Capacitance Sensing - Calibrating CapSense with the CSR User Module

AN2355

Author: Darrin Vallis
Associated Project: Yes
Associated Part Family: CY8C21x24, CY8C24x94
GET FREE SAMPLES HERE
Software Version: PSoC® Designer™ 4.2 SP3
Associated Application Notes: AN2233a, AN2277, AN2292, AN2318

Application Notes Abstract

This Application Note describes the steps necessary to select, place and calibrate the CSR (Capacitive Switch Relaxation Oscillator) User Module for CapSense applications.

Introduction

Cypress' PSoC is an extremely versatile system-on-chip device. Its capabilities allow engineers to solve problems and develop exciting new products. One of the most unique features available in PSoC is capacitive sensing. PSoC is able to detect the presence of a finger through glass, plastic, acrylic or many other non-metallic materials, using a simple PCB trace as the sensor.

See *Application Note AN2233a Capacitive Switch Scan"*for more on the actual physics and electrical engineering of PSoC capacitive sensing.

Designers are free to implement buttons, sliders or any type of touch-based user interface. This is a significant advantage for consumer electronics. Many of the most popular devices on the market today have a PSoC capacitive sense interface.

This Application Note helps an engineer implement their first capacitive sense design. Even though PSoC Designer has a user module that provides all necessary hardware connections, software APIs, and does a lot of sophisticated background processing, it still has to be configured correctly.

Background Requirements

Designers must be at least familiar with PSoC architecture and the PSoC Designer development environment. Cypress Field Applications Engineers are available for just such information. Go to http://www.cypress.com/, click on "About Cypress" in the sidebar and then "Cypress Locations." Local Sales Offices can put designers in touch with FAEs for questions or even a quick PSoC training.

Building a Project

This section describes how to select devices and how to select, place and configure necessary user modules (UMs).

Step 1: Creating a Project

Create a new project with a CapSense-enabled device. These CapSense-enabled parts include those in the CY8C21x34 and CY8C24x94 families. The example project for this Application Note uses the CY8C21434-24LFXI, a 32-pin QFN with 28 CapSense I/Os.

Step 2: Selecting User Modules

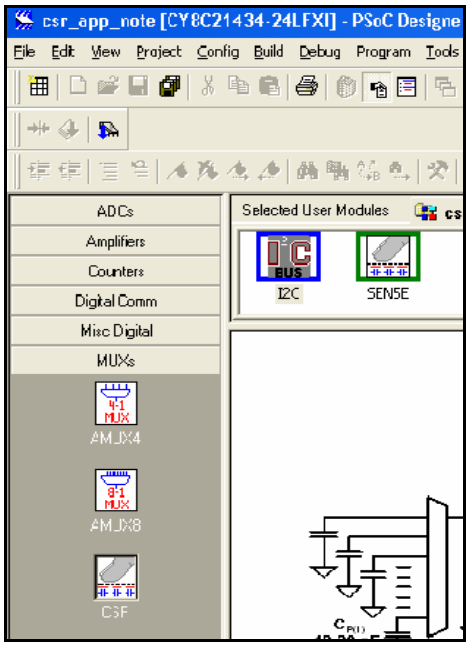
Under the User Module Selection View in PSoC Designer, select the category "MUXs" to see the CSR User Module. Double-click on the icon to place the CSR User Modules into your project. The default name for the UM is "CSR_1." It is possible to rename the UM to whatever is easy to use.

The example project for this Application Note has renamed the CSR UM "SENSE" for ease of coding later on.

It is also good practice to drop in a communication interface. The example project uses I^2C. The EZI2C User Module is easy to configure and work with. This UM has been renamed "I^2C."

User module selections are shown in Figure 1.

Figure 1. User Module Selection



Step 3: Placing User Modules

Before writing a main loop, it is necessary to place the user modules into their respective blocks. The CSR User Module occupies the first 3 digital blocks and the first analog column of the PSoC analog and digital resources. Right-click to place the CSR UM.

Place the CSR UM first. Blocks that can be used by the CSR are fixed.

Step 4: Assigning Pins

Once the UM is placed, right-click on any block to run the CSR Wizard. Enter the number of buttons and sliders in the boxes at the too of the screen. This is shown in Figure 3 (A) and (B). Once the number of sliders has been set, set the number of pins to be used in each slider, as shown in Figure 3 (C). Set the resolution of the slider, Figure 3 (D). After each setting, the wizard will prompt the user for confirmation as shown in Figure 2.

Figure 2. Wizard Setting Confirmation

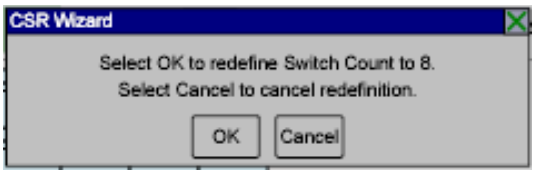
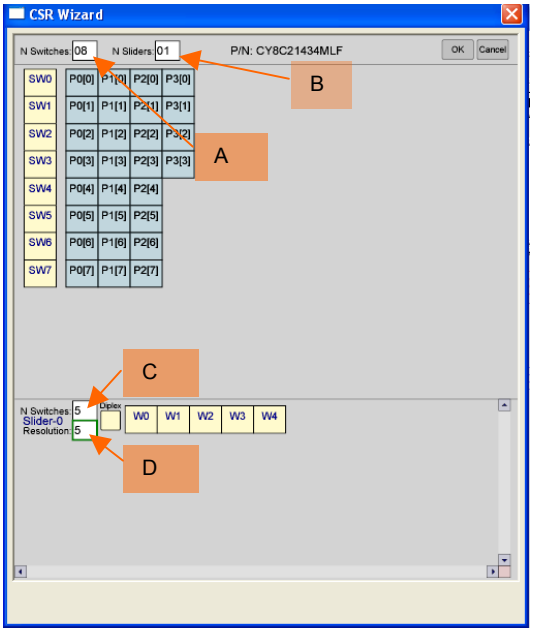
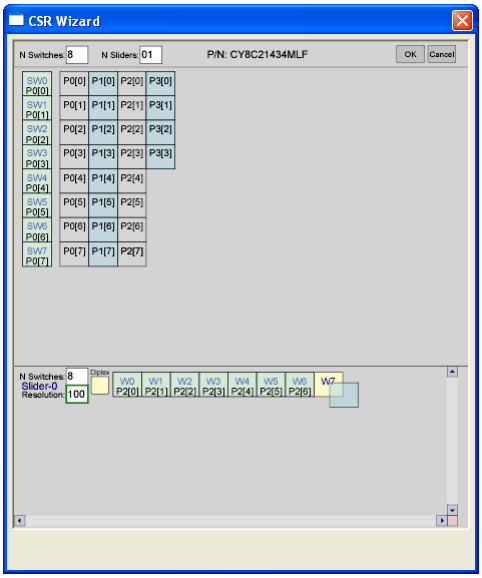


Figure 3. CSR Wizard



Drag and drop the switches you want onto the I/O ports, as shown in Figure 4.

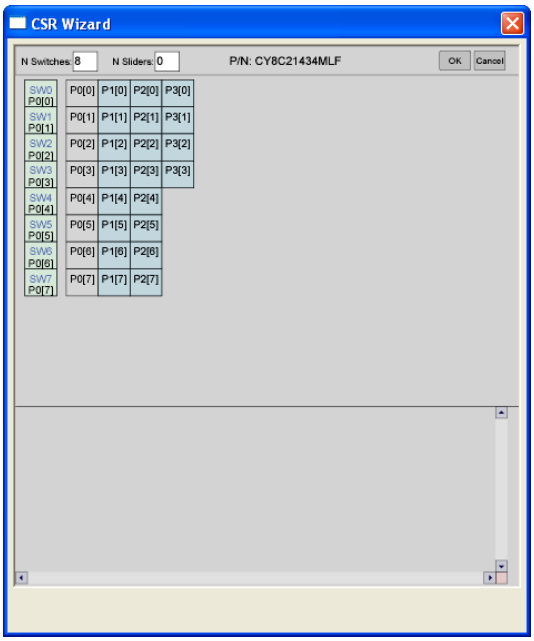
Figure 4. Wizard Operation Screen



Press "**OK**" to exit the Wizard.

In the example project for this Application Note, only 8 buttons are used. These buttons correspond to pins on Port 0. The final Wizard display for the example project is shown in Figure 5.

Figure 5. Example Project Wizard Setup

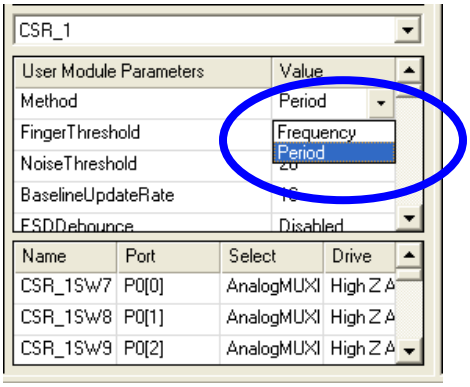


Step 5: Configuring User Modules

It is necessary to assign pins to other user modules if not already complete. For the example project, the I²C communication pins are P1[0] (SDA) and P1[1] (SCL).

The CSR User Module has several options in the User Module Parameters window. These depend on design features that are decided later in the project development. However, it is wise to set the Method to "Period" as shown in Figure 6.

Figure 6. Setting Scanning Method

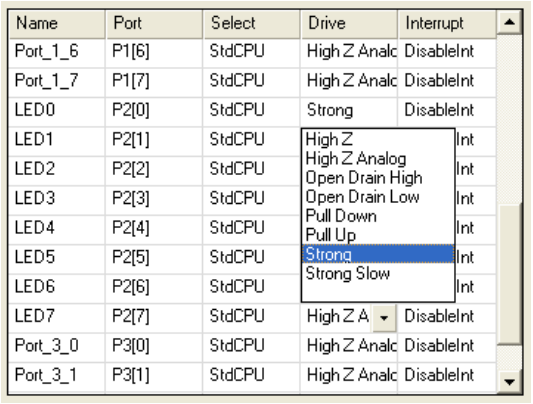


Step 6: Configuring Other Pins

Before building the project, select one pin per switch in the I/O list and name them for LEDs. Set the drive mode to "Strong." This is shown in Figure 7.

The switches are used to indicate when a switch is pressed. Build the project and navigate to the program, `main()`.

Figure 7. Setting LED Drives



Program Structure

It's always excellent coding practice to keep things simple. Example code for the project can be found in Appendix 1. The main loop basically does a one-time configuration of the I/O, I²C and the CSR parameters. After that, it loops forever, scanning buttons, handling an interrupt pin and setting LEDs. That's it.

Tuning the CSR User Module

While all the parameters exist to build and run a project, there are several values that must be calibrated and tuned to specific boards and specific buttons. This is done both in the Device Editor and the Application Editor of PSoC Designer.

Setting Up for Data Receipt

Before tuning can begin, it is necessary to install a method for viewing the capacitive sense data that is gathered by the CSR UM in the application. Though PSoC Designer includes excellent debug capabilities with the ICE and flex pods, a different approach is required when working with capacitive sensing.

The CSR senses finger presence via very small changes in capacitance. A flex pod will have VERY different electrical characteristics than a regular PSoC soldered onto your board.

If it is absolutely necessary to have the CSR working at the same time as a flex pod and ICE, an `#ifdef` statement can be used to re-define CSR parameters when a pod is attached.

For simple projects like the one in this Application Note, I²C is an excellent way to tune the CSR.

Note The author's project required the use of I²C communication. Therefore, I²C was the logical choice for data read. An I²C master from M3 Electronics (http://www.m3electronics.biz/) was used.

The data that must be monitored for tuning is best dealt with within a structure:

Code 1. Code for #ifdef Statement

```
struct I2C_RegType
{
    // Control register
    BYTE   Control;
    // LED On/Off register
    BYTE   LEDs;
    // Button state
    BYTE   Buttons;

#ifdef debug
    // Button raw data
    int counts[NBUTTONS];
    // Baseline data
    int baseline[NBUTTONS];
    // Button differences
    int difference[NBUTTONS];

#endif
}
I2Cregs;
```

The #ifdef is used to include debug data on I^2C for tuning the board. In PSoC Designer, go to PROJECT >> SETTINGS to see an entry box labeled "Macro Defines." Any symbol typed in here is passed to the compiler during build, allowing code to be included or excluded as needed.

As for the data structure:

- "Control" is an I^2C-accessible byte providing control of LEDs, interrupt polarity and other settings. This is optional, but often useful.
- "LEDs" is an I^2C-accessible register allowing the user to set LEDs over I^2C. This may not be necessary for some applications.
- "Buttons" is a byte-wide I^2C register showing the ON/OFF status of each button.

When compiled for debug:

- "Counts" shows the raw counts being read on each switch by the CSR UM.
- "Baseline" is automatically updated by the UM and provides compensation for environmental changes over time.
- "Difference" is very important. It shows the difference in counts between one button scan and the next. The CSR decides if a button has been pressed based on this "difference" data.

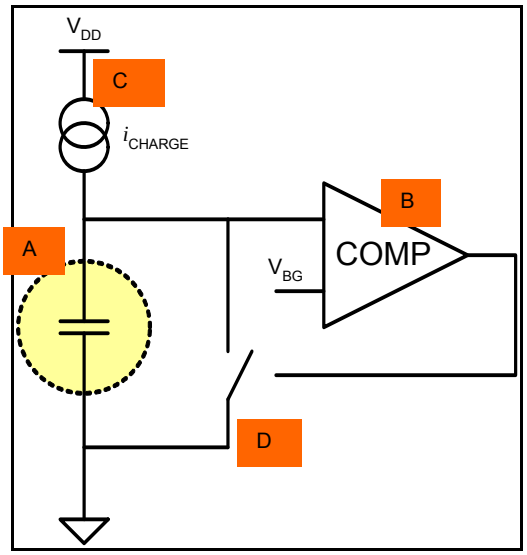
Electrical Description

Now that there are mechanisms to see what's going on inside PSoC during runtime, it is time to tune the CSR UM. First, a basic description of the electrical system is described below.

For a complete description of the electrical circuit for capacitive sensing using the PSoC, please see application note AN2233a.

Figure 8 shows the basic setup of the relaxation oscillator for capacitive sensing. The sensor pad capacitor (A) is charged to a threshold on the comparator (B) with an internal current source (C). When the comparator reaches the threshold, a switch (D) is closed and the sensor capacitor is discharged.

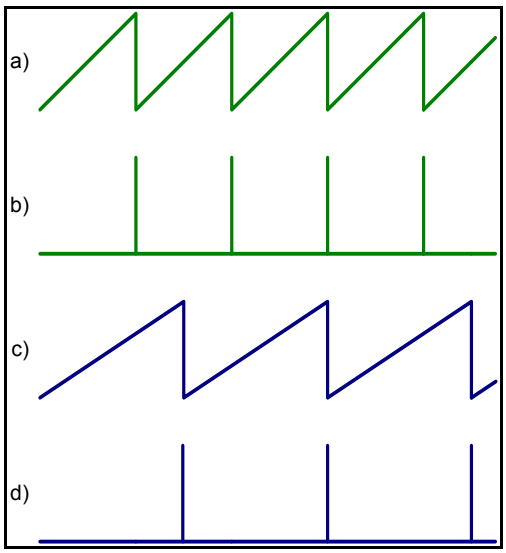
Figure 8. Basic Relaxation Oscillator Circuit



The CY8C21x34 series of PSoC devices has specialized circuitry for automatically charging and discharging an external capacitor. For capacitive sensing, the capacitor is a PCB pad and trace, whose capacitance is altered by the presence of conductive object such as a finger. As the capacitance varies, so does the rate at which the oscillator charges (in the CY8C21x34 devices, the discharge time is fixed at 2 clock cycles).

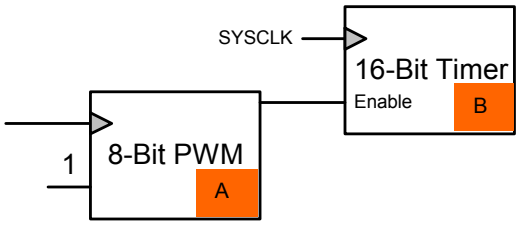
The voltage on the capacitor (sensor) is shown by the waveforms in Figure 9a (no finger) and 9c (finger). The comparator output is shown in Figure 9b (no finger) and 9d (finger).

Figure 9. (a) Sensor Capacitor Voltage, no Finger
(b) Comparator Output, no Finger
(c) Sensor Capacitor Voltage, Finger
(d) Comparator Output, Finger



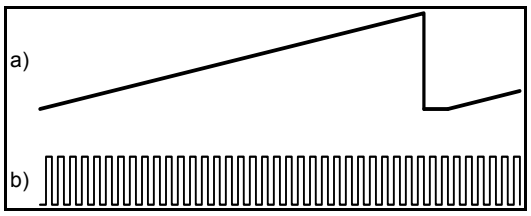
A timer is used to determine the length of time necessary to charge the capacitor to the threshold with the circuit described in Figure 8.

Figure 10. Counting Circuit



The clock for the PWM (A) is the relaxation oscillator's comparator output. The PWM gates a 16-bit timer (B). It is the number of clock cycles that is the basic unit of measurement for the CSR UM. The charge waveform and associate clock cycles are shown in Figure 11.

Figure 11. (a) Sensor Charge Waveform and
(b) Clock Cycles



All of the data that is used and sent by the PSoC is with regard to these clock cycles. PSoC is not measuring capacitance directly, rather the amount of time necessary to charge a capacitor.

Now, for Tuning…

The basic steps for tuning the CSR User Module are:

1. Set DAC current.
2. Set Scan Speed.
3. Adjust Thresholds (Finger and Noise).

Step 1: Setting the DAC Current

Tuning the CSR UM begins with setting frequency of oscillation for each sensor. This oscillation is the charge-discharge cycle on the sensor capacitor. The rate of charge on the capacitor is a factor of the size of the capacitor and the current that is used to charge it to the threshold voltage. The first parameter is fixed in hardware and is what is being measured. It is the second parameter, the DAC current, over which the designer has control.

To start, it is necessary to choose an arbitrary DAC current and set it. This is done using the SetDacCurrent( ) API call. An example setting the current to 10, using low range is shown in Code 2.

Code 2. Code for Setting the DAC Current

```
SENSE_SetDacCurrent(10,SENSE_DAC_LOW);
```

This drives a certain amount of current out of the pin and into the trace, resulting in a specific frequency of oscillation. The current setting is incremented 69 nA in low mode and 2 µA is high mode. In the example, the DAC current is 690 nA. Due to the implementation of CSR hardware within PSoC, the optimal operating frequency for this oscillation is 80 kHz or less.

Determine the current frequency by setting scan speed to three units with the following API call.

Code 3. Code for Determining the Current Frequency

```
SENSE_SetScanSpeed(3);
```

On every switch scan, the user module requires 2 cycles at the end for data processing. Therefore, setting a scan speed of 3 means button counts are gathered for 1 cycle only. Because the counts are sampled by a 24 MHz

system clock the formula to determine oscillator frequency is:

$$F_{OSC} = \frac{24MHz}{\#Counts} \quad \textbf{(1)}$$

ScanSpeed = 3.

Working backwards, reading 300 counts on a sensor at ScanSpeed(3) means F_{osc} = 80 kHz.

In other words, it is necessary to tune the DAC current so that each button reads approximately 300 counts with ScanSpeed = 3.

Table 1. Spreadsheet for Tuning DAC Current

Sensor		0		1		2		3		4		5		6		7	
DAC Current	Setting	2C		2A		24		23		20		1E		22		26	
	Value	3.04		2.90		2.48		2.42		2.21		2.07		2.35		2.62	
Raw Counts		Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
(ScanSpeed = 3)		293	125	294	126	299	12B	294	126	298	12A	298	12A	293	125	293	125
Oscillator Frequency		81.91		81.63		80.27		81.63		80.54		80.54		81.91		81.91	

Table 2. Spreadsheet for Tuning Scan Speed

Sensor	0		1		2		3		4		5		6		7	
DAC Current	2C		2A		24		23		20		1E		22		26	
Scan Speed	30		29		25		14		10		10		12		13	
Raw Counts	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
(Sensor Untouched)	6788	1A84	6688	1A20	6725	1A45	6802	1A92	6944	1B20	6692	1A24	6758	1A66	6673	1A11

This is a good time to introduce a spreadsheet. In the process of tuning, it is necessary to record counts, differences, calculated thresholds, etc. All this work is very well suited to a spreadsheet. Table 1 shows the spreadsheet for the project that is used to record the DAC current settings for each button and their corresponding oscillator frequency.

To achieve a count value of approximately 300 counts (80 kHz) for each sensor, different DAC current settings are required. This is due to variations in the parasitic capacitance that arise from board layout. Using a DAC current low enough so that sensor 5 runs at 80 kHz for all buttons results in other sensors' oscillators running at speeds that are not optimal.

So, what's the solution?

In the Scan_Buttons( ) routine, an array can be used to hold 8 different DAC current values. Instead of scanning all buttons, the code loads DAC current for a button, scans that button once only then loops to the next. This yields individual tuning control for each button, making the board exceptionally responsive and more noise immune.

Step 2: Setting the Scan Speed

Once each sensor is tuned for an 80 kHz oscillator frequency, it's time to set the scan speed. The scan speed (Period Mode) sets the number of oscillator pulses for which the PWM is high. Increasing the scan speed increases two things: the counts read by the PSoC and the time necessary to scan each sensor. This is how to get more data out of the sensor for calibrating sensitivity and accounting for noise.

First, verify the "counts" for each sensor without touching the sensor. This is accomplished in the same manner as counts were obtained while setting the DAC current. I^2C is used in the example project.

Second, gradually increase the scan speed parameter for each sensor. As stated above, the counts increase with scan speed. Setting scan speed to a higher number simply means that instead of collecting counts on a sensor for one scan and processing for two, counts are collected for N scans and processing occurs for two. In other words:

SetScanSpeed(N) = Collect Data for N-2 cycles, process for 2 cycles.

The optimum number of counts for each sensor depends on the project requirements. However, counts should not exceed 0x2000 (8192). This is explained below.

Notice that the scan speed necessary to achieve the desired number of counts is different for each sensor. This is again a result of the parasitic capacitances of each sensor.

Note The raw counts for each sensor cannot exceed 0x3FFF (16,383). This occurs when the scan speed is too high and the PSoC looks at a sensor for too long. The variable that stores the raw counts for each sensor is an integer, meaning that it has a functional limit of 0xFFFF (65,535). The baselining algorithm, described later in this document, further limits this value by one fourth. Values that exceed this limit overflow the integer (roll it past zero).

For example, if the DAC current and scan speed parameters return 0x5100 (20,736) counts, the integer will overflow and instead the CSR UM will detect 0x1100 (4352) counts. This causes problems in a number of routines including the baseline update and ESD debounce. This will lead sensors to exhibit strange behavior.

Step 3: Setting Thresholds

Once the DAC current and scan speed are set, it's time to tune the thresholds for the device. Thresholds are predefined difference counts that correspond to sensor activation status change.

First, some definitions:

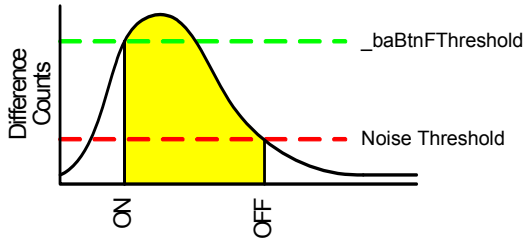
Difference counts are the number of counts by which the current raw counts differ from the baseline.

The *baseline* is a constantly updated zero value for each button. The baseline represents the parasitic capacitance and can be seen when no finger or other such conductive object is present on the sensor. The baseline is used to calculate both the noise and finger thresholds, which are simply added to the baseline.

The *noise threshold* is the number of counts above and below the baseline that are ignored by the PSoC. As long as the difference counts do not exceed the noise threshold, the baseline update function operates. The noise threshold also serves a sensor deactivation threshold, whereby the PSoC determines that a finger is no longer present on the sensor.

The *finger threshold* is the number of counts above the baseline that are necessary to detect the presence of a finger. The counts may fall below this value (due to noise) and the sensor will remain active. It is not until the counts fall below the noise threshold that the sensor state is changed to inactive.

Figure 12. Thresholds and Sensor Activation



Now for more detail…

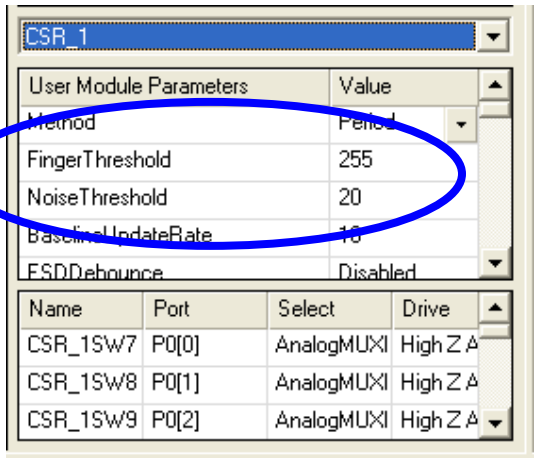
The noise threshold is a parameter set in the CSR User Module Parameter configuration screen of PSoC Designer. Any button scan showing a difference less than the noise threshold is ignored. This parameter makes the design more robust by filtering out low amplitude noise. Setting the noise threshold too high can filter out valid button presses.

Part of the BaselineUpdate algorithm states that if the counts exceed the noise threshold, the baseline is not updated. This ensures that a finger on the sensor does not add inappropriately high count values to the baseline update filter. Setting the noise threshold too low will increase the likelihood that the baseline is not updated and limit PSoC's ability to adjust to environmental changes.

Finger threshold is the decision point for an ON or OFF button. If the difference between two successive scans exceeds this threshold, the button is flagged ON in the variable *(name)_baSwOnMask. You can determine if a valid button press has occurred by polling this variable.

The noise threshold and finger threshold are set for all sensors in the CSR User Module Parameters window in the Device Editor as shown in Figure 13.

Figure 13. Setting Thresholds Globally



Now to calculate the threshold values.

Noise Threshold

The board and code must be set up to output data and scan continuously. Without a finger present on the sensor, the values fluctuate slightly for each sensor. The minimum and maximum values represent the lower and upper noise limits. This is another time at which a spreadsheet is very useful. See Table 3 for complete noise data for the example system.

For sensor[0], the raw counts on the sensor fluctuate between 0x1A84 (6788) and 0x1A56 (6742). This is an overall difference of 46. The noisiest sensor, however, is sensor[3] with a fluctuation of 53. Since sensor[3] is the noisiest sensor and because the noise threshold is a global setting in this project, the noise threshold is set high enough to accommodate the 53 counts of fluctuation.

Table 3. Spreadsheet for Determining Noise Threshold

Sensor		0		1		2		3		4		5		6		7	
		Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
Max Counts		6788	1A84	6688	1A20	6725	1A45	6802	1A92	6944	1B20	6692	1A24	6758	1A66	6673	1A11
Min Counts		6742	1A56	6654	19FE	6673	1A11	6749	1A5D	6896	1AF0	6656	1A00	6719	1A3F	6639	19EF
Noise	Absolute	46	2E	34	22	52	34	53	35	48	30	36	24	39	27	34	22
	Setting	25		19		28		29		26		20		22		19	

Table 4. Spreadsheet for Determining Finger Threshold

Sensor	0		1		2		3		4		5		6		7	
	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
Counts (Untouched)	6788	1A84	6688	1A20	6725	1A45	6802	1A92	6944	1B20	6692	1A24	6758	1A66	6673	1A11
Counts (Touched)	6903	1AF7	6834	1AB2	6881	1AE1	6929	1B11	7074	1BA2	6833	1AB1	6903	1AF7	6815	1A9F
Signal (Difference)	115	73	146	92	156	9C	127	7F	130	82	141	8D	145	91	142	8E
Finger Theshold	81		102		109		89		91		99		102		99	

The noise threshold setting is actually a positive and negative adder to the baseline. Therefore, divide the noise floor by two, and then add two or three to the result to give some room for corner cases that are not seen during the calibration view time. This yields an actual noise threshold setting of 29.

Finger Threshold

Finger threshold is used to denote when sensors become active. The same setup is used to determine the values for this setting. Instead of leaving the sensors untouched, each sensor must be activated and the result of that activation recorded, again, in a table. The results for the example application are shown in Table 4.

Finger threshold can be set globally in the User Module Parameters window (shown in Figure 13). Since each sensor in the example project has a different level of activation, it is necessary to set the finger threshold for each one individually. This is done in the initiation routine using Code 4.

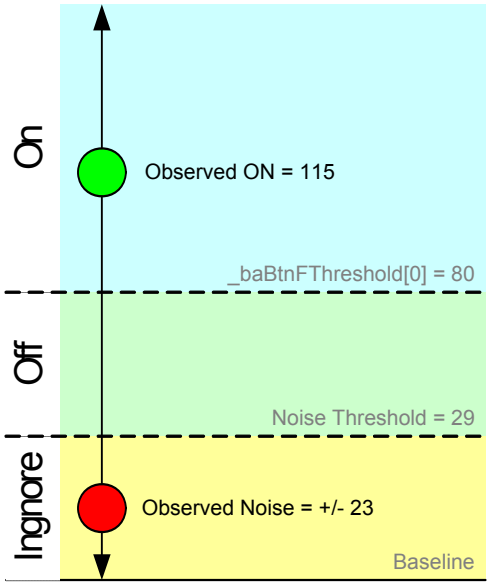
Code 4. Code to Set the Finger Threshold

```
SENSE_baBtnFThreshold[0] = 80;
```

This sets the finger threshold to 80 for one sensor. This can be done for each sensor or in a structure. The result is the same.

Figure 14 shows how these settings apply to sensor [0].

Figure 14. Button[0] Threshold Settings



Anything below the noise threshold is ignored. When the difference between raw counts from one scan to the next exceeds baBtnFThreshold[n], the sensor is “ON.” Once the difference falls below baBtnFThreshold[n], the sensor is “OFF.” Actually pretty simple, but very dependant on setting up these thresholds correctly.

Signal-to-Noise Ratio

While calculating the signal-to-noise ratio is not necessary, it is useful when determining the efficiency of a sensor.

Table 5. Spreadsheet for Calculating Signal-to-Noise Ratio

Sensor	**0**		**1**		**2**		**3**		**4**		**5**		**6**		**7**	
	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
SIGNAL	115	73	146	92	156	9C	127	7F	130	82	141	8D	145	91	142	8E
NOISE	23	17	17	11	26	1A	26.5	1A	24	18	18	12	19.5	13	17	11
SNR = 20LOG(S/N) db	14		19		16		14		15		18		17		18	

Appendix 1

main.c

```
//-----------------------------------------------------------------------------
// Cap Touch code for Application Note
//
// Author:   Darrin Vallis
//-----------------------------------------------------------------------------

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

void main()
{
      Setup_IO();                       //Set up GPIO for address inputs
      Setup_I2C();                      //Configure and start I2C engine
      Setup_Touch();                          //Configure cap touch
      M8C_EnableGInt;                   //Start everything

      while(1)
      {
            Scan_Buttons();             //Switch data in *SENSE_baSwOnMask
            Set_LEDS();                 //Turn on LEDs
            Set_Interrupt_Pin();              //Set or clear int pin
      }
}
```

main.h

```
#define A1                0x40   // A1 = P1_6
#define A0                0x08   // A0 = P1_3
#define I2CBASE           0x40   // I2C address = 0x86/0x86 with no pull-downs
#define LED_AUTO          0x04
#define NBUTTONS          8      // Number of buttons
#define      INT_PIN             0x10          // P1_4
#define INT_POL_MASK      0x02          // Control register, bit 1
#define INT_ENA_MASK      0x01          // Control register, bit 0
#define INT_STAT_MASK     0x80          // Control register, bit 7

struct I2C_RegType
{
      BYTE   Control;                         // Control register
      BYTE   LEDs;                            // LED On/Off register
      BYTE   Buttons;                         // Button state
#ifdef debug
      int          adc[NBUTTONS];                   // Button raw data
      int          baseline[NBUTTONS];        // Baseline data
      int          difference[NBUTTONS];      // Button differences
#endif
} I2Cregs;

BYTE dacCurrent[] = {44,42,36,35,32,30,34,38};
// Individual DAC current for switch 0 - 7
BYTE scanSpeed[]  = {30,29,25,14,10,10,12,13};
// Individual scan speed for switch 0 - 7
BYTE prev_buttons = 0;
// Keeps track of button press status

void Setup_IO(void)
{
      int i;
```

```
	//Make sure the LEDs are OFF
	PRT2DR               = 0xFF;
	I2Cregs.LEDs = 0x00;

	//ReConfigure I2C I/O.  Fixes Start-Up glitch by leaving IO
	//in tri-state on power-up, then switching to I2C mode
	PRT1DM2 |= 0xA0;
	PRT1DM1 |= 0xA0;
	PRT1DM0 |= 0xA0;

	//Set INT to default polarity (Active Low), i.e., no button pressed
	PRT1DR |= INT_PIN;

	//Set default INT values in control reg
	I2Cregs.Control = 0x00;
	I2Cregs.Control &= ~INT_ENA_MASK;//Enable interrupt pin
	                                 //Active low polarity
	                                 //No interrupt pending

	//Set Pull-Ups for A1/A0, P1_6/P1_3
	PRT1DR |= 0x48;
}

void Setup_I2C(void)
{
	unsigned char PortData, I2Caddr, I2Cstatus;

	//Set up I2C address
	//Internal to PSoC, the address is right shifted by 1 bit
	//I2CBASE = b0100 0000
	//If A0 is pulled low on the chip, we set A0, 0000 0001
	//If A1 is pulled low on the chip, we set A1, 0000 0010
	//When no resistors are placed, the address is 0100 0011
	//This translates to b10000110 or 0x87R, 0x86W. Clear ??
	PRT1DR        |= (A1|A0);                                                    //Turn
on internal pull-ups for A1,A0 pins
	PortData  = PRT1DR;                                                          //Read
A1,A0 data from P1_6, P1_3
	I2Caddr = I2CBASE;
	I2Caddr = (PortData & A0) ? (I2Caddr|0x01) : I2Caddr;
	//If A0 bit=1, OR address with 0x01
	I2Caddr = (PortData & A1) ? (I2Caddr|0x02) : I2Caddr;
	//If A1 bit=1, OR address with 0x02
	I2C_Start();                          //Start I2C hardware
	I2C_SetAddr(I2Caddr);                       //Set I2C address from strap pins
	I2C_SetRamBuffer(sizeof(I2Cregs), 6, (BYTE *) &I2Cregs);
	//I2Cregs defined in main.h

	//Init the other registers
	I2Cregs.Control |= LED_AUTO ;
	I2Cregs.Buttons = 0x00;
}

void Setup_Touch(void)
{
	SENSE_Start();

	SENSE_baBtnFThreshold[0] = 80;
	SENSE_baBtnFThreshold[1] = 100;
	SENSE_baBtnFThreshold[2] = 100;
	SENSE_baBtnFThreshold[3] = 90;
	SENSE_baBtnFThreshold[4] = 90;
	SENSE_baBtnFThreshold[5] = 100;
	SENSE_baBtnFThreshold[6] = 100;
```

```
       SENSE_baBtnFThreshold[7] = 100;
}

BYTE Scan_Buttons()
{
   BYTE i,status;

for( i=0; i < NBUTTONS; i++)
{
   SENSE_SetDacCurrent(dacCurrent[i],SENSE_DAC_LOW);
       //Set DAC current for this button
   SENSE_SetScanSpeed(scanSpeed[i]);
       //Set scan speed for this button
   SENSE_StartScan(i,1,SENSE_SCAN_ONCE);                    //Scan the button
   while(!(SENSE_GetScanStatus() & SENSE_SCAN_SET_COMPLETE));
       //Wait until it's done
#ifdef debug
    I2Cregs.adc[i] = SENSE_iReadSwitch(i);
    I2Cregs.baseline[i]  = SENSE_iaSwBaseline[i];
    I2Cregs.difference[i]= SENSE_iaSwDiff[i];
#endif
 }
       status = SENSE_bUpdateBaseline(0);

       if(*SENSE_baSwOnMask!=prev_buttons) prev_buttons = *SENSE_baSwOnMask;

       return(status);
}

void Set_Interrupt_Pin()
{
       if(*SENSE_baSwOnMask!=prev_buttons)
       {
              if( !(I2Cregs.Control&INT_ENA_MASK) )
              {
                     I2Cregs.Control |= INT_STAT_MASK; //Set interrupt in register
                     PRT1DR = (I2Cregs.Control&INT_POL_MASK)? PRT1DR|INT_PIN :
PRT1DR&~INT_PIN;  //And set INT pin
              }
       }

       if (I2C_GetActivity() & I2C_READ_ACTIVITY)           //On I2C read
       {
              I2Cregs.Control &= ~INT_STAT_MASK;            //Clear control register status
bit
              PRT1DR = (I2Cregs.Control&INT_POL_MASK)? PRT1DR&~INT_PIN : PRT1DR|INT_PIN;
//Clear INT pin
       }
}

void Set_LEDS()
{
       PRT2DR = *SENSE_baSwOnMask;
}
```

About the Author

Name: Darrin Vallis
Title: Principal Field Applications Engineer

In March of 2007, Cypress recataloged all of its Application Notes using a new documentation number and revision code. This new documentation number and revision code (001-xxxxx, beginning with rev. **), located in the footer of the document, will be used in all subsequent revisions.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com/