, all sensors

wCentroid = CSA_wGetCentroidPos(1); /estimated position
    if (wCentroid != 0xffff)  //0xffff means finger off slider
    {
      wPos = wCentroid;  //get position, range is 0 to 100
    }

if (wPos > 0)  //if position>0, then pulse all LEDs ON
    {
      PRT1DR &= ~0b00000100;  //turn-on LED on P1[2]
      PRT0DR &= ~0b00100000;  //turn-on LED on P0[5]
     PRT0DR &= ~0b00000010;  //turn-on LED on P0[1]
      PRT2DR &= ~0b10000000;  //turn-on LED on P2[7]
      PRT2DR &= ~0b00100000;  //turn-on LED on P2[5]

      for (wLED_PWM = 0; wLED_PWM < wPos*wPos/100; wLED_PWM++)
      {   //control LED pulse width by position^2
      }  //this control function looks nice

}
    // LED pulse ON is over for this period, turn all off
    PRT1DR |= 0b00000100;  //turn-off LED on P1[2]
    PRT0DR |= 0b00100000;  //turn-off LED on P0[5]
    PRT0DR |= 0b00000010;  //turn-off LED on P0[1]
    PRT2DR |= 0b10000000;  //turn-off LED on P2[7]
    PRT2DR |= 0b00100000;  //turn-off LED on P2[5]


}  //do next scan (while loop)
}
```

Further Reading

The following design guides are recommended after reading the CSD User Module datasheet. These documents are available on the Cypress Semiconductor website at 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

Version History

Version	Originator	Description
1.3	DHA	1. Changed default value for settling time from 20 to 160. 2. Capability to add additional slider. 3. Added Radial Slider functionality to the user module and configuration wizard.
1.40	DHA	Added overflow condition check on iDAC code subtraction. If overflow happens the iDAC code is set to 0.
1.50	DHA	1. Updated minimum value of sensor count if Duplex is enabled. 2. Added pre-compiler directive to prevent build errors when no slider is added.
1.50.b	DHA	Updated AC and DC Electrical Characteristics.
1.50.c	DHA	Updated the Electrical Specifications for the CSA User Module.

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 © 2006-2015 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.